مهندسی نرم افزار

ویراست هفتم

تألیف:
راجر اس. پرسمن

ترجمه:
عین‌الله جعفرنژاد قمی، ابراهیم عامل محرابی

دانش نگار

فصل ۱

نرم‌افزار و مهندسی نرم‌افزار

؟

نگاهی گذرا

نرم‌افزار چیست؟ نرم‌افزار کامپیوتری، محصولی است که مهندس نرم‌افزار طراحی می‌کند و می‌سازد. شامل برنامه‌هایی می‌شود که در کامپیوتری به هر اندازه و با هر معماری، قابل اجرا هستند، مستنداتی دارد که شامل فرم‌های واقعی و مجازی می‌شود و داده‌هایی دارد که ترکیبی از ارقام و حروف است و البته می‌تواند شامل اشکال نمایشی از قبیل اطلاعات تصویری، ویدیویی و صوتی باشد.

چه می‌کند؟ مهندسان نرم‌افزار آن را می‌سازند و در حقیقت هر کسی در دنیای صنعت چه مستقیم و چه غیر مستقیم از آن استفاده می‌کند.

چرا اهمیت دارد؟ چون تقریباً همه‌ی جنبه‌های زندگی ما را تحت‌تأثیر قرار می‌دهد و در تجارت، فرهنگ و فعالیت‌های روزمره ما نمایان است.

چه مراحلی دارد؟ نرم‌افزارهای کامپیوتری نیز همانند تمام محصولات موفق دیگر ساخته می‌شوند، یعنی با اجرای فرایندی چابک و انعطاف‌پذیر، منجر به نتیجه‌ای با کیفیت بالا می‌شود و نیازهای کاربران آن را برآورده می‌سازد. شما روش مهندسی نرم‌افزار را به کار خواهید بست.

محصول کار چیست؟ از دیدگاه مهندس نرم‌افزار، محصول کار، برنامه‌ها، مستندات و داده‌ها هستند که نرم‌افزار کامپیوتری است. ولی از دیدگاه کاربر، محصول کار، اطلاعاتی است که به نحوی به درد کاربر می‌خورند.

چطور مطمئن شوم که درست از عهده کار برآمده‌ام؟ بقیه کتاب را بخوانید، ایده‌های قابل اجرا روی نرم‌افزار خود را انتخاب و آن را در کار خود اجرا نمایید.

به‌نظر می‌رسید مدیر ارشد یک شرکت نرم‌افزاری بزرگ باشد – بیش از چهل سال سن داشت و موهای روی شقیقه‌اش خاکستری شده بود؛ اندامی تراشیده و ورزشکاری داشت با چشمانی که هنگام سخن‌گفتن در شنونده نفوذ می‌کردند. ولی چیزی گفت که مرا شوکه کرد. «نرم‌افزار مرده است.»

با شگفتی خودم را به نادانی زدم و با لبخندی گفتم: «شوخی می‌کنید، نه؟ دنیا را نرم‌افزارها اداره می‌کنند و شرکت شما هم سود خوبی از آن می‌برد. نرم‌افزار هنوز زنده است و رشد می‌کند.»

سرش را با تأکید تکان داد و گفت: «خیر. مرده است... حداقل آن طور که زمانی می‌شناختیم.»

به جلو خزیدم و گفتم: «ادامه بدهید.»

در حالی که برای تأکید بر گفته‌هایش روی میز ضرب گرفته بود، شروع به صحبت کرد: «این دیدگاه مکتب قدیمی به نرم‌افزار – که آن را می‌خری، مالک آن می‌شوی و مدیریت آن وظیفه تو است – به پایان رسیده است. امروز با ظهور وب ۲/۰ و قدرت بالای کامپیوترها شاهد نسل کاملاً متفاوتی از نرم‌افزارها خواهیم بود. نرم‌افزارها از طریق اینترنت تحویل داده خواهند شد و انگار که دقیقاً روی دستگاه هر کدام از کاربران قرار دارند... ولی در واقع روی یک سرور بسیار دور قرارداده شده‌اند.»

ناچار بودم موافقت کنم: «پس زندگی خیلی ساده‌تر می‌شود. آدم‌هایی مثل شما دیگر مجبور نیستند نگران پنج نسخه‌ی متفاوت از یک برنامه‌ی کاربردی باشند که ده‌ها هزار کاربر از آنها استفاده می‌کنند.»

او لبخندی زد: «دقیقاً. فقط آخرین نسخه‌ای که روی سرورهای ما قرار دارد. همین که تغییر یا تصحیحی به عمل آورده شود، قابلیت عملیات بهنگام شده را در اختیار همه‌ی کاربران قرار می‌دهیم و همه بلافاصله آن را در اختیار خواهند داشت.»

شکلکی در آوردم و گفتم: «ولی اگر مرتکب اشتباه هم بشوید، همه این اشتباه را هم بلافاصله خواهند دید.»

او با لبخند گفت: «درست است و به همین خاطر هم تلاش‌های خودمان را دو برابر کرده‌ایم تا مهندسی نرم‌افزار را بهتر کنیم. مشکل اینجاست که باید این کار را «سریع» انجام دهیم، چون بازار در هر حیطه‌ی کاربردی شتاب گرفته است.»

به صندلی تکیه دادم و دست‌هایم را پشت سرم گذاشتم: «می‌دانید، می‌گویند... می‌توانید سریع، درست و ارزان به آن برسید. از اینها دو مورد را انتخاب کنید!»

در حالی که از جای خود بر می‌خاست، گفت: «من سریع و درست را انتخاب می‌کنم.»

من هم از جای خودم بلند شدم و گفتم «پس واقعاً به مهندسی نرم‌افزار احتیاج دارید.»

در حالی که دور می‌شد گفت: «این را می‌دانم ولی مشکل این است که هنوز باید یک نسل دیگر از افراد فنی را قانع کنیم که این حرف درست است!»

آیا نرم‌افزار *واقعاً* مرده است؟ اگر چنین بود که این کتاب را نمی‌خواندید.

نرم‌افزار کامپیوتری همچنان مهمترین فن‌آوری در صحنه جهانی به شمار می‌رود. و در عین حال مثالی از قانون پیامدهای ناخواسته است. پنجاه سال قبل هیچ کس نمی‌توانست پیش‌بینی کند که نرم‌افزارها به یک فن‌آوری ضروری برای تجارت، علوم و مهندسی تبدیل خواهند شد؛ که با کمک نرم‌افزار فن‌آوری‌های جدیدی (مانند مهندسی ژنتیک و فن‌آوری نانو) خلق شود، فن‌آوری‌های موجود (مانند ارتباطات) بسط و توسعه یابند و در فن‌آوری‌های قدیمی‌تر (مانند صنعت چاپ) تغییرات بنیادی پدید آید؛ که نرم‌افزارها نیروی محرکه‌ای برای انقلاب در زمینه کامپیوترهای شخصی شوند؛ که

> «ایده‌ها و تکنیک‌های فن‌آوری نیروی محرکه رشد اقتصادی هستند.»
>
> مجله‌ی اقتصادی وال‌استریت

مشتریان بتوانند از مغازه سرکوچه نرم‌افزارها را در قالب بسته‌بندی‌های کوچک خریداری کنند؛ که نرم‌افزار به آهستگی از یک محصول به سرویسی تکامل پیدا کند که شرکت‌های نرم‌افزاری مطابق با درخواست مشتری در قالب یک عملکرد ویژه از طریق مرورگر وب در اختیار او قرار دهند؛ که یک شرکت نرم‌افزاری از تقریباً هر شرکت صنعتی هم عصر خودش بزرگتر و تأثیر گذارتر شود؛ که یک شبکه گسترده که با نرم‌افزار هدایت می‌شود و به آن اینترنت گفته می‌شود، تکامل یابد و همه چیز را از جستجو در کتابخانه گرفته تا خرید از فروشگاه و مسائل سیاسی تغییر دهد.

هیچ کس قادر به این پیش‌بینی نبود که نرم‌افزارها در هر نوع سیستمی تعبیه خواهند شد: حمل و نقل، پزشکی، ارتباطات، نظامی، صنعتی، سرگرمی، ماشین‌های اداری،... و این فهرست تقریباً پایانی ندارد. و اگر به قانون پیامدهای ناخواسته اعتقاد دارید، اثرات فراوانی وجود دارد که هنوز قابل پیش‌بینی نیست.

هیچ کس پیش‌بینی نمی‌کرد که میلیون‌ها برنامه کامپیوتری را باید اصلاح کرد، تطبیق داد و با گذر زمان بهبود بخشید. زحمت اجرای این فعالیت‌های «نگهداری» از کل کار لازم برای ایجاد نرم‌افزار جدید بیشتر است و به افراد و منابع بیشتری نیاز دارد.

از آنجا که اهمیت نرم‌افزارها افزایش یافته است، جامعه نرم‌افزاری پیوسته در تلاش بوده است تا فن‌آوری‌هایی را توسعه بخشد که آن را ساده تر، سریع‌تر و کم هزینه‌تر ساخته، کیفیت برنامه‌ها را در سطحی بالا حفظ کنند. برخی از این فن‌آوری‌ها با هدف یک دامنه‌ی کاربرد مشخص (مثلاً طراحی و پیاده‌سازی وب‌سایت)؛ عده‌ای دیگر بر دامنه‌ی فن‌آوری دیگری تأکید دارند (مثلاً سیستم‌های شیءگرا یا جنبه‌گرا)؛ و عده‌ای نیز پایه‌ای گسترده دارند (مانند سیستم‌های عامل نظیر Linux). ولی هنوز باید فن‌آوری نرم‌افزاری را توسعه بخشیم که همه‌ی این کارها را انجام دهد و احتمال ظهور یک چنین فن‌آوری در آینده اندک است. با این حال، انسان‌ها کار، راحتی، امنیت، سرگرمی، تصمیم گیری‌ها و زندگی خود را روی نرم‌افزارهای کامپیوتری می‌گذارند. پس بهتر است درست این کار را انجام دهند.

این کتاب چارچوبی ارائه می‌دهد که سازندگان نرم‌افزارهای کامپیوتری از آن بتوانند استفاده کنند – یعنی کسانی که باید این کار را درست انجام دهند. این چارچوب شامل یک فرایند، مجموعه‌ای از روش‌ها و آرایه‌ای از ابزارها می‌شود که آن را *مهندسی نرم‌افزار* می‌نامیم.

۱-۱ ماهیت نرم‌افزار

امروزه نرم‌افزار نقشی دوگانه دارد. نرم‌افزار نوعی محصول است و در عین حال وسیله‌ای نقلیه‌ای برای تحویل یک محصول. به‌عنوان محصول، توان محاسباتی بالقوه‌ی یک سخت‌افزار کامپیوتری یا به طور گسترده‌تر، شبکه‌ای از کامپیوترها را بالفعل می‌کند. نرم‌افزار چه در داخل یک تلفن همراه باشد و چه درون یک کامپیوتر بزرگ عمل کند، یک مبدل اطلاعات است. تولید، مدیریت، اکتساب، اصلاح، نمایش یا انتقال اطلاعاتی که می‌تواند به سادگی یک بیت باشد یا به پیچیدگی یک نمایش چندرسانه‌ای. نرم‌افزار به‌عنوان وسیله‌ی نقلیه‌ای برای تحویل یک محصول، مبنای کنترل کامپیوتر (سیستم‌عامل)، مخابرات اطلاعات (شبکه‌ها) و خلق و کنترل برنامه‌های دیگر (محیط‌ها و ابزارهای نرم‌افزاری) را تشکیل می‌دهد.

> **نکته‌ی کلیدی**
>
> نرم‌افزار هم محصول و هم وسیله نقلیه‌ای است که محصول را تحویل می‌دهد.

نرم‌افزار مهمترین محصول عصر ما را تحویل می‌دهد: *اطلاعات*. نرم‌افزار داده‌های شخصی (مانند تراکنش‌های مالی یک فرد) را تبدیل می‌کند به‌طوری که به چیزهای مفیدتری در یک حیطه‌ی محلی تبدیل شوند؛ اطلاعات تجاری را مدیریت می‌کند تا رقابت را بهبود بخشد؛ دروازه‌ای است به سوی شبکه‌های اطلاع‌رسانی جهانی (مانند اینترنت) و وسیله‌ای است برای به‌دست‌آوردن اطلاعات در تمامی اشکال آن.

نقش نرم‌افزارهای کامپیوتری طی ۵۰ سال گذشته دستخوش تغییرات فراوان شده است. پیشرفت‌های عالی در زمینه کارآیی سخت‌افزار، تغییرات بنیادی در معماری کامپیوتر، افزایش زیاد حافظه و ظرفیت ذخیره‌سازی و انواع دستگاه‌های ورودی و خروجی، همگی در پیچیده‌ترشدن سیستم‌های کامپیوتری سهیم بوده‌اند. پیچیدگی سیستم می‌تواند باعث حصول نتایج درخشان شود ولی برای کسانی که قرار است سیستمی پیچیده را بسازند، مشکلات عظیمی به بار می‌آورد.

امروزه صنعت عظیم نرم‌افزار به عاملی تعیین کننده در اقتصاد جهان صنعتی تبدیل شده است. تیم‌های متخصصان نرم‌افزار، که هر یک روی بخشی از فن‌آوری لازم برای تحویل یک برنامه‌ی کاربردی پیچیده کار می‌کنند، جایگزین برنامه‌نویسان تنهای گذشته شده‌اند. و هنوز پرسش‌هایی که از آن برنامه‌نویسان تنها پرسیده می‌شدند، همان‌هایی هستند که هنگام ساخته شدن سیستم‌های کامپیوتری مدرن پرسیده می‌شوند[1].

- چرا به پایان رساندن یک نرم‌افزار این قدر وقت می‌گیرد؟
- چرا ساخت نرم‌افزار هزینه‌ی بالایی دارد؟
- چرا نمی‌توانیم همه‌ی خطاها را پیش از تحویل نرم‌افزار به مشتری بیابیم؟
- چرا برای نگهداری یک نرم‌افزار موجود، این قدر وقت و هزینه صرف می‌کنیم؟
- چرا در اندازه‌گیری پیشرفت ساخت و نگهداری نرم‌افزار، دچار مشکل می‌شویم؟

این پرسش‌ها و بسیاری پرسش‌های دیگر، بیانیه‌ای هستند درباره دغدغه‌های مربوط به نرم‌افزار و شیوه توسعه‌ی آن – دغدغه‌هایی که نتیجه‌اش کار مهندسی نرم‌افزار بوده است.

«نرم‌افزار جایی است که در آن رویا کاشته می‌شود و کابوس درو می‌شود، یک برگه اسرارآمیز و انتزاعی که در آن ارواح پلید با اکسیرهای جادویی در رقابت‌اند، جهانی از انسان‌های گرگ‌نما و گلوله‌های نقره‌ای...»

براد ج. کاکس

نرم‌افزار را چگونه باید تعریف کنیم؟

۱-۱-۱ تعریف نرم‌افزار

امروزه اکثر حرفه‌ای‌ها و بسیاری از افراد عامه، سخت معتقدند که می‌دانند «نرم‌افزار» چیست. ولی آیا واقعاً می‌دانند؟

توصیفی درسی از نرم‌افزار می تواند به شکل زیر باشد:

نرم‌افزار عبارت است از: (۱) دستورالعمل‌ها (برنامه‌های کامپیوتری) که هنگام اجرا، ویژگی، عملکرد و کارایی مطلوب را فراهم می‌سازند؛ (۲) ساختمان‌های داده‌هایی که برنامه‌ها را قادر به پردازش مناسب داده‌ها کنند و (۳) اطلاعات توصیفی در هر دو قالب کپی سخت و مجازی که راه‌اندازی و استفاده از برنامه‌ها را شرح دهند.

تردیدی نیست که تعاریف دیگری از نرم‌افزار نیز قابل ارائه است.

[1] تام دومارکو [DeM95] در یک کتاب عالی در باب تجارت نرم‌افزار خلاف این مدعا را بحث می‌کند و می‌گوید: «به‌جای اینکه بپرسیم چرا نرم‌افزارها این همه هزینه‌بردار هستند دیگر باید بپرسیم: «چه کرده‌ایم تا نرم‌افزارهای امروز هزینه بسیار کمی داشته باشند؟» پاسخ این پرسش به ما کمک خواهد کرد تا سطح موفقیت فوق‌العاده‌ای را که همواره صنعت نرم‌افزار را برجسته ساخته است، حفظ کنیم.»

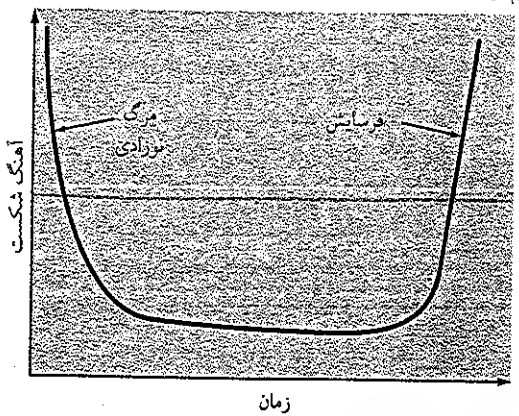
ولی یک تعریف رسمی‌تر، احتمالاً درک شما را به میزان محسوس‌تری افزایش نخواهد داد. برای دستیابی به این منظور، بررسی خصوصیات نرم‌افزار که آن را با سایر ساخته‌های دست بشر متفاوت می‌سازد، حائز اهمیت است. نرم‌افزار، بیشتر یک عنصر منطقی است تا یک عنصر سیستمی فیزیکی. بنابراین، نرم‌افزار دارای خصوصیاتی است که تفاوت چشمگیری با ساخت افزار دارد.

۱. *نرم‌افزار، مهندسی و بسط داده می‌شود و چیزی نیست که به معنای کلاسیک کلمه، ساخته شود.*

گرچه شباهت‌هایی میان بسط نرم‌افزار و ساخت سخت‌افزار وجود دارد، این دو عمل، تفاوت بنیادی دارند. در هر دو عمل، کیفیت بالا از طریق طراحی خوب به دست می‌آید، ولی فاز ساخت برای سخت‌افزار باعث بروز مشکلات کیفیتی می‌شود که برای نرم‌افزار وجود ندارند (یا به‌راحتی قابل رفع هستند). هر دو عمل وابسته به انسان هستند، ولی رابطه‌ی میان انسان و کاری که انجام می‌شود، کاملاً متفاوت است (فصل ۲۴). هر دو عمل مستلزم ساخت یک «محصول» هستند ولی روش‌ها متفاوت است. هزینه‌های نرم‌افزار در مهندسی آن متمرکز است. این بدان معناست که پروژه‌های نرم‌افزاری را نمی‌توان همانند پروژه‌های تولید معمولی مدیریت کرد.

نکته‌ی کلیدی

نرم‌افزارها ساخته نمی‌شوند، بلکه مهندسی می‌شوند.



شکل ۱-۱ منحنی شکست سخت‌افزار.

۲. *نرم‌افزار «فرسوده نمی‌شود».*

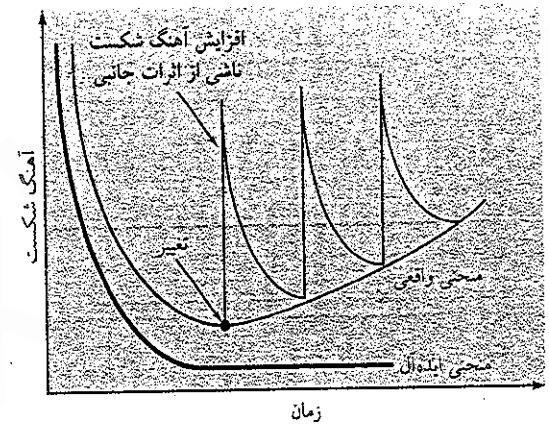
شکل ۱-۱ نمودار آهنگ شکست را به‌صورت تابعی از زمان برای سخت‌افزار نشان می‌دهد. این رابطه که غالباً «منحنی وانی» نامیده می‌شود، نشان می‌دهد که سخت‌افزار، آهنگ شکست نسبتاً شدیدی در ابتدای عمر خود نشان می‌دهد (این شکست‌ها را غالباً می‌توان به عیوب طراحی و تولید نسبت داد)؛ این عیوب تصحیح می‌شوند و آهنگ شکست برای یک دوره زمانی به مقداری ثابت نزول می‌کند (که امید می‌رود، بسیار پایین باشد). با گذشت زمان، سخت‌افزار شروع به فرسایش کرده دوباره آهنگ شکست شدت می‌گیرد.

نرم‌افزار نسبت به ناملایمات محیطی که باعث فرسایش آن می‌شود، نفوذپذیر نیست. بنابراین، در تئوری، منحنی شکست برای نرم‌افزار باید شکل منحنی ایده‌آل شکل ۲-۱ را به خود بگیرد. عیوب کشف نشده باعث آهنگ شکست شدید، در ابتدای عمر برنامه می‌شود. ولی، این عیوب برطرف می‌شوند (با این امید که خطاهای دیگر وارد نشود) و منحنی به‌صورتی که نشان

اندرز

اگر می‌خواهید فرسایش نرم‌افزار را کاهش دهید، باید طراحی بهتری انجام دهید (فصل‌های ۸ تا ۱۳).

داده شده است، هموار می‌شود. منحنی ایده‌آل نسبت به منحنی واقعی مدل‌های شکست نرم‌افزار، بسیار ساده‌تر است. ولی، معنای آن واضح است، نرم‌افزار هرگز دچار فرسایش نمی‌شود بلکه زوال می‌یابد!



شکل ۲-۱ منحنی‌های شکست واقعی و ایده‌آل برای نرم‌افزار.

این تناقض ظاهری را می‌توان با در نظر گرفتن «منحنی واقعی» به بهترین وجه توضیح داد (شکل ۲-۱). نرم‌افزار در دوران حیات خود[1] دستخوش تغییر می‌شود (نگهداری). با اعمال این تغییرات، احتمال دارد که برخی عیوب جدید وارد شوند و باعث خیز منحنی آهنگ شکست شوند (شکل ۲-۱). پیش از آنکه منحنی بتواند به آهنگ شکست منظم اولیه خود برسد، تغییر دیگری درخواست می‌شود که باعث خیز دوباره‌ی منحنی می‌شود. حداقل میزان شکست به آهستگی افزایش می‌یابد – نرم‌افزار در اثر تغییر فاسد می‌شود.

نکته‌ی کلیدی
در روش‌های مهندسی نرم‌افزار تلاش می‌شود که از بزرگی و بیرابزرگی شیب منحنی واقعی در شکل ۲-۱ کاسته شود.

یک جنبه‌ی دیگر از فرسایش نیز اختلاف میان سخت‌افزار و نرم‌افزار را نشان می‌دهد. هنگامی که یک مؤلفه از سخت‌افزار فرسوده می‌شود، با یک مؤلفه‌ی یدکی تعویض می‌شود. ولی نرم‌افزار قطعات یدکی ندارد. هر شکست نرم‌افزاری نشانگر خطایی در طراحی یا فرایندی است که طراحی از طریق آن به کدهای قابل اجرا روی ماشین تبدیل می‌شود. از این رو، نگهداری نرم‌افزار به مراتب پیچیده‌تر از نگهداری سخت‌افزار است.

۳. *گرچه صنعت در حال حرکت به سوی مونتاژ قطعات است، اکثر نرم‌افزارها همچنان به‌صورت سفارشی ساخته می‌شوند.*

به موازات تکامل یک رشته‌ی مهندسی، مجموعه‌ای از قطعات طراحی استاندارد ایجاد می‌شود. پیچ‌های استاندارد و مدارات مجتمع آماده،فقط دو مورد از هزاران مؤلفه‌ی استانداردی هستند که مهندسان مکانیک و برق در طراحی سیستم‌های جدید به کار می‌برند. قطعات قابل استفاده‌ی مجدد طوری طراحی شده‌اند که مهندس بتواند بر عناصر واقعاً جدید یک طراحی، یعنی قطعاتی از طراحی که ارائه‌دهنده‌ی چیزی تازه هستند، تمرکز داشته باشد. در جهان سخت‌افزار، استفاده‌ی

> «ایده‌ها قطعات سازنده ایده‌ها هستند.»
> جیسون زباسی

[1] درواقع، از همان لحظه‌ای که توسعه‌ی نرم‌افزار آغاز می‌گردد و مدت‌ها قبل از تحویل اولین نسخه، تغییرات ممکن است توسط طرف‌های ذینفع گوناگون درخواست گردد.

مجدد از قطعات، بخشی طبیعی از فرایند مهندسی است. در مهندسی نرم‌افزار این امر به تازگی مورد توجه قرار گرفته است.

یک مؤلفه‌ی نرم‌افزاری باید چنان طراحی و پیاده‌سازی شود که بتوان در برنامه‌های متفاوت از آن بهره برد. در دهه ۱۹۶۰، کتابخانه‌هایی از زیرروال‌های علمی ساختیم که در آرایه‌ی گسترده‌ای از کاربردهای مهندسی و علمی قابل استفاده بودند. این کتابخانه‌ها از الگوریتم‌هایی معین به شیوه‌ای کارامد استفاده می‌کردند، ولی دامنه‌ی کاربرد محدودی داشتند. امروزه، ایده‌ی استفاده‌ی مجدد نه تنها الگوریتم‌ها، بلکه ساختمان داده‌ها را نیز در بر می‌گیرد. قطعات مدرن قابل استفاده‌ی مجدد، هم داده‌ها و هم پردازشی را که در مورد آنها اعمال می‌گردد، پنهان‌سازی کرده مهندس نرم‌افزار را قادر می‌سازد تا از قطعات قابل استفاده‌ی مجدد، برنامه‌های کاربردی جدید بسازد[1]. برای مثال، واسط‌های کاربر گرافیکی امروزی با استفاده از قطعات قابل استفاده‌ی مجدد ساخته می‌شوند که ایجاد پنجره‌های گرافیکی، منوهای بازشونده و انواع راهکارهای محاوره را میسر می‌سازند.

۲-۱-۱ دامنه‌های کاربرد نرم‌افزار

امروزه هفت گروه وسیع از نرم‌افزارهای کامپیوتری، پیوسته باعث چالش برای مهندسان نرم‌افزار می‌شوند:

نرم‌افزارهای سیستمی. نرم‌افزار سیستمی مجموعه‌ای از برنامه‌هاست که برای سرویس‌دهی به برنامه‌های دیگر نوشته شده‌اند. برخی نرم‌افزارهای سیستمی (مثل کامپایلرها، ویراستارها و برنامه‌های کمکی مدیریت فایل) ساختارهای اطلاعاتی پیچیده ولی قطعی[2] دارند. برخی برنامه‌های سیستمی دیگر (نظیر قطعات سیستم‌عامل، راه‌اندازها، پردازنده‌های ارتباط راه دور) مقادیر زیادی از داده‌های مبانی را پردازش می‌کنند. در هر حال، مشخصه‌های حیطه‌ی نرم‌افزارهای سیستمی عبارتنداز: تعامل سنگین با سخت‌افزار کامپیوتر؛ استفاده سنگین توسط چند کاربر؛ عمل کنونی که مستلزم زمان‌بندی است؛ مدیریت فرایند پیچیده و اشتراک منابع؛ ساختمان داده‌های پیچیده و واسط‌های خارجی چندگانه.

نرم‌افزارهای کاربردی – برنامه‌های مستقلی که یک نیاز تجاری مشخص را بر طرف می‌سازند. برنامه‌های کاربردی در این حوزه، داده‌های تجاری و فنی را به شیوه‌ای پردازش می‌کنند که عملیات تجاری یا تصمیم‌گیری‌های مدیریتی/فنی را سهولت بخشند. علاوه بر برنامه‌های کاربردی مرسوم داده‌پردازی از نرم‌افزارهای کاربردی برای کنترل عملیات تجاری در زمان حقیقی (بی‌درنگ) (مانند پردازش تراکنش‌های نقطه فروش و کنترل فرایندهای تولیدی بی‌درنگ) استفاده می‌شود.

نرم‌افزارهای مهندسی/علمی. نرم‌افزارهای علمی توسط الگوریتم‌هایی مشخص می‌شوند که «ارقام و اعداد» را پردازش می‌کنند. کاربردهای آن از نجوم تا بررسی آتش‌فشان‌ها، از تحلیل فشار خودکار تا دینامیک مدار شاتل‌های فضایی و از زیست‌شناسی مولکولی تا مکانیزاسیون صنعتی را

مرجع وب
از جمله جامع‌ترین کلکسیون‌های موجود نرم‌افزارهای رایگان و اشتراکی می‌توان به سایت زیر اشاره کرد:
shareware.cnet.com

[1] توسعه‌ی مبتنی بر مؤلفه‌ها موضوع بحث فصل ۱۰ است.

[2] نرم افزار درصورتی *قطعی* است که ترتیب و زمان‌بندی ورودی‌ها، پردازش و خروجی‌ها قابل پیش‌بینی باشد. نرم افزار درصورتی *غیرقطعی* است که ترتیب و زمان‌بندی ورودی‌ها، پردازش و خروجی‌ها را نتوان از قبل در آن پیش‌بینی کرد.

در بر می‌گیرد. ولی، کاربردهای نوین در حیطه‌ی مهندسی و علمی از الگوریتم‌های عددی مرسوم فراتر رفته‌اند. طراحی به کمک کامپیوتر، شبیه‌سازی سیستم‌ها، و برنامه‌های کاربردی محاوره‌ای دیگر، رفته رفته خصوصیات نرم‌افزارهای بی‌درنگ و نرم‌افزارهای سیستمی را به خود می‌گیرند.

نرم‌افزارهای تعبیه شده. در حافظه فقط خواندنی جای دارند و برای کنترل محصولات و سیستم‌های مربوط به بازارهای صنعتی و مصرفی به کار می‌رود. نرم‌افزار تعبیه شده قادر به انجام اعمالی بسیار محدود و اختصاصی (از قبیل کنترل صفحه‌کلید برای فرهای مایکروویو) بوده یا وظایف مهم و قابلیت کنترل (مانند عملیات دیجیتال در خودروها از قبیل کنترل سوخت، صفحه نمایش داشبورد، سیستم ترمز و غیره) را بر عهده دارد.

نرم‌افزارهای خط تولید- برای فراهم آوردن یک قابلیت خاص، جهت استفاده توسط بسیاری از مشتریان مختلف طراحی می‌شوند. نرم‌افزارهای خط تولید ممکن است یک بازار محدود و خاص (مانند محصولات کنترل موجودی) را هدف قرار دهند یا بازارهای پرمشتری (مانند برنامه‌های کاربردی واژه‌پرداز، صفحه‌گسترده، گرافیک کامپیوتری، خبر رسانه‌ای، سرگرمی، مدیریت بانک‌های اطلاعاتی و اموال تجاری) را پوشش دهند.

برنامه‌های کاربردی تحت وب - این گروه از نرم‌افزارهای شبکه‌ای شامل مجموعه‌ی گسترده‌ای از برنامه‌های کاربردی می‌شود. برنامه‌های تحت وب می‌توانند قدری بیشتر از یک مجموعه فایل‌های اَبَرمتن[1] باشند که اطلاعات را با استفاده از متن و گرافیک محدود ارائه می‌دهند. ولی با ظهور وب ۲/۰، برنامه‌های تحت وب در حال تکامل به محیط‌های کامپیوتری پیچیده‌ای هستند که نه تنها ویژگی‌های مستقل فراهم می‌آورند بلکه شامل بانک‌های اطلاعاتی و برنامه‌های کاربردی تجاری نیز می‌شوند.

نرم‌افزارهای هوش‌مصنوعی. نرم‌افزارهای هوش‌مصنوعی (AI) برای حل مسائل پیچیده‌ای که به روش‌های عددی قابل حل نیستند، از الگوریتم‌های غیرعددی استفاده می‌کنند. *سیستم‌های خبره*، که سیستم‌های مبتنی بر آگاهی نیز نامیده می‌شوند؛ تشخیص الگوها (تصویری و صوتی)؛ شبکه‌های عصبی مصنوعی؛ اثبات قضایا و بازی، همگی مثال‌هایی از کاربرد این گروه هستند.

> هیچ کامپیوتری نیست که عمل سلیم داشته باشد.
> **ماروین مینسکی**

میلیون‌ها مهندس نرم‌افزار در سرتاسر جهان روی یک یا چند مورد از این گروه‌ها سخت مشغول کارند. در برخی موارد، سیستم‌های جدیدی در حال ساخته شدن هستند ولی در بسیاری موارد دیگر، برنامه‌های کاربردی موجود در حال تصحیح، انطباق و بهسازی هستند. به‌وفور پیش می‌آید که یک مهندس نرم‌افزار جوان روی برنامه‌ای کار کند که سن آن از خود او بیشتر باشد! نسل‌های گذشته در هر کدام از گروه‌های ذکرشده در بالا میراثی به‌جا گذاشته‌اند. امید می‌رود که این میراث توسط نسل جدید مهندسان پشت سر نهاده شود و زحمت مهندسان آینده کم شود. و به علاوه، چالش‌های جدیدی (فصل ۳۱) نیز پیش روی داریم.

کار با کامپیوتر در جهانی باز - رشد سریع شبکه‌های بی‌سیم ممکن است به زودی به محیطی واقعاً فراگیر و توزیع شده برای کار با کامپیوتر منجر شود. چالشی که مهندسان فراروی خود خواهند داشت، توسعه‌ی سیستم‌ها و برنامه‌های کاربردی است‌که برقراری ارتباط میان کامپیوترهای شخصی، دستگاه‌های همراه و سیستم‌های اداری را از طریق شبکه‌های گسترده میسر سازند.

[1] hypertext

تامین منابع از طریق شبکه - شبکه جهانی وب به سرعت در حال تبدیل به یک موتور کامپیوتری و نیز منبعی برای ارائه اطلاعات است. چالش برای مهندسان نرم‌افزار، معماری برنامه‌های کاربردی ساده (مانند برنامه‌های مالی شخصی) و پیچیده‌ای است که بازارهای کاربر نهایی هدف را در سرتاسر جهان متقع سازند.

کد منبع باز[1] - تمایل رو به رشدی است که منجر به توزیع کدهای منبع سیستم‌ها و برنامه‌های کاربردی (مانند سیستم‌های عامل، بانک‌های اطلاعاتی و محیط‌های برنامه‌سازی) شده است به‌طوری که افراد بسیاری بتوانند در توسعه‌ی آن سهیم شوند. چالش پیش روی مهندسان نرم‌افزار، ساختن کد منبعی است که خودگویا باشد و از آن مهمتر، توسعه‌ی تکنیک‌هایی که هم مشتری و هم توسعه‌دهنده را قادر سازد تا بدانند چه تغییراتی به عمل آمده است و این تغییرات در نرم‌افزار چگونه نمود می‌یابند.

هر کدام از این تغییرات جدید بدون تردید از *قانون پیامدهای ناخواسته*[2] پیروی می‌کنند و دارای اثراتی هستند (برای دست اندرکاران امور تجاری، مهندسان نرم‌افزار و کاربران نهایی) که امروز قابل پیش‌بینی نیستند. ولی مهندسان نرم‌افزار با ارائه‌ی فرایندی که به قدر کافی هوشمند و قابل انطباق باشد تا بتواند تغییرات فن‌آوری را سامان دهد و قوانین تجاری‌ای را که مطمئن هستند طی دهه بعد ظهور خواهند کرد، پوشش دهد، می‌توانند خود را آماده کنند.

> «همیشه نمی‌توان پیش‌بینی کرد ولی همیشه می‌توان آماده بود.»
> **گمنام**

۳-۱-۱ نرم‌افزارهای قدیمی

صدها هزار برنامه کامپیوتری در یکی از هفت دامنه‌ی وسیعی قرار می‌گیرند که در بخش قبل ذکر شدند. برخی از آنها نرم‌افزارهایی با فن‌آوری پیشرفته‌اند- که تنها برای افراد خاص، صنایع یا دولت ارائه می‌شوند. ولی سایر برنامه‌ها قدیمی‌تر و در برخی موارد، بسیار قدیمی‌ترند.

این برنامه‌های قدیمی‌تر- که غالباً از آنها به‌عنوان *نرم‌افزارهای قدیمی* یاد می‌شود- از دهه ۱۹۶۰ کانون توجه بودند. دیانی- فارد و همکاران [Day99] نرم‌افزارهای قدیمی را چنین توصیف می‌کنند:

> سیستم‌های نرم‌افزاری قدیمی... چند دهه قبل ساخته شده‌اند و پیوسته اصلاح شده‌اند تا تغییرات به عمل آمده در خواسته‌های تجاری و سکوهای محاسباتی را پاسخ‌گو باشند. ازدیاد این گونه سیستم‌ها باعث دردسر برای سازمان‌های بزرگی می‌شود که نگهداری از آنها را پر هزینه و تکامل بخشیدن به آنها را خطرناک می‌دانند.

لیو و همکاران [Liu98] این توصیف را با ذکر این نکته بسط می‌دهند که «بسیاری از سیستم‌های قدیمی همچنان عملیات اصلی و مرکزی تجاری را پشتیبانی می‌کنند و نمی‌توان از آنها چشم‌پوشی کرد.» از این رو، مشخصه‌های نرم‌افزارهای قدیمی عمر طولانی و اهمیت تجاری آنهاست.

متأسفانه، گاهی یک مشخصه اضافی وجود دارد که در نرم‌افزارهای قدیمی به چشم می‌خورد- کیفیت ضعیف[3]. سیستم‌های قدیمی گاهی دارای طراحی غیرقابل گسترش، کدهای پیچیده، مستندسازی ضعیف یا بدون مستندسازی، موارد و نتایج آزمونی که هرگز بایگانی نشده‌اند و یک

> اگر با یک سیستم قدیمی مواجه شوم که کیفیت ضعیفی دارد، چه باید بکنم؟

[1] open source
[2] law of unintended consequences
[3] در این مورد، کیفیت بر اساس تفکر جدید مهندسی نرم‌افزار قضاوت می‌شود- یک ملاک نسبتاً ناعادلانه چرا که برخی اصول و مفاهیم مهندسی نرم‌افزار نوین در زمان ساخته‌شدن نرم‌افزارهای قدیمی هنوز به خوبی شناخته نشده بودند.

تاریخچه تغییر با مدیریت ضعیف هستند که البته این فهرست را می‌توان باز هم ادامه داد و با تمام این تفاصیل، این سیستم‌ها «عملیات اصلی و مرکزی تجاری را پشتیبانی می‌کنند و نمی‌توان از آنها چشم‌پوشی کرد.» حال، تکلیف چیست؟

تنها پاسخ منطقی ممکن است این باشد که «کاری نکنیم.» دست کم تا زمانی که سیستم قدیمی باید دستخوش یک تغییر چشمگیر شود. اگر نرم‌افزار قدیمی نیازهای کاربران خودش را برآورده می‌سازد و با اطمینان کار می‌کند، خراب نیست و نیازی هم به ترمیم ندارد. ولی با گذشت زمان، سیستم‌های قدیمی غالباً به یک یا چند دلیل از دلایل زیر تکامل می‌یابند:

- نرم‌افزار باید برای برآورده‌ساختن نیازهای محیط‌های جدید کامپیوتری یا فن‌آوری‌های جدید اصلاح گردد.
- نرم‌افزار باید بهبود یابد تا خواسته‌های تجاری جدید را پیاده‌سازی کند.
- نرم‌افزار باید گسترش داده شود تا با سایر سیستم‌ها یا بانک اطلاعاتی جدیدتر قابلیت همکاری داشته باشد.
- نرم‌افزار باید دوباره معماری شود تا در یک محیط شبکه نیز قادر به ادامه‌ی حیات باشد.

چه نوع تغییراتی در سیستم‌های قدیمی به‌عمل آمده است؟

هنگامی که این شیوه‌های تکامل رخ دهد، یک سیستم قدیمی را باید دوباره مهندسی نمود (فصل ۲۹) به‌طوری که در آینده بتواند به حیات خود ادامه دهد. هدف مهندسی نرم‌افزار جدید، «ابداع روش‌شناسی‌هایی است که براساس مفهوم تکامل بنا نهاده می‌شوند.» یعنی این مفهوم که سیستم‌های نرم‌افزاری پیوسته در حال تغییرند، سیستم‌های نرم‌افزاری جدید از سیستم‌های قدیمی ساخته می‌شوند و... همه باید قادر به همکاری باشند.» [Day99]

۲-۱ ماهیت منحصر به فرد برنامه‌های کاربردی تحت وب

در روزهای اولیه شبکه جهانی وب (حدود ۱۹۹۰ تا ۱۹۹۵)، وب‌سایت‌ها شامل یک مجموعه فایل‌های آبرمتن پیونددار می‌شد که اطلاعات را با استفاده از متن و گرافیک محدود ارائه می‌دادند. با گذشت زمان، تکمیل HTML با ابزارهای برنامه‌نویسی (مانند XML و جاوا) به مهندسان وب این امکان را داد تا قابلیت‌های کامپیوتری را همراه با محتوای اطلاعاتی فراهم سازند. سیستم‌ها و برنامه‌های کاربردی مبتنی بر وب[1] (که آنها را در کل برنامه‌های تحت وب می‌نامیم) پا به عرصه وجود نهادند. امروزه برنامه‌های تحت وب به ابزارهای کامپیوتری پیچیده‌ای تکامل یافته‌اند که نه تنها عملکردی مستقل را در اختیار کاربر نهایی قرار می‌دهند، بلکه با بانک‌های اطلاعاتی و برنامه‌های کاربردی تجاری یکی شده‌اند.

اندرز
هر مهندس نرم‌افزاری باید بداند که تغییر، امری طبیعی است. نکوشید که با آن بجنگید.

همان‌طور که در بخش ۲-۱-۱ گفته شد، برنامه‌های تحت وب، یکی از گروه‌های متماز نرم‌افزارند. با این حال، می‌توان استدلال کرد که برنامه‌های تحت وب، متفاوتند. پاول [Pow98] پیشنهاد می‌کند که

«تا آن هنگام که هر گونه پایداری بیهوده، وب به چیزی کاملاً متفاوت تبدیل شده است.»
لویی مونیه

[1] در حیطه این کتاب، عبارت «برنامه‌ی تحت وب» شامل همه چیز از یک صفحه وب ساده می‌شود که ممکن است به مصرف کننده کمک کند تا بدهی خود را برای خرید خودرو پرداخت کند تا یک وب‌سایت جامع و فراگیر که خدمات مسافری کامل را برای بازرگانان و افراد عادی که به تعطیلات می‌روند، فراهم می‌آورد. این گروه شامل وب‌سایت‌های کامل، عملکردهای خاص موجود در وب‌سایت‌ها و برنامه‌های پردازش اطلاعات می‌شود که روی اینترنت یا روی یک اینترانت یا اکسترانت قرار دارند.

برنامه‌های کاربردی سیستم‌های مبتنی بر وب «شامل مخلوطی از انتشارات چاپی و توسعه‌ی نرم‌افزار، از بازاریابی و کار با کامپیوتر، از ارتباطات داخلی و ارتباطات خارجی و از هنر و فن‌آوری هستند.» در اکثریت وسیع برنامه‌های تحت وب، صفحه‌های زیر مشاهده می‌شود.

چه خصوصیاتی برنامه‌های تحت وب را از سایر نرم‌افزارها متمایز می‌سازند؟

میزان تمرکز شبکه. برنامه‌های تحت وب روی یک شبکه قرار دارند و باید نیازهای جامعه‌ای متنوع از کلاینت‌ها را برآورده سازند. شبکه ممکن است برقراری ارتباط و دستیابی جهانی را میسر سازد (یعنی اینترنت) یا برقراری ارتباط و دستیابی جهانی را در سطحی محدودتر امکان‌پذیر کند (مثلاً یک اینترنت شرکتی).

همروندی.[1] ممکن است یک باره تعداد بسیاری از کاربران به برنامه‌ی تحت وب دستیابی داشته باشند. در بسیاری موارد، الگوهای استفاده در میان کاربران نهایی ممکن است تفاوتی گسترده داشته باشد.

بار غیرقابل پیش‌بینی. تعداد کاربران برنامه‌های تحت وب ممکن است از روزی به روز دیگر ده یا صد برابر شود. ممکن است دوشنبه صد کاربر ظاهر شوند و روز سه‌شنبه ده‌هزار کاربر داشته باشد.

کارایی. اگر کاربر یک برنامه‌ی تحت وب باید مدتی طولانی منتظر بماند (برای دستیابی، پردازش از طرف سرور، فرمت‌بندی و نمایش از طرف کلاینت)، ممکن است تصمیم بگیرد به‌جای دیگری برود.

قابلیت دسترسی. گرچه انتظار ۱۰۰ درصد قابلیت دسترسی، غیر منطقی است، کاربران برنامه‌های تحت وب پرطرفدار غالباً تقاضای دسترسی ۲۴ ساعته در هفت روز هفته و ۱۲ ماه سال را دارند. کاربران مقیم در استرالیا یا آسیا ممکن است متقاضی دستیابی طی زمان‌هایی باشند که برنامه‌های کاربردی سنتی در امریکای شمالی برای امور نگهداری، از خط خارج شده‌اند.

داده-محوری.[2] عملکرد اصلی بسیاری از برنامه‌های تحت وب، استفاده از آبَررسانه‌ها[3] برای ارائه متون، گرافیک، صوت و تصاویر ویدیویی به‌کاربران نهایی است. به علاوه، برنامه‌های کاربردی تحت وب معمولاً برای دستیابی به اطلاعاتی به‌کار می‌رود که روی بانک‌های اطلاعاتی وجود دارند و بخشی از محیط مبتنی بر وب (نظیر برنامه‌های کاربردی تجارت الکترونیکی یا مالی) می‌شوند.

حساس به محتویات. کیفیت و ماهیت زیبایی‌شناختی محتویات از جمله مهمترین عوامل تعیین کننده کیفیت در برنامه‌های تحت وب است.

تکامل پیوسته. بر خلاف نرم‌افزارهای کاربردی سنتی که طی یک سری نسخه‌های برنامه‌ریزی‌شده و با فاصله زمانی تکامل پیدا می‌کنند، برنامه‌های تحت وب پیوسته در حال تکامل هستند. این برای برخی برنامه‌های تحت وب غیر عادی نیست (به ویژه از لحاظ محتویات) که بر اساس یک زمان‌بندی دقیقه به دقیقه بهنگام‌سازی شوندیا اینکه محتویات آنها بسته به درخواست، مستقلاً محاسبه شود.

[1] concurrency
[2] data driven
[3] hypermedia

بی‌واسطگی.[1] بی‌واسطگی- نیاز اجباری برای رساندن سریع نرم‌افزار به بازار- یکی از خصوصیات بسیاری از دامنه‌های کاربردی است ولی برنامه‌های تحت وب غالباً یک زمان رسیدن به بازار دارند که می‌تواند چند روز یا چند هفته باشد.[2]

امنیت. از آنجا که برنامه‌های تحت وب از طریق شبکه در دسترس قرار می‌گیرند، محدودکردن جمعیتی از کاربران نهایی که به برنامه دستیابی داشته باشند، اگر غیر ممکن نباشد، دشوار است. برای محافظت از محتویات حساس و فراهم ساختن شیوه‌های امن برای انتقال داده ها معیارهای امنیتی قوی‌ای باید پیاده‌سازی شوند.

زیبایی‌شناسی. یک بخش غیر قابل انکار از جاذبه‌ی برنامه‌های تحت وب، ظاهر آنهاست. هنگامی که یک برنامه‌ی کاربردی برای بازاریابی یا فروش محصولات یا ایده‌ها طراحی شده باشد، زیبایی شناسی نیز ممکن است به اندازه موفقیت در طراحی فنی اهمیت داشته باشد.

می‌توان استدلال کرد که سایر گروه‌های بحث شده در بخش ۲-۱-۱ گاهی برخی از صفات ذکرشده در بالا را از خود نشان دهند. ولی برنامه‌های تحت وب تقریباً همواره همه‌ی آنها را نشان می‌دهند.

۳-۱ مهندسی نرم‌افزار

به‌منظور ساخت نرم‌افزارهایی که آمادگی برآورده ساختن چالش‌های قرن بیست و یکم را داشته باشند، باید چند واقعیت ساده را بدانید.

نکته‌ی کلیدی
پیش از آنکه برای مساله راهکاری بیابید، آن را درک کنید.

- نرم‌افزارها در واقع به‌طور عمیق در همه‌ی شؤونات زندگی ما تعبیه شده‌اند و در نتیجه، تعداد افرادی که به ویژگی‌ها و عملکردهای فراهم آمده توسط یک برنامه‌ی کاربردی خاص علاقه دارند[3] به‌طور چشمگیری افزایش یافته است. هنگامی که قرار است یک برنامه‌ی کاربردی یا سیستم تعبیه‌شده‌ی جدید ساخته شود، صداهای زیادی را باید شنید و گاهی به‌نظر می‌رسد که هر کدام از آنها دارای عقیده‌ای با اندک تفاوت درباره ویژگی‌ها و عملکردهایی هستند که باید تحویل شود. *لازم می‌آید که برای درک مساله قبل از توسعه‌ی یک راهکار نرم‌افزاری، تلاشی هماهنگ به عمل آید.*
- خواسته‌های فن‌آوری اطلاعات مورد تقاضای افراد، شرکت‌های تجاری و دولت‌ها هر ساله پیچیده‌تر می‌شود. اکنون تیم‌های بزرگی از افراد، برنامه‌های کامپیوتری ایجاد می‌کنند که زمانی توسط یک نفر به تنهایی ساخته می‌شدند. نرم‌افزارهای پیچیده‌ای که زمانی در یک محیط کامپیوتری قابل پیش‌بینی و خود محتوی پیاده‌سازی می‌شدند، اکنون در هر چیزی از دستگاه‌های الکترونیکی مصرفی گرفته تا دستگاه‌های پزشکی و سیستم‌های تسلیحاتی تعبیه شده‌اند. پیچیدگی این سیستم‌های کامپیوتری جدید نیازمند بذل توجه دقیق به تعامل همه‌ی عناصر سیستم است. *لازم است که طراحی، فعالیتی محوری باشد.*

نکته‌ی کلیدی
طراحی، یکی از فعالیت‌های محوری در مهندسی نرم‌افزار است.

[1] immediacy

[2] با ابزارهای مدرن، صفحات وب پیچیده‌ای را در عرض چند دقیقه می‌توان تهیه کرد.

[3] این افراد را بعداً در این کتاب «طرف‌های ذی‌نفع» خواهیم خواند.

نکته‌ی کلیدی
کیفیت و قابلیت نگهداری، هردو نتیجه طراحی خوب هستند.

- افراد، شرکت‌های تجاری و دولت‌ها به‌طور فزاینده‌ای برای تصمیم‌گیری‌های راهبردی و تاکتیکی خود و نیز برای عملیات و کنترل روزمره خود به نرم‌افزارها تکیه می‌کنند. اگر نرم‌افزاری با شکست مواجه شود، افراد و شرکت‌های تجاری، ممکن است شاهد هر چیزی از یک ناراحتی کوچک گرفته تا شکست‌های فاجعه بار باشند. *لازم است که نرم‌افزار، کیفیت بالا یی از خود نشان دهد.*
- با رشد ارزش شناخته‌شده‌ی یک برنامه‌ی کاربردی خاص، این احتمال وجود دارد که تعداد کاربران آن و عمر استفاده از آن نیز رشد کند. با افزایش تعداد کاربران و زمان استفاده، تقاضا برای بهسازی و انطباق نیز رشد می‌کند. *لازم است که نرم‌افزار قابل نگهداری باشد.*

این واقعیت‌های ساده به یک نتیجه منجر می‌شود: *نرم‌افزار در تمامی اشکال خود و در همه‌ی دامنه‌های کاربردی‌اش باید مهندسی شود.* و این ما را به‌عنوان این کتاب- مهندسی نرم‌افزار – رهنمون می‌شود.

گرچه صدها نویسنده، تعاریفی شخصی از مهندسی نرم‌افزار ارائه داده‌اند، تعریفی که فریتزباور [Nau69] در یک همایش مهم ارائه کرده است هنوز هم مبنای بحث ما را تشکیل می‌دهد:

> [مهندسی نرم‌افزار عبارت است از] وضع اصول مهندسی بجا و مناسب و استفاده از آنها برای به دست آوردن یک نرم‌افزار مقرون به صرفه که قابل اطمینان بوده، روی ماشین‌های واقعی به طرزی کارآمد عمل کند.

خواننده وسوسه می‌شود که نکته‌ای به این تعریف بیفزاید[1]. این تعریف چیز زیادی درباره جنبه‌های فنی کیفیت نرم‌افزار نمی‌گوید؛ این تعریف به‌طور مستقیم، نیاز به راضی نمودن مشتری یا تحویل به موقع محصول را مشخص نمی‌کند؛ در آن ذکری از اهمیت موازین و استانداردها به عمل نمی‌آید؛ از اهمیت یک فرایند بالغ سخن به میان نمی‌آورد و با این همه، تعریف فریتزباور یک تعریف بنیادی است. «اصول مهندسی مناسب و بجا» کدامند که در بسط نرم‌افزار کامپیوتری قابل اجرا هستند؟ چطور می‌توان نرم‌افزار «مقرون به صرفه‌ای» ساخت که «قابل اطمینان» باشد؟ برای ایجاد برنامه‌های کامپیوتری که نه تنها یک ماشین واقعی بلکه «ماشین‌های واقعی» متفاوت به‌طرزی کارآمد عمل کنند، چه چیزهایی لازم است؟ اینها پرسش‌هایی است که همچنان مهندسان نرم‌افزار را به چالش فرا می‌خوانند.

مهندسی نرم‌افزار را چگونه تعریف می‌کنیم؟

IEEE [IEE93a] تعریف مفهومی‌تری ارائه می‌دهد:

> مهندسی نرم‌افزار: (۱) کاربرد یک روش سیستماتیک، علمی و کمیت‌پذیر در بسط، راه‌اندازی و نگهداری نرم‌افزار؛ یعنی استفاده از مهندسی در نرم‌افزار. (۲) مطالعه روش‌ها به‌صورت ذکرشده در (۱).

و با این وجود، یک روش «سیستماتیک، منضبط و کمیت‌پذیر» که یک تیم به‌کار می‌برد ممکن است برای تیم دیگر، پرزحمت به نظر آید. ما به انضباط نیاز داریم ولی به انطباق‌پذیری[2] و سرعت انتقال هم نیازمندیم.

نکته‌ی کلیدی
«مهندسی بیش از یک رشته علمی یا مجموعه اطلاعات است؛ مهندسی یک فعل، واژه‌ای برای کنش و راهی برای نگرش به مساله است.»
اسکات ویتمیر

[1] برای چندین تعریف دیگر از مهندسی *نرم‌افزار*، وب‌سایت زیر را ببینید:
www.answers.com/topic/software-engineering#wp-_note-13

[2] adaptability

شکل ۳-۱ لایه‌های مهندسی نرم‌افزار

مهندسی نرم‌افزار یک فن‌آوری لایه‌ای است. با توجه به شکل ۳-۱، هر روش مهندسی (از جمله مهندسی نرم‌افزار) باید متکی بر تعهد سازمانی به کیفیت باشد. مدیریت کیفیت فراگیر (TQM)[1]، شش سیگما[2] و سایر فلسفه‌های مشابه[3] رواج‌دهنده‌ی فرهنگ بهبود پیوسته‌ی فرایند هستند و همین فرایند است که سرانجام به توسعه‌ی روش‌های کارامدتر در مهندسی نرم‌افزار می‌انجامد. سنگ بنای نگهدارنده‌ی مهندسی نرم‌افزار، توجه به کیفیت است.

بنیاد مهندسی نرم‌افزار، لایه‌ی *فرایند* است. مهندسی نرم‌افزار به مثابه چسبی عمل می‌کند که لایه‌های فنّاوری را به هم نگه می‌دارد و بسط موجه و به‌موقع نرم‌افزارهای کامپیوتری را میسر می‌سازد. فرایند، *چارچوبی* را تعریف می‌کند که باید برای تحویل مؤثر فنّاوری مهندسی نرم‌افزار وضع شود. فرایند نرم‌افزار، پایه‌ای برای کنترل مدیریتی پروژه‌های نرم‌افزاری تشکیل داده بستری برای اِعمال روش‌های فنی، تولید محصولات کاری (مدل‌ها، مستندات، داده‌ها، گزارش‌ها، فرم‌ها و غیره)، تعیین مراحل، حصول اطمینان از کیفیت و مدیریت مناسب تغییرات ایجاد می‌کند.

روش‌های مهندسی نرم‌افزار، شیوه‌های فنی برای ساخت نرم‌افزار را فراهم می‌آورند. این روش‌ها شامل آرایه‌ی وسیعی از وظایف از جمله: تحلیل خواسته‌ها، طراحی، ساخت برنامه‌ها، آزمایش و پشتیبانی می‌شوند. روش‌های مهندسی نرم‌افزار متکی بر یک مجموعه اصول بنیادی است که بر تمام زمینه‌های فنّاوری حاکم بوده شامل فعالیت‌های مدلسازی و فنون توصیفی دیگر می‌شوند.

ابزارهای مهندسی نرم‌افزار، متضمن پشتیبانی خودکار یا نیمه خودکار برای فرایند و روش‌ها هستند. هنگامی که ابزارها گرد هم آیند به طوری که اطلاعات ایجاد شده توسط یک ابزار، توسط ابزارهای دیگر قابل استفاده باشند، سیستمی برای پشتیبانی بسط نرم‌افزار شکل می‌گیرد که *مهندسی نرم‌افزار به کمک کامپیوتر* (CASE)[4] نام دارد.

نکته‌ی کلیدی

مهندسی نرم‌افزار شامل یک فرایند، روش‌هایی برای مدیریت و مهندسی کردن نرم‌افزار و یک سری ابزار می‌شود.

مرجع وب

CrossTalk سایت‌های تخصصی است که درباره فرایند، روش‌ها و ابزارهای اطلاعاتی عملی فراهم می‌سازد. می‌توان آن را در وب سایت زیر یافت:
www.stsc.hill.af.mil

عناصر فرایند نرم‌افزار کدامند؟

۱-۴ فرایند نرم‌افزار

فرایند، مجموعه‌ای از فعالیت‌ها، کنش‌ها[5] و وظایف است که هنگام ایجاد یک محصول کاری اجرا می‌شوند. یک *فعالیت*، کوششی است در جهت رسیدن به هدفی گسترده (مانند برقراری ارتباط با افراد ذی‌نفع) و دامنه‌ی کاربرد، اندازه پروژه، پیچیدگی تلاش‌ها یا میزان جدیت به‌کارگیری مهندسی نرم‌افزار

[1] Total Quality Mnagement
[2] six sigma
[3] مدیریت کیفیت و روش‌های مرتبط در فصل ۱۴ و سرتاسر بخش سوم از این کتاب بحث شده‌اند.
[4] Computer Aided Software Engineering
[5] actions

هرچه که باشد، این کوشش باید انجام شود. یک *کنش* (مانند طراحی معماری) شامل مجموعه‌ای از وظایف می‌شود که یک محصول کاری عمده را تولید می‌کنند (مانند مدل طراحی معماری). *وظیفه* به یک شیء کوچک ولی کاملاً معین (مانند اجرای آزمون روی یک واحد) توجه دارد که نتیجه‌ای ملموس ایجاد کند.

در حیطه‌ی مهندسی نرم‌افزار، فرایند، دستورالعمل نهایی برای چگونگی ساخت نرم‌افزارهای کامپیوتری نیست بلکه یک روش انطباق‌پذیر است که افراد کننده‌ی کار (تیم نرم‌افزار) به کمک آن می‌توانند مجموعه‌ی مناسبی از کنش‌ها و وظایف کاری را برگزینند. هدف، همیشه تحویل سروقت نرم‌افزار با کیفیت کافی به‌منظور راضی نمودن کسانی است که ایجاد آن را پشتیبانی کرده‌اند و کسانی که از آن استفاده می‌کنند.

چارچوب فرایند با تعیین تعداد کوچکی از *فعالیت‌های چارچوبی* که برای کلیه پروژه‌های نرم‌افزاری قابل استفاده باشند، صرف نظر از اندازه و پیچیدگی آنها، شالوده‌ای برای یک فرایند مهندسی نرم‌افزار کامل پی‌ریزی می‌کند. به علاوه، چارچوب فرایند شامل مجموعه‌ای از *فعالیت‌های چتری* می‌شود که در سرتاسر فرایند نرم‌افزار قابل اِعمال هستند. یک «چارچوب فرایند» کلی برای مهندسی نرم‌افزار شامل پنج فعالیت می‌شود:

ارتباطات (Communication). پیش از اینکه هرگونه کار فنی آغاز شود، برقراری ارتباط و همکاری با مشتری (و سایر افراد ذی‌نفع[1]) بسیار مهم است. هدف، درک اهداف طرف‌های ذی‌نفع برای پروژه و جمع‌آوری خواسته‌هایی است که می‌توانند ویژگی‌ها و قابلیت‌های عملیاتی نرم‌افزار را تعیین کنند.

برنامه‌ریزی (Planning). هر سفر پیچیده‌ای را با در اختیار داشتن یک نقشه می‌توان ساده کرد. یک پروژه‌ی نرم‌افزاری، سفری پیچیده است و فعالیت برنامه‌ریزی، «نقشه‌ای» ایجاد می‌کند که به راهنمایی تیم در انجام این سفر کمک می‌کند. این نقشه – که نقشه پروژه نرم‌افزار نامیده می‌شود – با توصیف وظایف فنی که قرار است اجرا شوند، خطرات احتمالی، منابعی که مورد نیاز خواهند بود، محصولات کاری‌ای که باید تولید شوند و زمان‌بندی کاری، مهندسی نرم‌افزار را مشخص می‌کند.

مدل‌سازی (Modeling). شما خواه یک انبوه‌ساز باشید، خواه یک پل‌ساز یا مهندس هوانوردی، نجار باشید یا معمار، هر روز با مدل‌ها کار می‌کنید. اِتودی می‌زنید تا تصویر بزرگ را درک کنید – اینکه از نظر معماری چه ظاهری دارد، بخش‌های سازنده‌اش چگونه با هم جور در خواهند آمد، و بسیاری خصوصیت‌های دیگر. در صورت نیاز، این اتود را به جزئیات بیشتر و بیشتر پالایش می‌کنید تا مسأله و چگونگی حل آن را بهتر درک کنید. مهندس نرم‌افزار با ایجاد مدل‌هایی جهت درک بهتر خواسته‌ها و طراحی که به این خواسته‌ها برسد، همین کار را می‌کند.

ساخت (Construction). این فعالیت، تولید کدها (چه دستی و چه خودکار) و آزمون لازم برای آشکارکردن خطاهای موجود در کدها را با هم تلفیق می‌کند.

[1] طرف‌های ذی‌نفع به کسانی گفته می‌شود که در پیامدهای موفق پروژه سهم دارند-حدیران تجاری، کاربران نهایی، مهندسان نرم‌افزار، افراد پشتیبان و غیره.

«فرایند تعیین می‌کند که چه کسی چه کاری را در چه زمانی و چگونه انجام دهد تا به هدفی معین برسد.»
ایوار جیکابسون

پنج فعالیت چارچوبی در فرایند کلی را نام ببرید.

«دانستن استدلال می‌کرد که برای طبیعت باید توضیحی ساده وجود داشته باشد. چون خداوند از روی هوس کار نکرده است. ولی چنین ایمانی را به مهندس نرم‌افزار نمی‌توان داشت. بیشتر پیچیدگی‌هایی که او باید مدیریت کند، این گونه‌اند.»
فرد بروکز

استقرار (Deployment). نرم‌افزار (به‌عنوان یک موجودیت کامل یا در مرحله‌ای از تکامل) به مشتری تحویل می‌شود که محصول تحویل شده را ارزیابی کرده بر اساس این ارزیابی، بازخوردی ارائه دهد.

این پنج فعالیت کلیِ چارچوبی را می‌توان طی توسعه‌ی برنامه‌های کوچک و ساده، در ایجاد برنامه‌های تحت وب و برای مهندسی سیستم‌های کامپیوتری پیچیده و عظیم به‌کار برد. جزئیات فرایند نرم‌افزار در هر مورد کاملاً متفاوت خواهد بود، ولی فعالیت‌های چارچوبی همین‌ها خواهند بود.

برای بسیاری از پروژه‌های نرم‌افزاری، فعالیت‌های چارچوبی به موازات پیشرفت پروژه به‌صورت تکراری به‌کاربرده می‌شوند. یعنی فعالیت‌های ارتباطات، برنامه ریزی، مدل‌سازی، ساخت و استقرار به‌طور مکرر در چند دور تکرار پروژه به‌کار برده می‌شوند. در هر دور تکرار پروژه، یک *نسخه* (افزایش)[1] از نرم‌افزار ایجاد می‌شود که زیر مجموعه‌ای از قابلیت‌های عملیاتی و ویژگی‌های نرم‌افزار کامل را در اختیار افراد ذی‌نفع قرار می‌دهد. با تولید هر نمو، نرم‌افزار کامل و کامل‌تر می‌شود.

فعالیت‌های چارچوبی فرایند مهندسی نرم‌افزار توسط تعدادی از *فعالیت‌های چتری* تکمیل می‌شود. به‌طور کلی، فعالیت‌های چتری در سرتاسر یک پروژه‌ی نرم‌افزاری به‌کار برده می‌شوند و به تیم نرم‌افزاری کمک می‌کنند تا پیشرفت، کیفیت، تغیر و ریسک را کنترل کند. فعالیت‌های چتری متداول عبارتند از:

کنترل و پیگیری پروژه‌های نرم‌افزاری – به تیم نرم‌افزاری امکان می‌دهد تا پیشرفت را در مقایسه با نقشه‌ی پروژه بسنجد و هرگونه کنش لازم را برای حفظ زمان‌بندی به عمل آورد.

مدیریت ریسک – خطراتی را ارزیابی می‌کند که ممکن است بر نتیجه‌ی پروژه یا کیفیت محصول تأثیر بگذارند.

تضمین کیفیت نرم‌افزار – فعالیت‌های لازم برای حصول اطمینان از کیفیت نرم‌افزار را معین می‌کند.

بازبینی‌های فنی – محصولات کاریِ مهندسی نرم‌افزار را در تلاش برای آشکار کردن خطاها قبل از انتشار آنها در فعالیت بعدی و برطرف کردن آنها ارزیابی می‌کند.

اندازه‌گیری – موازینی از فرایند، پروژه و محصول را تعریف می‌کند که نیازهای طرف‌های ذی‌نفع را برطرف می‌سازند؛ از آن می‌توان همراه با سایر فعالیت‌های چارچوبی و چتری استفاده کرد.

مدیریت پیکربندی نرم‌افزار – اثرات تغییرات را در سرتاسر فرایند نرم‌افزار مدیریت می‌کند.

مدیریت قابلیت استفاده‌ی مجدد – ملاک‌های مربوط به استفاده‌ی مجدد (از جمله قطعات نرم‌افزاری) را تعریف می‌کند و سازوکارهایی برای دستیابی به قطعات قابل استفاده‌ی مجدد برقرار می‌سازد.

تهیه و تولید محصول کاری – شامل فعالیت‌های لازم برای ایجاد محصولات کاری از قبیل مدل‌ها، مستندات، وقایع نگارها (کارنامه‌ها)، فرم‌ها و فهرست‌ها می‌شود.

هر کدام از این فعالیت‌های چتری را در این کتاب به تفصیل شرح خواهیم داد. قبلاً در این بخش متذکر شدیم که فرایند مهندسی نرم‌افزار یک دستورالعمل نهایی و غیر قابل تغییر نیست که تیم نرم‌افزاری باید با تعصب از آن پیروی کند بلکه باید سریع الانتقال و انطباق‌پذیر باشد (برای مسأله،

نکته‌ی کلیدی

فعالیت‌های چتری در سرتاسر فرایند نرم‌افزار رخ می‌دهند و اکنون توجه آنها اساساً مدیریت پروژه، پیگیری و کنترل است.

نکته‌ی کلیدی

انطباق فرایندهای نرم‌افزار برای موفقیت پروژه ضروری است.

[1] increment

برای پروژه، برای تیم و برای فرهنگ سازمانی). بنابراین، فرایندی که برای یک پروژه پذیرفته می‌شود، ممکن است با فرایند پذیرفته شده برای پروژه‌ای دیگر تفاوتی چشمگیر داشته باشد. از جمله این تفاوت‌ها عبارتند از:

- جریان کلی فعالیت‌ها، کنش‌ها و وظایف و بستگی آنها به یکدیگر.
- درجه‌ی تعریف کنش‌ها و وظایف در هر فعالیت چارچوبی.
- درجه‌ی شناسایی محصولات کاری و نیاز به آنها.
- شیوه اِعمال فعالیت‌های تضمین کیفیت.
- درجه‌ی کلی جزئیات به‌کار رفته در توصیف فرایند.
- درجه‌ی دخالت مشتری و طرف‌های ذی‌نفع در پروژه.
- سطح استقلال داده شده به تیم نرم‌افزار.
- درجه‌ی توصیف نقش‌ها و سازمان‌دهی تیم.

در بخش اول این کتاب، فرایند نرم‌افزار را با جزئیات قابل ملاحظه‌ای بررسی خواهیم کرد. مدل‌های فرایندی تجویزی (فصل ۲) بر جزئیات تعریف، شناسایی و کاربرد فعالیت‌ها و وظایف فرایند تأکید دارند. هدف آنها بهبود بخشیدن به کیفیت سیستم، بالا بردن قابلیت مدیریت پروژه‌ها، قابل پیش‌بینی کردن تاریخ‌های تحویل و هزینه‌ها و راهنمایی تیم‌های مهندسان نرم‌افزار در اجرای کارهای لازم برای ساخت یک سیستم است. متاسفانه، مواقعی هست که این اهداف برآورده نمی‌شود. اگر مدل‌های تجویزی با تعصب و بدون انطباق‌پذیری به‌کاربرده شوند، می‌توانند سطح بوروکراسی مرتبط با سیستم‌های کامپیوتری را افزایش دهند و مشکلات جدی برای طرف‌های ذی‌نفع به‌بار آورند.

مدل‌های فرایندی چابک (فصل ۳) بر «سرعت» تأکید دارند و مجموعه‌ای از اصول را دنبال می‌کنند که به یک روش غیر رسمی‌تر (ولی به قول طرفداران آن، نه با کارایی کمتر) برای فرایند نرم‌افزار منجر می‌شود. این مدل‌های فرایندی عموماً با عنوان «چابک» مشخص می‌شود زیرا بر قابلیت مانور و انطباق‌پذیری تأکید دارند. این فرایندها برای انواع بسیاری از پروژه‌ها مناسب بوده به‌ویژه هنگام مهندسی برنامه‌های کاربردی تحت وب مفید واقع می‌شوند.

۱-۵ مهندسی نرم‌افزار در عمل

در بخش ۴-۱ یک مدل فرایند نرم‌افزار کلی معرفی کردیم که مرکب از مجموعه‌ای از فعالیت‌هاست که چارچوبی برای پیاده‌سازی مهندسی نرم‌افزار در عمل وضع می‌کنند. فعالیت‌های چارچوبی کلی – ارتباطات، برنامه ریزی، مدل‌سازی، ساخت و استقرار – و فعالیت‌های چتری یک معماری اسکلتی برای کار مهندسی نرم‌افزار وضع می‌کنند. ولی مهندسی نرم‌افزار در عمل چگونه درست از آب در خواهد آمد؟ در بخش‌هایی که به دنبال خواهد آمد، درکی بنیادی از مفاهیم و اصول کلی به دست خواهید آورد که در فعالیت‌های چارچوبی کاربرد دارند.[1]

۱-۵-۱ جوهر عمل

جورج پولیا در یک کتاب کلاسیک با عنوان «چگونگی حل مسأله» [Pol45] که قبل از وجود

[1] در آینده هنگام بحث درباره روش‌های مهندسی نرم‌افزار و فعالیت‌های چتری خاص، بهتر است بخش‌های مربوط را در این فصل دوباره مطالعه کنید.

مدل‌های فرایند چه تفاوت‌هایی با یکدیگر دارند؟

«احساس می‌کنم هر دستور غذا مثل آهنگی است که آشپز هوشمند می‌تواند هر بار آن را با یک واریاسیون جدید بنوازد.»

مادام بتوا

فرایند «چابک» چه خصوصیاتی دارد؟

مرجع وب

مجموعه‌ای متنوع از نقل‌قول‌های جالب درباره مهندسی نرم‌افزار در عمل را می‌توان در وب سایت زیر یافت:
www.literateprogramming.com

کامپیوترهای مدرن نوشته شده است، جوهر حل مسأله و در نتیجه «جوهر عمل» در مهندسی نرم‌افزار را چنین مطرح می‌کند:

۱. شناخت مسأله (برقراری ارتباط و تحلیل).
۲. طرح ریزی برای یک حل (مدل‌سازی و طراحی نرم‌افزار).
۳. اجرای برنامه‌ریزی (ایجاد کد).
۴. بررسی نتیجه برای صحت (آزمایش و تضمین کیفیت).

در حیطه مهندسی نرم‌افزار، این مراحل (که عقل سلیم آنها را حکم می‌کند) به یک سری پرسش‌های اساسی می‌انجامد [برگرفته از Pol45]:

شناخت مسأله. گاهی پذیرفتن این واقعیت دشوار است ولی بیشترمان هنگامی که مسأله‌ای به ما ارائه می‌شود، احساس می‌کنیم که به غرور ما لطمه وارد آمده است. چند ثانیه‌ای گوش می‌کنیم و سپس فکر می‌کنیم، *آهان، گرفتم، حالا حلش می‌کنیم.* متأسفانه، درک و شناخت مسأله همیشه آسان نیست. بد نیست زمان اندکی را صرف پاسخ گفتن به چند پرسش ساده کنیم:

- چه کسی از حل مسأله منتفع می‌شود؟ یعنی، طرف‌های ذی‌نفع چه کسانی هستند؟
- مجهولات کدام‌ها هستند؟ چه داده‌ها، توابع و ویژگی‌هایی برای حل مناسب مسأله لازم است؟
- آیا مسأله را می‌توان به قطعات کوچکتر تبدیل کرد که درک آنها ساده‌تر باشد؟
- آیا مسأله را می‌توان با یک نمودار ارائه داد؟ آیا یک مدل تحلیلی برای آن قابل ایجاد است؟

برنامه‌ریزی حل. اکنون مسأله را درک کرده‌اید (یا فکر می‌کنید که درک کرده‌اید) و نمی‌توانید برای نوشتن کدها صبر کنید. پیش از آن که اقدام به کدنویسی کنید قدری حوصله به خرج دهید و اندکی برنامه‌ریزی کنید:

- آیا مسائلی مشابه را قبلاً دیده اید؟ آیا الگوهایی وجود دارند که در حل احتمالی قابل تشخیص باشند؟ آیا نرم‌افزاری برای پیاده‌سازی داده‌ها، قابلیت‌های عملیاتی و ویژگی‌های مورد نیاز وجود دارد؟
- آیا مسائل مشابه را می‌توان حل کرد؟ اگر پاسخ مثبت است، آیا عناصر حل قابلیت استفاده‌ی مجدد را دارند؟
- آیا می‌توان مسأله را به مسائل فرعی تقسیم کرد؟ اگر پاسخ مثبت است، آیا حل مسأله به آسانی از مسائل فرعی هویدا هست؟
- آیا می‌توان حلی را به شیوه‌ای ارائه کرد که به پیاده‌سازی اثربخش منجر گردد؟ آیا یک مدل طراحی قابل ایجاد هست؟

اجرای برنامه‌ریزی. طراحی‌ای که شما ایجاد کرده‌اید، برای سیستمی که در صدد ساخت آن هستید، به‌عنوان یک نقشه راه عمل می‌کند. ممکن است راه‌های انحرافی غیر منتظره وجود داشته باشد و امکان دارد که در طی مسیر، راه‌های بهتری هم کشف کنید ولی این «نقشه» به شما کمک می‌کند تا بدون اینکه گم بشوید، به راه خود ادامه دهید.

- آیا راهکار با برنامه‌ریزی مطابقت دارد؟ آیا کد منبع تا مدل طراحی قابل ردگیری هست؟
- آیا هر بخش از راهکار درست است؟ آیا طراحی و کدها بازبینی شده‌اند یا بهتر از آن، آیا الگوریتم‌ها از نظر درستی بررسی شده‌اند؟

اندرز

ممکن است استدلال کنید که روش پولیا چیزی جز عقل سلیم نیست. درست، ولی جالب است که همین عقل سلیم غالب اوقات در جهان نرم‌افزار به‌کاربرده نمی‌شود.

«در حل هر مسأله، ذره‌ای کشف وجود دارد.»
جورج پولیا

بررسی نتیجه. نمی‌توان اطمینان یافت که راهکار ارائه شده کامل است ولی می‌توان مطمئن شد که تعداد آزمون‌های کافی برای برملاساختن هرچه بیشتر خطاها طراحی شده‌اند.

- آیا می‌توان هر مؤلفه از راهکار را آزمود؟ آیا راهبرد آزمون منطقی پیاده‌سازی شده است؟
- آیا این راهکار، نتایجی ایجاد می‌کند که با داده‌ها، قابلیت‌های عملیاتی و ویژگی‌های مورد نیاز مطابقت داشته باشد؟ آیا نرم‌افزار از لحاظ کلیه‌ی خواسته‌های طرف‌های ذی‌نفع وارسی شده است؟

جای شگفتی نیست که این رویکرد عمدتاً از عقل سلیم نشأت گرفته است. در واقع، منطقی است که بگوییم یک روش مبتنی بر عقل سلیم برای مهندسی نرم‌افزار هرگز شما را گمراه نخواهد کرد.

۲-۵-۱ اصول کلی

واژه‌ی *اصل* (principle) به معنای یک قانون یا فرض زیربنایی است که در یک سیستم فکری وجود آن ضروری است. در سرتاسر این کتاب درباره اصولی با سطوح انتزاع متفاوت بحث خواهیم کرد. برخی از این اصول به مهندسی نرم‌افزار به‌عنوان یک کلیت توجه دارند و برخی دیگر به یک فعالیت چارچوبی کلی مشخص (مانند **ارتباطات**)، می‌پردازند و از آن گذشته عده‌ای نیز بر کنش‌های مهندسی نرم‌افزار (مانند طراحی معماری) یا وظایف فنی (مانند نوشتن یک سناریوی کاربرد) تأکید دارند. این اصول، در هر سطحی از تأکید که قرار داشته باشند، به شما در برقراری یک فضای فکری برای مهندسی نرم‌افزار در عمل، کمک خواهند کرد و اهمیت آنها از این لحاظ است. دیوید هوکر [Hoo96] هفت اصل را مطرح نموده است که کانون توجه آنها کار در مهندسی نرم‌افزار به‌عنوان یک کلّیت است. این اصول را در بندهای زیر عیناً تکرار می‌کنیم[1]:

اصل نخست: دلیل وجود سیستم

هر سیستم به یک وجود نیاز دارد: این که برای کاربرانش ارزشی فراهم سازد. همه‌ی تصمیم گیری‌ها باید با مد نظر داشتن این نکته انجام شود. پیش از مشخص‌کردن یک خواسته‌ی سیستم، پیش از توجه به قطعه‌ای از قابلیت عملیاتی یک سیستم و قبل از تعیین سکوی سخت افزاری یا فرایندهای توسعه‌ای، از خود پرسش‌هایی از این قبیل بپرسید: «آیا این کار ارزشی واقعی به سیستم اضافه می‌کند؟» اگر پاسخ منفی است، آن کار را نکنید. همه‌ی اصول دیگر مؤید این اصل هستند.

اصل دوم: ساده نگه داشتن

طراحی نرم‌افزار یک فرایند تصادفی نیست و در هر تلاش برای طراحی باید عوامل فراوانی را در نظر گرفت. همه‌ی طراحی‌ها باید تا حد امکان ساده باشد ولی نه ساده‌تر. این باعث می‌شود که داشتن یک سیستم قابل‌فهم‌تر با قابلیت نگهداری بالاتر آسان تر شود. این بدان معنا نیست که ویژگی‌های سیستمی، حتی آنهایی که درونی هستند باید به بهانه ساده‌سازی کنار گذاشته شوند. در واقع، طراحی‌های ظریف‌تر معمولاً طراحی‌های ساده‌ترند. ساده در عین حال به معنی «سریع و نامنظم» هم نیست. در حقیقت، ساده‌سازی نیاز به مقادیر معتنابهی فکر و کار در چند دور تکرار دارد. نتیجه، نرم‌افزاری با قابلیت نگهداری بالاتر و کم خطاتر خواهد بود.

اندرز

پیش از شروع یک پروژه نرم‌افزاری، بهتر حاصل کند که نرم‌افزار دارای هدفی تجاری است و کاربران در آن ارزشی خواهند یافت.

«در سادگی شکوه خاصی نهفته است که والاتر از ظرافت نهفته در خوش طبعی است.»
الکساندر پوپ

[1] برگرفته از نویسنده [Hoo96] با کسب اجازه. هوکر الگوهایی را برای این اصول در صفحه زیر تعریف می‌کند:
http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment

اصل سوم: حفظ چشم‌انداز (vision)

برای موفقیت یک پروژه‌ی نرم‌افزاری، چشم‌اندازی روشن، ضروری است و بدون آن پروژه تقریباً همواره به جایی می‌رسد که دو یا چند ایده بر آن حاکم خواهد شد. یک سیستم بدون یکپارچگی مفهومی، به مجموعه‌ی ناجوری از طراحی‌های ناسازگار تبدیل می‌شود که به یکدیگر وصله-پینه شده باشند... مسامحه در خصوص چشم انداز معماری یک سیستم نرم‌افزاری باعث تضعیف سیستمی با طراحی خوب و سرانجام از کار افتادن آن می‌شود. داشتن یک معمار خوب که بتواند چشم انداز را حفظ کند و همخوانی را تقویت نماید، به حصول اطمینان از یک پروژه‌ی نرم‌افزاری بسیار موفق کمک خواهد کرد.

اصل چهارم: آنچه که شما تولید می‌کنید، دیگران مصرف می‌کنند

به‌ندرت پیش می‌آید که یک سیستم نرم‌افزاری پرقدرت صنعتی در محیطی خالی ساخته و به‌کار گرفته شود. به نحوی از انحا، یک شخص دیگر از سیستم شما استفاده و آن را مستندسازی و نگهداری خواهد کرد، و در غیر این صورت، برای اینکه قادر به شناخت سیستم شما باشد، باید به شما وابسته باشد. بنابراین، همواره تعیین مشخصات، طراحی و پیاده‌سازی را طوری انجام دهید که دیگران نیز قادر به درک کار شما باشند. تعداد مخاطبان هر محصول توسعه‌ی نرم‌افزار بالقوه زیاد است. تعیین مشخصات را با در نظر گرفتن کاربران انجام دهید. طراحی را با مدنظر قرار دادن پیاده‌سازی انجام دهید. هنگام کدنویسی، آنها را در نظر داشته باشید که باید سیستم را نگهداری کنند و آن را گسترش دهند. ممکن است شخصی بخواهد کدی را که شما نوشته اید، اشکال زدایی کند و این باعث می‌شود که بتواند از کدهای شما استفاده کند. آسان‌تر کردن کار آنها به ارزش سیستم می‌افزاید.

نکته‌ی کلیدی

اگر نرم‌افزاری ارزش داشته باشد، طی دوره حیات مفید خود تغییر خواهد کرد. به همین دلیل، نرم‌افزار باید طوری ساخته شود که قابل نگهداری باشد.

اصل پنجم: آینده‌نگری

سیستمی با طول عمر بالا از ارزش بیشتری برخوردار است. در محیط‌های کامپیوتری امروزی، که مشخصات به‌طور لحظه‌ای تغییر می‌کنند و تنها با گذشت چند ماه، سکوهای سخت افزاری، کهنه و قدیمی به شمار می‌روند، طول عمر نرم‌افزارها به جای سال بر حسب ماه سنجیده می‌شود. ولی، سیستم‌های «صنعتی پرقدرت» باید بیش از اینها دوام بیاورند. برای انجام موفقیت آمیز این کار، سیستم‌ها باید آمادگی انطباق بر این تغییرات و سایر تغییرات را داشته باشند. سیستم‌هایی که این ویژگی را با موفقیت ارائه می‌دهند، از ابتدا با این ویژگی طراحی می‌شوند. *هرگز طوری طراحی نکنید که خود را گرفتار کنید.* همواره از خود بپرسید، «اگر فلان مورد پیش آید، چه خواهد شد؟» و با ایجاد سیستمی که مسأله‌ی کلی را و نه فقط موردی خاص را حل می‌کند، خود را برای همه‌ی پاسخ‌های ممکن آماده کنید[1]. به این ترتیب، امکان استفاده‌ی مجدد از کل سیستم بسیار بالا خواهد بود.

اصل ششم: برنامه‌ریزی پیشاپیش برای استفاده‌ی مجدد

استفاده‌ی مجدد باعث صرفه جویی در زمان و کار می‌شود[2]. می‌توان استدلال کرد که دست یافتن

[1] این اندرز اگر بیش از حد به کار بسته شود می‌تواند خطرناک باشد. طراحی برای «مسأله کلی» گاهی مستلزم فداکردن کارایی است و باعث می‌شود راهکارهای ویژه ناکارآمد شوند.

[2] گرچه این برای کسانی که از نرم‌افزار برای پروژه‌های آتی استفاده‌ی مجدد می‌کنند، صحت دارد، استفاده‌ی مجدد ممکن است برای آنها که باید قطعات قابل استفاده‌ی مجدد را طراحی کنند و بسازند، هزینه بردارد. مطالعات نشان می‌دهد که طراحی و ساخت مؤلفه‌های قابل استفاده‌ی مجدد، ممکن است بین ۲۵ تا ۲۰۰ درصد بیشتر از نرم‌افزار هدف هزینه بردارد. در برخی موارد، اختلاف هزینه‌ها توجیه‌بردار نیست.

به سطح بالایی از قابلیت استفاده‌ی مجدد، سخت ترین هدف در رسیدن به یک سیستم نرم‌افزاری است. استفاده‌ی مجدد از کدها و طراحی‌ها به‌عنوان مزیت اصلی فن‌آوری‌های شیءگرا مطرح شده است. ولی عایدی این سرمایه گذاری به‌صورت خودکار به دست نمی‌آید. امکان استفاده‌ی مجددی که برنامه‌نویسی شیءگرا (یا سنتی) فراهم می‌آورد، نیاز به برنامه‌ریزی قبلی دارد. برای تحقق بخشیدن به استفاده‌ی مجدد در هر سطح از فرایند توسعه‌ی سیستم، تکنیک‌های بسیاری وجود دارد... برنامه‌ریزی پیشاپیش برای استفاده‌ی مجدد باعث کاهش هزینه‌ها و افزایش ارزش قطعات قابل استفاده‌ی مجدد و نیز سیستمی می‌شود که این قطعات در آن به‌کار برده می‌شوند.

اصل هفتم: تفکر!

این آخرین اصل، احتمالاً بیش از بقیه مورد بی مهری قرار می‌گیرد. تعقل و تفکر کامل و روشن قبل از اقدام به عمل، همواره نتایج بهتری به بار می‌آورد. هنگامی که درباره چیزی فکر کنید، احتمال اینکه آن را درست انجام دهید، بیشتر می‌شود. به علاوه، درباره انجام دوباره‌ی آن اطلاعاتی به دست خواهید آورد. اگر درباره چیزی فکر کنید و هنوز آن را اشتباه انجام دهید، این به یک تجربه با ارزش تبدیل می‌شود. یک اثر جانبی تفکر، این است که پی می‌برید هنگامی که چیزی را نمی‌دانید، در کدام نقطه می‌توانید در جستجوی پاسخ باشید. با تفکر روشن درباره سیستم، ارزش آن بالا می‌رود. به‌کارگیری شش اصل نخست نیاز به تفکر عمیق دارد و در این صورت، فایده بسیاری از آن عاید خواهد شد.

اگر هر مهندس نرم‌افزار و هر تیم نرم‌افزار از این هفت اصل هوکر پیروی کرده بود، بسیاری از مشکلاتی که در ساخت سیستم‌های کامپیوتری پیچیده تجربه می‌کنیم، برطرف می‌شدند.

۶-۱ پندارهای باطل نرم‌افزاری

ریشه‌ی پندارهای باطل نرم‌افزاری – باورهای نادرستی درباره نرم‌افزارها و **فرایند** به‌کاررفته در ساخت آنها- را می‌توان تا اولین روزهای کار با کامپیوتر دنبال نمود. پندارهای باطل نرم‌افزاری دارای چند ویژگی هستند که آنها را زیانبار ساخته‌اند؛ برای نمونه، به ظاهر، بیانی منطقی از واقعیتها بوده‌اند (گاه چند عنصر واقعی در آنها وجود دارد) ولی دارای احساسی نبوغ آمیز بوده غالباً توسط برنامه‌نویسان کارآزموده‌ای که «قدر می‌دانند» به آگاهی عموم می‌رسند.

امروزه اکثر مهندسان نرم‌افزار حرفه‌ای و باهوش با پندارهای باطل در زمینه‌ی نرم‌افزار آشنایی دارند-می‌دانند که این پندارها باعث گمراهی مدیران و دست اندرکاران می‌شوند و مشکلاتی جدی را به بار می‌آورند. ولی اصلاح نگرش‌ها و عادت‌های کهنه دشوار است و هنوز بقایایی از این پندارهای باطل برجای مانده‌اند.

پندارهای باطل مدیریتی. مدیرانی که مسؤولیت نرم‌افزاری دارند، همانند مدیران دیگر، غالباً تحت فشار کاهش هزینه‌ها، جلوگیری از بی‌برنامه‌گی و بهبود بخشیدن به کیفیت هستند. مدیر نرم‌افزاری همانند غریقی که به هر چیزی دست می‌اندازد، غالباً به پندارهای باطل نرم‌افزاری اعتقاد پیدا می‌کند، اگر بداند این اعتقاد باعث کاهش فشار می‌شود (حتی به‌طور موقت).

پندار باطل: *ما از قبل کسابی داریم که آکنده از استانداردها و روال‌های لازم برای ساختن نرم‌افزارهاست. آیا این کتاب آنچه را که افراد من باید بدانند در اختیارشان قرار نخواهد داد؟*

«صنعت جدیدی مثل نرم‌افزار، در غیاب استانداردهای مناسب، ناگزیر از اتکا به عرف خواهد بود.»

تام دومارکو

مرجع وب

شبکه مدیریت پروژه‌های نرم‌افزاری در www.spmn.com می‌تواند شما را در شناسایی و دور-ساختن این پندارهای باطل و موارد دیگر یاری دهد.

واقعیت: ممکن است کتاب استانداردهای خیلی خوبی وجود داشته باشد، ولی آیا از آن استفاده می‌شود؟ آیا سازندگان نرم‌افزار از وجود آن آگاهند؟ آیا مهندسی نرم‌افزار نوین را ارائه می‌دهد؟ آیا کامل است؟ آیا آنقدر روان هست که زمان تحویل را بهبود بخشد و در عین حال کیفیت را حفظ کند؟ در بسیاری از موارد، پاسخ اکثر این پرسش‌ها «خیر» است.

پندار باطل: *اگر از برنامه عقب بیفتیم، می‌توانیم بر تعداد برنامه‌نویسان بیفزاییم و عقب‌افتادگی را جبران کنیم (این وضعیت را گاه «یورش مغولی» می‌گویند).*

واقعیت: ایجاد نرم‌افزار، یک فرایند مکانیکی نظیر ساخت تولیدات معمولی نیست. به قول بروکز [Bro95]: «... با افزودن افراد دست‌اندرکار به نرم‌افزاری که تأخیر دارد، بر میزان تأخیر آن افزوده خواهد شد». در نگاه نخست ممکن است این گفته خلاف منطق به نظر برسد، ولی با از راه رسیدن افراد جدید، افراد قدیمی باید زمانی را صرف آموزش آنها کنند و درنتیجه زمانی که باید صرف کار روی نرم‌افزار شود، هدر می‌رود. اضافه‌کردن افراد، عملی است ولی به شیوه‌ای هماهنگ و با برنامه‌ریزی منظم.

پندار باطل: *اگر تصمیم به «برون‌سپاری»[1] یک پروژه‌ی نرم‌افزاری به شرکت دیگری بگیرم، می‌توانم خودم را آسوده سازم و بگذارم تا آن شرکت آن را بسازد.*

واقعیت: اگر سازمانی نداند که چگونه پروژه‌های نرم‌افزاری را از نظر داخلی مدیریت و کنترل کند، هنگامی که پروژه‌های نرم‌افزاری را برون سپاری کند، خود را به تقلایی بیهوده انداخته است.

اندرز

پیش از شروع به کار بکوشید که دریابید چه باید بکنید. ممکن است قادر به بسط و توسعه‌ی همه جزئیات نباشید ولی هرچه بیشتر بدانید، خطر کمتری در کمین شماست.

پندارهای باطل مشتریان. مشتری‌ای که درخواست یک نرم‌افزار کامپیوتری دارد، ممکن است پشت میز کناری باشد، یک گروه تکنیکی در آن سوی سالن باشد، بخش فروش و بازاریابی باشد، یا یک شرکت دیگر باشد که قراردادی برای نرم‌افزار منعقد نموده است. در بسیاری موارد، مشتری به پندار باطل‌هایی درباره نرم‌افزارها اعتقاد دارد، زیرا مدیران نرم‌افزار و سازندگان آن کمتر سعی در برطرف‌کردن سوءتفاهم‌ها دارند. این پندار باطل منجر به انتظارات نادرست (از جانب مشتری) و درنهایت عدم رضایت از سازنده می‌شود.

پندار باطل: *بیانی کلی از اهداف، برای شروع به نوشتن برنامه‌ها کفایت می‌کند—جزئیات را بعداً می‌توانیم پر کنیم.*

واقعیت: گرچه بیانی جامع و پایدار از خواسته‌ها همواره امکان‌پذیر نیست، «بیان مبهم اهداف» دستورالعملی برای مصیبت است. خواسته‌های نامبهم (که معمولاً با تکرار به دست می‌آیند) تنها از طریق برقراری ارتباط پیوسته و اثربخش میان مشتری و سازنده شکل می‌گیرند.

[1] outsourcing

پندار باطل: *نیازهای پروژه پیوسته در حال تغییر است، ولی این تغییرات را به‌راحتی می‌توان در نرم‌افزار جای داد زیرا نرم‌افزار انعطاف‌پذیر است.*

واقعیت: این درست است که نیازمندی‌های نرم‌افزار تغییر می‌کند، ولی تأثیر تغییر به زمان اِعمال تغییر بستگی دارد. اگر به تعریف صریح توجه جدی شود، درخواست‌های اولیه برای تغییر را به‌راحتی می‌توان پاسخ گفت. مشتری می‌تواند نیازمندی‌ها را مرور کند و اصلاحاتی را با تأثیر نسبتاً کم بر هزینه‌ها توصیه کند.[1] ولی با گذر زمان، هزینه‌ها به سرعت بالا می‌رود. منابع مصرف شده‌اند و یک چارچوب طراحی مشخص شده است. تغییر می‌تواند باعث تغییرات مشکل آفرینی شود که نیاز به منابع اضافی و اصلاح اساسی طراحی دارد.

اندرز

هرگاه به این فکر افتادید که برای مهندسی نرم‌افزار وقت ندارید، از خود بپرسید که آیا برای انجام دوباره‌ی آن وقت دارید یا خیر.

پندارهای باطل سازندگان. پندارهای باطلی که نرم‌افزارنویسان به آنها باور دارند، نتیجه‌ی ۵۰ سال فرهنگ برنامه‌نویسی است. در نخستین دهه‌های ساخت نرم‌افزار، برنامه‌نویسی شکلی از هنر پنداشته می‌شد. سنت‌های قدیمی دیر از بین می‌روند.

پندار باطل: *هنگامی که برنامه را نوشتیم و برنامه کار کرد، دیگر کار تمام است.*

واقعیت: یک بار کسی گفته بود: «هر چه زودتر دست به کار نوشتن دستورهای برنامه شوید، زمان بیشتری صرف به پایان بردن آن خواهید کرد». داده‌های صنعتی نشان می‌دهد که بین ۶۰٪ تا ۸۰٪ از همه‌ی کوشش‌های صرف شده روی نرم‌افزارها، پس از نخستین بار تحویل آنها به مشتری صورت می‌پذیرد.

پندار باطل: *تا هنگامی که برنامه را «اجرا» نکرده‌ام، راهی برای ارزیابی کیفیت آن ندارم.*

واقعیت: یکی از مؤثرترین راهکارهای تضمین کیفیت نرم‌افزار از زمان آغاز پروژه قابل اجراست – یعنی مرور تکنیکی. مرور نرم‌افزار (فصل ۱۵) یک فیلتر کیفیتی است که از آزمایش نرم‌افزار برای یافتن گروههای معینی از معایب نرم‌افزاری مؤثرتر است.

پندار باطل: *تنها چیز قابل تحویل برای یک پروژه موفق، برنامه‌ای است که کار کند.*

واقعیت: یک برنامه‌ی کاری، تنها بخشی از پیکربندی نرم‌افزاری است که شامل عناصر فراوان می‌شود. انواع محصولات کاری (از قبیل مدل‌ها، مستندات، طرح‌ها) بستری برای مهندسی موفق و مهمتر از آن، راهنمایی برای پشتیبانی نرم‌افزار، فراهم می‌آورند.

پندار باطل: *مهندسی نرم‌افزار، ما را وادار می‌سازد که مستندات حجیم و بیهوده تهیه کنیم و از سرعت کار ما می‌کاهد.*

[1] بسیاری از مهندسان نرم‌افزار به روش «چابک» روی آورده‌اند که به تغییرات به‌طور تدریجی پاسخ می‌دهد و لذا کنترل تأثیرات و هزینه‌ها نیز تدریجی است. روش‌های چابک در فصل ۳ بحث خواهند شد.

SafeHome

چگونگی آغاز یک پروژه

صحنه: اتاق کنفرانس در شرکت CPI، یک شرکت (خیالی) که محصولات مصرفی برای استفاده‌ی خانگی و تجاری می‌سازد.

نقش آفرینان: مال گولدن، مدیر ارشد، توسعه‌ی محصول؛ لیزا پرز، مدیر بازرگانی؛ لی وارن، مدیر مهندسی؛ جو کامالری، معاون مدیر اجرایی، توسعه‌ی تجاری.

مکالمه

جو: خب، لی، شنیده‌ام افرادت در حال ساخت یک جعبه بی‌سیم جهانی هستند.

لی: چیز خیلی خوبی است. به اندازه یه قوطی کبریت کوچک... می‌توانیم آن را به همه جور حس‌گری وصل کنیم، یا حتی به دوربین‌های دیجیتال؛ تقریباً به هرچیزی. با استفاده از پروتوکل بی‌سیم g802.11 می‌توانیم بدون سیم به خروجی دستگاه‌ها دستیابی داشته باشیم. ما فکر می‌کنیم این یه یک نسل کاملاً جدید از محصولات ختم می‌شود.

جو: مال؟ تو موافقی؟

مال: بله موافقم. در واقع با فروش بدون رشدی که امسال داشتیم، به یک چیز جدید احتیاج داریم. من و لیزا یک مقدار تحقیق در بازار انجام داده‌ایم و فکر می‌کنم به خط جدیدی از محصولات دست پیدا کردیم که می‌تواند عالی باشد.

جو: چقدر عالی...

مال (درحالی که از تعهد مستقیم شانه خالی می‌کند): ایده‌ات را برایش بگو لیزا.

لیزا: این یک نسل کاملاً جدید است که ما به آن می‌گوییم «محصولات مدیریت خانگی» اسم آن را گذاشته‌ایم «SafeHome». این محصولات از یک رابط بی‌سیم استفاده می‌کنند و سیستمی را در اختیار خانه‌دارها و مالکان شرکت‌های کوچک قرار می‌دهند که توسط کامپیوتر شخصی آنها کنترل می‌شود- امنیت منزل، پایش منزل، کنترل لوازم خانگی- مثلاً روشن کردن کولر منزل در راه رسیدن به خانه- و خلاصه از این چیزها.

لی (حرف لیزا را قطع می‌کند): بخش مهندسی، مطالعه امکان‌سنجی را انجام داده. جو، با هزینه ساخت پایین، شدنی است. بیشتر سخت افزار را می‌توان آماده تهیه کرد. مشکل، نرم‌افزار است ولی چیزی نیست که از پس آن برنیاییم.

جو: جالب است.

مال: کامپیوترهای شخصی در بیش از ۷۰٪ از منازل ایالات متحده نفوذ کرده‌اند. اگر بتوانیم این محصول را درست قیمت‌گذاری کنیم، می‌تواند غوغا کند. هیچ شرکت دیگری این جعبه بی‌سیم ما را ندارد... یک محصول انحصاری. در رقابت با شرکت‌های دیگر یک پرش دو ساله خواهیم داشت و از نظر درآمدی، ۳۰ تا ۴۰ میلیون دلار در سال دوم خواهیم داشت.

واقعیت: مهندسی نرم‌افزار، مستندسازی نیست بلکه به ایجاد محصولی با کیفیت مربوط می‌شود. کیفیت بهتر، به دوباره کاری کمتر می‌انجامد. کاهش دوباره کاری به تحویل سریع تر محصول می‌انجامد.

بسیاری از حرفه‌ای‌های نرم‌افزار به اشتباه بودن پندارهای باطل بالا واقفند. متأسفانه برداشتها و روش‌های مرسوم باعث ضعف مدیریت و عملکرد تکنیکی می‌شود، حتی هنگامی که واقعیت، روش بهتری را حکم می‌کند. شناخت واقعیتهای نرم‌افزار، نخستین گام در جهت فرمول‌بندی راهکارهای علمی برای ساخت نرم‌افزار است.

۱-۷ شروع به‌کار

هر پروژه‌ی نرم‌افزاری با یک نیاز تجاری شروع می‌شود-نیاز به تصحیح یک نقص در برنامه‌ی کاربردی موجود؛ نیاز به تطبیق یک «سیستم قدیمی» بر یک محیط تجاری در حال تغییر؛ نیاز به توسعه‌ی قابلیت‌های عملیاتی و ویژگی‌های یک برنامه کاربردی؛ یا حتی نیاز به ایجاد یک محصول، سرویس یا سیستمی جدید.

در شروع یک پروژه‌ی نرم‌افزاری، نیاز تجاری غالباً به‌طور غیر رسمی به‌عنوان بخشی از یک مکالمه ساده بیان می‌شود. نمونه‌ای از این مکالمه در کادر صفحه‌ی قبل آورده شده است.

به استثنای یک ارجاع گذرا، از نرم‌افزار به ندرت در این مکالمه ذکری به میان آمد و با این حال، نرم‌افزار است که باعث ایجاد خط تولید محصول «*SafeHome*» یا به شکست انجامیدن آن می‌شود. تلاش مهندسی در صورتی موفق خواهد شد که نرم‌افزار *SafeHome* موفق شود. بازار در صورتی این محصول را خواهد پذیرفت که نرم‌افزار تعبیه‌شده در آن به طرزی مناسب، نیازهای مشتری (که هنوز بیان نشده است) برآورده سازد. در بسیاری از فصول آیند، پیشرفت مهندسی نرم‌افزار را در خصوص این محصول دنبال خواهیم کرد.

۱-۸ خلاصه

نرم‌افزار، عنصر کلیدی در تکامل محصولات و سیستم‌های کامپیوتری و یکی از مهمترین فن‌آوری‌ها در دنیا به شمار می‌رود. طی ۵۰ سال اخیر، نرم‌افزارها از یک ابزار تخصصی حل مسأله و تحلیل اطلاعات، خود به صنعتی جداگانه تکامل یافته‌اند. با این همه، هنوز در توسعه‌ی سروقت نرم‌افزار با بودجه‌ی تعیین شده مشکل داریم.

نرم‌افزار-که شامل برنامه‌ها، داده‌ها و اطلاعات می‌شود- مجموعه‌ای گسترده از کاربردها و فن‌آوری‌ها را در بر می‌گیرد. نرم‌افزارهای قدیمی همچنان چالش‌های خاصی را فرا روی کسانی قرار می‌دهند که باید از آنها نگهداری کنند.

سیستم‌ها و برنامه‌های مبتنی بر وب، از مجموعه‌های ساده‌ای از محتوای اطلاعات، به سیستم‌هایی پیچیده تکامل پیدا کرده‌اند که قابلیت‌های عملیاتی پیچیده و محتویات چند رسانه‌ای را ارائه می‌کنند. گرچه این برنامه‌های تحت وب دارای ویژگی‌ها و خواسته‌های منحصر به فردند، باز هم نرم‌افزارند.

مهندسی نرم‌افزار شامل فرایند، روش‌ها و ابزارهایی می‌شود که به کمک آن می‌توان سیستم‌های کامپیوتری پیچیده را با زمان‌بندی ساده و با کیفیت، ایجاد کرد. فرایند نرم‌افزار شامل پنج فعالیت چارچوبی می‌شود-ارتباطات، برنامه ریزی، مدل‌سازی، ساخت و استقرار-که در کلیه پروژه‌های نرم‌افزاری قابل به‌کارگیری‌اند. مهندسی نرم‌افزار در عمل یک فعالیت حل مسأله است که از مجموعه‌ای از اصول بنیادی پیروی می‌کند.

مجموعه‌ی گسترده‌ای از پندارهای باطل نرم‌افزاری همچنان باعث گمراهی مدیران و دست اندرکاران می‌شود، هرچند که دانش کلی ما از نرم‌افزار و فن‌آوری‌های لازم برای ساخت آن رشد کرده است. با آموختن مطالب بیشتر درباره مهندسی نرم‌افزار، رفته رفته خواهید دانست که چرا این پندارهای باطل را به محض مواجهه باید به کناری نهاد.

مسائل و نکاتی برای تعمق

۱-۱ دست کم پنج مثال ارائه دهید که چگونگی به‌کارگیری قانون پیامدهای ناخواسته را در نرم‌افزارهای کامپیوتری نشان دهد.

۱-۲ چند مثال (مثبت و منفی) بیاورید که نشان‌گر تأثیر نرم‌افزار بر جامعه ما باشد.

۱-۳ پاسخ‌هایی را که به پنج پرسش مطرح شده در آغاز بخش ۱-۱ دادید، توسعه دهید. آنها را با هم کلاسی‌های خود به بحث بگذارید.

۱-۴ بسیاری از برنامه‌های کاربردی مدرن به‌وفور تغییر می‌کنند-پیش از آنکه به‌کاربر نهایی ارائه شوند و سپس بعد از به‌کارگرفته شدن نخستین نسخه‌ی آنها. چند شیوه برای ساخت نرم‌افزار به منظور متوقف کردن تباه شدگی ناشی از تغییر پیشنهاد کنید.

۱-۵ هفت گروه نرم‌افزار ارائه شده در بخش ۱-۲-۱ را در نظر بگیرید. آیا تصور می‌کنید برای همه‌ی این گروهها روش یکسانی از مهندسی نرم‌افزار را می‌توان به‌کار برد؟

۱-۶ در شکل ۱-۳، سه لایه مهندسی نرم‌افزار روی لایه‌ای با عنوان «تأکید بر کیفیت» قرار داده می‌شوند. این به معنای یک برنامه کیفیتی سازمانی نظیر مدیریت کیفیت فراگیر است. درباره برنامه مدیریت کیفیت فراگیر قدری تحقیق کنید و اصول کلیدی آن را مطرح نمایید.

۱-۷ آیا مهندسی نرم‌افزار به هنگام ایجاد برنامه‌های تحت وب نیز قابل اعمال است؟ در صورت مثبت بودن پاسخ، چگونه می‌توان آن را اصلاح کرد تا خصوصیات منحصر به فرد برنامه‌های تحت وب را در برگیرد؟

۱-۸ با فراگیرتر شدن نرم‌افزارها، خطراتی که عموم را تهدید می‌کند (به دلیل خطا در کار برنامه‌ها) به یک نگرانی فزاینده تبدیل می‌شود. یک سناریوی فاجعه بار (ولی واقع‌بینانه) بنویسید که در آن، خطا در کار یک برنامه‌ی نرم‌افزاری به ضرری هنگفت (مالی یا جانی) بینجامد.

۱-۹ یک فعالیت چارچوبی را به زبان ساده شرح دهید. هنگامی که می‌گوییم فعالیت‌های چارچوبی برای همه‌ی پروژه‌ها قابل به‌کارگیری هستند، آیا این بدان معناست که وظایف کاری یکسانی برای همه‌ی پروژه‌ها، صرف نظر از اندازه و پیچیدگی آنها، قابل استفاده‌اند؟ توضیح دهید.

۱-۱۰ فعالیت‌های چتری در سرتاسر فرایند نرم‌افزار رخ می‌دهند. آیا فکر می‌کنید که این فعالیت‌ها به‌طور یکنواخت در سرتاسر فرایند به‌کار گرفته می‌شوند یا اینکه برخی در یک یا چند فعالیت چارچوبی متمرکز شده‌اند؟

۱-۱۱ به پندارهای باطل فهرست‌شده در بخش ۱-۶ دو مورد اضافه کنید؛ واقعیت مربوط به هر یک را نیز ذکر نمایید.

بخش نخست

فرایند نرم‌افزار

در این بخش از کتاب، درباره‌ی فرایندی که بر مهندسی نرم‌افزار یک چارچوب فراهم می‌آورد، مطالبی خواهید آموخت.

در فصل‌های آینده به این پرسش‌ها خواهیم پرداخت:

- فرایند نرم‌افزار چیست؟
- فعالیت‌های چارچوبی کلی موجود در هر فرایند نرم‌افزار کدام‌اند؟
- فرایندها چگونه مدل‌سازی می‌شوند و الگوهای فرایند چیستند؟
- مدل‌های فرایند تجویزی چیستند و نقاط قوت و ضعف آن‌ها کدام‌اند؟
- چرا «چابکی» در کار مهندسی نرم‌افزار یک واژه‌ی کلیدی به‌شمار می‌رود؟
- توسعه‌ی نرم‌افزار «چابک» چیست و چه تفاوتی با مدل‌های فرایند سنتی دارد؟

هنگامی که به این پرسش‌ها پاسخ گفته شد، بهتر آماده خواهید شد تا حیطه‌ای را که مهندسی نرم‌افزار در آن به‌کارگرفته می‌شود، بشناسید.

فصل ۲

مدل‌های فرایند

نگاهی گذرا

فرایند چیست؟ وقتی کار می‌کنید تا یک سیستم یا یک محصول بسازید، حتماً باید یک سری مراحل قابل پیش‌بینی را چک کنید: یک نقشه راه که در ایجاد نتیجه‌ای با کیفیت بالا و به موقع شما را یاری می‌کند. این نقشه که آن را دنبال می‌کنید، «فرایند نرم‌افزار» نام دارد.

چه کسی آن را انجام می‌دهد؟ مهندسان نرم‌افزار و مدیران آنها، فرایند را با نیازهای خود مطابقت داده سپس آن را دنبال می‌کنند. به‌علاوه، کسانی که نرم‌افزار را درخواست کرده‌اند، در فرایند نرم‌افزار نقش دارند.

چرا اهمیت دارد؟ زیرا باعث ثبات، کنترل و سازمان‌دهی فعالیتی می‌شود که اگر به‌حال خود گذاشته شود ممکن است باعث آشوب شود. ولی یک رویکرد نوین در مهندسی نرم‌افزار باید «چابک[1]» باشد. تنها باید آن دسته از فعالیت‌ها، کنترل‌ها و محصولات کاری را طلب کند که مناسب تیم پروژه و محصولی باشد که قرار است تولید شود.

چه مراحلی دارد؟ در سطح مشروح، فرایندی که برمی‌گزینید، به نرم‌افزاری که می‌خواهید بسازید، بستگی دارد. یک فرایند ممکن است برای ایجاد نرم‌افزار مربوط به سیستم هوا-فضای یک هواپیما مناسب باشد، حال آنکه برای ایجاد یک وب‌سایت ممکن است فرآیندی کاملاً متفاوت موردنیاز باشد.

حاصل کار چیست؟ از دیدگاه مهندس نرم‌افزار، حاصل کار، برنامه‌ها، داده‌ها و مستنداتی است که به‌عنوان نتیجه‌ای از فعالیت‌های مهندسی نرم‌افزارِ مشخص‌شده توسط فرایند، تولید می‌شوند.

چطور مطمئن شوم که درست از عهده‌ی کار برآمده‌ام؟ چند راهکار ارزیابی فرایند نرم‌افزار وجود دارد که سازمان‌ها را قادر به تعیین «بلوغ» فرایند نرم‌افزار می‌سازد. ولی کیفیت، به‌موقع بودن و کارایی درازمدتِ محصولی که ساخته‌اید، بهترین ملاک‌ها برای بازدهی فرایند مورد استفاده شما هستند.

---

[1] agile

هاورد باتیر [Bae98] در کتاب فوق‌العاده‌ی خویش، دیدی اقتصادی از نرم‌افزار و مهندسی نرم‌افزار ارائه می‌دهد. او در خصوص فرایند نرم‌افزار می‌گوید:

> چون نرم‌افزار همانند همه‌ی سرمایه‌ها، تجسم آگاهی و دانش است و چون این آگاهی و دانش در ابتدا بسیار پراکنده، بالقوه و تلویحی است، توسعه‌ی نرم‌افزار یک فرایند یادگیری اجتماعی است. فرایند، گفتگویی است که در آن دانشی که باید به نرم‌افزار تبدیل گردد، جمع‌آوری می‌شود و به‌صورت نرم‌افزار تجسم می‌یابد. فرایند، ارتباط متقابل میان طراحان و کاربران، بین کاربران و ابزار در حال تکامل، و بین طراحان و ابزارها (فن‌آوری) فراهم می‌آورد. این یک فرایند تکراری است که در آن با هر دور جدید از گفتگو که مطالب بیشتری از افراد مربوط به‌دست می‌آید، ابزار در حال تکامل، خود به‌عنوان رسانه‌ای برای برقراری ارتباط عمل می‌کند.

درحقیقت، ساختن نرم‌افزارهای کامپیوتری، یک فرایند یادگیری تکراری است و نتیجه، یا به قول باتیر «سرمایه نرم‌افزاری»، تجسم اطلاعات و آگاهی جمع‌آوری شده، تلخیص شده و سازمان‌دهی‌شده در اثنای اجرای فرایند است.

ولی یک فرایند نرم‌افزار از دیدگاه فنی دقیقاً چیست؟ در این کتاب، فرایند نرم‌افزار را به‌عنوان چارچوبی برای اعمال موردنیاز جهت ساخت نرم‌افزاری با کیفیت بالا تعریف می‌کنیم. آیا «فرایند» مترادف با مهندسی نرم‌افزار است؟ پاسخ این است: «بلی» و «خیر». فرایند نرم‌افزار روش مهندسی را مشخص می‌کند. ولی مهندسی نرم‌افزار شامل فن‌آوری‌هایی که فرایند را تشکیل می‌دهند - روش‌های فنی و ابزارهای خودکار - نیز می‌شود.

مهم‌تر اینکه، مهندسی نرم‌افزار توسط افرادی خلاق و آگاه انجام می‌شود و باید در چارچوب یک فرایند نرم‌افزاری مشخص کار کنند که مناسب محصولاتِ ساخته‌ی دست آنها برده بازار خاص خود را طلب کند.

۱-۲ یک مدل فرایند کلی

در فصل ۱، فرایند به‌عنوان مجموعه‌ای از فعالیت‌های کاری، کنش‌ها و وظایف تعریف شد که هنگام ایجاد یک محصول باید اجرا شوند. هر کدام از این فعالیت‌ها، کنش‌ها و وظایف در یک چارچوب یا مدل قرار دارند که رابطه‌ی آنها را با فرایند و با یکدیگر تعریف می‌کند.

طرحی از فرایند نرم‌افزار در شکل ۱-۲ نشان داده شده است. با رجوع به شکل، هر فعالیت چارچوبی حاوی مجموعه‌ای از کنش‌های مهندسی نرم‌افزار است. هر کنش مهندسی نرم‌افزار به‌وسیله مجموعه‌ای از وظایف تعیین می‌شود که مشخص می‌کند به چه وظایفی باید عمل شود، چه محصولاتی باید تولید شوند، به چه نقاط تضمین کیفیتی نیاز است و چه نقاط عطفی برای نشان دادن پیشرفت فرایند به‌کار گرفته خواهد شد.

همان طور که در فصل ۱ بحث شد، چارچوب فرایند کلی برای مهندسی نرم‌افزار، پنج فعالیت چارچوبی-**ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار**- را تعریف می‌کند. به‌علاوه، مجموعه‌ای از فعالیت‌های چتری-کنترل و پیگیری پروژه، مدیریت خطر (ریسک)، تضمین کیفیت، مدیریت پیکربندی، بازبینی‌های فنی و غیره- در سرتاسر پروژه به‌کار گرفته می‌شوند.

نکته‌ی کلیدی

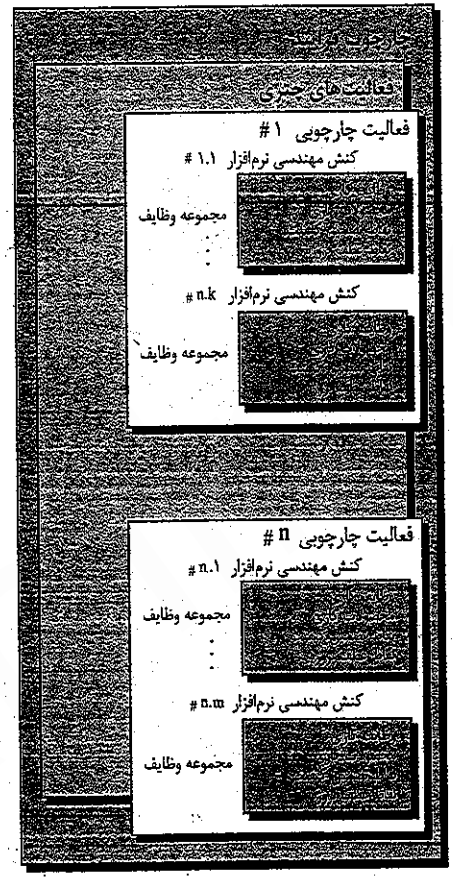
سلسله مراتب کارهای فنی در فرایند نرم‌افزار به‌صورت فعالیت‌هایی است که شامل کنش‌ها می‌شوند و هر کنش خود شامل چند وظیفه می‌شود.

باید توجه داشت که یک جنبه‌ی مهم از فرایند نرم‌افزار هنوز بحث نشده است. این جنبه-که جریان فرایند نامیده می‌شود- شرح می‌دهد که فعالیت‌های چتری و کنش‌ها و وظایفی که در داخل هر فعالیت چارچوبی رخ می‌دهند از نظر ترتیب زمانی چگونه سازمان‌دهی می‌شوند (شکل۲-۲).

در یک **جریان فرایند خطی**، هر کدام از پنج فعالیت چارچوبی به‌ترتیب اجرا می‌شود، به‌طوری که با ارتباطات آغاز و به استقرار ختم می‌شود (شکل۲-۳الف). در یک جریان فرایند مبتنی بر تکرار، پیش از رفتن به دور تکرار بعدی، یک یا چند فعالیت تکرار می‌شود (شکل۲-۳ب). در جریان فرایند تکاملی، فعالیت‌ها به‌شیوه‌ای «حلقوی» اجرا می‌شوند. هر مدار از پنج فعالیت عبور می‌کند که به نسخه‌ی کامل‌تری از نرم‌افزار می‌انجامد (شکل۲-۳پ). در **جریان فرایند موازی** (شکل۲-۳ت) یک یا چند فعالیت به موازات سایر فعالیت‌ها انجام می‌شوند (مثلاً مدل‌سازی برای یک جنبه از نرم‌افزار ممکن است به موازات ساختار، جنبه‌ی دیگری از نرم‌افزار اجرا گردد.)

> «ما تصور می‌کنیم که سازندگان نرم‌افزار یک حقیقت حیاتی را فراموش کرده‌اند: اکثر سازمان‌ها نمی‌دانند که چه می‌کنند. آنها خیال می‌کنند که می‌دانند، ولی نمی‌دانند.»
>
> **تام دومارکو**

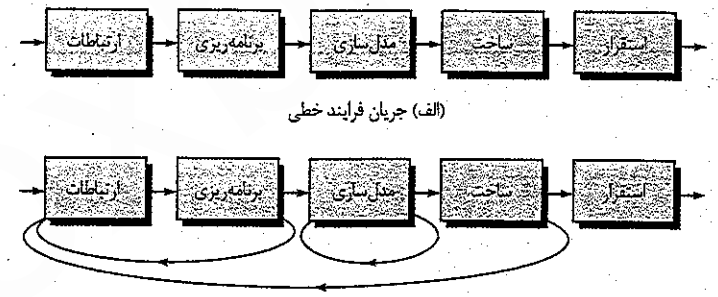
فرایند نرم‌افزار



شکل۱-۲ یک چارچوب فرایند نرم‌افزار

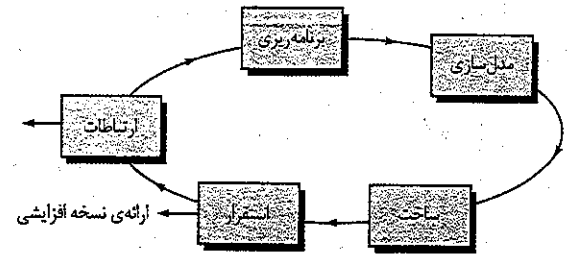
۲-۱-۱ تعریف یک فعالیت چارچوبی

گرچه پنج فعالیت چارچوبی را توصیف کردیم و تعریفی مقدماتی از هر کدام در فصل ۱ ارائه نمودیم، یک تیم نرم‌افزاری پیش از اینکه بتواند هر کدام از این فعالیت‌ها را به‌طور مناسب به‌عنوان بخشی از فرایند نرم‌افزار اجرا کند، باید اطلاعات به‌مراتب بیشتری در اختیار داشته باشد. پس یک پرسش کلیدی پیش رو دارید: اگر ماهیت مسأله، خصوصیات افرادی که کار را انجام می‌دهند و طرف‌های ذی‌نفعی که پروژه را پشتیبانی می‌کند، معلوم باشد، کدام کنش‌ها برای یک فعالیت چارچوبی مناسب خواهد بود؟

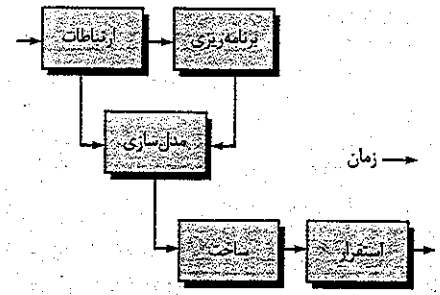


(الف) جریان فرایند خطی

(ب) جریان فرایند مبتنی بر تکرار



(پ) جریان فرایند تکاملی



(ت) جریان فرایند موازی

شکل ۲-۲ جریان فرایند.

برای یک پروژه‌ی نرم‌افزاری کوچک که یک نفر با خواسته‌های صریح و ساده (در مکانی دوردست) درخواست می‌کند، فعالیت برقراری ارتباط ممکن است شامل چیزی در حد یک تماس تلفنی با ذی‌نفع مورد نظر باشد. بنابراین، تنها کنش لازم، مکالمه تلفنی است و وظایف کاری (مجموعه وظایف) که این کنش شامل آنها می‌شود، عبارتند از:

برقراری تماس با فرد ذی‌نفع از طریق تلفن.

۱. بحث درباره خواسته‌ها و یادداشت برداشتن.

۲. سازمان‌دهی یادداشت‌ها در یک بیان مختصر از خواسته‌ها.

۳. ارسال ایمیل برای بازبینی و تصویب.

اگر پروژه به واسطه وجود تعداد زیادی از طرف‌های ذی‌نفع پیچیدگی بیشتری می‌یافت، که هر کدام مجموعه‌ای متفاوت از خواسته‌ها را دارند (که گاهی با یکدیگر در تضادند)، فعالیت ارتباطات ممکن است شش کنش متمایز داشته باشد (که در فصل ۵ توصیف شده‌اند): شروع (inception)، استخراج (elictation)، شناخت (elaboration)، مذاکره (negotiation)، تعیین مشخصات (specification) و اعتبارسنجی (validation). هر کدام از این کنش‌های مهندسی نرم‌افزار دارای چندین وظیفه‌ی کاری و تعدادی محصولات کاری متمایزند.

یک فعالیت چارچوبی چگونه با تغییر ماهیت پروژه تغییر می‌کند؟

۲-۱-۲ تعیین مجموعه وظایف

با رجوع دوباره به شکل ۲-۱، هر کنش مهندسی نرم‌افزار (مثلاً استخراج که کنشی مرتبط با فعالیت ارتباطات است) را می‌توان با تعدادی از مجموعه وظایف متفاوت نشان داد که هر کدام مجموعه‌ای از وظایف کاری در مهندسی نرم‌افزار مرتبط با محصولات کاری، نقاط تضمین کیفیت و نقاط عطف پروژه‌اند. باید مجموعه وظایفی را برگزینید که نیازهای پروژه را به بهترین نحو برآورده سازد و خصوصیات تیم شما را در بر گیرد. این بدان معناست که یک کنش مهندسی نرم‌افزار را می‌توان بر نیازهای خاص پروژه نرم‌افزار و خصوصیات تیم نرم‌افزاری تطبیق داد.

نکته‌ی کلیدی
پروژه‌های متفاوت، مجموعه وظایف متفاوتی را طلب می‌کند. تیم نرم‌افزاری، مجموعه وظایف را بر اساس خصوصیات پروژه و مسأله انتخاب می‌کند.

۲-۱-۳ الگوهای فرایند

هر تیم نرم‌افزاری به موازات پیشروی در فرایند نرم‌افزار با مشکلاتی مواجه می‌شود. اگر راهکارهای اثبات‌شده برای این مشکلات به‌راحتی در دسترس تیم قرار داشته باشند، به‌طوری که بتوان به این مشکلات پرداخت و آنها را به‌راحتی برطرف کرد، مفید خواهد بود. الگوی فرایند[1]، مشکلی مرتبط با فرایند را توصیف می‌کند که طی کار مهندسی نرم‌افزار با آن مواجه شده باشند، محیطی را که در آن مشکل مشاهده شده است، مشخص می‌کند و یک یا چند راهکار برای آن مشکل پیشنهاد می‌کند. الگوی فرایند، به بیان کلی‌تر، یک قالب[2] [Amb98]- روشی سازگار برای توصیف راهکارهای مسأله در حیطه‌ی فرایند نرم‌افزار- در اختیار شما قرار می‌دهد. تیم نرم‌افزاری با ترکیب‌کردن این الگوها می‌تواند مسائل را حل کند و فرایندی را بنا نهد که به بهترین وجه، نیازهای یک پروژه را برآورده سازد.

الگوی فرایند چیست؟

[1] در فصل ۱۲ بحث مشروحی درباره الگوها ارائه خواهد شد.

[2] template

اطلاعات

مجموعه وظایف

در یک مجموعه وظایف، کارهای واقعی که باید برای پیشبرد اهداف یک کنش مهندسی نرم‌افزار انجام شوند، تعیین می‌شود. برای مثال، استخراج، یا به عبارت عوامانه‌تر آن، «جمع‌آوری خواسته‌ها» یک کنش مهم در مهندسی نرم‌افزار است که طی فعالیت ارتباطات رخ می‌دهد. هدف از جمع‌آوری خواسته‌ها، آن است که بدانیم طرف‌های ذی‌نفع گوناگون، از نرم‌افزاری که قرار است ساخته شود، چه انتظاری دارند.

برای یک پروژه ساده و نسبتاً کوچک، مجموعه وظایف برای جمع‌آوری خواسته‌ها ممکن است به‌صورت زیر باشد:

۱. تهیه فهرستی از طرف‌های ذی‌نفع در پروژه.
۲. دعوت از همه‌ی طرف‌های ذی‌نفع برای یک جلسه غیررسمی.
۳. درخواست از طرف‌های ذی‌نفع برای تهیه فهرستی از ویژگی‌ها و قابلیت‌های مورد نیاز.
۴. بحث درباره خواسته‌ها و تهیه فهرست نهایی.
۵. اولویت‌بندی خواسته‌ها.
۶. ذکر حوزه‌های عدم قطعیت.

برای یک پروژه‌ی نرم‌افزاری بزرگتر و پیچیده‌تر، ممکن است به مجموعه وظایف متفاوتی نیاز باشد. این مجموعه می‌تواند شامل وظایف کاری زیر باشد:

۱. تهیه فهرستی از طرف‌های ذی‌نفع در پروژه.
۲. مصاحبه جداگانه با هر کدام از طرف‌های ذی‌نفع برای تعیین خواسته‌ها و نیازهای کلی.
۳. تهیه فهرستی مقدماتی از قابلیت‌ها و ویژگی‌ها براساس ورودی طرف‌های ذی‌نفع.
۴. زمان‌بندی برای یک سری جلسات برای تسهیل تعیین مشخصات برنامه‌های کاربردی.
۵. برگزاری جلسات.
۶. تولید سناریوهای کاربری غیررسمی به‌عنوان بخشی از هر جلسه.
۷. پالایش سناریوهای کاربری بر اساس بازخورد از طرف‌های ذی‌نفع.
۸. تهیه فهرست بازبینی‌شده‌ی خواسته‌های طرف‌های ذی‌نفع.
۹. استفاده از فنون استقرار عملیات کیفیتی برای اولویت‌بندی خواسته‌ها.
۱۰. بسته‌بندی خواسته‌ها به‌طوری که به‌صورت افزایشی قابل تحویل باشند.
۱۱. ذکر قیدوبندهایی که بر سیستم نهاده خواهد شد.

بحث درباره روش‌های اعتبارسنجی سیستم.

هر دو مجموعه وظایف فوق «جمع‌آوری خواسته‌ها» را پیش می‌برند، ولی از نظر عمق و رسمیت، کاملاً متفاوت هستند. تیم نرم‌افزار، مجموعه وظایفی را انتخاب می‌کند که به کمک آن بتواند به هدف هر کنش دست پیدا کند و در عین حال، کیفیت و چابکی را حفظ کند.

> «تکرار الگوها چیزی کاملاً متفاوت با تکرار بخش‌هاست. در واقع؛ بخش‌های متفاوت، منحصر به فرد خواهند بود زیرا الگوها یکسان هستند.»
>
> **کریستوفر الکساندر**

الگوها را در هر سطحی از انتزاع می‌توان تعریف کرد[1]. در برخی موارد، از یک الگو می‌توان برای

[1] الگوها برای بسیاری از فعالیت‌های مهندسی نرم‌افزار قابل استفاده‌اند. تحلیل، طراحی و آزمودن الگوها در فصل‌های ۷، ۹، ۱۰، ۱۲ و ۱۴ بحث شده است.

توصیف یک مسأله مرتبط با یک مدل فرایند کامل (مثلاً ساخت نمونه‌ی اولیه) یا راهکار آن مسأله استفاده کرد. در شرایط دیگر، می‌توان از الگوها برای توصیف یک مشکل (و راهکار آن) مرتبط با یک فعالیت چارچوبی (مانند برنامه‌ریزی) یا کنشی در یک فعالیت چارچوبی (مثلاً برآورد پروژه) استفاده نمود.

امبلر [Amb98] برای توصیف الگوی فرایند، یک قالب پیشنهاد نموده است:

نام الگو. به الگو نامی با معنی داده می‌شود که آن را در حیطه‌ی فرایند نرم‌افزار توصیف می‌کند (مثلاً **TechnicalReviews**).

نیروها. محیطی که الگو در آن مشاهده می‌شود و مواردی که مسأله را پدیدار می‌کنند و ممکن است بر راهکار آن تأثیر بگذارند.

نوع. نوع الگو مشخص می‌شود. امبلر [Amb98] سه نوع الگو پیشنهاد می‌کند:

۱. *الگوی مرحله‌ای* – مساله‌ای مرتبط با یک فعالیت چارچوبی را برای فرایند تعریف می‌کند. چون یک فرایند چارچوبی شامل چند کنش و وظیفه کاری می‌شود، الگوی مرحله‌ای شامل چند الگوی وظیفه‌ای می‌شود (بند بعدی را ببینید)که به آن مرحله (فعالیت چارچوبی) مربوط می‌شوند. مثالی از الگوی مرحله‌ای می‌تواند **EstabilishingComminucation** باشد. این الگو شامل الگوی وظیفه‌ای **RequirementsGathering** و چند الگوی دیگر می‌شود.
۲. *الگوی وظیفه‌ای*– مساله‌ای مرتبط با یک کنش یا وظیفه کاری نرم‌افزاری را تعریف می‌کند که کار مهندسی نرم‌افزار موقق به آن بستگی دارد (مثلاً **RequirementsGathering** یک الگوی وظیفه‌ای است).
۳. *الگوی فازی* (*Phase*)– یک سری فعالیت‌های چارچوبی را تعریف می‌کند که درون فرایند رخ می‌دهند حتی هنگامی که جریان کلی فعالیت‌ها ماهیتی تکراری داشته باشند. مثالی از الگوی فازی می‌تواند **SpiralModel** یا **Prototyping** باشد.[1]

حیطه‌ی اولیه. شرایطی را توصیف می‌کند که الگو در آن کاربرد دارد. پیش از آغازکردن الگو: (۱) چه فعالیت‌های سازمانی یا مرتبط یا تیمی قبلاً رخ داده است؟ (۲) حالت ورودی برای فرایند چیست؟ (۳) چه اطلاعاتی درباره مهندسی نرم‌افزار یا پروژه از قبل موجود است؟

برای مثال، الگوی **Planning** (برنامه‌ریزی) مستلزم آن است که

۱. مشتریان و مهندسان نرم‌افزار یک ارتباط مبتنی بر همکاری برقرار کرده باشند؛
۲. چند الگوی وظیفه‌ای [مشخص شده] برای الگوی **Communication** کامل شده باشد؛ و
۳. حوزه‌ی پروژه، خواسته‌های تجاری اساسی و قیدهای پروژه معلوم شده باشند.

مسأله. مسأله‌ی خاصی که قرار است توسط این الگو حل شود.

راهکار. چگونگی پیاده‌سازی موفق الگو را توصیف می‌کند. در این بخش شرح داده می‌شود که حالت اولیه‌ی فرایند (که پیش از پیاده‌سازی الگو وجود دارد) چگونه به‌عنوان پیامدی از شروع الگو اصلاح می‌شود. همچنین چگونگی تبدیل اطلاعات مهندسی نرم‌افزار یا اطلاعات پروژه که قبل از آغاز الگو در دسترس است، به‌عنوان پیامدی از اجرای موفق الگو در همین بخش شرح داده می‌شود.

> **نکته‌ی کلیدی**
> یک قالب الگویی، ابزاری سازگار برای توصیف الگو فراهم می‌آورد.

[1] این الگوهای فازی در بخش ۳-۳-۲ بحث می‌شوند.

اطلاعات

مثالی از یک الگوی فرایند

در الگوی فرایند خلاصه‌شده‌ی زیر، روشی شرح داده شده است که وقتی می‌تواند قابل استفاده باشد که طرف‌های ذی‌نفع ایده‌ای کلی از آنچه باید انجام شود، در ذهن داشته باشند، ولی از خواسته‌های مشخص خود مطمئن نیستند.

نام الگو. RequirementsUnclear

مقاصد. در این الگو روش ساخت مدلی شرح داده می‌شود که به‌صورت تکراری توسط طرف‌های ذی‌نفع برای شناسایی یا تعیین خواسته‌های نرم‌افزاری قابل ارزیابی باشد.

نوع. الگوی فازی.

حیطه‌ی اولیه. پیش از شروع این الگو، شرایط زیر باید برقرار باشد: (۱) طرف‌های ذی‌نفع شناسایی شده باشند؛ (۲) شیوه‌ی ارتباطی میان طرف‌های ذی‌نفع و تیم نرم‌افزاری تعیین شده باشد؛ (۳) طرف‌های ذی‌نفع، مسأله‌ای را که نرم‌افزار باید حل کند، مشخص کرده باشند؛ (۴) درک اولیه‌ای از حوزه‌ی پروژه، خواسته‌های تجاری اساسی و قیدهای پروژه به‌عمل آمده باشد.

مسأله. خواسته‌ها مبهم هستند یا اصلاً وجود ندارند، ولی در عین حال، می‌دانیم که مسأله‌ای هست که باید حل شود و این مسأله باید با یک راهکار نرم‌افزاری حل شود. طرف‌های ذی‌نفع از آنچه می‌خواهند، اطمینان ندارند؛ یعنی، نمی‌توانند خواسته‌های نرم‌افزاری را با ذکر جزئیات توصیف کنند.

راهکار. در اینجا توصیفی از فرایند ایجاد نمونه‌ی اولیه ارائه می‌شود که بعداً در بخش ۳-۳-۲ شرح داده خواهد شد.

حیطه‌ی حاصل. نمونه‌ی اولیه‌ای از نرم‌افزار که خواسته‌های اساسی را تعیین می‌کند (مثل شیوه‌های تعامل، ویژگی‌های محاسباتی، عملیات پردازشی) توسط طرف‌های ذی‌نفع به تصویب می‌رسد. سپس، (۱) نمونه‌ی اولیه ممکن است از طریق یک سری گام‌های پیاپی تکامل یابد تا به نرم‌افزار نهایی برسد یا (۲) نمونه‌ی اولیه ممکن است کنار گذاشته شود و تیم تولید از یک الگوی فرایند دیگر استفاده کند.

الگوی مرتبط. الگوهای زیر با این الگو در ارتباط هستند: **Customer Communication**، **Iterative Design**، **Iterative Development**، **Customer Assessment**، **Requirement Extraction**

کاربردها و مثال‌های شناخته‌شده. تهیه نمونه‌ی اولیه هنگامی توصیه می‌شود که خواسته‌ها قطعی نباشد.

حیطه‌ی حاصل. شرایطی را توصیف می‌کند که ماحصل پیاده‌سازی موفق الگو هستند: (۱) کدام فعالیت‌های سازمانی یا تیمی باید رخ داده باشد؟ (۲) حالت خروج برای فرایند چیست؟ (۳) کدام اطلاعات مهندسی نرم‌افزار یا اطلاعات پروژه توسعه یافته‌اند؟

الگوهای مرتبط. فهرستی از همه‌ی الگوهای فرایند تهیه کنید که با این الگو ارتباط مستقیم دارند. این را می‌توان به‌صورت یک سلسله مراتب یا هر شکل نموداری دیگری نمایش داد. برای مثال،

الگوی مرحله‌ای **Communication** شامل الگوهای وظیفه‌ای **Project Team**، **Collaborative Constraint Description**، **Requirement Gathering**، **Scope Isolation**، **Guidelines** و **Scenario Description** می‌شود.

کاربردها و مثال‌های شناخته‌شده. موارد خاصی را مشخص کنید که الگو در آنها قابل اعمال باشد. برای مثال، **Communication** در آغاز هر پروژه‌ی نرم‌افزاری، لازم‌الاجرا است، در سرتاسر پروژه‌ی نرم‌افزاری توصیه می‌شود و هنگامی که فعالیت استقرار در شُرُف انجام است، لازم‌الاجرا می‌شود.

الگوهای فرایند، سازوکاری اثربخش برای پرداختن به مشکلات مرتبط با هر فرایند نرم‌افزار فراهم می‌آورند. به کمک این الگوها می‌توانید توصیفی سلسله مراتبی از فرایند ارائه کنید که در سطح بالایی از انتزاع آغاز می‌شود (یک الگوی فازی). سپس این توصیف به مجموعه‌ای از الگوهای مرحله‌ای پالایش می‌شود که فعالیت‌های چارچوبی را توصیف کرده پس از پالایش بیشتر به‌شیوه‌ای سلسله مراتبی به الگوهای وظیفه‌ای برای هر الگوی مرحله‌ای تقسیم می‌شود که جزئیات بیشتر را در بر دارند. هنگامی که الگوهای فرایند توسعه یافتند، از آنها می‌توان برای تعریف شکل‌های متفاوت فرایند استفاده کرد-یعنی تیم نرم‌افزاری می‌تواند با استفاده از این الگوها به‌عنوان قطعات سازنده، یک مدل فرایند سفارشی را تعریف نماید.

مرجع وب

منابعی جامع درباره الگوهای فرایند را می‌توان در وب‌سایت زیر یافت.

www.ambysoft.com/processPatternsPage.html

۲-۲ ارزیابی فرایند و بهبودی

وجود یک فرایند نرم‌افزاری، تضمینی برای تحویل به‌موقع نرم‌افزار با توانایی برآوردن نیازهای مشتری یا ارائه‌ی خصوصیات فنی که به خصوصیات کیفیتی دراز مدت منجر شود (فصل‌های ۱۴ و ۱۶)، نیست. الگوهای فرایند باید با کار مهندسی نرم‌افزار (بخش دوم کتاب) همراه شود. به‌علاوه، خود فرایند را می‌توان مورد ارزیابی قرار داد تا اطمینان حاصل شود که ضروری‌بودن مجموعه‌ای از ملاک‌های اساسی برای یک مهندسی نرم‌افزار موفق، نشان داده شده است[1].

نکته‌ی کلیدی

هدف از ارزیابی، شناخت وضعیت فعلی فرایند نرم‌افزار به قصد بهبود بخشیدن به آن است.

چند روش متفاوت برای ارزیابی فرایند نرم‌افزار و بهسازی آن طی چند دهه‌ی گذشته پیشنهاد شده است:

روش ارزیابی استاندارد CMMI برای بهسازی (SCAMPI)- یک مدل ارزیابی فرایند پنج مرحله‌ای فراهم می‌آورد که شامل پنج فاز می‌شود: شروع، عیب‌یابی، ساخت، عملیات و یادگیری. در روش SCAMPI از SEI CMMI به‌عنوان مبنایی برای ارزیابی استفاده می‌شود [SEI00].

ارزیابی مبتنی بر CMM برای بهبودبخشیدن به فرایندهای داخلی (CBA IPI)- یک تکنیک عیب‌یابی برای ارزیابی بلوغ نسبی یک سازمان نرم‌افزاری فراهم می‌آورد؛ در این تکنیک از SEI CMM به‌عنوان مبنایی برای ارزیابی استفاده می‌شود [Dun 01].

SP/CE (ISO/IEC15504)- استانداردی که مجموعه‌ای از خواسته‌ها را برای ارزیابی فرایند نرم‌افزار تعریف می‌کند. هدف این استاندارد، کمک به سازمان‌ها در توسعه‌ی یک ارزیابی عینی از بازدهی هرگونه فرایند نرم‌افزار تعریف شده است [ISO08].

چه تکنیک‌های رسمی‌ای برای ارزیابی فرایند نرم‌افزار موجود است؟

---

[1] در [CMM07] شرحی از فرایند نرم‌افزار ارائه شده است و ملاک‌های مربوط به مهندسی موفق با جزئیات فراوان توصیف شده‌اند.

ISO 9001:2000 برای نرم‌افزار- یک استاندارد عمومی که برای هر سازمانی که مایل به بهسازی کیفیت کلی محصولات، سیستم‌ها یا سرویس‌های ارائه‌شده‌اش باشد، قابل استفاده است. بنابراین، استاندارد مذکور به‌طور مستقیم در سازمان‌ها و شرکت‌های نرم‌افزاری قابل استفاده است [Ant06].

بحث مشروح ارزیابی نرم‌افزار و روش‌های بهسازی فرایند در فصل ۳۰ ارائه خواهد شد.

۲-۳ مدل‌های فرایند تجویزی

مدل‌های فرایند تجویزی[1] در ابتدا برای نظم‌بخشیدن به آشوب موجود در توسعه‌ی نرم‌افزار پیشنهاد شدند. تاریخ نشان داده است که این مدل‌های سنتی به میزان معینی به‌کار مهندسی نرم‌افزار ساختار بخشیده‌اند و راهنمای اثربخشی برای تیم‌های نرم‌افزاری فراهم ساخته‌اند، ولی کار مهندسی نرم‌افزار و محصولی که از آن به‌دست می‌آید، در «آستانه‌ی آشوب» قرار می‌گیرد.

در یک مقاله‌ی جالب در خصوص رابطه‌ی عجیب میان نظم و آشوب در جهان نرم‌افزار، نوگریا و همکاران [Nog00] بیان می‌کنند که:

> آستانه‌ی آشوب به‌صورت «یک حالت طبیعی میان نظم و آشوب، یک مصالحه‌ی بزرگ میان ساختار و شگفتی» تعریف می‌شود [Kau95]. آستانه‌ی آشوب را می‌توان به‌صورت یک حالت ساخت‌یافته‌ی جزئی و ناپایدار تجسم کرد... ناپایدار است چون پیوسته جذب آشوب یا نظم مطلق می‌شود.
>
> ما تمایل داریم فکر کنیم که نظم، حالت ایده‌آل طبیعت است و این می‌تواند اشتباه باشد. پژوهش... مؤید این نظریه است که عملکرد دور از تعادل باعث ایجاد خلاقیت، فرایندهای خود سازمان و افزایش برگشتی می‌شود [Roo96]. نظم مطلق به معنای نبود تغییرات است که می‌تواند تحت شرایط غیرقابل پیش‌بینی، مزیت باشد. تغییر هنگامی رخ می‌دهد که قدری ساخت‌یافتگی وجود داشته باشد، به‌طوری که تغییر را بتوان سازمان‌دهی کرد، ولی نه چنان سخت که نتواند رخ دهد، ولی آشوب بیش‌ازحد، می‌تواند هماهنگ‌سازی و یکپارچگی را غیر ممکن کند. فقدان ساخت‌یافتگی همواره به‌معنای بی‌نظمی است.

این گفته‌ها معنایی فلسفی برای مهندسی نرم‌افزار دارد. اگر مدل‌های فرایند تجویزی[2] در جستجوی ساختار و نظم باشند، آیا برای یک جهان نرم‌افزاری که با تغییرات پیشرفت می‌کند، نامناسب هستند؟ به‌علاوه، اگر مدل‌های فرایند سنتی (و نظم ناشی از آنها) را رد کنیم و چیزی با ساخت‌یافتگی کمتر را جایگزین آنها کنیم، آیا دستیابی به هماهنگی و یکپارچگی در کار نرم‌افزاری را غیر ممکن ساخته‌ایم؟

پاسخ گفتن به این پرسش‌ها آسان نیست، ولی متغیرهای متفاوتی فراروی مهندسان نرم‌افزار وجود دارد. در بخش‌هایی که به‌دنبال خواهد آمد، به بررسی روش فرایند تجویزی خواهیم پرداخت که در آن، نظم و سازگاری پروژه، مسائل عمده به‌شمار می‌رود. ما آنها را «تجویزی» می‌خوانیم زیرا مجموعه‌ای از عناصر فرایند-فعالیت‌های چارچوبی، عملیات مهندسی نرم‌افزار، وظایف، محصولات کاری، تضمین کیفیت و سازوکارهای کنترل تغییرات را برای هر پروژه تجویز می‌کنند. هر مدل فرایند همچنین یک جریان فرایند (یا جریان کاری) را تعریف می‌کند که شیوه‌ی ارتباط میان عناصر فرایند را تعریف می‌کند.

> «سازمان‌های نرم‌افزاری کمبودهای چشمگیری در توانایی خود برای کسب تجربه از پروژه‌های کامل شده از خود نشان داده‌اند.»
>
> ناسا

> «اگر فرایند درست باشد، نتایج خودشان درست خواهند بود.»
>
> تاکاشی اوساداً

نکته‌ی کلیدی

مدل‌های فرایند تجویزی، مجموعه‌ای از عناصر تجویز شده و جریان کار تجویز شده را تعریف می‌کنند.

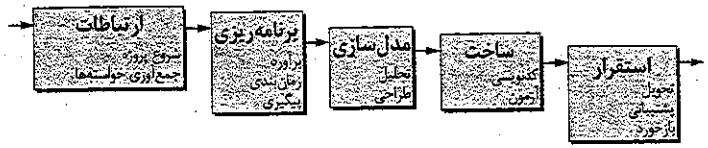
[1] prescriptive

[2] گرچه در مدل آبشاری اولیه‌ای که وینستون رویس [Roy70] پیشنهاد کرد، تدابیری برای «حلقه‌های بازخورد» اندیشیده شده بود، اکثریت وسیع سازمان‌هایی که این مدل فرایند را به‌کار می‌بردند، به آن به دیده‌ای کاملاً خطی می‌نگرند.

همه‌ی مدل‌های فرایند می‌توانند فعالیت‌های چارچوبی عمومی توصیف شده در فصل ۱ را در خود جای دهند، ولی هر کدام به‌طرزی متفاوت بر این فعالیت‌ها تأکید می‌گذارد و یک جریان فرایندی تعریف می‌کند که به‌شیوه‌ای متفاوت به هر فعالیت چارچوبی (و نیز کنش‌ها و وظایف مهندسی نرم‌افزار) نظر دارد.

۱-۳-۲ مدل آبشاری[1]

گاه پیش می‌آید که خواسته‌های مربوط به یک مسأله به‌خوبی شناخته شده‌اند- هنگامی که کار به طریق خطی، از برقراری ارتباط تا استقرار جریان پیدا می‌کند (شکل ۳-۲). گاه این وضعیت هنگامی مشاهده می‌شود که تطبیق خوش تعریف یا بهسازی یک سیستم موجود ضرورت پیدا می‌کند (نظیر تغییر نرم‌افزار حسابداری که به دلیل تغییرات در ساختار دولت ضرورت پیدا کرده است).



شکل ۳-۲ مدل آبشاری.

این وضعیت همچنین ممکن است در تعداد محدودی از تلاش‌های توسعه‌ای پیش آید، ولی تنها هنگامی که خواسته‌ها به‌خوبی مشخص شده باشند و از پایداری مناسبی برخوردار باشند.

در مدل آبشاری، که گاه از آن به‌عنوان چرخه‌ی حیاتی کلاسیک یاد می‌شود، روشی سیستماتیک و ترتیبی[2] برای توسعه‌ی نرم‌افزار پیشنهاد می‌شود که با مشخص‌کردن خواسته‌ها توسط مشتری آغاز می‌شود و از طریق برنامه‌ریزی، مدل‌سازی، ساخت و استقرار پیش می‌رود و در پشتیبانی مستمر نرم‌افزار کامل شده به اوج می‌رسد.

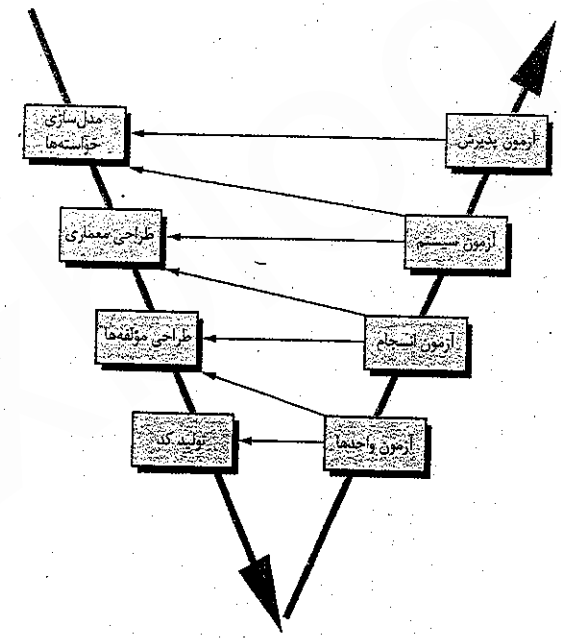
شکل دیگری در نمایش مدل آبشاری به‌عنوان مدل V شناخته می‌شود. مدل V، که در شکل ۴-۲ نمایش داده شده است، [Buc99] رابطه‌ی تضمین کیفیت با کنش‌های مرتبط با ارتباط، مدل‌سازی و فعالیت‌های ساخت اولیه را تصویر می‌کند. با حرکت تیم نرم‌افزاری به طرف پایین و سمت چپ V، خواسته‌های اساسی مسأله رفته رفته پالایش می‌شوند و جزئیات بیشتری از آنها تعیین می‌شود و مسأله و راهکار آن بهتر نمایش داده می‌شود. هنگامی که کدها نوشته شد، تیم در طرف راست V به طرف بالا حرکت می‌کند و اساساً یک سری آزمون اجرا می‌کند (کنش‌های تضمین کیفیت) تا هر کدام از مدل‌های ایجادشده در مدت حرکت تیم به طرف پایین را وارسی کند.[3] در جهان واقعیت، هیچ اختلاف بنیادی میان چرخه‌ی حیات کلاسیک و مدل V وجود ندارد. مدل V راهی برای تجسم بخشیدن به چگونگی وارسی و اعتبارسنجی در ابتدای کار نرم‌افزار فراهم می‌آورد.

نکته‌ی کلیدی

مدل V چگونگی ارتباط کنش‌های وارسی و اعتبار بخشی با کنش‌های فنی مهندسی را نشان می‌دهد.

[1] waterfall model

[3] بحثی مشروح درباره کنش‌های تضمین کیفیت در بخش سوم این کتاب ارائه شده است.



شکل ۴-۲ مدل V.

مدل ترتیبی خطی، قدیمی‌ترین و پرکاربردترین الگو برای مهندسی نرم‌افزار است. ولی، نقد این الگو باعث شده که حتی هواداران فعال آن نیز بازدهی آن را مورد تردید قرار دهند [Han95]. از جمله مشکلاتی که به هنگام اجرای مدل ترتیبی خطی پیش می‌آید، می‌توان به موارد زیر اشاره کرد:

۱. پروژه‌های واقعی به‌ندرت جریان ترتیبی پیشنهادشده توسط این مدل را دنبال می‌کنند. گرچه مدل خطی می‌تواند پذیرای تکرار باشد، این عمل را به‌طور غیرمستقیم انجام می‌دهد. درنتیجه، با پیش رفتن تیم پروژه، ممکن است تغییرات باعث سردرگمی شوند.

۲. غالباً برای مشتری دشوار است که همه‌ی نیازهای خود را به وضوح بیان کند. مدل ترتیبی خطی، به بیان واضح نیاز دارد و به‌خوبی از پس موارد غیرقطعی که در آغاز اکثر پروژه‌ها وجود دارند، برنمی‌آید.

۳. مشتری باید حوصله داشته باشد. یک نسخه‌ی کاری از برنامه‌ها تا آخرین روزهای پروژه در دسترس او قرار نخواهد گرفت. یک اشتباه عمده که تا زمان بازبینی برنامه‌ی کاری از دید پنهان بماند، می‌تواند بسیار دردسرآفرین باشد.

چرا مدل آبشاری گاهی به شکست می‌انجامد؟

برادای [Bra94] طی تحلیل جالبی که روی پروژه‌های واقعی انجام داده است، دریافته است که طبیعت خطی چرخه‌ی حیات کلاسیک به «حالت‌های مسدود کننده‌ای» منجر می‌شود که در آنها برخی اعضای تیم پروژه باید منتظر سایر اعضای تیم بمانند تا وظایف وابسته انجام شود. در واقع، زمان

صرف‌شده برای این انتظار می‌تواند از زمان صرف‌شده روی کاری با بهره‌وری بالا بیشتر شود! حالت‌های مسدودکننده در ابتدا و انتهای یک فرایند ترتیبی خطی بیشتر رایج‌اند.

امروزه کار نرم‌افزاری با گام‌هایی سریع انجام می‌شود و در معرض جریانی بی‌پایان از تغییرات (در ویژگی‌ها، در عملکردها و در محتوای اطلاعاتی) قرار دارند. مدل آبشاری غالباً برای چنین کاری نامناسب است، ولی در شرایطی که خواسته‌ها ثابت هستند و قرار است کار تا پایان به‌شیوه‌ای خطی پیش برود، می‌توان به‌عنوان مدلی مفید عمل کند.

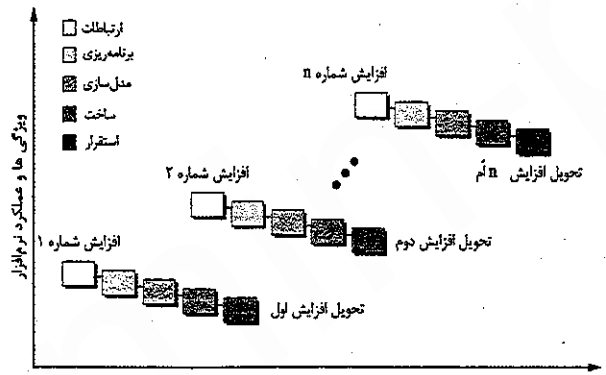
> «کار نرم‌افزاری غالب اوقات از قانون اول دوچرخه‌سواری پیروی می‌کند: در هر مسیری که باشید، سربالایی و باد مخالف را پیش رو دارید.»
>
> ناشناس

۲-۳-۲ مدل‌های فرایند افزایشی

وضعیت‌های فراوانی وجود دارد که در آنها خواسته‌های اولیه‌ی نرم‌افزار به‌خوبی تعریف شده‌اند، ولی حوزه‌ی کلی تلاش‌های به‌عمل‌آمده در توسعه‌ی نرم‌افزار، مانع از یک فرایند خطی محض می‌شود. به‌علاوه، ممکن است نیاز به فراهم کردن سریع مجموعه‌ی محدودی از عملکردهای نرم‌افزار برای کاربران و سپس پالایش و بسط بر اساس آن عملکردها در نسخه‌های بعدی نرم‌افزار ضرورت پیدا کند. در چنین مواردی، می‌توانید یک مدل فرایند انتخاب کنید که برای تولید نرم‌افزار به‌شیوه‌ی افزایشی طراحی شده باشد.

> **نکته‌ی کلیدی**
>
> مدل افزایشی یک سری نرم‌افزار تحویل می‌دهد که هر کدام یک گام تا مبدء می‌شود و این گام‌ها هر یک نسبت به سلف خود عملکرد بیشتری در اختیار مشتری قرار می‌دهند.

مدل افزایشی، عناصر مدل ترتیبی خطی را با جریان‌های فرایند خطی و موازی بحث‌شده در بخش ۲-۱ تلفیق می‌کند. با رجوع به شکل ۵-۲، مشاهده می‌شود که مدل افزایشی، مراحل ترتیبی خطی را به شیوه‌ای باورنکردنی با پیشرفت زمانی تقویم اجرا می‌کند. هر ترتیب خطی یک «افزایش» قابل تحویل از نرم‌افزار را [McD93] را به شیوه‌ای ارائه می‌دهد که مشابه با افزایش‌های تولیدشده توسط یک جریان فرایند تکاملی است (بخش ۳-۳-۲).



شکل ۵-۲ مدل افزایشی.

برای مثال، نرم‌افزار واژه‌پردازی که با استفاده از الگوی افزایشی توسعه یافته است، ممکن است اعمالی از قبیل مدیریت فایل، تولید و ویرایش مستندات را در نسخه‌ی اول، قابلیت‌های پیچیده‌تر ویرایشی و تولید مستندات در نسخه‌ی دوم؛ چک‌کردن املاء و دستور در نسخه‌ی سوم؛ و قابلیت‌های پیشرفته صفحه‌بندی را در نسخه‌ی چهارم تحویل دهد. باید توجه داشت که جریان فرایند برای هر افزایش می‌تواند الگوی ساخت نمونه‌ی اولیه را در خود داشته باشد.

هنگامی که از یک مدل افزایشی استفاده شود، افزایش نخست غالباً محصول هسته‌ای است، یعنی به خواسته‌های پایه می‌پردازد، ولی بسیاری از ویژگی‌های مکمل (که برخی معلوم و برخی نامعلوم هستند) تحویل داده نمی‌شوند. محصول هسته‌ای توسط مشتری مورد استفاده (یا بازبینی مفصل) قرار می‌گیرد. درنتیجه‌ی استفاده و/یا ارزیابی، طرحی برای افزایش بعدی توسعه می‌یابد. این طرح حاوی اصلاحاتی است که نیازهای مشتری و تحویل قابلیت‌ها و ویژگی‌های اضافی را بهبود می‌بخشد. این فرایند به‌دنبال تحویل هر قطعه تکرار می‌شود تا اینکه محصول کامل تولید شود.

اندرز

مشتری شما تاریخ تحویلی را درخواست می‌کند که غیرممکن است. تحویل یک یا چند نسخه از نرم‌افزار را تا آن تاریخ پیشنهاد کنید و بقیه را بعداً تحویل دهید.

مدل فرایند افزایشی، همانند مدل ساخت نمونه‌ی اولیه (بخش ۵-۲) و روش‌های تکاملی دیگر، ماهیتی تکراری دارد. ولی برخلاف مدل ساخت نمونه‌ی اولیه، مدل افزایشی بر تحویل قطعه‌ای در هر افزایش تأکید می‌ورزد. قطعات اولیه، نسخه‌های «دست‌وپاشکسته‌ای» از محصول نهایی هستند، ولی قابلیت ارائه خدمات به کاربر را داشته به‌عنوان محیطی برای ارزیابی توسط کاربر نیز عمل می‌کنند[1].

توسعه افزایشی به‌ویژه هنگامی مفید واقع می‌شود که تعداد کارمندان لازم برای تکمیل پیاده‌سازی پروژه در مهلت کاری مقرر، در دسترس نباشد. «افزایش‌های» اولیه را با تعداد کمتری از افراد می‌توان پیاده‌سازی نمود. اگر محصول هسته‌ای به‌خوبی دریافت شود، کارمندان دیگری را (در صورت نیاز) می‌توان اضافه کرد و افزایش بعدی را پیاده‌سازی کرد. به‌علاوه، می‌توان افزایش‌ها را طوری برنامه‌ریزی کرد که خطرات تکنیکی قابل مدیریت باشند. برای مثال، یک سیستم اصلی ممکن است نیاز به سخت‌افزار جدیدی داشته باشد که فعلاً در حال توسعه است و تاریخ تحویل آن قطعی نیست. ممکن است برنامه‌ریزی افزایش‌های اولیه به‌شیوه‌ای که از به‌کارگیری این سخت‌افزار پرهیز شود، میسر باشد و در نتیجه، بخشی از عملکردها بدون تاخیر چشمگیر به کاربر نهایی تحویل شود.

۳-۳-۲ مدل‌های فرایند تکاملی

نرم‌افزارها نیز همانند همه‌ی سیستم‌های پیچیده‌ی دیگر، در اثر مرور زمان تکامل می‌یابند. خواسته‌های تجارتی محصول، غالباً به موازات توسعه، تغییر می‌یابند و منجر به ساخت محصول نهایی غیرواقعی می‌شوند؛ مهلت‌های زمانیِ محدود بازار، کامل‌کردن یک محصول نرم‌افزاری مفهومی را غیرممکن می‌سازند، ولی یک نسخه‌ی محدود را باید وارد بازار کرد تا فشارهای رقابتی یا کاری را مرتفع سازد؛ مجموعه‌ای از خواسته‌های اصلی و محوری سیستم یا محصول به‌خوبی درک می‌شود، ولی جزئیات محصول یا سیستم هنوز باید مشخص شود. در این وضعیت‌ها یا وضعیت‌های مشابه، مهندسان نرم‌افزار به مدل فرایندی نیاز دارند که به‌طور مشخص برای محصول طراحی شده باشد و با گذشت زمان تکامل می‌یابد.

نکته‌ی کلیدی

در مدل‌های فرایند تکاملی در هر دور از تکرار نسخه‌ی کامل‌تری از نرم‌افزار تولید می‌شود.

ساخت نمونه‌ی اولیه. غالباً، مشتری یک مجموعه اهداف کلی برای نرم‌افزار تعیین می‌کند، ولی جزئیات خواسته‌های مربوط به قابلیت‌ها یا ویژگی‌ها را مشخص نمی‌کند. در موارد دیگر، ممکن است سازنده از بازدهی یک الگوریتم، قابلیت تطابق با یک سیستم عامل خاص یا شکل تعامل «انسان – ماشین» مطمئن نباشد. در این شرایط و بسیاری از شرایط دیگر، الگوی ساخت نمونه‌ی اولیه ممکن است بهترین روش باشد.

«طوری طراحی کنید که نمونه‌ی اولیه را بعداً کنار بگذارند. چون به‌هرحال این کار را باید بکنید. تنها راهی که دارند این است که بگویند یک محصول موقتی به مشتری بفروشید.»

فردریک پ. بروکس

[1] شایان ذکر است که فلسفه فرایند افزایشی در تمامی مدل‌های فرایند چابکی که در فصل ۳ بحث خواهند شد نیز مفید است.

اندرز

هنگامی که مشتری شما خواسته‌ای مشروح دارد، ولی جزئیات چندانی در اختیار شما قرار نداده است، به‌عنوان نخستین قدم، یک نمونه‌ی اولیه بسازید.

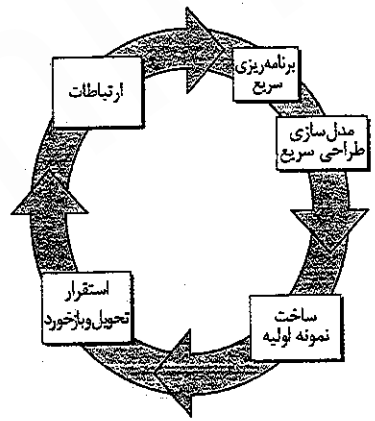
گرچه از تهیه نمونه‌ی اولیه می‌توان به‌عنوان یک مدل فرایند مستقل استفاده کرد، از آن بیشتر به‌عنوان تکنیکی استفاده می‌شود که می‌توان در حیطه‌ی هر کدام از مدل‌های فرایند ذکرشده در این فصل آن را پیاده کرد. شیوه‌ی به‌کار گرفته شده هر چه که باشد، الگوی تهیه‌ی نمونه‌ی اولیه به شما و سایر طرف‌های ذی‌نفع کمک خواهد کرد که بهتر درک کنید هنگام مبهم بودن خواسته‌ها، چه چیزی قرار است ساخته شود.

الگوی ساخت نمونه‌ی اولیه (شکل ۶-۲) با جمع‌آوری خواسته‌ها آغاز می‌شود. مشتری و سازنده با هم ملاقات می‌کنند و اهداف کلی نرم‌افزار را تعیین می‌کنند، همه‌ی خواسته‌های معلوم را شناسایی می‌کنند و زمینه‌هایی را مطرح می‌کنند که تعریف بیشتر در آنها الزامی است. سپس یک «طراحی سریع» صورت می‌پذیرد. در طراحی سریع، هدف اصلی، ارائه آن دسته از ویژگی‌های نرم‌افزار است که به چشم مشتری/کاربر می‌آیند (مثل روش‌های واردکردن اطلاعات و فرمت‌های خروجی). طراحی سریع منجر به ساخت یک نمونه‌ی اولیه می‌شود. نمونه‌ی اولیه مورد ارزیابی مشتری/کاربر قرار گرفته از آن برای پالایش خواسته‌های نرم‌افزار مورد نظر استفاده می‌شود. با تنظیم نمونه‌ی اولیه برای برآوردن نیازهای مشتری، تکرار رخ می‌دهد و در عین حال، سازنده بهتر می‌فهمد که چه نیازهایی باید برآورده شود.

در حالت ایده‌آل، نمونه‌ی اولیه به‌عنوان راهکاری برای تشخیص خواسته‌های نرم‌افزار عمل می‌کند. اگر یک نمونه‌ی اولیه‌ی کاری ساخته شود، سازنده می‌کوشد تا از قطعات برنامه‌ی موجود استفاده کند یا از ابزارهایی (مانند مولد گزارش، مدیریت پنجره و غیره) استفاده کند تا برنامه‌های کاری به سرعت تولید شود.

ولی هنگامی که نمونه‌ی اولیه، اهداف ذکر شده در بالا را برآورده ساخت، با آن چه می‌کنیم؟ بروکز [Bro75] چنین پاسخ می‌دهد:

> در اکثر پروژه‌ها، نخستین سیستمی که ساخته می‌شود چندان قابل استفاده نیست. ممکن است بیش از حد آهسته باشد، بیش از حد بزرگ باشد، استفاده از آن دشوار باشد، یا این هر سه عیب را با هم داشته باشد. چاره‌ای جز شروع دوباره وجود ندارد. باید نسخه‌ی دیگری ساخت که این مشکلات در آن حل شده باشد ...



شکل ۶-۲ الگوی ساخت نمونه‌ی اولیه.

SafeHome

انتخاب یک مدل فرایند

صحنه: اتاق کنفرانس گروه مهندسی نرم‌افزار در شرکت CPI، شرکتی (نخیلی) که محصولات خانگی و تجاری تولید می‌کند.

نقش‌آفرینان: لی وارن، مدیر مهندسی؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ جیمی لازار، عضو تیم نرم‌افزار؛ وینود رامان، عضو تیم نرم‌افزار؛ اد رابینز، عضو تیم نرم‌افزار.

گفتگوها

لی: بسیار خوب. جمع‌بندی می‌کنیم من مدتی را صرف بحث درباره‌ی خط تولید *SafeHome* کردم. بدون شک، یک عالم کار داریم تا بتوانیم تعریف ساده‌ای از محصول ارائه بدهیم، ولی دوست دارم شماها فکر بکنید ببینید چطور می‌شود به بخش نرم‌افزاری پروژه نزدیک شد.

داگ: به‌نظر می‌رسد ما در گذشته خیلی در نگاه‌کردن به نرم‌افزار منظم عمل نکردیم.

اد: نمی‌دانم داگ، ما همیشه محصول را بیرون دادیم.

داگ: این درست، ولی با یک عالم ادبیت و آزار و این پروژه هم که به‌نظر می‌رسد از هرچه تا به حال انجام داده ایم، بزرگتر باشد.

جیمی: اندازه‌ها هم سخت به‌نظر نمی‌رسد، ولی موافقم- روشی که برای پروژه‌های قبلی به‌کار می‌بردیم اینجا جواب نمی‌دهد؛ مخصوصاً اگر یک برنامه‌ی زمانی فشرده داشته باشیم.

داگ (با لبخند): دوست دارم در روشی که استفاده می‌کنیم، یک قدری حرفه‌ای‌تر باشیم. من هفته قبل در یک دوره‌ی آموزشی کوتاه شرکت کردم و چیزهای زیادی درباره مهندسی نرم‌افزار یاد گرفتم. مطالب جالبی بود. اینجا به یک فرایند نیاز داریم.

جیمی (با اخم): کار من ساختن نرم‌افزار است نه کاغذبازی.

داگ: قبل از این که با من مخالفت کنی، به من فرصت بده منظورم این است. [داگ چارچوب فرایند شرح داده شده در این فصل و مدل فرایند تجویزی را توضیح می‌دهد.]

داگ: خلاصه، به‌نظر من مدل خطی به درد ما نمی‌خورد، چون در این مدل این طور فرض می‌شود که همه‌ی خواسته‌ها معلوم است، که این خیلی محتمل نیست.

وینود: بله، و در ضمن، زیادی ITگرایانه به‌نظر می‌رسد. احتمالاً برای ساختن یک سیستم کنترل موجودی یا چیزهای شبیه آن مناسب است، ولی برای *SafeHome* خوب نیست.

داگ: موافقم.

اد: این روش تهیه‌ی نمونه‌ی اولیه، به‌نظر مناسب می‌آید. خیلی از کارهایی که انجام می‌دهیم، همین طوری است.

وینود: این مشکل ایجاد می‌کند. من نگران این هستم که ساخت‌یافتگی کافی را در اختیار ما قرار ندهد.

داگ: جای نگرانی نیست. ما گزینه‌های زیادی داریم و از شما می‌خواهم بهترین گزینه را برای تیم و برای پروژه انتخاب کنید.

اندرز

در برابر فشار توسعه‌ی نمونه‌ی اولیه به محصول، مقاومت کنید. درنتیجه‌ی این روند، همواره کیفیت آسیب می‌بیند.

نمونه‌ی اولیه می‌تواند به‌عنوان «نخستین سیستم» عمل کند. یعنی همان‌طور که بروکز توصیه می‌کند، دور انداخته شود. ولی این دیدگاه ممکن است ایده‌آل باشد. این درست است که برخی نمونه‌های اولیه به این منظور ساخته می‌شوند که دور انداخته شوند، ولی عده دیگری نیز طبیعتی تکاملی دارند از این لحاظ که نمونه‌ی اولیه به تدریج به سیستم واقعی تکامل می‌یابد.

هم افراد ذی‌نفع و هم سازندگان، الگوی ساخت نمونه‌ی اولیه را دوست دارند. کاربران احساس می‌کنند که یک سیستم واقعی را آزمایش می‌کنند و سازندگان چیزی را بلافاصله ساخته‌اند. با این همه، ساخت نمونه‌ی اولیه نیز می‌تواند به دلایل زیر مشکل‌آفرین باشد:

۱. افراد ذی‌نفع چیزی را می‌بینند که ظاهراً یک نسخه‌ی کاری از نرم‌افزار است. ولی نمی‌دانند که این نمونه‌ی اولیه با «موم» سرهم‌بندی شده است، نمی‌دانند که به لحاظ شتابی که در به کارگیری داشته‌ایم، کیفیت کلی نرم‌افزار و قابلیت نگهداری درازمدت ملتظر نبوده است. هنگامی که مطلع می‌شود محصول باید بازسازی شود تا به سطوح بالای کیفیت برسد، از کوره در می‌رود و تقاضا می‌کند با چند ترمیم جزئی این نمونه‌ی اولیه به یک محصول کاری تبدیل شود. اکثر اوقات هم مدیریت ساخت نرم‌افزار کوتاه می‌آید.

۲. مهندس نرم‌افزار غالباً برای به‌کارگیری هرچه سریع‌تر نمونه‌ی اولیه، در پیاده‌سازی دقیق آن کوتاه می‌آید. ممکن است از یک سیستم عامل یا زبان برنامه‌نویسی نامناسب استفاده شود، صرفاً به‌خاطر این که در دسترس و شناخته شده است؛ ممکن است یک الگوریتم ناکارآمد پیاده‌سازی شود. صرفاً برای آنکه قابلیت برنامه نشان داده شود. پس از مدتی ممکن است برنامه‌نویس با این انتخاب‌ها مأنوس شود و کلاً فراموش کند که چرا نامناسب بوده‌اند. انتخاب «کمتر از ایده‌آل» اکنون به بخشی از سیستم تبدیل شده است.

ممکن است مشکلاتی رخ دهد، ولی ساخت نمونه‌ی اولیه می‌تواند الگوی مؤثری برای مهندسی نرم‌افزار باشد. کلید کار، تعیین قواعد بازی در همان آغاز است؛ یعنی افراد ذی‌نفع و سازنده هر دو باید بپذیرند که نمونه‌ی اولیه بدین منظور ساخته می‌شود که به‌عنوان سازوکاری برای تعیین خواسته‌ها عمل کند. سپس (دست‌کم بخش‌هایی از آن) دور انداخته می‌شود و نرم‌افزار واقعی با مدنظر قرار دادن کیفیت، مهندسی می‌شود.

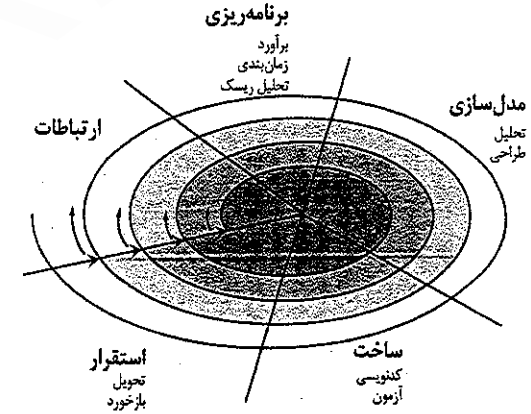
مدل مارپیچی (حلزونی). مدل مارپیچی که نخستین بار بوهم [Boe88] آن را پیشنهاد کرد، یک مدل فرایند نرم‌افزاری تکاملی است که ماهیت تکراری مدل ساخت نمونه‌ی اولیه را با جنبه‌های کنترلی و سیستماتیک مدل ترتیبی خطی (آبشاری) تلفیق می‌کند. این مدل پتانسیل لازم برای بسط سریع نسخه‌های تکاملی نرم‌افزار را داراست. بوهم [Boe01a] این مدل را به شیوه زیر توصیف می‌کند:

> مدل توسعه‌ی مارپیچی، یک مولد مدل فرایند مبتنی بر ریسک است که برای هدایت مهندسی همزمان طرف‌های ذی‌نفع سیستم‌های نرم‌افزاری به‌کار می‌رود. این مدل دو ویژگی متمایز دارد. یکی روش چرخه‌ای برای رشد تدریجی درجه‌ی تعریف سیستم و پیاده‌سازی آن در حالی که درجه‌ی ریسک آن کاهش می‌یابد. دیگری، مجموعه‌ای از *نقاط عطف لنگری*[1] برای حصول اطمینان از تعهد طرف‌های ذی‌نفع به راهکارهای رضایت‌بخش و امکان‌پذیر.

[1] anchor milestones

با استفاده از مدل مارپیچی، نرم‌افزار به‌صورت یک سری نگارش‌های تکاملی توسعه می‌یابد. طی نخستین دورهای تکرار، نگارش تکاملی، ممکن است یک مدل کاغذی یا یک نمونه‌ی اولیه باشد. طی تکرارهای بعدی، هر بار نسخه‌ی کامل‌تری از سیستم، مهندسی شده تولید می‌شود.

مدل مارپیچی به چند فعالیت چارچوبی تقسیم می‌شود که توسط تیم نرم‌افزار تعریف می‌شوند. برای روشن‌شدن مطلب، از فعالیت‌های چارچوبی مطرح‌شده در بخش‌های قبلی استفاده می‌کنیم[1]. هر یک از فعالیت‌های چارچوبی نشان‌گر یک قطعه از مسیر مارپیچی نشان داده شده در شکل ۷-۲ است. با شروع این فرایند تکاملی، تیم نرم‌افزاری فعالیت‌هایی را اجرا می‌کند در یک دور از مارپیچ در جهت ساعتگرد تعیین و از مرکز شروع می‌شود. ریسک (فصل ۲۸) با طی‌کردن هر دور در نظر گرفته می‌شود. نقاط عطف لنگری- تلفیقی از محصولات کاری و شرایطی که در مسیر مارپیچی برقرار می‌شود- در هر دور از مارپیچ مورد توجه قرار می‌گیرد.



شکل ۷-۲ یک مدل مارپیچی متداول.

نخستین مدار مارپیچ ممکن است منجر به ایجاد مشخصه‌ای از محصول گردد، ولی عبورهای بعدی در اطراف مارپیچ برای ایجاد یک نمونه‌ی اولیه و سپس نسخه‌های پیچیده‌تری از نرم‌افزار به‌کار می‌رود. هر بار گذر از ناحیه برنامه‌ریزی، منجر به تنظیم دوباره طرح پروژه می‌شود. هزینه و زمان‌بندی براساس بازخورد حاصل از ارزیابی مشتری تنظیم می‌شود. به‌علاوه، مدیر پروژه تعداد تکرارهای موردنیاز برای کامل‌شدن نرم‌افزار را تنظیم می‌کند.

برخلاف سایر مدل‌های فرایند کلاسیک که با تحویل نرم‌افزار پایان می‌یابند، مدل مارپیچی را می‌توان طوری تطبیق داد که در سرتاسر عمر نرم‌افزار کامپیوتری قابل به‌کارگیری باشد. بنابراین، مدار اول حول مارپیچ نشان‌گر یک «پروژه‌ی توسعه‌ی مفهوم» است که از مرکز مارپیچ آغاز می‌شود و آن‌قدر تکرار می‌شود[2] تا توسعه‌ی مفهوم کامل شود. اگر قرار باشد این مفهوم در یک محصول واقعی

نکته‌ی کلیدی

مدل مارپیچی را می‌توان طوری تطبیق داد که در سرتاسر چرخه‌ی حیات یک برنامه‌ی کاربردی، از توسعه مفاهیم گرفته تا نگهداری، قابل استفاده باشد.

مرجع وب

اطلاعات مفیدی درباره مدل مارپیچی را می‌توان در وب‌سایت زیر یافت:
www.sei.cmu.edu/publications/documents/00.reports/00sr008.html

[1] مدل مارپیچی که در این بخش بحث شد، شکل تغییریافته‌ای از مدل بوهم است. برای اطلاعات بیشتر درباره مدل مارپیچی اولیه، [Boe88] را ببینید. برای بحث‌های جدیدتر درباره مدل مارپیچی بوهم [Boe98] را ببینید.

[2] پیکان‌هایی که روی محور جداکننده ناحیه اعزام از ناحیه ارتباطات قراردارند و به طرف داخل مارپیچ اشاره دارند، نشان‌گر توان بالقوه برای تکرار محلی در راستای مسیر یکسانی از مارپیچ‌اند.

SafeHome

انتخاب یک مدل فرایند، بخش ۲

صحنه: اتاق کنفرانس گروه مهندسی نرم‌افزار در شرکت CPI، شرکتی (تخیلی) که محصولات خانگی و تجاری تولید می‌کند.

نقش‌آفرینان: لی وارن، مدیر مهندسی؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ وینود و جیمی، اعضای تیم مهندسی نرم‌افزار.

گفتگوها: [داگ گزینه‌های فرایندهای تکاملی را شرح می‌دهد.]

جیمی: حالا چیزی دیدم که خوشم آمد. روش «افزایشی» معقول به‌نظر می‌رسد و من جداً این مدل مارپیچی را دوست دارم. باعث می‌شود که واقعی به‌نظر برسد.

وینود: موافقم. ما یک افزایش را تحویل مشتری می‌دهیم و از بازخوردش چیزهایی دستگیرمان می‌شود؛ دوباره برنامه‌ریزی می‌کنیم و افزایش بعدی را تحویل می‌دهیم. می‌توانیم به سرعت یک چیزی را روانه‌ی بازار کنیم و بعد با هر نسخه یا افزایش جدید، قابلیت‌ها را زیادتر کنیم.

لی: صبر کن ببینم داگ. تو می‌گویی در هر دور، باید دوباره برنامه‌ریزی کنیم؟ این خیلی جالب نیست. ما یک برنامه‌ریزی و یک زمان‌بندی می‌خواهیم و باید به آن پایبند باشیم.

داگ: این طرز تفکر دیگر قدیمی است، لی. همان‌طور که جیمی گفت، باید واقعی باشد. من قول می‌دهم که تغییر برنامه‌ریزی به موازاتی که چیزهای بیشتری دستگیرمان می‌شود و انجام تغییرات درخواست می‌شود، بهتر است. این طوری کار واقع‌بینانه‌تر انجام می‌شود. اگر برنامه‌ریزی واقعیت را منعکس نکند، به چه دردی می‌خورد؟

لی (اخم می‌کند): فکر می‌کنم درست می‌گویی، ولی مدیران ارشد خوششان نخواهد آمد. آنها یک برنامه‌ریزی ثابت می‌خواهند.

داگ (با لبخند): پس باید یک دوره بازآموزی برایشان بگذاریم رفیق.

تجلی پیدا کند، فرایند از طریق مکعب بعدی (نقطه بعدی ورود به پروژه در بسط محصول جدید) پیش می‌رود و یک «پروژه‌ی توسعه‌ی جدید» آغاز می‌شود. محصول جدید از طریق چند تکرار حول مارپیچ و با دنبال کردن مسیری تکامل می‌یابد که نواحی روشن‌تری از ناحیه مرکزی را مرزبندی می‌کنند. در اصل، هنگامی که مارپیچ به این شیوه مشخص می‌شود، تا پایان کار نرم‌افزار فعال باقی می‌ماند. زمانی فرا می‌رسد که فرایند، غیرفعال می‌شود. ولی هرگاه تغییری آغاز شود، فرایند در نقطه ورودی مناسب (مثلاً بهبود محصول) آغاز می‌شود.

مدل مارپیچی یک روش واقع‌گرا برای توسعه‌ی نرم‌افزارها و سیستم‌هایی در مقیاس انبوه است. از آنجا که نرم‌افزار به موازات پیشرفت فرایند، تکامل می‌یابد، سازنده و مشتری در هر سطح تکامل، ریسک‌ها را بهتر درک کرده به آن واکنش نشان می‌دهند. مدل مارپیچی از ساخت نمونه‌ی اولیه به‌عنوان راهکاری برای کاهش ریسک استفاده می‌کند، ولی مهم‌تر آنکه سازنده را قادر می‌سازد تا روش ساخت نمونه‌ی اولیه را در هر مرحله از تکامل محصول به‌کار بندد. این مدل همان روش مرحله‌ای پیشنهادشده توسط چرخه‌ی حیات کلاسیک را حفظ می‌کند، ولی آن را با یک چارچوب تکراری همراه می‌کند که جهان واقعی را واقعی‌تر منعکس می‌کند. در مدل مارپیچی، در نظر گرفتن ریسک‌های

اندرز

اگر مدیریت شما توسعه‌ی با بودجه‌ی ثابت را دوست دارد (که عموماً ایده‌ی جالبی نیست) مدل مارپیچی می‌تواند مشکل‌آفرین شود. با کامل‌شدن هر دور از مارپیچ، هزینه‌ی پروژه مورد بازبینی قرار می‌گیرد.

فنی در همه‌ی مراحل، ضروری است و اگر به‌طور مناسب به‌کار برده شود، باید ریسک را پیش از آنکه مشکل‌آفرین شوند، کاهش دهد.

ولی همانند الگوهای دیگر، این مدل نیز علاج همه‌ی دردها نیست. ممکن است به سختی بتوان مشتری را قانع کرد (به‌ویژه در شرایط قرارداد) که روش تکاملی قابل کنترل است. این مدل، مهارت ارزیابی خطر فراوانی را طلب می‌کند، و برای موفقیت، بر همین مهارت متکی است. اگر یک خطر عمده کشف و اداره نشود، بدون شک مشکلاتی به بار خواهد آمد.

> «من فقط این قدر، دورم و تنها فرداست که راهنمای من خواهد بود»
>
> دیو ماتیوس بند

۲-۳-۴ مدل توسعه‌ی همروند

مدل توسعه‌ی همروند[1] که گاه از آن با عنوان *مهندسی همروند* نیز یاد می‌شود، به تیم نرم‌افزار این امکان را می‌دهد عناصر تکراری و همروند هر کدام از مدل‌های فرایند توصیف شده در این فصل را ارائه نماید. برای مثال، فعالیت مدل‌سازی تعریف شده برای مدل مارپیچی با توجه به وظایف زیر قابل حصول است: ساخت نمونه‌ی اولیه، تحلیل، و طراحی.[2]

شکل ۸-۲ طرحی از یک فعالیت با مدل توسعه‌ی همروند ارائه می‌دهد. این فعالیت – **مدل‌سازی** – ممکن‌است در هر زمان مفروض، در یکی از حالت‌های[3] ذکر شده باشد. به‌طور مشابه، فعالیت‌های دیگر (مانند **ارتباطات** یا **ساخت**) را می‌توان به‌شیوه‌ای مشابه نمایش داد. همه‌ی فعالیت‌ها به‌صورت همروند وجود دارند ولی در حالت‌های متفاوت قرار می‌گیرند.

> **اندرز**
>
> مدل هم‌زمان غالباً برای پروژه‌های مهندسی‌ای مناسب است که تیم‌های مهندسی متقاوتی در آن شرکت دارند.

برای مثال، در ابتدای یک پروژه، فعالیت برقراری ارتباط (که در شکل نشان داده نشده است) نخستین تکرار خود را به پایان رسانده است و در حالت «**انتظار تغییرات**» قرار دارد. فعالیت مدل‌سازی (که هنگام کامل‌شدن ارتباط اولیه با مشتری در حالت **غیرفعال** قرار داشت) اکنون دستخوش گذار به حالت **تحت توسعه** می‌شود. ولی اگر مشتری متذکر شود که تغییراتی در خواسته‌ها باید صورت پذیرد، فعالیت تحلیل از حالت «تحت توسعه» به حالت «انتظار تغییرات» می‌رود.

مدل توسعه‌ی همروند، یک سری رویداد تعریف می‌کند که باعث گذار از حالتی به حالت دیگر برای هر یک از فعالیت‌های مهندسی نرم‌افزار می‌شوند. برای مثال، طی اولین مراحل طراحی (یکی از کنش‌های اصلی در مهندسی نرم‌افزار که طی فعالیت مدل‌سازی انجام می‌شود)، یک ناسازگاری در مدل تحلیل کشف می‌شود. این باعث تولید رویداد *تصحیح مدل تحلیل* می‌شود که گذار از کنش تحلیل خواسته‌ها را از حالت **انجام‌شده** به حالت **انتظار تغییرات** سبب می‌شود.

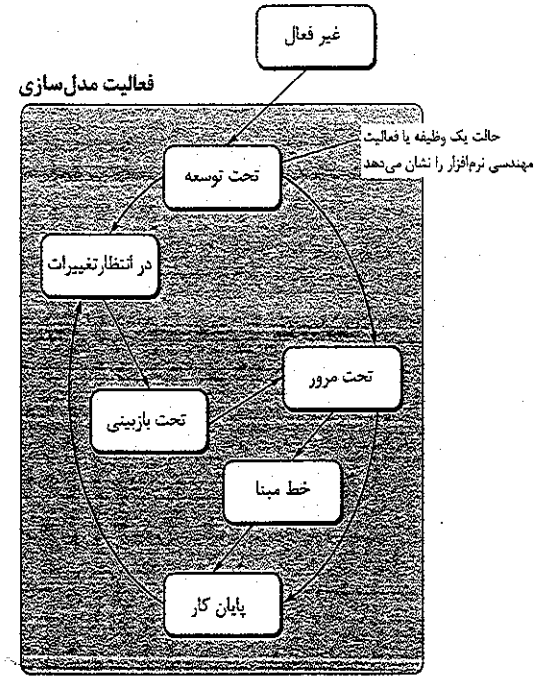
مدل‌سازی همروند برای انواع روش‌های توسعه‌ی نرم‌افزار قابل استفاده است و تصویری صحیح از وضعیت فعلی یک پروژه فراهم می‌سازد. مهندسی نرم‌افزار به‌جای محدودکردن فعالیت‌ها، کنش‌ها و وظایف به یک سری رویدادها، شبکه‌ای از فرایندها را تعریف می‌کند. رویدادهای ایجادشده در یک نقطه از این شبکه‌ی فرایندها، گذار در میان حالت‌ها را آغاز می‌کند.

> «هر فرایندی در سازمان شما یک مشتری دارد و بدون مشتری، فرایند شما هدفی ندارد.»
>
> و. دانیل هانت

[1] concurrent model

[2] لازم به ذکر است که تحلیل و طراحی، وظایفی پیچیده‌اند که نیاز به بحث اساسی دارند. در بخش دوم این کتاب به تفصیل به این مباحث خواهیم پرداخت.

[3] حالت، رفتاری از سیستم است که از بیرون قابل مشاهده باشد.



شکل ۸-۲ یک عنصر از فرایند همروند.

۲-۳-۵ سخن آخر درباره‌ی فرایندهای تکاملی

قبلاً گفتیم که مشخصه‌ی نرم‌افزارهای مدرن، تغییر پیوسته، در فواصل زمانی بسیار به هم فشرده و با تأکید بسیار بر رضایت مشتری-کاربر است. در بسیاری موارد، زمان رساندن محصول به بازار، مهمترین خواسته‌ی مدیریتی است. اگر زمان مقرر برای ارائه به بازار از دست برود، خود پروژه‌ی نرم‌افزاری ممکن است دیگر بی‌معنا شود[1].

تصور می‌شد مدل‌های فرایند تکاملی این مشکلات را برطرف سازند و با این وجود، آنها نیز به‌عنوان طبقه‌ای عمومی از مدل‌های فرایند، نقطه‌ضعف‌هایی دارند. نوگریا و همکاران او [Nog00] این نقاط ضعف را چنین خلاصه کرده‌اند:

> به رغم مزایای غیرقابل تردید فرایندهای نرم‌افزاری تکاملی، دغدغه‌هایی نیز وجود دارد. نخست اینکه تهیه‌ی نمونه‌ی اولیه [و سایر فرایندهای تکاملی پیچیده‌تر] به دلیل قطعی نبودن تعداد چرخه‌های لازم برای ساخته‌شدن محصول، برای برنامه‌ریزی پروژه ایجاد مشکل می‌کند. اکثر تکنیک‌های برآورد و مدیریت پروژه برپایه چیدمان خطی فعالیت‌ها استوارند، لذا به‌خوبی در این الگو نمی‌گنجند.

[1] به‌هرحال، لازم به ذکر است که اول بودن در دستیابی به بازار، تضمینی بر موفقیت نیست. در واقع، محصولات نرم‌افزاری موفق فراوانی وجود دارند که دومین یا حتی سومین محصول برده‌اند که وارد بازار شده‌اند (و از اشتباهات اسلاف خود درس گرفته‌اند.)

دوم اینکه فرایندهای تکاملی حداکثر سرعت تکامل را تعیین نمی‌کنند. اگر تکامل بیش از حد سریع رخ دهد، بدون اینکه زمان آسایشی داشته باشند، فرایند به‌طور قطع به آشوب کشیده خواهد شد. از طرف دیگر، اگر سرعت بیش از حد کم باشد، بهره‌وری تحت تأثیر قرار خواهد گرفت...

سوم، در فرایندهای نرم‌افزاری، انعطاف‌پذیری (flexibility) و بسط‌پذیری (extensibility) باید بیش از کیفیت بالا مورد توجه قرار گیرد. این تأکید قدری ترسناک به‌نظر می‌رسد، ولی ما باید به سرعت توسعه‌ی نرم‌افزار، اولویتی بیش از خطای صفر بدهیم. پیش رفتن در کار توسعه به‌منظور رسیدن به سطح بالایی از کیفیت می‌تواند به تحویل دیرهنگام محصول منجر شود و بازار هدف از دست برود. این جابجایی در الگو نتیجه‌ی رقابت در آستانه‌ی آشوب است.

در حقیقت، یک فرایند نرم‌افزار که در آن بر انعطاف‌پذیری، بسط‌پذیری و سرعت توسعه بیش از کیفیت بالا تأکید می‌شود، ناعاقلانه به‌نظر می‌رسد. و در عین حال، چند کارشناس معتبر در زمینه‌ی مهندسی نرم‌افزار آن را پیشنهاد کرده‌اند (مانند [You95]، [Bac97]).

هدف مدل‌های تکاملی، توسعه‌ی نرم‌افزارهایی با کیفیت بالا[1] به‌شیوه‌ای افزایشی یا تعاملی است، ولی استفاده از یک فرایند تکاملی برای تأکید ورزیدن بر انعطاف‌پذیری، بسط‌پذیری و سرعت توسعه، امکان‌پذیر است. چالش برای تیم‌های نرم‌افزاری و مدیران آنها برقراری موازنه میان این پارامترهای حیاتی پروژه و محصول و رضایت مشتری (هدف نهایی کیفیت نرم‌افزار) است.

۲-۴ مدل‌های فرایند تخصص‌یافته

مدل‌های فرایند تخصص‌یافته، شامل بسیاری از ویژگی‌های یک یا چند مدل سنتی ارائه شده در بخش‌های پیشین می‌شوند. ولی، این مدل‌ها را معمولاً هنگامی به‌کار می‌برند که یک روش مهندسی تخصصی یا روشی با مشخصات دقیق انتخاب می‌شود.[2]

۲-۴-۱ توسعه مبتنی بر مؤلفه‌ها (Component-Based Development)

مؤلفه‌های نرم‌افزاری آماده (COTS)، توسط عده‌ای از فروشندگان این مؤلفه‌ها توسعه داده می‌شوند، عملکرد مورد نظر را با واسط‌هایی مناسب فراهم می‌آورند، به‌طوری که مؤلفه را می‌توان به‌خوبی در سیستم در حال ساخت الحاق کرد. توسعه‌ی مبتنی بر مؤلفه‌ها بسیاری از خصوصیات مدل مارپیچی را در بر می‌گیرد. ماهیتی تکاملی دارد [Nie92] و برای ایجاد نرم‌افزار به رویکردی تکراری نیاز دارد. به‌هرحال، در مدل توسعه مبتنی بر مؤلفه‌ها، برنامه‌های کاربردی از به هم پیوستن مؤلفه‌های نرم‌افزار آماده ساخته می‌شوند.

فعالیت‌های مدل‌سازی و ساخت با شناسایی مؤلفه‌های کاندیدا آغاز می‌شود. این مؤلفه‌ها را

[1] در این حیطه، کیفیت نرم‌افزاری تعریفی کاملاً گسترده دارد و نه تنها شامل رضایت مشتری بلکه انواع ملاک‌های فنی می‌شود که در فصل‌های ۱۴ و ۱۶ بحث خواهد شد.

[2] در برخی موارد این مدل‌های فرایندی تخصص‌یافته را شاید بتوان به صورت مجموعه‌ای از تکنیک‌ها برای دستیابی به یک هدف خاص در توسعه نرم‌افزار مشخص کرد. ولی آنها خود نیز به معنای یک فرایند هستند.

می‌توان به‌صورت پیمانه‌های نرم‌افزاری سنتی یا کلاس‌ها و پکیج‌های شیءگرا طراحی کرد[1]. فن‌آوری مورد استفاده در ایجاد مؤلفه‌ها هر چه که باشد، مدل توسعه مبتنی بر مؤلفه‌ها شامل مراحل زیر می‌شود (که با استفاده از رویکردی تکاملی پیاده‌سازی می‌شود):

۱. محصولات مبتنی بر مؤلفه‌ی موجود، از نظر دامنه‌ی کاربرد مورد نظر بررسی و ارزیابی می‌شوند.
۲. مسائل مربوط به انسجام مؤلفه‌ها در نظر گرفته می‌شوند.
۳. برای چیدمان مؤلفه‌ها یک معماری نرم‌افزار طراحی می‌شود.
۴. مؤلفه‌ها در این معماری قرار داده می‌شوند.
۵. آزمونی جامع برای حصول اطمینان از عملکرد درست، به عمل می‌آید.

مدل توسعه‌ی مبتنی بر مؤلفه‌ها، استفاده‌ی مجدد از نرم‌افزار را میسر می‌سازد و قابلیت استفاده‌ی مجدد چند مزیت سنجش‌پذیر در اختیار مهندسان نرم‌افزار قرار می‌دهد. اگر استفاده‌ی مجدد از مؤلفه‌ها فرهنگ‌سازی شود، تیم مهندسی نرم‌افزار شما می‌تواند به کاهش زمان چرخه‌ی توسعه و نیز کاهش هزینه‌های پروژه دست پیدا کند. توسعه‌ی مبتنی بر مؤلفه‌ها را در فصل ۱۰ به تفصیل بیشتر بحث خواهیم نمود.

۲-۴-۲ مدل روش‌های رسمی[2]

مدل روش‌های رسمی شامل مجموعه‌ای از فعالیت‌ها می‌شود که به مشخص‌کردن ریاضی و رسمی نرم‌افزار کامپیوتری منجر می‌شود. روش‌های رسمی، مهندس نرم‌افزار را قادر می‌سازند تا با اِعمال یک نظم ریاضی شدید، سیستم کامپیوتری را مشخص کند، بسط دهد و وارسی کند. شکل دیگری از این روش، که *مهندسی نرم‌افزار اتاق تمیز*[3] نامیده می‌شود [Mil87, Dye92] در حال حاضر توسط برخی سازمان‌های نرم‌افزارسازی به‌کار می‌رود.

هنگامی که روش‌های رسمی (فصل ۲۱) در اثنای توسعه‌ی نرم‌افزار به‌کار برده می‌شوند، راهکاری برای حذف بسیاری از مشکلات فراهم می‌آورند که غلبه بر آنها با استفاده از الگوهای مهندسی دیگر، دشوار است. ابهام، ناقص‌بودن و ناسازگاری را می‌توان راحت‌تر کشف و تصحیح کرد، نه از طریق بازبینی خاص بلکه از طریق به‌کارگیری تحلیل ریاضی. هنگامی که روش‌های رسمی در اثنای طراحی به‌کار برده می‌شوند، به‌عنوان مبنایی برای وارسی برنامه عمل کرده از این رو مهندس نرم‌افزار را قادر به کشف و تصحیح خطاهایی می‌سازند که ممکن بود در غیر این صورت مخفی بمانند.

«مدل روش‌های رسمی» گرچه چندان عمومیت نخواهد یافت، نویدبخش نرم‌افزاری عاری از نقص است. با این حال، ملاحظات مربوط به قابلیت اجرای آن در محیط‌های تجارتی چنین اعلام شده است.

- توسعه‌ی مدل‌های رسمی در حال حاضر بسیار وقت‌گیر و پرهزینه است.

[1] مفاهیم شیءگرا در پیوست ۲ بحث خواهد شد و در سرتاسر قسمت دوم این کتاب از آنها استفاده خواهیم کرد. در این حیطه، کلاس شامل مجموعه‌ای از داده‌ها و روال‌هایی می‌شود که آن داده‌ها را پردازش می‌کنند. یک پکیج از کلاس‌ها، مجموعه‌ای از کلاس‌هاست که برای دست‌یابی به نتیجه‌ی نهایی با هم کار می‌کنند.

[2] formal methods model

[3] cleanroom software engineering

- از آنجا که تعداد معدودی از نرم‌افزارسازان دارای زمینه‌ی لازم برای اجرای روش‌های رسمی هستند، آموزش گسترده‌ای مورد نیاز است.
- استفاده از مدل‌ها به‌عنوان راهکار ارتباطی با مشتریانی که دید فنی ندارند، دشوار است.

نظر به این ملاحظات، روش‌های رسمی احتمالاً در میان نرم‌افزارنویسانی هوادار پیدا می‌کند که باید نرم‌افزارهای ایمنی- حیاتی (safety-critical) (مثلاً نرم‌افزارهای دستگاه‌های پزشکی و هوافضا) بسازند یا در میان آنهایی که در صورت بروز خطا در نرم‌افزار دستخوش زیان‌های اقتصادی کلان می‌شوند.

اگر روش‌های رسمی قادر به نشان‌دادن درستی نرم‌افزار هستند، چرا از آن‌ها به‌طور گسترده استفاده نمی‌شود؟

۳-۴-۲ توسعه‌ی نرم‌افزار به روش جنبه‌گرا (Aspect Oriented)

هر فرایند نرم‌افزار که انتخاب شود، سازندگان نرم‌افزارهای پیچیده، مجموعه‌ای از ویژگی‌ها، عملکردها و محتویات اطلاعاتی متمرکز را پیاده‌سازی می‌کنند. این خصوصیات نرم‌افزاری متمرکز به‌صورت مؤلفه‌هایی (مثلاً کلاس‌های شیءگرا) مدل‌سازی و سپس در حیطه‌ی یک معماری سیستم بنا می‌شوند. با پیچیده‌ترشدن سیستم‌های کامپیوتری مدرن، دغدغه‌های خاصی- خواص مورد نیاز مشتری یا مسائل فنی- کل معماری را در بر می‌گیرد. برخی از این دغدغه‌ها خواص سطح بالای سیستم (نظیر امنیت و تحمل‌پذیری خطا) هستند. عده‌ای دیگر بر وظایف سیستم تأثیر می‌گذارند (مثل إعمال قواعد تجاری) در حالی که عده‌ای هم سیستماتیک هستند (مانند هم‌زمان‌سازی وظایف یا مدیریت حافظه).

هنگامی که این دغدغه‌ها در وظایف، ویژگی و اطلاعات سیستمی با یکدیگر تلاقی می‌کنند، غالباً از آنها به‌عنوان دغدغه‌های متلاقی (crossing concerns) یاد می‌شود. خواسته‌های جنبه‌گرا، آن دسته از دغدغه‌های متلاقی را تعریف می‌کنند که بر معماری نرم‌افزار تأثیر می‌گذارند. از توسعه‌ی نرم‌افزار به روش جنبه‌گرا (AOSD)، که از آن غالباً به‌عنوان برنامه‌نویسی جنبه‌گرا نیز یاد می‌شود، یک الگوی مهندسی نسبتاً جدید است که رویکردی فرایندی و روش‌شناختی برای تعریف، مشخص‌سازی، طراحی و ساخت جنبه‌ها ارائه می‌دهد-«سازوکارهایی غیر از زیرروال‌ها و وراثت برای متمرکز ساختن بیان یک دغدغه‌ی متلاقی» [Elr01].

گراندی [Gru02] در حیطه‌ای که آن را مهندسی مؤلفه‌های جنبه‌گرا (AOCE) می‌نامد، درباره جنبه‌ها بیشتر بحث می‌کند:

> AOCE از مفهوم برش‌های افقی (horizontal slices) در مؤلفه‌های نرم‌افزاری استفاده می‌کند که به‌طور عمودی تجزیه شده‌اند و «جنبه» نام دارند؛ هدف از این مفهوم، مشخص‌کردن خواص عملیاتی و غیر عملیاتی متلاقی مؤلفه‌هاست. جنبه‌های متداول و سیستماتیک عبارتند از واسطه‌ها، همکاری‌ها، توزیع، دوام، مدیریت حافظه، پردازش تراکنش‌ها، امنیت و غیره. مؤلفه‌ها ممکن است به یک یا چند مورد از «جزئیات جنبه» مرتبط با یک جنبه‌ی خاص نیاز داشته باشند یا آن را فراهم آورند؛ برخی از این جنبه ها عبارتند از سازوکار مشاهده و نوع واسط (جنبه‌های واسط کاربری)؛ تولید رویدادها، انتقال و دریافت (جنبه‌های توزیع)؛ نگهداری/بازیابی داده‌ها و شاخص‌سازی (indexing) (جنبه‌های پایداری)؛ قوانین احراز هویت، کدگذاری و دستیابی (جنبه‌های امنیتی)؛ اتمی‌بودن تراکنش‌ها، کنترل هم‌زمانی و راهبرد ایجاد کارنامه (logging strategy) (جنبه‌های تراکنشی) و غیره. جزئیات هر جنبه، چند خاصیت مرتبط با ویژگی‌های عملیاتی و/یا غیرعملیاتی جزئیات آن جنبه دارد.

مرجع وب

مجموعه بزرگی از منابع و اطلاعات مربوط به AOP را می‌توان در aosd.net یافت.

نکته‌ی کلیدی

AOSD به تعریف وجه‌هایی می‌پردازد که دغدغه‌های مشتریان را بیان می‌کند و بر چند قابلیت و ویژگی از سیستم تأثیر می‌گذارند.

ابزارهای نرم‌افزاری

مدیریت فرایند

هدف: کمک به تعریف، اجرا و مدیریت مدل‌های فرایند تجویزی.

مکانیک: تیم یا سازمان نرم‌افزاری به کمک ابزارهای مدیریت فرایند می‌تواند یک مدل فرایند کامل (فعالیت‌های چارچوبی، کنش‌ها، وظایف، تضمین کیفیت، نقاط عطف و محصولات کاری) را تعریف کند. به‌علاوه، این ابزارها راهنمایی برای کارهای فنی مهندسان نرم‌افزار و الگویی برای مدیرانی که می‌خواهند فرایند نرم‌افزار را کنترل کنند، فراهم می‌سازد.

چند ابزار نمونه:

GDPA، یک مجموعه ابزار تحقیقاتی برای تعریف فرایند است و در دانشگاه برمن آلمان توسعه یافته است (www.informatik.uni-bremen.de/uniform/gdpa/home.htm) و آرایه‌ای گسترده از مدل‌سازی فرایند و وظایف مدیریتی فراهم می‌سازد.

SpeeDev، که توسط شرکت SpeeDev (www.speedev.com) ارائه شده است و شامل مجموعه‌ای از ابزارها برای تعریف فرایند، مدیریت خواسته‌ها، رفع مشکلات، برنامه‌ریزی پروژه و کنترل می‌شود.

Provision BPMx که توسط Proforma (www.proformacorp.com) ارائه شده است و نماینده‌ی بسیاری از ابزارهاست که به تعریف فرایند و خودکارسازی جریان کاری کمک می‌کنند.

فهرست مفیدی از چندین ابزار مرتبط با فرایند نرم‌افزار را می‌توانید در صفحه‌ی زیر بیابید:
www.processwave.net/Links/tool.links.htm

هنوز فرایند جنبه‌گرای متمایزی به بلوغ نرسیده است، ولی این احتمال وجود دارد که چنین فرایندی خصوصیات هر دو نوع مدل تکاملی و همروند را داشته باشد. مدل تکاملی برای شناسایی و ساخت جنبه‌ها مناسب است. ماهیت موازی در توسعه‌ی همروند نیز ضرورت دارد؛ زیرجنبه‌ها مستقل از مؤلفه‌های نرم‌افزار مهندسی می‌شوند و در عین حال، جنبه‌ها تأثیری مستقیم بر این مؤلفه‌ها دارند. از این رو، بر قراری ارتباط ناهم‌زمان میان فعالیت‌های نرم‌افزاری به‌کاررفته در مهندسی و ساخت جنبه‌ها و مؤلفه‌ها اهمیتی اساسی دارد.

برای بحث جامعی درباره توسعه‌ی نرم‌افزارهای جنبه‌گرا می‌توانید به کتاب‌هایی رجوع کنید که در همین زمینه نگاشته شده‌اند. در صورت علاقه می‌توانید به [Saf08]، [Cla05]، [Jac04]، [Gra03] نگاهی بیندازید.

۲-۵ فرایند یکپارچه (Unified Process)

ایوار جیکابسون، گرادی بوچ و جیمز رومباف [Jac99] در کتاب خود با عنوان فرایند یکپارچه، طی بیانات زیر، نیاز به یک فرایند نرم افزاری «مبتنی بر "use case"، معماری، تکرار و افزایش» را مورد بحث قرار می‌دهند.

امروزه در نرم‌افزار، سیستم‌های پیچیده‌تر و بزرگ‌تر بیشتر مورد نظرند. این امر تا حدی از این واقعیت ناشی می‌شود که هر ساله بر قدرت کامپیوترها افزوده می‌شود و در نتیجه، انتظارات کاربران نیز از آنها بیشتر می‌شود. این روند از کاربرد فزاینده‌ی اینترنت برای تبادل انواع اطلاعات نیز تأثیر پذیرفته است... وقتی که در می‌یابیم یک محصول از نسخه‌ای به نسخه‌ی دیگر چقدر قابلیت بهبود دارد، اشتهای ما برای نرم‌افزارهایی با پیچیدگی بیش از پیش رشد می‌کند . ما نرم‌افزارهایی می‌خواهیم که بهتر بر نیازهای ما منطبق باشند، ولی این به نوبه خود فقط باعث پیچیدگی بیشتر می‌شود. به‌طور خلاصه، بیشتر می‌خواهیم.

فرایند یکپارچه (UP) از جهاتی تلاش برای گرد هم آوردن بهترین ویژگی‌ها و خصوصیات مدل‌های فرایند سنتی است، ولی آنها را به‌شیوه‌ای مشخص می‌کند که بسیاری از بهترین اصول توسعه‌ی نرم‌افزار چابک (فصل ۳) را پیاده‌سازی کند. در فرایند یکپارچه اهمیت برقراری ارتباط با مشتریان و روش‌های ساده و روان برای توصیف دیدگاه مشتریان (use case[1]) یک سیستم به‌خوبی درک می‌شود. در این فرایند بر اهمیت نقش معماری نرم‌افزار تأکید می‌شود و به معمار کمک می‌شود تا به اهداف درستی از قبیل قابلیت درک، اتکا بر تغییرات آتی و استفاده‌ی مجدد توجهی خاص داشته باشد. [Jac99]. در این فرایند، یک جریان فرایند مبتنی بر تکرار و افزایشی پیشنهاد می‌شود و یک حس و حال تکاملی را ایجاد می‌کند که در توسعه‌ی نرم‌افزارهای مدرن ضروری است.

۲-۵-۱ تاریخچه

اوایل دهه ۱۹۹۰، جیمز رومباف [Rum91]، گرادی بوچ [Boo94] و ایوار جیکابسون [Jac92] کار روی یک «روش یکپارچه» را آغاز کردند که بهترین ویژگی‌های هر کدام از روش‌های طراحی و تحلیل شیءگرا را تلفیق می‌کرد و ویژگی‌هایی از سایر کارشناسان (مثلاً [wir90]) در مدل‌سازی شیءگرا را بر آن منطبق می‌ساخت. نتیجه، زبان مدل‌سازی یکپارچه (UML) است که حاوی یک نمادگذاری قدرت‌مند برای مدل‌سازی و توسعه‌ی سیستم‌های شیءگراست. تا سال ۱۹۹۷، UML به یک استاندارد صنعتی غیر رسمی برای توسعه‌ی نرم‌افزارهای شیءگرا تبدیل شد.

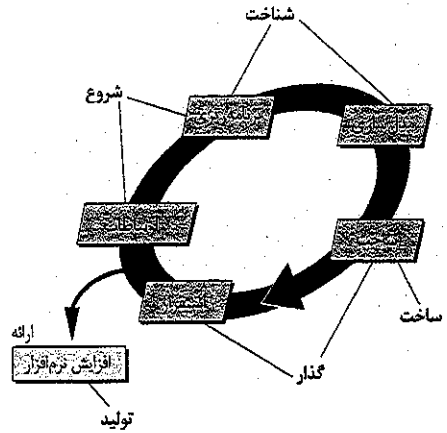
از UML در سرتاسر قسمت دوم این کتاب برای نمایش دادن خواسته‌ها و نیز مدل‌های طراحی استفاده می‌شود. در پیوست ۱، یک خودآموز مقدماتی برای خوانندگان نا آشنا با قواعد مدل‌سازی و نمادگذاری UML ارائه شده است. نمایش جامعی از UML به بهترین نحو در کتاب‌هایی با همین عنوان ارائه شده است که در پیوست ۱ چند مورد از این کتاب‌ها معرفی شده است.

UML فن‌آوری لازم برای پشتیبانی از مهندسی نرم‌افزار شیءگرا را فراهم ساخت، ولی چارچوب فرایندی لازم را برای راهنمایی تیم‌های پروژه در به‌کارگیری این فن‌آوری ارائه نداد. طی چند سال بعد، جیکابسون، رومباف و بوچ، فرایند یکپارچه را توسعه دادند که چارچوبی برای مهندسی نرم‌افزار شیءگرا با استفاده از UML است. امروزه، فرایند یکپارچه (UP) و UML به‌وفور در انواع مختلفی از پروژه‌های شیءگرا به‌کار گرفته می‌شوند. مدل مبتنی بر تکرار و افزایشی‌ای که UP پیشنهاد می‌کند، برای برآوردن نیازهای پروژه‌ای خاص، قابل انطباق بوده و باید هم باشد.

[1] *use case* (فصل ۵) به متن روایی یا گرته‌ای گفته می‌شود که ویژگی یا قابلیتی از سیستم را از دیدگاه کاربر توصیف می‌کند. use case توسط کاربر نوشته می‌شود و به‌عنوان مبنایی برای ایجاد مدل جامعی از خواسته‌ها استفاده می‌شود.

۲-۵-۲ مراحل فرایند یکپارچه[1]

پیش‌تر در این فصل، پنج فعالیت چارچوبی کلی را مورد بحث قرار دادیم و استدلال کردیم که از آنها می‌توان برای توصیف هر مدل فرایند نرم‌افزاری استفاده کرد. فرایند یکپارچه نیز از این قاعده مستثنی نیست. در شکل ۹-۲، مراحل UP و ارتباط آنها با فعالیت‌های بحث شده در فصل ۱، تصویر شده است.



شکل ۹-۲ فرایند یکپارچه

نکته‌ی کلیدی
مراحل UP از نظر مقاصدی که دنبال می‌کند، فعالیت‌های چارچوبی کلی تعریف شده در این کتاب شباهت دارند.

مرحله‌ی آغازین در UP شامل هر دو فعالیت برقراری ارتباط با مشتریان و برنامه‌ریزی می‌شود. از طریق همکاری با طرف‌های ذی‌نفع، خواسته‌های تجاری برای نرم‌افزار شناسایی می‌شود؛ یک معماری تقریبی برای سیستم پیشنهاد می‌شود و برنامه‌ریزی بنیادی برای ماهیت تکراری و افزایشی پروژه‌ی آتی توسعه داده می‌شود. خواسته‌های تجاری بنیادی از طریق یک مجموعه use case مقدماتی (فصل ۵) توصیف می‌شود که نشان می‌دهند کدام ویژگی‌ها و وظایف، مطلوب هر دسته از تمایلات کاربران است. معماری در این نقطه، چیزی نیست جز یک نقشه‌ی آزمایشی از زیرسیستم‌های اصلی و وظایف و ویژگی‌هایی که آنها را تشکیل می‌دهند. بعداً این معماری پالایش خواهد شد و به مجموعه‌ای از مدل‌ها بسط داده خواهد شد که سیستم را از دیدگاه‌های مختلف نشان خواهند داد. در برنامه‌ریزی، منابع شناسایی می‌شوند، خطرات اصلی ارزیابی می‌شوند، یک برنامه‌ی زمانی تدوین می‌شود و برای مراحلی که قرار است به‌عنوان گام در حال توسعه‌ی نرم‌افزار به‌کار برده شوند، مبنایی به‌وجود می‌آید.

مرحله‌ی شناخت شامل فعالیت‌های برقراری ارتباط و مدل‌سازی در مدل فرایند کلی می‌شود (شکل ۹-۲). در این مرحله، «use case‌های» مقدماتی که به‌عنوان بخشی از مرحله‌ی آغازین ایجاد شوند، پالایش یافته و بسط داده می‌شوند؛ به‌علاوه، در این مرحله‌ی نمایش معماری نیز بسط داده می‌شود تا پنج نمای متفاوت نرم‌افزار را در برگیرد-این پنج نما عبارتند از مدل use case، مدل خواسته‌ها، مدل طراحی، مدل پیاده‌سازی و مدل استقرار. در برخی موارد، در مرحله‌ی شناخت، یک «خط مبنای معماری قابل اجرا» [Arl02] ایجاد می‌شود که «اولین برش» از سیستم قابل اجرا

[1] فرایند یکپارچه را گاهی فرایند *یکپارچه‌ی گورپا* (Rational Unified Process) نیز می‌نامند.

ارائه می‌شود.[1] خط مبنای معماری نشان‌گر ماندگاری معماری است، ولی همه‌ی ویژگی‌ها و عملکردهای لازم برای استفاده از سیستم را فراهم نمی‌آورد. به‌علاوه، طرح در اوج مرحله‌ی شناخت به دقت بازبینی می‌شود تا اطمینان حاصل شود که حوزه‌ی عملیاتی، خطرات و تاریخ تحویل در حدی منطقی باقی می‌ماند. اصلاحات روی برنامه‌ریزی غالباً در این زمان اعمال می‌شوند.

مرحله‌ی ساخت در UP، هم‌ارز فعالیت ساخت است که برای فرایند نرم‌افزار کلی تعریف شد. مرحله‌ی ساخت با استفاده از مدل معماری به‌عنوان ورودی، مؤلفه‌های نرم‌افزاری را که هر کدام از use case ها را عملیاتی می‌کنند، ایجاد می‌کند یا آنها را توسعه می‌دهد. برای دستیابی به این هدف، خواسته‌ها و مدل‌های طراحی که طی مرحله‌ی شناخت آغاز شده بودند، کامل می‌شوند تا آخرین نسخه از هر «افزایش» نرم‌افزار منعکس شود. سپس همه‌ی ویژگی‌های لازم و ضروری و عملکردها برای هر افزایش (نسخه‌ی) نرم‌افزار در کد منبع پیاده‌سازی می‌شوند. به موازاتی که مؤلفه‌ها پیاده‌سازی می‌شوند، آزمون واحدها[2] برای هر کدام طراحی و اجرا می‌شوند. به‌علاوه، فعالیت‌های انسجام بخشی (مونتاژ مؤلفه‌ها و آزمون انسجام) نیز اجرا می‌شود. موارد استفاده برای به‌دست آوردن مجموعه‌ای از آزمون‌های پذیرش به‌کار گرفته می‌شوند که پیش از شروع مرحله‌ی بعدی UP اجرا می‌شوند.

مرحله‌ی گذار (transition phase) در UP شامل آخرین مراحل در فعالیت ساخت در مدل کلی و اولین بخش از استقرار در مدل کلی (تحویل و بازخورد) می‌شود. نرم‌افزار برای آزمون بتا به کاربران نهایی داده می‌شود و بازخورد به‌دست آمده از کاربران هم نقایص و هم تغییرات لازم را گزارش می‌کند. به‌علاوه، تیم نرم‌افزار، اطلاعات پشتیبانی را (از قبیل جزوات راهنمای کاربران، دستورالعمل اشکال‌زدایی، روال‌های نصب) که برای نسخه‌ی مورد نظر لازم هستند، ایجاد می‌کند. در پایان مرحله‌ی گذار، هر «افزایش» نرم‌افزار به یک نسخه‌ی قابل استفاده تبدیل می‌شود.

مرحله‌ی تولید در UP منطبق بر فعالیت استقرار در فرایند کلی است. طی این مرحله، استفاده از نرم‌افزار، پایش می‌شود، محیط عملیاتی (زیرساخت) پشتیبانی می‌شود و گزارش نقایص و درخواست برای تغییرات، تسلیم و ارزیابی می‌شود. این احتمال هست که در همان زمان اجرای مراحل ساخت، گذار و تولید، کار روی گام بعدی نرم‌افزار نیز شروع شده باشد. این بدان معناست که پنج مرحله‌ی UP یکی پس از دیگری رخ نمی‌دهند بلکه ممکن است هم زمان با هم در جریان باشند.

یک جریان کاری مهندسی نرم‌افزار در سرتاسر فازهای UP توزیع می‌شود. در حیطه‌ی UP، جریان کاری مشابه با یک مجموعه وظایف است (که قبلاً در همین فصل شرح داده شد). یعنی، جریان کاری، وظایف لازم برای دستیابی به یک کنش مهم در مهندسی نرم‌افزار و محصولات کاری تولید شده در نتیجه‌ی انجام موفقیت آمیز این وظایف را تعیین می‌کند. لازم به ذکر است که همه‌ی وظایف تعیین شده برای یک جریان کاری در UP برای هر پروژه‌ی نرم‌افزاری اجرا نمی‌شوند، بلکه تیم، فرایند (کنش‌ها، وظایف، وظایف فرعی و محصولات کاری) را بر نیازهای خود تطبیق می‌دهد.

مرجع وب

یک بحث جالب درباره UP در حیطه‌ی توسعه‌ی چابک را می‌توان در وب‌سایت زیر یافت:

www.ambysoft.com/unifiedprocess/agileUP.html

۶-۲ مدل‌های فرایند تیمی و شخصی

بهترین فرایند نرم‌افزار، فرایندی است که به کسانی که کار می‌کنند، نزدیک باشد. اگر یک مدل فرایند

[1] توجه به این نکته حائز اهمیت است که خط مبنای معماری یک نمونه اولیه به شمار نمی‌رود از این لحاظ که کنار گذاشته نمی‌شود. در عوض، خط مبنا طی مرحله بعدی UP نمو می‌یابد.

[2] بحث جامعی درباره آزمون نرم‌افزار (از جمله آزمون واحد) در فصل‌های ۱۷ تا ۲۰ ارائه شده است.

در سطح شرکتی یا سازمانی توسعه یافته باشد، فقط در صورتی می‌تواند موثر واقع شود که قابلیت تطبیق در آن باشد، به گونه‌ای که قادر به برآوردن نیازهایی باشد که واقعاً کار مهندسی نرم‌افزار را انجام می‌دهند. در یک شرایط ایده‌آل، فرایندی را باید ایجاد کنید که به بهترین وجه بر نیازهای شما تطبیق یابد و در عین حال، نیازهای گسترده‌تر تیم و سازمان را نیز پوشش دهد. به طریق دیگر، تیم می‌تواند برای خودش فرایند نرم‌افزار ایجاد کند و در عین حال نیازهای مشخص‌تر افراد و نیازهای کلی‌تر سازمان را نیز در آن برآورده سازد. واتس هامفری [Hum97] و [Hum00] چنین استدلال می‌کند که ایجاد یک «فرایند نرم‌افزار شخصی» و/یا یک «فرایند نرم‌افزار تیمی» امکان‌پذیر است. هر دو روش مستلزم کار سخت، آموزش و هماهنگی است، ولی هر دو قابل انجام است.[1]

> «کسی که موفق است، ضرفاً عادت کرده است کارهایی را انجام دهد که آدم‌های ناموفق انجام نخواهند داد.»
>
> دکستر یاگر

۶-۲-۱ فرایند نرم‌افزار شخصی (PSP)

هر سازنده‌ای برای ساختن نرم‌افزار کامپیوتری از یک فرایند استفاده می‌کند. این فرایند ممکن است برحسب اتفاق شکل گرفته باشد یا با هدفی خاص؛ ممکن است روزانه تغییر کند، ممکن است اثربخش نباشد، موثر نباشد، یا حتی موفقیت‌آمیز هم نباشد، ولی فرایندی «وجود دارد». واتس هامفری [Hum97] پیشنهاد می‌کند که به‌منظور تغییر دادن یک فرایند شخصی فاقد اثربخشی، شخص باید چهار مرحله را پشت سر بگذارد که هر یک نیاز به تعلیم و تجهیزات دقیق دارد. فرایند نرم‌افزار شخصی (PSP) بر اندازه‌گیری شخصی محصول کاری تولیدشده و کیفیت حاصل از محصول کاری تأکید دارد. به‌علاوه، در PSP مجری کار است که مسؤول برنامه‌ریزی پروژه (مثلاً انجام برآوردها و زمان‌بندی) است و کنترل کیفیت همه‌ی محصولات کاری نرم‌افزاری ساخته شده بر عهده‌ی خود او است. در مدل PSP پنج فعالیت چارچوبی تعریف می‌شود:

برنامه‌ریزی. در این فعالیت، خواسته‌ها شناسایی می‌شود و برآورد منابع و تعیین اندازه پروژه انجام می‌شود. به‌علاوه، برآوردی از نقایص (تعداد نقایص پیش‌بینی شده برای کار) به عمل می‌آید. همه‌ی معیارها روی کاربرگ‌ها یا قالب‌ها ثبت می‌شوند. سرانجام، وظایف لازم برای توسعه‌ی نرم‌افزار تعیین و زمان‌بندی پروژه انجام می‌شود.

طراحی سطح بالا. مشخصات خارجی برای هر کدام از مؤلفه‌هایی که قرار است تعیین شود و طراحی مؤلفه‌ها انجام می‌شود. نمونه‌های اولیه ساخته می‌شوند، در حالی که هنوز عدم قطعیت‌هایی موجود است. همه‌ی مسائل و مشکلات، ثبت و پیگیری می‌شوند.

مرورطراحی سطح بالا. روش‌های وارسی رسمی فصل (۲۱) برای یافتن خطاهای طراحی، اعمال می‌شوند. معیارهای مربوط به وظایف مهم و نتایج کار، نگهداری می‌شود.

توسعه. طراحی سطح بالا پالایش و بازبینی می‌شود. کدها تهیه، بازبینی، کامپایل و آزموده می‌شوند. معیارها برای کلیه‌ی وظایف مهم و نتایج کاری حفظ می‌شوند.

پایان کار. با استفاده از معیارها و موازین جمع‌آوری شده (که مقادیر چشمگیری از داده‌ها را شامل می‌شود و باید مورد تحلیل آماری قرار گیرد)، اثربخشی فرایند تعیین می‌شود. موازین و معیارها باید راهنمایی برای اصلاح فرایند و بهبودبخشیدن به اثربخشی آن فراهم آورند.

مرجع وب

مجموعه گسترده‌ای از منابع مربوط به PSP را می‌توان در وب‌سایت زیر یافت:

www.ipd.uka.de/PSP/

طی PSP از چه فعالیت‌های چارچوبی استفاده خواهد شد؟

[1] شایان ذکر است که مدافعان توسعه چابک (فصل ۳) نیز استدلال می‌کنند که فرایند باید به تیم نزدیک بماند. آنها برای این منظور، روش دیگری پیشنهاد می‌کنند.

در PSP تأکید بر شناسایی زود هنگام خطاهاست و شناخت انواع خطاهایی که احتمال ارتکاب آنها وجود دارد نیز به همان اندازه اهمیت دارد. این هدف از طریق یک فعالیت ارزیابی طاقت‌فرسا قابل دستیابی است که باید روی کلیه محصولات کاری تولیدشده اجرا شود.

PSP نمایان‌گر روشی منضبط و مبتنی بر معیارها برای مهندسی نرم‌افزار است که ممکن است برای بسیاری از دست‌اندرکاران موجب شوک فرهنگی شود، ولی هنگامی که PSP به‌طرزی مناسب به مهندسان نرم‌افزار معرفی گردد، [Hum96]، بهبود حاصل در بهره‌وری مهندسی نرم‌افزار و کیفیت محصول، چشم‌گیر خواهد بود [Fer96]، ولی از PSP در سرتاسر این صنعت به‌طور گسترده استقبال نشده است. دلایل این امر متاسفانه بیشتر به طبیعت انسانی و لُختی سازمانی مربوط می‌شود و ربطی به نقاط قوت و ضعف روش PSP ندارد. PSP هوشمندانه ایجاد چالش می‌کند و سطح بالایی از تعهد (توسط دست‌اندرکاران و مدیران ایشان) را طلب می‌کند که همواره دستیابی به آن امکان‌پذیر نیست. آموزش نسبتاً طولانی است و هزینه‌ی آن هم بالاست. در این روش، دستیابی به سطح بالای سنجش مورد نیاز، به لحاظ فرهنگی برای بسیاری از افراد دشوار است.

آیا از PSP می‌توان به‌عنوان یک فرایند نرم‌افزار اثربخش در سطحی شخصی استفاده کرد؟ پاسخ، به‌روشنی «مثبت» است، ولی PSP حتی اگر به‌طور کامل به کاربرده نشود، بسیاری از مفاهیم آن در زمینه بهبود فرایندهای شخصی ارزش یادگیری را دارد.

نکته‌ی کلیدی

PSP بر نیاز به ثبت و تحلیل انواع خطاهای مرتکب شده تأکید دارد، به‌طوری که بتوانید راهبردهایی برای حذف آنها ارائه دهید.

۲-۶-۲ فرایند نرم‌افزار تیمی (TSP)

از آنجا که بسیاری از پروژه‌های نرم‌افزاری در پایه‌ی صنعتی را تیمی از دست‌اندرکاران انجام می‌دهند، واتس هامفری درس‌هایی را که از معرفی PSP فرا گرفته بود، بسط داد و یک فرایند نرم‌افزار تیمی (TSP)[1] نیز پیشنهاد داد. هدف TSP تشکیل یک تیم پروژه‌ی «خودهدایت‌گر»[2] است که سازمان‌دهی برای تولید نرم‌افزارهای پرکیفیت را خود عهده‌دار می‌شود. هامفری [Hum98] فعالیت‌های زیر را برای TSP تعریف می‌کند:

- تشکیل تیم‌های «خودهدایت‌گری» که کار خود را برنامه‌ریزی و پیگیری می‌کنند، اهداف را تعیین می‌کنند و خود به تعیین فرایندها و طرح‌ها اقدام می‌نمایند. این تیم‌ها می‌توانند تیم‌های نرم‌افزاری محض یا تیم‌های محصولات انسجام یافته (IPT) شامل سه تا حدود بیست مهندس باشند.
- نشان دادن شیوه‌ی راهبری و ایجاد انگیزه در تیم‌ها به مدیران و چگونگی کمک به آنها در حفظ حداکثر کارایی.
- شتاب بخشیدن به بهبود فرایند نرم‌افزار با نهادینه ساختن CMM سطح ۵.[3]
- فراهم ساختن دستورالعمل بهسازی برای سازمان‌های بالغ.
- تسهیل آموزش دانشگاهی مهارت‌های تیمی در سطح صنعتی.

مرجع وب

اطلاعات مربوط به تیم‌های با کارایی بالا با استفاده از TSP و PSP را می‌توانید در وب‌سایت زیر بیابید:
www.sei.amu.edu/tsp

[1] Team Software Process
[2] self directed
[3] مدل بلوغ شایستگی‌ها (CMM) که میزانی از اثربخشی یک فرایند نرم‌افزار است و در فصل ۳۰ بحث خواهد شد.

یک تیم «خودهدایت‌گر» درک سازگاری از اهداف و مقاصد خود دارد؛ نقش‌ها و مسؤولیت‌ها را برای هر عضو تیم تعریف می‌کند؛ داده‌های کمّی پروژه (مربوط به بهره‌وری و کیفیت) را زیر نظر دارد؛ تیم پروژه‌ای را تعیین می‌کند که برای پروژه مناسب باشد و برای پیاده‌سازی فرایند، یک راهبرد تعیین می‌کند؛ استانداردهای محلی قابل استفاده را برای کار مهندسی نرم‌افزار تعریف می‌کند؛ خطرات را پیوسته ارزیابی می‌کند و به آن واکنش نشان می‌دهد؛ و سرانجام اینکه وضعیت پروژه را پیگیری، مدیریت و گزارش می‌کند.

در TSP، فعالیت‌های چارچوبی زیر تعریف می‌شود: **آغاز پروژه، طراحی سطح بالا، پیاده‌سازی، انسجام‌دهی و آزمون، و پایان کار.** این فعالیت‌ها مانند همتاهای خود در PSP (توجه دارید که اصطلاحات قدری تفاوت دارند)، تیم را قادر به برنامه‌ریزی، طراحی و ساخت نرم‌افزار به‌شیوه‌ای منضبط است در حالی که در عین حال، اندازه‌گیری کمی فرایند و محصول انجام می‌شود. مرحله‌ی پایان کار صحنه را برای بهسازی فرایند آماده می‌کند.

در TSP گستره‌ی وسیعی از اسکریپت‌ها، فرم‌ها و استانداردها به‌کار برده می‌شوند که به راهنمای اعضای تیم در انجام وظایفشان کمک می‌کند. «اسکریپت ها» فعالیت‌های فرایندی خاصی (یعنی آغاز پروژه، طراحی، پیاده‌سازی، انسجام دهی و آزمون سیستم و پایان کار) و جزئیات بیشتری از سایر وظایف کاری (مثل برنامه‌ریزی توسعه، توسعه‌ی خواسته‌ها، مدیریت پیکربندی نرم‌افزار و آزمون واحدها) را تعریف می‌کنند که بخشی از فرایند تیمی به‌شمار می‌روند.

در TSP بهترین تیم‌های نرم‌افزاری، تیم‌های خودهدایت‌گرند.[1] اعضای تیم، اهداف پروژه را تعیین می‌کنند، فرایند را طوری تطبیق می‌دهند تا نیازهای آنها را برآورده کنند، زمان‌بندی پروژه را کنترل می‌کنند و از طریق اندازه‌گیری و تحلیل معیارهای جمع‌آوری‌شده، پیوسته روش تیم برای مهندسی نرم‌افزار را بهبود می‌بخشند.

TSP همانند PSP، روشی طاقت‌فرسا برای مهندسی نرم‌افزار است که مزایایی شاخص و قابل اندازه‌گیری در کیفیت و بهره‌وری به‌دنبال دارد. تیم باید تعهد کامل به فرایند داشته باشد و برای حصول اطمینان از به‌کارگیری مناسب روش، آموزش کامل دیده باشد.

اندرز

برای تشکیل یک تیم «خودهدایت گر»، باید از یک سطح همکاری داخلی خوب و ارتباط خارجی عالی برخوردار باشید.

نکته‌ی کلیدی

اسکریپت‌های TSP، عناصر فرایند تیمی و فعالیت‌هایی را که در این فرایند رخ می‌دهند، تعریف می‌کند.

۲-۷ فن‌آوری فرایند

یک یا چند مورد از مدل‌های فرایندی که در بخش‌های قبل بحث شدند، باید توسط یک تیم پروژه‌ی نرم‌افزاری به‌کار برده شوند. برای رسیدن به این هدف، ابزارهای فن‌آوری فرایند، جهت کمک به سازمان‌های نرم‌افزاری در تحلیل فرایند جاری خود، سازمان‌دهی وظایف کاری، کنترل فرایند و نظارت بر آن، و مدیریت کیفیت فنی به‌وجود آمده‌اند.

ابزارهای فن‌آوری فرایند، به سازمان نرم‌افزاری امکان می‌دهند تا یک مدل خودکار از چارچوب فرایند مشترک، مجموعه وظایف و فعالیت‌های چتری بحث شده در بخش ۳-۲ را بسازد. سپس این مدل را که معمولاً به‌صورت یک شبکه ارائه می‌شود، می‌توان برای تعیین جریان کاری معمول تحلیل نمود و ساختارهای فرایند دیگری را بررسی کرد که استفاده از آنها ممکن است باعث کاهش هزینه و زمان شود.

[1] در فصل ۳ درباره تیم‌های «خودسازمان‌ده» به‌عنوان عنصری مهم در توسعه‌ی چابک بحث خواهیم کرد.

هنگامی که یک فرایند قابل قبول ایجاد شد، از ابزارهای دیگر فن‌آوری فرایند می‌توان برای تخصیص، نظارت و حتی کنترل کلیه وظایف مهندسی نرم‌افزار تعیین شده به‌عنوان بخشی از مدل فرایند استفاده نمود. هر یک از اعضای تیم پروژه‌ی نرم‌افزاری می‌تواند از چنین ابزاری برای تهیه لیست کنترلی از کارهایی که باید انجام شود، محصولات کاری که باید تولید شود و فعالیت‌های تضمین کیفیتی که باید به اجرا درآید، استفاده کند. ابزار فن‌آوری فرایند را می‌توان برای هماهنگ ساختن ابزارهای دیگر مهندسی نرم‌افزار که برای یک وظیفه‌ی خاص مناسب باشند نیز به‌کار برد.

ابزارهای نرم‌افزاری

ابزارهای مدل‌سازی فرایند

هدف: اگر سازمانی برای بهبود بخشیدن به یک فرایند تجاری (یا نرم‌افزاری) کار کند، ابتدا باید آن را بشناسد. ابزارهای مدل‌سازی فرایند (که فن‌آوری فرایند یا ابزارهای مدیریت فرایند نیز نامیده می‌شوند) در ارائه‌ی عناصر کلیدی یک فرایند به‌کار می‌روند، به‌طوری که درک آن آسان‌تر شود. چنین ابزارهایی می‌توانند توصیف‌هایی از فرایند فراهم سازند که به‌دست‌اندرکاران فرایند در فهم کنش‌ها و وظایف کاری مورد نیاز برای اجرای آن کمک کند. ابزارهای مدل‌سازی فرایند ارتباط با سایر ابزارهایی را فراهم می‌سازند که فعالیت‌های فرایندی تعریف شده را پشتیبانی می‌کنند.

مکانیک: ابزارهای موجود در این گروه به تیم این امکان را می‌دهند که عناصر یک مدل فرایند منحصر به فرد (کنش‌ها، وظایف، محصولات کاری، نقاط تضمین کیفیت)، راهنمای مفصلی درباره محتویات یا توصیف هر کدام از عناصر فرایند را فراهم آورند و سپس فرایند را به هنگام اجرا، مدیریت می‌کنند. در برخی موارد، ابزارهای فن‌آوری فرایند شامل وظایف استاندارد مدیریت پروژه از قبیل برآورد، زمان بندی، پیگیری و کنترل می‌شوند.

ابزارهای نمونه:

ابزارهای فرایند Igrafx- ابزارهایی که به تیم امکان طراحی، اندازه‌گیری و مدل‌سازی فرایند نرم‌افزار را می‌دهند (www.microgrfx.com)

سرور Adeptia BMP- برای مدیریت، خودکارسازی و بهینه‌سازی فرایندهای تجاری طراحی شده است (www.adeptia.com)

Speed Dev Suite- مجموعه‌ای از شش ابزار با تأکید زیاد بر مدیریت ارتباطات و فعالیت‌های مدل‌سازی (www.speedev.com)

۲-۸ محصول و فرایند

اگر فرایند ضعیف باشد، محصول نهایی بدون شک ضعیف خواهد بود. ولی اتکای بیش از حد به فرایند نیز خطرناک است. ماگارت دیویس [DAV95] در یک مقاله کوتاه درباره دوگانگی محصول و فرایند توضیح می‌دهد:

حدوداً هر ۱۰ سال یا هر ۵ سال، جامعه‌ی نرم‌افزاری با جابجاکردن نقطه توجه از محصول به فرایند، به تعریف دوباره «مسأله» می‌پردازد. از این رو، زبان‌های برنامه‌نویسی ساخت‌یافته (محصول)، سپس روش‌های تحلیل ساخت‌یافته (فرایند)، به‌دنبال آن، پنهان‌سازی داده‌ها (محصول) و سپس تأکید کنونی بر مدل بلوغ قابلیت‌های توسعه‌ی نرم‌افزار، بنیاد مهندسی نرم‌افزار را در آغوش گرفتیم.

هنگامی که قرار است پاندول در جایی بین دو حد نهایی قرار بگیرد، توجه جامعه‌ی نرم‌افزاری به‌طور پیوسته دستخوش دگرگونی می‌شود، زیرا نیروی جدید هنگامی وارد می‌شود که پاندول از آخرین حرکت نوسانی خود باز می‌ماند. این نوسانات زیان‌بارند، چون نرم‌افزارنویس متوسط را با تغییرات بنیادی در معنای انجام کار، دچار سردرگمی می‌کنند. این نوسانات «مسأله» را حل نمی‌کنند زیرا مادامی که تصور شود فرایند و محصول با هم تضاد دارند و به دوگانگی آنها توجه نشود، محکوم به شکست هستند.

در جامعه‌ی علمی هنگامی که تضادهای موجود در مشاهدات را نتوان با یکی از دو نظریه رقیب توضیح داد، موضوع دوگانگی پیش کشیده می‌شود. ماهیت دوگانه نور که به‌نظر می‌رسد در آن واحد هم ذره باشد و هم نور، از سال ۱۹۲۰ یعنی زمانی که لویی دوبروی آن را پیشنهاد کرد، مورد پذیرش قرار گرفت. من معتقدم مشاهداتی که می‌توانیم روی نرم‌افزار و توسعه آن داشته باشیم یک دوگانگی بنیادی میان محصول و فرایند را نشان می‌دهد. اگر چیزی را تنها به‌عنوان یک فرایند یا به‌عنوان یک محصول در نظر بگیرید هرگز نخواهید توانست کاربرد، محتوا، معنا و ارزش آن را به‌طور کامل دریابید ... .

همه‌ی فعالیت‌های انسان ممکن‌است یک فرایند باشند، ولی هر یک از ما از فعالیت‌هایی که منجر به نمونه‌ای قابل استفاده یا مورد تقدیر دیگران بشود، بارها و بارها استفاده بشود، یا در جای دیگری که تصور آن نمی‌رفت مفید واقع شود، احساس غرور می‌کنیم. یعنی از استفاده‌ی مجدد محصولات خود توسط دیگران یا خودمان احساس رضایت می‌کنیم.

از این رو، در حالی که جذب سریع اهداف استفاده‌ی مجدد در توسعه‌ی نرم‌افزار، به‌طور بالقوه باعث افزایش احساس رضایت سازنده‌ی نرم‌افزار می‌شود، ضرورت پذیرش دوگانگی فرایند و محصول را نیز افزایش می‌دهد. در نظر گرفتن یک قطعه‌ی قابل استفاده‌ی مجدد تنها به‌عنوان یک محصول یا تنها به‌عنوان یک فرایند، شیوه‌ها و جایگاه استفاده از آن را دچار ابهام می‌سازد یا این واقعیت را مبهم می‌سازد که هر بار استفاده منجر به محصولی می‌شود که به نوبه‌ی خود به‌عنوان یک ورودی برای یک فعالیت توسعه‌ی نرم‌افزاری دیگر به‌کار می‌رود. ارجح دانستن یک دیدگاه بر دیدگاه دیگر، به‌طرز چشمگیری فرصت استفاده‌ی مجدد را کاهش داده فرصت افزایش رضایت از کار، از دست می‌رود.

افراد به همان اندازه که از محصول نهایی احساس رضایت می‌کنند، از فرایند خلاق نیز احساس رضایت می‌کنند (بلکه بیشتر). یک نقاش به همان اندازه که از نتیجه کار لذت می‌برد، از حرکت قلم‌مو بر روی بوم نیز لذت می‌برد. یک نویسنده به همان اندازه که از کتاب کامل شده لذت می‌برد از گشتن به‌دنبال استعاره و کنایه‌ای مناسب نیز لذت می‌برد. یک نرم‌افزارنویس حرفه‌ای خلاق نیز باید به همان اندازه که از محصول نهایی احساس رضایت می‌کند، از فرایند نیز راضی باشد.

۲-۹ خلاصه

یک مدل فرایند کلی برای مهندسی نرم‌افزار شامل مجموعه‌ای از فعالیت‌های چارچوبی و چتری، کنش‌ها و وظایف کاری می‌شود. هر کدام از انواع مدل‌های فرایند موجود را می‌توان با یک جریان فرایندی متفاوت توصیف کرد- شرحی از چگونگی فعالیت‌های چارچوبی، کنش‌ها و وظایف

به‌صورت ترتیبی و زمان‌بندی شده سازمان‌دهی می‌شوند. از الگوهای فرایند می‌توان برای حل مسائل متداول در فرایند نرم‌افزار استفاده کرد.

طی سال‌ها تلاش برای نظم بخشیدن و ساختاردهی به توسعه‌ی نرم‌افزار، از مدل‌های تجویزی استفاده شده است. در هر کدام از این مدل‌ها یک جریان فرایندی با قدری اختلاف از دیگری پیشنهاد می‌شود، ولی در همه‌ی آنها مجموعه فعالیت چارچوبی یکسانی انجام می‌شود که عبارتند از: برقراری ارتباط، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار.

مدل‌های فرایند ترتیبی نظیر مدل‌های آبشاری و V، قدیمی‌ترین الگوهای مهندسی نرم‌افزارند. در این مدل‌ها، یک جریان فرایند خطی پیشنهاد می‌شود که غالباً با واقعیت‌های مدرن (مانند تغییر پیوسته، سیستم‌های در حال تکامل، جدول‌های زمانی فشرده) در جهان نرم‌افزار سازگاری ندارند، ولی این مدل‌ها قابلیت کاربرد در شرایطی را دارند که خواسته‌ها در آنها کاملاً مشخص و پایدارند.

مدل‌های افزایشی ماهیتی تکراری دارند و نسخه‌هایی کاری از نرم‌افزار را با سرعت زیاد تولید می‌کنند. در مدل‌های فرایند تکاملی، ماهیت افزایشی اکثر پروژه‌های مهندسی نرم‌افزار مورد توجه قرار می‌گیرد و طراحی به گونه‌ای انجام می‌شود که تغییرات را پاسخ‌گو باشد. مدل‌های تکاملی، نظیر ساخت نمونه‌ی اولیه و مدل حلزونی، محصولات کاری افزایشی را به سرعت ایجاد می‌کنند. از این مدل‌ها می‌توان برای همه‌ی فعالیت‌های مهندسی نرم‌افزار(از توسعه‌ی مفاهیم تا نگهداری درازمدت سیستم‌ها) استفاه کرد.

با مدل فرایند همروند، تیم نرم‌افزاری می‌تواند عناصر تکراری و هم زمان هر مدل فرایند را به نمایش بگذارد. مدل‌های تخصص یافته شامل یک مدل مبتنی بر مؤلفه‌هایی می‌شوند که بر استفاده‌ی مجدد از مؤلفه‌ها و مونتاژ آنها تأکید دارد؛ مدل روش‌های رسمی که یک روش مبتنی بر مفاهیم ریاضی را برای توسعه‌ی نرم‌افزار و وارسی آن پیشنهاد می‌کند؛ و مدل جنبه‌گرا که دغدغه‌های متقاطع در برگیرنده‌ی کل معماری سیستم را در خود جای می‌دهد. فرایند یکپارچه یک فرایند نرم‌افزار مبتنی بر use case معماری، تکرار و افزایش است که به‌عنوان چارچوبی برای روش‌ها و ابزارهای UML طراحی می‌شود.

مدل‌های تیمی و شخصی نیز برای فرایند نرم‌افزار پیشنهاد شده‌اند. در هر دو مدل، اندازه گیری، برنامه‌ریزی و خود هدایت گری به‌عنوان مصالح اصلی برای یک فرایند نرم‌افزار موفق مورد تأکید قرار می‌گیرند.

مسائل و نکاتی برای تعمق

۲-۱ در مقدمه‌ی این فصل باتیر اشاره می‌کند که: «فرایند تعامل میان کاربران و طراحان، میان کاربران و ابزارهای در حال تکامل و میان طراحان و ابزارهای در حال تکامل [فن‌آوری] را فراهم می‌سازد.» پنج پرسش بنویسید که (الف) طراحان باید از کاربران بپرسند، (ب) کاربران باید از طراحان بپرسند، (پ) کاربران باید درباره محصول نرم‌افزاری که قرار است ساخته شود، از خود بپرسند، (ت) طراحان باید درباره محصول نرم‌افزاری که قرار است ساخته شود و نیز درباره فرایندی در ساخت آن به کاربرده شود، از خود بپرسند.

۲-۲ تلاش کنید برای فعالیت برقراری ارتباط یک مجموعه کنش توسعه دهید. یکی از این کنش‌ها را انتخاب کرده مجموعه وظایفی برای آن تعریف کنید.

۲-۳ یک مسأله‌ی متداول طی «برقراری ارتباط» هنگامی رخ می‌دهد که با دو تن از طرف‌های ذی‌نفع مواجه می‌شوید که درباره‌ی ماهیت نرم‌افزار، نظرات و آرای متضاد دارند. یعنی خواسته‌هایی متناقض پیش روی شما قرار می‌گیرد. با استفاده از قالب ارائه شده در بخش ۳-۱-۲ که به این مشکل می‌پردازد، یک الگوی فرایند (که یک الگوی صحنه خواهد بود) توسعه دهید.

۲-۴ قدری روی PSP تحقیق کنید و طی یک سمینار مختصر، انواع اندازه گیری‌های قابل استفاده برای بهبود بخشیدن به اثربخشی شخصی را شرح دهد.

۲-۵ استفاده از «اسکریپت ها» (سازوکاری لازم در TSP) در جامعه‌ی نرم‌افزاری مورد تایید همگانی نیست. فهرستی از مزایا و معایب مربوط به اسکریپت‌ها تهیه کنید و دست کم دو وضعیت پیشنهاد کنید که در آن اسکریپت‌ها مفید باشند و دو وضعیت دیگر ذکر کنید که استفاده از اسکریپت‌ها چندان مزیتی نداشته باشد.

۲-۶ [Nog00] را بخوانید و یک مقاله‌ی دو صفحه‌ای و سه صفحه‌ای بنویسید که تأثیر «آشوب» را بر مهندسی نرم‌افزار بحث کنید.

۲-۷ سه مثال از پروژه‌های نرم‌افزاری ذکر کنید که در مدل آبشاری قابل پیاده‌سازی باشند. به جزئیات بپردازید.

۲-۸ سه مثال از پروژه‌های نرم‌افزاری ذکر کنید که در مدل ساخت نمونه‌ی اولیه قابل پیاده‌سازی باشند. به جزئیات بپردازید.

۲-۹ چه تطبیق‌هایی برای فرایند مورد نیاز است اگر نمونه‌ی اولیه به یک سیستم یا محصول قابل تحویل تکامل یابد.

۲-۱۰ سه مثال از پروژه‌های نرم‌افزاری ذکر کنید که در مدل افزایشی، قابل پیاده‌سازی باشند. به جزئیات بپردازید.

۲-۱۱ با حرکت به طرف بیرون در جریان فرایند مارپیچی (حلزونی)، درباره نرم‌افزاری که درحال توسعه یا نگهداری‌شدن است، چه می‌توان گفت؟

۲-۱۲ آیا امکان تلفیق فرایندها وجود دارد؟ در صورت مثبت بودن پاسخ، مثال بیاورید.

۲-۱۳ مدل فرایند همروند مجموعه‌ای از «حالت ها» را تعریف می‌کند. به زبان ساده شرح دهید که این حالت‌ها چه چیزی را نشان می‌دهند و سپس بگویید که در مدل فرایند هم زمان چگونه وارد عمل می‌شوند.

۲-۱۴ مزایا و معایب نرم‌افزارهای در حال توسعه‌ای که کیفیت آنها «به قدر کافی خوب» است، چیست؟ یعنی، هنگامی که بر سرعت بیش از کیفیت محصول تأکید می‌شود، چه اتفاقی رخ می‌دهد؟

۲-۱۵ سه مثال از پروژه‌های نرم‌افزاری ذکر کنید که در مدل مبتنی بر مؤلفه‌ها قابل پیاده‌سازی باشند. به جزئیات بپردازید.

۲-۱۶ این را می‌توان اثبات کرد که یک مؤلفه‌ی نرم‌افزاری و حتی کل برنامه درست است. پس چرا همه این کار را نمی‌کنند؟

۲-۱۷ آیا فرایند یکپارچه و UML یکسان هستند؟ درباره پاسخ خویش توضیح دهید.

فصل ۳

توسعه‌ی چابک

نگاهی گذرا

توسعه‌ی چابک[1] چیست؟ مهندسی نرم‌افزار چابک، تلفیقی از یک فلسفه و مجموعه‌ای از دستورالعمل‌های توسعه است. این فلسفه مشوق جلب رضایت مشتری و تحویل افزایشی نرم‌افزار از همان ابتدای پروژه؛ تیم‌های پروژه‌ی کوچک با انگیزه بالا؛ روش‌های غیر رسمی؛ حداقل محصولات کاری مهندسی نرم‌افزار؛ و سادگی کلی در توسعه‌ی نرم‌افزار است. دستورالعمل‌های توسعه بر تحویل نرم‌افزار بر اساس تحلیل و طراحی (گرچه این فعالیت‌ها تشویق نمی‌شوند) و برقراری ارتباط فعال و پیوسته میان توسعه دهندگان نرم‌افزار و مشتریان آن تأکید دارد.

چه کسی این کار را انجام می‌دهد؟ مهندسان نرم‌افزار و سایر طرف‌های ذی‌نفع در پروژه (مدیران، مشتریان و کاربران نهایی) با یکدیگر کار می‌کنند و یک تیم چابک تشکیل می‌دهند- تیمی که سازمان‌دهی‌اش بر عهده خودش است و سرنوشت‌اش را خود کنترل می‌کند. تیم چابک، به ارتباطات و همکاری میان همه‌ی افراد تیم رسیدگی می‌کند.

چرا اهمیت دارد؟ محیط کاری جدیدی که سیستم‌های کامپیوتری و محصولات نرم‌افزاری را در بر می‌گیرد، با گام‌های سریع به پیش می‌رود و پیوسته در حال تغییر است. مهندسی نرم‌افزار چابک، برای مهندسی سنتی گروه یعنی از نرم‌افزارها و انواع معینی از پروژه‌های نرم‌افزاری، جایگزینی منطقی ارائه می‌دهد. نشان داده شده است که با این شیوه، سیستم‌های موفق به سرعت تحویل می‌شوند.

مراحل کار کدام است؟ توسعه‌ی چابک را شاید به بهترین وجه بتوان «مهندسی نرم‌افزار» نامید. فعالیت‌های چارچوبی پایه- ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار- همچنان به قوت خود باقی خواهند ماند، ولی به یک مجموعه وظایف کمینه تغییر شکل می‌دهند که تیم نرم‌افزاری را به سمت ساخت و تحویل هدایت می‌کنند (عده‌ای چنین استدلال می‌کنند که این به قیمت حذف تحلیل مسأله و طراحی راهکار تمام می‌شود).

محصول کار چیست؟ مشتری و مهندس نرم‌افزار هر دو دیدگاهی یکسان دارند-تنها محصول کاری مهم واقعی، یک «نرم‌افزار» عملیاتی است که در تاریخ مناسب به مشتری تحویل شود.

چگونه اطمینان حاصل کنیم که کار، درست انجام شده است؟ اگر تیم چابک با پردازش کارها موافقت کند و در مرحله‌های تکاملی، نرم‌افزارهای قابل تحویل تولید نماید که رضایت مشتری جلب شود، کار، به درستی انجام شده است.

---

[1] agile development

سال ۲۰۰۱، کنت بک و شانزده نفر دیگر از سازندگان نرم‌افزار، نویسندگان و مشاوران [Bec01a] (که نام آنها به‌عنوان «اتحادیه چابک» یاد می‌شود) «بیانیه توسعه‌ی نرم‌افزاری چابک» را امضا کردند. مفاد این بیانیه به قرار زیر بود:

ما با انجام توسعه‌ی نرم‌افزار و کمک به دیگران در انجام این کار به کشف راه‌های بهتری نائل آمده‌ایم. با این کار به این نتیجه رسیده‌ایم که:

افراد و تعامل‌ها را بر فرایندها و ابزارها
نرم‌افزار عملیاتی را بر مستندات جامع
همکاری با مشتری را بر مذاکره قرارداد
و پاسخ به تغییر را بر دنبال‌کردن یک برنامه برتری دهیم.

یعنی در حالی که در آیتم‌های طرف چپ هم ارزش وجود دارد، به آیتم‌های طرف راست ارزش بیشتری خواهیم داد.

بیانیه‌ها معمولاً به جنبش‌های سیاسی مربوط می‌شوند- چیزی که به سیستم قدیمی حمله می‌کند و تغییری انقلابی را پیشنهاد می‌کند (به امید بهسازی). از جهاتی، این دقیقاً چیزی است که توسعه‌ی چابک با آن سروکار دارد.

گرچه ایده‌های زیربنایی که توسعه‌ی چابک را هدایت می‌کنند، سال‌ها با ما بوده‌اند، کمتر از دو دهه است که این ایده‌ها در قالب یک «جنبش» متبلور شده‌اند. در اصل، روش‌های چابک[1] در نتیجه‌ی تلاش برای غلبه بر ضعف‌های واقعی و دریافتی در مهندسی نرم‌افزار سنتی توسعه یافته‌اند. توسعه‌ی چابک می‌تواند مزایای مهمی به همراه داشته باشد، ولی برای همه‌ی پروژه‌ها، همه‌ی محصولات، همه‌ی افراد و همه‌ی شرایط قابل استفاده نیست. این روش فقط برای مهندسی نرم‌افزار نیست بلکه به‌عنوان فلسفه‌ای جایگزین برای همه‌ی کارهای نرم‌افزاری قابل استفاده است.

در اقتصاد نوین، پیش‌بینی چگونگی تکامل یافتن یک سیستم کامپیوتری (مثلاً یک برنامه‌ی کاربردی مبتنی بر وب) درگذر زمان، غالباً کاری دشوار است. شرایط بازار به سرعت تغییر می‌کند، نیازهای کاربران نهایی تکامل می‌یابد و تهدیدهای رقابتی بدون هشدار قبلی ظهور می‌کند. در بسیاری شرایط، قادر به تعریف کامل خواسته‌ها قبل از شروع پروژه نخواهید بود. باید به قدر کافی چابک باشید تا بتوانید به یک محیط تجاری متغیر پاسخ دهید.

تغییر، هزینه بردار است. به ویژه اگر کنترل نشده باشد یا مدیریت ضعیفی بر آن اعمال شود. یکی از بارزترین ویژگی‌های روش چابک، توانایی آن در کاهش دادن هزینه‌های ناشی از تغییر در سرتاسر فرایند نرم‌افزار است.

آیا این بدان معناست که شناخت چالش‌های پدید آمده از یک واقعیت جدید، باعث می‌شود که همه‌ی اصول، مفاهیم، روش‌ها و ابزارهای ارزشمند مهندسی نرم‌افزار را به کناری بگذارید؟ مطلقاً خیر! مهندسی نرم‌افزار نیز همانند کلیه رشته‌های مهندسی به تکامل خود ادامه می‌دهد و می‌توان آن را طوری تطبیق داد که چالش‌های ناشی از تقاضا برای سرعت را نیز پاسخ‌گو باشد.

[1] در ادامه‌ی چابک را گاه روش‌های سبک یا روش‌های ناب نیز می‌نامند.

الیستر کاکبرن در یک کتاب درباره توسعه‌ی چابک [Coc02] استدلال می‌کند که مدل‌های فرایند پرسکریپتیوی معرفی شده در فصل ۲ یک اشکال عمده دارند. در این مدل‌ها، ضعف و سستی کسانی که نرم‌افزارها می‌سازند، فراموش می‌شود. مهندسان نرم‌افزار، روبات نیستند. آنها در سبک‌های کاری با هم تفاوت‌های بزرگ دارند؛ اختلاف‌های چشمگیر در سطح مهارت، خلاقیت، نظم و انضباط، سازگاری و سرعت عمل. برخی به شکل کتبی قادر به برقراری ارتباط با دیگران هستند و برخی خیر. کاکبرن استدلال می‌کند که در مدل‌های فرایند می‌توان نقاط ضعف متداول افراد را با انضباط یا تحمل اداره کرد و در اکثر مدل‌های تجویزی، انضباط انتخاب می‌شود. او می‌گوید: «چون سازگاری در کنش از نقاط ضعف انسان است، روش‌هایی با انضباط بالا، شکننده‌اند.»

اگر قرار به‌کارکردن روی مدل‌های فرایند باشد، باید سازوکاری واقع بینانه برای تشویق انضباط لازم فراهم آورند، یا باید به نحوی آنها را مشخص کرد که برای افرادی که کار مهندسی نرم‌افزار را انجام می‌دهند، «تحمل» نشان دهند. برای کسانی که در کار نرم‌افزار هستند، روش‌های با تحمل راحت‌تر قابل پذیرش است، ولی (چنان که کاکبرن هم می‌پذیرد) ممکن است بهره‌وری آنها کمتر شود. همانند اکثر چیزهایی که در زندگی وجود دارد، مصالحه میان عوامل را نیز باید در نظر گرفت.

۱-۳. چابکی چیست؟

در حیطه‌ی کار مهندسی نرم‌افزار، چابکی دقیقاً چه معنایی دارد؟ ایوار جیکابسون [Jac02a] در این خصوص بحث مفیدی ارائه می‌دهد:

> این روزها هنگام توصیف یک فرایند نرم‌افزار مدرن، چابکی واژه‌ای است که به‌وفور به گوش می‌رسد. تیم چابک تیمی فرز و چالاک است که قادر است به تغییرات پاسخ مناسب بدهد. تغییر چیزی است که در توسعه‌ی نرم‌افزار، بسیار با آن مواجه می‌شویم. تغییرات در نرم‌افزارهایی که در حال ساخته‌شدن هستند، تغییرات در اعضای تیم، تغییرات به دلیل فن‌آوری جدید، تغییرات از هر نوع که ممکن است بر محصول در حال ساخت یا محصولی که محصول نهایی را ایجاد می‌کند، تأثیر گذار باشند. هر آنچه در نرم‌افزار انجام می‌دهیم باید خود حاوی ویژگی‌هایی برای پشتیبانی از تغییرات باشد؛ چیزی که قلب و روح نرم‌افزار است. تیم چابک می‌داند که نرم‌افزار توسط افرادی توسعه می‌یابد که در قالب تیمی کار می‌کنند و مهارت‌های این افراد و توانایی ایشان در همکاری، هسته‌ی اصلی موفقیت پروژه است.

از دید جیکابسون، فراگیر بودن تغییر، دلیل اصلی برای چابکی است. مهندس نرم‌افزار اگر می‌خواهد به تغییراتی که جیکابسون توصیف می‌کند، پاسخ مناسب بدهد، باید سریع عمل کند.

ولی چابکی، چیزی بیش از پاسخ‌دهی موثر به تغییرات است. این روش شامل فلسفه‌ی ذکر شده در بیانیه‌ی ابتدای این فصل نیز می‌شود. ایجاد ساختارها و صفاتی را در تیم تشویق می‌کند که برقراری ارتباطات (در میان اعضای تیم، بین طرف‌های تجاری و فنی، میان مهندسان نرم‌افزار و مدیران آنها) را تسهیل کنند. این روش، بر تحویل سریع نرم‌افزارهای عملیاتی تأکید دارد و تأکید را از روی اهمیت محصولات کاری بینابینی (که همواره هم چیز خوبی نیستند) بر می‌دارد؛ در این رویکرد، مشتری به‌عنوان بخشی از تیم توسعه پذیرفته می‌شود و کوشش می‌شود که حس و حال «ما و ایشان»، که هنوز بر بسیاری از پروژه‌های نرم‌افزاری سایه افکنده است، حذف شود؛ در این روش به این نکته توجه می‌شود که برنامه‌ریزی در دنیایی با عدم قطعیت، محدودیت‌های خاص خود را دارد و برنامه‌ریزی یک پروژه باید انعطاف‌پذیر باشد.

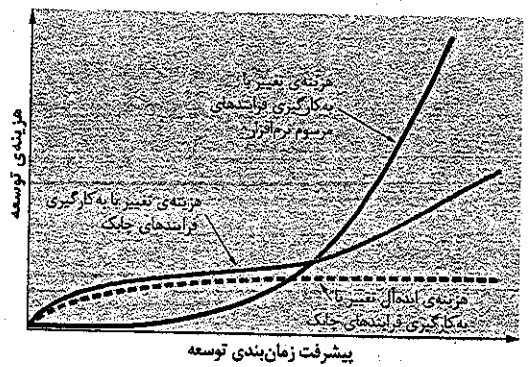
چابکی: ۱همه‌ی چیزهای دیگر: صفر.
تام دومارکو

اندرز
مرتکب این اشتباه نشوید که چابکی این مجوز را به شما می‌دهد که بدون برنامه‌ریزی به راهکار برسید. یک فرایند لازم است و انضباط ضروری است.

چابکی را می‌توان در هر فرایند نرم‌افزار به‌کار برد، ولی برای رسیدن به این هدف، طراحی فرایند باید به گونه‌ای باشد که تیم پروژه قادر به انجام وظایف باشد و بتواند آنها را به جریان بیندازد، برنامه‌ریزی به‌شیوه‌ای صورت گیرد که متغیربودن یک روش توسعه‌ی چابک در آن دیده شده باشد، همه‌ی محصولات کاری به جز آنها که ضروری هستند، حذف شوند و بر راهبرد تحویل افزایشی که نرم‌افزار را هرچه سریع‌تر برای مشتری قابل استفاده کند تأکید ورزد.

۳-۲ چابکی و هزینه‌های تغییر

عقل سلیم در توسعه‌ی نرم‌افزار (که چند دهه تجربه پشتیبان آن است) حکایت از آن دارد که هزینه‌ی تغییر به‌صورت غیر خطی با پیشرفت پروژه افزایش می‌یابد (شکل ۱-۳، منحنی با خط توپر). پاسخ‌گویی به تغییر هنگامی که تیم نرم‌افزاری مشغول جمع‌آوری خواسته‌هاست، نسبتاً آسان است (در همان ابتدای پروژه). یک سناریوی کاربرد (usage scenario) ممکن است نیاز به اصلاح داشته باشد، ممکن است فهرستی از قابلیت‌ها گسترش یابد، یا ممکن است مشخصات مکتوب ویرایش شود. هزینه‌های انجام این کار، اندک است و زمان مورد نیاز تأثیری سوء بر نتیجه‌ی پروژه ندارد، ولی اگر چند ماه جلوتر برویم، چطور؟ تیم در میانه‌ی آزمون وارسی است (چیزی که نسبتاً در اواخر پروژه رخ می‌دهد) و یک طرف ذی‌نفع مهم، درخواست تغییری عمده در قابلیت‌های سیستم دارد. این تغییر نیاز به اصلاح طراحی معماری نرم‌افزار، طراحی و ساخت سه قطعه‌ی جدید دارد، پنج قطعه دیگر باید اصلاح شوند، آزمون‌های جدیدی باید طراحی شود و غیره. هزینه‌ها به سرعت بالا می‌رود و زمان و هزینه‌ی لازم برای حصول اطمینان از اینکه تغییرات بدون برجای گذاشتن اثرات جانبی اعمال شود، قابل چشم پوشی نخواهد بود.



شکل ۱-۳ هزینه‌ی تغییرات به‌عنوان تابعی از زمان در توسعه‌ی نرم‌افزار.

هواداران چابکی (مثل [Bec00]، [Amb04]) استدلال می‌کنند که یک فرایند چابک با طراحی خوب، منحنی هزینه‌ی تغییر را «تسطیح» می‌کند (شکل ۱-۳، منحنی خط توپر و سایه دار) و به این ترتیب، تیم نرم‌افزاری قادر به پاسخ گویی به تغییرات در اواخر پروژه خواهند بود بدون اینکه ضربه‌ای قابل ملاحظه از نظر زمان و هزینه بر پروژه وارد آید. قبلاً دانستیم که فرایند چابک شامل تحویل افزایشی محصول می‌شود. هنگامی که تحویل افزایشی با سایر روش‌های چابک از قبیل آزمون

> «چابکی، در آغوش گرفتن تغییراتی بویژه در مشخص مخواست که به سرعت رخ می‌دهد و گرایش به رشد دارد»
>
> استیون گلدمن

نکته‌ی کلیدی

یک فرایند چابک هزینه‌ی تغییرات را کاهش می‌دهد چون نرم‌افزار در قالب چند گام روانه می‌شود و تغییرات را در یک گام بهتر می‌شود کنترل کرد.

واحدهای پیوسته و برنامه‌نویسی جفتی تلفیق شود، هزینه‌ی اِعمال تغییرات کاهش می‌یابد. گرچه بحث و جدل درباره میزان تسطیح منحنی هزینه همچنان ادامه دارد، شواهد نشان می‌دهد [Cac01a] که کاهش چشمگیری در هزینه‌ها قابل دستیابی است.

۳-۳ فرایند چابک چیست؟

هر فرایند نرم‌افزار چابک به گونه‌ای مشخص می‌شود که تعدادی از فرض‌های کلیدی [Fow02] را درباره اکثریت پروژه‌های نرم‌افزاری پاسخ‌گو باشد:

۱. پیش‌بینی اینکه کدام خواسته‌های نرم‌افزاری باقی خواهند ماند و کدام یک از آنها تغییر می‌کند، دشوار است. پیش‌بینی اینکه اولویت‌های مشتری با پیشرفت پروژه چگونه تغییر می‌کند نیز به همان اندازه دشوار است.

۲. برای بسیاری از انواع نرم‌افزارها، طراحی و ساخت در بین هم انجام می‌شوند. یعنی هر دو فعالیت را باید به‌طور موازی انجام داد، به‌طوری که مدل‌های طراحی به هنگام ایجاد، به اثبات برسند. پیش‌بینی اینکه چه مقدار طراحی مورد نیاز است، قبل از به‌کارگیری ساخت برای به اثبات رساندن طراحی، دشوار است.

۳. تحلیل، طراحی، ساخت و آزمون ممکن است (از دیدگاه برنامه‌ریزی) به آن اندازه که ما دوست داریم، قابل پیش‌بینی نباشد.

با توجه به فرض‌های فوق، یک پرسش مهم مطرح می‌شود: چگونه فرایندی ایجاد کنیم که قادر به مدیریت موارد غیر قابل پیش‌بینی باشد؟ چنان که پیش از این نیز گفته شد، پاسخ در انطباق‌پذیری (adaptability) فرایند (یعنی تغییر دادن سریع شرایط فنی و پروژه) نهفته است. پس فرایند چابک باید از انطباق‌پذیری برخوردار باشد.

ولی انطباق پیوسته و بدون پیشروی از موفقیت چندانی برخوردار نیست. بنابراین، در یک فرایند نرم‌افزار چابک، باید روند انطباق به‌طور افزایشی انجام پذیرد. برای دستیابی به انطباق تدریجی، تیم نرم‌افزاری چابک نیاز به بازخورد از مشتری دارد (تا بتواند انطباق‌های لازم را انجام دهد). یک عاملی شتاب دهنده به بازخورد مشتریان، نمونه‌ی اولیه‌ای از سیستم عملیاتی یا بخشی از آن است. از این رو، یک راهبرد توسعه‌ی افزایشی را باید نهادینه ساخت. نسخه‌های نرم‌افزار (نمونه‌های اولیه قابل اجرا یا بخش‌هایی از سیستم عملیاتی) باید در دوره‌های زمانی کوتاه مدت تحویل شوند تا روند انطباق بتواند همگام با روند تغییرات ادامه یابد (عدم قابلیت پیش‌بینی). مشتری به کمک این روش مبتنی بر تکرار می‌تواند هر نسخه از نرم‌افزار را مرتباً ارزیابی کند، بازخورد لازم برای تیم نرم‌افزاری را فراهم سازد و بر انطباق‌های به‌عمل‌آمده جهت پاسخ گویی به بازخوردها تأثیر بگذارد.

۳-۳-۱ اصول چابکی

در بیانیهٔ چابک (Agile Alliance) دوازده اصل برای کسانی که می‌خواهند به چابکی دست پیدا کنند، تعریف شده است ([Fow01]، [Agi03]):

۱. جلب رضایت مشتری از طریق تحویل زود هنگام و پیوسته‌ی نرم‌افزارهای ارزشمند، بیشترین اولویت را نزد ما دارد.

مرجع وب

مجموعه جامعی از مقالات درباره فرایندهای چابک را می‌توانید در آدرس زیر بیابید:
www.aanpo.org/articles/index

نکته‌ی کلیدی

گرچه فرایندهای چابک با آغوش باز به استقبال تغییرات می‌روند، هنوز هم بررسی دلایل تغییرات اهمیت دارد.

۲. پذیرا بودن تغییرات در خواسته‌ها حتی در اواخر فرایند توسعه. در فرایندهای چابک، تغییرات برای مزایای رقابتی مشتریان تحت کنترل هستند.
۳. تحویل پیوسته‌ی نرم‌افزارهای کاری از دو هفته گرفته تا دو ماه، که بازه‌های زمانی کوتاه‌تر باید در اولویت قرار داده شوند.
۴. دست‌اندرکاران و افراد تجاری باید در سرتاسر پروژه هر روز با هم کار کنند.
۵. سپردن پروژه به افراد باانگیزه، فراهم‌سازی محیط و پشتیبانی مورد نیاز آنها و اطمینان‌کردن به آنها در انجام کارها.
۶. اثربخش‌ترین و موثرترین روش انتقال اطلاعات به درون و بیرون تیم توسعه، گفتگوی رودررو است.
۷. نرم‌افزار کاری، میزان اصلی در سنجش پیشرفت است.
۸. فرایندهای چابک، توسعه پایدار را ارتقا می‌بخشند. حامیان، سازندگان و کاربران باید قادر به حفظ سرعت ثابت در پیشرفت کار باشد.
۹. توجه پیوسته به اعتلای فنی و طراحی خوب، باعث بهبود افزایش چابکی می‌شود.
۱۰. سادگی- هنر به حداکثر رساندن کارهایی که انجام نمی‌شوند- ضروری است.
۱۱. بهترین معماری‌ها، خواسته‌ها و طراحی‌ها از تیم‌های خودسازمان‌دهی شده ظهور می‌کنند.
۱۲. تیم در بازه‌های منظم، بازخوردی از میزان بهبود اثربخشی خود ارائه می‌دهد و سپس رفتار خود را مطابق این بازخورد تنظیم می‌کند.

این دوازده اصل در تمامی فرایندهای چابک با وزن مساوی به‌کار برده نمی‌شوند و در برخی مدل‌ها از اهمیت یک یا چند اصل چشم پوشی می‌شود (یا دست کم نقش آنها کمرنگ‌تر می‌شود). این اصول، تعیین‌کننده‌ی جوهره‌ای چابک هستند که در هر کدام از مدل‌های فرایند ارائه شده در این فصل حفظ خواهد شد.

اندرز
نرم‌افزاری که کار کند مهم است، ولی فراموش نکنید که انواع صفات کیفیتی از جمله قابلیت اطمینان، قابلیت استفاده و قابلیت نگهداری را هم باید داشته باشد.

۳-۳-۲ سیاست توسعه‌ی چابک

درباره مزایا و قابلیت کاربرد توسعه‌ی نرم‌افزار چابک در مقابل فرایندهای سنتی‌تر در مهندسی نرم‌افزار، بحث و جدل فراوان شده است. جیم های‌اسمیت [Hig02a] هنگام بر شمردن خصوصیات اردوگاه چابک‌ها مواردی حدی را ذکر می‌کند. «روش‌شناسان سنتی، یک مشت آدم‌های فاقد خلاقیت هستند که ترجیح می‌دهند مستنداتی بدون نقص تهیه کنند تا اینکه سیستمی کاری ارائه دهند که نیازهای تجاری را برآورده کند.» او در نقطه‌ی مقابل، موفقیت اردوگاه مهندسی نرم‌افزار سنتی را چنین توصیف می‌کند: «روش شناسان سبک بال یا «چابک» یک مشت نفوذگر با استعدادند که وقتی تلاش می‌کنند اسباب بازی‌های خود را بزرگ کنند و به نرم‌افزارهایی در سطح شرکت‌ها تبدیل کنند، کلی ذوق‌زده می‌شوند.»

این مناظره‌ی روش‌شناسی همانند همه‌ی استدلال‌های فن‌آوری نرم‌افزار، خطر اضمحلال به یک جنگ عقیدتی را دارد. اگر جنگ مغلوبه شود، عقل سلیم از میان می‌رود و باورها جای واقعیت‌هایی را می‌گیرند که باید راهنمای تصمیم‌گیری باشند.

هیچ کس با چابکی مخالفتی ندارد. پرسش واقعی این است که: بهترین راه برای انجام آن چیست؟ و به همان اهمیت، چطور نرم‌افزاری می‌سازید که نیازهای امروز مشتری را برطرف سازد و

اندرز
نباید بین چابکی و مهندسی نرم‌افزار یکی را انتخاب کنید بلکه باید یک رویکرد مهندسی نرم‌افزار انتخاب کنید که چابک باشد.

خصوصیات کیفیتی را از خود بروز دهد که قابلیت بسط و توسعه برای برآوردن نیازهای دراز مدت مشتری را هم در آن امکان‌پذیر سازد؟

هیچ پاسخ مطلقی برای هیچ کدام از این پرسش‌ها وجود ندارد. حتی در خود مکتب چابکی، مدل‌های فراوانی برای فرایند پیشنهاد شده است (بخش ۴-۳) که هر یک تفاوتی ظریف در نگرش به چابکی دارند. در داخل هر مدل مجموعه‌ای از ایده‌ها وجود دارد (چابک گراها دوست دارند آنها را «وظایف کاری» بنامند) که خروج چشمگیری از مهندسی نرم‌افزار سنتی را نشان می‌دهند. با این وجود بسیاری از مفاهیم چابکی، صرفاً برگرفته از مفاهیم خوب مهندسی نرم‌افزارند. نکته مهم اینکه، با در نظر گرفتن بهترین ایده‌ها از هر دو مکتب، بیشترین بهره عاید خواهد شد و چیزی از تخریب دیگری به‌دست نخواهد آمد.

اگر به بحث بیشتر در این خصوص علاقه دارید، به [Hig01]، [Hig02a]،[DeM02] رجوع کنید تا خلاصه‌ای از سایر مسائل فنی و سیاسی مهم را مشاهده نمایید.

۳-۳-۳ عوامل انسانی

مدافعان روش چابک برای توسعه‌ی نرم‌افزار، متحمل رنج فراوانی می‌شوند تا بر اهمیت «عوامل انسانی» تأکید کنند. چنان که کاکبرن و های‌اسمیت [Coc01] گفته‌اند، «توسعه‌ی چابک بر استعدادها و مهارت‌های افراد و شکل دهی به فرایند بر اساس افراد و تیم‌های موجود تأکید دارد.» نکته کلیدی در این گفته آن است که فرایند باید بر اساس نیازهای افراد و تیم‌ها شکل پیدا کند نه برعکس.[1]

اگر قرار باشد اعضای تیم نرم‌افزاری خصوصیات فرایندی را به‌دست آورند که برای ساخت نرم‌افزار به‌کار گرفته می‌شود، چند خصوصیت کلیدی باید در میان افراد تیم چابک و خود تیم وجود داشته باشد:

رقابت: در حیطه‌ی توسعه‌ی چابک (و نیز در مهندسی نرم‌افزار)، «رقابت» شامل استعداد ذاتی، مهارت‌های خاص مرتبط با نرم‌افزار و آگاهی کلی از فرایندی است که تیم برای استفاده برگزیده است. مهارت و آگاهی از فرایند به همه‌ی افرادی که به‌عنوان عضوی از تیم چابک خدمت می‌کنند قابل آموزش است و باید آموزش داده شود.

کانون توجه مشترک. گرچه ممکن است اعضای تیم چابک وظایف متفاوتی را به انجام برسانند و مهارت‌های متفاوتی را وارد پروژه کنند، همه‌ی آنها باید یک هدف واحد را کانون توجه خود قرار دهند - تحویل نسخه‌ی جدیدی از نرم‌افزار به مشتری در زمان مقرر. به‌علاوه، تیم باید برای دستیابی به این هدف، پیوسته بر انطباق‌های کوچک و بزرگ تأکید داشته باشد تا فرایند را بر نیازهای تیم مطابقت دهد.

همکاری. مهندسی نرم‌افزار (با هر فرایندی که انجام شود) عبارت است از ارزیابی، تحلیل و به‌کارگیری اطلاعاتی که با تیم نرم‌افزاری تبادل می‌شود؛ ایجاد اطلاعاتی که همه‌ی طرف‌های ذی‌نفع را در فهم کار تیم یاری دهد؛ و انتشار دادن اطلاعات (نرم‌افزار کامپیوتری و بانک‌های اطلاعاتی مربوط) که برای مشتری ارزش تجاری در برداشته باشد. برای دستیابی به این وظایف، اعضای تیم باید همکاری کنند - با یکدیگر و با طرف‌های ذی‌نفع.

«روش‌های چابک بیشتر خاصیت خود را مرهون دانش نهفته در تیم است نه در دانش نوشته شده روی کاغذ.»
بری بوهم

اعضای یک تیم نرم‌افزاری به چه صفات مهمی باید آراسته باشند؟

[1] سازمان‌های موفق در زمینه مهندسی نرم‌افزار به این واقعیت اذعان دارند. حال مدل فرایندی انتخاب شده هر چه می‌خواهد باشد.

توانایی تصمیم‌گیری. هر تیم نرم‌افزاری خوب (از جمله تیم‌های چابک) باید آزادی کنترل سرنوشت خود را داشته باشد. این بدان معناست که تیم باید خود مختاری داشته باشد – یعنی اجازه تصمیم‌گیری برای مسائل فنی و پروژه.

توانایی حل مسأله با منطق فازی. مدیران نرم‌افزار باید بدانند که تیم چابک پیوسته ناگزیر از مقابله با ابهام است و پیوسته در معرض تغییر قرار دارد. در برخی موارد، تیم باید پذیرای این واقعیت باشد که مسأله‌ای که امروز در حال حل‌کردن آن است، ممکن است فردا مسأله‌ای باشد که دیگر نیازی به حل آن نباشد، ولی درس‌هایی که از حل هر مسأله گرفته می‌شود (از جمله حل مسائل اشتباهی) ممکن است بعداً در پروژه به‌کار آید.

احترام و اطمینان متقابل. تیم چابک باید به چیزی تبدیل شود که دومارکو و لیستر [Dem98] آن را تیم «قوام‌یافته» می‌نامند (فصل ۲۴). تیم «قوام یافته» اطمینان و احترامی را از خود به نمایش می‌گذارد که به واسطه‌ی آن «اعضای تیم چنان به هم پیوسته می‌شوند که کلیت حاصل چیزی بیش از مجموع اجزای تشکیل‌دهنده باشد.»[Dem98].

خودسازمان‌دهی. در حیطه‌ی توسعه‌ی چابک، خودسازمان‌دهی به معنای سه چیز است (۱) تیم چابک، خودش را سازمان‌دهی می‌کند تا کارها به انجام برسد، (۲) تیم، فرایند را سازمان‌دهی می‌کند تا به بهترین نحو در محیط محلی اسکان یابد، (۳) تیم، زمان‌بندی کاری را سازمان‌دهی می‌کند تا به بهترین نحو، تحویل یک نسخه از نرم‌افزار را امکان‌پذیر سازد. خودسازمان‌دهی چند مزیت فنی دارد، ولی مهم‌تر اینکه به بهبود همکاری و تقویت روحیه تیمی کمک می‌کند. در اصل، تیم، مدیریت خودش را خود بر عهده می‌گیرد. کِن شوابر [Sch02] در این مورد چنین می‌نویسد: «تیم است که تصمیم می‌گیرد چه مقدار کار می‌تواند در هر دور از تکرار انجام دهد و تیم است که متعهد انجام این کار می‌شود. هیچ چیز به این اندازه انگیزه را در یک تیم از بین نمی‌برد که آدم دیگری برایش تعیین تکلیف کند. هیچ چیز به اندازه‌ی پذیرش مسؤولیت برای تعیین وظایف و تعهدات در تیم ایجاد انگیزه نمی‌کند.

> «چیزی که برای یک تیم، کافی به‌شمار می‌رود، برای تیم دیگر می‌تواند یا زیادی کافی باشد یا ناکافی.»
>
> **آلیستر کاکبرن**

نکته‌ی کلیدی

تیم خودسازمان‌ده کنترل کارهایش را بر عهده دارد. این تیم خودش به تعهداتش عمل می‌کند و طرح‌هایی برای انجام آنها تعریف می‌کند.

۳-۴ برنامه‌نویسی حدی (XP)

به‌منظور روشن‌تر ساختن فرایند چابک و واردشدن در جزئیات بیشتر، دیدی اجمالی از برنامه‌نویسی حدی (XP) ارائه خواهیم داد که پرکاربردترین رویکرد در توسعه‌ی نرم‌افزار به روش چابک است. گرچه کارهای اولیه‌ای که روی ایده‌ها و روش‌های مرتبط با XP در اواخر دهه‌ی ۱۹۸۰ انجام شد، کارهای موثر در این خصوص توسط کنت بک [Bec04a] نوشته شده است. به تازگی، شکل دیگری از XP موسوم به XP صنعتی (IXP) پیشنهاد شده است. IXP پالایشی از XP است که فرایند چابک را مشخصاً برای استفاده در سازمان‌های بزرگ هدف قرار داده است.

۱-۴-۳ ارزش‌های XP

بک [Bec04a] مجموعه‌ای از پنج *ارزش* را تعریف می‌کند که مبنایی برای همه‌ی کارهای انجام شده در XP تشکیل می‌دهند–ارتباطات، سادگی، بازخورد، جرأت و احترام. هر کدام از این ارزش‌ها به‌عنوان مجرکه‌ای برای فعالیت‌ها، کنش‌ها و وظایف XP به‌کار می‌رود.

> **اندرز**
>
> هرگاه می‌توانید، سادگی را حفظ کنید، ولی بدانید که «بازآرایی» کردن پیوسته می‌تواند زمان و منابع فراوانی را جذب کند.

XP به‌منظور دستیابی به ارتباطات اثربخش میان مهندسان نرم‌افزار و سایر طرف‌های ذی‌نفع (مثلاً برقراری قابلیت‌ها و ویژگی‌های لازم برای نرم‌افزار)، بر همکاری نزدیک و در عین حال غیر رسمی (لفظی) میان مشتریان و سازندگان، برقراری استعاره‌های اثربخش[1] برای به مشورت گذاشتن مفاهیم مهم، بازخورد پیوسته و پرهیز از مستندات پرحجم به‌عنوان واسطه‌ای ارتباطی تأکید دارد.

XP برای دستیابی به *سادگی*، سازندگان را محدود می‌کند تا تنها برای نیازهای فوری کار طراحی را انجام دهند نه اینکه همه‌ی نیازهای آینده را در نظر بگیرد. هدف، ایجاد یک طراحی ساده است که به آسانی در قالب کدنویسی قابل پیاده‌سازی باشد. اگر طراحی باید بهبود یابد، می‌توان آن را بعداً بازآرایی کرد.[2]

بازخورد از سه منبع قابل حصول است: خود نرم‌افزار، مشتری و سایر اعضای تیم نرم‌افزاری. با طراحی و پیاده‌سازی یک راهبرد آزمون اثربخش (فصل‌های ۱۷ تا ۲۰). نرم‌افزار (از طریق نتایج آزمون) بازخوردی در اختیار تیم چابک قرار می‌دهد. XP از آزمون واحدی به‌عنوان تاکتیک اولیه در انجام آزمون‌ها استفاده می‌کند. با توسعه یافتن هر کلاس، تیم یک آزمون واحدی برای تمرین دادن هر کدام از عملیات مطابق با قابلیت مشخص آن توسعه می‌دهد. با تحویل یک نسخه از نرم‌افزار به مشتری، داستان‌های کاربر (user stories) یا use case (فصل ۵) که توسط آن نسخه پیاده‌سازی می‌شوند، به‌عنوان مبنایی برای آزمون‌های پذیرش به‌کار می‌روند. میزان پیاده‌سازی خروجی، عملکرد و رفتار ذکر شده در یک use case، شکلی از بازخورد است. سرانجام، با به‌دست آمدن خواسته‌های جدید به‌عنوان بخشی از برنامه‌ریزی تکراری، تیم یک بازخورد سریع از تاثیر زمان‌بندی و هزینه در اختیار مشتری قرار می‌دهد.

بک [Bec04a] چنین استدلال می‌کند که پایبندی سفت و سخت به برخی جنبه‌های XP به جسارت و جرأت نیاز دارد. یک واژه‌ی بهتر می‌تواند *انضباط* باشد. برای مثال، غالباً جهت طراحی برای خواسته‌های آینده، فشار وجود دارد. اکثر تیم‌های نرم‌افزاری هم سر تسلیم فرود می‌آورند با این استدلال که «طراحی برای آینده» در دراز مدت به صرفه جویی در زمان و تلاش کمک می‌کند. تیم XP چابک باید انضباط (جرأت) لازم برای طراحی برای امروز را داشته باشد و در عین حال بداند که خواسته‌های آینده ممکن است به‌طرزی چشمگیر تغییر کند و بنابراین، ممکن است به مقادیر معتنابهی از دوباره‌کاری در طراحی و کدهای پیاده‌سازی شده نیاز داشته باشد.

تیم چابک، با دنبال‌کردن هر کدام از این ارزش‌ها، *احترام* را در میان اعضای خود، در میان سایر طرف‌های ذی‌نفع و اعضای تیم و به‌طور مستقیم برای خود نرم‌افزار نهادینه می‌کند. با تحویل موفق نسخه‌های نرم‌افزار، این تیم احترام بیشتری برای فرایند XP جلب می‌کند.

> XP پاسخ این سؤال است که «چقدر می‌توان کوچک عمل کرد و هنوز نرم‌افزارهای بزرگ ساخت.»
>
> **ناشناس**

> **موجع وب**
>
> مروری عالی بر «قواعد» XP را می‌توانید در آدرس زیر بیابید:
>
> **www.extremeprogramming.org/rules.html**

۲-۴-۳ فرایند XP

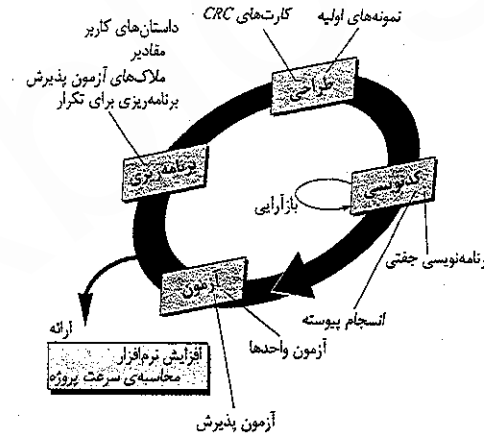
در برنامه‌نویسی حدی از یک روش شیءگرا (پیوست ۲) به‌عنوان الگوی توسعه استفاده می‌شود و این

---

[1] *استعاره* (metaphor) در حیطه‌ی XP داستانی است که هرکسی – مشتری، برنامه‌نویس، مدیر – می‌تواند درباره چگونگی کارکردن سیستم روایت کند.

[2] مهندس نرم‌افزار با بازآرایی‌کردن می‌تواند ساختار درونی یک طراحی (یا کد منبع) را بهبود بخشد بدون اینکه رفتار یا عملکرد آن را تغییر دهد. در اصل، بازآرایی را می‌توان برای بهبود بخشیدن به بازدهی، خوانایی، یا کارایی یک طراحی یا کدی به کاربرد که آن طراحی را پیاده‌سازی می‌کند.

فرایند شامل مجموعه‌ای از قواعد و اصول می‌شود که در حیطه‌ی چهار فعالیت چارچوبی رخ می‌دهند: برنامه‌ریزی، طراحی، کدنویسی و آزمون. در شکل ۲-۳، فرایند XP نشان داده شده است و برخی ایده‌های کلیدی و وظایف مرتبط با هر فعالیت چارچوبی نشان داده شده است. فعالیت‌های کلیدی XP در پاراگراف‌های زیر خلاصه شده‌اند.



شکل ۲-۳ فرایند برنامه‌نویسی حدی.

برنامه‌ریزی. فعالیت برنامه‌ریزی (که بازی برنامه‌ریزی نیز نامیده می‌شود) با گوش سپردن آغاز می‌شود- فعالیتی برای جمع‌آوری خواسته‌ها که اعضای تیم XP را قادر به شناخت حیطه‌ی تجاری مناسب برای نرم‌افزار ساخته، حس و حال گسترده‌ای از خروجی مورد نیاز و ویژگی‌ها و عملکردهای عمده به‌دست می‌دهد. گوش سپردن، به ایجاد مجموعه‌ای از «داستان ها» می‌انجامد که خروجی لازم، برای نرم‌افزاری که قرار است ساخته شود، ویژگی‌ها و عملکردها را توصیف می‌کند. هر *داستان* (مشابه use case ها در فصل ۵) توسط مشتری نوشته می‌شود و روی یک کارت شاخص قرار داده می‌شود. مشتری بر اساس ارزش تجاری کلی آن ویژگی یا عملکرد، یک *ارزش* (اولویت) به داستان نسبت می‌دهد.[1] سپس اعضای تیم XP هر کدام از داستان‌ها را ارزیابی کرده هزینه‌ای را به آن نسبت می‌دهند (این هزینه بر حسب تعداد هفته‌های لازم برای توسعه بیان می‌شود). اگر برآورد شود که داستانی برای توسعه به بیش از سه هفته زمان نیاز دارد، از مشتری خواسته می‌شود که داستان را به داستان‌های کوچکتر تقسیم کند و دوباره همان فرایند انتساب ارزش و هزینه رخ می‌دهد. لازم به ذکر است که نوشتن داستان‌های جدید در هر زمان امکان‌پذیر است.

مشتریان و سازندگان با همکاری یکدیگر تصمیم می‌گیرند که چگونه داستان‌ها را در نسخه‌ی بعدی (افزایش بعدی نرم‌افزار) که قرار است تیم XP توسعه دهد، گروه‌بندی کنند. هنگامی که قرار اولیه (توافق بر سر داستان‌هایی که باید لحاظ شود، تاریخ تحویل و سایر مواد پروژه) برای یک نسخه‌ی جدید گذاشته شد، تیم XP این داستان‌ها را مرتب می‌کند تا به یکی از سه شیوه‌ی زیر توسعه دهد: (۱) همه‌ی داستان‌ها بلافاصله پیاده‌سازی می‌شود (در عرض چند هفته)، (۲) داستان‌هایی با

[1] ارزش یک داستان ممکن است به وجود یک داستان دیگر نیز بستگی داشته باشد.

«داستان» در XP چیست؟

مرجع وب

یک بازی برنامه‌ریزی جالب را می‌توانید در وب‌سایت زیر بیابید:
C2.com/cgi/wiki?planningGame

بیشترین ارزش در بالای جدول زمان‌بندی قرار داده می‌شوند و ابتدا آن‌ها باید پیاده‌سازی شوند یا (۲) داستان‌هایی با بیشترین ریسک در بالای جدول زمان‌بندی قرار داده می‌شوند و ابتدا آنها پیاده‌سازی می‌شوند.

پس از ارائه‌ی نخستین نسخه‌ی پروژه (که «افزایش» نرم‌افزار نیز نامیده می‌شود)، تیم XP سرعت پروژه را محاسبه می‌کند. به بیان ساده، سرعت پروژه برابر با تعداد داستان‌های مشتری است که در نسخه‌ی نخست پیاده‌سازی شده‌اند. از سرعت پروژه می‌توان برای (۱) کمک به برآورد تاریخ‌های تحویل و زمان‌بندی برای روایت‌های بعدی و (۲) تعیین این نکته استفاده کرد که آیا برای همه‌ی داستان‌های ذکرشده در کل پروژه‌ی توسعه، زیاده‌روی شده است یا خیر. درصورت مشاهده‌ی زیاده‌روی، محتوای نسخه‌ها اصلاح یا تاریخ تحویل نهایی تغییر داده می‌شود.

با پیشرفت کار توسعه، مشتری می‌تواند داستان‌هایی اضافه کند، ارزش یک داستان موجود را تغییر دهد، داستان‌ها را تقسیم کند یا آنها را حذف نماید. سپس تیم XP دوباره به همه‌ی روایت‌های باقیمانده خواهد پرداخت و برنامه‌ریزی‌ها را بر همان اساس اصلاح می‌کند.

نکته‌ی کلیدی

سرعت پروژه، میزان ظرفیتی است از بهره‌وری تیم.

طراحی. طراحی XP قویاً از اصل KIS (حفظ سادگی) پیروی می‌کند. یک طراحی ساده، همواره بر ارائه‌ای پیچیده‌تر ترجیح داده می‌شود. به‌علاوه، در طراحی، راهنمای پیاده‌سازی برای هر داستان به موازات نوشته‌شدن آن ارائه می‌شود-نه چیزی کمتر و نه چیزی بیشتر. طراحی عملکرد اضافی (که سازنده تصور کند بعداً ممکن است لازم شود) تشویق نمی‌شود.[1]

در XP استفاده از کارت‌های CRC (فصل ۷) سازوکاری مؤثر برای تفکر درباره نرم‌افزارهای شیءگرا محسوب می‌شود. کارت‌های CRC (کلاس-مسؤولیت-همکار[2]) کلاس‌های شیءگرایی[3] را شناسایی و سازمان‌دهی می‌کنند که به نسخه‌ی فعلی نرم‌افزار مربوط می‌شوند. تیم XP عمل طراحی را با استفاده از فرایندی مشابه با فرایند توصیف شده در فصل ۸ اجرا می‌کند. کارت‌های CRC تنها محصول طراحی هستند که به‌عنوان بخشی از فرایند XP تولید می‌شود.

اگر در بخشی از طراحی یک داستان، یک مشکل طراحی مشاهد شود، XP ایجاد فوری یک نمونه‌ی اولیه‌ی عملیاتی را برای آن بخش از طراحی توصیه می‌کند. این نمونه‌ی اولیه طراحی که *راهکار خیزشی*[4] نامیده می‌شود، پیاده‌سازی و ارزیابی می‌شود. هدف از این کار، پایین‌آوردن خطر در هنگام پیاده‌سازی واقعی و نیز اعتبارسنجی برآوردهای اولیه برای داستان حاوی مسأله‌ی طراحی است.

در بخش قبل، گفتیم که در XP، بازآرایی ترغیب می‌شود-یک تکنیک ساخت که روشی برای بهینه‌سازی طراحی نیز هست. فاولر [Fow00]، بازآرایی را به‌شیوه زیر توصیف می‌کند:

> بازآرایی عبارت است از تغییردادن یک سیستم نرم‌افزاری به‌شیوه‌ای که رفتار خارجی کد را تغییر ندهد و در عین حال، ساختار داخلی را بهبود بخشد. شیوه‌ای منضبط برای پاک‌کردن کدها [و اصلاح/ساده‌سازی طراحی داخلی] است که احتمال واردشدن خطاها را کاهش می‌دهد. در اصل، هنگامی که بازآرایی می‌کنید، طراحی کدها را پس از نوشتن آنها بهبود می‌بخشید.

اندرز

در XP تأکید از روی اهمیت طراحی برداشته می‌شود که البته همه با آن موافق نیستند. در واقع، مواقعی پیش می‌آید که باید بر طراحی تأکید شود.

مرجع وب

ابزارها و تکنیک‌های بازآرایی کردن را در وب‌سایت زیر می‌توانید بیابید:
www.refactoring.com

[1] این دستورالعمل‌های طراحی باید در هر روش مهندسی طراحی دنبال شوند، هر چند مواردی هست که نمادگذاری و اصطلاح‌شناسی پیچیده در طراحی ممکن است سد راه سادگی شود.

[2] Class-Responsibility-Collaborator

[3] کلاس‌های شیءگرا در پیوست ۲، در فصل ۸ و در سرتاسر بخش دوم این کتاب بحث می‌شوند.

[4] spike solution

از آنجا که در طراحی XP در واقع از هیچ نمادگذاری استفاده نمی‌شود و غیر از کارت‌های CRC و راهکارهای خیزشی، محصولات زیادی (در صورت وجود) تولید نمی‌شود، طراحی به‌عنوان یک محصول گذرا در نظر گرفته می‌شود که در اثنای ساخت، پیوسته قابل اصلاح است و باید اصلاح شود.

هدف از بازآرایی، کنترل این اصلاحات ازطریق پیشنهاد تغییرات اندک در طراحی است که «می‌توانند به‌طور ریشه‌ای طراحی را بهبود بخشند» [Fow00]؛ ولی لازم به ذکر است که تلاش لازم برای بازآرایی می‌تواند با رشد اندازه‌ی برنامه‌ی کاربردی به‌طور چشمگیری رشد کند.

یک مفهوم محوری در XP آن است که طراحی هم قبل و هم بعد از شروع کدنویسی رخ می‌دهد. بازآرایی به این معناست که طراحی پیوسته به موازات ساخت سیستم انجام می‌گیرد. در واقع، فعالیت ساخت، خودش راهنمایی لازم برای چگونگی بهبود بخشیدن به طراحی را فراهم می‌سازد.

کدنویسی. پس از توسعه یافتن داستان‌ها و انجام‌شدن کارهای طراحی مقدماتی، تیم به کدنویسی نمی‌پردازد، بلکه یک سری «آزمون واحد» تهیه می‌کند که هر کدام از داستان‌هایی را که قرار است در نسخه‌ی فعلی لحاظ شوند، مورد آزمون قرار می‌دهد.[1] هنگامی که آزمون واحد[2] تهیه شد، سازنده بهتر می‌تواند توجه خود را به آن چیزی معطوف کند که باید پیاده‌سازی شود تا آزمون را با موفقیت پشت سر بگذارد. هیچ چیز فرعی اضافه نمی‌شود (KIS). هنگامی که کدها کامل شدند، می‌توان آزمون واحدی را بلافاصله انجام داد و در نتیجه، بازخوردی فوری در اختیار سازنده قرار داده می‌شود.

یک مفهوم کلیدی طی فعالیت کدنویسی (و یکی از بحث انگیزترین جنبه‌های XP) *برنامه‌نویسی جفتی* است. XP توصیه می‌کند که دو نفر با هم روی یک ایستگاه کاری کار کنند و کد مربوط به یک داستان را بنویسند. به این ترتیب، سازوکاری برای حل مسأله به‌صورت بی‌درنگ (دو فکر غالباً بهتر از یکی است) و تضمین کیفیت بی‌درنگ (کد به محض نوشته‌شدن بازبینی می‌شود) فراهم می‌شود. در این روش، سازندگان نیز باید پیوسته به مسأله مورد نظر توجه داشته باشد. در عمل، هر شخصی دارای نقشی است که قدری با دیگران متفاوت است. برای مثال، یک شخص ممکن است درباره جزئیات کدنویسی بخش خاصی از طراحی فکر کند، در حالی که دیگری اطمینان حاصل کند که استانداردهای کدنویسی (بخشی لازم از XP) رعایت می‌شوند یا کد مربوط به داستان، آزمون واحدی را که برای اعتبارسنجی کد از نظر داستان تدارک دیده شده است، با موفقیت می‌گذارند.

به موازاتی که برنامه‌نویسان جفتی کار خود را کامل می‌کنند، کدی که می‌نویسند در کنار کار دیگران قرار داده می‌شود. در برخی موارد، این کار به‌صورت روزانه توسط تیم انسجام دهنده انجام می‌شود. در موارد دیگر، برنامه‌نویسان جفتی مسؤولیت انسجام بخشی را نیز برعهده دارند. این راهبرد یعنی «انسجام بخشی پیوسته» به پرهیز از مشکلات سازگاری و ایجاد واسط کمک می‌کند و یک محیط آزمون دود (smoke testing) فراهم می‌سازد (فصل ۱۷) که به کشف زود هنگام خطاها کمک می‌کند.

نکته‌ی کلیدی

بازآرایی کردن، ساختار درونی طراحی (یا کد منبع) را بهبود می‌بخشد بدون اینکه عملکرد یا رفتار خارجی را تغییر دهد.

مرجع وب

از وب‌سایت زیر می‌توانید اطلاعات حالی درباره XP به‌دست آورید:
www.xprogramming

برنامه‌نویسی جفتی چیست؟

اندرز

اگر تیم‌های نرم‌افزاری پر هستند از آدم‌های تک‌رو، اگر می‌خواهید برنامه‌نویسی جفتی پیاده‌سازی شود، باید ابتدا ذهنیت آنها را تغییر دهید.

[1] این روش مشابه آن است که سؤالات امتحانی را قبل از مطالعه بدانید. مطالعه مطالب فقط با بذل توجه به سؤالاتی که پرسیده می‌شوند، کار بسیار آسان‌تر می‌شود.

[2] در آزمون واحدها، که به تفصیل در فصل ۱۷ بحث خواهد شد، تنها یک مؤلفه از نرم‌افزار مورد توجه قرار می‌گیرد، واسط آن مؤلفه، ساختمان‌های داده‌ها و عملکرد مورد بررسی قرار می‌گیرد تا خطاهای موجود در آن مؤلفه تشخیص داده شود.

شیوه‌ی استفاده از آزمون واحدها در XP از چه قراری است؟

آزمون. پیش از این گفتیم که ایجاد آزمون واحدها قبل از شروع کدنویسی، از ویژگی‌های کلیدی روش XP است. آزمون واحدها که ایجاد می‌شوند باید با استفاده از چارچوبی پیاده‌سازی شوند که آنها را قادر به خودکارسازی کند (تا به این ترتیب بتوان آنها را به سهولت و به کرات اجرا کرد). بنابراین، هرگاه که کدها اصلاح شوند، راهبرد آزمون رگرسیون (فصل ۱۷) مفید واقع می‌شود (که غالباً فلسفه *بازآرایی* کد نامیده می‌شود).

با سازمان‌دهی آزمون واحدها در قالب یک «مجموعه آزمون سرتاسری» [Wel99]، آزمون انسجام و اعتبارسنجی سیستم را می‌توان به‌صورت روزانه انجام داد. به این ترتیب، تیم XP پیوسته در جریان پیشرفت کار قرار خواهد داشت و می‌تواند در صورت خراب‌شدن اوضاع، در همان ابتدای امر هشدارهای لازم را صادر کند. ولز [Wel99] می‌گوید: «برطرف‌کردن مشکلات کوچک در هر چند ساعت یک بار، نسبت به برطرف‌کردن مشکلات بزرگ درست قبل از پایان مهلت، زمان‌کمتری می‌برد.

آزمون‌های پذیرش[1] XP، که آزمون‌های مشتری نیز نامیده می‌شوند، توسط مشتری مشخص می‌شوند و شامل آن دسته از ویژگی‌های سیستم می‌شوند که مشتری قادر به دیدن آنهاست و می‌تواند آنها را مرور کند. آزمون‌های پذیرش از داستان‌های کاربران به‌دست می‌آیند که به‌عنوان بخشی از نسخه‌ی نرم‌افزار، پیاده‌سازی شده‌اند.

نکته‌ی کلیدی

آزمون‌های پذیرش XP از داستان‌های کاربر به‌دست می‌آیند.

۳-۴-۳ XP صنعتی

جاشوا کریوسکی [Ker05] برنامه‌نویسی حدی صنعتی (IXP) را به‌شیوه زیر تعریف می‌کند: «IXP تکاملی ارگانیک از XP است. این روش از کمینه‌گرایی، مشتری‌مداری و آزمون‌مداری XP الهام گرفته است. بیشترین تفاوت IXP با XP، اِعمال مدیریت بیشتر، گسترش نقش مشتریان و ارتقای روش‌های فنی است.» IXP شامل شش عمل جدید می‌شود که برای کمک به حصول اطمینان از عملکرد موفق پروژه‌ی XP در یک سازمان بزرگ طراحی می‌شوند.

چه چیزهای جدیدی به XP افزوده می‌شود تا IXP حاصل آید؟

ارزیابی آمادگی. پیش از شروع یک پروژه‌ی IXP، سازمان باید *ارزیابی آمادگی* را انجام دهد. در این عمل اطمینان حاصل می‌شود که: (۱) یک محیط توسعه‌ی مناسب وجود دارد که IXP را پشتیبانی می‌کند، (۲) تیم شامل مجموعه مناسبی از طرف‌های ذی‌نفع است، (۳) سازمان دارای یک برنامه‌ی کیفیت متمایز است و بهبود مستمر را پشتیبانی می‌کند، (۴) فرهنگ سازمانی، پشتیبان ارزش‌های جدید یک تیم جدید است و (۵) جامعه‌ی وسیع‌تر پروژه از افراد مناسبی تشکیل شده است.

جامعه‌ی پروژه. در XP کلاسیک، پیشنهاد می‌شود که تیم چابک از افراد مناسب تشکیل شود تا تیم در کارش موفق شود. این بدان معناست که افراد تیم باید به‌خوبی آموزش دیده باشند، انطباق‌پذیر باشند، از مهارت‌های لازم برخوردار باشند و به لحاظ شخصیتی قادر به شرکت در تیم‌های «خودسازمان‌ده» باشند. وقتی قرار باشد که برای یک پروژه‌ی مهم در سازمانی بزرگ از روش XP استفاده شود، مفهوم تیم باید به *«جامعه»* تغییر شکل پیدا کند. این جامعه می‌تواند شامل یک متخصص فن‌آوری و مشتریانی باشد که در موفقیت پروژه نقش محوری دارند و نیز شامل طرف‌های ذی‌نفع (نظیر کارمندان حقوقی، ممیزان کیفیتی و کارمندان بخش تولید و فروش) باشد

«توانایی به آن چیزی گفته می‌شود که قادر به انجام آن هستید. انگیزه تعیین می‌کند که چه کار می‌کنید. نگرش تعیین می‌کند که چقدر خوب آن کار را انجام می‌دهید.»
لو هولتز

[1] acceptance test

که « غالباً در حاشیه‌ی پروژه‌ی IXP قرار دارند و در عین حال نقش‌های مهمی در پروژه داشته باشند» [Ker05]. در IXP، اعضای جامعه و نقش هرکدام از آنها باید به صراحت تعریف شود و سازوکارهای مربوط به برقراری ارتباط و هماهنگی میان اعضای جامعه باید برقرار گردد.

چارترکردن پروژه. تیم IXP، خود به ارزیابی پروژه می‌پردازد تا تعیین کند آیا یک توجیه تجاری مناسب برای پروژه وجود دارد و آیا پروژه اهداف و مقاصد کلی سازمان را پیش می‌برد یا خیر. با عمل چارتر کردن، حیطه‌ی پروژه برای تعیین چگونگی کامل‌شدن، توسعه یافتن یا جایگزین ساختن سیستم‌ها یا فرایندهای موجود تعیین می‌شود.

مدیریت مبتنی بر آزمون. یک پروژه‌ی IXP نیاز به ملاک‌های قابل سنجش برای ارزیابی وضعیت پروژه و میزان پیشرفت آن دارد. در مدیریت مبتنی بر آزمون، یک سری «مقاصد» قابل سنجش تعیین می‌شود [Ker05] و سپس سازوکارهایی برای دستیابی به این مقاصد تعریف می‌شود.

بازنگری (Retrospective). یک تیم IXP پس از تحویل نسخه‌ی جدید نرم‌افزار آن را مورد بازبینی فنی و تخصصی قرار می‌دهد (فصل ۱۵). در این مرور و بازبینی که *بازنگری* نامیده می‌شود، «مسائل، رویدادها، و درس‌های آموخته شده» در طول یک نسخه از نرم‌افزار و/یا کل نسخه‌ی نرم‌افزار بررسی می‌شود. هدف، بهبودبخشیدن به فرایند IXP است.

آموزش پیوسته. از آنجا که آموزش، بخشی حیاتی از بهبود مستمر فرایند است، اعضای تیم XP تشویق می‌شوند تا روش‌ها و تکنیک‌های جدیدی را بیاموزند که به محصول با کیفیت بالاتر منجر شود.

علاوه بر شش عملی که در بالا بحث شد، IXP چند عمل موجود در XP را نیز اصلاح می‌کند. *توسعه مبتنی بر داستان* (SDD) اصرار دارد که داستان‌های مربوط به آزمون‌های پذیرش پیش از تولید حتی یک خط از کد نوشته شوند. *طراحی مبتنی بر دامنه*[1] (DDD) بهبودی بر مفهوم «استعاره‌ی سیستم» (system metaphor) است که در XP استفاده می‌شود. DDD [Eva03] ایجاد تکاملی یک مدل دامنه‌ای را توصیه می‌کند که «به‌طور صحیح چگونگی تفکر کارشناسان دامنه را درباره موضوع به نمایش می‌گذارد.» [Ker05]. *جفت‌کردن* (pairing)، مفهوم برنامه‌نویسی جفتی را بسط می‌دهد، به‌طوری که مدیران و طرف‌های ذی‌نفع را نیز در برگیرد. هدف از بهبود بخشیدن به اشتراک معلومات در میان اعضایی از تیم XP است که ممکن است در توسعه‌ی فنی، شرکت مستقیم نداشته باشند. *قابلیت کاربرد تکراری* (iterative usability)، طراحی واسط‌های پر زرق و برق را به نفع طراحی کاربردگرا مردود می‌داند، به‌طوری که نتیجه‌ی آن نسخه‌های نرم‌افزاری است که تحویل می‌شوند و تعامل کابر با نرم‌افزار مطالعه می‌شود.

IXP در سایر عملیات XP اصلاحات کوچکی به عمل می‌آورد و نقش‌ها و مسؤولیت‌های معینی را دوباره تعریف می‌کند تا برای پروژه‌های مهم در سازمان‌های بزرگ مناسب‌تر شوند. برای بحث بیشتر درباره IXP، وب‌سایت **http://industrialx.org** را ببینید.

۴-۴-۳ مشاجره‌ی XP

همه‌ی روش‌ها و مدل‌های فرایند جدید باعث ایجاد بحث‌های ارزش‌مند می‌شوند که در برخی موارد

[1]

این بحث‌ها به مشاجره منجر می‌گردد. برنامه‌نویسی حدی نیز از این قاعده مستثنا نبوده است. استفنز و روزنبرگ [Ste03] در کتاب جالبی که اثربخشی XP را بررسی کرده‌اند چنین استدلال می‌کنند که بسیاری از کارهای XP ارزش‌مند هستند، ولی در مورد تعدادی دیگر گزافه‌گویی شده است و چند تایی هم اصلاً مشکل‌آفرین هستند. این نویسندگان پیشنهاد می‌کنند که قابلیت‌های XP هم دارای نقاط قوت و هم دارای نقاط ضعف است. از آنجا که بسیاری از سازمان‌ها فقط زیر مجموعه‌ای از قابلیت‌های XP را به‌کار می‌گیرند، اثربخشی کل فرایند را تضعیف می‌کنند. مدافعان آن در مخالفت با این نظر می‌گویند که XP پیوسته در حال تکامل است و بسیاری از مسائلی که منتقدان مطرح می‌کنند، با بلوغ XP برطرف شده‌اند. از میان مسائلی که همچنان ذهن منتقدان XP را مشغول داشته است، می‌توان به موارد زیر اشاره کرد[1]:

کدام مسائل هستند که به مشاجره XP منتهی می‌شوند؟

- *متغیربودن خواسته‌ها.* چون مشتری عضو فعالی از تیم XP است، تغییراتی که در خواسته‌ها به عمل می‌آید به‌صورت غیر رسمی تقاضا می‌شود. در نتیجه، ممکن است حوزه‌ی پروژه تغییر کند و کارهای اولیه برای پاسخ‌گویی به نیازهای جدید، نیاز به اصلاح داشته باشد. مدافعان، چنین استدلال می‌کنند که این اتفاق در هر نوع فرایند دیگری نیز ممکن است رخ دهد و XP سازوکاری برای کنترل خزش حوزه (scope creep) فراهم می‌آورد.
- *نیازهای متناقض مشتریان.* بسیاری از پروژه‌ها چند مشتری دارند که هر یک مجموعه نیازهای خاص خود را دارد. در XP، خود تیم است که باید نیازهای مشتریان متفاوت را به رسمیت بشناسد و این وظیفه‌ای است که ممکن است خارج از حوزه‌ی مسؤولیت تیم باشد.
- *خواسته‌ها به‌صورت غیر رسمی بیان می‌شوند.* داستان‌های کاربران و آزمون‌های پذیرش، تنها نمود بارز خواسته‌ها در XP به‌شمار می‌روند. منتقدان چنین استدلال می‌کنند که برای حصول اطمینان از اینکه چیزی از قلم نیفتاده است و ناسازگاری‌ها و خطاها پیش از ساخته‌شدن سیستم کشف می‌شوند، به مدلی رسمی‌تر نیاز است. مدافعان با این نظر مخالف هستند و می‌گویند ماهیت متغیر خواسته‌ها این مدل‌ها را تقریباً به محض توسعه یافتن از رده خارج می‌کنند.
- *فقدان طراحی رسمی.* XP در بسیاری از نمونه‌ها تأکیدی بر طراحی معماری ندارد و پیشنهاد می‌کند که طراحی از هر نوع باید نسبتاً غیر رسمی باشد. منتقدان چنین استدلال می‌کنند که هنگام ساخت سیستم‌های پیچیده، باید بر طراحی تأکید گردد تا اطمینان حاصل شود که ساختار کلی نرم‌افزار، کیفیت و قابلیت نگهداری لازم را از خود نشان دهد. مدافعان XP هم استدلال می‌کنند که ماهیت افزایشی فرایند XP، پیچیدگی را محدود می‌سازد (سادگی یک ارزش محوری است) و از این رو، نیاز به طراحی گسترده را کاهش می‌دهد.

شایان توجه است که هر فرایند نرم‌افزار دارای نقایصی است و بسیاری از سازمان‌های نرم‌افزاری، XP را با موفقیت به‌کار گرفته‌اند. مهم این است که بدانید ضعف فرایند در کجاست و آن را بر نیازهای خاص سازمان خود وفق دهید.

[1] برای نگاهی مفصل به یک نقد اندیشمندانه از XP به وب‌سایت زیر مراجعه کنید.
www.softwarereality.com/Extremeprogramming.jsp

در نظر گرفتن فرایند چابک

صحنه: دفتر داگ میلر

نقش آفرینان: داگ میلر، مدیر مهندسی نرم‌افزار؛ جیمی لازار، عضو تیم نرم‌افزاری؛ وینود رامان، عضو تیم نرم‌افزاری.

گفتگوها:

(ضربه‌ای به در می‌خورد و جیمی با وینود وارد دفتر داگ می‌شوند)

جیمی: داگ، یک دقیقه وقت داری؟

داگ: البته جیمی. چه خبر شده؟

جیمی: ما داشتیم درباره یحتی صحبت می‌کردیم که دیروز درباره فرایند داشتیم... اینکه قرار است چه فرایندی را برای این پروژه‌ی جدید ***SafeHome*** انتخاب کنیم.

داگ: خب؟

وینود: من با یکی از دوستانم در یک شرکت دیگر حرف زدم و او از برنامه‌نویسی حدی صحبت می‌کرد. یک مدل فرایند چابک است. چیزی درباره‌اش نشنیدی؟

داگ: اره. یک سری چیزهای خوب و یک سری چیزهای بد.

جیمی: خوب برای ما خیلی خوب به‌نظر می‌آید. با این مدل می‌توانی نرم‌افزار را واقعاً به سرعت توسعه بدهی؛ از یک چیزی به اسم برنامه‌نویسی جفتی استفاده می‌شود که در این صورت امکان کنترل کیفیت در همان موقع را می‌دهد... به‌نظر من که خیلی خوب است.

داگ: ایده‌های واقعاً خوب، زیاد دارد. مثلاً از همین مفهوم برنامه‌نویسی جفتی خیلی خوشم می‌آید و اینکه طرف ذی‌نفع هم باید عضوی از تیم باشد.

جیمی: جی؟ منظورت این است که بازاریابی هم با ما در تیم پروژه باید کار کند؟

داگ (در حالی که سرش را تکان می‌دهد): خب آنها هم از طرف‌های ذی‌نفع هستند دیگر.

جیمی: خدایا... در این صورت هر پنج دقیقه درخواست تغییرات دارند.

وینود: لزومی ندارد. دوستم گفت برای رفتن به استقبال تغییرات در طول پروژه‌ی XP روش‌هایی هست.

داگ: پس شماها فکر می‌کنید باید روش XP را انتخاب کنیم؟

جیمی: قطعاً ارزش دارد که به آن فکر کنیم.

داگ: موافقم. و حتی اگر یک مدل افزایشی را به‌عنوان رویکرد انتخاب کنیم، دلیلی ندارد که نتوانیم آن را با خیلی از مزایای XP همراه کنیم.

وینود: داگ، قبلاً گفتی یک سری چیزهای خوب و یک سری چیزهای بد. بدش چه بود؟

داگ: چیزی که خوشم نمی‌آید این است که XP نقش تحلیل و طراحی را کمرنگ می‌کند... یک جورهایی می‌خواهد بگوید که نوشتن کد نقطه شروع...

(اعضای تیم به هم نگاه می‌کنند و لبخند می‌زنند)

داگ: پس با روش XP موافق هستید؟

جیمی (از طرف هردو): رییس، ما کارمان کدنویسی است!

داگ (با خنده): درست است، ولی دوست دارم ببینم که زمان کمتری را صرف کدنویسی و بعد کدنویسی دوباره کنید و در عوض کمی بیشتر وقت بگذارید تا چیزی را که قرار است انجام شود، تحلیل و راهکار را طراحی کنید.

وینود: شاید بتوانیم هر دو تا را با هم داشته باشیم، چابکی با قدری انضباط.

داگ: من فکر می‌کنم بتوانیم وینود. در واقع شک ندارم.

۳-۵ سایر مدل‌های فرایند چابک

تاریخ مهندسی نرم‌افزار آکنده است از ده‌ها فرایند و روش‌شناسی، مفاهیم و روش‌های مدل‌سازی، ابزارها و فن‌آوری که دیگر از آنها استفاده نمی‌شود. هر یک مشکلاتی داشته است که چیز بهتر و جدیدتری جایگزین آن شده است. جنبش چابک نیز با واردکردن آرایه گسترده‌ای از مدل‌های فرایند جدید-که هر یک برای پذیرفته‌شدن در جامعه‌ی نرم‌افزاری با بقیه در حال رقابت است-همان مسیر را دنبال می‌کند.[1]

چنان که در بخش قبل گفته شد، پرکاربردترین مدل فرایند چابک، برنامه‌نویسی حدی (XP) است، ولی مدل‌های فرایند چابک دیگری پیشنهاد شده‌اند و در صنعت مورد استفاده قرار گرفته‌اند. از میان متداول‌ترین آنها می‌توان به موارد زیر اشاره نمود:

- توسعه‌ی وفقی نرم‌افزار (ASD)[2]
- اسکرام (scrum)
- روش توسعه سیستم‌های پویا (DSDM)
- کریستال
- توسعه‌ی ویژگی‌محور (FDD)
- توسعه‌ی نرم‌افزار ناب (LSD)
- مدل‌سازی چابک (AM)
- فرایند یکپارچه‌ی چابک (AUP)

در بخش‌هایی که به‌دنبال خواهد آمد، نگاهی بسیار مختصر به هر کدام از این مدل‌های فرایند چابک خواهیم داشت. توجه به این نکته ضروری است که تمامی این مدل‌های فرایند چابک (کم و بیش) از بیانیه‌ی توسعه‌ی نرم‌افزاری چابک و اصول ذکر شده در بخش ۱-۳-۳ پیروی می‌کنند. برای جزئیات بیشتر به مراجع ذکر شده در هر بخش رجوع کنید و برای تحقیق بیشتر، درایه «agile software development» را در ویکی‌پدیا ببینید.[3]

«حرفه ما در حال تجربه‌ی روش‌شناسی‌های گوناگون است. درست همان طور که یک نوجوان ۱۴ ساله در حال تجربه لباس‌های مختلف است.»

استفن هاوریش و جیم روپرخت

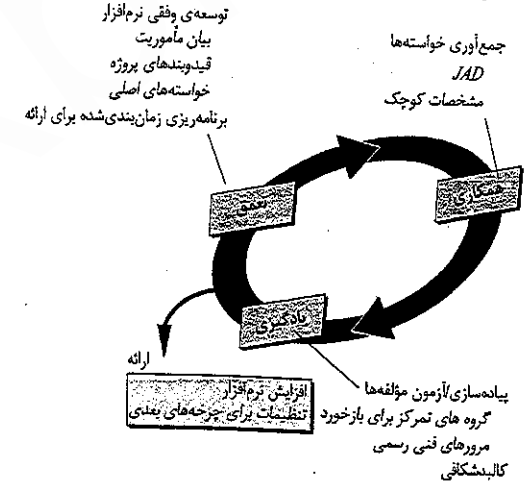
[1] این چیز بدی نیست. پیش از پذیرفته شدن یک یا چند مدل یا روش به‌عنوان استانداردی غیر رسمی، همه باید برای به‌دست آوردن موافقت مهندسان نرم‌افزار با هم رقابت کنند. «برنده ها» به بهترین روش تکامل می‌یابند در حالی که «بازنده‌ها» یا ناپدید می‌شوند یا در مدل‌های برنده ادغام می‌شوند.

[2] Adaptive Software Development

[3] http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods را ببینید.

۳-۵-۱ توسعه‌ی وفقی نرم‌افزار (ASD)

توسعه‌ی وفقی نرم‌افزار (ASD) را جیم های‌اسمیت [Hig00] به‌عنوان تکنیکی برای ساخت نرم‌افزارها و سیستم‌های پیچیده پیشنهاد کرده است. پایه‌های فلسفی ASD بر همکاری انسانی و خودسازمان‌دهی تیمی تأکید دارند. های‌اسمیت چنین استدلال می‌کند که یک روش چابک و وفقی برای توسعه‌ی نرم‌افزار که مبتنی بر همکاری است، در تعامل‌های پیچیده ما به اندازه‌ی انضباط و مهندسی می‌تواند منبعی از نظم و ترتیب باشد. او یک «چرخه حیات» برای ASD تعریف می‌کند (شکل ۳-۳) شامل سه مرحله‌ی تعمق (تفکر)، همکاری و یادگیری می‌شود.



شکل ۳-۳ توسعه‌ی وفقی نرم‌افزار

در طی مرحله‌ی *تعمق*، پروژه آغاز می‌شود و *برنامه‌ریزی چرخه‌ی وفقی* اجرا می‌شود. در برنامه‌ریزی چرخه‌ی وفقی از اطلاعات شروع پروژه-بیان ماموریت مشتری، قیدوبندهای پروژه (از قبیل تاریخ تحویل و توصیف‌های کاربر) و خواسته‌های پایه-برای تعیین مجموعه‌ای از چرخه‌های نسخه‌های نرم‌افزار (افزایش‌های نرم‌افزار) استفاده می‌شود که برای پروژه، مورد نیاز است.

طرح چرخه‌ای، صرف نظر از اینکه چقدر کامل باشد و تا چه حد در آن دوراندیشی منظور شده باشد، بی تردید تغییر خواهد کرد. بر اساس اطلاعات به‌دست آمده در تکمیل چرخه‌ی نخست، طرح طوری بازبینی و تنظیم می‌شود که کار برنامه‌ریزی شود و بهتر با واقعیت‌های فراروی تیم ASD همخوانی داشته باشد.

افرادی که از انگیزه کافی برخوردارند از «*همکاری*» به‌شیوه‌ای استفاده می‌کنند که استعداد و خروجی خلاقانه آنها چند برابر شود. این رویکرد، زمینه‌ای است که در همه‌ی روش‌های چابک به چشم می‌خورد، ولی همکاری آسان نیست. چیزی است که شامل برقراری ارتباط و کار تیمی می‌شود، ولی در عین حال بر فردگرایی نیز تأکید دارد زیرا خلاقیت فردی نقشی مهم در تفکر همکاری دارد. گذشته از همه‌ی اینها، موضوع اعتماد نیز در میان است. افرادی که با هم کار می‌کنند، باید به یکدیگر اعتماد داشته باشند تا (۱) بدون غرض‌ورزی انتقاد کنند، (۲) بدون ناراحتی کمک کنند، (۳) به‌سختی دیگران یا سخت‌تر از آنها کار کنند، (۴) مجموعه مهارت‌های لازم برای کار مورد نظر را داشته باشند و (۵) مسائل و دغدغه‌ها را به‌شیوه‌ای با یکدیگر در میان بگذارند که به کنش مؤثر بینجامد.

مرجع وب

منابع مفیدی برای ASD را می‌توانید در وب‌سایت زیر بیابید:
www.adptivesd.com

اندرز

همکاری مؤثر با مشتری تنها در صورتی امکان‌پذیر خواهد بود که هرگونه نگرش «ما و آنها» را کار بگذارید.

نکته‌ی کلیدی

ASD بر یادگیری به‌عنوان عنصری کلیدی در دستیابی به تیم «خود سازمان ده» تأکید دارد.

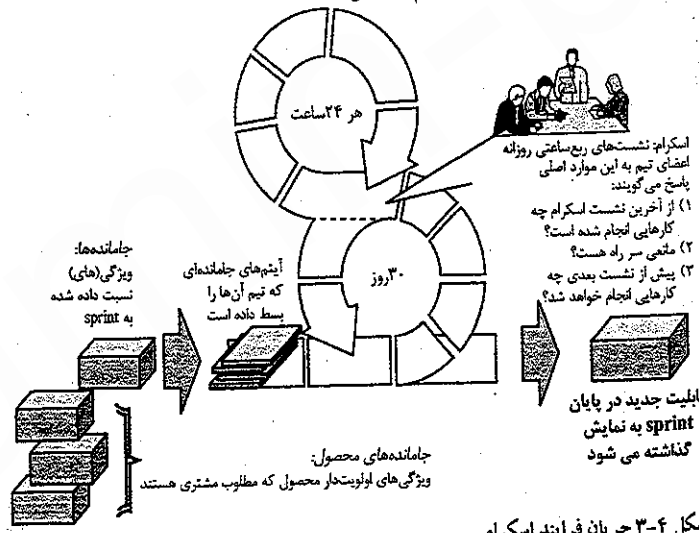
به موازاتی که اعضای تیم ASD شروع به توسعه‌ی مؤلفه‌های یک چرخه‌ی وفقی می‌کنند، تا پایان یافتن آن چرخه، «یادگیری» مورد تأکید قرار می‌گیرد. در واقع، های‌اسمیت [Hig00] استدلال می‌کند که سازندگان نرم‌افزار غالباً درک خود (از فن‌آوری، فرایند و پروژه‌ها) را بیش از مقدار واقعی برآورد می‌کنند و این یادگیری به آنها کمک می‌کند تا سطح شناخت واقعی خود را بهبود بخشند. تیم‌های ASD به سه شیوه می‌توانند یاد بگیرند: گروه‌های توجه (فصل ۵)، بازبینی‌های فنی (فصل ۱۴) و کالبدشکافی پروژه.

فلسفه ASD صرف نظر از نوع مدل فرایند مورد استفاده دارای مزایاست. تأکید کلی ASD بر پویایی تیم‌های خود سازمان ده، همکاری میان افراد و یادگیری فردی و تیمی، به تشکیل تیم‌هایی منجر می‌شود که احتمال موفقیت آنها بسیار بالاتر است.

۳-۵-۲ اسکرام (Scrum)

اسکرام (این نام از فعالیتی گرفته شده است که در بازی راگبی رخ می‌دهد[1]) یک روش توسعه‌ی چابک است که توسط جف ساترلند و تیم توسعه او در اوایل دهه ۱۹۹۰ شکل گرفت. در سال‌های اخیر، توسعه‌ی بیشتر روش‌های اسکرام، توسط شوابر و بیدل [Sch01a] انجام شده است.

اصول اسکرام با بیانیه‌ی چابکی سازگاری دارد و از آنها در هدایت فعالیت‌های توسعه در فرایندی استفاده می‌شود که شامل فعالیت‌های چارچوبی زیر می‌شود: خواسته‌ها، تحلیل، طراحی، تکامل و تحویل. در داخل هر کدام از این فعالیت‌های چارچوبی، وظایف کاری در یک الگوی فرایند موسوم به *sprint* رخ می‌دهد (که در پاراگراف بعدی شرح داده خواهد شد). کاری که در هر sprint انجام می‌شود (تعداد sprintهای لازم برای هر فعالیت چارچوبی بسته به اندازه و پیچیدگی محصول متغیر است) بر مسأله‌ی مورد نظر وفق داده می‌شود و در همان زمان توسط تیم اسکرام تعریف و غالباً اصلاح می‌شود. جریان کلی فرایند اسکرام در شکل ۳-۴ نشان داده شده است.



شکل ۳-۴ جریان فرایند اسکرام.

مرجع وب

اطلاعات و منابع مفیدی درباره اسکرام را می‌توانید در آدرس زیر بیابید:
www.controlchaos.com

[1] گروهی از بازیکنان دور توپ حلقه می‌زنند و هم‌تیمی‌ها با هم کارمی‌کنند (گاهی با خشونت!) تا توپ را به‌پایین میدان برسانند.

نکته‌ی کلیدی

اسکرام شامل مجموعه‌ای از الگوهای فرایند می‌شود که بر اولویت‌های پروژه، واحدهای کاری قطعه‌بندی شده، ارتباطات و بازخورد پیاپی از مشتری تأکید دارد.

اسکرام بر کاربرد مجموعه‌ای از الگوهای فرایند نرم‌افزار تأکید دارد [Noy02] که برای پروژه‌هایی با مهلت زمانی فشرده، خواسته‌های در حال تغییر و پروژه‌های تجاری بحرانی مفید واقع شده‌اند. در هر کدام از این الگوهای فرایند، مجموعه‌ای از کنش‌های توسعه تعریف می‌شود:

فهرست جامانده‌ها (backlogs)–فهرستی اولویت‌بندی شده از خواسته‌های پروژه یا ویژگی‌هایی که برای مشتری ارزش تجاری به همراه دارند. در هر زمان می‌توان به این فهرست چیزی اضافه کرد (این گونه است که تغییرات وارد می‌شوند). مدیر تولید فهرست جامانده‌ها را ارزیابی می‌کند و اولویت‌ها را در صورت نیاز بهنگام می‌کند.

sprint– شامل واحدهای کاری می‌شوند که برای دستیابی به خواسته‌ای تعیین شده در فهرست جامانده‌ها لازم هستند و این واحدها باید در یک کادر زمانی[1] ازپیش‌تعیین‌شده (معمولاً ۳۰ روزه) بگنجند. تغییرات (مثلاً اقلام کاری فهرست جامانده‌ها) در طول sprint وارد نمی‌شوند. از این رو، sprint به اعضای تیم این امکان را می‌دهد که در محیطی کوتاه مدت، ولی پایدار کار کنند.

نشست‌های اسکرام– جلساتی کوتاه (معمولاً ۱۵ دقیقه‌ای) هستند که هر روز توسط تیم اسکرام برگزار می‌شوند. سه پرسش مهم پرسیده می‌شود که همه‌ی اعضای تیم باید به آن پاسخ گویند [Noy02]:

- از آخرین جلسه‌ای که تیم داشت چه کار کردید؟
- با چه موانعی مواجه شدید؟
- در جلسه‌ی بعدی چه چیزی برای ارائه دارید؟

رهبر تیم که به او *استاد اسکرام* گفته می‌شود، جلسه را اداره می‌کند و پاسخ‌های هر کدام از اعضای تیم را مورد ارزیابی قرار می‌دهد. جلسه اسکرام به تیم کمک می‌کند تا مشکلات بالقوه را هر چه زود هنگام تر، کشف و برملا سازد. به‌علاوه، این نشست‌های روزانه به «جمعی‌شدن آگاهی‌ها» منجر می‌شود [Bee99] و از این رو یک ساختار تیمی خودسازمان‌ده را ارتقا می‌بخشد.

دموها– نسخه‌ی نرم‌افزاری را به مشتری تحویل می‌دهید تا عملکرد پیاده‌سازی شده را به نمایش در آورید و مشتری بتواند آن را ارزیابی کند. توجه به این نکته شایان اهمیت است که دمو ممکن است حاوی همه‌ی عملکردهای برنامه‌ریزی شده نباشد بلکه فقط آنهایی را ارائه دهد که در داخل کادر زمانی مشخص شده قابل تحویل بوده‌اند.

بیدل و همکارانش [Bee99] بحثی جامع درباره این الگوها ارائه داده‌اند و در آن چنین عنوان کرده‌اند: «در اسکرام وجود آشوب، امری غیر محال فرض می‌شود...» تیم نرم‌افزاری به کمک الگوهای اسکرام می‌تواند با موفقیت در دنیایی به‌کارش ادامه دهد که در آن، عدم قطعیت پدیده‌ای ذاتی است.

۳-۵-۳ روش توسعه سیستم‌های پویا (DSDM)

روش توسعه سیستم‌های پویا (DSDM) [Sta97] یک روش دیگر توسعه‌ی نرم‌افزار چابک است. این روش، چارچوبی برای ساخت و نگهداری سیستم‌هایی فراهم می‌آورد که قیدوبندهای زمانی فشرده را از طریق به‌کارگیری نمونه‌ی اولیه در یک محیط پروژه کنترل شده برآورده می‌سازند [CCS02]. فلسفه‌ی DSDM از یک نسخه‌ی تصحیح شده‌ی اصل پارتو به عاریت گرفته شده است (۸۰ درصد از

مرجع وب

منابع مفیدی برای DSSD را می‌توان در آدرس زیر یافت:
www.dsdm.org

[1] *کادر زمانی* (time box) عبارتی در مدیریت پروژه است (بخش چهارم این کتاب) و یک دوره زمانی را نشان می‌دهد که به انجام یک وظیفه مشخص اختصاص داده می‌شود.

یک برنامه‌ی کاربردی را می‌توان در ۲۰ درصد از زمان لازم برای تحویل برنامه‌ی کاربردی کامل (۱۰۰ درصد) تحویل داد).

DSDM یک فرایند نرم‌افزار تکراری است که در آن هر دور تکرار از قاعده ۸۰ درصد پیروی می‌کند. یعنی برای هر نسخه به کار کافی نیاز است تا حرکت به سوی نسخه‌ی بعدی تسهیل گردد. جزئیات باقی مانده را بعداً می‌توان کامل کرد، یعنی هنگامی که خواسته‌های تجاری بیشتری مشخص شود یا تغییراتی درخواست و انجام شوند.

کنسرسیوم DSDM (www.dsdm.org) یک گروه جهانی از شرکت‌هاست که در کنار یکدیگر وظیفه‌ی حفظ این روش را بر عهده دارند. این کنسرسیوم، یک مدل فرایند چابک موسوم به چرخه‌ی حیات DSDM تعریف کرده است که سه چرخه‌ی تکرار متفاوت را مشخص می‌کند و دو فعالیت چرخه‌ی حیاتی اضافی قبل از آن انجام می‌شود:

امکان‌سنجی (feasibility study) – خواسته‌های تجاری پایه و قیدوبندهای مرتبط با نرم‌افزار مورد نظر را تعیین می‌کند و سپس مشخص می‌سازد که آیا آن نرم‌افزار کاندیدای لایقی برای فرایند DSDM هست یا خیر.

مطالعه تجاری – خواسته‌های عملیاتی و اطلاعاتی را تعیین می‌کند که به برنامه‌ی کاربردی ارزش تجاری می‌دهند؛ همچنین معماری پایه را برای نرم‌افزار تعیین می‌کند و خواسته‌های مربوط به قابلیت نگهداری را برای نرم‌افزار مشخص می‌سازد.

تکرار مدل‌های عملیاتی – مجموعه‌ای از نمونه‌های اولیه‌ی افزایشی که عملکرد را برای مشتری به نمایش می‌گذارند. (توجه: همه‌ی نمونه‌های اولیه DSDM به‌منظور تکامل به نرم‌افزار قابل تحویل تهیه می‌شوند.) هدف از این چرخه‌ی تکرار، جمع‌آوری خواسته‌های اضافی یا توجه به بازخورد از کاربران در حین تمرین روی نمونه‌ی اولیه است.

تکرار طراحی و ساخت – بازبینی نمونه‌های اولیه ساخته شده طی مرحله‌ی تکرار مدل‌های عملیاتی برای حصول اطمینان از اینکه هر کدام به‌شیوه‌ای مهندسی شده است که بتواند برای کاربران نهایی ارزش تجاری به همراه داشته باشد. در برخی موارد، *تکرار مدل‌های عملیاتی* و *تکرار طراحی و ساخت* به‌صورت همزمان رخ می‌دهد.

پیاده‌سازی – قرار دادن آخرین نسخه‌ی نرم‌افزار (یک نمونه «عملیاتی شده») در محیط کاری است. لازم به ذکر است که (۱) این نسخه ممکن است ۱۰۰٪ کامل نباشد یا (۲) با استقرار این نسخه، هنوز هم تغییراتی درخواست شود. در هر حال، کار توسعه DSDM با برگشت به فعالیت تکرار مدل‌های عملیاتی ادامه می‌یابد.

نکته‌ی کلیدی

DSDM فرایندی چارچوبی است که می‌تواند تا کتک‌های سایر رویکردهای چابک نظیر XP را اقتباس کند.

DSDM را می‌توان با XP تلفیق کرد (بخش ۴-۳) و روشی ترکیبی به‌دست آورد که یک مدل فرایند محکم (چرخه حیات DSDM) تعریف شود، به‌طوری که حاوی اجزای لازم برای ساخت نسخه‌های نرم‌افزار از جنس XP باشد. به‌علاوه، مفاهیم همکاری و تیم‌های خودسازمان‌ده را می‌توان بر این مدل فرایند ترکیبی منطبق ساخت.

۴-۵-۳ کریستال

آلیستر کاکبرن [Coc05] و جیم های‌اسمیت [Hig02b]، مجموعه‌ای از روش‌های چابک را با عنوان

کریستال ابداع کرده‌اند.[1] هدف آنها دستیابی به روشی برای توسعه‌ی نرم‌افزار بوده است که اولین اولویت را به «قابلیت مانور» در طول پروژه بدهد؛ دوره‌ای که آنها از آن به‌عنوان *«بازی همکاری»* با محدودیت منابع برای ابداع و برقراری ارتباط، با هدف اولیه‌ی *تحویل نرم‌افزاری مفید و کاری* و هدف ثانویه‌ی *شروع بازی بعدی»* یاد می‌کنند [Coc02].

کاکبرن و های‌اسمیت برای دستیابی به این قابلیت مانور، مجموعه‌ای از روش شناسی‌ها را تعریف کرده‌اند که همگی در یک سری عناصر محوری مشترک بوده هر یک دارای نقش‌ها، الگوهای فرایند و عملکرد خاص است. این مجموعه‌ی کریستال در واقع مجموعه مثال‌هایی از فرایندهای چابک است که برای انواع پروژه‌های متفاوت موثر واقع شده‌اند. مقصود این است که تیم‌های چابک بتوانند عضوی از این مجموعه را انتخاب کنند که بیشترین مناسبت را با پروژه و محیط آنها داشته باشد.

۵-۵-۳ توسعه‌ی ویژگی-محور (FDD)

توسعه‌ی ویژگی-محور (FDD) در آغاز توسط پیتر کود و همکاران وی [Coa99] به‌عنوان یک مدل فرایند عملی برای مهندسی نرم‌افزار شیءگرا شکل گرفت. استفن پالمر و جان فلسینگ [Pal02] کارهای کود را بهبود و توسعه بخشیدند و فرایندی چابک و انطباق‌پذیر را توصیف کردند که برای پروژه‌هایی با ابعاد متوسط و بزرگ قابل استفاده است.

FDD همانند سایر روش‌های چابک، فلسفه‌ای را اقتباس کرده است که (۱) بر همکاری میان اعضای تیم FDD تأکید دارد؛ (۲) پیچیدگی مسأله و پروژه را با استفاده از تجزیه مبتنی بر ویژگی‌ها و سپس منسجم ساختن نسخه‌های نرم‌افزار مدیریت می‌کند، و (۳) ارتباط میان جزئیات فنی را با استفاده از ابزارهای لفظی، تصویری و متنی برقرار می‌سازد. FDD با تشویق راهبرد توسعه‌ی افزایشی، استفاده از وارسی طراحی و کدها، به‌کارگیری معیزی‌های تضمین کیفیت نرم‌افزار (فصل ۱۶)، جمع‌آوری معیارها، و به‌کارگیری الگوها (برای تحلیل، طراحی و ساخت)، بر فعالیت‌های تضمین کیفیت نرم‌افزار تأکید دارد.

در حیطه‌ی FDD، *ویژگی*[2] «یک عملکرد است که نزد متقاضی دارای ارزش بوده در کمتر از دو هفته قابل پیاده‌سازی است.» [Coa99]. تأکید بر تعریف «ویژگی‌ها» مزایای زیر را به همراه دارد:

- چون ویژگی‌ها قطعات کوچکی از قابلیت‌های قابل تحویل‌اند، کاربران راحت‌تر می‌توانند آنها را توصیف کنند؛ چگونگی ارتباط آنها با یکدیگر را بهتر درک کنند و بهتر آنها را از نظر ابهام، خطا یا موارد جا افتاده بازبینی کنند.
- ویژگی‌ها را می‌توان در قالب گروه‌های سلسله مراتبی و بر اساس ارتباط تجاری میان آنها سازمان‌دهی کرد.
- از آنجا که هر ویژگی یک نسخه‌ی قابل تحویل در FDD به‌شمار می‌رود، تیم باید عملکردها را هر دو هفته یک بار توسعه دهد.
- از آنجا که ویژگی‌ها کوچک هستند، وارسی موثر طراحی و کدهای آنها آسان‌تر است.

نکته‌ی کلیدی

کریستال به مجموعه‌ای از مدل‌های فرایند با «کد عمومی» مشترک، ولی روش‌های متفاوت برای تطبیق بر خصوصیات پروژه‌ی مورد نظر گفته می‌شود.

مرجع وب

گستره‌ی وسیعی از مقالات و قابل‌های پاور پوینت درباره FDD را می‌توانید در آدرس زیر پیدا کنید:
www.featuredrivendevelopment.com

[1] نام کریستال از خصوصیات کریستال‌های زمین‌شناختی گرفته شده است که هر یک دارای رنگ، شکل و سختی خاص خود است.

[2] feature

- سلسله مراتب ویژگی‌هاست که برنامه‌ریزی، زمان‌بندی و پیگیری پروژه را به پیش می‌برد نه مجموعه‌ای از وظایف مهندسی نرم‌افزار که به دلخواه تعیین شده باشد.

کود و همکاران [Coa99] قالب زیر را برای تعریف یک ویژگی پیشنهاد کرده‌اند (از راست به چپ بخوانید):

<action> the <result> <by|for|to> a(n) <object>

که در این قالب بندی، <شیء> یک «شخص، مکان یا هر یا هر چیز دیگری (از جمله نقش‌ها، لحظاتی از زمان یا بازه‌های زمانی، یا توصیفی شبه کاتالوگی)» می‌تواند باشد. مثال‌های از ویژگی‌های مربوط به یک نرم‌افزار تجارت الکترونیک در زیر داده شده است:

افزودن محصول به سبد خرید

نمایش مشخصات فنی محصول

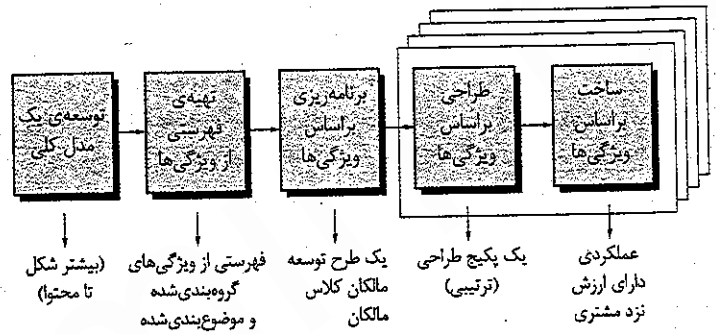
نگهداری اطلاعات حمل برای مشتری

در مجموعه ویژگی‌ها، ویژگی‌ها در قالب گروه‌هایی با ارتباط تجاری دسته‌بندی می‌شوند؛ مجموعه ویژگی‌ها به‌صورت زیر تعریف می‌شود [Coa99]:

<action><ing> a(n) <object>

برای مثال، به *فروش رساندن یک محصول*، مجموعه ویژگی‌هایی است که شامل ویژگی‌های ذکر شده در قبل و ویژگی‌های دیگر می‌شود.

در رویکرد FDD پنج فعالیت چارچوبی «مبتنی بر همکاری» [Coa99] تعریف می‌شود که در شکل ۵-۳ نشان داده شده‌اند (این فعالیت در FDD، فرایند نامیده می‌شوند).



شکل ۵-۳ توسعه‌ی ویژگی محور [Coa99].

در FDD بیش از هر روش چابک دیگر، بر تکنیک‌ها و دستورالعمل‌های مدیریت پروژه تأکید می‌شود. با رشد اندازه و پیچیدگی پروژه‌ها، مدیریت پروژه به‌شیوه‌ای موردی غالباً ناکافی به‌نظر می‌رسد. درک وضعیت پروژه-اینکه چه پیشرفت‌هایی انجام شده است و چه مشکلاتی به بار آمده است- برای سازندگان، مدیران و سایر ذی‌نفع‌ها ضروری است. اگر فشار مهلت زمانی چشمگیر باشد، تعیین اینکه آیا افزایش‌های نرم‌افزاری (ویژگی‌ها) به‌خوبی زمان‌بندی شده‌اند، اهمیت بسیار دارد. FDD برای این منظور، شش نقطه‌ی عطف در طول طراحی و پیاده‌سازی یک ویژگی تعیین کرده است: «بررسی در طراحی، طراحی، بازرسی طراحی، کدنویسی، بازرسی کد، ارتقا به ساخت» [Coa99].

۳-۵-۶ توسعه‌ی نرم‌افزار ناب (LSD)

در *توسعه‌ی نرم‌افزار ناب* (LSD) اصول تولید ناب (Lean Manufacturing) برای مهندسی نرم‌افزار اقتباس شده‌اند. اصول ناب که الهام بخش فرایند LSD بوده‌اند به‌صورت *حذف ضایعات*، *تعبیه کیفیت در داخل محصول*، *ایجاد دانش*، *احترام به تعهدات*، *تحویل سریع*، *احترام به افراد* و *بهینه‌سازی کلی* خلاصه می‌شوند.

هر کدام از این اصول را می‌توان بر فرایند نرم‌افزار منطبق ساخت. برای مثال، *حذف ضایعات* در حیطه‌ی یک پروژه‌ی نرم‌افزاری چابک به این صورت قابل تفسیر و تعبیر است [Das05]: (۱) اضافه نکردن ویژگی‌ها یا قابلیت‌های زائد، (۲) ارزیابی تاثیراتی که هر خواسته‌ی جدید بر هزینه و زمان‌بندی می‌گذارد، (۳) حذف هرگونه مراحل غیر ضروری از فرایند، (۴) وضع سازوکارهایی برای بهبود بخشیدن به‌شیوه‌ای که اعضای تیم اطلاعات را می‌یابند، (۵) حصول اطمینان از اینکه آزمون‌های انجام شده حداکثر خطای ممکن را برملا خواهند ساخت، (۶) کاهش دادن زمان لازم برای اتخاذ تصمیمی که بر نرم‌افزار یا فرایند به‌کار رفته در تولید آن تاثیر می‌گذارد و (۷) به جریان انداختن شیوه‌ای برای انتقال اطلاعات به همه‌ی طرف‌های ذی‌نفع فرایند.

برای بحث مشروحی درباره LSD و دستورات عملی جهت پیاده‌سازی این فرایند، می‌توانید [Pop06a] و [Pop06b] را ببینید.

۳-۵-۷ مدل‌سازی چابک (AM)

شرایط بسیاری وجود دارد که مهندسان نرم‌افزار باید سیستم‌های بزرگ و با اهمیت تجاری بالا بسازند. حوزه‌ی پیچیدگی این گونه سیستم‌ها باید طوری مدل‌سازی شود که (۱) همه‌ی طرف‌ها بتوانند دریابند که چه نیازهایی باید برآورده شوند، (۲) مسأله را بتوان به‌طور مؤثر در میان افرادی تقسیم کرد که قرار است آن را حل کنند و (۳) کیفیت را بتوان به موازات مهندسی و ساخته‌شدن سیستم، ارزیابی کرد.

طی سی سال گذشته، گستره‌ی وسیعی از روش‌های مدل‌سازی در مهندسی نرم‌افزار برای تحلیل و طراحی (چه به‌صورت سلسله مراتبی و چه در سطح مؤلفه ای) پیشنهاد شده است. این روش‌ها هر کدام مزایایی دارند، ولی ثابت شده است که به‌کارگیری آنها دشوار است و نمی‌توان آنها را بدون چالش روی پروژه‌های فراوان به‌کار برد. بخشی از مشکل به «وزن» این روش‌های مدل‌سازی بر می‌گردد. منظور از این گفته، حجم نمادگذاری لازم، درجه‌ی رسمیت پیشنهاد شده، حوزه‌ی گسترده‌ی این مدل‌ها برای پروژه‌های بزرگ و دشواری نگهداری مدل (ها) با رخ دادن تغییرات است. با این حال، مدل‌سازی تحلیل و طراحی باز هم مزایایی چشمگیر برای پروژه‌های بزرگ دارند- حتی اگر دلیل آن فقط این باشد که پروژه‌ها را از نظر فکری قابل مدیریت سازیم. آیا رویکردی چابک برای مدل‌سازی مهندسی نرم‌افزار وجود دارد که بتواند یک راهکار دیگر فراروی ما قرار دهد؟

اسکات امبلر [Amb02a] در «وب‌سایت رسمی مدل‌سازی چابک» مدل‌سازی چابک را به‌شیوه زیر توصیف می‌کند:

> مدل‌سازی چابک (AM) روشی مبتنی بر عمل برای مدل‌سازی و مستندسازی اثربخش سیستم‌های کامپیوتری است. به بیان ساده، مدل‌سازی چابک (AM) مجموعه‌ای از ارزش‌ها، اصول و اعمال مربوط به مدل‌سازی نرم‌افزار است که به‌شیوه‌ای مؤثر و سبک‌بال، روی پروژه‌های توسعه‌ی نرم‌افزار قابل اجراست. مدل‌های چابک از مدل‌های سنتی اثربخش‌ترند زیرا صرفاً خوب هستند و ضرورتی ندارد که کامل باشند.

مرجع وب

اطلاعات جامع درباره مدل‌سازی چابک از سایت زیر قابل دریافت است.
www.agilemodeling.com

مدل‌سازی چابک، کلیه‌ی ارزش‌های سازگار با بیانیه‌ی چابک را می‌پذیرد. در فلسفه‌ی مدل‌سازی چابک، این اصل به رسمیت شناخته می‌شود که تیم چابک باید جرأت تصمیم‌گیری برای رد یک طراحی یا بازآرایی را داشته باشد. این تیم همچنین باید از چنان فروتنی برخوردار باشد که بداند کارشناسان فن‌آوری، همه‌ی پاسخ‌ها را ندارند و کارشناسان تجاری و سایر طرف‌های ذی‌نفع را نیز باید محترم شمرد و با آغوش باز پذیرا بود.

گرچه در AM آرایه وسیعی از اصول مدل‌سازی «محوری» و «مکمل» پیشنهاد می‌شود، اصولی که AM را از سایر روش‌ها متمایز می‌سازند، عبارتند از [Amb02b]:

مدل‌سازی هدفمند. سازنده‌ای که از AM استفاده می‌کند باید پیش از ایجاد مدل، هدفی مشخص (مانند قراردادن اطلاعات در اختیار مشتری یا کمک به بهسازی شناخت جنبه‌ای از نرم‌افزار) در ذهن داشته باشد. هنگامی که هدف مدل تعیین شد، نوع نمادگذاری مورد استفاده و سطح جزئیات مورد نیاز آشکارتر خواهد شد.

استفاده از مدل‌های چندگانه. مدل‌ها و نمادگذاری‌های متفاوت بسیاری وجود دارند که می‌توان از آن‌ها در توصیف نرم‌افزار استفاده کرد. تنها زیرمجموعه‌ی کوچکی از آنها برای اکثر پروژه‌ها ضروری است. AM پیشنهاد می‌کند که برای به‌دست آوردن دید لازم، هر مدل باید جنبه‌ای متفاوت از سیستم را نشان دهد و تنها آن مدل‌هایی باید استفاده شوند که برای مخاطب هدف، ارزشی را ارائه می‌کنند.

سبک‌بار سفر کنید. با پیش‌رفتن کار مهندسی نرم‌افزار، فقط مدل‌هایی را حفظ کنید که ارزش‌های دراز مدت فراهم می‌سازند و بقیه را کنار بگذارید. هر محصولی که در حال کار است، درصورت نیاز به تغییرات، باید اصلاح شود و این امر باعث کندشدن سرعت تیم می‌گردد. آمبلر [Amb02a] می‌گوید: «هر بار که تصمیم می‌گیرید مدلی را حفظ کنید، از خیر چابکی می‌گذرید، به این امید که به‌راحتی به اطلاعات در دسترس تیم خود به‌شیوه‌ای انتزاعی دسترسی داشته باشید (و در نتیجه به‌طور بالقوه ارتباط خود با تیم و نیز طرف‌های ذی‌نفع پروژه را بهبود بخشید.)»

محتوا مهم‌تر از نمایش است. مدل‌سازی باید اطلاعاتی به مخاطب هدف ارائه دهد. مدلی که به‌لحاظ نحوی (syntax) کامل است، ولی اطلاعات ناچیزی به مخاطب می‌دهد، به اندازه‌ی مدلی که نقص نمادگذاری دارد، ولی محتوای با ارزشی در اختیار مخاطب خود قرار می‌دهد، ارزش ندارد.

شناخت مدل‌ها و ابزارهایی که در ایجاد آنها به‌کار می‌برید. نقاط قوت و ضعف هر مدل و ابزارهای به‌کار رفته در ایجاد آنها را بشناسید.

انطباق محلی. روش مدل‌سازی باید بر نیازهای تیم چابک انطباق یابد. بخش بزرگی از جامعه‌ی مهندسی نرم‌افزار، زبان مدل‌سازی یکپارچه (UML)[1] را به‌عنوان روش ارجح برای نشان دادن مدل‌های طراحی و تحلیل به رسمیت شناخته است. فرایند یکپارچه (فصل ۲) به‌منظور فراهم ساختن چارچوبی برای به‌کارگیری UML توسعه یافته است. اسکات آمبلر [Amb06] نسخه‌ای ساده از UP (فرایند یکپارچه) را توسعه داده است که فلسفه‌ی مدل‌سازی او را در بر دارد.

[1] خودآموز مختصری درباره UML در پیوست ۱ داده شده است.

> «روزی در داروخانه بودم و می‌خواستم قرص سرما-خوردگی بگیرم... آسان نبود. یک قفسه‌ی کامل از محصولات وجود دارد که به آنها نیاز داری. با خودت می‌گویی خوب است این یکی زود اثر می‌کند، ولی اثر آن یکی درازمدت است... کدام یک بیشتر اهمیت دارد؟ حال یا آینده؟»
>
> **جری ساینفلد**

اندرز

سبک‌بار سفر کردن، فلسفه مناسبی برای همه‌ی کارهای مهندسی نرم‌افزار است. فقط مدل‌هایی را بسازند که ارزش داشته باشند که بیشتر و به کمتر...

۳-۵-۸ فرایند یکپارچه‌ی چابک (AUP)

فرایند یکپارچه چابک (AUP) شامل یک فلسفه‌ی «ترتیب در مقیاس انبوه» و «مبتنی بر تکرار در مقیاس کوچک» برای ساخت سیستم‌های کامپیوتری می‌شود [Amb06]. AUP با اقتباس فعالیت‌های فازبندی‌شده‌ی کلاسیک UP–*دریافت*، *همکاری*، *ساخت* و *گذار*– ترتیبی خطی از فعالیت‌های مهندسی نرم‌افزار را فراهم می‌سازد که به تیم کمک می‌کند تا جریان کلی فرایند را برای یک پروژه‌ی نرم‌افزاری مجسم کند، ولی در داخل هر کدام از این فعالیت‌ها، تیم برای تحقق بخشیدن به چابکی و تحویل هرچه سریع‌ترنسخه‌های بامعنی از نرم‌افزار به کاربر نهایی باید به روش تکراری متوسل گردد. در هر دور تکرار AUP به فعالیت‌های زیر پرداخته می‌شود [Amb06]:

- *مدل‌سازی*. نمایش‌های UML از دامنه‌های تجاری و مسأله ایجاد می‌شود، ولی برای حفظ چابکی، این مدل‌ها «فقط باید به قدر کفایت خوب باشند» [Amb06] تا تیم بتواند به پیشروی ادامه دهد.
- *پیاده‌سازی*. مدل‌ها به کدهای منبع ترجمه می‌شوند.
- *آزمون*. همانند XP، تیم یک سری آزمون طراحی و اجرا می‌کند تا خطاها کشف شوند و اطمینان حاصل شود که کدهای نوشته شده خواسته‌های موجود را برآورده می‌سازند.
- *استقرار*. همانند فعالیت کلی بحث شده در فصل‌های ۱ و ۲، استقرار در اینجا نیز بر تحویل یک نسخه از نرم‌افزار و گرفتن بازخورد از کاربران نهایی توجه دارد.
- *مدیریت پیکربندی و پروژه*. در حیطه AUP، مدیریت پیکربندی (فصل ۲۲) به مدیریت تغییرات، مدیریت ریسک و کنترل هرگونه محصول پایداری می‌پردازد[1] که توسط تیم تولید می‌شود. مدیریت پروژه، پیشرفت تیم را پایش و کنترل می‌کند و هماهنگی فعالیت‌های تیم را بر عهده دارد.
- *مدیریت محیطی*. مدیریت محیطی وظیفه‌ی هماهنگی زیر ساخت فرایند را بر عهده دارد که شامل استانداردها، ابزارها و سایر فن‌آوری‌های پشتیبان مورد نیاز تیم می‌شود. گرچه AUP ارتباط‌های فنی و تاریخی با زبان مدل‌سازی یکپارچه (UML) دارد، باید توجه داشت که مدل‌سازی UML را می‌توان مرتبط با هر کدام از مدل‌های فرایند چابک دیگری که در بخش ۵-۳ شرح داده شد، به‌کار برد.

۳-۶ مجموعه‌ای از ابزارها برای فرایند چابک

برخی مدافعان فلسفه‌ی چابک چنین استدلال می‌کنند که ابزارهای خودکار نرم‌افزارسازی (مثلاً ابزارهای طراحی) را باید مکملی فرعی و کم اهمیت در فعالیت‌های تیم دانست که به هیچ وجه در موفقیت تیم اهمیت محوری ندارند، ولی آلیستر کاکبرن [Coc04] معتقد است که ابزارها می‌توانند دارای مزیت باشند و می‌گوید: «تیم‌های چابک بر به‌کارگیری ابزارهایی تأکید دارند که به درک و شناخت سریع کمک می‌کنند. برخی از این ابزارها اجتماعی‌اند و حتی در مرحله‌ی استخدام شروع می‌شوند. برخی جنبه‌ی فنی دارند و به تیم‌های توزیع شده کمک می‌کنند تا حضور فیزیکی را

[1] محصول کاری پایدار، یک مدل یا مستند یا مورد آزمون است که تیم تهیه می کند و برای مدت زمانی نامعین آن را نگهداری می کند. محصول کاری پایدار با تحویل یک نسخه از نرم‌افزار دور انداخته نمی‌شود.

ابزارهای نرم‌افزاری

توسعه‌ی چابک

هدف: هدف ابزارهای توسعه‌ی چابک، کمک به یک یا چند جنبه از توسعه‌ی چابک با تأکید بر تسهیل و سرعت بخشیدن به تولید یک نرم‌افزار عملیاتی است. از این ابزارها می‌توان در هنگام به‌کارگیری مدل‌های فرایند تجویزی استفاده کرد.

مکانیک: این ابزارها مکانیک متفاوتی دارند. به‌طور کلی، مجموعه ابزارهای چابک، شامل پشتیبانی خودکار برای برنامه‌ریزی پروژه، توسعه‌ی use case و جمع‌آوری خواسته‌ها، طراحی سریع، کدنویسی و آزموده می‌شوند.

ابزارهای نمونه

توجه: از آنجا که توسعه‌ی چابک مبحثی داغ به‌شمار می‌رود، اکثر فروشندگان ابزارهای نرم‌افزاری ادعا می‌کنند ابزارهایی می‌فروشند که روش چابک را پشتیبانی می‌کنند.

ابزارهایی که در اینجا ذکر می‌شوند، دارای این خصوصیت هستند که به‌طور اخص برای پروژه‌های چابک مناسبند.

OnTime، محصول Axosoft (**www.axosoft.com**) که مدیریت فرایند چابک را برای انواع فعالیت‌های فنی در فرایند پشتیبانی می‌کند.

Ideogramic UML، محصول Ideogramic (**www.ideogramic.com**) که یک مجموعه ابزار UML است و مشخصاً برای استفاده در فرایندهای چابک تهیه شده است.

Together Tool Set، که توسط Borland (**www.borland.com**) توزیع شده است، مجموعه ابزارهایی را فراهم می‌سازد که فعالیت‌های فنی بسیاری را در فرایندهای XP و سایر فرایندهای چابک، پشتیبانی می‌کنند.

شیه‌سازی کنند. بسیاری از این ابزارها فیزیکی بوده به افراد، امکان‌کار در قالب کارگاهی را می‌دهند.»

از آنجا که دسترسی به افراد مناسب (استخدام)، همکاری تیمی، برقراری ارتباط میان طرف‌های ذی‌نفع و مدیریت غیرمستقیم، عناصر کلیدی در کلیه مدل‌های فرایند چابک به‌شمار می‌روند، کاکبرن چنین استدلال می‌کند که «ابزارهای» مرتبط با این امور، عوامل مهمی در موفقیت برای چابکی به‌شمار می‌روند. برای مثال، ممکن است برای اینکه عضو آینده‌ی تیم را وادار به چند ساعت برنامه‌نویسی جفتی با یکی از اعضای فعلی تیم سازیم، ممکن است به یک «ابزار» استخدامی نیاز باشد. به این ترتیب فرد مناسب را می‌توان بلافاصله ارزیابی کرد.

«ابزارهای» همکاری و ارتباطی عموماً از فن‌آوری چندان بالایی برخوردار نیستند و می‌توانند شامل هر سازوکاری («مجاورت فیزیکی، تخته سفید، برگه‌های پوستر، کارت یادداشت و کاغذهای یادداشت» [Coc04]) باشد که اطلاعات را در میان اعضای تیم چابک قرار دهد و ارتباط را برقرار سازد. برقراری ارتباط فعال از طریق پویایی تیم (مثلاً برنامه‌نویسی جفتی) قابل انجام است، در حالی که برقراری ارتباط انفعالی از طریق «نشر دهنده‌های اطلاعات» (نظیر یک صفحه نمایش تخت که وضعیت کلی مؤلفه‌های متفاوت یک نسخه‌ی نرم‌افزار را نشان می‌دهد) صورت می‌گیرد. ابزارهای مدیریت پروژه‌ی دیگر بر نمودار گانت تأکید ندارند و نه«مودارهای ارزش کسبی»[1] را جایگزین آن

[1] Earned Value Charts

می‌کنند؛ «نمودارهایی از آزمون‌های ایجاد شده در مقایسه با قبولی آزمون... سایر ابزارهای چابک در بهینه‌سازی محیطی به‌کار می‌روند که تیم چابک در آن کار می‌کند (مثلاً مکان‌های مناسب تری برای برگزاری جلسات)، بهبود بخشیدن به فرهنگ تیمی با بذل توجه به تعامل‌های اجتماعی (مثلاً تیم‌های متحد)، دستگاه‌های فیزیکی (تخته سفیدهای الکترونیکی) و بهبود فرایند (مثلاً برنامه‌نویسی جفتی یا تعیین پنجره‌های زمانی).» [Coc04].

آیا هیچ کدام از اینها که گفته شد، واقعاً ابزار به‌شمار می‌روند؟ آری، اگر وظایف محول شده به یک عضو تیم چابک را تسهیل کنند و کیفیت محصول نهایی را بهبود بخشند.

۳-۷ خلاصه

در اقتصاد مدرن، شرایط بازار به سرعت تغییر می‌کند، نیازهای مشتری و کاربر نهایی تکامل می‌یابد و تهدیدهای رقابتی جدید بی هیچ هشداری مطرح می‌شود. دست‌اندرکاران باید چنان رویکردی به مهندسی نرم‌افزار داشته باشند که بتوانند به کمک آن چابک باقی بمانند- و فرایندهایی با قابلیت مانور، انطباق‌پذیر و ناب تعریف کنند که قادر به پاسخ گویی نیازهای تجاری مدرن باشند.

در فلسفه‌ی چابکی برای مهندسی نرم‌افزار، چهار مسأله کلیدی مورد تأکید است: اهمیت خودسازمان‌دهی تیم‌هایی که بر کارکرد خود کنترل دارند، برقراری ارتباط و همکاری میان اعضای تیم و میان دست‌اندرکاران و مشتریان آنها، اعتقاد به این که تغییرات با خود فرصت‌ها را به همراه دارند و تأکید بر تحویل سریع نرم‌افزاری که رضایت مشتری را برآورده سازد. برای هر کدام از این مسائل، مدل‌های فرایند چابک طراحی شده است.

برنامه‌نویسی حدی (XP) پرکاربردترین فرایند چابک است. XP که در قالب چهار فعالیت چارچوبی سازمان‌دهی می‌شود- برنامه‌ریزی، طراحی، کدنویسی و آزمون- چند تکنیک نوآورانه و پر قدرت پیشنهاد می‌کند که به تیم چابک این امکان را می‌دهد نسخه‌های پیاپی از نرم‌افزار ایجاد کند که ویژگی‌ها و قابلیت‌های توصیف شده و اولویت‌بندی شده توسط طرف‌های ذی‌نفع را تحویل دهند. سایر مدل‌های فرایند چابک نیز بر همکاری انسانی و خودسازمان‌دهی تیمی تأکید دارند، ولی فعالیت‌های چارچوبی خاص خود را تعریف می‌کنند و بر نقاط متفاوتی تأکید می‌کنند. برای مثال، ASD از یک فرایند مبتنی بر تکرار استفاده می‌کند که شامل برنامه‌ریزی چرخه‌ای وقفی، روش‌های نسبتاً پر کار برای جمع‌آوری داده‌ها و یک چرخه‌ی توسعه‌ی مبتنی بر تکرار می‌شود که «گروه‌های نظارت مشتری» (customer focus groups) و بازبینی‌های فنی رسمی را به‌عنوان سازوکارهای بازخورد بی‌درنگ در بر می‌گیرد. اسکرام بر به‌کارگیری مجموعه‌ای از الگوهای فرایند نرم‌افزار تأکید دارد که برای پروژه‌هایی با مهلت‌های زمانی محدود، خواسته‌های در حال تغییر و اهمیت تجاری بالا مفید واقع شده‌اند. هر الگوی فرایندی، مجموعه‌ای از وظایف توسعه را تعریف می‌کند و به تیم اسکرام این امکان را می‌دهد که فرایندی منطبق بر نیازهای پروژه ایجاد کند. روش توسعه سیستم‌های پویا (DSDM) به استفاده از زمان‌بندی در چارچوب پنجره‌های زمانی روی می‌آورد و پیشنهاد می‌کند که برای هر نسخه‌ی نرم‌افزار فقط باید آن مقدار که لازم است کار انجام شود تا حرکت به سوی نسخه‌ی بعدی تسهیل گردد. کریستال به خانواده‌ای از مدل‌های فرایند چابک گفته می‌شود که بر خصوصیات ویژه‌ی یک پروژه قابل انطباق باشند.

توسعه‌ی ویژگی محور (FDD)، قدری رسمی‌تر از سایر روش‌های چابک است، ولی همچنان با جلب توجه تیم پروژه به توسعه‌ی یک سری ویژگی ها- قابلیت‌هایی که نزد متقاضی ارزش دارند و در کمتر از دو هفته قابل پیاده‌سازی هستند- چابکی را حفظ می‌کند. توسعه‌ی نرم‌افزار ناب (LSD) اصول تولید ناب را وارد دنیای مهندسی نرم‌افزار کرده است. مدل‌سازی چابک (AM) پیشنهاد می‌کند که مدل‌سازی برای همه‌ی سیستم‌ها ضروری است، ولی پیچیدگی، نوع، و اندازه مدل باید متناسب با نرم‌افزاری که قرار است ساخته شود، تنظیم گردد. فرایند چابک یکپارچه (AUP) فلسفه‌ی «ترتیب در مقیاس انبوه» و «تکرار در مقیاس کوچک» را بر ساخت نرم‌افزار مطرح می‌سازد.

مسائل و نکاتی برای تعمق

۳-۱ «بیانیه توسعه‌ی نرم‌افزار چابک» را که در ابتدای فصل آورده شد، دوباره بخوانید. آیا می‌توانید به شرایطی فکر کنید که در آن یک یا چند مورد از ارزش‌های ذکر شده تیم نرم‌افزاری را به دردسر دچار کند؟

۳-۲ چابکی را (برای پروژه‌های نرم‌افزاری) به زبان ساده شرح دهید.

۳-۳ چرا یک فرایند مبتنی بر تکرار، مدیریت تغییر را آسان‌تر می‌سازد؟ آیا همه‌ی فرایندهای چابکی که در این فصل بحث شدند، مبتنی بر تکرارند؟ آیا می‌توان پروژه را تنها یک دور تکرار کامل کرد و باز هم چابک بود؟ پاسخ‌های خود را توضیح دهید.

۳-۴ آیا هر کدام از فرایندهای چابک را می‌توان با استفاده از فعالیت‌های چارچوبی کلی ذکر شده در فصل ۲ توضیح داد؟ جدولی تهیه کنید و فعالیت‌های کلی را به فعالیت‌های تعریف شده برای هر کدام از فرایندهای چابک ربط دهید.

۳-۵ سعی کنید به «یک اصل چابکی» دیگر برسید که باز هم قابلیت مانور بیشتری به تیم نرم‌افزاری بدهد.

۳-۶ یکی از اصول چابکی ذکر شده در بخش ۳-۱ را برگزینید و بکوشید تعیین کنید که آیا هر کدام از مدل‌های فرایند ارائه شده در این فصل، آن اصل را در بردارد یا خیر. [توجه: در این فصل تنها نگاهی اجمالی از این مدل‌های فرایند ارائه شده است، لذا تعیین اینکه آیا اصلی در یک یا چند مدل در بر گرفته شده است، ممکن نخواهد بود مگر اینکه درباره آنها به تحقیق و پژوهش بپردازید (که برای این مسأله لازم نیست).]

۳-۷ چرا خواسته‌ها این قدر سریع تغییر می‌کنند؟ واقعا مردم نمی‌دانند که چه می‌خواهند؟

۳-۸ اکثر مدل‌های فرایند چابک، ارتباط رو در رو را توصیه می‌کنند. با این حال، امروزه اعضای تیم نرم‌افزاری و مشتریان آنها ممکن است از نظر جغرافیایی با هم فاصله داشته باشند. آیا تصور می‌کنید که به این ترتیب باید از فاصله‌های جغرافیایی پرهیز کرد؟ آیا می‌توانید به راه‌هایی برای غلبه بر این مشکل فکر کنید؟

۳-۹ یک داستان کاربری XP بنویسید که توصیفی باشد از ویژگی «مکان‌های مطلوب» یا همان «چوب‌آلف» (Bookmark) که در مرورگرهای وب در دسترس است.

۳-۱۰ راهکار خیزشی در XP چیست؟

۳-۱۱ مفاهیم بازآرایی‌کردن و برنامه‌نویسی جفتی را به زبان ساده شرح دهید.

۳-۱۲ قدری مطالعه کنید و توضیح دهید که کادر زمانی چیست. این مفهوم چگونه به تیم ASD کمک می‌کند تا نسخه‌های نرم‌افزاری را در دوره‌های زمانی کوتاه تحویل دهد؟

۳-۱۳ آیا قاعده ۸۰٪ در DSDM و روش کادر زمانی تعریف شده برای ASD نتیجه‌های یکسان در بر دارند؟

۳-۱۴ با استفاده از الگوهای فرایند ارائه شده در فصل ۲، یک الگوی فرایند برای یکی از الگوهای اسکرام (بخش ۳-۵-۲) ارائه کنید.

۳-۱۵ چرا به گروهی از روش‌های چابک نام کریستال داده شده است؟

۱۶-۳ با استفاده از قالب ویژگی‌ها در FDD که در بخش ۵-۵-۳ شرح داده شد، یک مجموعه ویژگی برای مرورگرهای وب بنویسید. اکنون برای این مجموعه ویژگی‌ها، مجموعه‌ای از ویژگی را بنویسید.

۱۷-۳ از وب‌سایت رسمی مدل‌سازی چابک بازدید کنید و فهرست کاملی از اصول محوری و مکمل AM تهیه کنید.

۱۸-۳ مجموعه ابزارهای پیشنهاد شده در بخش ۶-۳ بسیاری از جنبه‌های «نرم» روش‌های چابک را پشتیبانی می‌کنند. چون برقراری ارتباط بسیار مهم است، یک مجموعه ابزار واقعی توصیه کنید که بتوان در بهبود بخشیدن به ارتباط میان طرف‌های ذی‌نفع یک تیم چابک از آن بهره برد.

بخش دوم

مدل‌سازی

در این بخش از کتاب، مطالبی درباره اصول، مفاهیم و تکنیک‌های به‌کاررفته در ایجاد مدل‌های تحلیل (مدل‌های خواسته‌ها) با کیفیتی بالا خواهید آموخت.

در فصل‌های آینده به این پرسش‌ها خواهیم پرداخت:

- چه مفاهیم و اصولی راهنمای کار مهندسی هستند؟
- مهندسی خواسته‌ها چیست و چه مفاهیمی، بسترساز تحلیل خوب و مناسب خواسته‌ها هستند؟
- مدل خواسته‌ها چگونه ایجاد می‌شود و عناصر آن کدام‌اند؟
- عناصر یک طراحی خوب کدام‌اند؟
- طراحی معماری چگونه چارچوبی برای سایر کنش‌های طراحی فراهم می‌سازد و از چه مدل‌هایی در آن استفاده می‌شود؟
- مؤلفه‌های نرم‌افزاری با کیفیت بالا را چگونه طراحی می‌کنیم؟
- از کدام مفاهیم، مدل‌ها و روش‌ها در طراحی واسط کاربری می‌توان استفاده نمود؟
- الگوی مبتنی بر طراحی چیست؟
- در طراحی برنامه‌های تحت وب از چه راهبردها و روش‌هایی استفاده می‌شود؟

هنگامی که به این پرسش‌ها پاسخ گفته شد، بهتر می‌توانید آماده‌ی به‌کارگیری کار مهندسی نرم‌افزار شوید.

فصل ۴

اصول راهنما در مهندسی نرم‌افزار

نگاهی گذرا

اصول راهنما چیستند؟ مهندسی نرم‌افزار، آرایه وسیعی از اصول، مفاهیم، روش‌ها و ابزارهاست که باید در برنامه‌ریزی برای توسعه یک نرم‌افزار در نظر گرفت. اصول راهنمای این کار، بستری فراهم می‌سازد که مهندسی نرم‌افزار را می‌توان بر آن استوار ساخت.

چه کسی این کار را انجام می‌دهد؟ نرم‌افزارنویسان (مهندسان نرم‌افزار) و مدیران آنها انواع وظایف مهندسی نرم‌افزار را بر عهده دارند.

چرا اهمیت دارد؟ فرایند نرم‌افزار برای رسیدن به هدفی موفق، یک نقشه راه در اختیار همه‌ی افراد دخیل در ایجاد یک سیستم یا محصول کامپیوتری قرار می‌دهد. کار مهندسی، جزئیات لازم برای طی این مسیر را برای شما فراهم می‌سازد. به شما می‌گوید کجا پل هست، راه در چه نقاطی بسته است و کجا با دو راهی مواجه می‌شوید. به شما کمک می‌کند تا مفاهیم و اصولی را که باید درک و رعایت شوند تا با سرعت و با اطمینان به پیش بروید، بهتر بشناسید. شیوه‌ی پیش رفتن را به شما می‌آموزد و مشخص می‌کند که کجا باید از سرعت خود بکاهید و کجا باید سرعت بگیرید. در حیطه‌ی مهندسی نرم‌افزار، کار مهندسی چیزی است که در طول روز انجام می‌دهید تا نرم‌افزار را از یک ایده به واقعیت برسانید.

مراحل کار کدام است؟ سه عنصر کار مهندسی در همه‌ی انواع مدل‌های فرایند، کاربرد دارند. عنصر چهار-یعنی ابزارهای مورد استفاده-بسته به مدل به‌کار رفته متفاوت است.

محصول کار چیست؟ کار مهندسی شامل فعالیت‌های فنی می‌شود که همه‌ی محصولات کاری تعریف شده توسط مدل فرایند نرم‌افزاری انتخاب شده را تولید می‌کند.

چطور اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ نخست، از اصول کاری که هر لحظه در حال انجام آن هستید (مثلاً طراحی) درکی درست داشته باشید. سپس یقین حاصل کنید که روشی مناسب برای کار انتخاب کرده‌اید، حتماً چگونگی به‌کارگیری روش را درک کنید، از ابزارهای خودکار در صورت مناسب بودن برای وظیفه‌ی مورد نظر بهره ببرید و درباره نیاز به استفاده از تکنیک‌ها عزمی راسخ داشته باشید تا از کیفیت محصولات کاری تولید شده اطمینان پیدا کنید.

الن اولمان [Ull97] در کتابی که زندگی و افکار مهندسان نرم‌افزار را مورد کندوکاو قرار می‌دهد، بخشی از زندگی آنها را چنین به تصویر می‌کشد و افکار آنها را تحت فشار کاری نمایان می‌سازد:

> نمی‌دانم ساعت چند است. این دفتر هیچ پنجره و هیچ ساعتی ندارد؛ فقط LED قرمز یک اجاق مایکروفر که چشمک می‌زند و مدام ساعت دوازده را نشان می‌دهد. من و جوئل چند روز است که مشغول برنامه‌نویسی بوده‌ایم. یک اشکال از نوع ناجورش داریم. این LED قرمز انگار دارد فعالیت مغز ما را نشان می‌دهد، چون یک جورهایی آهنگ آن با آهنگ چشمک زدن این LED یکی است...
>
> ما روی چه چیزی داریم کار می‌کنیم؟... الان جزئیات از دستم در رفته است. شاید داریم به آدم‌های ضعیف کمک می‌کنیم یا شاید هم یک سری روال‌های سطح پایین تنظیم می‌کنیم تا بیت‌های روی یک پروتکل بانک اطلاعاتی توزیع شده را وارسی کنیم- برایم اهمیتی ندارد. باید به بخش دیگری از وجودم اهمیت بدهم- بعداً وقتی که از این اتاق مملو از کامپیوتر بیرون رفتم- باید ببینم چرا، برای چه کسی و به چه هدفی دارم نرم‌افزار می‌نویسم، ولی فعلاً خیر. من از پرده‌ای گذشته‌ام که در آن، جهان واقعی و کابردهایش دیگر اهمیتی ندارند. من مهندس نرم‌افزارم...

قطعاً این تصویر روشنی از کار مهندسی نرم‌افزار نیست، ولی پس از تعمق، بسیاری از خوانندگان این کتاب با آن ارتباط برقرار خواهند کرد.

کسانی که نرم‌افزار کامپیوتری می‌سازند، هنر، حرفه یا رشته‌ای[1] را تجربه می‌کنند که به آن مهندسی نرم‌افزار گفته می‌شود، ولی «کار مهندسی نرم‌افزار چیست؟» از یک دیدگاه کلی، کار مهندسی به مجموعه‌ای مفاهیم، اصول، روش‌ها و ابزارها گفته می‌شود که یک مهندس نرم‌افزار در طول روز با آنها سروکار دارد. کار مهندسی به مدیران این امکان را می‌دهد که پروژه‌های نرم‌افزاری را مدیریت کنند و به مهندسان نرم‌افزار این امکان را می‌دهد که برنامه‌های کامپیوتری بسازند. کار مهندسی، یک مدل فرایند نرم‌افزاری را از دستورالعمل‌های فنی و مدیریت‌های لازم پر می‌کند تا پروژه به انجام برسد. با کار مهندسی، یک روش بی‌حساب و کتاب، به رویکردی سازمان یافته و اثربخش‌تر تبدیل می‌شود که احتمال موفقیت آن بیشتر است.

جنبه‌های گوناگون کار مهندسی نرم‌افزار را در سرتاسر این کتاب مورد بررسی قرار خواهیم داد. در این فصل، به اصول و مفاهیمی خواهیم پرداخت که به‌طور کلی راهنمای کار مهندسی نرم‌افزار هستند.

۴-۱ دانش مهندسی نرم‌افزار

استیو مک کانل در سر مقاله IEEE Software یک دهه قبل توضیح زیر را ارائه داد [Mcc99]:

> بسیاری از نرم‌افزارنویسان، دانش مهندسی نرم‌افزار را تقریباً به‌طور انحصاری، اطلاع داشتن از فن‌آوری‌های خاص می‌دانند: جاوا، پرل، html، C++، Linux، Windows NT و غیره. اطلاع داشتن از جزئیات فن‌آوری برای انجام برنامه‌نویسی کامپیوتری لازم است. اگر کسی از شما بخواهد برنامه‌ای به زبان C++ بنویسید، باید چیزهایی از C++ بدانید تا برنامه‌تان نتیجه‌بخش باشد.

[1] برخی نویسندگان سعی می‌کنند یکی از این واژه‌ها را به‌کار ببرند و دو واژه دیگر را به کار نبرند. ولی واقعیت این است که مهندسی نرم‌افزار هر سه اینهاست.

> زیاد می‌شنوید که نیمه عمر دانش توسعه‌ی نرم‌افزار، سه سال است: نیمی از آنچه که امروز می‌دانید، سه سال بعد دیگر به کارتان نخواهد آمد. در دامنه دانش‌های مرتبط با فن‌آوری، این احتمالاً درست است، ولی یک نوع دیگر دانش در توسعه‌ی نرم‌افزار وجود دارد- نوعی که ما آن را به‌عنوان «اصول مهندسی نرم‌افزار» در نظر می‌گیریم- که نیمه عمر آن سه سال نیست. این اصول مهندسی نرم‌افزار احتمالاً در سرتاسر زندگی کاری یک برنامه نویس حرفه‌ای به او کمک می‌کنند.

مک کانل در ادامه‌ی بحث خود چنین استدلال می‌کند که توده‌ی دانش مهندسی نرم‌افزار (تقریباً در سال ۲۰۰۰) به یک «هسته‌ی پایدار» متکامل شده است که براساس برآورد او نشان‌گر حدوداً ۷۵٪ از دانش مورد نیاز برای توسعه یک سیستم پیچیده است، ولی این هسته‌ی پایدار حاوی چیست؟

چنان که مک کانل خاطرنشان می‌سازد، اصول هسته‌ای- ایده‌های پایه‌ای که مهندسان نرم‌افزار را در انجام کارهایشان، راهنمایی می‌کنند-اکنون بستری فراهم می‌سازند که مدل‌های نرم‌افزاری، روش‌ها و ابزارها را در آن بستر می‌توان به‌کار برد و ارزیابی کرد.

۴-۲ اصول هسته‌ای

مجموعه‌ای از اصول هسته‌ای وجود دارد که راهنمای مهندسی نرم‌افزار است و به استفاده از یک فرایند نرم‌افزار با معنی و اجرای اثربخش روش‌های مهندسی نرم‌افزار کمک می‌کند. در سطح فرایند، اصول فرایند، یک بنیاد فلسفی ایجاد می‌کنند که تیم نرم‌افزاری را به هنگام اجرای فعالیت‌های چارچوبی و چتری هدایت می‌کند، جریان فرایند را مورد کاوش قرار می‌دهد و مجموعه‌ای از محصولات کاری مهندسی نرم‌افزار تولید می‌کند. در سطح کاری، اصول مهندسی نرم‌افزار، مجموعه‌ای از ارزش‌ها و قواعد را تعیین می‌کند که شما را در تحلیل یک مسأله، طراحی یک راهکار، پیاده‌سازی و آزمون آن راهکار و سرانجام استقرار نرم‌افزار در جامعه‌ی کاربری راهنمایی می‌کند.

در فصل ۱، مجموعه‌ای از اصول کلی را مشخص کردیم که شامل فرایند و کار مهندسی نرم‌افزار می‌شوند: (۱) فراهم ساختن ارزش برای کاربر نهایی، (۲) حفظ سادگی، (۳) حفظ چشم انداز (محصول و پروژه)، (۴) دانستن این مطلب که دیگران آنچه را که شما ساخته‌اید مصرف می‌کنند (و باید آن را درک کنند)، (۵) نگاه به آینده، (۶) برنامه‌ریزی برای استفاده‌ی مجدد، و (۷) تفکر! گرچه این اصول کلی اهمیت دارند، در چنان سطح بالایی از انتزاع بیان می‌شوند که گاهی ترجمه‌ی آنها به کارهای روزمره در مهندسی نرم‌افزار دشوار است. در بخش‌هایی که به‌دنبال خواهد آمد، اصول هسته‌ای را که راهنمای کار مهندسی و فرایند مهندسی خواهند بود، با جزئیات بیشتر بحث خواهیم کرد.

۴-۲-۱ اصول راهنمای فرآیند مهندسی

در بخش اول این کتاب درباره اهمیت فرایند نرم‌افزاری سخن گفته شد و مدل‌های فراوان و متفاوتِ پیشنهادشده برای مهندسی نرم‌افزار شرح داده شدند. مدل انتخاب شده خطی باشد یا مبتنی بر تکرار، تجویزی باشد یا چابک، تفاوتی ندارد و می‌توان آن را با استفاده از یک چارچوب کلی مشخص کرد که برای همه‌ی مدل‌های فرایند قابل استفاده است. مجموعه اصول هسته‌ای زیر را می‌توان برای چارچوب، و با بسط دادن آن برای هر فرایند نرم‌افزار به‌کار گرفت.

> «به لحاظ نظری هیچ تفاوتی میان نظریه و عمل نیست، ولی در عمل، هست.»
>
> یان وان اسنیپشوت

اصل ۱. چابک باشید. مدل فرایند انتخابی شما تجویزی باشد یا چابک، اصول فلسفی توسعه‌ی چابک باید بر رویکردتان حاکم باشد. تمامی جنبه‌های کاری که انجام می‌دهید، باید بر اقتصاد کنش تاکید داشته باشد- در رویکرد فنی خود تا حد امکان سادگی را حفظ کنید، در محصولات کاری‌ای که تولید می‌کنید تا حد امکان ایجاز را رعایت کنید و هر گاه که امکان داشته باشد، تصمیم‌گیری‌ها را محلی کنید.

اصل ۲. در هر مرحله، کیفیت را در کانون توجه قرار دهید. شرط خروج از هر فعالیت، کنش و وظیفه‌ی فرایند باید توجه به کیفیت محصول کاری تولید شده باشد.

اصل ۳. آمادگی انطباق را داشته باشید. فرایند، چیزی نیست که تعصب در آن راه داشته باشد. در صورت نیاز، رویکرد خود را بر محدودیت‌های اعمال شده از طرف مسأله، آدم‌ها و خود پروژه وفق دهید.

اصل ۴. تیمی اثربخش تشکیل دهید. فرایند و کار مهندسی نرم‌افزار اهمیت دارند، ولی سنگ بنای اصلی پروژه را آدم‌های آن تشکیل می‌دهد. یک تیم خودسازمان‌ده تشکیل دهید که اعضای آن از احترام و اطمینان متقابل برخوردار باشد.

اصل ۵. سازوکارهایی برای برقراری ارتباط و هماهنگی ایجاد کنید. اگر اطلاعات مهم در کانون توجه قرار نگیرند و/یا طرف‌های ذی‌نفع از هماهنگ ساختن تلاش‌های خود برای ایجاد یک محصول نهایی موفق، عاجز باشند، پروژه‌ها به شکست می‌انجامند. این‌ها مسائلی مدیریتی‌اند که باید به آنها پرداخته شود.

اصل ۶. مدیریت تغییرات. رویکرد مورد استفاده ممکن است رسمی یا غیررسمی باشد، ولی برای مدیریت شیوه‌ی درخواست، ارزیابی، تصویب و پیاده‌سازی تغییرات باید سازوکارهایی وضع شود.

اصل ۷. ارزیابی ریسک. به موازات توسعه‌ی نرم‌افزار، اشتباهات بسیاری ممکن است رخ دهد. ارائه طرح‌های آینده‌نگر ضروری است.

اصل ۸. ایجاد محصولات کاری که برای دیگران ارزش فراهم می‌کنند. فقط آن دسته از محصولات کاری را ایجاد کنید که برای سایر فعالیت‌ها، کنش‌ها و وظایف فرایند، ارزشی به ارمغان می‌آورند. هر محصول کاری که به‌عنوان بخشی از کار مهندسی نرم‌افزار تولید می‌شود، به دیگری سپرده می‌شود. فهرستی از عملکردها و ویژگی‌های مورد نیاز به شخصی (اشخاصی) داده می‌شود که یک طراحی را توسعه می‌دهند، این طراحی به آنهایی سپرده می‌شود که کدها را می‌نویسند و به همین ترتیب... اطمینان حاصل کنید که هر محصول کاری، اطلاعات لازم را بدون ابهام یا جاافتادگی ارائه دهد.

بخش چهارم این کتاب به مسائل پروژه و مدیریت فرایند اختصاص یافته است و به تفصیل به جنبه‌های گوناگون هر کدام از این اصول خواهد پرداخت.

اندرز

هر پروژه و هر تیمی منحصر به فرد است. این بدان معناست که باید به‌طریقی فرایند خود را به بهترین وجه بر نیازهای خود منطبق سازید.

«حقیقت مطلب این است که شما همیشه می‌دانید کار درست کدام است. بخش دشوار کار، انجام آن است.»

ژنرال اچ. نورمن شوارتزکف

۲-۲-۴ اصول راهنمای کار مهندسی نرم‌افزار

کار مهندسی نرم‌افزار یک هدف غالب دارد- تحویل به‌موقع و با کیفیت بالای نرم‌افزاری عملیاتی که حاوی ویژگی‌ها و قابلیت‌های لازم برای برآورده ساختن نیازهای همه‌ی طرف‌های ذی‌نفع باشد. برای دستیابی به این هدف، باید مجموعه‌ای از اصول را پذیرا باشید که راهنمای فنی شما شوند. هر

روش تحلیل و طراحی‌ای که اعمال کنید، از هر تکنیک ساختی که استفاده کنید (مثلاً زبان‌های برنامه‌نویسی یا ابزارهای خودکار)، یا هر رویکرد اعتبارسنجی و وارسی را که انتخاب کنید، این اصول مزایای خاص خود را خواهند داشت. مجموعه اصول هسته‌ای زیر اساس کار مهندسی نرم‌افزار را تعیین می‌کنند.

نکته‌ی کلیدی

مسائل را اگر به دغدغه‌های جداگانه‌ای تقسیم کنید که هر یک به‌طور جداگانه قابل حل و اعتبارسنجی باشند، بهتر می‌توان آنها را حل کرد.

اصل ۱. تقسیم و حل. به بیان فنی‌تر، در تحلیل و طراحی باید همواره بر جداسازی دغدغه‌ها تاکید داشت. یک مسأله بزرگ را اگر به مجموعه‌ای از عناصر (یا دغدغه‌ها) تقسیم کنیم، حل آن راحت‌تر می‌شود. به‌طور ایده‌آل، هر دغدغه‌ای یک قابلیت متمایز عرضه می‌کند که قابل توسعه بوده در برخی موارد می‌توان آن را مستقل از سایر دغدغه‌ها اعتبارسنجی کرد.

اصل ۲. درک به‌کارگیری انتزاع‌ها. انتزاع، شکل ساده‌ی عنصر پیچیده‌ای از سیستم به‌کاررفته در انتقال معنا در یک عبارت منفرد است. هنگامی که از انتزاع *صفحه‌گسترده* استفاده می‌کنیم، فرض می‌کنیم که مخاطب می‌داند صفحه گسترده چیست، ساختار کلی محتویات ارائه شده توسط یک صفحه گسترده را می‌شناسد و می‌داند چه عملیاتی روی آن قابل اجراست. در کار مهندسی نرم‌افزار، از چندین سطح انتزاع استفاده خواهید کرد که هر کدام معنایی دارد که باید انتقال داده شود. در کار تحلیل و طراحی، تیم نرم‌افزاری معمولاً با مدل‌هایی شروع می‌کند که نشان‌گر سطوح بالایی از انتزاع هستند (مانند یک صفحه گسترده) و به آهستگی این مدل‌ها را به سطوح پایین‌تری از انتزاع (مثلاً یک ستون یا تابع SUM) پالایش می‌کنند.

جوئل اسپولسکی [Spo02] پیشنهاد می‌کند که «همه‌ی انتزاع‌های حائز اهمیت تا حدی نشتی دارند.» هدف یک انتزاع، حذف نیاز به انتقال دادن جزئیات در برقراری ارتباط است، ولی گاهی اوقات، اثرات مشکل آفرین که توسط این جزئیات تکثیر می‌شوند، نشتی پیدا می‌کنند. بدون شناخت این جزئیات، تعیین علت یک مشکل نمی‌تواند آسان باشد.

اصل ۳. تلاش برای سازگاری. خواه در حال ایجاد مدل خواسته‌ها باشید، خواه توسعه‌ی یک طراحی نرم‌افزار یا تولید کد منبع یا ایجاد use case اصل سازگاری بیان می‌کند که یک حیطه‌ی آشنا، استفاده از نرم‌افزار را آسان‌تر می‌کند. به‌عنوان مثال، طراحی واسط کاربر را برای یک برنامه‌ی کاربردی تحت وب در نظر بگیرید. تعیین مکان گزینه‌های منو، استفاده از الگوی رنگ سازگار و به‌کارگیری سازگاری آیکون‌های قابل تشخیص، همگی کمک می‌کنند که واسط دارای ظاهری ارگونومیک باشد.

اصل ۴. توجه ویژه به انتقال اطلاعات. کار نرم‌افزار، انتقال دادن اطلاعات است- از بانک اطلاعاتی به کاربر نهایی، از یک سیستم قدیمی به یک برنامه‌ی کاربردی تحت وب، از یک قطعه‌ی نرم‌افزار به قطعه‌ای دیگر، از کاربر نهایی به واسط گرافیکی کاربر (GUI)، از سیستم عامل به یک برنامه- و این فهرست تقریباً پایانی ندارد. در هر مورد، اطلاعات از طریق یک واسط جریان پیدا می‌کند و در نتیجه، فرصت‌هایی برای خطا، جاافتادگی یا ابهام پیش می‌آید. بنا به این اصل باید توجه ویژه‌ای به تحلیل، طراحی، ساخت و آزمون واسط‌ها مبذول داشت.

اصل ۵. توسعه‌ی نرم‌افزاری که ساختار پیمانه‌ای اثربخش داشته باشد. جداسازی دغدغه‌ها (اصل ۱) فلسفه‌ای برای نرم‌افزار پایه گذاری می‌کند. ساختار پیمانه‌ای، سازوکاری برای تحقق بخشیدن به این فلسفه فراهم می‌سازد. هر سیستم پیچیده‌ای را می‌توان به چند پیمانه (مؤلفه، قطعه)

تقسیم کرد، ولی کار مهندسی نرم‌افزار بیش از این‌ها را طلب می‌کند. ساختار پیمانه‌ای باید اثربخش هم باشد. یعنی، هر پیمانه باید انحصاراً جنبه‌ای از سیستم را کانون توجه قرار دهد که قیدوبندهای آن به خوبی مشخص است-یعنی از نظر عملکرد باید یکپارچه باشد و/یا در حیطه‌ای که ارائه می‌دهد، ابعادی مشخص داشته باشد. به علاوه، ارتباط میان پیمانه‌ها باید به‌شیوه‌ای نسبتاً ساده برقرار شود- هر پیمانه باید ارتباط اندکی با سایر پیمانه‌ها، منابع داده‌ها و سایر جنبه‌های محیطی داشته باشد.

اصل ۶. جستجو به‌دنبال الگوها. براد اپلتون [App00] پیشنهاد می‌کند که:

> هدف الگوها در جامعه‌ی نرم‌افزاری تهیه‌ی نوشتاری است که سازندگان را در حل مشکلات تکراری در سرتاسر فرایند توسعه‌ی نرم‌افزار یاری دهد. الگوها به ایجاد زبانی مشترک برای ارتباط مفاهیم و تجربه درباره مسائل و راهکارهای آنها کمک می‌کنند. تدوین رسمی این راهکارها و روابط آنها به ما این امکان را می‌دهد که به توده‌ی اطلاعاتی دست پیدا کنیم که درک ما از معماری خوب، برای برآوردن نیازهایمان تعیین کند.

اصل ۷. هرگاه که امکان دارد، مسأله و راهکار آن را از چند دیدگاه متفاوت به نمایش بگذارید. هنگامی که یک مسأله و راهکار آن از چند دیدگاه متفاوت به نمایش گذارده شوند، این احتمال که دید بهتری از آن به‌دست آید و خطاها و جاافتادگی‌ها کشف شوند، بیشتر خواهد شد. برای مثال، یک مدل از خواسته‌ها را می‌توان با به‌کارگیری دیدگاهی داده گرا، دیدگاهی عملیات گرا، یا دیدگاهی رفتارگرا به نمایش گذاشت (فصل‌های ۶ و ۷). هر کدام از این‌ها نمایی از مسأله و خواسته‌های آن را فراهم می‌سازند.

اصل ۸. به خاطر داشته باشید که نرم‌افزار را نگهداری خواهید کرد. در درازمدت، نرم‌افزار با کشف‌شدن نقایص بهبود خواهد یافت، خودش را با تغییرات محیط منطبق خواهد ساخت و با درخواست قابلیت‌های بیشتر از سوی ظرف‌های ذی‌نفع ارتقا خواهد یافت. این فعالیت‌های نگهداری را در صورتی می‌توان تسهیل کرد که کار مهندسی نرم‌افزار در سرتاسر فرایند نرم‌افزار لحاظ گردد.

این اصول همه‌ی آن چیزی نیست که برای ساخت نرم‌افزارهای با کیفیت بالا لازم است. بلکه مبنایی برای هر کدام از روش‌های مهندسی نرم‌افزار بحث‌شده در این کتاب فراهم می‌سازند.

اندرز

جهت کسب تجربه و دانش برای نسل‌های بعدی مهندسان نرم‌افزار، از الگوها (فصل ۱۲) استفاده کنید.

> «مهندس ایده‌آل ترکیبی از چند چیز است... دانشمند نیست، ریاضی‌دان نیست، جامعه‌شناس یا نویسنده هم نیست؛ ولی ممکن است از دانش و فن هر کدام از این رشته‌ها در حل مسائل مهندسی استفاده کند.»
>
> ان. دبلیو. دافرنی

۴-۳ اصول راهنمای فعالیت‌های چارچوبی

در بخش‌هایی که به‌دنبال خواهد آمد، به اصولی می‌پردازیم که تاثیری جدی بر موفقیت هر کدام از فعالیت‌های چارچوبی تعریف شده به‌عنوان بخشی از فرایند نرم‌افزار دارند. در بسیاری موارد، اصول مورد بحث برای هر کدام از فعالیت‌های چارچوبی، شکل پالایش یافته‌ای از اصول ارائه شده در بخش ۲-۴ هستند. در واقع اینها همان اصول هسته‌ای‌اند اما در سطح پایین‌تری از انتزاع قرار دارند.

۴-۳-۱ اصول ارتباطی

پیش از آنکه بتوان به تحلیل، مدل‌سازی یا مشخص‌کردن خواسته‌های مشتریان پرداخت، آنها را باید از طریق فعالیت‌های ارتباطی جمع آوری کرد. مشتری، مسأله‌ای دارد که می‌توان راهکاری کامپیوتری

برای آن ارائه کرد. شما به درخواست مشتری پاسخ می‌دهید. ارتباط برقرار شده است، ولی مسیر برقراری ارتباط تا شناخت، غالباً پر از دست انداز است.

برقراری ارتباط موثر (در میان همکاران فنی، با مشتری و سایر طرف‌های ذی‌نفع و با مدیران پروژه) از جمله چالش برانگیزترین فعالیت‌هایی است که با آن مواجه خواهید شد. در این حیطه، اصول ارتباطی را در کاربرد آنها برای برقراری ارتباط با مشتری مورد بحث قرار خواهیم داد. به‌هرحال، بسیاری از این اصول در سایر شکل‌های ارتباطی که در یک پروژه‌ی نرم‌افزاری رخ می‌دهند نیز کاربرد دارند.

اصل ۱. گوش‌سپردن. تلاش کنید به سخنان گوینده گوش فرا دهید، نه اینکه در آن اثنا به فکر آماده‌کردن پاسخ خود باشید. اگر چیزی برایتان واضح نیست از گوینده بخواهید تا منظور خود را به وضوح بیان کند، ولی مدام حرف او راقطع نکنید. هنگامی که طرف در حال صحبت است، هرگز در کلام یا رفتارتان ستیزه جویی نشان ندهید (مثلاً با حرکات چشم یا تکان‌دادن سر).

اصل ۲. خود را قبل از برقراری ارتباط آماده کنید. پیش از ملاقات با دیگران، قدری وقت برای دانستن مسأله صرف کنید. در صورت نیاز، قدری پژوهش کنید تا با اصطلاحات تجاری حوزه‌ی مورد نظر آشنا شوید. اگر مسؤولیت برگزاری جلسه با شماست، از قبل دستور کاری برای جلسه تهیه کنید.

اندرز

پیش از برقراری ارتباط حتماً از دیدگاه طرف مقابل شناخت داشته باشید، قدری درباره نیازهایش اطلاع حاصل کنید و سپس گوش کنید.

اصل۳. یک نفر باید این فعالیت را تسهیل کند. هر جلسه ارتباطی باید دارای رهبر (تسهیل‌گری) باشد که (۱) مکالمه را در جهتی پیش ببرد که بهره‌وری داشته باشد، (۲) هرگونه تقابلی را که رخ می‌دهد، میانجی گری کند و (۳) اطمینان حاصل کند که اصول دیگر رعایت می‌شوند.

اصل ۴. بهترین راه، ارتباط رودرروی است. ولی معمولاً در صورت وجود شکل دیگری از ارائه اطلاعات، بهتر هم می‌شود. برای مثال، یکی از شرکت کنندگان می‌تواند تصاویر یا مطالبی آماده کند تا محور بحث مشخص گردد.

اصل ۵. یادداشت بردارید و تصمیم‌گیری را مستند کنید. احتمال اینکه چیزها فراموش شوند یا نادیده انگاشته شوند، زیاد است. یکی از حاضران در جلسه باید به‌عنوان «منشی» عمل کند و نکات و تصمیم گیری‌های مهم را یادداشت کند.

> «پرسش‌های ساده و پاسخ‌های ساده، کوتاه‌ترین راه برای رسیدن به سردرگمی است.»
>
> مارک تواین

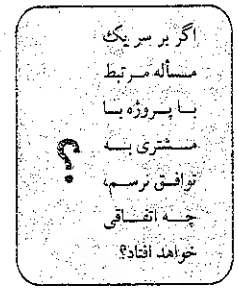
اصل ۶. تلاش برای همکاری. همکاری و اتفاق نظر هنگامی رخ می‌دهد که از دانش جمعی اعضای تیم برای توصیف قابلیت‌ها و ویژگی‌های سیستم یا محصول استفاده شود. هر همکاری کوچک، به بالا بردن اطمینان در میان اعضای تیم و ایجاد هدفی مشترک برای تیم کمک می‌کند.

اصل ۷. توجه خود را معطوف کنید؛ بحث خود را پیمانه‌ای کنید. هرچه تعداد افراد حاضر در یک ارتباط بیشتر باشد، احتمال اینکه بحث از شاخه‌ای به شاخه دیگر برود، بیشتر است. تسهیل‌گر باید مکالمه را پیمانه‌ای کند و تنها زمانی یک مبحث را ترک کند که برای آن تصمیمی گرفته شده باشد (به‌هرحال، اصل ۹ را هم ببینید.)

اصل ۸. اگر چیزی واضح نبود، یک تصویر بکشید. ارتباط لفظی حد و مرز دارد. گاهی که واژه‌ها از بیان معنی عاجزند، یک طرح یا تصویر می‌تواند مطلب را روشن کند.

اصل ۹. (الف) هنگامی که بر سر مبحثی به توافق رسیدید، به مبحث دیگر بپردازید. (ب) اگر به توافق نرسیدید، به مبحث دیگر بپردازید. (پ) اگر ویژگی یا قابلیتی واضح نیست و نمی‌توان در حال حاضر آن را واضح کرد، باز هم به مبحث دیگر بپردازید. برقراری ارتباط، نظیر هر فعالیت دیگر در مهندسی نرم‌افزار، زمان می‌برد. به‌جای تکرارهای بی پایان، افراد شرکت کننده باید بدانند که بسیاری از مباحث نیاز به بحث دارند (اصل ۲) و «پرداختن به مبحث بعدی» گاهی بهترین شیوه برای دستیابی به چابکی در برقراری ارتباط است.

اصل ۱۰. مذاکره، یک مسابقه یا بازی نیست. وقتی بهترین نتیجه را می‌دهد که هر دو طرف برنده باشند. به‌وفور پیش خواهد آمد که شما و سایر طرف‌های ذی‌نفع باید بر سر قابلیت‌ها و ویژگی‌ها، اولویت‌ها و تاریخ تحویل مذاکره کنید. اگر اعضای تیم همکاری خوبی داشته باشند، همه‌ی طرف‌ها هدفی مشترک خواهند داشت. هنوز هم مذاکره، مستلزم مصالحه از تمام طرف‌هاست.



اطلاعات

اختلاف میان مشتری و کاربر نهایی

مهندسان نرم‌افزار با طرف‌های ذی‌نفع فراوانی ارتباط برقرار می‌کنند، ولی مشتریان و کاربران نهایی بیشترین تاثیر را بر کارهای فنی بعدی دارند. در برخی موارد، مشتری و کاربر نهایی، یکی هستند، ولی در بسیاری از پروژه‌ها، مشتری و کاربر نهایی افرادی متفاوتاند که برای مدیران متفاوت و در سازمان‌های تجاری متفاوت کار می‌کنند.

مشتری به شخص یا گروهی گفته می‌شود که (۱) ابتدا درخواست می‌کند نرم‌افزاری ساخته شود، (۲) اهداف تجاری کلی برای نرم‌افزار را تعریف می‌کند، (۳) خواسته‌های پایه را فراهم می‌سازد و (۴) بودجه پروژه را تأمین می‌کند. در یک شرکت تولید سیستم یا محصول، مشتری غالباً بخش بازاریابی است. در یک محیط فن‌آوری اطلاعات، مشتری ممکن است بخش تجاری باشد.

کاربر نهایی به شخص یا گروهی گفته می‌شود که (۱) نرم‌افزار ساخته شده را برای رسیدن به یک هدف تجاری واقعاً مورد استفاده قرار دهد و (۲) جزئیات عملیاتی نرم‌افزار را تعیین کند تا هدف تجاری قابل حصول گردد.

۲-۳-۴ اصول برنامه‌ریزی

فعالیت برقراری ارتباط به شما کمک می‌کند تا اهداف و مقاصد کلی خود را تعریف کنید (البته با گذر زمان در معرض تغییر است). ولی، درک این اهداف و مقاصد به معنای تعریف طرحی برای رسیدن به آنها نیست. فعالیت برنامه‌ریزی شامل مجموعه‌ای از امور مدیریتی و فنی می‌شود که تیم نرم‌افزاری را قادر به تعریف نقشه راه در سفر به سوی اهداف راهبردی و مقاصد تاکتیکی‌اش می‌سازد.

هرچه هم که تلاش کنیم، پیش‌بینی چگونگی تکامل‌یافتن یک پروژه‌ی نرم‌افزاری غیر ممکن است. هیچ راه آسانی وجود ندارد که از طریق آن بتوان تعیین کرد چه مسائل فنی پیش‌بینی نشده‌ای ممکن است رخ دهد، چه اطلاعات فنی ممکن است تا انتهای پروژه از دیدها پنهان بمانند، چه سوء تفاهم‌هایی ممکن است رخ دهد یا کدام امور تجاری ممکن است رخ دهد. با این حال، یک تیم نرم‌افزاری خوب باید برای رویکرد خود برنامه‌ریزی کند.

«در آماده شدن برای نبرد، همواره دریافته‌ام که طرح و نقشه بی‌فایده است، ولی برنامه‌ریزی، ضروری است.»

ژنرال دوایت دی. آیزن‌هاور

SafeHome

اشتباه در برقراری ارتباط

صحنه: فضای کاری تیم مهندسی نرم‌افزار.

نقش‌آفرینان: جیمی لازار، عضو تیم نرم‌افزاری؛ وینود رامان، عضو تیم نرم‌افزاری؛ اد رابینز، عضو تیم نرم‌افزاری.

گفتگوها:

اد: چیزی از پروژه‌ی ***SafeHome*** شنیدید؟

وینود: جلسه‌ی اول برای هفته‌ی بعد تنظیم شده.

جیمی: من تا حالا یک کمی تحقیق کردم، ولی خوب پیش نرفته.

اد: منظورت چیست؟

جیمی: خب. من با لیزا چرز تماس گرفتم. او مسؤول این پروژه است.

وینود: و...؟

جیمی: از او خواستم درباره ویژگی‌ها و قابلیت‌های ***SafeHome*** حرف بزنند... از این چیزها. در عوض، او شروع کرد به سؤال‌کردن درباره سیستم‌های امنیتی، سیستم‌های اعلام حریق... من هم که در این زمینه سررشته ندارم.

وینود: چه نتیجه‌ای می‌گیری؟ (جیمی شانه‌اش را بالا می‌اندازد.)

وینود: اینکه بخش بازاریابی به ما به‌عنوان مشاور نیاز دارد و بهتر است قبل از جلسه‌ی اول، تکلیف‌شب‌هایمان را انجام بدهیم. داگ گفت که می‌خواهد با مشتری همکاری کنیم، پس بهتر است یاد بگیریم که چطوری همکاری کنیم.

اد: احتمالاً بهتر است به دفترش برویم. برای این جور کارها، تماس تلفنی نتیجه‌ی خوبی نمی‌دهد.

جیمی: هر دو شما درست می‌گویید. باید کارهایمان را هماهنگ کنیم وگرنه برقراری ارتباط از همان اول درجازدن خواهد بود.

وینود: دیدم که داگ داشت یک کتاب درباره مهندسی خواسته‌ها می‌خواند. شرط می‌بندم یک فهرست از اصول برقراری ارتباط خوب وجود دارد. می‌خواهم کتاب را از او قرض بگیرم.

جیمی: نظر خوبی است... بعد هم می‌توانی به من یاد بدهی.

وینود (با لبخند): بله. درست است.

فلسفه‌های فراوان و متفاوتی برای برنامه‌ریزی وجود دارد.[1] عده‌ای که «کمینه‌گرا» هستند چنین استدلال می‌کنند که تغییر، غالباً نیاز به برنامه‌ریزی مفصل را منتفی می‌سازد. عده‌ای دیگر که «سنت‌گرا» هستند معتقدند که برنامه‌ریزی، یک نقشه‌ی راه اثربخش فراهم می‌آورد و هرچه جزئیات آن بیشتر باشد، احتمال گم شدن تیم کمتر می‌شود. گروه دیگری هم هستند (چابک گرایان) که می‌گویند یک «بازی برنامه‌ریزی» سریع ممکن است ضروری باشد، ولی نقشه راه چیزی است که با «کار واقعی» روی نرم‌افزار شروع می‌شود.

[1] بحث مشروحی درباره برنامه ریزی و مدیریت پروژه‌های نرم‌افزاری در بخش چهارم این کتاب ارائه شده است.

چه باید کرد؟ در بسیاری از پروژه‌ها، برنامه‌ریزی بیش از حد، کاری وقت گیر و بی ثمر است (خیلی چیزها تغییر می‌کنند)، ولی از طرف دیگر کوتاهی در برنامه‌ریزی نیز به آشوب منجر می‌شود. همانند بسیاری از پدیده‌های زندگی، در برنامه‌ریزی نیز باید اعتدال را رعایت کرد، آن قدری که راهنمایی مفید برای تیم باشد- نه بیشتر و نه کمتر. برنامه‌ریزی با هر میزان سخت‌گیری که اجرا شود، اصولی که به‌دنبال خواهد آمد، همواره کاربرد خواهد داشت:

اصل ۱. شناخت حوزه‌ی پروژه. اگر ندانید مقصد کجاست، استفاده از نقشه راه غیر ممکن است. حوزه‌ی پروژه ، مقصدی برای تیم نرم‌افزاری ترسیم می‌کند.

اصل ۲. طرف‌های ذی‌نفع را در فعالیت برنامه‌ریزی دخالت دهید. طرف‌های ذی‌نفع اولویت‌ها و قیدوبندهای پروژه را تعیین می‌کنند. برای پاسخ‌گویی به این واقعیت‌ها، مهندسان نرم‌افزار باید غالباً بر سر تاریخ تحویل، زمان‌بندی و سایر مسائل مرتبط با پروژه، مذاکره کنند.

اصل ۳. این را بدانید که برنامه‌ریزی ماهیتی مبتنی بر تکرار دارد. برنامه‌ریزی پروژه چیزی نیست که روی سنگ حک شده باشد. با شروع کار، احتمال زیادی وجود دارد که اوضاع تغییر کند. در نتیجه، برنامه‌ریزی باید طوری تنظیم شود که این تغییرات را پاسخ‌گو باشد. به علاوه، در مدل‌های فرایند افزایشی و مبتنی بر تکرار، برنامه‌ریزی دوباره، پس از تحویل هر نسخه از نرم‌افزار بر اساس بازخوردهای گرفته شده از کاربران، حکمی قطعی است.

اصل ۴. برآوردهای خود را بر اساس آنچه که می‌دانید، انجام دهید. هدف از برآورد، فراهم ساختن تصویری از هزینه‌ها، کار انجام شده و مدت انجام وظایف بر اساس درک فعلی تیم از کاری است که قرار است انجام شود. اگر اطلاعات، مبهم یا غیر قابل اطمینان باشد، برآوردها نیز به همان میزان غیر قابل اطمینان خواهند بود.

اصل ۵. هم‌زمان با برنامه‌ریزی، ریسک را هم در نظر بگیرید. اگر ریسک‌هایی تعیین کرده‌اید که تاثیر و احتمال آنها بالاست، برنامه‌ریزی برای حوادث محتمل ضروری است. به علاوه، برنامه‌ریزی پروژه (که شامل زمان‌بندی هم می‌شود) باید طوری تنظیم شود که احتمال یک یا چند مورد از این ریسک‌ها در آن دیده شده باشد.

اصل ۶. واقع‌بین باشید. مردم هر روز صد در صد کار نمی‌کنند. امکان وارد شدن نویز در ارتباطات انسانی همواره وجود دارد. جاافتادگی‌ها و ابهامات، حقایق زندگی‌اند. تغییر رخ خواهد داد. حتی بهترین مهندسان نرم‌افزار هم مرتکب اشتباه می‌شوند. این واقعیت‌ها و سایر واقعیت‌ها را به هنگام تعریف برنامه‌ریزی پروژه باید در نظر داشت.

اصل ۷. هنگام تعریف برنامه‌ریزی، گرانولیته (granulity) را تعیین کنید. منظور از گرانولیته، سطحی از جزئیات است که در برنامه‌ریزی پروژه به آن پرداخته می‌شود. در برنامه‌ریزی با گرانولیته‌ی بالا، جزئیات کاری چشمگیری ارائه می‌شود که روی بازه‌های زمانی نسبتاً کوتاه برنامه‌ریزی می‌شود (به‌طوری که امور پیگیری و کنترل را بتوان به‌وفور انجام داد). در برنامه‌ریزی با گرانولیته‌ی پایین، وظایف کاری گسترده تری تعیین می‌شوند که انجام آنها روی بازه‌های زمانی گسترده‌تر برنامه‌ریزی می‌شود. به‌طور کلی، با دور شدن خط زمانی پروژه از تاریخ فعلی، سطح گرانولیته از بالا به پایین تغییر می‌کند. طی چند ماه یا چند هفته بعدی، می‌توان پروژه را با جزئیات بیشتری برنامه‌ریزی کرد. فعالیت‌هایی که تا چند ماه بعد انجام نخواهند شد، نیازی به گرانولیته‌ی بالا ندارند (تغییرات بسیاری ممکن است رخ دهد).

مرجع وب

یک منبع عالی از اطلاعات مدیریت پروژه و برنامه‌ریزی را می‌توان در آدرس زیر یافت:

www.4pm.com/repository.htm

«موفقیت، بیشتر تابعی است از عقل سلیم تا نبوغ.»

آن وانگ

نکته‌ی کلیدی

اصطلاح *گرانولیته* به سطحی از جزئیات اطلاق می‌شود که عنصری از برنامه‌ریزی در آن ارائه یا اجرا می‌شود.

اصل ۸. تعیین کنید که چگونه می‌خواهید از کیفیت اطمینان یابید. برنامه‌ریزی شما باید مشخص کند که تیم نرم‌افزاری چگونه می‌خواهد از کیفیت محصول اطمینان حاصل کند. اگر قرار باشد بازبینی‌های فنی[1] انجام شود، باید آنها را زمان‌بندی کرد. اگر قرار است از برنامه‌نویسی جفتی (فصل ۳) استفاده شود، این امر باید به وضوح در برنامه‌ریزی ذکر شود.

اصل ۹. چگونگی انجام‌دادن تغییرات را شرح دهید. حتی بهترین برنامه‌ریزی نیز ممکن است با تغییرات کنترل‌نشده، اعتبار خود را از دست بدهد. باید مشخص کنید که تغییرات را چگونه می‌توان با پیشرفت کار مهندسی نرم‌افزار انجام داد. برای مثال، آیا مشتری هر زمان درخواست تغییر دارد؟ اگر تغییری درخواست شود، آیا تیم ناگزیر از پیاده‌سازی فوری آن است؟ تاثیر و هزینه‌ی تغییر را چگونه باید ارزیابی کرد؟

اصل ۱۰. برنامه‌ریزی را به‌وفور پیگیری کنید و در صورت نیاز، تنظیماتی به عمل آورید. پروژه‌های نرم‌افزاری گاهی از زمان‌بندی عقب می‌افتند. بنابراین، پیگیری روزانه‌ی پیشرفت پروژه، جستجو به‌دنبال نواحی مشکل آفرین و شرایطی که در آن کارهای زمان‌بندی شده با کار انجام شده همخوانی ندارد، منطقی به‌نظر می‌رسد. در صورت هرگونه لغزش، طرح را باید به فراخور، تنظیم کرد.

برای این که برنامه‌ریزی بیشترین تاثیر را داشته باشد، همه‌ی اعضای تیم باید در فعالیت برنامه‌ریزی شرکت کنند.

۴-۳-۳ اصول مدل‌سازی

ما مدل‌ها را برای درک بهتر یک موجودیت واقعی که قرار است ساخته شود، ایجاد می‌کنیم. هنگامی که این موجودیت یک چیز فیزیکی باشد (مثلاً ساختمان، کارخانه یا ماشین)، می‌توانیم ماکتی بسازیم که از نظر شکل و فرم با آن یکسان باشد، ولی در مقیاسی کوچکتر. ولی، هنگامی که موجودیت ساختنی مورد نظر، نرم‌افزار باشد، مدل ما شکل متفاوتی به خود خواهد گرفت. این مدل باید قادر به نمایش اطلاعاتی که نرم‌افزار تبدیل می‌کند، معماری و عملکردهایی که رخ دادن این تبدیل را میسر می‌سازند، ویژگی‌های مطلوب کاربران و رفتار سیستم در زمان رخ دادن تبدیل، باشد. مدل‌ها باید این اهداف را در سطوح متفاوتی از انتزاع برآورده سازند- ابتدا نرم‌افزار را از دیدگاه مشتری به تصویر می‌کشد و سپس آن را در سطحی فنی‌تر به نمایش می‌گذارد.

در کار مهندسی نرم‌افزار، دو نوع مدل ایجاد می‌شود: مدل‌های خواسته‌ها و مدل‌های طراحی. مدل‌های خواسته‌ها (که مدل *تحلیل* نیز نام دارند) خواسته‌های مشتری را با تصویرکردن نرم‌افزار در سه دامنه متفاوت به نمایش می‌گذارند: دامنه‌ی اطلاعاتی، دامنه عملیاتی و دامنه رفتاری. مدل‌های طراحی، نشان‌گر خصوصیاتی از نرم‌افزارند که به نرم‌افزار نویس کمک می‌کنند تا آن را بهتر بسازد: معماری، واسط کاربر و جزئیات در سطح مؤلفه‌ها.

اسکات امبلر و ران جفریز [Amb02b] در کتاب خود که به مدل‌سازی چابک مربوط می‌شود، مجموعه‌ای از اصول مدل‌سازی[2] را تعیین می‌کنند که برای استفاده کنندگان از مدل فرایند چابک

نکته‌ی کلیدی

مدل خواسته‌ها، خواسته‌های مشتری را به نمایش می‌گذارد. مدل طراحی، یک سری مشخصات معین برای ساخت نرم‌افزار ارائه می‌دهد.

[1] بازبینی فنی موضوع فصل ۱۵ است.

[2] اصول ذکرشده در این فصل برای اهدافی که در این کتاب دنبال می‌شوند، خلاصه و دوباره بیان شده‌اند.

(فصل ۳) نوشته‌اند. ولی برای همه‌ی مهندسان نرم‌افزاری که وظایف و کنش‌های مدل‌سازی را انجام می‌دهند، مناسب هستند.

اصل ۱. هدف اصلی تیم نرم‌افزاری ساخت نرم‌افزار است نه ایجاد مدل. چابکی به معنای رساندن نرم‌افزار به مشتری در سریع‌ترین زمان ممکن است. مدل‌هایی که به رخ دادن این اتفاق کمک می‌کنند، ارزش ایجاد را دارند، ولی از مدل‌هایی که فرایند را کند کنند یا سود چندانی نداشته باشند، باید پرهیز شود.

اصل ۲. سبک‌بار سفر کنید- مدل‌هایی بیش از نیاز خود ایجاد نکنید. هر مدلی که ایجاد می‌شود باید با رخ دادن تغییرات، بهنگام‌سازی شود. مهمتر اینکه ایجاد هر مدل جدیدی زمان می‌برد که در غیر این صورت می‌توان آن را صرف مرحله‌ی ساخت (کدنویسی و آزمون) کرد. بنابراین، فقط مدل‌هایی را ایجاد کنید که ساخت نرم‌افزار را سریع‌تر و آسان‌تر سازند.

اصل ۳. بکوشید ساده‌ترین مدلی را بسازید که مسأله یا نرم‌افزار را توصیف کند. نرم‌افزار را بزرگتر از حد لازم نسازید [Amb02b]. با ساده نگه داشتن مدل‌ها، نرم‌افزار حاصل نیز ساده خواهد بود. نتیجه، نرم‌افزاری خواهد بود که انسجام بخشیدن، آزمودن و نگهداری (تغییر دادن) آن آسان‌تر است. به علاوه درک و نقد مدل‌های ساده برای اعضای تیم راحت‌تر است و حاصل کار شکل مداومی از بازخورد است که نتیجه‌ی نهایی را بهینه می‌کند.

اصل ۴. مدل‌ها را طوری بسازید که قابل تغییر باشد. فرض کنید که مدل‌های شما تغییر می‌کند، ولی اجازه ندهید که این فرض به بی نظمی شما منجر گردد. برای مثال، چون خواسته‌ها تغییر می‌کنند، معمولاً توجه چندانی به مدل خواسته‌ها نمی‌شود. چرا؟ چون می‌دانید که در هر حال تغییر خواهند کرد. مشکل این نگرش آن است که بدون یک مدل کامل از خواسته‌ها، مدل طراحی که ایجاد می‌کنید، ناگزیر فاقد یک سری قابلیت‌ها و ویژگی‌ها خواهد بود.

اصل ۵. توانایی بیان صریح هدف هر مدل ایجاد شده را داشته باشید. هر بار که مدلی را ایجاد می‌کنید از خود بپرسید چرا چنین می‌کنید. اگر نمی‌توانید توجیه قانع‌کننده‌ای برای وجود مدل ارائه کنید، وقتی صرف آن نکنید.

اصل ۶. مدل‌هایی را که توسعه می‌دهید بر سیستم مورد نظر مطابقت دهید. ممکن است برای مطابقت دادن مدل بر برنامه‌ی کاربردی به نمادگذاری یا قواعدی نیاز باشد؛ برای مثال، یک بازی کامپیوتری ممکن است به تکنیک مدل‌سازی متفاوت با یک نرم‌افزار تعبیه‌شده‌ی بی‌درنگ (که موتور خودروها را کنترل می‌کند) نیاز داشته باشد.

اصل ۷. سعی کنید مدل‌های مفید بسازید، ولی ساخت مدل‌های کامل را فراموش کنید. هنگام ساخت مدل خواسته‌ها و طراحی، مهندس نرم‌افزار به نقطه‌ای می‌رسد که دیگر ادامه‌ی کار فایده‌ای ندارد. یعنی، رسیدن به مدلی کامل و با سازگاری درونی، به تلاشی نیاز دارد که به مزایای آن نمی‌ارزد. شاید تصور کنید منظور این است که مدل‌سازی باید ناقص و با کیفیت پایین باشد؟ خیر، مدل‌سازی باید با در نظر داشتن مراحل بعدی مهندسی نرم‌افزار انجام شود. تکرار بی پایان برای رسیدن به مدلی «کامل» کمکی به چابکی نمی‌کند.

اصل ۸. در مورد قالب و نحو مدل، تعصب به خرج ندهید. اگر در انتقال مفاهیم موفق است، نمایش در مرحله‌ی دوم اهمیت قرار دارد. گرچه همه‌ی اعضای تیم نرم‌افزاری باید بکوشند تا

اندروز

هدف و مقصود هر مدل، انتقال اطلاعات است. برای رسیدن به این هدف، از قالبی سازگار استفاده کنید. فرض کنید که شما برای توضیح دادن مدل حضور ندارید. پس آن را طوری بسازید که به حضور شما نیاز نداشته باشد.

هنگام مدل‌سازی از نمادگذاری سازگار استفاده کنند. مهمترین خصلت مدل، به اشتراک گذاشتن اطلاعاتی است که وظیفه‌ی بعدی مهندسی نرم‌افزار را میسر سازد. اگر مدلی این منظور را برآورده سازد، ممکن است نحو نادرست قابل گذشت باشد.

اصل ۹. اگر غریزه شما می‌گوید مدلی درست نیست، هرچند که روی کاغذ درست به‌نظر می‌رسد، احتمالاً دلیلی برای این نگرانی دارید. اگر یک مهندس نرم‌افزار مجرب هستید، به غریزه خودتان اطمینان کنید. از کار نرم‌افزار درس‌های زیادی می‌توان فرا گرفت- که برخی از آنها در سطحی از ناخود آگاهی رخ می‌دهد. اگر چیزی به شما بگوید که یک مدل طراحی محکوم به شکست است (هر چند که دلیلی برای اثبات آن نداشته باشید) وقت بیشتری صرف بررسی مدل یا توسعه‌ی یک مدل دیگر کنید.

اصل ۱۰. به محض این که توانستید، بازخورد بگیرید. هر مدلی باید مورد بازبینی اعضای تیم نرم‌افزاری قرار گیرد. هدف از این بازبینی‌ها دریافت بازخوردی است که می‌توان آن را در تصحیح مدل‌های نادرست، تغییر دادن سوء تعبیرها و افزودن ویژگی‌ها یا قابلیت‌هایی به برنامه‌ی کاربردی که سهواً حذف شده‌اند یا جا افتاده‌اند، به‌کار گرفت.

اصول مدل‌سازی ساخته‌ها. طی سه دهه اخیر، تعداد زیادی از روش‌های مدل‌سازی خواسته‌ها توسعه داده شده است. پژوهشگران، مسائل تحلیل خواسته‌ها و علل آنها را شناسایی کرده اند و انواع نمادگذاری‌های مدل‌سازی و مجموعه‌های ابتکاری را برای غلبه بر این مشکلات توسعه داده‌اند. هر روش تحلیلی دارای دیدگاهی منحصر به فرد است. ولی، همه‌ی روش‌های تحلیل با مجموعه‌ای از اصول عملیاتی با هم مرتبط هستند.

اصل ۱. دامنه‌ی اطلاعاتی یک مسأله باید نمایش داده و درک شود. دامنه‌ی اطلاعاتی شامل داده‌هایی که به درون سیستم جریان می‌یابند (از کاربران نهایی، سیستم‌های دیگر، یا دستگاه‌های خارجی)، داده‌هایی که به خارج سیستم جریان می‌یابند (از طریق واسط کاربر، واسط‌های شبکه، گزارش‌ها، تصاویر گرافیکی و سایر ابزارها) و داده‌های ذخیره شده‌ای می‌شود که اشیای داده ماندگار (یعنی داده‌هایی که به نگهداری دائم نیاز دارند) را جمع‌آوری می‌کنند.

اصل ۲. عملکردهای نرم‌افزار باید تعریف شوند. قابلیت‌های نرم‌افزارند که بهره مستقیم را به کاربران نهایی می‌رسانند و همچنین برای ویژگی‌های قابل رؤیت برای کاربران، پشتیبانی داخلی فراهم می‌سازند. برخی از عملکردها، داده‌های جریان یافته به درون سیستم را تبدیل می‌کنند. در موارد دیگر، این قابلیت‌ها بر سطحی از کنترل روی پردازش داخلی نرم‌افزار یا عناصر سیستم خارجی تأثیر می‌گذارند. عملکردها را در سطوح متفاوتی از انتزاع می‌توان توصیف کرد، که از بیان عمومی هدف تا توصیف مفصل عناصر پردازشی را در بر می‌گیرد.

اصل ۳. رفتار نرم‌افزار (به‌عنوان نتیجه‌ای از رویدادهای خارجی) باید نمایش داده شود. رفتار نرم‌افزارهای کامپیوتری را تعامل آن با محیط خارجی تعیین می‌کند. ورودی فراهم شده توسط کاربران نهایی، داده‌های کنترلی فراهم شده توسط یک سیستم خارجی یا داده‌های پایشی جمع آوری شده روی یک شبکه، همگی باعث می‌شوند که نرم‌افزار به‌شیوه‌ای خاص رفتار کند.

اصل ۴. مدل‌هایی که اطلاعات، قابلیت‌ها و رفتارها را تصویر می‌کنند باید به‌شیوه‌ای تقسیم‌بندی شوند که جزئیات را به گونه‌ای لایه‌ای (یا سلسله مراتبی) نمایش دهند. مدل‌سازی

نکته‌ی کلیدی

در مدل‌سازی تحلیل، سه ویژگی نرم‌افزار مورد توجه قرار می‌گیرد: اطلاعاتی که باید پردازش شوند، قابلیت‌هایی که باید تحویل شوند و رفتاری که باید به نمایش گذارده شود.

خواسته‌ها، نخستین گام حل مسأله در مهندسی نرم‌افزار به‌شمار می‌رود. به کمک آن می‌توانید مسأله را بهتر درک کنید و مبنایی برای راهکار (طراحی) پایه گذاری کنید. به‌طور کلی، مسائل پیچیده را به دشواری می‌توان حل کرد. به همین دلیل، باید از راهبرد تقسیم و حل استفاده کنید. یک مسأله بزرگ و پیچیده آنقدر به مسائل فرعی تقسیم می‌شود تا اینکه هر کدام از این مسائل فرعی را بتوان به مسائل فرعی تقسیم کرد تا هر مسأله فرعی را به آسانی بتوان درک کرد. این مفهوم را *افراز* یا جداسازی دغدغه‌ها می‌نامند که راهبردی کلیدی در مدل‌سازی خواسته‌هاست.

اصل ۵. وظیفه‌ی تحلیل باید از اطلاعات ضروری به سمت جزئیات پیاده‌سازی حرکت کند. مدل‌سازی خواسته‌ها با توصیف مسأله از دیدگاه کاربر نهایی آغاز می‌شود. «جوهره‌ی» مسأله بدون در نظر گرفتن چگونگی پیاده‌سازی راهکار توصیف می‌شود. برای مثال، یک بازی کامپیوتری، بازیکن را ملزم می‌سازد که در حرکت‌کردن در داخل یک هزار توی خطرناک، شخصیت بازی را در جهتی «راهنمایی» کند. این جوهره‌ی مسأله است. جزئیات پیاده‌سازی (که معمولاً به صورت بخشی از مدل طراحی توصیف می‌شود) مشخص می‌سازد که این جوهره چگونه پیاده‌سازی خواهد شد. برای این بازی کامپیوتری، ممکن است از ورودی صوتی استفاده شود. به طریق دیگر، ممکن است فرمانی از صفحه کلید وارد شود، یک دسته‌ی بازی (یا ماوس) در جهتی مشخص حرکت داده شود، یا یک دستگاه حساس به حرکت در هوا به اهتزاز در آید.

مهندسی نرم‌افزار با بکارگیری این اصول، رویکردی سیستماتیک به مسائل خواهد داشت، ولی این اصول را چگونه در عمل می‌توان به‌کار گرفت؟ پاسخ این پرسش در فصل‌های ۵ تا ۷ داده خواهد شد.

اصول مدل‌سازی طراحی. مدل طراحی نرم‌افزار مشابه با طرح‌های معماری برای خانه است. مدل با به نمایش گذاشتن کلیت چیزی که قرار است ساخته شود، آغاز می‌شود (مثلاً ماکتی سه بعدی از خانه) و به آهستگی آن را پالایش می‌کند تا راهنمایی برای تعیین هر کدام از جزئیات فراهم آید (مثلاً در طرح لوله کشی). به‌طور مشابه، مدل طراحی ایجاد شده برای نرم‌افزار، نماهای متفاوتی از سیستم را فراهم می‌آورد.

برای به‌دست آوردن عناصر گوناگون یک طراحی نرم‌افزاری، روش‌های متعددی وجود دارد. برخی از این روش‌ها، داده‌محورند، به این معنی که ساختمان داده‌هاست که معماری برنامه و مؤلفه‌های پردازشی حاصل را تعیین می‌کند. عده‌ای دیگر، الگو محورند، یعنی برای توسعه سبک‌های معماری و الگوهای پردازشی از اطلاعات مربوط به دامنه‌ی مسأله (مدل خواسته‌ها) استفاده می‌کنند. عده‌ای هم شیءگرایند و از اشیای دامنه‌ی مسأله به‌عنوان محرکه‌ای برای ایجاد ساختمان داده‌ها و متدهای دستکاری آنها استفاده می‌کنند. با این وجود، همه‌ی این روش‌ها مجموعه‌ای از اصول طراحی را شامل می‌شوند که برای هر نوع از این روش‌ها قابل استفاده‌اند:

اصل ۱. طراحی باید تا مدل خواسته‌ها قابل ردگیری باشد. مدل خواسته‌ها، دامنه‌ی اطلاعاتی مسأله، عملکردهای قابل رؤیت کاربر، رفتار سیستم و مجموعه‌ای از کلاس‌های خواسته‌ها را توصیف می‌کند که اشیای تجاری را با متدهای عمل کننده روی آنها بسته‌بندی می‌کنند. مدل طراحی، این اطلاعات را به یک معماری (مجموعه‌ای از سیستم‌های فرعی که قابلیت‌های اصلی را پیاده‌سازی می‌کنند و مجموعه‌ای از مؤلفه‌ها که تحقق کلاس‌های خواسته‌ها هستند) ترجمه می‌کند. عناصر مدل طراحی باید تا مدل خواسته‌ها قابل ردگیری باشند.

> «نخستین مشکل مهندس در هر طراحی، کشف مسأله واقعی است.»
>
> ناشناس

> «نخست ببینید که آیا طراحی عاقلانه و درست هست و سپس با عزم و اراده آن را دنبال کنید؛ با یک ضربه از آنچه که اراده کرده‌اید، عقب ننشینید.»
>
> ویلیام شکسپیر

اصل ۲. همواره معماری سیستمی را که قرار است ساخته شود، در نظر داشته باشید. معماری نرم‌افزار (فصل ۹) اسکلت سیستمی است که قرار است ساخته شود. بر واسط‌ها، ساختمان داده‌ها، جریان کنترل برنامه و رفتار، شیوه‌ی انجام آزمون‌ها، قابلیت نگهداری سیستم حاصل و بسیاری موارد دیگر تأثیر می‌گذارد. به همه‌ی دلایلی که گفته شد، طراحی باید با ملاحظات معماری آغاز گردد. تنها پس از اینکه معماری تعیین شد، مسائل مربوط به مؤلفه‌ها را باید در نظر گرفت.

اصل ۳. طراحی داده‌ها به اندازه‌ی طراحی عملکردها اهمیت دارد. طراحی داده‌ها عنصر اساسی در طراحی معماری به‌شمار می‌رود. شیوه‌ی تحقق بخشیدن به اشیای داده در یک طراحی را نباید به بخت و اقبال واگذار کرد. طراحی‌ای که داده‌ها در آن به خوبی ساختاردهی شده باشند به ساده‌سازی جریان برنامه‌ها کمک می‌کند، طراحی و پیاده‌سازی مؤلفه‌های نرم‌افزار را ساده‌تر می‌کند و در کل، پردازش را اثربخش‌تر می‌سازد.

اصل ۴. واسط‌ها (چه درونی و چه بیرونی) باید با احتیاط طراحی شوند. شیوه‌ی جریان یافتن داده‌ها میان مؤلفه‌های یک سیستم ارتباط زیادی با کارایی پردازش، انتشار خطا و سادگی طراحی دارد. واسطی با طراحی خوب، کار انسجام‌دهی را آسان‌تر می‌کند و آزمون‌گر را در امر اعتبارسنجی عملکردهای یک مؤلفه یاری می‌دهد.

اصل ۵. طراحی واسط کاربر باید مطابق با نیازهای کاربر نهایی تنظیم گردد، ولی در هر مورد، باید برسهولت کاربرد نیز تاکید ورزیده شود. واسط کاربر نمود آشکار نرم‌افزار است. یک نرم‌افزار هر قدر هم که دارای عملکردهای درونی پیچیده باشد، هر قدر هم که ساختمان داده‌ها در آن فراگیر باشد، هر قدر که معماری آن از طراحی خوبی برخوردار باشد، اگر طراحی واسط آن ضعیف باشد، غالباً برداشت می‌شود که نرم‌افزار بد است.

اصل ۶. طراحی در سطح مؤلفه‌ها باید مستقل از عملکرد باشد. استقلال عملیاتی، میزانی از «یکپارچگی فکری» در یک مؤلفه نرم‌افزاری است. عملکردی که مؤلفه ارائه می‌دهد، باید یکپارچه باشد – یعنی باید یک و تنها یک عملکرد را کانون توجه قرار دهد.[1]

اصل ۷. مؤلفه‌ها باید با یکدیگر و با محیط خارجی ارتباطی سست داشته باشند. ارتباط به‌شیوه‌های گوناگون قابل حصول است– از طریق واسط مؤلفه‌ها، با پیام‌رسانی و از طریق داده‌های سرتاسری. با افزایش سطح ارتباط، احتمال انتشار خطا نیز بالا می‌رود و از قابلیت نگهداری نرم‌افزار کاسته می‌شود. بنابراین، ارتباط میان مؤلفه‌ها باید در سطحی منطقی حفظ گردد.

اصل ۸. نمایش‌ها (مدل‌های) طراحی باید به آسانی قابل درک باشند. هدف از طراحی، انتقال دادن اطلاعات به کسانی است که کدها را می‌نویسند، به آنها که نرم‌افزار را آزمایش می‌کنند و به سایر کسانی است که ممکن است وظیفه‌ی نگهداری از نرم‌افزار را در آینده برعهده داشته باشند. اگر درک طراحی دشوار باشد، نمی‌تواند به‌عنوان یک رسانه‌ی ارتباطی اثربخش عمل کند.

اصل ۹. طراحی باید به صورت تکراری توسعه یابد. در هر دور از تکرار، طراحی باید بکوشد تا سادگی بیشتر شود. طراحی نیز نظیر تقریباً هر فعالیت خلاقانه دیگر به‌صورت تکراری رخ می‌دهد. در اولین دورهای تکرار، طراحی پالایش و خطاها تصحیح می‌شود، ولی در دورهای نهایی تکرار، باید سعی شود که طراحی تا حد امکان ساده باشد.

[1] بحث بیشتر درباره یکپارچگی را در فصل ۸ می‌توانید بیابید.

مرجع وب

نظراتی مفید درباره فرایند طراحی همراه با بحث درباره زیبایی‌شناسی را می‌توان در مرجع زیر مشاهده کرد:
Cs.wwc.edu/~aabyan/Design/

> «اختلاف‌ها جزئی نیستند– چیزی شبیه به اختلاف میان موتسارت و سالیری. مطالعه پس از مطالعه نشان می‌دهد که بهترین طراحان، ساختارهایی ایجاد می‌کنند که سریع‌تر، واضح‌تر و ساده‌ترند و با تلاش کمتری ایجاد می‌شوند.»
>
> شرشریک، پی. بروکس

هنگامی که این اصول طراحی به‌طور مناسب به‌کار برده شوند، طراحی، هر دو نوع عوامل کیفیتی خارجی و داخلی را به نمایش می‌گذارد [Mye78]. *عوامل کیفیتی خارجی* به خواصی از نرم‌افزار گفته می‌شود که به راحتی توسط کاربران قابل مشاهده اند (مثل سرعت، قابلیت اطمینان، صحت و قابلیت استفاده). *عوامل کیفیتی داخلی*، نزد مهندسان نرم‌افزار اهمیت دارند. این عوامل از دیدگاهی فنی به طراحی با کیفیت بالا منجر می‌شوند. طراح برای دستیابی به عوامل کیفیتی داخلی باید مفاهیم طراحی پایه را درک کند (فصل ۸).

۴-۳-۴ اصول ساخت

فعالیت ساخت شامل مجموعه‌ای از وظایف کدنویسی و آزمایش می‌شود که نتیجه‌ی آن، نرم‌افزاری عملیاتی و آماده تحویل به مشتری یا کاربر نهایی است. در مهندسی نرم‌افزار مدرن، کدنویسی می‌تواند: (۱) ایجاد مستقیم کد منبع در زبان برنامه‌نویسی (مثلاً جاوا) باشد، (۲) تولید خودکار کد منبع با استفاده از یک نمایش شبه طراحی از مؤلفه‌ای باشد که قرار است ساخته باشد یا (۳) تولید خودکار کد قابل اجرا با استفاده از یک زبان برنامه‌نویسی نسل چهارم (مانند Visual C++) باشد.

در مرحله‌ی آزمون، توجه به سیستم ابتدا در سطح مؤلفه‌ها رخ می‌دهد. این رویکرد را *آزمون واحدها* می‌نامند. سایر سطوح آزمون عبارتند از (۱) آزمون انسجام (که با ساخت سیستم اجرا می‌شود)، آزمون اعتبارسنجی که برآورده شدن خواسته‌ها در سیستم (یا نسخه‌ی نرم‌افزار) کامل شده را ارزیابی می‌کند و (۳) آزمون پذیرش که توسط مشتری و به‌عنوان تلاشی برای تمرین روی همه‌ی قابلیت‌ها و ویژگی‌های خواسته شده اجرا می‌شود. مجموعه مفاهیم و اصول بنیادی زیر در کدنویسی و آزمون کاربرد دارند:

اصول کدنویسی. اصول راهگشا در وظیفه‌ی کدنویسی، بستگی تنگاتنگی با سبک برنامه نویسی، زبان برنامه‌نویسی و شیوه‌ی برنامه‌نویسی مورد نظر دارند، اما چند اصل بنیادی در این زمینه قابل بیان است:

اصول آماده‌سازی: پیش از آنکه حتی یک خط برنامه بنویسید، اطمینان حاصل کنید که

- می‌دانید چه مسأله‌ای را قرار است حل کنید.
- اصول و مفاهیم طراحی پایه را می‌دانید.
- زبانی برای برنامه‌نویسی انتخاب کنید که نیازهای نرم‌افزار و محیطی را که قرار است در آن کار کند، برآورده سازد.
- محیطی برای برنامه‌نویسی انتخاب کنید که ابزارهای لازم برای آسان ترکردن کار را در اختیارتان قرار دهد.
- مجموعه‌ای از آزمون‌های پایه را ایجاد کنید که با کامل شدن کدنویسی مؤلفه بتوانید آنها را به‌کار ببرید.

اصول برنامه نویسی: با شروع به کدنویسی، اطمینان حاصل کنید که

- الگوریتم‌هایتان را با دنباله‌روی از برنامه‌سازی ساخت یافته، مقید کنید [Boh00].
- استفاده از برنامه‌نویسی جفتی را در نظر داشته باشید.

«من بیشتر عمرم را صرف نگاه‌کردن به کدهای دیگران کرده‌ام. گاهی یک قطعه جواهر واقعی پیدا می‌کنم، یک برنامه با ساختاری خوب که به سبکی سازگار نوشته شده است، طوری توسعه یافته است که هر مؤلفه‌ی آن ساده و منظم است و طوری طراحی شده است که محصول را می‌توان به راحتی تغییر داد.»

دیوید پارنز

- ساختمان داده‌هایی را انتخاب کنید که نیازهای طراحی را برآورده کند.
- معماری نرم‌افزار را بشناسید و واسط‌هایی سازگار با آن بسازید.
- منطق شرطی را تا حد امکان ساده نگه دارید.
- حلقه‌های تو در تو را به‌شیوه‌ای بنویسید که به آسانی قابل آزمون باشند.
- نام‌هایی با معنی برای متغیرها انتخاب کنید و از سایر استانداردهای کدنویسی محلی پیروی کنید.
- کدهایی بنویسید که خود مستندسازی شده باشند.
- یک چیدمان بصری ایجاد کنید (مثلاً با تو رفتگی و خطوط خالی) که به فهم کدهای شما کمک کند.

اصول اعتبارسنجی: پس از به پایان رساندن اولین دور کدنویسی، حتماً

- در صورت امکان، گشتی در میان کدها بزنید.
- آزمون واحدها را اجرا کنید و خطاهایی را که کشف می‌شوند، تصحیح نمایید.
- کدها را بازآرایی کنید.

درباره برنامه‌نویسی (کدنویسی) و اصول و مفاهیم آن بیش از هر مبحث دیگری در فرایند نرم‌افزار کتاب نوشته شده است. کتاب‌هایی در این مبحث شامل کارهای اولیه روی سبک برنامه‌نویسی [Ker78]، ساخت عملی نرم‌افزار [McC04]، اندیشه‌های ناب برنامه‌نویسی [Ben99]، هنر برنامه‌نویسی [Knu98]، مسائل عملی برنامه‌نویسی [Hun99] و بسیاری از موضوع‌های دیگر می‌شوند. بحث جامعی درباره این اصول و مفاهیم، از حوزه‌ی این کتاب خارج است. در صورت تمایل می‌توانید به منابع ذکرشده رجوع کنید.

اصول آزمون. گلن مایرز در کتابی که برای آزمون نرم‌افزار نوشته است [Mye79] چند قاعده را بیان می‌کند که می‌توان آنها را به خوبی به‌عنوان اهداف آزمون در نظر گرفت:

- آزمون، فرایند اجرای برنامه به قصد یافتن خطاهاست.
- یک مورد آزمون خوب باید خطاهای کشف نشده را با احتمال زیادی کشف کند.
- آزمون موفق، آزمونی است که خطای کشف نشده تاکنون را کشف کند.

این اهداف نشان‌گر تغییر دیدگاهی مهیج برای برخی نرم‌افزارنویسان است. آنها بر خلاف این دیدگاه رایج حرکت می‌کنند که «آزمون موفق، آزمونی است که در آن هیچ خطایی یافت نشود.» هدف شما طراحی آزمون‌هایی است که به صورت سیستماتیک انواع متفاوت خطاها را کشف کند و این وظیفه را در کمترین مقدار از زمان و کار به انجام رسانند.

اگر قرار باشد که آزمون با موفقیت به انجام رسد (مطابق با اهدافی که پیش از این بیان شد)، خطاهای موجود در نرم‌افزار را کشف خواهد کرد. به‌عنوان یک مزیت ثانویه، آزمون نشان می‌دهد که عملکردهای نرم‌افزار ظاهراً مطابق با مشخصات ذکر شده کار می‌کنند و به‌نظر می‌رسد که خواسته‌های رفتاری و کارایی برآورده شده‌اند. به علاوه، داده‌های جمع آوری شده به هنگام انجام آزمون، شاخص خوبی از قابلیت اطمینان نرم‌افزار و شاخصی از کیفیت نرم‌افزار در کل به‌دست می‌دهند. ولی آزمون نمی‌تواند نبودن خطاها و نقایص را نشان بدهد؛ فقط می‌تواند نشان دهد که خطاها و نقایص وجود دارند. هنگام اجرای آزمون‌ها همواره باید این جمله (نسبتاً غم انگیز) را به خاطر داشت.

اندرز

از توسعه یک برنامه ظریف و زیبا که مسأله‌ای اشتباهی را حل می‌کند، بپرهیزید. به اولین اصل آمادگی، توجهی خاص داشته باشید.

مرجع وب

گستره‌ی وسیعی از پیوندهای حاوی استانداردهای کدنویسی را می‌توانید در آدرس زیر بیابید:
www.literateprogramming.com/fpstyle.html

اهداف آزمون نرم‌افزار چیست؟

اندرز

در یک حیطه‌ی گسترده‌تر طراحی نرم‌افزار به خاطر بسپارید که با بذل توجه به معماری نرم‌افزار در مقیاس بزرگ شروع می‌کنید و در پایان با بذل توجه به مؤلفه‌ها، در مقیاس کوچک ادامه می‌دهید. برای آزمون، کافی است، این روند را معکوس کنید.

دیویس [Dav95b] مجموعه‌ای از اصول آزمون[1] را پیشنهاد می‌کند که برای استفاده در این کتاب قدری آنها را تغییر داده ایم:

اصل ۱. همه‌ی آزمون‌ها تا خواسته‌های مشتری قابل ردگیری باشند.[2] هدف از آزمون نرم‌افزار، کشف خطاهاست. پس اکثر تقایص شدید (از دیدگاه مشتری) آنهایی هستند که باعث می‌شوند برنامه نتواند خواسته‌ها را برآورده سازد.

اصل ۲. آزمون‌ها را باید مدت‌ها قبل از شروع آزمون برنامه‌ریزی کرد. برنامه‌ریزی آزمون‌ها (فصل ۱۷) را می‌توان به محض کامل شدن مدل خواسته‌ها آغاز کرد. تعریف جزئیات هر مورد آزمون را می‌توان به محض شکل گرفتن مدل طراحی آغاز کرد. بنابراین، همه‌ی آزمون‌ها را می‌توان قبل از تولید هرگونه کدی برنامه‌ریزی کرد.

اصل ۳. اصل پارتو در آزمون نرم‌افزار کاربرد دارد. در این حیطه، اصل پارتو بدان معناست که اثر ۸۰٪ از همه‌ی خطاهای کشف شده طی آزمون را احتمالاً در ۲۰٪ از کل مؤلفه‌های نرم‌افزار می‌توان پیدا کرد. بدیهی است که مشکل، جداکردن این مؤلفه‌های مظنون و آزمون کامل آنهاست.

اصل ۴. آزمون باید در مقیاس کوچک آغاز شود و به سمت مقیاس بزرگ پیش برود. نخستین آزمون‌های برنامه‌ریزی و اجرا شده عموماً مؤلفه‌های **منفرد** را کانون توجه قرار می‌دهند. با پیش رفتن آزمون، این کانون توجه به سمت تلاش برای یافتن خطاها در خوشه‌های منسجمی از مؤلفه‌ها و سرانجام در کل سیستم جابجا می‌شود.

اصل ۵. آزمون کامل امکان‌پذیر نیست. تعداد حالت‌های ممکن حتی برای یک برنامه با اندازه متوسط، به‌طور نمایی، بزرگ است. به همین دلیل، اجرای هر ترکیبی از مسیرها طی انجام آزمون‌ها غیر ممکن می‌شود، ولی این امکان وجود دارد که منطق برنامه به‌طور مناسب پوشش داده شود تا اطمینان حاصل شود که همه‌ی شرایط طراحی در سطح مؤلفه‌ها تمرین داده شده‌اند.

۴-۳-۵ اصول استقرار

چنان که پیش از این نیز در بخش اول کتاب گفته شد، فعالیت استقرار شامل سه کنش می‌شود: تحویل، پشتیبانی و بازخورد. چون مدل‌های فرایند نرم‌افزار مدرن، ماهیتی تکاملی یا افزایشی دارند، استقرار به‌یک‌باره رخ نمی‌دهد بلکه با حرکت نرم‌افزار به‌سوی تکامل، چند بار تکرار می‌شود. هر چرخه‌ی تحویل، یک نسخه‌ی عملیاتی از نرم‌افزار در اختیار مشتری و کاربران نهایی قرار می‌دهد که قابلیت‌ها و ویژگی‌های جدیدی فراهم می‌سازد. هر چرخه‌ی پشتیبانی، کمک انسانی و مستندسازی برای کلیه قابلیت‌ها و ویژگی‌های ارائه شده طی همه‌ی چرخه‌های استقرار تا آن زمان را فراهم می‌سازد. هر چرخه‌ی بازخورد، راهنمایی‌های مهمی را در اختیار تیم نرم‌افزاری قرار می‌دهد که به اصلاح قابلیت‌ها، ویژگی‌ها و رویکرد در نظر گرفته شده برای نسخه‌ی بعدی نرم‌افزار می‌انجامند.

[1] فقط زیرمجموعه کوچکی از اصول آزمایش دیویس در اینجا ذکر شده است. برای اطلاعات بیشتر، [Dav95b] را ببینید.

[2] این اصل به آزمون‌های عملیاتی اشاره دارد، یعنی آزمون‌هایی که بر خواسته‌ها تاکید دارند. آزمون‌های ساختاری (آزمون‌هایی که بر جزئیات معماری یا منطقی تاکید دارند) ممکن است خواسته‌های مشخصی را مستقیماً مورد توجه قرار ندهند.

تحویل یک نسخه از نرم‌افزار، نشان‌گر نقطه‌ی عطف مهمی برای هر پروژه‌ی نرم‌افزاری است. در همان حال که تیم آماده تحویل یک نسخه‌ی جدید می‌شود، چند اصل کلیدی را باید رعایت کند:

اصل ۱. انتظارات مشتری برای نرم‌افزار باید مدیریت شود. به‌وفور پیش می‌آید که انتظارات مشتری بیش از آن چیزی باشد که تیم قول آن را داده است و بلافاصله نارضایتی شروع می‌شود. این امر منجر به بازخوردی می‌شود که فاقد بهره است و باعث دلسردی تیم می‌شود. نائومی کارتن [Kar94] در کتاب خود که به مدیریت انتظارات مربوط می‌شود، چنین می‌گوید: «نقطه شروع برای انتظارات مشتری، وظیفه‌شناسی بیشتر درباره نحوه برقراری ارتباط و موضوع این ارتباط است.» او معتقد است که مهندس نرم‌افزار باید درباره ارسال پیام‌های متضاد (مثلاً قول تحویل بیش از آنچه که در بازه زمانی موجود امکان‌پذیر است یا تحویل بیش از آنچه که برای یک نسخه قول داده‌اید و تحویل کمتر از آنچه که برای نسخه‌ی دیگر قول داده اید) احتیاط کند.

اصل ۲. پکیج تحویل کامل باید مونتاژ و آزمایش شود. یک CD-ROM یا سایر رسانه‌ها (از جمله دانلودهای مبتنی بر وب) حاوی کلیه نرم‌افزارهای اجرایی، فایل‌های داده‌ای پشتیبان، مستندات پشتیبان و سایر اطلاعات مرتبط را باید در پکیج لحاظ کرد و با کاربران واقعی به‌طور کامل مورد آزمون بتا[1] قرار داد. همه‌ی اسکریپت‌های نصب و سایر ویژگی‌های عملیاتی را باید روی هر تعداد ممکن از پیکربندی‌های کامپیوتری متفاوت (یعنی سخت افزار، سیستم‌های عامل، دستگاه‌های جانبی، چیدمان شبکه) به‌طور کامل تمرین داد.

اصل ۳. پیش از تحویل نرم‌افزار، یک روال پشتیبانی باید مشخص کرد. وقتی پرسش یا مشکلی پیش می‌آید، کاربر نهایی انتظار پاسخ‌گویی و اطلاعات صحیح دارد. اگر پشتیبانی، تک‌منظوره باشد، یا بدتر از آن، اصلاً وجود نداشته باشد، مشتری بلافاصله ناراضی خواهد شد. پشتیبانی باید برنامه‌ریزی شود، مواد پشتیبانی باید آماده شود و سازو کارهایی مناسب برای حفظ سوابق باید وضع شود تا تیم نرم‌افزاری بتواند انواع پشتیبانی را مورد ارزیابی قرار دهد.

اصل ۴. مواد آموزشی مناسب باید برای کاربران نهایی تهیه شود. تیم‌های نرم‌افزاری، چیزی بیش از خود نرم‌افزار تحویل می‌دهند. کمک آموزشی‌های مناسب (در صورت نیاز) باید تهیه شود؛ دستورالعمل‌هایی برای اشکال‌زدایی باید ارائه شود و در صورت نیاز، جزوه‌ای باید منتشر شود تا شرح دهد که این نسخه از نرم‌افزار چه تفاوتی با نسخه‌های قبلی دارد.[2]

اصل ۵. نرم‌افزار مشکل‌دار ابتدا باید اصلاح و بعداً تحویل داده شود. برخی سازمان‌های نرم‌افزاری تحت فشار زمانی، نسخه‌هایی با کیفیت ضعیف تحویل می‌دهند با این هشدار که اشکال‌های موجود در نسخه‌ی بعدی نرم‌افزار برطرف خواهد شد. این اشتباه است. معروف است که می‌گویند: «مشتریان فراموش خواهند کرد که نرم‌افزاری با کیفیت بالا را چند روزی دیرتر تحویل داده‌اید، ولی هرگز مشکلات ناشی از یک محصول با کیفیت پایین را فراموش نخواهند کرد. این نرم‌افزار هر روز این مشکل را به‌یادشان خواهد آورد.»

آندرز

اطمینان حاصل کنید که مشتری شما می‌داند قبل از تحویل نسخه‌ی نرم‌افزار چه انتظاراتی باید داشته باشد. در غیر این صورت، شک نکنید که انتظاراتش بیش از آن چیزی خواهد بود که شما تحویل می‌دهید.

[1] beta test

[2] طی فعالیت برقراری ارتباط، تیم نرم‌افزاری باید تعیین کند که کاربر چه انواعی از مواد کمکی را لازم دارد.

نرم‌افزار تحویل‌شده برای کاربر نهایی مزیت به همراه دارد، ولی برای تیم نرم‌افزاری نیز بازخوردهای مفیدی به همراه خواهد داشت. با به‌کار گرفته شدن نسخه‌ی جدید نرم‌افزاری، کاربران نهایی باید به اظهار نظر درباره قابلیت‌ها و ویژگی‌ها، سهولت کاربرد، قابلیت اطمینان و هر خصوصیت دیگری که مناسب به‌نظر می‌رسد، تشویق گردند.

۴-۴ خلاصه

کار مهندسی نرم‌افزار شامل اصول، مفاهیم، روش‌ها و ابزارهایی می‌شود که مهندسان نرم‌افزار در سرتاسر فرایند مهندسی نرم‌افزار به‌کار می‌برند. هر پروژه‌ی مهندسی نرم‌افزار با پروژه‌ی دیگر تفاوت دارد. با این وجود، مجموعه‌ای از اصول کلی در یک فرایند به صورت یک کلیت و در انجام هر کدام از فعالیت‌های چارچوبی کاربرد دارند و این به نوع پروژه یا محصول بستگی ندارد.

مجموعه‌ای از اصول هسته‌ای به استفاده از یک فرایند نرم‌افزار و اجرای روش‌های اثربخش مهندسی نرم‌افزار کمک می‌کنند. در سطح فرایند، اصول هسته‌ای، بنیادی فلسفی فراهم می‌سازند که تیم نرم‌افزاری را در سرتاسر فرایند نرم‌افزار یاری می‌دهد. در سطح کاری، این اصول هسته‌ای مجموعه‌ای از ارزش‌ها و قواعد را مستقر می‌سازند که به‌عنوان راهنما، شما را در تحلیل مسأله، طراحی راهکار، پیاده‌سازی و آزمون راهکار و سرانجام، استقرار نرم‌افزار در جامعه‌ی کاربران یاری می‌دهند.

اصول ارتباطی، نیاز به کاهش نویز و بهبود بخشیدن به پهنای باند در مکالمه‌ی میان دست‌اندرکاران و مشتریان تأکید دارند. هر دو طرف باید برای رخ دادن بهترین ارتباطات، همکاری کنند.

اصول برنامه‌ریزی، دستورالعمل‌هایی برای تهیه‌ی بهترین نقشه راه برای ساخت سیستم یا محصول کامل فراهم می‌سازند. این برنامه‌ریزی و نقشه ممکن است تنها برای یک نسخه از نرم‌افزار طراحی شده باشد یا ممکن است برای کل پروژه تعریف شود. در هر حال، برنامه‌ریزی باید کار مورد نظر، کنندگان آن کار، و زمان به انجام رسیدن آن را در بر گیرد.

مدل‌سازی شامل هر دو فعالیت طراحی و تحلیل و توصیف نمایش‌هایی از نرم‌افزار می‌شود که پیوسته بر جزئیات آنها افزوده می‌شود. هدف از مدل‌سازی، تبلور شناخت شما از کاری که باید انجام شود و فراهم آوردن راهنمایی فنی برای آنهاست که نرم‌افزارها را پیاده‌سازی می‌کنند. اصول مدل‌سازی به‌عنوان مبنایی برای روش‌ها و نمادگذاری مورد استفاده در ایجاد نمایش‌هایی از نرم‌افزار عمل می‌کنند.

مرحله‌ی ساخت شامل یک چرخه‌ی کدنویسی و آزمون می‌شود که در آن کد منبع برای یک مؤلفه، ایجاد و آزموده می‌شود. اصول کدنویسی، کنش‌هایی کلی را تعریف می‌کنند که پیش از نوشته شدن کدها، در حال نوشته شدن آنها و پس از کامل شدن آنها رخ می‌دهند. گرچه اصول فراوانی برای آزمون وجود دارد، تنها یک اصل است که غالب است: آزمون، عبارت است از اجرای یک برنامه به قصد یافتن خطاها.

مرحله‌ی استقرار با ارائه شدن هر نسخه از نرم‌افزار به مشتری رخ می‌دهد و شامل تحویل، پشتیبانی و بازخورد می‌شود. در اصول کلیدی مربوط به تحویل نرم‌افزار، مدیریت انتظارات مشتری و فراهم ساختن اطلاعات پشتیبانی مناسب برای نرم‌افزار مد نظر بوده است. پشتیبانی مستلزم آمادگی قبلی است. بازخورد به مشتری این امکان را می‌دهد که تغییراتی با ارزش تجاری پیشنهاد کند و برای چرخه‌ی بعدی مهندسی نرم‌افزار، خوراک ورودی تأمین می‌کند.

مسائل و نکاتی برای تعمق

۴-۱ از آنجا که توجه به کیفیت، نیاز به صرف وقت و منابع دارد، آیا می‌توان چابک بود و در عین حال، بر کیفیت تأکید داشت؟

۴-۲ از هشت اصل هسته‌ای که راهنمای فرایند هستند (بخش ۱-۲-۴) به اعتقاد شما کدام یک بیشترین اهمیت را دارد؟

۴-۳ مفهوم جداسازی دغدغه‌ها را به زبان ساده شرح دهید.

۴-۴ یک اصل ارتباطی مهم می‌گوید «پیش از برقراری ارتباط، خود را آماده کنید.» این آمادگی چگونه باید در کارهای اولیه‌ای که انجام می‌دهید، نمود پیدا کند؟ به‌عنوان نتیجه‌ای از این آمادگی اولیه چه محصولات کاری نتیجه خواهد شد؟

۴-۵ روی موضوع «تسهیل» برای یک فعالیت ارتباطی قدری پژوهش کنید (از منابع ذکر شده یا هر منبع دیگر استفاده کنید) و مجموعه‌ای از دستورالعمل‌ها را تهیه کنید که صرفاً بر تسهیل تأکید دارند.

۴-۶ برقراری ارتباط چابک چه تفاوتی با برقراری ارتباط در مهندسی نرم‌افزار سنتی دارد؟ چه شباهتی با آن دارد؟

۴-۷ چرا «حرکت به جلو» ضروری است؟

۴-۸ روی «مذاکره» برای یک فعالیت ارتباطی قدری «پژوهش» کنید و مجموعه‌ای از دستورالعمل‌ها را تهیه کنید که صرفاً بر مذاکره تأکید دارند.

۴-۹ شرح دهید که گرانولیته در حیطه‌ی زمان‌بندی پروژه چه معنایی دارد.

۴-۱۰ چرا مدل‌ها در کار مهندسی نرم‌افزار اهمیت دارند؟ آیا همیشه به آنها نیاز است؟ آیا می‌توانید درباره پاسخی که در خصوص این نیاز داده‌اید، توضیح بیشتری بدهید؟

۴-۱۱ سه «دامنه‌ای» که باید طی مدل‌سازی خواسته‌ها به‌کار گرفت، کدام اند؟

۴-۱۲ بکوشید یک اصل دیگر به اصول بیان شده برای کدنویسی در بخش ۴-۳-۴ اضافه کنید.

۴-۱۳ آزمون موفق کدام است؟

۴-۱۴ برای مخالفت یا موافقت خود با این جمله، دلایلی ارائه دهید: «چون ما چند نسخه از نرم‌افزار را به مشتری ارائه می‌دهیم، چرا باید در همان نسخه‌های اولیه، دغدغه‌ی کیفیت را داشته باشیم- می‌توانیم مشکلات را در دورهای بعدی تکرار برطرف کنیم.»

۴-۱۵ چرا بازخورد برای تیم نرم‌افزاری اهمیت دارد؟

فصل ۵

شناخت خواسته‌ها

نگاهی گذرا

خواسته‌ها چیستند؟ پیش از شروع هر کار فنی، فکر خوبی است که یک مجموعه وظایف مهندسی برای خواسته‌ها تعیین کنید. این وظایف به درک اثر تجاری نرم‌افزار، آنچه که مشتری می‌خواهد و چگونگی تعامل کاربران نهایی با نرم‌افزار می‌انجامد.

چه می‌کند؟ مهندسان نرم‌افزار (که در دنیای IT گاه از آنها به‌عنوان مهندس سیستم یا «تحلیل‌گر» یاد می‌شود) و سایر طرف‌های ذی‌نفع در پروژه (مدیران، مشتریان و کاربران نهایی) همگی در مهندسی خواسته‌ها مشارکت دارند.

چرا اهمیت دارد؟ طراحی و ساخت یک برنامه کامپیوتری زیبا که مسأله‌ای نادرست را حل می‌کند، نیازی را از کسی برطرف نمی‌سازد. از همین رو، پیش از شروع به طراحی و ساخت یک سیستم کامپیوتری، درک و شناخت آنچه که مشتری می‌خواهد، اهمیت دارد.

مراحل کار کدام است؟ نخستین مرحله در مهندسی خواسته‌ها، مرحله‌ی *شروع* (inception) است-این وظیفه‌ی حوزه و ماهیت مسأله‌ای را که باید حل شود، تعیین می‌کند. در ادامه‌ی آن نوبت به وظیفه‌ی دیگری می‌رسد که *استخراج* (elicitation) نام دارد؛ این وظیفه به طرف‌های ذی‌نفع کمک می‌کند تا آنچه را که مورد نیاز است، تعریف کنند و سپس نوبت به *شناخت* (elaboration) می‌رسد- در این مرحله، خواسته‌ها پالایش و اصلاح می‌شود. آن هنگام که طرف‌های ذی‌نفع، مسأله را تعریف می‌کنند، مذاکره آغاز می‌شود- بر سر اینکه اولویت‌ها و ضروریات کدامند و چه هنگام مورد نیازند؟ سرانجام، مسأله به نحوی مشخص می‌گردد و سپس مرور و اعتبارسنجی می‌شود تا اطمینان حاصل شود که شناخت شما از مسأله و شناخت طرف‌های ذی‌نفع از مسأله با هم مطابقت دارد.

محصول کار چیست؟ هدف و مقصود از مهندسی خواسته‌ها، فراهم ساختن یک گزارش مکتوب از مسأله برای همه‌ی طرف‌هاست. این هدف از طریق چند محصول کاری قابل دستیابی است: سناریوهای کاربرد، فهرست ویژگی‌ها و عملکردها، مدل خواسته‌ها یا یک مشخصه.

چگونه اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ محصولات کاری در مهندسی خواسته‌ها با طرف‌های ذی‌نفع مرور می‌شود تا اطمینان حاصل شود که آنچه شما فهمیده‌اید، واقعاً همان چیزی است که منظور آنها بوده است. و یک هشدار: حتی پس از توافق همه‌ی طرف‌ها، اوضاع تغییر می‌کند و این تغییرات در سرتاسر پروژه ادامه خواهد یافت.

شناخت خواسته‌های یک مسأله از جمله دشوارترین وظایفی است که مهندس نرم‌افزار با آن مواجه است. در نگاه نخست، شناخت خواسته‌ها کار چندان دشواری به‌نظر نمی‌رسد. هر چه که باشد، آیا مشتری نمی‌داند که چه می‌خواهد؟ آیا کاربران نهایی نباید درک خوبی از ویژگی‌ها و قابلیت‌های سیستم داشته باشند؟ شگفت اینکه در بسیاری موارد، پاسخ به این پرسش، منفی است. حتی اگر مشتریان و کاربران نهایی در بیان نیازهایشان صراحت داشته باشند، آن نیازها در سرتاسر پروژه تغییر خواهد کرد.

من در پیش گفتار کتابی از رالف بانگ [You01] درباره تعیین مؤثر خواسته‌ها چنین نوشتم:

> بدترین کابوس شما همین است. مشتری قدم به دفترتان می‌گذارد، می‌نشیند، مستقیم در چشم شما می‌نگرد و می‌گوید: «می‌دانم که فکر می‌کنید می‌فهمید چه گفتم، ولی چیزی که نمی‌فهمید همان است که گفتم، نه آن که منظورم بوده است.» این اتفاق در اواخر پروژه و زمانی که مهلت‌های پروژه به پایان رسیده است، اعتبار شغلی آدم‌ها مورد سؤال قرار گرفته است و پول زیادی خرج شده است، رخ می‌دهد.
>
> هر یک از ما که در تجارت نرم‌افزار و سیستم‌ها چند سالی را کار کرده باشد، با این کابوس زندگی کرده‌ایم و تازه تعدادی از ما هم یاد گرفته‌اند که آن را از خود دور کنند. ما تلاش می‌کنیم که خواسته‌ها را از مشتری بیرون بکشیم. در فهم اطلاعاتی که به‌دست می‌آوریم مشکل داریم. غالباً خواسته‌ها را به‌شیوه‌ای سازمان‌دهی نشده ثبت می‌کنیم و زمان بسیار اندکی را صرف وارسی آنچه ثبت شده است، می‌کنیم. به‌جای آنکه سازوکارهایی برای کنترل تغییرات وضع کنیم، اجازه می‌دهیم که تغییرات ما را کنترل کنند. به‌طور خلاصه، عاجزیم از اینکه بنیادی محکم برای سیستم یا نرم‌افزار برقرار سازیم. هر کدام از این مشکلات ایجاد چالش می‌کند و این مشکلات در کنار هم حتی مجرب‌ترین مدیران و دست‌اندرکاران را به وحشت می‌اندازد، ولی راهکارهایی هم وجود دارد.

منطقی است که استدلال کنیم تکنیک‌های بحث شده در این فصل، «راهکار» واقعی برای چالش‌های ذکر شده به‌شمار نمی‌روند، ولی رویکردی مناسب برای پرداختن به آنها فراهم می‌آورند.

۱-۵ مهندسی خواسته‌ها (Requirements Engineering)

طراحی و ساخت نرم‌افزارهای کامپیوتری کاری است چالش‌برانگیز، خلاقانه و در عین حال جالب. در واقع، ساخت نرم‌افزار چنان جذاب است که بسیاری از سازندگان پیش از آنکه به درستی بدانند چه چیزی مورد نیاز است، شروع به ساخت نرم‌افزار می‌کنند. استدلال آنها از این قرار است که: به موازات پیشرفت فرایند ساخت نرم‌افزار، خواسته‌ها معلوم می‌شود، طرف‌های ذی‌نفع تنها پس از بررسی و آزمون دوره‌های اولیه‌ی تکرار نرم‌افزار می‌توانند نیازهای خود را شناسایی کنند، اوضاع چنان سریع تغییر می‌کند که هر گونه تلاش در شناسایی مشروح خواسته‌ها اتلاف وقت است، هدف اصلی ساخت نرم‌افزاری است که کار کند و هر چیز دیگری در وهله دوم اهمیت قرار می‌گیرد. نکته فریبنده در خصوص این استدلال‌ها آن است که تنها عناصری از واقعیت در آنها وجود دارد.[1] ولی هر کدام از آنها دارای عیب و نقص است و می‌تواند باعث شکست پروژه‌ی نرم‌افزاری شود.

> «دشوارترین بخش در ساخت یک سیستم نرم‌افزاری، تصمیم‌گیری در این خصوص است که چه باید ساخته شود. هیچ بخش از کار نیست که اگر درست انجام نشود به این اندازه باعث وارد آمدن ضربه به سیستم شود. درست کردن هیچ بخش دیگری در آینده تا این حد دشوار نخواهد بود.»
>
> فرد بروکس

[1] این به‌ویژه برای پروژه‌های کوچک (کمتر از یک ماه) و نرم‌افزارهایی ساده و نسبتاً کوچک صادق است. با رشد اندازه و پیچیدگی نرم‌افزار، این استدلال‌ها به شکست می‌انجامد.

طیف گسترده‌ای از وظایف و فنونی که به شناخت خواسته‌ها می‌انجامد، *مهندسی خواسته‌ها* نامیده می‌شود. از دیدگاه فرایند نرم‌افزاری، مهندسی خواسته‌ها یک کنش اصلی در مهندسی نرم‌افزار به‌شمار می‌رود که طی فعالیت *برقراری ارتباط* آغاز می‌شود و تا فعالیت *مدل‌سازی* ادامه می‌یابد و در آن هم ادامه دارد. این کنش را باید بر نیازهای فرایند، پروژه، محصول و دست‌اندرکاران آن منطبق ساخت.

مهندسی خواسته‌ها پلی است به سوی طراحی و ساخت، ولی نقطه‌ی شروع این پل کجاست؟ می‌توان استدلال کرد که نقطه‌ی شروع آن درست جلوی پای طرف‌های ذی‌نفع (مدیران، مشتریان، کاربران نهایی) است، آنجا که نیازهای تجاری تعیین می‌شود، سناریوهای کاربری توصیف می‌شود، قابلیت‌ها و ویژگی‌های عملیاتی مشخص می‌شوند و قیدوبندهای پروژه شناسایی می‌شود. عده‌ای دیگر ممکن است پیشنهاد کنند که این کنش با یک تعریف سیستمی وسیع‌تر آغاز شود؛ آنجا که نرم‌افزار چیزی نیست جز مؤلفه‌ای از یک سیستم بزرگتر، ولی نقطه شروع هر چه که باشد، با طی‌کردن این پل است که می‌توانید پروژه را آغاز کنید و این امکان فراهم می‌شود که حیطه‌ی کاری را بررسی کنید؛ نیازهای خاصی که در طراحی و ساخت باید به آنها پرداخته شود؛ اولویت‌هایی که ترتیب به انجام رساندن کارها را مشخص می‌کنند؛ و اطلاعات، وظایف و رفتارهایی که تأثیری عمیق بر طراحی خواهند داشت.

مهندسی خواسته‌ها برای شناخت آنچه که مشتری می‌خواهد، تحلیل نیازها، امکان‌سنجی، مذاکره بر سر یک راهکار منطقی، تعیین مشخصات راهکار به‌طور واضح، اعتبارسنجی این مشخصات و مدیریت خواسته‌ها به موازاتی که به یک سیستم عملیاتی تبدیل می‌شوند، سازوکار مناسب را فراهم می‌آورد [Tha97]. این کنش شامل هفت وظیفه‌ی متمایز می‌شود: شروع، استخراج، شناخت، مذاکره، تعیین مشخصات، اعتبارسنجی و مدیریت. ذکر این نکته حائز اهمیت است که برخی از این وظایف به‌صورت موازی قابل انجام بوده بر نیازهای پروژه مطابقت داده می‌شوند.

شروع. یک پروژه‌ی نرم‌افزاری چگونه آغاز می‌شود؟ آیا رویدادی وجود دارد که به تنهایی کاتالیزوری برای ایجاد یک محصول یا سیستم کامپیوتری جدید می‌شود، یا اینکه نیازها به مرور زمان تکامل می‌یابند؟ پاسخ مشخصی بر این پرسش‌ها وجود ندارد. در برخی موارد، تنها چیزی که برای به جریان انداختن یک تلاش مهندسی نرم‌افزار مورد نیاز است، گفتگویی معمولی است، ولی به‌طور کلی، اکثر پروژه‌ها با تعیین یک نیاز تجاری یا یک بازار بالقوه جدید یا کشف یک سرویس جدید آغاز می‌شوند. ذی‌نفع‌ها، از جامعه‌ی تجاری (مثلاً مدیران تجاری، بازاریاب‌ها، مدیران تولید) برای ایده‌ی مورد نظر یک مورد تجاری تعریف می‌کنند، تلاش می‌کنند وسعت و عمق بازار را بیابند، یک امکان‌سنجی تقریبی انجام دهند و توصیفی کاری از دامنه‌ی پروژه ارائه دهند. همه‌ی این اطلاعات در معرض تغییرات قرار دارد، ولی برای شروع بحث و تبادل نظر با سازمان مهندسی نرم‌افزار کفایت می‌کند.[1]

در مرحله‌ی شروع[2]، شناختی پایه‌ای از مسأله، افرادی که خواهان راهکاری برای آن هستند، ماهیت راهکار مطلوب و اثربخشی ارتباطات مقدماتی و همکاری میان سایر طرف‌های ذی‌نفع و تیم نرم‌افزاری به‌دست می‌آید.

> **نکته‌ی کلیدی**
>
> مهندسی خواسته‌ها، بنیادی محکم برای طراحی و ساخت فراهم می‌سازد. نرم‌افزار حاصل بدون آن به احتمال زیاد نیازهای مشتری را برآورده نخواهد کرد.

> **اندرز**
>
> انتظار انجام قدری طراحی در کار خواسته‌ها و قدری کار خواسته‌ها در طراحی را داشته باشید.

> «بذر اکثر مصیبت‌های نرم‌افزاری معمولاً در سه ماه اول شروع پروژه پاشیده می‌شود.»
>
> کاپر جونز

[1] اگر یک سیستم کامپیوتری قرار باشد ساخته شود، بحث و تبادل نظر در حیطه‌ی فرایند مهندسی سیستم آغاز می‌شود. برای بحث مشروح درباره مهندسی سیستم، به وب‌سایت این کتاب رجوع کنید.

[2] به خاطر دارید که در فرایند یکپارچه (فصل ۲) یک «فاز شروع» فراگیرتر تعریف می‌شود که شامل همین وظایف دریافت، استخراج و جزئیات بحث شده در این فصل می‌شود.

استخراج. این مرحله به قدر کافی ساده به‌نظر می‌رسد–از مشتری، کاربران و سایرین بپرسیم که اهداف محصول یا سیستم کدامند، چه چیز باید انجام شود، سیستم یا محصول چگونه باید بر نیازهای تجاری مطابقت داشته باشند و سرانجام اینکه سیستم یا محصول قرار است چگونه مورد استفاده قرار گیرد، ولی این ساده نیست– بسیار هم دشوار است. کریستال و کانگ [Cri92] چند مورد از مشکلاتی را که در طول مرحله‌ی استخراج ممکن است پیش آید، شناسایی کرده‌اند:

- **مشکلات مربوط به حوزه‌ی پروژه.** مرزهای سیستم به‌خوبی مشخص نشده است یا مشتریان/کاربران، جزئیات فنی غیر ضروری را مشخص می‌کنند که ممکن است اهداف سیستم را بیشتر مبهم سازند تا اینکه باعث وضوح شوند.
- **مشکلات مربوط به درک پروژه.** مشتریان/کاربران به‌طور کامل مطمئن نیستند که چه چیزهایی مورد نیاز است، از قابلیت و محدودیت‌های محیط کامپیوتری خود شناخت ضعیفی دارند، درک کاملی از دامنه‌ی مسأله ندارند، در برقراری ارتباط با مهندس سیستم برای انتقال دادن نیازهای خود مشکل دارند، اطلاعاتی را که به‌نظر آنها بدیهی به‌نظر می‌رسد، حذف می‌کنند، خواسته‌هایی را مشخص می‌کنند که با نیازهای سایر کاربران/مشتریان تضاد دارد، یا خواسته‌هایی مبهم و ناپایدار را مطرح می‌کنند.
- **مشکلات مربوط به تغییرپذیری.** خواسته‌ها با گذشت زمان تغییر می‌کنند. برای کمک به غلبه بر این مشکلات، باید جمع‌آوری خواسته‌ها را به‌شیوه‌ای سازمان یافته به اجرا گذاشت.

چرا به‌دست آوردن شناختی واضح از آنچه که مشتری می‌خواهد، دشوار است؟

شناخت. اطلاعات به‌دست آمده از مشتری در طول مراحل دریافت و استخراج، بسط داده‌شده در مرحله‌ی شناخت، پالایش می‌یابد. آنچه در این وظیفه مورد توجه قرار می‌گیرد، توسعه‌ی مدلی پالایش یافته است (فصول ۷ و ۸) که جنبه‌های گوناگون عملکرد نرم‌افزار، رفتار و اطلاعات آن را مشخص سازد.

اندرز
شناخت، چیز خوبی است اما باید بدانید که چه هنگام آن را متوقف سازید. کلید آن هم توصیف مسأله است به‌گونه‌ای که پستری مستحکم برای طراحی پایه‌ریزی کنید. اگر ورای این نقطه کار کنید، در حال کار طراحی هستید.

نیروی محرکه‌ی جزئیات، ایجاد و پالایش سناریوهای کاربری است که چگونگی تعامل کاربر نهایی (و سایر کنش‌گران) با سیستم را توصیف می‌کند. هر سناریو به یک سری کلاس‌های تحلیل استخراج شده تجزیه می‌شود– موجودیت‌های دامنه‌ی تجاری که کاربر نهایی قادر به دیدن آنهاست. صفات هر کلاس تحلیل، تدوین و سرویس‌های[1] لازم برای هر کلاس تحلیل تعریف می‌شوند. روابط و همکاری‌های میان کلاس‌ها شناسایی می‌شود و انواع نمودارهای مکمل، ترسیم می‌شود.

مذاکره. اینکه مشتریان و کاربران خواسته‌هایی داشته باشند که بر اساس محدودیت‌های موجود در منابع تجاری، برآورده کردن آنها امکان‌پذیر نباشد، پدیده‌ای عادی است. این هم نسبتاً عادی است که کاربران یا مشتریان متفاوت، خواسته‌هایی متضاد داشته باشند، با این استدلال که خواسته‌ی آنها برای نیازهای ویژه شرکت، ضروری است.

این تضادها را باید از طریق یک فرایند مذاکره به توافق برسانید. از مشتریان، کاربران و سایر طرف‌های ذی‌نفع خواسته می‌شود که نیازهای خود را رتبه‌بندی کنند و سپس تضادها و مغایرت‌ها را

اندرز
در یک مذاکره‌ی اثربخش، نه برنده و نه بازنده‌ای نباید وجود داشته باشد. برد باید با هر دو طرف باشد چون معامله‌ای بسته‌ایده است که هر دو طرف در آن قادر به ادامه‌ی حیات باشند.

[1] یک سرویس، داده‌های پنهان‌سازی شده در یک کلاس را دستکاری می‌کند. از واژه‌های عمل یا متد نیز استفاده می‌شود. اگر با مفاهیم شیءگرا آشنا نیستید، پیوست ۲ را ببینید.

اطلاعات

قالب تعیین مشخصات خواسته‌های نرم‌افزار

مشخصات خواسته‌های نرم‌افزار (SRS) سندی است که هنگامی ایجاد می‌شود که توصیفی مشروح از همه‌ی جنبه‌های نرم‌افزار مورد نظر فراهم شده باشد. لازم به ذکر است که همواره یک SRS رسمی نوشته نمی‌شود. در واقع، نمونه‌های فراوانی وجود دارد که در آنها، تلاش‌های به‌عمل‌آمده برای ایجاد یک SRS را بهتر است روی فعالیتی دیگر صرف کرد، ولی هنگامی که قرار است نرم‌افزار را یک طرف سوم بسازد، هنگامی که فقدان مشخصات باعث مشکلات تجاری جدی می‌شود، یا هنگامی که سیستمی بی اندازه پیچیده است یا اهمیت تجاری بسیار دارد، ایجاد SRS می‌تواند توجیه داشته باشد.

کارل ویگرز [Wie03] از شرکت Process Impact الگوی با ارزشی ابداع کرده است (قابل دسترس در **www.processimpact.com/process-assets/srs-template.doc**) که می‌تواند به‌عنوان دستورالعملی برای تهیه یک SRS کامل عمل کند.

فهرست محتویات

۵. سایر خواسته‌های غیر عملکردی

۵-۱ خواسته‌های کارایی

۵-۲ خواسته‌های ایمنی

۵-۳ خواسته‌های امنیتی

۵-۴ صفات کیفیت نرم‌افزار

۶. سایر خواسته‌ها

پیوست الف: واژگان

پیوست ب: مدل‌های تحلیل

پیوست پ: فهرست مشکلات

شرح مفصلی از هر کدام از مباحث SRS را می‌توانید با دانلود قالب SRS از آدرس ذکر شده در حاشیه صفحه قبل به‌دست آورید.

بر اساس اولویت‌ها مورد بحث قرار دهند. استفاده از یک روش مبتنی بر تکرار که خواسته‌ها را اولویت‌بندی می‌کند، هزینه و ریسک آنها را ارزیابی می‌نماید، تضادهای داخلی را برطرف می‌سازد، خواسته‌هایی را حذف و تعدادی از آنها را با هم تلفیق و/ یا اصلاح می‌کند، به‌طوری که همه‌ی طرف‌ها به میزانی از رضایت برسند.

تعیین مشخصات. در حیطه‌ی سیستم‌های کامپیوتری (و نرم‌افزار) واژه مشخصات برای افراد متفاوت، معنای متفاوت دارد. مشخصات می‌تواند یک مجموعه مستندات مکتوب، یک مجموعه مدل‌های گرافیکی، یک مدل ریاضی رسمی، مجموعه‌ای از سناریوهای کاربرد، یک نمونه‌ی اولیه یا هر ترکیبی از موارد ذکر شده باشد.

عده‌ای پیشنهاد می‌کنند که [Sam97] برای تعیین مشخصات باید یک «الگوی استاندارد» توسعه یابد و به‌کار گرفته شود، با این استدلال که به این ترتیب، خواسته‌ها به‌شیوه‌ای سازگارتر و بنابراین پایدارتر، ارائه خواهند شد. ولی، گاهی حفظ انعطاف‌پذیری به هنگام تعیین مشخصات ضروری است. برای یک سیستم بزرگ، مستندات مکتوب، تلفیق توصیف‌های زبان طبیعی و مدل‌های گرافیکی ممکن است بهترین روش باشد، ولی برای محصولات یا سیستم‌های کوچکتر که در محیط‌های فنی مشخص به‌کار گرفته می‌شوند، سناریوهای کاربردی می‌توانند کافی باشند.

نکته‌ی کلیدی

میزان رسمیت و قالب تعیین مشخصات بسته به اندازه و پیچیدگی نرم‌افزاری که قرار است ساخته شود، متغیر است.

اعتبارسنجی. محصولات کاری تولیدشده به‌عنوان نتیجه‌ای از مهندسی خواسته‌ها در مرحله‌ی اعتبارسنجی مورد ارزیابی کیفی قرار می‌گیرند. در اعتبارسنجی خواسته‌ها، مشخصات بررسی می‌شود[1] تا اطمینان حاصل شود که: همه‌ی خواسته‌های نرم‌افزار بدون هرگونه ابهام بیان شده‌اند؛ ناسازگاری‌ها، جاافتادگی‌ها و خطاها شناسایی و تصحیح شده‌اند و محصولات کاری از استانداردهای وضع‌شده برای فرایند، پروژه و محصول پیروی می‌کنند.

[1] به‌خاطر بسپارید که ماهیت مشخصات با هر پروژه‌ای تغییر خواهد کرد. « مشخصات» در برخی موارد، مجموعه‌ای از سناریوهای کاربری و قدری چیزهای دیگر است. در موارد دیگر، مشخص‌سازی ممکن است مستندی باشد حاوی سناریوها، مدل‌ها و توصیف‌های مکتوب.

سازوکار اصلی برای اعتبارسنجی خواسته‌ها عبارت از مرور فنی این خواسته‌هاست (فصل ۱۵). تیم مرور که خواسته‌ها را اعتبارسنجی می‌کند، شامل مهندسان نرم‌افزار، مشتریان، کاربران و سایر طرف‌های ذی‌نفعی می‌شود که مشخصات را بررسی می‌کنند و به‌دنبال خطاهایی در محتویات خواسته‌ها یا تعابیر آنها، نقاطی که ممکن است به توضیح نیاز داشته باشند، اطلاعات ناقص، ناسازگاری‌ها (مشکل بزرگی که هنگام کار با سیستم‌های بزرگ پیش می‌آید)، خواسته‌های متضاد، یا خواسته‌های غیرواقع‌بینانه (غیر قابل دستیابی) می‌گردند.

اندرز

یک دغدغه مهم طی اعتبارسنجی خواسته‌ها، سازگاری است. برای اینکه از بیان سازگار خواسته‌ها اطمینان پیدا کنید، از مدل تحلیل استفاده نمایید.

اطلاعات

چک‌لیست اعتبارسنجی خواسته‌ها

بررسی هر کدام از خواسته‌ها در مقابل مجموعه‌ای از پرسش‌های یک چک‌لیست، غالباً کار مفیدی است. زیر مجموعه کوچکی از این گونه پرسش‌ها در زیر داده شده است:

- آیا خواسته‌ها به وضوح بیان شده‌اند؟ آیا امکان سوء تعبیر از آنها وجود دارد؟
- آیا منبع خواسته (شخص، قانون یا مستند) معلوم است، آیا بیان نهایی خواسته با منبع مربوط چک شده است؟
- آیا خواسته دارای قیدوبندهای کمّی است؟
- کدام خواسته‌های دیگر با این خواسته در ارتباط هستند؟ آیا از طریق یک ماتریس مرجع متقاطع (cross-reference matrix) یا سازوکار دیگری به‌وضوح بیان شده‌اند؟
- آیا خواسته از هر گونه قیدوبند در دامنه‌ی سیستم عدول می‌کند؟
- آیا خواسته، آزمون‌پذیر است؟ اگر هست، آیا برای بررسی آن می‌توان آزمون‌هایی (که گاه ملاک‌های اعتبارسنجی نامیده می‌شوند) وضع کرد؟
- آیا خواسته تا هر مدل سیستمی‌ای که ایجاد شده است قابل ردگیری هست؟
- آیا خواسته تا اهداف محصول/ سیستم در کل قابل ردگیری هست؟
- آیا مشخصات از چنان ساختاری برخوردار هستند که به درک آسان، ارجاع آسان و تبدیل آسان به محصول کاری فنی‌تر منجر شوند؟
- آیا برای مشخصات، شاخصی ایجاد شده است؟
- آیا رابطه‌ی میان خواسته‌ها و کارایی، رفتار و خصوصیات عملیاتی به وضوح بیان شده است؟ کدام خواسته‌ها به صراحت بیان نشده‌اند؟

مدیریت خواسته‌ها. خواسته‌ها برای سیستم‌های کامپیوتری تغییر می‌کنند، و این گرایش به تغییر خواسته‌ها در سرتاسر حیات سیستم باقی می‌ماند. مدیریت خواسته‌ها عبارت از مجموعه‌ای از فعالیت‌هاست که تیم پروژه را در شناسایی، کنترل، پیگیری خواسته‌ها و تغییرات به‌عمل‌آمده در آنها در هر زمان از پیشرفت پروژه یاری می‌دهد[1]. بسیاری از این فعالیت‌ها همان تکنیک‌های مدیریت پیکربندی سیستم (SCM) هستند که در فصل ۲۲ بحث شده‌اند.

[1] مدیریت خواسته‌های رسمی تنها برای پروژه‌های بزرگی انجام می‌شود که صدها خواسته‌ی قابل شناسایی دارند. برای پروژه‌های کوچک، این کنش مهندسی خواسته تا حد چشمگیری کمتر رسمی است.

ابزارهای نرم‌افزاری

مهندسی خواسته‌ها

هدف: ابزارهای مهندسی نرم‌افزار به جمع‌آوری خواسته‌ها، مدل‌سازی خواسته‌ها، مدیریت خواسته‌ها و اعتبارسنجی خواسته‌ها کمک می‌کنند.

مکانیک: مکانیک این ابزارها متفاوت است. به‌طور کلی، ابزارهای مهندسی خواسته‌ها انواع مدل‌های گرافیکی (مانند UML) را می‌سازند که جنبه‌های اطلاعاتی، عملیاتی و رفتاری یک سیستم را به تصویر می‌کشند. این مدل‌ها مبنایی برای سایر فعالیت‌ها در فرایند نرم‌افزار فراهم می‌سازند.

ابزارهای نمونه

فهرست جامع (و بهنگام شده ای) از ابزارهای مهندسی خواسته‌ها را می‌توان در سایت منابع Volvere Requirements (در آدرس www.volvere.co.uk/tools.htm) مشاهده کرد. ابزارهای مدل‌سازی خواسته‌ها در فصل ۶ و ۷ بحث خواهند شد. ابزارهایی که در زیر ذکر شده‌اند بر مدیریت خواسته‌ها تأکید دارند.

Easy RM که توسط Cybernetic Intelligence GmbH (www.easy-rm.com) توسعه یافته است، یک واژه نامه مختص پروژه می‌سازد که حاوی توصیف مشروحی از خواسته‌ها و صفات آنهاست.

Rational Requisite Pro که توسط شرکت نرم‌افزاری Rational Software (www.360.ibm.com/software/awdtools/reqpro/) به کاربر امکان ساخت یک بانک اطلاعاتی از خواسته‌ها را می‌دهد؛ روابط میان خواسته‌ها را به نمایش در می‌آورد؛ و خواسته‌ها را سازمان‌دهی، اولویت‌بندی و ردیابی می‌کند.

ابزارهای مدیریت فراوان دیگری را می‌توان در سایت Volvere که پیش از این ذکر شد در آدرس زیر پیدا کرد:

www.jiludwig.com/Requirements-Management-Tools.html

۵-۲ تدارک مقدمات کار

در شرایط ایده‌آل، طرف‌های ذی‌نفع و مهندسان نرم‌افزار در یک تیم کار می‌کنند.[1] در این گونه موارد، مهندسی خواسته‌ها صرفاً انجام مکالمات با معنی با همکارانی است که اعضای شناخته شده‌ای از تیم هستند، ولی واقعیت چیز دیگری است.

مشتری(ها) یا کاربران نهایی ممکن است در شهر یا کشوری دیگر باشند، ممکن است از آنچه که مورد نیاز است فقط تصویری مبهم در ذهن داشته باشند، ممکن است درباره سیستمی که قرار است ساخته شود، آرای متناقض داشته باشند، ممکن است دانش فنی محدودی داشته باشند و ممکن است زمان چندانی برای تعامل با مهندس خواسته‌ها نداشته باشند. هیچ یک از این موارد، مطلوب نیست، ولی همه‌ی آنها کاملاً رایج هستند و شما غالباً ناگزیر از کار در قیدوبندهای ناشی از این شرایط هستید.

[1] این روش قویاً برای پروژه‌هایی توصیه می‌شود که فلسفه توسعه‌ی چابک برای آنها برگزیده می‌شود.

در بخش‌هایی که به‌دنبال خواهد آمد، مراحل مورد نیاز برای تدارک مقدمات شناخت خواسته‌های نرم‌افزار را مورد بحث قرار خواهیم داد- تا پروژه به‌شیوه‌ای آغاز گردد که برای رسیدن به راهکاری موفق حرکت خود را آغاز کند.

۵-۲-۱ شناسایی طرف‌های ذی‌نفع

سامرویل و سایر [Sam97]، ذی‌نفع را به‌عنوان «هر کسی که به‌طور مستقیم یا غیر مستقیم از سیستم در حال توسعه بهره مند خواهد شد» تعریف می‌کنند. من قبلاً به این موارد اشاره کرده‌ام: مدیران عملیات تجاری، مدیران تولید، بازاریاب‌ها، مشتریان داخلی و خارجی و کاربران نهایی، مشاوران، مهندسان تولید، مهندسان نرم‌افزار، مهندسان پشتیبانی و نگهداری و سایران. هر کدام از این طرف‌های ذی‌نفع، از زاویه‌ای متفاوت به سیستم نگاه می‌کند و هنگامی که سیستم با موفقیت توسعه یافت، بهره‌ی متفاوتی از آن خواهد برد و در صورتی که توسعه‌ی نرم‌افزار به شکست بینجامد، متحمل خطرات متفاوتی خواهد شد.

در مرحله‌ی «شروع» باید از کسانی که هنگام استخراج خواسته‌ها سهمی در این کار دارند، فهرستی تهیه کنید (بخش ۳-۵). این فهرست اولیه با برقراری تماس با طرف‌های ذی‌نفع رشد خواهد کرد، چون از هرکدام از این افراد این سؤال پرسیده خواهد شد که «دیگر با چه‌کسی باید صحبت کنیم؟»

نکته‌ی کلیدی

طرف ذی‌نفع هر کسی می‌تواند باشد که از سیستم در حال ساخت به‌طور مستقیم بهره‌مند می‌شود یا به آن توجهی خاص دارد.

۵-۲-۲ شناخت دیدگاه‌های چندگانه

از آنجا که طرف‌های ذی‌نفع فراوانی وجود دارند، خواسته‌های سیستم از دیدگاه‌های متفاوت بسیار بررسی می‌شود. برای مثال، گروه بازاریابی به قابلیت‌ها و ویژگی‌هایی علاقه مند است که بازار بالقوه را برانگیخته کند و فروش سیستم جدید را آسان‌تر نماید. مدیران تجاری به مجموعه‌ای از قابلیت‌ها علاقه نشان می‌دهند که با بودجه‌ی موجود قابل ساخت باشند و آمادگی برآوردن پارامترهای تعریف شده برای بازار را داشته باشد. کاربران نهایی ممکن است ویژگی‌هایی را بخواهند که با آنها آشنایی دارند و یادگیری و استفاده از آنها آسان باشد. مهندسان نرم‌افزار ممکن است به قابلیت‌هایی توجه کنند که از دید طرف‌های غیر فنی پنهان می‌ماند، ولی زیر ساختی را فراهم می‌سازند که قابلیت‌ها و ویژگی‌های بهتری برای پشتیبانی از بازار ارائه می‌دهند. مهندسان پشتیبانی ممکن است توجه خود را به قابلیت نگهداری نرم‌افزار معطوف نمایند.

«سه طرف ذی‌نفع را در اتاقی قرار دهید و از آنها بپرسید که چه نوع سیستمی می‌خواهند. احتمالاً بیش از سه ایده متفاوت خواهید داشت.»

ناشناس

هر کدام از این گروه‌ها اطلاعاتی را در فرایند مهندسی خواسته‌ها به‌اشتراک خواهندگذاشت. همچنان که اطلاعات از دیدگاه‌های چندگانه جمع‌آوری می‌شوند، خواسته‌های نوظهور ممکن است با یکدیگر در تناقض یا ناسازگار باشند. شما باید همه‌ی اطلاعات به‌دست آمده از طرف‌های ذی‌نفع (از جمله خواسته‌های ناسازگار و متناقض) را به‌شیوه‌ای گروه‌بندی کنید که به تصمیم گیران این امکان را بدهید تا مجموعه‌ای از خواسته‌ها را انتخاب کنند که از سازگاری درونی برخوردار باشد.

۵-۲-۳ تلاش برای همکاری

اگر پنج طرف ذی‌نفع درگیر یک پروژه‌ی نرم‌افزاری باشند، ممکن است پنج ایده‌ی متفاوت (یا حتی بیشتر) درباره مجموعه‌ی خواسته‌های مناسب داشته باشید. در سرتاسر فصل‌های گذشته، گفته ام که مشتریان (و سایر طرف‌های ذی‌نفع) باید با یکدیگر (با پرهیز از جدل‌های کودکانه) و با دست‌اندرکاران مهندسی نرم‌افزار همکاری کنند تا سیستمی موفق ساخته شود.

وظیفه مهندسی خواسته‌ها، شناسایی وجوه اشتراک (یعنی خواسته‌هایی که همه طرف‌های ذی‌نفع در آنها اتفاق نظر دارند) و موارد متضاد یا ناسازگار (یعنی خواسته‌هایی که یک طرف می‌پسندد، ولی با خواسته‌های طرف دیگر تناقض دارد) است. بدیهی است که گروه دوم خواسته‌هاست که ایجاد چالش می‌کند.

همکاری الزاماً به این معنی نیست که خواسته‌ها را به‌صورت گروهی تعیین کنند. در بسیاری موارد، طرف‌های ذی‌نفع با ارائه دیدگاهی از خواسته‌ها همکاری می‌کنند، ولی تصمیم‌گیری نهایی درباره انتخاب خواسته‌ها برعهده‌ی یک «پهلوان پروژه» (مثلاً مدیر تجاری یا مسؤول ارشد فن‌آوری) است.

اطلاعات

استفاده از «اُمتیازهای اولویت‌دار»

یک راه برای برطرف‌کردن خواسته‌های متناقض و در عین حال، شناخت بهتر اهمیت نسبی همه‌ی خواسته‌ها، استفاده از یک الگوی «رأی‌گیری» مبتنی بر امتیازهای اولویت‌دار است. چند امتیاز اولویت‌دار در اختیار همه‌ی طرف‌های ذی‌نفع قرار داده می‌شود که باید این امتیازها را «خرج» تعدادی از اولویت‌ها کنند. فهرستی از خواسته‌ها ارائه می‌شود و هر کدام از طرف‌های ذی‌نفع، اهمیت نسبی هر خواسته را (از دید خودش) با دادن یک یا چند امتیاز به آن خواسته تعیین می‌کند. از امتیازهای خرج شده، دیگر نمی‌توان دوباره استفاده کرد. هنگامی که شخص همه‌ی امتیازهای خود را خرج کرد، دیگر نمی‌تواند به خواسته دیگری امتیاز بدهد. کل امتیازهای خرج شده روی هر خواسته توسط همه‌ی طرف‌های ذی‌نفع، شاخصی از اهمیت نسبی آن خواسته در اختیار می‌گذارد.

۵-۲-۴ پرسیدن نخستین سؤالات

پرسش‌هایی که در مرحله‌ی شروع پروژه مطرح می‌شود باید «مستقل از حیطه‌ی پروژه» باشد [Gau89]. در نخستین مجموعه از پرسش‌های «مستقل از حیطه»، مشتری و سایر طرف‌های ذی‌نفع، اهداف کلی پروژه و منافع آن کانون توجه قرار می‌گیرند. برای مثال، ممکن است بپرسید:

- چه کسی این کار را خواسته است؟
- چه کسی از راهکار ارائه شده استفاده خواهد کرد؟
- راهکار موفق چه مزایای اقتصادی به همراه خواهد داشت؟
- منبع دیگری برای راهکار وجود دارد که به آن نیاز داشته باشید؟

به کمک این پرسش‌ها می‌توانید همه‌ی طرف‌های ذی‌نفعی را که به نوعی به نرم‌افزار مورد نظر توجه نشان می‌دهند، شناسایی کنید. به‌علاوه، این پرسش‌ها مزیت قابل اندازه‌گیری را برای پیاده‌سازی موفق و سایر راه‌های موجود برای سفارشی‌کردن توسعه‌ی نرم‌افزار را تعیین می‌کنند.

به کمک مجموعه پرسش‌های بعدی، می‌توانید درک بهتری از مسأله به‌دست آورید و مشتری می‌تواند آنچه از یک راهکار در ذهن دارد، به زبان آورد:

- از خروجی «خوبی» که یک راهکار موفق ایجاد می‌کند، چه توصیفی دارید؟
- این راهکار به چه مسأله(هایی) اختصاص دارد؟
- آیا می‌توانید محیط تجاری‌ای را که این راهکار در آن استفاده می‌شود به من نشان دهید؟ (یا آن را توصیف کنید؟)

«دانستن بعضی پرسش‌ها بهتر از دانستن همه‌ی جواب‌هاست.»

جیمز تربر

کدام پرسش‌ها شما را در رسیدن به درکی مقدماتی از مسأله کمک خواهد کرد؟

- آیا مسائل یا قیدوبندی‌های عملکردی خاص، بر شیوه‌ی نگرش به راهکار تأثیر دارد؟

در واپسین مجموعه از این پرسش‌ها، مؤثر بودن خود فعالیتِ برقراری ارتباط است که مورد توجه قرار می‌گیرد. گاوز و واینبرگ [Gau89] این‌ها را «شبه‌پرسش» نام نهاده و فهرست (خلاصه‌شده) زیر را پیشنهاد کرده‌اند:

- آیا شما فرد مناسب برای پاسخ گویی به این پرسش‌ها هستید؟ آیا پاسخ‌های شما «رسمی» هست؟
- آیا پرسش‌های من ربطی به مسأله‌ی شما دارد؟
- من زیاد سؤال نمی‌کنم؟
- کس دیگری هست که اطلاعات دیگری ارائه کند؟
- آیا چیزی دیگری هست که بپرسم؟

این پرسش‌ها (و سایر پرسش‌ها) به «فتح باب» و شروع برقراری ارتباط لازم برای استخراجی موفق کمک خواهد کرد، ولی پرسش و پاسخ در قالب یک جلسه، رویکردی نیست که موفقیت چشمگیری داشته باشد. در واقع، جلسه‌ی پرسش و پاسخ را فقط باید برای اولین برخورد به‌کار برد و از آن پس یک قالب استخراج خواسته‌ها را به‌کار برد که عناصر حل مسأله، مذاکره و تعیین مشخصات با هم تلفیق می‌کند. رویکردی از این نوع در بخش ۳-۵ ارائه شده است.

«آن که سؤالی می‌پرسد، فقط پنج دقیقه نادان است؛ آن که سؤالی نمی‌پرسد، تا ابد نادان باقی می‌ماند.»

ضرب المثل چینی

۵-۳ استخراج خواسته‌ها

استخراج خواسته‌ها (که جمع‌آوری خواسته‌ها نیز گفته می‌شود) عناصر حل مسأله، شناخت، مذاکره و تعیین مشخصات را در هم می‌آمیزد. به‌منظور تشویق به یک رویکرد «تیم محور» و با روحیه همکاری برای جمع‌آوری خواسته‌ها، طرف‌های ذی‌نفع با هم کار می‌کنند تا مسأله را شناسایی کنند، عناصر حل را پیشنهاد کنند، بر سر رویکردهای متفاوت مذاکره کنند و مجموعه‌ای مقدماتی از خواسته‌های راهکار را مشخص سازند [Zah90][1]

۵-۳-۱ همکاری در جمع‌آوری خواسته‌ها

روش‌های فراوان و متفاوتی برای همکاری در جمع‌آوری خواسته‌ها پیشنهاد شده است. در هر روش سناریوی متفاوتی استفاده می‌شود، ولی همه‌ی آنها با قدری تغییرات، مبتنی بر دستورالعمل‌های اصلی زیرند:

- در جلسات هم مهندسان نرم‌افزار و هم سایر طرف‌های ذی‌نفع، شرکت دارند.
- قواعد آماده‌سازی و مشارکت وضع می‌شود.
- دستور کار پیشنهادی به قدر کافی رسمیت دارد که همه‌ی نکات مهم را پوشش دهد و در عین حال به قدر کافی غیر رسمی هست که جریان آزاد ایده‌ها را میسر سازد.
- یک «تسهیل‌گر» (facilator) (که می‌تواند از مشتریان، سازندگان یا فردی خارجی باشد) کنترل جلسه را به‌دست می‌گیرد.

دستورالعمل‌های اصلی برای اجرای جلسه‌ی جمع‌آوری خواسته‌ها چیست؟

[1] از این رویکرد گاه به‌عنوان تکنیک تسهیل شده‌ی مشخص‌سازی کاربرد یاد می‌شود.

- یک «سازوکار تعریف» (که می‌تواند کاربرگ، نمودار گردشی یا پوستر دیواری یا حتی تابلو اعلانات دیواری، اتاق چت یا (میزگرد مجازی باشد) به‌کار گرفته می‌شود.

هدف، شناسایی مسأله، پیشنهاد عناصر راهکار، مذاکره بر سر رویکردهای متفاوت و مشخص‌کردن یک مجموعه‌ی مقدماتی از خواسته‌های راهکار در محیطی است که دستیابی به راهکار را دلالت کند. برای درک بهتر جریان رویدادها به‌ترتیبی که رخ می‌دهند، سناریوی مختصری ارائه داده‌ایم که سلسله رویدادهای منتهی به جلسه‌ی جمع‌آوری خواسته‌ها، رویدادهایی را که در حین جلسه و نیز پس از آن رخ می‌دهد، خلاصه می‌کند.

طی مرحله‌ی «دریافت» (بخش ۲-۵) پرسش‌ها و پاسخ‌های پایه، دامنه‌ی مسأله را تعیین می‌کنند و دیدی کلی از راهکار به‌دست می‌دهند. نتیجه‌ی این جلسات اولیه، یک متن یکی دو صفحه‌ای با عنوان «درخواست محصول» است که توسط سازنده و فروشنده به نگارش در می‌آید.

مکان، زمان و تاریخی برای جلسه انتخاب می‌شود؛ فردی به‌عنوان «تسهیل‌گر» برگزیده می‌شود؛ و افرادی برای حضور در جلسه از میان تیم نرم‌افزاری و سایر سازمان‌های ذی‌نفع فراخوانده می‌شوند. قبل از برگزاری جلسه، درخواست محصول بین اعضای جلسه توزیع می‌شود.

به‌عنوان یک مثال،[1] گزیده‌ای از یک درخواست محصول را در نظر بگیرید که مسؤول بازاریابی در پروژه‌ی *SafeHome* آن را نوشته است. این شخص، درباره عملکرد ایمنی در منزل که قرار است بخشی از ***SafeHome*** باشد، چنین می‌نویسد:

پژوهش‌های ما حکایت از آن دارد که بازار سیستم‌های مدیریت منزل سالانه به میزان ۴۰٪ در حال رشد است. نخستین قابلیتی که ***SafeHome*** به بازار عرضه می‌کند، باید قابلیت ایمنی منزل باشد. خیلی‌ها با «سیستم‌های هشداردهی» آشنایی دارند لذا فروش آن نباید کار دشواری باشد.

قابلیت ایمنی منزل، خانه‌ها را در مقابل انواع «شرایط» نامطلوب از قبیل ورود غیر قانونی، آتش سوزی، بالا آمدن آب، افزایش میزان کربن مونوکسید و سایر موارد محافظت می‌کند. این سیستم برای آشکارسازی هر کدام از این شرایط، از حس‌گرهای بی‌سیم استفاده خواهد کرد. صاحب‌خانه می‌تواند آن را برنامه‌ریزی کند و به‌طور خودکار در صورت مشاهده هر کدام از شرایط فوق به یک موجودیت پایش گر(مانند آتش نشانی) تلفن کند.

در واقع، سایر افراد در مطالب نوشته شده در طول جلسه‌ی جمع‌آوری داده‌ها سهم دارند و اطلاعات بسیار بیشتری در دسترس خواهد بود، ولی حتی با اطلاعات اضافی هم ابهام وجود خواهد داشت، احتمال حذف برخی موارد هست و خطاهایی ممکن است رخ دهد. فعلاً، همین توصیف عملیاتی اولیه کفایت می‌کند.

هنگام مرور درخواست محصول در روزهای قبل از برگزاری نشست، از هر کدام از حضار خواسته می‌شود تا از اشیای محیط اطراف سیستم، اشیایی که سیستم باید ایجاد کند و اشیایی که سیستم برای اجرای وظایف خود به‌کار می‌گیرد، فهرستی تهیه کند. به‌علاوه، هر کدام از حضار باید از سرویس‌ها (فرایندها یا وظایفی) که با این اشیا تعامل دارند یا آنها را دستکاری می‌کنند،

[1] در بسیاری از فصل های آینده از این مثال (با بسطها و شکل‌های دیگرش) در روشن‌ساختن روش‌های مهم مهندسی نرم‌افزار استفاده خواهد شد. به‌عنوان تمرین، خوب است خودتان جلسه‌ای برای جمع‌آوری خواسته‌ها تشکیل دهید و مجموعه‌ای از فهرست‌ها را برای آنها تهیه کنید.

«ما زمان فراوانی را- یعنی بخش اعظم تلاش‌های پروژه- نه صرف پیاده‌سازی یا آزمون، بلکه صرف تصمیم‌گیری برای چیزی که باید ساخته شود، می‌کنیم.»

برایان لاورنس

مرجع وب

توسعه الحاقی کاربردها (JAD) تکنیکی پر طرف‌دار برای جمع‌آوری خواسته‌هاست. توصیف خوبی از این تکنیک را می‌توانید در وب‌سایت زیر بیابید:
www.carolla.com/wp.jad.htm

اندرز

اگر سیستم یا محصولی به کاربران بسیار سرویس دهد، مطلقاً یقین داشته باشید که خواسته‌ها از نمونه‌ای از کاربران استخراج می‌شوند. اگر تنها یک کاربر همه‌ی خواسته‌ها را تعریف کند، ریسک پذیرش بالا می‌رود.

فهرستی جداگانه ارائه دهد. سرانجام، فهرست‌هایی از قیدوبندها (مثلاً هزینه، اندازه، قوانین تجاری) و ملاک‌های کارایی (مثلاً سرعت و صحت) نیز باید تهیه شود. به اطلاع حضار رسانده می‌شود که ضرورتی ندارد این فهرست‌ها خیلی جامع و فراگیر باشد، بلکه کافی است انعکاس دهنده‌ی ادراک و برداشت فرد از سیستم باشد.

اشیای توصیف شده برای ***SafeHome*** ممکن است شامل پانل کنترل، آشکارسازهای دود، حس‌گرهای در و پنجره، آشکار سازهای حرکت، آژیر هشدار، رویداد (فعال‌شدن یک حس‌گر)، صفحه نمایش، کامپیوتر، شماره تلفن‌ها، تماس تلفنی و غیره باشند. فهرست سرویس‌ها ممکن است شامل پیکربندی سیستم، *تعیین وضعیت آژیر*، *پایش* حس‌گرها، *گرفتن* شماره تلفن، برنامه‌ریزی پانل کنترل و *خواندن* صفحه نمایش شود (توجه دارید که سرویس‌ها روی اشیا کاری انجام می‌دهند). به‌شیوه‌ای مشابه، هر کدام از حضار، فهرستی از قیدوبندها (مثلاً سیستم باید در صورت عمل نکردن حس‌گرها این را تشخیص دهد، باید استفاده از آن آسان باشد، باید مستقیماً به یک خط تلفن استاندارد قابل اتصال باشد) و ملاک‌های کارایی (مثلاً یک رویداد حس‌گر باید در عرض یک ثانیه تشخیص داده شود و یک الگوی اولویت برای رویدادها باید پیاده‌سازی شود) نیز تهیه خواهند کرد.

«حقایق با نادیده انگاشتن آنها از وجود ساقط نمی‌شوند.»

آندوس هاکسلی

اندرز

پرهیزید از اینکه به یک یازه به مشتری بگویید ایده‌اش «زیادی هزینه‌بر» است یا «غیر عملی» است. قرار بر این است که روی فهرستی مذاکره شود که قابل قبول همه باشد. برای این منظور، باید ذهنی باز داشته باشید.

فهرست اشیا را می‌توان با استفاده از برگه‌های بزرگ کاغذ یا کاغذهای با پشت چسب‌دار به دیوار زد یا روی تخته سفیدهای دیوار نوشت. به طریق دیگر، این فهرست‌ها را می‌توان در یک بولتن الکترونیکی در معرض دید دیگران قرار داد یا در محیط اتاق چت، قبل از برگزاری جلسه به دیگران عرضه کرد. در حالت ایده‌آل، هر درایه (entry) فهرست‌شده باید قابلیت دستکاری جداگانه را داشته باشد، به‌طوری که فهرست‌ها را بتوان در هم آمیخت، درایه‌ها را بتوان اصلاح کرد و مواردی را به آنها افزود. در این مرحله، نقد و مناظره اکیداً ممنوع است.

پس از ارائه‌ی تک‌تک فهرست‌ها در یک مبحث مشترک، گروه با حذف درایه‌های زائد، و افزودن ایده‌های جدیدی که در طول بحث مطرح می‌شوند یک فهرست تلفیقی ایجاد می‌کند. ولی چیزی را حذف نمی‌کند. پس از آن که فهرست‌های تلفیقی را برای همه‌ی مباحث تهیه کردید، بحث- با هماهنگی تسهیل‌گر- بلافاصله آغاز خواهد شد. فهرست تلفیق شده کوتاه، بلند یا بازنویسی می‌شود تا محصول/ سیستم مورد نظر را به‌خوبی منعکس سازد. هدف، توسعه فهرستی از اشیا، سرویس‌ها، قیدوبندها و کارایی برای سیستم است که مورد توافق همگان قرار گرفته باشد.

در موارد بسیار، یک شیء یا سرویس توصیف شده در یک فهرست، به توضیح بیشتر نیاز دارد. برای این منظور، طرف‌های ذی‌نفع یک سری ریزمشخصات برای درایه‌های هر فهرست توسعه می‌دهند.[1] هر مشخصه‌ی کوچک، جزئیاتی از یک شیء یا سرویس است. برای مثال، مشخصات کوچک برای شیء *پانل کنترل* در ***SafeHome*** می‌تواند به شرح زیر باشد:

پانل کنترل، یک واحد قابل نصب روی دیوار با ابعاد حدودی ۲۰ در ۱۲ سانتی‌متر است. پانل کنترل از طریق بی‌سیم به حس‌گرها و کامپیوتر شخصی متصل است. تعامل با کاربر از طریق یک صفحه کلید حاوی ۱۲ کلید صورت می‌گیرد. یک صفحه نمایش LCD رنگی به ابعاد ۷ در ۷ سانتی‌متر، اطلاعات را در اختیار کاربر قرار می‌دهد. نرم‌افزار پیام‌ها، اکو و سایر عملکردهای مشابه را فراهم می‌سازد.

[1] بسیاری از تیم‌های نرم‌افزاری به‌جای ایجاد یک سری ریزمشخصات، سناریوهای کاربری موسوم به *use case* تهیه می‌کند که درباره آن در بخش ۴-۵ و فصل ۶ به تفصیل سخن خواهیم گفت.

SafeHome

برگزاری جلسه جمع‌آوری خواسته‌ها

صحنه: اتاق کنفرانس. اولین جلسه جمع‌آوری خواسته‌ها در حال برگزاری است.

نقش‌آفرینان: جیمی لازار، عضو تیم نرم‌افزاری؛ وینود رامان، عضو تیم نرم‌افزار؛ اد رابینز، عضو تیم نرم‌افزار؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ سه عضو بازاریابی؛ نماینده‌ای از مهندسی تولید؛ و یک نفر تسهیل‌گر.

گفتگوها:

تسهیل‌گر (در حالی که به تخته سفید اشاره می‌کند): خلاصه، این، فهرست فعلی اشیا و سرویس‌های مربوط به عملکرد ایمنی منزل است.

عضو بازاریابی: که تقریباً دیدگاه‌های ما را پوشش می‌دهد.

وینود: کسی به این نکته اشاره نکرد که می‌خواهند همه‌ی عملکردهای *SafeHome* از طریق اینترنت قابل دستیابی باشد؟ این شامل قابلیت ایمنی منزل هم می‌شود، نه؟

عضو بازاریابی: بله، درست است... باید این عملکرد و اشیای مناسب را هم اضافه کنیم.

تسهیل‌گر: این به قیدوبندها هم چیزی اضافه می‌کند؟

جیمی: بله، هم فنی و هم قانونی.

نماینده تولید: چطور؟

جیمی: بهتر است اطمینان پیدا کنیم که یک غریبه نمی‌تواند در سیستم نفوذ کند، آن را از کار بیندازد و از محل دزدی کند یا حتی بدتر. اگر این اتفاق بیفتد حسابی شرمنده خواهیم شد.

داگ: کاملاً درست است.

عضو بازاریابی: به‌هرحال به این قابلیت نیاز داریم... پس فکری بکنید که مانع از ورود این غریبه به سیستم بشوید.

اد: گفتنش آسان است و...

تسهیل‌گر (حرفش را قطع می‌کند): الآن نمی‌خواهم درباره این مشکل بحث بشود. حالا آن را به‌عنوان یک مورد در فهرست مشکلات یادداشت می‌کنیم و ادامه می‌دهیم.

(داگ که به‌عنوان منشی جلسه عمل می‌کند، این نکته را یادداشت می‌کند.)

تسهیل‌گر: احساس می‌کنم هنوز مطالب بیشتری هست که باید در نظر گرفته شود.

(گروه، بیست دقیقه بعدی را صرف پالایش و توسعه‌ی جزئیات قابلیت ایمنی منزل می‌کند.)

مشخصات کوچک به همه‌ی طرف‌های ذی‌نفع ارائه می‌شود تا درباره آن بحث کنند. حذفیات و اضافات انجام می‌شود و جزئیات بیشتری تعیین می‌شود. در برخی موارد، توسعه‌ی مشخصات کوچک باعث پدیدارشدن اشیا، سرویس‌ها، قیدوبندها یا خواسته‌های کارایی کوچکی می‌شود که به فهرست‌های اولیه افزوده خواهند شد. طی همه‌ی بحث‌ها، تیم ممکن است با مشکلی مواجه شود که طی جلسه قابل حل نباشد. باید فهرستی از مشکلات تهیه شود تا به این ایده‌ها بعداً پرداخته شود.

۱. آیا تغییر در SCI «برجسته» شده است؟ آیا داده‌های تغییر و صاحب تغییر مشخص شده است؟ آیا صفات شیء پیکربندی، این تغییر را منعکس می‌کنند؟

۲. آیا روال‌های SCM برای ذکر تغییر، ثبت آن و گزارش آن دنبال شده‌اند؟

۳. آیا همه‌ی SCIهای مربوط به‌طور مناسب بهنگام‌سازی شده‌اند؟

۵-۳-۲ استقرار عملکرد کیفیت (Quality Function Deployment)

استقرار عملکرد کیفیت (QFD) تکنیک تضمین کیفیتی است که نیازهای مشتری را به خواسته‌های فنی برای نرم‌افزار ترجمه می‌کند. QFD در آغاز در ژاپن توسعه یافت و اولین بار در کشتی‌سازی صنایع سنگین میتسوبیشی در اوایل دهه ۱۹۷۰ مورد استفاده قرار گرفت. این روش، بر به حداکثر رساندن رضایت مشتری از فرایند مهندسی نرم‌افزار تأکید دارد [Zul92]. QFD برای نیل به این هدف، بر شناخت چیزهایی که نزد مشتری باارزش هستند تأکید کرده سپس این ارزش‌ها را از طریق فرایند مهندسی مستقر می‌سازد. QFD سه نوع خواسته را مشخص می‌کند [Zul92]:

خواسته‌های عادی. اهداف و مقاصدی که طی جلسه با مشتری برای محصول یا سیستم بیان می‌شوند. اگر این خواسته‌ها موجود باشند، مشتری راضی خواهد بود. مثال‌هایی از خواسته‌های عادی عبارتنداز انواع صفحه‌نمایش‌های گرافیکی، عملکردهای سیستمی خاص و سطوح مشخصی از کارایی.

خواسته‌های موردانتظار. این خواسته‌ها برای سیستم یا محصول ضروری هستند، ولی ذکری از آنها به میان نمی‌آید و ممکن است چنان بنیادی باشند که مشتری آنها را به وضوح بیان نکند. نبود آنها، نارضایتی را به‌دنبال خواهد داشت. مثال‌هایی از خواسته‌های موردانتظار عبارتنداز: سهولت تعامل انسان با ماشین، درستی و قابلیت اعتماد عملیاتی کلی و سهولت نصب نرم‌افزار.

خواسته‌های مهیج. این ویژگی‌ها ورای انتظارات مشتری بوده در صورت وجود، رضایت مشتری را بسیار بالا می‌برند. برای مثال، نرم‌افزار مربوط به یک تلفن همراه با ویژگی‌های استانداردی ارائه می‌شود، ولی اگر با قابلیت‌های غیر منتظره همراه شود (مثلاً صفحات لمسی، پست صوتی تصویری) هر کاربر محصول را خوشنود می‌سازد.

گرچه مفاهیم QFD را می‌توان در کل فرایند نرم‌افزار به‌کار برد [Par96a]، تکنیک‌های خاصی از QFDدر فعالیت استخراج خواسته‌ها کاربرد دارند. QFD از مصاحبه با مشتریان و مشاهده، نظر خواهی و بررسی داده‌های تاریخی (گزارش‌های مسأله)، به‌عنوان داده‌های خام برای جمع‌آوری خواسته‌ها استفاده می‌کند. این داده‌ها سپس به جدولی از خواسته‌ها ترجمه می‌شوند- که *جدول صدای مشتری*[1] نامیده می‌شود؛ این جدول توسط مشتری و سایر طرف‌های ذی‌نفع مرور و بازبینی می‌شود. سپس انواع نمودارها، معیارها و روش‌های ارزیابی برای استخراج خواسته‌های مورد انتظار و خواسته‌های مهیج به‌کار گرفته می‌شود.

۵-۳-۳ سناریوهای کاربرد (Use Scenarios)

به موازات جمع‌آوری خواسته‌ها، چشم اندازی کلی از عملکردها و ویژگی‌های سیستم شروع به شکل‌گیری می‌کند، ولی تا زمانی که چگونگی به‌کارگیری این قابلیت‌ها و ویژگی‌ها توسط دسته‌های متفاوت کاربران نهایی را درک نکرده باشید، حرکت به سوی فعالیت‌های فنی‌تر مهندسی نرم‌افزار دشوار خواهد بود. برای نیل به این مقصود، سازندگان و کاربران می‌توانند یک مجموعه سناریو ایجاد کنند که برای سیستم مورد نظر رشته‌ای از کاربرد را تعیین کند. این سناریوها که غالباً از آنها به‌عنوان use case یاد می‌شود [Jac92] توصیفی از چگونگی به‌کارگیری سیستم فراهم می‌آورند. use case را با جزئیات بیشتر در بخش ۵-۴ به بحث خواهیم گذاشت.

---

[1] Customer Voice Table

نکته‌ی کلیدی

QFD خواسته‌ها را به‌شیوه‌ای تعیین می‌کند که رضایت مشتری به حداکثر برسد.

هشدار

همه می‌خواهند خواسته‌های مهیج فراوان پیاده‌سازی شود، ولی مراقب باشید. «خزش خواسته‌ها» این گونه آغاز می‌شود. از طرف دیگر، خواسته‌های مهیج به محصولی استثنایی منجر می‌شوند.

مرجع وب

اطلاعات مفیدی درباره QFD را می‌توانید در وب‌سایت زیر مشاهده کنید:
www.qfdi.org

SafeHome

توسعه‌ی یک سناریوی کاربرد مقدماتی

صحنه: اتاق کنفرانس، ادامه اولین جلسه‌ی جمع‌آوری خواسته‌ها.

نقش‌آفرینان: جیمی لازار، عضو تیم نرم‌افزار؛ وینود رامان، عضو تیم نرم‌افزار؛ اد رابینز، عضو تیم نرم‌افزار؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ سه عضو بازاریابی؛ نماینده‌ای از مهندسی تولید؛ یک تسهیل‌گر

گفتگوها:

تسهیل‌گر: ما درباره امنیت مربوط به قابلیت دستیابی اینترنتی به ***SafeHome*** صحبت کردیم. من می‌خواهم یک چیز را امتحان کنم. بیایید برای دستیابی به قابلیت ایمنی منزل یک سناریوی کاربرد بنویسیم.

جیمی: چطوری؟

تسهیل‌گر: این کار را می‌توانیم به چند روش متفاوت انجام بدهیم، ولی فعلاً می‌خواهم همه چیز واقعاً غیر رسمی باقی بماند. بگو ببینم (با اشاره به عضو بازاریابی) چه شیوه‌ای را برای دستیابی به سیستم در نظر داری؟

عضو بازاریابی: خب... در مورد من، وقتی از خانه دور هستم و مجبورم کسی را به خانه راه بدهم، مثلاً یک تعمیرکار یا نظافت‌چی، که کد امنیتی را ندارد.

تسهیل‌گر (با لبخند): این که گفتی، دلیل‌اش است... به من بگو این کار را چطور باید انجام داد.

عضو بازاریابی: [... اولین چیزی که لازم دارم، یک کامپیوتر شخصی است. وارد وب‌سایتی می‌شوم که برای همه‌ی کاربران ***SafeHome*** در نظر گرفته شده است. شناسه کاربری و ...

وینود (حرفش را قطع می‌کند): این صفحه وب باید ایمنی داشته باشد و پنهان‌سازی شده باشد تا بتوانیم مطمئن شویم که امنیت...

تسهیل‌گر (حرفش را قطع می‌کند): اینها اطلاعات خوبی است وینود، ولی این یک مسأله فنی است. فعلاً بگذار فقط به نحوه استفاده‌ی کاربر نهایی از این قابلیت توجه کنیم. خب؟

وینود: حتماً.

عضو بازاریابی: خلاصه، همان طور که گفتم با دادن شناسه کاربری و دو سطح از کلمات عبور وارد وب‌سایت می‌شوم.

جیمی: و اگر کلمه‌ی عبور را فراموش کردم؟

تسهیل‌گر (حرفش را قطع می‌کند): نکته خوبی بود جیمی، ولی فعلاً به آن کاری نداشته باشیم. این نکته را یادداشت می‌کنیم و آن را یک *استثنا* می‌نامیم. حتم دارم موارد دیگری هم هست.

عضو بازاریابی: بعد از واردکردن کلمات عبور، صفحه‌ای ظاهر می‌شود که همه‌ی عملکردهای ***SafeHome*** را به نمایش می‌گذارد. من گزینه مربوط به «ایمنی منزل» را انتخاب می‌کنم. سیستم ممکن است از من درخواست کند که هویت خودم را به اثبات برسانم. مثلاً با پرسیدن آدرس یا شماره تلفن. بعد تصویری از پانل کنترل سیستم ایمنی را به نمایش در می‌آورد که در

کنار آن فهرستی از وظایف قابل انجام به چشم می‌خورد؛ مثل راه‌انداختن سیستم، از کار انداختن آن، از کار انداختن گزینشی یک یا چند حس‌گر. من تصور می‌کنم حتی می‌تواند امکان پیکربندی دوباره‌ی نواحی امنیتی را هم فراهم کند و خیلی کارهای دیگر که مطمئن نیستم.

(در همان حال که عضو بازاریابی مشغول صحبت است، داگ یادداشت زیادی بر می‌دارد؛ این یادداشت‌ها مبنایی برای اولین سناریوی کاربرد غیر رسمی محسوب می‌شود. به‌طریق دیگر، از عضو بازاریابی خواسته می‌شد که سناریو را بنویسد، ولی این در خارج از جلسه صورت می‌پذیرفت.)

۵-۳-۶ محصولات کاری استخراج (Elicitation Work Product)

محصولات کاری تولید شده به‌عنوان نتیجه‌ای از استخراج خواسته‌ها بسته به اندازه‌ی سیستم یا محصولی که قرار است ساخته شود، متغیر است. برای اکثر سیستم‌ها، محصولات کاری عبارتند از:

- بیان نیازها و امکان‌سنجی
- بیان محدود حوزه‌ی مربوط به سیستم یا محصول
- فهرستی از مشتریان، کاربران و سایر طرف‌های ذی‌نفع که در استخراج خواسته‌ها مشارکت داشته‌اند.
- توصیفی از محیط فنی سیستم.
- فهرستی از خواسته‌ها (که ترجیحاً بر اساس عملکرد، سازمان‌دهی شده‌اند) و قیدوبندهایی دامنه که در مورد هر کدام کاربرد دارند.
- مجموعه‌ای از سناریوهای کاربرد که دیدی از کاربرد سیستم یا محصول، تحت شرایط عملیاتی متفاوت به‌دست می‌دهند.
- هر نمونه‌ی اولیه‌ای که برای تعریف بهتر خواسته‌ها توسعه یافته باشد.

در نتیجه‌ی جمع‌آوری خواسته‌ها چه اطلاعاتی ایجاد می‌شود؟

هر یک از این محصولات کاری، مورد بازبینی تمامی افراد شرکت کننده در استخراج خواسته‌ها قرار می‌گیرد.

۵-۴ توسعه‌ی use case

آلیستر کاکبرن [Coc01b] در کتابی که چگونگی نوشتن use case اثربخش را شرح می‌دهد، چنین می‌نویسد که «مورد استفاده، قراردادی است ...[که] رفتار سیستم را تحت شرایط گوناگون در پاسخ گیری به درخواستی از سوی یکی از طرف‌های ذی‌نفع توصیف می‌کند...» در اصل، use case داستانی است با سبک و سیاق درباره چگونگی تعامل کاربر نهایی (که یکی از چند نقش ممکن را بر عهده دارد) با سیستم تحت مجموعه شرایط معین. این داستان می‌تواند متنی حکایتی، خلاصه‌ای از وظایف و تعامل‌ها، توصیفی مبتنی بر قالب، یا نمایشی نمودار گونه باشد. شکل آن هر چه که باشد، تصویرگر سیستم یا نرم‌افزار از دیدگاه کاربر نهایی است.

گام نخست در نوشتن یک use case، تعریف مجموعه‌ای از «کنش‌گران[1]» است که در داستان مشارکت

[1] actors

دارند. *کنش‌گران*، افراد (یا دستگاه های) متفاوتی هستند که از سیستم یا محصول، در حیطه‌ی عملکرد یا رفتاری که قرار است توصیف شود، استفاده می‌کنند. کنش‌گران نشان‌گر نقش‌هایی هستند که افراد (یا دستگاه‌ها) در حین کار سیستم، عهده‌دار آن می‌شوند. «کنش‌گر» بنا به تعریفی رسمی تر، هر چیزی است که با سیستم یا محصول ارتباط برقرار می‌کند و خارج از خود سیستم قرار دارد. هر کنش‌گر هنگام استفاده از سیستم، یک یا چند هدف دارد.

شایان ذکر است که کنش‌گر و کاربر نهایی الزاماً یکسان نیستند. یک کاربر معمولی ممکن است هنگام استفاده از سیستم چند نقش را بر عهده بگیرد، در حالی که کنش‌گر نماینده‌ی طبقه‌ای از موجودیت‌های خارجی (غالباً، ولی نه همیشه، آدم هایی) است که فقط یک نقش در حیطه‌ی use case دارند. به‌عنوان مثال، اپراتور (یا کاربر) ماشینی را در نظر بگیرید که با کامپیوتر کنترل‌کننده برای یک سلول تولیدی تعامل دارد؛ این سلول حاوی چند روبات و ماشین‌هایی است که به‌صورت عددی کنترل می‌شوند. پس از مرور دقیق خواسته‌ها، نرم‌افزار مربوط به کامپیوتر کنترل کننده نیاز به چهار حالت (یا نقش) متفاوت برای تعامل دارد. حالت برنامه‌ریزی، حالت آزمون، حالت پایش و حالت اشکال‌زدایی. در برخی موارد، اپراتور ماشین، می‌تواند همه‌ی این نقش‌ها را عهده‌دار شود. در سایر موارد، افراد متفاوت ممکن است نقش هر کنش‌گر را بر عهده داشته باشند.

از آنجا که استخراج خواسته‌ها یک فعالیت تکاملی است، همه‌ی کنش‌گرها در اولین دور تکرار شناسایی نمی‌شوند. شناسایی کنش‌گرهای نوع اول، طی نخستین دور تکرار امکان‌پذیر است [Jac92] و کنش‌گرهای نوع دوم را با کسب اطلاعات بیشتر درباره‌ی سیستم می‌توان شناسایی کرد. کنش‌گرهای نوع اول با هم تعامل می‌کنند تا عملکرد لازم برای سیستم حاصل شود و مزایای مورد نظر از آن به‌دست آید. آنها به‌طور مستقیم و غالباً با نرم‌افزار کار می‌کنند. کنش‌گرهای نوع دوم، سیستم را پشتیبانی می‌کنند، به‌طوری که کنش‌گرهای نوع اول بتوانند کار خود را انجام دهند.

هنگامی که کنش‌گرها شناسایی شدند، use case را می‌توان توسعه داد. جیکابسون [Jac92] چند پرسش مطرح می‌کند[1] که در یک use case به آنها پاسخ گفته شود:

- کنش‌گر نوع اول و کنش‌گر نوع دوم چه کسانی هستند؟
- اهداف کنش‌گر چیست؟
- قبل از شروع داستان چه پیش‌شرط‌هایی باید وجود داشته باشد؟
- چه وظایف اصلی توسط کنش‌گر اجرا می‌شود؟
- چه استثناهایی را باید با توصیف داستان در نظر داشت؟
- چه تغییراتی در تعامل کنش‌گران امکان‌پذیر است؟
- کنش‌گر چه اطلاعاتی را به‌دست می‌آورد، تولید می‌کند یا تغییر می‌دهد؟
- آیا کنش‌گر باید سیستم را از تغییرات به‌عمل‌آمده در محیط خارجی آگاه سازد؟
- کنش‌گر چه اطلاعاتی را از سیستم می‌خواهد؟
- آیا کنش‌گر می‌خواهد درباره تغییرات غیر منتظره مطلع شود؟

[1] پرسش‌های جیکابسون بسط و توسعه یافته‌اند تا نمای کامل‌تری از محتویات مورد کاربرد فراهم گردد.

نکته‌ی کلیدی

use case از دیدگاه یک کنش‌گر تعریف می‌شوند. کنش‌گری نقشی است که افراد (کاربران) یا دستگاه‌ها در تعامل با نرم‌افزار بر عهده دارند.

مرجع وب

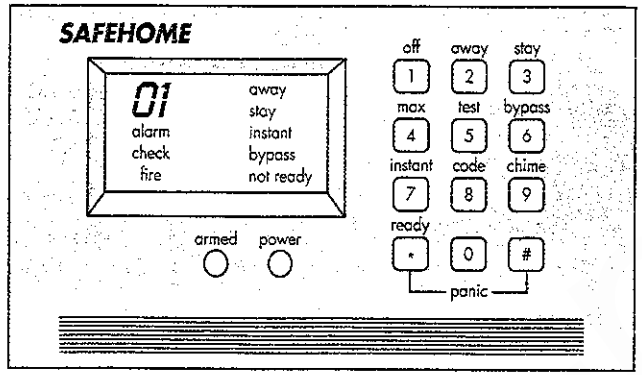
مقاله‌ای عالی درباره use case را می‌توان از آدرس زیر دانلود کرد.

www.ibm.com/developerworks/webservices/library/codesign7.html

برای توسعه‌ی یک use case موثر چه باید بدانم؟

با به‌خاطر آوردن خواسته‌های اساسی ***SafeHome*** چهار کنش‌گر را می‌توانیم تعیین کنیم: **صاحب‌خانه** (کاربر)، **مدیر راه‌اندازی** (احتمالاً همان صاحب‌خانه، ولی با نقش متفاوت)، **حسگرها** (دستگاه‌های متصل به سیستم) و **زیرسیستم پایش و پاسخ** (ایستگاه مرکزی که عملکرد ایمنی منزل در محصول ***SafeHome*** را پایش می‌کند). برای اهداف این مثال، فقط کنش‌گر اول یعنی **صاحب‌خانه** را در نظر می‌گیریم. کنش‌گر **صاحب‌خانه** به چند شیوه‌ی متفاوت با استفاده از پانل کنترل آژیر یا کامپیوتر شخصی با قابلیت ایمنی منزل تعامل دارد:

- کلمه‌ی عبوری را وارد می‌کند تا همه‌ی تعامل‌های دیگر امکان‌پذیر گردد.
- درباره وضعیت یک ناحیه امنیتی پرسش می‌کند.
- درباره وضعیت یک حس‌گر پرسش می‌کند.
- در یک وضعیت اضطراری، دکمه‌ی لازم را فشار می‌دهد.
- سیستم امنیتی را فعال/غیرفعال می‌کند.



شکل ۱-۵ پانل کنترل *SafeHome*

با در نظرگرفتن شرایطی که در آن، صاحب‌خانه از پانل کنترل استفاده می‌کند، use case پایه برای فعال‌سازی سیستم، به شرح زیر خواهد بود:[1]

۱. صاحب‌خانه پانل کنترل ***SafeHome*** را مشاهده می‌کند (شکل ۱-۵) تا تعیین شود که آیا سیستم برای ورودی آماده هست یا خیر. اگر سیستم آماده نباشد، یک پیام «not ready» روی صفحه LCD به نمایش در خواهد آمد و صاحب‌خانه باید به‌صورت فیزیکی درها یا پنجره‌ها را ببندد. تا پیام «not ready» ناپدید شود [پیام «not ready» بدان معناست که حس‌گری باز است یعنی در یا پنجره‌ای باز است.]

[1] توجه دارید که این مورد کاربرد با وضعیتی که دستیابی به سیستم از طریق اینترنت انجام می‌شود، تفاوت دارد. در این مورد، تعامل از طریق پانل کنترلی رخ می‌دهد نه از طریق یک واسط کاربری گرافیکی (GUI) که هنگام استفاده از PC ارائه می‌شود.

۲. صاحب‌خانه از صفحه کلید برای واردکردن یک کلمه‌ی عبور چهار رقمی استفاده می‌کند. اگر کلمه‌ی عبور درست نباشد، پانل کنترل یک بار بوق می‌زند و خود را برای ورودی بعدی آماده می‌کند. اگر کلمه عبور درست باشد، پانل کنترل منتظر عملیات دیگر می‌ماند.

۳. صاحب‌خانه با استفاده از صفحه کلید یکی از حالت‌های stay یا away را برای فعال‌کردن سیستم انتخاب می‌کند (شکل ۲-۵). stay فقط حس‌گرهای محیطی را فعال می‌کند (حس‌گرهای حرکتی داخلی غیر فعال باقی می‌مانند). در حالت away تمامی حس‌گرها فعال می‌شوند.

۴. پس از فعال شدن، صاحب‌خانه یک نور قرمز مشاهده می‌کند.

این use case پایه، نشان‌گر یک داستان سطح بالاست که تعامل میان کنش‌گر و سیستم را توصیف می‌کند.

در بسیاری از موارد، use case باز هم تجزیه و پالایش می‌شود تا جزئیات بیشتری درباره تعامل فراهم گردد. برای مثال، کاک برن [Coc 01b] الگوی زیر را برای توصیف مشروح use case پیشنهاد کرده است:

use case: *Initiate Monitoring* (راه اندازی پایش)

کنش‌گر نوع اول: صاحب‌خانه

هدف در حیطه: تنظیم سیستم برای پایش حس‌گرها هنگامی که صاحب‌خانه منزل را ترک می‌کند یا در داخل خانه می‌ماند.

پیش‌شرط‌ها: سیستم برای دریافت کلمه‌ی عبور و شناسایی حس‌گرهای گوناگون برنامه‌ریزی شده است.

راه‌انداز: صاحب‌خانه تصمیم می‌گیرد که سیستم را «روشن کند» یعنی قابلیت‌های آژیر را فعال کند.

سناریو:

۱. صاحب‌خانه: پانل کنترل را مشاهده می‌کند.

۲. صاحب‌خانه: کلمه‌ی عبور را وارد می‌کند.

۳. صاحب‌خانه: «stay» یا «away» را انتخاب می‌کند.

۴. صاحب‌خانه: چراغ آژیر را مشاهده می‌کند که نشان می‌دهد ***SafeHome*** فعال شده است.

استثناها:

۱. پانل کنترل آماده نیست (not ready): صاحب‌خانه همه‌ی حس‌گرها را چک می‌کند تا تعیین کند کدام یک از آنها باز است؛ آنها را که باز هستند، می‌بندد.

۲. کلمه‌ی عبور نادرست است (پانل کنترل یک بار سوت می‌زند): صاحب‌خانه این بار کلمه‌ی عبوری درست را وارد می‌کند.

۳. کلمه‌ی عبور شناخته نمی‌شود: باید با زیر سیستم پایش و پاسخ تماس گرفته شود تا کلمه‌ی عبور دوباره تعیین شود.

اندرز

use case غالباً به زبانی غیر رسمی نوشته می‌شوند. به‌هرحال برای حصول اطمینان از اینکه همه‌ی مسائل کلیدی پوشش داده می‌شوند، از این قالب استفاده کنید.

۴. حالت «stay» انتخاب می‌شود: پانل کنترل دو بار سوت می‌زند و چراغ stay روشن می‌شود؛ حس‌گرهای محیطی فعال می‌شوند.

۵. حالت «away» انتخاب می‌شود: پانل کنترل سه بار سوت می‌زند و چراغ away روشن می‌شود؛ همه‌ی حس‌گرها فعال می‌شوند.

اولویت: ضروری؛ باید پیاده‌سازی شود.

چه هنگام در دسترس قرار گیرد: در اولین نسخه

فراوانی کاربرد: چندین بار در روز

کانال ارتباطی با کنش‌گر: از طریق واسط پانل کنترل

کنش‌گرهای نوع دوم: تکنسین پشتیبانی، حس‌گرها

کانال ارتباط با کنش‌گرهای نوع دوم:

تکنسین پشتیبانی: خط تلفن

حس‌گرها: واسط‌های فرکانس رادیویی و سیم کشی

مشکلات باز:

۱. آیا باید راهی برای فعال‌کردن سیستم بدون استفاده از کلمه‌ی عبور یا با یک کلمه‌ی عبوری مختصر وجود داشته باشد؟

۲. آیا باید پانل کنترل پیام‌های متنی دیگری برای نمایش دادن داشته باشد؟

۳. صاحب‌خانه از زمان فشار دادن اولین کلید چقدر زمان برای واردکردن کلمه‌ی عبور دارد؟

۴. آیا راهی وجود دارد که سیستم را پیش از آنکه واقعاً فعال شود، غیر فعال کرد؟

use case برای سایر تعامل‌های **صاحب‌خانه** نیز به‌شیوه‌ای مشابه توسعه خواهد یافت. مرور دقیق هر use case اهمیت دارد. اگر عنصری از تعامل مبهم باشد، این احتمال هست که با مرور use case، مشکلی مشخص گردد.

ابزارهای نرم‌افزاری

توسعه‌ی use case

هدف: کمک به توسعه‌ی use case با فراهم‌ساختن قالب‌های خودکار و سازوکارهایی جهت ارزیابی وضوح و سازگاری.

سازوکارها: مکانیک ابزارها با هم تفاوت دارد. به‌طورکلی، ابزارهای مربوط به use case قالب‌هایی به‌شکل فرم‌های پرکردن جای خالی فراهم می‌آورند که نتیجه آنها ایجاد use case اثربخش است. اکثر عملکرد یک use case در مجموعه وسیع‌تری ازعملکردهای مهندسی‌خواسته‌ها ادغام می‌شود.

ابزارهای نمونه

اکثر ابزارهای مدل‌سازی تحلیل مبتنی بر UML هر دو نوع پشتیبانی متنی و گرافیکی را برای توسعه و مدل‌سازی use case فراهم می‌سازند.

وب‌سایت Objects by Design حاوی پیوندهایی جامع به این گونه ابزارهاست:

www.objectbydesign.com/tools/umltools-bycompany.html

SafeHome

توسعه‌ی نمودار سطح بالا برای یک use case

صحنه: اتاق کنفرانس، ادامه‌ی جلسه جمع‌آوری خواسته‌ها

نقش‌آفرینان: جیمی لازار، عضو تیم نرم‌افزار؛ وینود رامان، عضو تیم نرم‌افزار؛ اد رابینز، عضو تیم نرم‌افزار؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ سه عضو بازاریابی؛ نماینده‌ای از مهندسی تولید؛ یک تسهیل‌گر

گفتگوها:

تسهیل‌گر: ما وقت خوبی صرف صحبت درباره عملکرد ایمنی منزل در *SafeHome* کردیم. در طول مدت استراحت، من یک نمودار use case رسم کردم تا سناریوهای مهمی را که بخشی از این قابلیت هستند، خلاصه کنم. یک نگاهی بیندازید.

(همه‌ی حاضران به نمودار شکل ۲-۵ نگاه می‌کنند).

جیمی: من تازه دارم نمادگذاری UML را یاد می‌گیرم. عملکرد ایمنی منزل با یک چهارگوش بزرگ و بیضی‌های داخلش مشخص می‌شود؟ و بیضی‌ها هر کدام نشان دهنده یکی از use case‌ی هستند که آنها را به‌صورت متنی نوشتیم؟

تسهیل‌گر: آره. و آن آدمک‌ها کنش‌گرها را نشان می‌دهند- آدم‌ها یا چیزهایی که به‌صورت توصیف‌شده در use case با سیستم تعامل دارند... در ضمن من برای کنش‌گرهای غیر انسانی از چهارگوش‌های نشان‌دار استفاده کردم که اینجا حس‌گرها هستند.

داگ: این در UML قانونی است؟

تسهیل‌گر: قانونی‌بودن، مسأله‌ای ایجاد نمی‌کند. چیزی که مهم است، برقراری ارتباط است. من استفاده از آدمک‌ها برای نمایش دادن یک دستگاه را گمراه کننده می‌دانم. به همین خاطر هم قدری چیزها را تغییر دادم و تصور نمی‌کنم مشکلی ایجاد کند.

وینود: بسیار خوب. پس ما برای هر کدام از بیضی‌ها شرحی از یک use case داریم. آیا باید شرح‌های مبتنی بر الگویی را که خودم درباره آنها مطالعه کرده‌ام، توسعه بدهیم؟

تسهیل‌گر: احتمالاً، ولی این را می‌توانیم به زمانی موکول کنیم که به عملکردهای دیگر *SafeHome* هم رسیدگی کرده باشیم.

عضو بازاریابی: صبر کنید، من داشتم به این نمودار نگاه می‌کردم و یک دفعه دیدم که چیزی را فراموش کرده‌ایم.

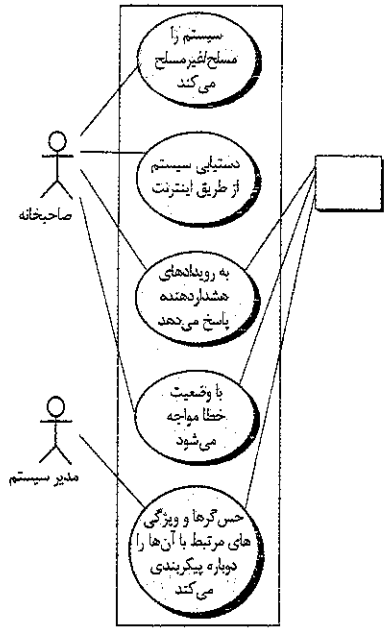
تسهیل‌گر: جدی؟ بگو ببینیم چه چیزی فراموش شده؟

(این جلسه ادامه دارد).

۵-۵ ساخت مدل‌های خواسته‌ها[1]

هدف از این مدل تحلیل، فراهم ساختن توصیفی از دامنه‌های اطلاعاتی، عملیاتی و رفتاری مورد نیاز برای سیستم کامپیوتری است. همچنان که مطالب بیشتری درباره سیستم مورد نظر می‌آموزید و سایر

[1] در سرتاسر این کتاب از دو عبارت *مدل تحلیل* و *مدل خواسته‌ها* به‌عنوان مفاهیمی مترادف استفاده خواهیم کرد. هر دو عبارت به نمایش دامنه‌های اطلاعاتی، عملیاتی و رفتاری توصیف‌کننده‌ی خواسته‌های مسأله اشاره دارند.



شکل ۲-۵ نمودار UML برای use case عملکرد ایمنی منزل در *SafeHome*

طرف‌های ذی‌نفع بیشتر در می‌یابند که واقعاً چه نیازهایی دارند، این مدل تغییر می‌کند. به این دلیل، مدل تحلیل در هر زمان به مثابه عکسی فوری از خواسته‌هاست. باید انتظار تغییرات را داشته باشید. به موازاتی که مدل خواسته‌ها تکامل پیدا می‌کند، عناصر معینی به پایداری نسبی می‌رسند، و بنیادی محکم برای کارهای طراحی بعدی فراهم می‌سازند، ولی سایر عناصر مدل ممکن است فرّار باشند و این نشان می‌دهد که طرف‌های ذی‌نفع هنوز به‌طور کامل خواسته‌های سیستم را درک نکرده‌اند. مدل تحلیل و روش‌های به‌کار رفته در ساخت آن به تفصیل در فصل‌های ۶ و ۷ ارائه شده‌اند. در بخش‌هایی که به‌دنبال خواهد آمد، نگاهی اجمالی به این مباحث خواهیم داشت.

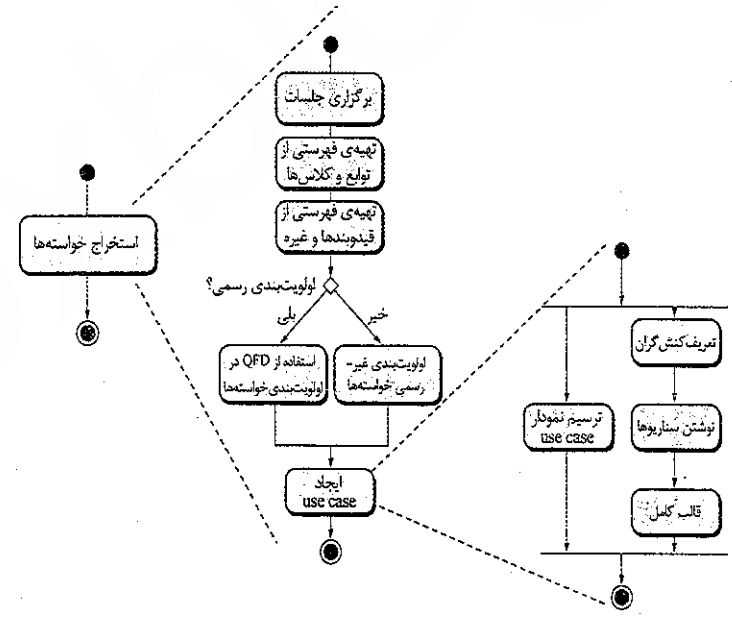
۵-۵-۱ عناصر مدل خواسته‌ها

برای توجه به خواسته‌های یک سیستم کامپیوتری، راه‌های متفاوت بسیاری وجود دارد. برخی دست‌اندرکاران نرم‌افزار استدلال می‌کنند که بهترین کار، انتخاب یک حالت نمایش (مثلاً use case) و به‌کارگیری آن در طرد همه‌ی مدل‌های دیگر است. سایر دست‌اندرکاران بر این باورند که استفاده از چند حالت نمایشی متفاوت برای به تصویر کشیدن مدل خواسته‌ها، ارزش‌مند است. حالت‌های متفاوتی از نمایش، شما را وا می‌دارد که مدل خواسته‌ها را از دیدگاه‌های متفاوت در نظر بگیرید- در این رویکرد، احتمال آشکارشدن موارد جاافتاده، ناسازگاری‌ها و ابهامات بیشتر می‌شود.

عناصر خاصی از مدل خواسته‌ها توسط روش مدل‌سازی دیکته می‌شوند (فصل‌های ۶ و ۷). به‌هرحال، مجموعه‌ای از عناصر کلی در اکثر این مدل‌ها مشترک هستند.

عناصر مبتنی بر سناریو. سیستم از دیدگاه کاربر با استفاده از رویکردی مبتنی بر سناریو توصیف می‌شود. برای مثال، use case پایه (بخش ۴-۵) و نمودارهای use case مرتبط با آنها (شکل ۲-۵)

به use case مبتنی بر الگو و با جزئیات بیشتر، تکامل پیدا می‌کنند. عناصر مبتنی بر سناریو در مدل خواسته‌ها غالباً نخستین بخش از مدلی هستند که توسعه می‌یابد. به این ترتیب، این عناصر به‌عنوان ورودی برای ایجاد سایر عناصر مدل‌سازی عمل می‌کنند. در شکل ۳-۵، یک نمودار UML از فعالیت‌ها برای استخراج خواسته‌ها و نمایش آنها با استفاده از use case نشان داده شده است[1]. سه سطح از شناخت مشاهده می‌شود که در یک نمایش مبتنی به سناریو به اوج خود می‌رسد.



شکل ۳-۵ نمودارهای فعالیت UML برای استخراج خواسته‌ها.

عناصر مبتنی بر کلاس. هر سناریوی کاربرد نشان‌گر مجموعه‌ای از اشیاست که با تعامل یک کنش‌گر با سیستم، دستکاری می‌شوند. این اشیا در قالب چند کلاس طبقه‌بندی می‌شوند- مجموعه‌ای از چیزها که صفات و رفتارهای مشابه دارند. برای مثال، از نمودار کلاس‌های UML می‌توان به‌منظور نمایش کلاس **حس‌گر** برای قابلیت ایمنی منزل در محصول ***SafeHome*** بهره برد (شکل ۴-۵). توجه دارید که در این نمودار صفات حس‌گرها (مثلاً نوع و نام) و عملیات قابل انجام برای اصلاح این صفات (مانند شناسایی و فعال شدن) فهرست شده است. علاوه بر نمودارهای کلاس‌ها، سایر عناصر مدل‌سازی تحلیل، شیوه‌ی همکاری کلاس‌ها با یکدیگر و روابط تعامل میان کلاس‌ها را نیز به تصویر می‌کشند. این روابط را با جزئیات بیشتر در فصل ۷ به بحث خواهیم گذاشت.

عناصر رفتاری. رفتار یک سیستم کامپیوتری می‌تواند اثری عمیق بر انتخاب طراحی و رویکرد مورد استفاده در پیاده‌سازی داشته باشد. بنابراین، مدل خواسته‌ها باید عناصر مدل‌سازی لازم برای تصویرکردن این رفتار را فراهم سازد.

اندرز

در گیر کردن طرف‌های ذی‌نفع همیشه ایده خوبی است. یکی از بهترین راه‌ها برای انجام این کار، آن است که از هر طرف ذی‌نفع بخواهیم، use case یا برای توصیف چگونگی استفاده از نرم‌افزار بنویسد.

اندرز

یک راه برای جداسازی کلاس‌ها، جستجو به‌دنبال اسم‌های توصیفی در یک use case است. حداقل برخی از این اسم‌ها نامزدهایی برای کلاس‌ها خواهند بود. توضیحات بیشتر در فصل ۸.

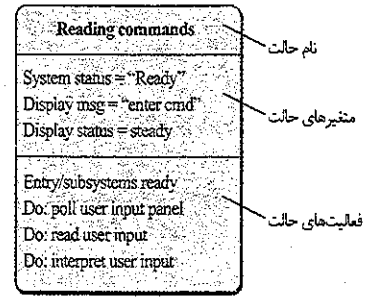
حس‌گر
نام نوع مکان ناحیه خصوصیات
Identefy() Enable() Disable() Reconfigure()

شکل ۴-۵ نمودار کلاس‌ها برای حس‌گر.

[1] برای آنان که با UML ناآشنا هستند خودآموز مختصری در پیوست ۱ داده شده است.

نمودار حالت، یکی از روش‌های نمایش رفتار سیستم با به تصویر کشیدن حالت‌ها و رویدادهایی است که باعث می‌شوند سیستم تغییر حالت دهد. *حالت* به هر شیوه‌ی رفتاری سیستم گفته می‌شود که از بیرون قابل مشاهده باشد. به‌علاوه، نمودار حالت نشان‌گر کنش‌هایی (مثلاً فعال‌سازی فرایند) است که به‌عنوان پیامدی از یک رویداد خاص انجام می‌شوند.

برای روشن‌شدن کاربرد یک نمودار حالت، نرم‌افزار تعبیه‌شده در پانل کنترل ***SafeHome*** را در نظر بگیرید که مسؤولیت خواندن ورودی کاربر را بر عهده دارد. یک نمودار حالت UML ساده شده در شکل ۵-۵ نشان داده شده است.



شکل ۵-۵ نمادگذاری نمودار حالت‌ها در UML.

نکته‌ی کلیدی

حالت به یک شیوه‌ی رفتاری قابل مشاهده از خارج سیستم گفته می‌شود. محرک‌های خارجی باعث گذار میان حالت‌ها می‌شوند.

علاوه بر نمایش‌های رفتاری سیستم به‌عنوان یک کل، رفتار تک تک کلاس‌ها را نیز می‌توان مدل‌سازی کرد. بحث بیشتر درباره مدل‌سازی رفتاری را به فصل ۷ موکول می‌کنیم.

عناصر جریان‌گرا. انتقال اطلاعات با جریان یافتن در یک سیستم کامپیوتری صورت می‌پذیرد. سیستم به شکل‌های گوناگون، ورودی می‌پذیرد، عملیاتی را برای تبدیل و تحول آنها به‌کار می‌گیرد و خروجی را به شکل‌های متنوع تولید می‌کند. ورودی می‌تواند سیگنال کنترلی منتشرشده توسط یک مبدل یک سری اعداد تایپ شده توسط اپراتور انسانی، بسته‌ای از اطلاعات انتقال یافته روی یک لینک شبکه‌ای یا فایل حجیمی از داده‌های بازاریابی شده از روی یک حافظه‌ی ثانویه باشد. این تبدیل(ها) ممکن است شامل تنها یک مقایسه منطقی، یک الگوریتم عددی پیچیده، یا استنباط قانون از یک سیستم خبره باشد. خروجی می‌تواند روشن‌شدن یک چراغ LED یا تولید گزارشی ۲۰۰ صفحه‌ای باشد. در عمل می‌توانیم برای هر سیستم کامپیوتری با هر اندازه و هر میزان از پیچیدگی، یک مدل جریان تهیه کنیم. بحث کامل تری از مدل‌سازی جریان در فصل ۷ ارائه خواهد شد.

۵-۵-۲ الگوهای تحلیل

هر کسی که کار مهندسی خواسته‌ها را روی چند پروژه‌ی نرم‌افزاری انجام داده باشد، رفته رفته متوجه می‌شود که مشکلات معینی در تمامی پروژه‌ها در یک دامنه‌ی کاربردی مشخص دوباره رخ می‌دهند.[1] این الگوهای *تحلیل* [Fow97] راهکارهایی (مثلاً یک کلاس، یک عملکرد، یک رفتار) در این دامنه‌ی کاربردی پیشنهاد می‌کنندکه در مدل‌سازی بسیاری از برنامه‌های کاربردی قابل استفاده‌ی مجدد است.

[1] در برخی موارد، مشکلات، جدا از دامنه‌ی کاربرد، درباره رخ می‌دهد. برای مثال، ویژگی‌ها و قابلیت‌های به کار رفته برای حل مشکلات واسط کاربری جدا از دامنه‌ی کاربرد مورد نظر رایج هستند.

SafeHome

مدل‌سازی مقدماتی رفتار

ادامه جلسه خواسته‌ها.

نقش‌آفرینان: جیمی لازار، عضو تیم نرم‌افزاری؛ وینود رامان، عضو تیم نرم‌افزار؛ اد رابینز، عضو تیم نرم‌افزار؛ داگ میلر، مدیر مهندسی نرم‌افزار؛ سه عضو بازاریابی؛ نماینده‌ای از مهندسی تولید؛ و یک نفر تسهیل‌گر.

گفتگوها:

تسهیل‌گر: کم کم داریم به پایان بحث درباره عملکرد ایمنی منزل در *SafeHome* می‌رسیم، ولی قبل از آن، می‌خواهم درباره رفتار این قابلیت صحبت کنیم.

عضو بازاریابی: نمی‌فهمم. منظورتان از رفتار چیست؟

اد (با لبخند): همین که وقتی محصولی درست رفتار نکند، به آن «وقت اضافه» می‌دهید.

تسهیل‌گر: دقیقاً نه. بگذارید توضیح بدهم.

(تسهیل‌گر اصول مدل‌سازی را برای تیم جمع‌آوری خواسته‌ها توضیح می‌دهد.)

عضو بازاریابی: قدری فنی به‌نظر می‌رسد. گمان نکنم من بتوانم در این زمینه کمکی بکنم.

تسهیل‌گر: حتماً می‌توانی. از دیدگاه یک کاربر چه رفتاری را مشاهده می‌کنی؟

عضو بازاریابی: خوب... سیستم، حس‌گرها را پایش خواهد کرد. فرمان‌های صادر شده از سوی صاحب‌خانه را خواهد خواند و حالت خودش را نمایش خواهد داد.

تسهیل‌گر: دیدی؟ می‌توانی.

جیمی: به‌علاوه باید گوش به زنگ PC هم باشد تا اگر ورودی از آن رسید، مثلاً دستیابی اینترنتی یا اطلاعات پیکربندی، آنها را دریافت کند.

وینود: آره. در واقع، پیکربندی سیستم هم به نوبه خودش یک حالت به‌شمار می‌رود.

داگ: شماها دارید تقلای بیهوده می‌کنید. بهتر است بیشتر درباره آن فکر کنیم... راهی برای ارائه این مطالب در قالب نمودار نیست؟

تسهیل‌گر: چرا، هست، ولی می‌گذاریم برای بعد از جلسه.

گه‌یر-شولتز و هاشلر [Gey01] دو مزیت را پیشنهاد می‌کنند که با به‌کارگیری الگوهای تحلیل می‌توان آنها را مرتبط دانست:

> نخست، این الگوهای تحلیل، توسعه‌ی مدل‌های تحلیل انتزاعی را سرعت می‌بخشند؛ این مدل‌های تحلیل انتزاعی، خواسته‌های اصلی مسأله را با فراهم ساختن مدل‌های تحلیل قابل استفاده‌ی مجدد مشخص می‌کنند؛ هر مدل تحلیل شامل یک سری مثال و نیز توصیفی از مزایا و محدودیت‌های موجود است. دوم، الگوهای تحلیل، تبدیل مدل تحلیل به یک مدل طراحی را با پیشنهاد الگوهای طراحی و راهکارهای قابل اطمینان برای مسائل رایج تسهیل می‌کند.

الگوهای تحلیل با ارجاع به نام الگو در مدل تحلیل قرار داده می‌شوند. این الگوها همچنین در یک مخزن نگهداری می‌شوند تا مهندسان خواسته‌ها بتوانند با استفاده از تسهیلات جستجوگر آنها را بیابند

و مورد استفاده قرار دهند. اطلاعات مربوط به یک الگوی تحلیل (و انواع الگوهای دیگر) در یک قالب استاندارد ارائه می‌شود [Gey01] [1] که با جزئیات بیشتر در فصل ۱۲ بحث خواهد شد. مثال‌هایی از الگوهای تحلیل و بحث بیشتر درباره این مبحث در فصل ۷ عرضه خواهد شد.

۵-۶ مذاکره بر سر خواسته‌ها

در یک حیطه‌ی ایده‌آل از مهندسی خواسته‌ها، وظایف دریافت، استخراج و جزئیات، خواسته‌های مشتری را با تفصیل کافی تعیین می‌کنند تا بتوان به سمت سایر فعالیت‌های مهندسی نرم‌افزار حرکت کرد، ولی متاسفانه این اتفاق کمتر رخ می‌دهد. در واقع، ممکن است با یک یا چند طرف ذی‌نفع وارد مشاجره شوید. در اکثر موارد، از طرف‌های ذی‌نفع خواسته می‌شود که عملکردها، کارایی و سایر خصوصیات محصول یا سیستم را در مقابل هزینه و زمان ارائه به بازار موازنه کنند. هدف از این مذاکره توسعه‌ی یک طرح پروژه است که نیازهای طرف ذی‌نفع را بر طرف سازد و در همان حال قیدوبندهای جهان واقعی (مانند زمان، آدم‌ها، بودجه) را که تیم نرم‌افزاری با آنها مواجه است، منعکس می‌کند.

> «مصالحه، هنر تقسیم یک کیک است به گونه‌ای که هر کس بر این باور باشد که بزرگترین قطعه نصیب او شده است.»
>
> لودویگ ارهارد

اطلاعات

هنر مذاکره

فراگیری نحوه‌ی مذاکره موثر می‌تواند شما را در سرتاسر زندگی شخصی و فنی تان یاری دهد. در نظر گرفتن دستورات زیر می‌تواند ارزش مند باشد:

۱. *این که بدانید رقابتی در کار نیست.* برای موفقیت، هر دو طرف باید احساس کنند که برنده‌اند و چیزی عایدشان شده است. هر دو طرف باید به مصالحه برسند.

۲. *راهبردی را ترسیم کنید.* تصمیم بگیرید که چه چیزی می‌خواهید عایدتان شود؛ طرف مقابل به‌دنبال چیست و چه می‌توانید بکنید که هر دو اتفاق رخ دهد.

۳. *فعالانه گوش بدهید.* در حالی که طرف دیگر در حال صحبت است، مشغول کار روی پاسخ خودتان نباشید. به او گوش بدهید. احتمالاً اطلاعاتی به‌دست می‌آورید که به شما کمک می‌کند تا بهتر درباره موقعیت خود مذاکره کنید.

۴. *به علائق طرف دیگر توجه کنید.* اگر می‌خواهید از تضاد و تناقض پرهیز کنید، جبهه‌گیری نکنید.

۵. *اجازه ندهید مسائل جنبه‌ی شخصی پیدا کند.* به مسأله‌ای که قرار است حل شود، توجه کنید.

۶. *خلاق باشید.* اگر در تنگنا قرار گرفتید به فکر چاره‌ای برای خروج از آن باشید.

۷. *آمادگی تعهد را داشته باشید.* پس از اینکه به توافق رسیدید، به آن عمل کنید؛ به قول خود عمل کنید و ادامه دهید.

[1] در برخی موارد، دامنه کاربرد هر چه که باشد، مسأله درباره رخ می‌نماید. برای مثال، ویژگی‌ها و قابلیت‌های عملیاتی به‌کاررفته در حل مسائل واسط کاربری در هر دامنه کاربردی که باشند، مستقل از دامنه مورد نظرند.

SafeHome

شروع مذاکره

صحنه: دفتر لیزا پرز، پس از نخستین جلسه جمع‌آوری خواسته‌ها.

نقش‌آفرینان: داگ میلر، مدیر مهندسی نرم‌افزار و لیزا پرز، مدیر بازاریابی.

گفتگوها:

لیزا: شنیدم که اولین جلسه به‌خوبی برگزار شده.

داگ: راستش، بله. تو آدم‌های خوبی به جلسه فرستاده بودی... واقعاً سهم داشتند.

لیزا (با لبخند): آره، در واقع آنها به من گفتند که خودشان را خوب قاطی کردند و همه‌ی بحث به مسائل فنی محدود نشده.

داگ (با خنده): دفعه بعد که آنها را ببینم، اصطلاحات فنی خودم را رو می‌کنم... ببین لیزا، من فکر می‌کنم با آن تاریخ‌هایی که مدیریت شما برای سیستم ایمنی منزل در نظر گرفته، نتوانیم همه‌ی قابلیت‌ها را تأمین کنیم. زود است، می‌دانم، ولی همین الان هم ما یک کم از برنامه عقبیم و...

لیزا (با اخم): داگ، ما باید در آن تاریخ آماده‌اش کنیم. تو از کدام قابلیت حرف می‌زنی؟

داگ: من می‌دانم که می‌توانیم قابلیت ایمنی منزل را به‌طور کامل تا آن تاریخ تحویل بدهیم، ولی برای دستیابی اینترنتی باید تا نگارش دوم صبر کنیم.

لیزا: داگ، همین دستیابی اینترنتی است که به ***SafeHome*** یک جاذبه غیر عادی می‌دهد. ما می‌خواهیم کل فعالیت‌های بازاریابی خودمان را روی آن بنا کنیم. این ویژگی را حتماً باید داشته باشیم.

داگ: من شرایط شما را درک می‌کنم، واقعاً درک می‌کنم. مشکل اینجاست که برای دستیابی به اینترنت باید یک وب‌سایت با امنیت کامل بسازیم و راه‌اندازی کنیم. و این نیاز به زمان و نیروی کار دارد. به‌علاوه باید یک عالم قابلیت‌های اضافی را در نگارش اول بسازیم... گمان نکنم با منابعی که در اختیار داریم، از پس آن بر بیاییم.

لیزا (هنوز اخم بر چهره دارد): می‌فهمم. ولی باید راهی برای انجام آن پیدا کنی. این قضیه برای قابلیت‌های ایمنی منزل و حتی بقیه قابلیت‌های سیستم، اهمیت محوری دارد... حالا آنها را می‌شود به نگارش بعد موکول کرد... با این موافقم.

به‌نظر می‌رسد لیزا و داگ در تنگنا قرار گرفته‌اند و هنوز باید برای حل این مشکل به مذاکره ادامه دهند. آیا هر دو آنها می‌توانند برنده‌ی این بازی باشند؟ شما در نقش میانجی چه پیشنهادی دارید؟

هدف بهترین مذاکره‌ها، نتیجه‌ای «بُرد- بُرد» است.[1] یعنی طرف‌های ذی‌نفع با دریافت سیستم یا محصولی که اکثر نیازهای آنها را برآورده می‌کند، برنده شده باشد و شما (به‌عنوان عضوی از تیم نرم‌افزاری) با کار در مهلت‌ها و بودجه‌های واقع‌بینانه و قابل دستیابی برنده شده باشید.

[1] درباره مهارت‌های چانه‌زنی ده‌ها کتاب نوشته شده است (مثل [Lew06]، [Rai06]، [Fis06]). این مهارت، یکی از مهمترین مهارت‌هایی است که باید فرا بگیرید. پس یکی از این کتاب‌ها را بخوانید.

بوهم [Boe98] مجموعه‌ای از فعالیت‌های مذاکره را در شروع هر دور تکرار از فرایند نرم‌افزار تعریف می‌کند. به‌جای یک فعالیت منفرد برقراری ارتباط با مشتری، فعالیت‌های زیر تعیین می‌شود:

۱. شناسایی طرف‌های ذی‌نفع مهم سیستم یا زیر سیستم.
۲. تعیین شرایط برد برای طرف‌های ذی‌نفع.
۳. مذاکره بر سر شرایط برد طرف‌های ذی‌نفع برای رساندن آنها به شرایط برد-برد، برای تمامی طرف‌ها (از جمله تیم نرم‌افزار).

با تکمیل موفقیت آمیز این مراحل آغازی، نتیجه‌ی برد-برد عاید خواهد شد که ملاک کلیدی برای پیش‌رفتن به فعالیت‌های بعدی در مهندسی نرم‌افزار است.

مرجع وب

مقاله مختصری درباره مذاکره برای خواسته‌های نرم‌افزار را می‌توانید از آدرس زیر دانلود کنید.

www.alexander-egyed.com/publications/software Requirements-NegotiationSome-Lessons-Learned.html

۵-۷ اعتبارسنجی خواسته‌ها

با ایجاد هر کدام از عناصر مدل خواسته‌ها، باید آن را از نظر ناسازگاری، ابهام و موارد جاافتاده بررسی کرد. خواسته‌هایی که مدل به نمایش می‌گذارد به موازات رشد و توسعه‌ی نرم‌افزار، توسط طرف‌های ذی‌نفع، اولویت‌بندی و در داخل بسته‌هایی گروه‌بندی می‌شود.

هنگام بازبینی خواسته‌ها چه باید پرسم؟

با مروری بر مدل خواسته‌ها به پرسش‌های زیر باید پاسخ گفته شود:

- آیا هر خواسته‌ای با اهداف کلی سیستم/ محصول همخوانی دارد؟
- آیا همه‌ی خواسته‌ها در سطح مناسبی از انتزاع، مشخص شده‌اند؟ یعنی آیا در برخی خواسته‌ها سطحی از جزئیات فنی ارائه شده است که در این مرحله مناسب نباشد؟
- آیا این خواسته واقعاً لازم است یا یک ویژگی اضافی را نشان می‌دهد که ممکن است برای هدف سیستم ضروری نباشد؟
- آیا همه‌ی خواسته‌ها خالی از ابهام هستند؟
- آیا همه‌ی خواسته‌ها دارای صفت هستند؟ یعنی یک منبع (عموماً فردی مشخص) برای هر خواسته ذکر شده است؟
- آیا هر خواسته‌ای با سایر خواسته‌ها تضاد ندارد؟
- آیا هر خواسته‌ای در محیط فنی سیستم یا محصول قابل دستیابی است؟
- آیا هر خواسته‌ای پس از پیاده‌سازی قابل آزمون هست؟
- آیا مدل خواسته‌ها به‌طرزی مناسب، اطلاعات، عملکرد و رفتار سیستم مورد نظر را منعکس می‌سازد؟
- آیا مدل خواسته‌ها به‌شیوه‌ای «افراز» شده است که به‌طور فزاینده اطلاعات جزئی تری را درباره سیستم آشکار سازد؟
- آیا از الگوهای خواسته‌ها برای ساده‌سازی مدل خواسته‌ها استفاده شده است؟ آیا همه‌ی الگوها به‌طور مناسب اعتبارسنجی شده‌اند؟ آیا همه‌ی الگوها با خواسته‌های مشتریان سازگارند؟

این پرسش‌ها و نظایر آن باید پرسیده شوند و به آنها پاسخ داده شود تا اطمینان حاصل شود که مدل خواسته‌ها انعکاس درستی از نیازهای ذی‌نفع‌هابوده، بنیادی محکم برای طراحی فراهم می‌سازد.

۵-۸ خلاصه

وظایف مهندسی خواسته‌ها برای ایجاد بنیادی محکم جهت طراحی و ساخت نرم‌افزار اجرا می‌شود. مهندسی خواسته‌ها طی فعالیت‌های برقراری ارتباط و مدل‌سازی رخ می‌دهد که برای فرایند مهندسی کلی تعریف شده‌اند. هفت وظیفه در مهندسی خواسته‌ها وجود دارد- دریافت، استخراج، تعیین جزئیات، تعیین مشخصات، اعتبارسنجی و مدیریت- که اعضای تیم نرم‌افزاری باید آنها را انجام دهند.

در مرحله‌ی شروع، طرف‌های ذی‌نفع خواسته‌های اصلی مسأله را تعیین می‌کنند، قیدوبندهای پروژه را مشخص می‌کنند و به ویژگی‌ها و قابلیت‌هایی می‌پردازند که باید در سیستم موجود باشند تا به اهداف خود برسد. این اطلاعات در مرحله‌ی استخراج، پالایش و بسط داده می‌شوند- فعالیتی در جمع‌آوری خواسته‌ها که از جلسات تسهیل شده، QFD و توسعه‌ی سناریوهای کاربرد استفاده می‌کند.

در مرحله‌ی تعیین جزئیات، خواسته‌ها باز هم در یک مدل بسط داده می‌شوند- این مدل مجموعه‌ای از عناصر مبتنی بر سناریو، مبتنی بر کلاس، رفتاری و جریان گراست. مدل مذکور ممکن است به الگوهای تحلیل ارجاع دهد یعنی راهکارهایی برای مسائل تحلیلی که مشاهده شده است در کاربردهای متفاوت دوباره رخ می‌دهند.

با شناخته‌شدن خواسته‌ها و ایجادشدن مدل خواسته‌ها، تیم نرم‌افزاری و سایر طرف‌های ذی‌نفع بر سر اولویت‌ها، دسترسی‌ها و هزینه‌ی نسبی هر خواسته مذاکره می‌کنند. مقصود از این مذاکره، توسعه یک طرح پروژه واقع بینانه است، به‌علاوه، هر کدام از خواسته‌ها و مدل خواسته‌ها در کل در مقابل مشتری باید اعتبارسنجی شود تا اطمینان حاصل شود که درست ساخته خواهد شد.

مسائل و نکاتی برای تعمق

۵-۱ چرا بسیاری از سازندگان نرم‌افزار توجه کافی به مهندسی خواسته‌ها نمی‌کنند؟ آیا شرایطی وجود دارد که بتوان آن را نادیده گرفت؟

۵-۲ مسؤولیت استخراج خواسته‌ها به شما واگذار شده است و مشتری می‌گوید سرش شلوغ‌تر از آن است که بتواند با شما ملاقات کند، چه باید بکنید؟

۵-۳ برخی مشکلات را که ممکن است هنگام دریافت خواسته‌ها از سه یا چهار مشتری متفاوت پدید آید مورد بحث قرار دهید.

۵-۴ چرا می‌گوییم که مدل خواسته‌ها نشان‌گر عکسی از سیستم در زمان است؟

۵-۵ فرض می‌کنیم که مشتری را قانع کرده‌اید تا با همه‌ی درخواست‌های شما به‌عنوان سازنده‌ی نرم‌افزار موافقت کند (چون فروشنده‌ی خوبی هستید). آیا باید شما را استاد مذاکره دانست؟

۵-۶ دست کم سه «پرسش مستقل از حیطه» ارائه دهید که می‌توان در طول مرحله‌ی شروع از یک ذی‌نفع پرسید.

۵-۷ یک کیت جمع‌آوری خواسته‌ها بسازید. این کیت باید شامل مجموعه‌ای از دستورالعمل‌ها برای اجرای یک جلسه جمع‌آوری خواسته‌ها و موادی باشد که بتوان از آنها در تسهیل ایجاد فهرست‌ها و هر چیز دیگری استفاده کرد که ممکن است به تعریف خواسته‌ها کمک کند.

۵-۸ استادتان کلاس را به گروه‌های پنج یا شش نفره تقسیم خواهد کرد. نیمی از هر گروه نقش بخش بازاریابی و نیم دیگر نقش مهندسی نرم‌افزار را بر عهده دارد. وظیفه‌ی شما تعریف خواسته‌ها برای قابلیت ایمنی در محصول ***SafeHome*** است که در این فصل توصیف شده است. جلسه‌ای برای جمع‌آوری خواسته‌ها با استفاده از دستورالعمل‌های ارائه شده در این فصل اجرا کنید.

۵-۹ یک use case برای یکی از فعالیت‌های زیر توسعه دهید:

الف. گرفتن پول نقد از یک خود پرداز.

ب. استفاده از کارت بانکی برای پرداخت پول غذا در رستوران.

پ. خرید سهام با استفاده از حساب سرور بورس.

ت. جستجو به‌دنبال کتاب (در یک مبحث خاص) با استفاده از کتابفروشی آنلاین.

ث. فعالیتی که استادتان مشخص کرده است.

۵-۱۰ use case یی با عنوان «استثنائات» چه چیزی ارائه می‌دهد.

۵-۱۱ *الگوی تحلیل* را به زبان ساده شرح دهید.

۵-۱۲ با استفاده از الگوی ارائه شده در بخش ۵-۵-۲ یک یا چند الگوی تحلیل برای دامنه‌های کاربردی زیر پیشنهاد کنید:

الف. نرم‌افزارهای حسابداری.

ب. نرم‌افزارهای پست الکترونیکی.

پ. مرورگرهای اینترنت.

ت. نرم‌افزارهای واژه پرداز.

ث. نرم‌افزارهای ایجاد وب‌سایت.

ج. دامنه‌ی کاربردی که مربی شما مشخص می‌کند.

۵-۱۳ منظور از برد- برد در حیطه‌ی مذاکره طی فعالیت مهندسی خواسته‌ها چیست؟

۵-۱۴ فکر می‌کنید وقتی در اعتبارسنجی خواسته‌ها خطایی کشف شود، چه اتفاقی رخ می‌دهد؟ چه کسی در تصحیح این خطا باید دخالت کند؟

فصل ۶

مدل‌سازی خواسته‌ها: سناریوها، اطلاعات و کلاس‌های تحلیل

نگاهی گذرا!

مدل‌سازی خواسته‌ها چیست؟ سخنان مکتوب، وسیله‌ای عالی برای برقراری ارتباط به‌شمار می‌روند، ولی ضرورتاً بهترین راه برای نمایش خواسته‌های مربوط به یک نرم‌افزار کامپیوتری نیستند. در مدل‌سازی خواسته‌ها، تلفیقی از شکل‌های متنی و نموداری برای به تصویر کشیدن خواسته‌ها استفاده می‌شود، به شیوه‌ای که درک آن آسان‌تر باشد و مهمتر از آن، به سهولت بتوان آن را برای تصحیح، تکمیل و سازگاری، مورد بازبینی قرار داد.

چه کسی آن را انجام می‌دهد؟ این مدل را یک مهندس نرم‌افزار (که گاهی *تحلیل‌گر* نامیده می‌شود) با به‌کارگیری خواسته‌های استخراج شده از مشتری می‌سازد.

چرا اهمیت داد؟ برای اعتبارسنجی خواسته‌های نرم‌افزار، باید آنها را از چند دیدگاه متفاوت بررسی کنیم. در این فصل، به بحث مدل‌سازی خواسته‌ها از سه دیدگاه متفاوت خواهیم پرداخت: مدل‌های مبتنی بر سناریو، مدل‌های داده‌ای (اطلاعاتی) و مدل‌های مبتنی بر کلاس. در هر کدام از این مدل‌ها، خواسته‌ها از «بُعدی» متفاوت به نمایش در می‌آید و از این رو، احتمال برملاشدن خطاها، روشدن ناسازگاری‌ها و کشف جاافتادگی‌ها افزایش می‌یابد.

مراحل کار کدام است؟ مدل‌های مبتنی بر سناریو، سیستم را از دیدگاه کاربر به نمایش می‌گذارند. مدل‌سازی داده‌ها، فضای اطلاعاتی را به نمایش در می‌آورد و اشیای داده را که نرم‌افزار دستکاری می‌کند و همچنین روابط میان آنها را به تصویر می‌کشد. مدل‌سازی مبتنی بر کلاس‌ها به تعریف اشیاء صفات و روابط می‌پردازد. پس از این که مدل‌های مقدماتی تهیه شدند، مورد پالایش و تحلیل قرار می‌گیرند تا به وضوح، کمال و سازگاری لازم برسند. در فصل ۷، این ابعاد مدل‌سازی را با نمایش‌های بیشتری بسط و توسعه خواهیم داد و از خواسته‌ها دیدی قوی‌تر ارائه خواهیم کرد.

محصول کاری چیست؟ آرایه گسترده‌ای از فرم‌های متنی و نموداری را می‌توان برای مدل خواسته‌ها برگزید. هر کدام از این نمایش‌ها، دیدگاهی از یک یا چند عنصر مدل فراهم می‌سازند.

چطور اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ محصولات کاری مدل‌سازی خواسته‌ها را باید از نظر صحت، کمال و سازگاری بازبینی کرد. این محصولات باید منعکس کننده‌ی نیازهای همه‌ی طرف‌های ذی‌نفع باشند و بستری برای انجام طراحی فراهم سازند.

در سطح فنی، مهندسی نرم‌افزار با یک سری وظایف مدل‌سازی آغاز می‌شود که به تعیین مشخصات خواسته‌ها و نمایش طراحی برای نرم‌افزاری که قرار است ساخته شود، منجر می‌شود. مدل خواسته‌ها[1] -که در واقع مجموعه‌ای از مدل هاست- نخستین نمایش فنی از یک سیستم به‌شمار می‌رود.

تام دومارکو [DeM79] در کتابی مربوط به روش‌های مدل‌سازی خواسته‌ها، این فرایند را چنین توصیف می‌کند:

> من با رجوع به مشکلات و شکست‌های فاز تحلیل، پیشنهاد می‌کنم که باید مواردی را که به دنبال می‌آید به اهداف فاز تحلیل خود اضافه کنیم: محصولات تحلیل باید از قابلیت نگهداری بالایی برخوردار باشند. این *به‌ویژه* برای مستندات هدف [تعیین مشخصات خواسته‌های نرم‌افزار] کاربرد دارد. با مشکلات ناشی از اندازه باید با به‌کارگیری روش‌های‌افزاری اثربخش روبه رو شد. تعیین مشخصات به روش‌های دوره ویکتوریا دیگر جواب نمی دهد. هر جا که امکان داشته باشد، از تصاویر باید استفاده شود.
>
> بین ملاحظات منطقی [اساسی] و فیزیکی [پیاده‌سازی] باید تفاوت قائل شد... در کمترین سطح... به چیزی نیاز داریم که ما را در افراز خواسته‌هایمان یاری دهد و آن افراز را پیش از تعیین مشخصات مستندسازی کند... وسیله‌ای برای پایش و ارزیابی واسط‌ها... ابزارهای جدید برای توصیف منطق و خط مشی، چیزی بهتر از متون روایی.

گرچه دومارکو این مطالب را بیش از یک ربع قرن پیش درباره صفات مدل‌سازی تحلیل نوشته است، نظریات او همچنان در روش‌ها و نمادگذاری مدرن مدل‌سازی خواسته‌ها کاربرد دارند.

۱-۶ تحلیل خواسته‌ها

تحلیل خواسته‌ها به تعیین مشخصات خصوصیات عملیاتی نرم‌افزار منجر می‌شود. واسط نرم‌افزار با سایر عناصر سیستم را مشخص می‌کند و قید و بندهایی را که نرم‌افزار باید رعایت کند، تعیین می‌نماید. تحلیل خواسته به شما (هر نامی که داشته باشید اعم از مهندس نرم‌افزار، تحلیل گر، مدل ساز) این امکان را می‌دهد که طی وظایف دریافت، استخراج و چانه زنی (فصل ۵) جزئیات خواسته‌های پایه را تعیین کنید.

کنش مدل‌سازی خواسته به یک یا چند نوع از مدل‌های زیر می‌انجامد:

- *مدل‌های مبتنی بر سناریو* از دیدگاه «کنش گران» گوناگون سیستم.
- *مدل‌های داده‌ای* که دامنه‌ی اطلاعاتی مسأله را تصویر می‌کنند.
- *مدل‌های مبتنی بر کلاس‌ها* که کلاس‌های شیءگرا (صفات و عملیات) و شیوه‌ی همکاری این کلاس‌ها برای دستیابی به خواسته‌های سیستم را به نمایش می‌گذارند.
- *مدل‌های جریان‌گرا* که عناصر عملیاتی سیستم و چگونگی تبدیل داده‌ها توسط این عناصر را به هنگام حرکت درسیستم نمایش می‌دهند.

> «هر نمایی از خواسته‌ها به تنهایی برای درک یا توصیف رفتار مطلوب یک سیستم پیچیده، ناکافی است.»
>
> **آلن ام. دیویس**

[1] در ویراست‌های گذشته این کتاب از عبارت مدل تحلیل به‌جای مدل خواسته‌ها استفاده شده بود. در این ویراست تصمیم گرفتم برای اشاره به فعالیتی که جنبه‌های گوناگون مسأله را تعریف می‌کنند، از هر دو عبارت استفاده کنیم. تحلیل کنشی است که در به دست آوردن خواسته انجام می‌شود.

- *مدل‌های رفتاری* که چگونگی رفتار نرم‌افزار را به‌عنوان نتیجه‌ای از «رویدادهای» بیرونی به تصویر می‌کشند.

این مدل‌ها اطلاعاتی را در اختیار طراح نرم‌افزار قرار می‌دهند که به طراحی معماری، طراحی واسط‌ها و طراحی در سطح مؤلفه‌ها قابل ترجمه‌اند. سرانجام، مدل خواسته‌ها (و تعیین مشخصات خواسته‌های نرم‌افزار) ابزارهای لازم برای ارزیابی کیفیت را در اختیار سازنده و مشتری قرار می‌دهند.

در این فصل، مدل‌سازی مبتنی بر سناریو را کانون توجه قرار می‌دهیم-تکنیکی که محبوبیت آن در جامعه‌ی مهندسی نرم‌افزار در حال رشدی فزاینده است؛ مدل‌سازی داده‌ها- که تکنیکی تخصصی‌تر به‌شمار می‌رود و *به‌ویژه* هنگامی مناسب است که قرار است نرم‌افزار مورد نظر یک فضای اطلاعاتی پیچیده را ایجاد یا دستکاری کند؛ و *مدل‌سازی کلاس ها*- نمایشی از کلاس‌های شیءگرا و همکاری‌های میان آنها که به سیستم امکان می‌دهند تا قابلیت‌های خودش را بروز دهد. مدل‌های جریان‌گرا، مدل‌های رفتاری، مدل‌سازی مبتنی بر الگوها و مدل‌های مربوط به برنامه‌های تحت وب در فصل ۷ بحث خواهند شد.

۱-۱-۶ فلسفه و اهداف کلی

در سرتاسر مدل‌سازی خواسته‌ها، آنچه که در وهله نخست کانون توجه قرار می‌گیرد، **چیستی** است نه **چگونگی**. در شرایطی خاص چه نوع تعامل‌های کاربری رخ می‌دهد، سیستم چه اشیایی را دستکاری می‌کند، سیستم چه عملیاتی را باید انجام دهد، چه رفتاری باید از خود به نمایش بگذارد، چه واسط‌هایی تعریف می‌شوند و چه قید و بندهایی اعمال می‌شود؟[1]

در فصل‌های اولیه گفتیم که ممکن است تعیین مشخصات کامل خواسته‌ها در این مرحله امکان‌پذیر نباشد. مشتری ممکن است دقیقاً از آنچه که برای جنبه‌های معینی از سیستم مورد نیاز است، مطمئن نباشد. سازنده ممکن است مطمئن نباشد که با یک رویکرد خاص به‌طور مناسب به عملکرد و کارایی خواهد رسید. این واقعیت‌ها باعث می‌شوند که انتخاب رویکردی مبتنی بر تکرار برای مدل‌سازی و تحلیل خواسته‌ها موجه به نظر برسد. تحلیل‌گر باید دانسته‌ها را مدل‌سازی کند و از مدل بدست آمده به‌عنوان مبنایی برای طراحی نسخه‌ی نرم‌افزار استفاده کند.[2]

مدل خواسته‌ها باید به سه هدف دست پیدا کند: (۱) توصیف آنچه که مشتری نیاز دارد، (۲) ایجاد مبنایی برای تهیه طراحی نرم‌افزار و (۳) تعریف مجموعه‌ای از خواسته‌ها که پس از ساخته‌شدن نرم‌افزار بتوان آنها را اعتبارسنجی کرد. مدل تحلیل، پلی است میان توصیف در سطح سیستم (که کل سیستم یا قابلیت عملیات تجاری را آن گونه که مورد دستیابی نرم‌افزار، سخت‌افزار، داده‌ها، انسان یا سایر عناصر سیستمی قرار می‌گیرد، توصیف می‌کند) و یک طراحی نرم‌افزار (فصل‌های ۸ تا ۱۳) که معماری کاربرد نرم‌افزار، واسط‌های کاربری و ساختار را در سطح مؤلفه‌ها توصیف می‌کند. این رابطه در شکل ۱-۶ نشان داده شده است.

> «خواسته‌ها معماری نیستند. خواسته‌ها طراحی نیستند و واسط کاربر هم نیستند. خواسته‌ها نیازند.»
>
> **اندرو هانت و دیوید توماس**

نکته‌ی کلیدی

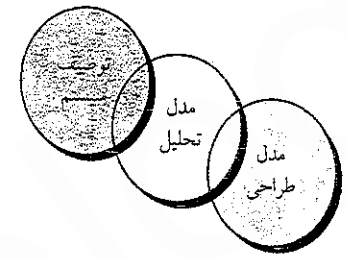
مدل تحلیل و تعیین مشخصات خواسته‌ها، ابزاری برای ارزیابی کیفیت پس از ساخته‌شدن نرم‌افزار فراهم می‌آورد.

نکته‌ی کلیدی

مدل تحلیل باید آنچه را که مشتری می‌خواهد، توصیف کند، بستری برای طراحی ایجاد کند و هدفی برای اعتبارسنجی تعین کند.

[1] لازم به ذکر است که مشتریان از نظر فنی اطلاعات بیشتری کسب می‌کنند و در کنار چیستی، چگونگی نیز باید برای آنها مشخص گردد. ولی توجه اولیه باید برهمان چیستی متمرکز باشد.

[2] به طریق دیگر، تیم نرم‌افزاری ممکن است یک نمونه اولیه ایجاد کند (فصل ۲) تا خواسته‌های سیستم بهتر شناخته شود.



شکل ۱-۶ مدل خواسته‌ها به‌عنوان پلی میان توصیف سیستم و مدل طراحی.

ذکر این نکته حائز اهمیت است که همه‌ی عناصر در مدل خواسته‌ها به‌طور مستقیم تا بخش‌هایی از مدل طراحی قابل ردیابی خواهند بود. تقسیم‌بندی واضح وظایف تحلیل و طراحی میان این دو فعالیت مهم مدل‌سازی، همواره امکان‌پذیر نیست. مقداری از طراحی به‌عنوان بخشی از تحلیل و مقداری از تحلیل در طول طراحی انجام می‌شود.

۶-۱-۲ قواعد ساده‌ی تحلیل

آرلو و نوی اشتات [Arl02] چند قاعده ساده و با ارزش پیشنهاد می‌کنند که هنگام ایجاد مدل تحلیل بهتر است رعایت شوند:

- *مدل باید خواسته‌هایی را کانون توجه قرار دهد که در دامنه‌ی تجاری یا مسأله، قابل مشاهده باشند. سطح انتزاع باید نسبتاً بالا باشد.* «خود را درگیر جزئیاتی نکنید [Arl02] که سعی کنید چگونگی کارکردن سیستم را توضیح دهید.
- *هر عنصر از مدل خواسته‌ها باید در کل چیزی به درک ما از خواسته‌های نرم‌افزار بیفزاید و دیدی از دامنه‌ی اطلاعاتی، عملکرد و رفتار سیستم فراهم سازد.*
- *ملاحظات زیر ساختی و سایر مدل‌های غیر عملیاتی را تا طراحی به تأخیر اندازید.* یعنی ممکن است به یک بانک اطلاعاتی نیاز داشته باشید، ولی کلاس‌های مورد نیاز برای پیاده‌سازی آن، عملکردهای لازم برای دستیابی به آن و رفتاری که به هنگام استفاده از خود نشان خواهد داد، مواردی هستند که تنها پس از کامل‌شدن تحلیل دامنه‌ی مسأله باید به آنها پرداخت.
- *ارتباط‌ها را در سرتاسر سیستم به حداقل برسانید.* نمایش روابط میان کلاس‌ها و عملکردها مهم است، ولی اگر سطح «ارتباط میان مؤلفه‌ها» بسیار بالا باشد، باید تلاش کرد تا این سطح کاهش یابد.
- *مدل خواسته‌ها باید حتماً رضایت همه‌ی طرف‌های ذی‌نفع را جلب کند.* هر گروهی کاربردهای خاص خود را از مدل می‌خواهد. برای مثال، ذی‌نفع‌های تجاری باید از این مدل برای اعتبارسنجی خواسته‌ها استفاده کنند؛ طراحان باید از این مدل به‌عنوان مبنایی برای طراحی استفاده کنند؛ اعضای تضمین کیفیت باید این مدل را برای برنامه‌ریزی آزمون‌های پذیرش به‌کار ببرند.
- *سادگی مدل را تا حد امکان حفظ کنید.* اگر *نموداری* هیچ اطلاعات جدیدی فراهم نمی آورد، آن را اضافه نکنید. اگر یک فهرست ساده کفایت می‌کند، از شکل‌های نمادگذاری پیچیده استفاده نکنید.

آیا دستور العمل‌های پایه‌ای وجود دارد که بتواند ما را در کار تحلیل خواسته‌ها یاری دهد؟

«مسائلی که ارزش حمله کردن دارند، ارزش خود را با پاسخ دادن به حملات به اثبات می‌رسانند.»

پیت هاین

۶-۱-۳ تحلیل دامنه

در بحث مهندسی خواسته‌ها (فصل ۵)، گفتیم که الگوهای تحلیلی غالباً در میان بسیاری از کاربردها در یک دامنه‌ی تجاری خاص دوباره مشاهده می‌شوند. اگر الگوها به شیوه‌ای تعریف و گروه‌بندی شده باشند که بتوانید آنها را برای حل مسائل مشترک به‌کار ببرید، ایجاد مدل تحلیل، تسریع می‌شود. مهمتر اینکه احتمال به‌کارگیری الگوهای طراحی و مؤلفه‌های قابل اجرای نرم‌افزار به‌طور چشمگیری بالا می‌رود. این باعث تسریع در ارائه محصول به بازار و کاهش هزینه‌های توسعه می‌شود، ولی الگوهای تحلیل و کلاس‌ها را چگونه در وهله نخست باید شناسایی کرد؟ چه کسی آنها را تعیین و دسته‌بندی می‌کند و آنها را برای استفاده در پروژه‌های بعدی آماده می‌کند؟ پاسخ به این پرسش‌ها در *تحلیل دامنه* نهفته است. فایر اسمیت [Fir93] تحلیل دامنه را چنین توصیف می‌کند:

> تحلیل دامنه نرم‌افزار عبارت است از شناسایی، تحلیل و تعیین مشخصات خواسته‌های رایج از یک دامنه‌ی کاربردی خاص، معمولاً برای استفاده‌ی مجدد در چندین پروژه که در همان دامنه‌ی کاربردی قرار دارند... [تحلیل دامنه به روش شیء‌گرا عبارت است از] شناسایی، تحلیل و مشخص کردن توانایی‌های قابل استفاده‌ی مجدد مشترک در یک دامنه‌ی کاربردی خاص؛ بر حسب اشیاء، کلاس‌ها و چارچوب‌های مشترک.

این «دامنه‌ی کاربرد خاص» می‌تواند از هوانوردی تا بانکداری، از بازی‌های چندرسانه‌ای تا نرم‌افزارهای تعبیه‌شده در دستگاه‌های پزشکی را در بر گیرد. هدف تحلیل دامنه، صریح است: یافتن یا ایجاد کلاس‌های تحلیل و/یا الگوهای تحلیلی که دارای کاربردی گسترده‌اند و می‌توان دوباره از آنها استفاده کرد.[1]

با استفاده از واژه‌های ارائه شده در ابتدای این کتاب، تحلیل دامنه را می‌توان به‌عنوان فعالیتی چتری برای فرایند نرم‌افزار در نظر گرفت. منظور این است که تحلیل دامنه یک فعالیت مستمر در مهندسی نرم‌افزار است که تنها با یک پروژه‌ی نرم‌افزاری در ارتباط نیست. نقش تحلیل‌گر دامنه از یک لحاظ مشابه با نقش ابزارساز (toolsmith) در یک محیط صنایع سنگین است. وظیفه‌ی این ابزارساز، طراحی و ساخت ابزارهایی است که ممکن است بسیاری از افرادی که کار مشابه، ولی نه الزاماً یکسان انجام می‌دهند، بتوانند از آنها استفاده کنند. نقش تحلیل‌گر دامنه[2] کشف و تعیین الگوهای تحلیل، کلاس‌های تحلیل و اطلاعات مرتبطی است که ممکن است بسیاری از افراد در حال کار روی نرم‌افزارهای مشابه، ولی نه الزاماً یکسان از آنها بهره مند گردند.

در شکل ۲-۶ [Ara89] ورودی‌ها و خروجی‌های کلیدی برای فرایند دامنه تحلیل نشان داده شده است. منابع اطلاعاتی دامنه، مورد نظرخواهی قرار می‌گیرد تا اشیای قابل استفاده‌ی مجدد در آن دامنه شناسایی گردد.

۶-۱-۴ روش‌های مدل‌سازی خواسته‌ها

در یک نما (view) از مدل‌سازی خواسته‌ها، که به *تحلیل ساخت‌یافته* موسوم است، داده‌ها و

مرجع وب

بسیاری از منابع مفید برای تحلیل دامنه را می‌توان در آدرس زیر یافت:

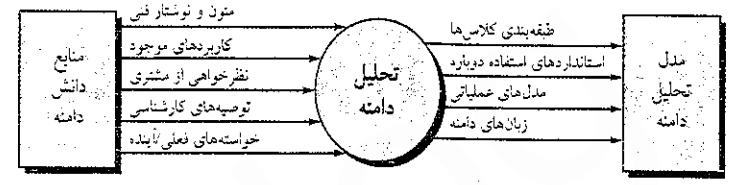
www.iturls.com/English/SoftwareEngineering/SE-mod5.asp

نکته‌ی کلیدی

تحلیل دامنه، به یک برنامه‌ی کاربردی خاص توجه نمی‌کند بلکه دامنه‌ای را مورد توجه قرار می‌دهد که برنامه‌ی کاربردی در آن قرار دارد. هدف آن، شناسایی عناصر مشترک حل مسأله است که می‌توان در همه‌ی برنامه‌های کاربردی دیگر آن دامنه استفاده کرد.

[1] دید مکملی از تحلیل دامنه «شامل مدل‌سازی دامنه می‌شود به طوری که مهندسان نرم‌افزار و سایر ذی‌نفع‌ها بهتر بتوانند از آن مطلب بیاموزند... همه کلاس‌های دامنه الزاماً به توسعه ی کلاس‌های قابل استفاده دوباره منجر نمی شوند...» [Let03a]

[2] تصور نکنید چون تحلیل‌گر دامنه در حال کار است، مهندس نرم‌افزار نیازی به شناخت دامنه‌ی کاربرد ندارد. هر عضو تیم نرم‌افزار باید از دامنه‌ای که نرم‌افزار در آن قرار دارد، درکی نسبی داشته باشد.

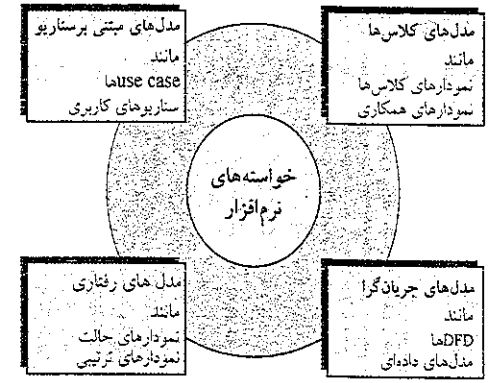


شکل ۲-۶ ورودی و خروجی برای تحلیل دامنه.

فرایندهایی که این داده‌ها را تبدیل می‌کنند، به‌عنوان موجودیت‌هایی مجزا در نظر گرفته می‌شوند. اشیای داده به شیوه‌ای مدل‌سازی می‌شوند که صفات و روابط میان آنها را تعریف کند. فرایندهایی که این اشیای داده را دستکاری می‌کنند، به شیوه‌ای مدل‌سازی می‌شوند که چگونگی تبدیل داده‌ها را به هنگام جریان یافتن اشیای داده در سیستم، نشان دهند.

رویکرد دوم برای مدل‌سازی تحلیل، که *تحلیل شیءگرا* نامیده می‌شود، بر تعریف کلاس‌ها و شیوه‌ی همکاری آنها با یکدیگر برای برآورده ساختن خواسته‌های مشتری تاکید دارد. UML و فرایند یکپارچه (فصل ۲) عمدتاً شیءگرا هستند.

گرچه مدل خواسته‌های پیشنهادی در این کتاب تلفیقی است از ویژگی‌های هر دو رویکرد، تیم مهندسی نرم‌افزارها غالباً تنها یک رویکرد را انتخاب و نمایش‌های مربوط به رویکرد دیگر را از کار خود طرد می‌کنند. مسأله این نیست که کدام رویکرد بهتر است، بلکه باید بپرسیم چه ترکیبی از نمایش‌ها بهترین مدل خواسته‌های نرم‌افزار را در اختیار طرف‌های ذی‌نفع قرار می‌دهد و اثربخش‌ترین پل به طراحی نرم‌افزار خواهد بود.



شکل ۳-۶ عناصر مدل تحلیل.

هر عنصر از مدل خواسته‌ها (شکل ۳-۶) مسأله را از دیدگاهی متفاوت نشان می‌دهد. عناصر مبتنی بر سناریو، چگونگی تعامل کاربر با سیستم و فعالیت‌های خاصی را تصویر می‌کند که هنگام به‌کارگیری نرم‌افزار رخ می‌دهند. عناصر مبتنی بر کلاس‌ها، اشیایی را که سیستم دستکاری می‌کند، عملیاتی را که روی اشیا انجام می‌شود تا این دستکاری‌ها اثر کنند، روابط میان این اشیا (قدری سلسله مراتبی) و همکاری‌هایی را که بین کلاس‌های تعریف می‌شود، مدل‌سازی می‌کنند. عناصر رفتاری، چگونگی تغییر یافتن حالت سیستم یا کلاس‌های موجود در آن توسط رویدادهای خارجی را به

> «... تحلیل کاری است ناراحت کننده، پر از روابط پیچیده میان افراد، نامعین و دشوار. در یک کلام، افسون کننده است. وقتی که به آن عادت کردید، لذت ساخت سیستم به روش قدیمی دیگر هرگز شما را راضی نخواهد کرد.»
>
> تام دومارکو

در توصیف مدل خواسته‌ها چرا می‌توان از دیدگاه‌های متفاوتی بهره برد؟

SafeHome

تحلیل دامنه

صحنه: دفتر داگ میلر، پس از جلسه با بازاریابی.

نقش‌آفرینان: داگ میلر، مدیر مهندسی نرم‌افزار و وینود رامان، عضو گروه مهندسی نرم‌افزار.

مکالمه:

داگ: برای یک پروژه‌ی خاص به تو احتیاج دارم وینود . می‌خواهم تو را از نشست‌های جمع‌آوری خواسته‌ها بیرون بکشم.

وینود (با اخم): خیلی بد شد. آن قالب واقعاً جواب می‌داد... داشت یک چیزهایی از آن دستگیرم می‌شد... موضوع چی هست؟

داگ: جیمی و اد جای تو هستند. خلاصه، بازاریابی اصرار دارد که قابلیت اینترنتی ایمنی منزل را در همان نسخه‌ی اول *SafeHome* ارائه کنیم. در این مورد زیر فشاریم... وقت یا آدم اضافی هم نداریم پس باید حل هر دو تا مسأله-یعنی واسط PC و واسط وب-را با هم و فوری حل کنیم.

وینود (گیج به نظر می‌رسد): نمی دانستم که برنامه ریزی‌ها انجام شده... ما حتی جمع‌آوری داده‌ها را هم تمام نکردیم.

داگ (با لبخندی کمرنگ): می‌دانم، ولی زمان‌بندی آنقدر فشرده است که تصمیم گرفتم الآن با بازاریابی کنار بیایم... خلاصه، وقتی اطلاعاتِ همه‌ی جلساتِ جمع‌آوری خواسته‌ها را در اختیار داشتیم، طرح آزمایشی را بازبینی می‌کنیم.

وینود: بسیار خب. حالا من باید چه کار کنم؟

داگ: تو می‌دانی «تحلیل دامنه» چیست؟

وینود: یک جورهایی. وقتی که داری نرم‌افزاری می‌سازی، در نرم‌افزارهایی با حیطه‌ی کاربرد مشابه، دنبال الگوهای مشابه می‌گردی تا در صورت امکان از این الگوها دوباره استفاده کنی.

داگ: درست است. چیزی که می‌خواهم انجام بدهی، این است که شروع به تحقیق کنی و واسط‌های کاربری موجود برای کنترل دستگاه‌هایی مثل *SafeHome* را پیدا کنی. می‌خواهم یک مجموعه الگو و کلاس‌های تحلیل را پیشنهاد کنی که در واسط PC و واسط اینترنتی دستگاه به‌صورت مشترک قابل استفاده باشند.

وینود: می‌توانیم با یکسان ساختن آنها در وقت صرفه جویی کنیم... چرا این کار را نکنیم؟

داگ: خوب است که آدم‌هایی با طرز تفکر تو داریم. نکته اصلی همین است- اگر هر دو واسط تقریباً یکسان باشند، با کد یکسان پیاده‌سازی شوند و غیره، می‌توانیم در وقت صرفه جویی کنیم.

وینود: پس شما چه می‌خواهید؟ کلاس‌ها، الگوهای تحلیل و الگوهای طراحی؟

داگ: همه‌ی اینها را می‌خواهیم. فعلاً هیچ چیز رسمی وجود ندارد. فقط می‌خواهم کار طراحی و تحلیل درونی از یک جا شروع شود.

وینود: سری به کتابخانه کلاس‌هایمان می‌زنم تا ببینم چه داریم. از شابلون الگوهایی که تازگی‌ها در یک کتاب خوانده بودم هم استفاده می‌کنم.

داگ: خوب است. شروع کن.

تصویر می‌کشند. سرانجام، عناصر جریان‌گرا، سیستم را به‌عنوان یک تبدیل اطلاعات نمایش می‌دهند و چگونگی تبدیل اشیای داده را به هنگام جریان یافتن در سرتاسر عملکردهای گوناگون سیستم به تصویر می‌کشد.

نتیجه‌ی مدل‌سازی تحلیل، به‌دست آمدن هر کدام از این عناصر مدل‌سازی است. ولی، محتوای هر عنصر (یعنی نمودارهایی که برای ساخت آن عنصر و آن مدل به‌کار می‌روند) ممکن است از پروژه‌ای به پروژه‌ی دیگر متفاوت باشد. همان طور که در این کتاب چند بار ذکر شد، تیم مهندسی نرم‌افزار باید در حفظ سادگی بکوشد. تنها آن عناصر مدل‌سازی که ارزشی به مدل اضافه می‌کنند، باید به‌کار گرفته شوند.

۶-۲ مدل‌سازی مبتنی بر سناریو

گرچه موفقیت یک سیستم یا محصول کامپیوتری به طرق گوناگون سنجیده می‌شود، رضایت کاربر در صدر فهرست قرار دارد. اگر بدانید که کاربران نهایی (و سایر کنش گران) چگونه می‌خواهند با یک سیستم تعامل کنند، تیم مهندسی نرم‌افزار شما بهتر قادر به مشخص کردن خواسته‌ها و ساخت مدل‌های تحلیل و طراحی با معنی خواهد بود. از این رو، مدل‌سازی خواسته‌ها با UML [1] با ایجاد سناریوهایی به شکل use case، نمودارهای فعالیت و نمودارهای گردشی آغاز می‌شود.

۶-۲-۱ ایجاد یک use case مقدماتی

آلیستر کاکبرن، use case را به‌عنوان «قرارداد رفتاری» توصیف می‌کند [Coc01b]. چنان که در فصل ۵ بحث شد، این «قرارداد» شیوه‌ی استفاده‌ی یک کنش گر [2] از سیستم کامپیوتری برای رسیدن به هدفی مشخص را تعریف می‌کند. در اصل، use case تعامل‌هایی را به نمایش می‌گذارد که میان تولید کنندگان و مصرف کنندگان اطلاعات و خود سیستم رخ می‌دهد. در این بخش، خواهیم دید که موارد کاربرد چگونه به‌عنوان بخشی از فعالیت مدل‌سازی خواسته‌ها توسعه می‌یابند.[3]

در فصل ۵ ذکر کردیم که use case توصیفی است از یک سناریوی کاربردی خاص به زبانی فصیح از دیدگاه یک کنش‌گر معین، ولی چطور می‌شود فهمید که (۱) درباره چه چیز باید نوشته شود، (۲) چه مقدار باید نوشته شود، (۳) توصیف ما تا چه حد از جزئیات را در برگیرد و (۴) این توصیف چگونه باید سازمان دهی شود؟ اینها پرسش‌هایی هستند که باید پاسخ داده شوند تا use case بتواند ارزش مورد نظر را به‌عنوان ابزار مدل‌سازی خواسته‌ها فراهم سازد.

درباره چه چیز باید نوشت؟ دو وظیفه‌ی نخست در مهندسی خواسته‌ها-دریافت و استخراج- اطلاعات مورد نیاز برای شروع به نوشتن موارد کاربرد را در اختیارتان قرار می‌دهند. نشست‌های جمع‌آوری خواسته‌ها، QFD و سایر سازوکارهای مهندسی خواسته‌ها برای شناسایی ذی‌نفع‌ها، تعریف دامنه

[1] UML به‌عنوان نمادگذاری مدل‌سازی در سرتاسر این کتاب به کار گرفته خواهد شد. در پیوست ۱، خودآموز مختصری برای خوانندگان ناآشنا با نمادگذاری پایه UML ارائه شده است.

[2] کنش‌گر یک فرد خاص نیست بلکه نقشی است که یک فرد (یا دستگاه) در حیطه‌ای مشخص بر عهده دارد. کنش‌گر «سیستم را فراخوانی می‌کند تا یکی از سرویس‌های خود را تحویل دهد» [Coc01b].

[3] use case، بخش به‌ویژه مهمی از مدل‌سازی تحلیل برای واسط‌های کاربری هستند. تحلیل واسط‌ها موضوع فصل ۱۱ است.

«چرا باید مدل بسازیم؟ چرا فقط خود سیستم را نسازیم؟ پاسخ این است که مدل‌ها را طوری می‌سازیم که ویژگی‌های مهم و معینی از سیستم در آن‌ها برجسته‌نمایی شود و در عین حال، بر جنبه‌های دیگری از سیستم، کمتر تأکید گردد.»

اد یوردان

«[use case]ها صرفاً به تعریف آنچه که در بیرون سیستم قرار دارد (کنش‌گر) و آنچه که باید توسط سیستم انجام شود (use case) کمک می‌کند.»

ایوار جیکابسون

اندرز

در برخی وضعیت‌ها، use caseها، به سازوکار غالب در مهندسی خواسته‌ها تبدیل می‌شوند. ولی، این بدان معنا نیست که باید سایر روش‌های مدل‌سازی را کنار بگذارید.

SafeHome

توسعه یک سناریوی کاربری مقدماتی

صحنه: اتاق کنفرانس، طی دومین جلسه جمع‌آوری خواسته‌ها.

نقش‌آفرینان: جیمی لازار، عضو تیم مهندسی نرم‌افزار؛ اد رابینز، عضو تیم مهندسی نرم‌افزار؛ داگ میلر، مدیر نرم‌افزار؛ سه عضو بازاریابی؛ نماینده‌ای از مهندسی تولید؛ و تسهیل‌گر.

گفتگو:

تسهیل‌گر: وقت آن رسیده که صحبت درباره قابلیت پایشی *SafeHome* را شروع کنیم. بیاید یک سناریو کاربری برای دستیابی به قابلیت پایشی بنویسیم.

جیمی: چه کسی نقش کنش‌گر را بازی کند؟

تسهیل‌گر: فکر می‌کنم مردیت (یکی از اعضای بازاریابی) روی این قابلیت کار کرده. تو این نقش را بازی کن.

مردیت: می‌خواهید مثل دفعه‌ی قبل عمل کنیم. درست است؟

تسهیل‌گر: درست است. مثل دفعه قبل.

مردیت: واضح است که دلیل وجود این پایش، این است که به صاحبخانه امکان بدهد خانه را وقتی که بیرون است، زیر نظر داشته باشد و بتواند تصاویر ویدیویی گرفته شده را مشاهده کند. از این جور چیزها.

اد: برای ذخیره و نگهداری تصاویر از فشرده‌سازی هم استفاده می‌کنیم؟

تسهیل‌گر: سؤال خوبی بود اد، ولی اجازه بده مسائل پیاده‌سازی را به بعد موکول کنیم. مردیت؟

مردیت: بله، پس اساساً این قابلیت پایش دو بخش دارد... اول سیستم را پیکربندی می‌کند (از جمله از نظر نقشه ساختمان- باید ابزارهایی فراهم کنیم که به صاحبخانه برای این منظور کمک کند- و بخش دوم، خود عملکرد پایش واقع است. چون تعیین نقشه ساختمان بخشی از فعالیت پیکربندی است، توجه خودم را به خود قابلیت پایشی معطوف می‌کنم.

تسهیل گر(با لبخند): حرف از دل من زدی.

مردیت: من... می‌خواهم هم از طریق PC و هم اینترنت به قابلیت پایش دسترسی داشته باشم. احساس می‌کنم که دستیابی اینترنتی بیشتر استفاده می‌شود. به هر حال، باید این امکان را داشته باشم که نمای دوربین‌ها را روی یک PC نمایش بدهم و بتوانم زوم و زاویه دوربین‌ها را از طریق یک کنسول کنترل کنم. به‌علاوه، می‌خواهم امکان مسدود کردن یک یا چند دوربین را با وارد کردن کلمه‌ی عبور داشته باشم. این گزینه را هم می‌خواهم که پنجره‌های کوچکی را ببینیم که نمای همه‌ی دوربین‌ها را به من نشان دهند و بعد بتوانم هر کدام را که خواستم انتخاب کنم تا تصویر بزرگ شود.

جیمی: به اینها می‌گویند نمای شستی.

مردیت: بسیار خب. پس من نمای شستی همه‌ی دوربین‌ها را می‌خواهم. به‌علاوه، می‌خواهم شکل ظاهری واسط قابلیت پایش مثل همه‌ی واسط‌های دیگر *SafeHome* باشد. می‌خواهم گویا باشد طوری که نیاز به جزوه راهنما نداشته باشد.

تسهیل‌گر: احسنت. حالا این قابلیت را با یک کمی جزئیات بیشتر بررسی می‌کنیم...

و حوزه‌ی مسأله، مشخص کردن اهداف عملیاتی کلی، تعیین اولویت‌ها، مطرح کردن همه‌ی خواسته‌های عملیاتی شناخته‌شده و توصیف اشیای دستکاری‌شده توسط سیستم، به‌کارگرفته می‌شوند.

برای شروع به توسعه‌ی یک مجموعه use case عملیات یا فعالیت‌هایی را که یک کنش‌گر خاص انجام می‌دهد، فهرست کنید. می‌توانید این اطلاعات را از فهرست قابلیت‌های درخواست شده برای سیستم، از طریق مکالمه و گفتگو با طرف‌های ذی‌نفع یا توسط ارزیابی نمودارهای فعالیت (که به‌عنوان بخشی از مدل‌سازی خواسته‌ها تهیه می‌شوند) به‌دست آورید.

قابلیت (زیرسیستم) پایش در محصول *SafeHome* که در کادر قبلی بحث شد، قابلیت‌های زیر را مشخص می‌کند (فهرستی خلاصه شده) که کنش‌گر **homeowner** آنها را انجام می‌دهد:

- انتخاب دوربین برای مشاهده
- درخواست تصاویر کوچکی از همه‌ی دوربین‌ها
- به نمایش درآوردن نمای دوربین‌ها در یک پنجره PC
- کنترل زاویه و زوم یک دوربین مشخص
- ضبط انتخابی خروجی دوربین‌ها
- پخش خروجی دوربین‌ها
- دستیابی به پایش دوربین‌ها از طریق اینترنت

با پیشرفت گفتگو با طرف ذی‌نفع (که نقش صاحبخانه را بازی می‌کند)، تیم جمع‌آوری خواسته‌ها، برای هر کدام از قابلیت‌های ذکر شده، use case تهیه می‌کند. به‌طور کلی، use caseها ابتدا به شیوه‌ای روایی و غیر رسمی نوشته می‌شوند. در صورت نیاز به رسمیت بیشتر، همان use case با استفاده از یک قالب ساخت‌یافته نظیر آنچه که در فصل ۵ پیشنهاد شد (و دوباره در این بخش در حاشیه آورده خواهد شد) بازنویسی می‌شود.

برای روشن‌ترشدن مطلب، عملکردی با عنوان *دستیابی به پایش دوربینی از طریق اینترنت- نمایش خروجی دوربین‌ها* (ACS-DCV) را در نظر بگیرید. طرف ذی‌نفعی که نقش کنش‌گر **homeowner** را برعهده گرفته است، ممکن است شکل روایی زیر را نوشته باشد:

use case: دستیابی به پایش دوربینی از طریق اینترنت- نمایش خروجی دوربین‌ها

(ACS-DCV)

کنش‌گر: homeowner

اگر در مکانی دور دست باشم، می‌توانم از هر PC با یک نرم‌افزار مرورگر مناسب وارد وب‌سایت محصولات *SafeHome* شوم. نام کاربری و دو کلمه‌ی عبور را وارد کنم و هنگامی که هویت خود را به اثبات رساندم، به همه‌ی قابلیت‌های سیستم *SafeHome* که در منزلم نصب شده است، دستیابی داشته باشم. برای دستیابی به نمای یک دوربین معین، از طریق دکمه‌های عملیاتی اصلی نمایش داده شده، «پایش» را انتخاب می‌کنم. سپس با انتخاب گزینه‌ی «انتخاب دوربین»، نقشه ساختمان به نمایش در می‌آید و می‌توانم دوربین مورد نظر را انتخاب کنم. به طریق دیگر، می‌توانم با انتخاب گزینه‌ی «همه‌ی دوربین‌ها» تصاویر کوچک همه‌ی دوربین‌ها را همزمان به نمایش در آورم. پس از انتخاب دوربین، با انتخاب گزینه‌ی «نما» دوربین پنجره مذکور با شماره شناسایی دوربین مشخص می‌شود. اگر بخواهم دوربین را تغییر دهم، با گزینه «انتخاب دوربین» پنجره اولیه محو می‌شود و دوباره نقشه‌ی خانه به نمایش در می‌آید و سپس دوربین مورد نظر را انتخاب می‌کنم و پنجره جدیدی ظاهر می‌شود.

در شکل دیگری از use case روایی، تعامل به‌صورت یک سری کنش‌های ترتیبی ارائه می‌شود. هر کنش به‌صورت یک جمله خبری نمایش داده می‌شود. با بازبینی قابلیت ACS-DCV چنین خواهید نوشت:

use case: دستیابی به پایش دوربینی از طریق اینترنت-نمایش خروجی دوربین‌ها

(ACS-DCV)

کنش‌گر: homeowner

۱. صاحبخانه وارد وب سایت محصولات *SafeHome* می‌شود.
۲. صاحبخانه نام کاربری خودش را وارد می‌کند.
۳. صاحبخانه دو کلمه‌ی عبور (هر کدام حداقل به طول هشت کاراکتر) وارد می‌کند.
۴. سیستم همه‌ی دکمه‌های عملیاتی اصلی را به نمایش در می‌آورد.
۵. صاحبخانه «پایش» را از دکمه‌های اصلی انتخاب می‌کند.
۶. صاحبخانه «انتخاب دوربین» را بر می‌گزیند.
۷. سیستم نقشه ساختمان را نمایش می‌دهد.
۸. صاحبخانه آیکون یکی از دوربین‌ها را از روی نقشه انتخاب می‌کند.
۹. صاحبخانه دکمه «نما» را انتخاب می‌کند.
۱۰. سیستم یک پنجره نمایش ظاهر می‌کند که با شماره شناسایی دوربین مشخص می‌شود.
۱۱. سیستم، خروجی دوربین را در پنجره نمایش با سرعت یک فریم در ثانیه نشان می‌دهد.

لازم به ذکر است که در این نمایش ترتیبی هیچ تعامل دیگری در نظر گرفته نشده است (شکل روایی آن قدری آزادتر بود و چند موردی را به نمایش می‌گذاشت). use case هایی از این نوع، گاهی سناریوهای اولیه نامیده می‌شوند [Sch98a].

> «use caseها را می‌توان در بسیاری از فرایندهای نرم‌افزاری به‌کار برد. چیزی که مطلوب ماست، فرایندی مبتنی بر تکرار و ریسک محور است.»
>
> **گری اشنایدر و جیسون وینتزر**

۲-۲-۶ پالایش یک use case مقدماتی

شرحی از تعامل‌های متفاوت برای درک کامل قابلیت توصیف شده در یک use case ضروری است. بنابراین، هر مرحله از سناریوی اولیه با پرسیدن سؤالات زیر ارزیابی می‌شود [Sch98a]:

- *آیا کنش‌گر در این نقطه، کنش دیگری انجام می‌دهد؟*
- *آیا این امکان وجود دارد که کنش‌گر در این نقطه به شرایط خطا برخورد کند؟* اگر پاسخ مثبت است، این شرایط خطا چه می‌تواند باشد؟
- *آیا این امکان وجود دارد که کنش‌گر در این نقطه با رفتار دیگری مواجه گردد (مثلاً رفتاری که علت آن رویدادی خارج از کنترل کنش‌گر باشد)؟* اگر پاسخ مثبت است، آن رفتار چه می‌تواند باشد؟

> هنگامی که یک use case را توسعه می‌دهم، چگونه اقدام‌های دیگر را بررسی کنم؟

پاسخ این پرسش‌ها به ایجاد مجموعه‌ای از سناریوهای ثانویه می‌انجامد که بخشی از use case اولیه‌اند، ولی رفتارهای دیگر را نشان می‌دهند. برای مثال، مراحل ۶ و ۷ را در سناریو اولیه‌ای که در بالا ارائه شد، در نظر بگیرید:

۶. صاحبخانه «انتخاب دوربین» را بر می‌گزیند.
۷. سیستم، نقشه ساختمان را نمایش می‌دهد.

آیا کنش‌گر در این نقطه کنش دیگری انجام می‌دهد؟ پاسخ، مثبت است. با رجوع به همان نسخه‌ی روایی use case در می‌یابیم که کنش‌گر می‌تواند مشاهده‌ی همزمان تصویر کوچک همه‌ی دوربین‌ها را انتخاب کند. از این رو، سناریوی ثانویه می‌تواند «مشاهده‌ی تصویر کوچک همه‌ی دوربین‌ها» باشد.

آیا ممکن است کنش‌گر در این نقطه به شرایط خطا برخورد کند؟ در کار کردن با یک سیستم کامپیوتری، هر تعداد خطایی ممکن است رخ دهد. در این حیطه، تنها شرایط خطایی را در نظر می‌گیریم که ممکن است به‌عنوان نتیجه‌ی مستقیم کنش شرح داده شده در مرحله‌ی ۶ یا مرحله‌ی ۷ رخ دهد.

دوباره، پاسخ، مثبت است. ممکن است نقشه ساختمان با آیکون‌های نشان دهنده‌ی دوربین‌ها هرگز پیکربندی نشده باشد. از این رو، با گزینش «انتخاب دوربین» شرایط خطایی رخ خواهد داد: «هیچ نقشه‌ای برای این خانه پیکربندی نشده است.»[1] این شرایط خطا یک سناریو ثانویه خواهد شد.

آیا این امکان وجود دارد که کنش‌گر در این نقطه با رفتار دیگری مواجه گردد؟ پاسخ این پرسش نیز مثبت است. با رخ دادن مراحل ۶ و ۷، سیستم ممکن است با شرایط هشدار مواجه گردد. این منجر به نمایش هشداری توسط سیستم (نوع، مکان، کنش سیستم) می‌شود و چند نوع عملیات مرتبط با ماهیت هشدار در اختیار کنش‌گر قرار می‌دهد. از آنجا که این سناریوی ثانویه ممکن است هر لحظه، و در واقع برای همه‌ی تعامل‌ها رخ دهد، بخشی از ACS-DCV use case نخواهد شد بلکه باید یک use case جداگانه- **مواجهه با شرایط هشدار**- نوشته شود و در صورت نیاز در use caseهای دیگر به آن ارجاع شود.

هر کدام از وضعیت‌های شرح داده شده در پاراگراف‌های بالا به‌عنوان یک استثنا برای use case مشخص می‌شود. *استثنا* وضعیتی است (خواه یک شرایط شکست باشد خواه شرایط دیگری که کنش‌گر انتخاب کرده باشد) که باعث می‌شود سیستم رفتاری متفاوت از خود به نمایش بگذارد.

کاکبرن [Coc01b] استفاده از یک جلسه «طوفان فکری» را برای به‌دست آوردن مجموعه کاملی از استثناهای مربوط به هر use case توصیه می‌کند. علاوه بر آن سه پرسش کلی که قبلاً در این بحث مطرح شد، مسائل زیر را نیز باید مطرح کرد:

- *آیا مواردی هست که در آن یک نوع «عمل اعتبارسنجی» در حین این use case رخ دهد؟* این بدان معناست که عمل اعتبارسنجی در خواست می‌شود و یک شرایط خطای بالقوه ممکن است رخ دهد.
- *آیا مواردی هست که در آن یک قابلیت (یا کنش‌گر) پشتیبان از پاسخ دهی مناسب باز بماند؟* برای مثال، کنشی از سوی کاربر منتظر پاسخ بماند، ولی قابلیتی که باید این پاسخ را بدهد، به موقع عمل نکند.
- *آیا عملکرد ضعیف سیستم به کنش‌های غیر منتظره یا نامناسب منجر می‌شود؟* برای مثال، یک واسط مبتنی بر وب بیش از حد آهسته پاسخ دهد و در نتیجه، کاربر، دکمه‌ای را چند بار انتخاب کند. این انتخاب‌های پیاپی ممکن است ایجاد یک صف نامناسب کند که نتیجه‌اش شرایط خطاست.

[1] در این مورد، کنش‌گر دیگر، **system administrator** باید نقشه خانه را پیکربندی کند، دوربین‌ها را نصب و راه اندازی کند (مثلاً یک شماره شناسایی به آنها بدهد) و هر کدام از دوربین‌ها را آزمایش کند تا یقین حاصل کند که از طریق سیستم و از طریق نقشه قابل دستیابی اند.

فهرست موارد بسط داده شده به‌عنوان نتیجه‌ای از این پرسش و پاسخ‌ها را باید با استفاده ملاک‌هایی که به دنبال خواهد آمد، «توجیه کرد» [Coc01b].

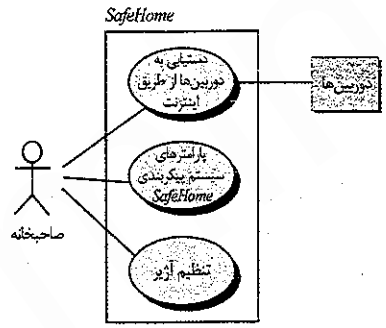
استثنا را در صورتی باید در use case توصیف کرد که نرم‌افزار قادر به تشخیص شرایط توصیف‌شده و سپس انجام اقدام مناسب در صورت تشخیص آن باشد. در برخی موارد، استثنا باعث به تعلیق در آمدن توسعه یک use case دیگر می‌شود (تا برای آن شرایط کاری صورت گیرد).

۶-۲-۳ نوشتن یک use case رسمی

use caseهای غیر رسمی که در بخش ۶-۲-۱ ارائه شدند گاهی برای مدل‌سازی خواسته‌ها کفایت می‌کنند. ولی، هنگامی که یک use case شامل فعالیتی مهم می‌شود یا مجموعه پیچیده‌ای از مراحل را با تعداد چشمگیری از استثناها توصیف می‌کند، روشی رسمی‌تر ممکن است مطلوب باشد.

در **ACS-DCV use case** که در کادر زیر نشان داده شد، از یک فرمت‌بندی متداول برای use caseهای رسمی پیروی شده است. *هدف حیطه‌ای* حوزه کلی use case را مشخص می‌کند. *پیش‌شرط*، چیزی را مشخص می‌کند که باید «برقرار» باشد تا use case عمل کند. راه‌انداز (trigger)، رویداد یا شرطی را مشخص می‌کند که «باعث شروع به‌کار use case می‌شود» [Coc01b]. در سناریو، کنش‌های مورد نیاز کنش‌گر و پاسخ‌های مناسب سیستم، فهرست می‌شود. در *استثناها*، شرایطی مشخص می‌شود که با پالایش use case مقدماتی کشف می‌شوند (بخش ۶-۲-۲). عناوین دیگری هم ممکن است به چشم بخورد که خودشان به روشنی توضیح می‌دهند به چه منظور آورده شده‌اند.

در بسیاری موارد، نیازی به ایجاد نمایش گرافیکی از use case نیست، ولی نمودارها می‌توانند درک و شناخت را تسهیل کنند *به‌ویژه* زمانی که سناریو پیچیده باشد. چنان که قبلاً در این کتاب ذکر شد، با UML قادر به نمایش use caseها در قالبی نموداری هستیم. در شکل ۶-۴ یک نمودار مقدماتی use caseها برای محصول *SafeHome* نشان داده شده است. هر use case توسط یک بیضی نشان داده شده است. در این بخش تنها **ACS-DCV use case** را مورد بحث قرار دادیم.



شکل ۶-۴ نمودار یک use case مقدماتی برای سیستم *SafeHome*

هر نمادگذاری مدل‌سازی با محدودیت‌هایی همراه است و use case نیز از این قاعده مستثنا نیست. use case همانند هر شکل دیگری از توصیف مکتوب، فقط به اندازه‌ی نویسندگانش خوب است. اگر این توصیف روشن و واضح نباشد، use case ممکن است باعث گمراهی یا ابهام شود.

چه هنگامی یک use case به پایان می‌رسد؟ برای بحث ارزشمندی درباره این موضوع، وب سایت زیر را ببینید.
ootips.org/use-cases-done.html

SafeHome

برای پایش *use case* قالب‌بندی

use case: دستیابی به پایش دوربینی از طریق اینترنت- نمایش خروجی دوربین‌ها (DCV-ACS)

دور تکرار: ۲، آخرین اصلاح: ۱۴ ژانویه توسط وینود رامان.

کنش‌گر اولیه: صاحبخانه

هدف حیطه‌ای: مشاهده‌ی خروجی دوربین‌های کار گذاشته شده در سراسر خانه از هر مکان دوردست از طریق اینترنت.

پیش‌شرط‌ها: سیستم باید کاملاً پیکربندی شده باشد، نام کاربری و کلمات عبور مناسب باید در اختیار کاربر قرار داده شده باشد.

راه‌انداز: صاحبخانه وقتی که دور از خانه است تصمیم می‌گیرد نگاهی به داخل خانه بیندازد.

سناریو:

۱. صاحبخانه وارد وب سایت محصولات SafeHome می‌شود.
۲. صاحبخانه نام کاربری خودش را وارد می‌کند.
۳. صاحبخانه دو کلمه‌ی عبور (هر کدام حداقل به طول هشت کاراکتر) وارد می‌کند.
۴. سیستم، همه‌ی دکمه‌های عملیاتی اصلی را به نمایش در می‌آورد.
۵. صاحبخانه «پایش» را از دکمه‌های اصلی انتخاب می‌کند.
۶. صاحبخانه «انتخاب دوربین» را بر می‌گزیند.
۷. سیستم، نقشه ساختمان را نمایش می‌دهد.
۸. صاحبخانه آیکون یکی از دوربین‌ها را از روی نقشه انتخاب می‌کند.
۹. صاحبخانه دکمه «نما» را انتخاب می‌کند.
۱۰. سیستم، یک پنجره نمایش ظاهر می‌کند که با شماره شناسایی دوربین مشخص می‌شود.
۱۱. سیستم، خروجی دوربین را در پنجره نمایش با سرعت یک کادر در ثانیه نشان می‌دهد.

استثناها:

۱. نام کاربری یا کلمات عبور نادرست هستند یا تشخیص داده نمی‌شوند- use case اعتبارسنجی نام کاربری و کلمات عبور را ببینید.
۲. قابلیت پایش برای این سیستم پیکربندی نشده است- سیستم، پیام خطای مناسب را به نمایش در می‌آورد؛ use case پیکربندی قابلیت پایش را ببینید.
۳. صاحبخانه گزینه «مشاهده تصاویر شستی همه‌ی دوربین‌ها» را انتخاب می‌کند- use case مشاهده تصاویر شستی همه‌ی دوربین‌ها را ببینید.
۴. نقشه خانه وجود ندارد یا هنوز پیکربندی نشده است- پیام خطای مناسبی به نمایش در آورید و use case مواجهه با شرایط هشدار را ببینید.

اولویت:

اولویت میانه، پیاده‌سازی پس از قابلیت‌های اصلی.

نوبت دسترسی: در گام سوم نرم‌افزار.

کانال ارتباطی با کنش‌گر: از طریق مرورگر وب PC و اتصال اینترنتی.

کنش‌گران ثانویه: مدیر سیستم، دوربین‌ها.

کانال‌های ارتباطی با کنش‌گران ثانویه:

۱. مدیر سیستم: PC
۲. دوربین‌ها: اتصال بی سیم

مسائل باز

۵. چه سازوکارهایی، مشتری را در برابر استفاده غیر مجاز از این قابلیت توسط کارمندان شرکت محافظت می‌کنند؟
۶. آیا امنیت کافی است؟ با نفوذگری در این قابلیت، حریم خصوصی افراد به‌طور جدی به خطر می‌افتد.
۷. آیا پاسخ سیستم از طریق اینترنت با توجه به پهنای باند لازم برای دیدن خروجی دوربین‌ها قابل دستیابی هست؟
۸. آیا برای کاربرانی که پهنای باند بیشتری در اختیار دارند، سرعتی بیش از یک کادر در ثانیه برای مشاهده دوربین‌ها می‌توان ارائه کرد؟

use case بر خواسته‌های رفتاری و عملیاتی تاکید دارد و عموماً برای خواسته‌های غیر عملیاتی نامناسب است. در وضعیت‌هایی که مدل خواسته‌ها باید دارای جزئیات و دقت بالا باشد (مثلاً در سیستم‌های امنیتی بحرانی)، use case ممکن است کافی نباشد.

به‌هرحال، مدل‌سازی مبتنی بر سناریو برای اغلب وضعیت‌ها که به‌عنوان مهندس نرم‌افزار با آنها برخورد خواهید داشت، مناسب است. use case اگر خوب نوشته شده باشد، می‌تواند به‌عنوان یک ابزار مدل‌سازی، مزایای اساسی در برداشته باشد.

۶-۳ مدل‌های UML که use case را تکمیل می‌کنند

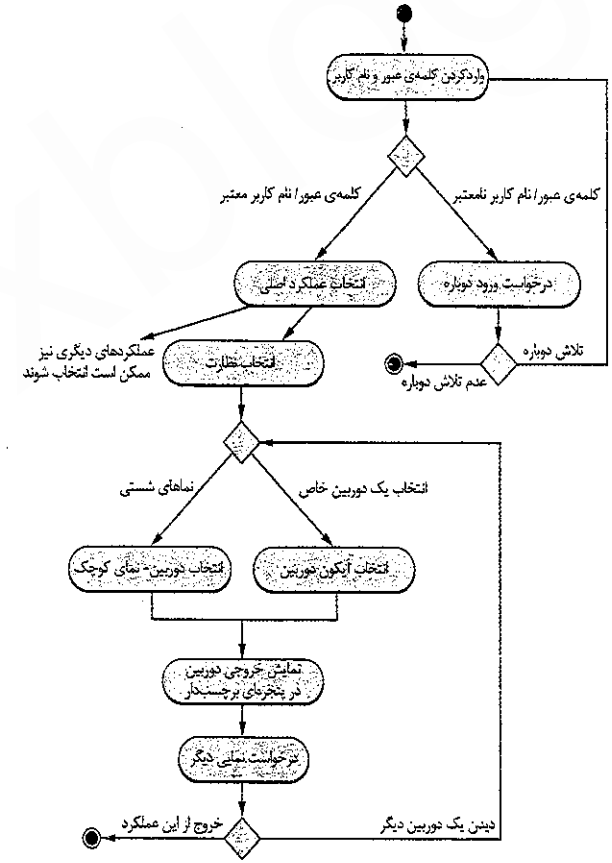
وضعیت‌های زیادی در مدل‌سازی خواسته‌ها وجود دارد که در آنها مدل‌های مبتنی بر متن- حتی مدلی به سادگی یک use case- ممکن است اطلاعات را به طرزی واضح و دقیق ارائه ندهد. در چنین مواردی، آرایه وسیعی از مدل‌های گرافیکی UML را در اختیار دارید.

۶-۳-۱ توسعه‌ی نمودار فعالیت‌ها

نمودار فعالیت‌های UML، use case را با فراهم‌ساختن نمایش گرافیکی جریان تعامل در یک سناریو مشخص، تکمیل می‌کنند. نمودار فعالیت‌ها همانند نمودار گردشی، از مستطیل‌های گوشه گرد برای نشان دادن عملکردهای سیستم، از پیکان‌ها برای نمایش جریان در سیستم، از لوزی‌های تصمیم‌گیری برای به تصویر کشیدن انشعاب‌های تصمیم‌گیری (هر پیکان خروجی از لوزی نشان‌گذاری می‌شود) و از خطوط افقی توپر برای نشان دادن فعالیت‌های موازی استفاده می‌شود. ACS-DCV use case در شکل ۶-۵ نشان داده شده است. لازم به ذکر است که نمودار فعالیت‌ها، جزئیاتی را اضافه می‌کند که

نکته‌ی کلیدی
نمودار فعالیت‌های UML، کنش‌ها و تصمیم گیری‌هایی را که در اجرای یک عملکرد رخ می‌دهند، به نمایش می‌گذارد.

مستقیماً در use case ذکر نمی‌شوند (ولی می‌توان آنها را استنباط کرد). برای مثال، کاربر ممکن است فقط مجاز باشد نام کاربری و کلمه‌ی عبور را به تعداد دفعات محدود وارد کند. این را با یک لوزی تصمیم‌گیری زیر «درخواست برای وارد کردن دوباره» می‌توان نشان داد.



شکل ۵-۶ نمودار فعالیت‌ها برای دستیابی پایش دوربین‌ها از طریق اینترنت-عملکرد مشاهده دوربین.

۶-۳-۲ نمودارهای بخش‌بندی (Swimlane)

نمودار بخش‌بندی UML شکل دیگری از نمودار فعالیت‌هاست که به کمک آن می‌توانید جریان فعالیت‌های توصیف شده در یک use case را به نمایش درآورید و در عین حال ، نشان دهید که کدام کنش‌گر (در صورتی که use case چند کنش‌گر را شامل شود) یا کلاس تحلیل (که بعداً در همین فصل بحث خواهد شد) مسؤولیت کنش توصیف شده توسط یک مستطیل فعالیت است. مسؤولیت‌ها توسط بخش‌های موازی نمایش داده می‌شوند که نمودار را به‌صورت عمودی بخش‌بندی می‌کنند. سه کلاس تحلیل(**Homeowner**، **Camera** و **Interface**) در حیطه‌ی نمودار فعالیتِ ارائه شده در شکل ۵-۶ مسؤولیت‌های مستقیم یا غیر مستقیم دارند. در شکل ۶-۶ نمودار فعالیت‌ها طوری

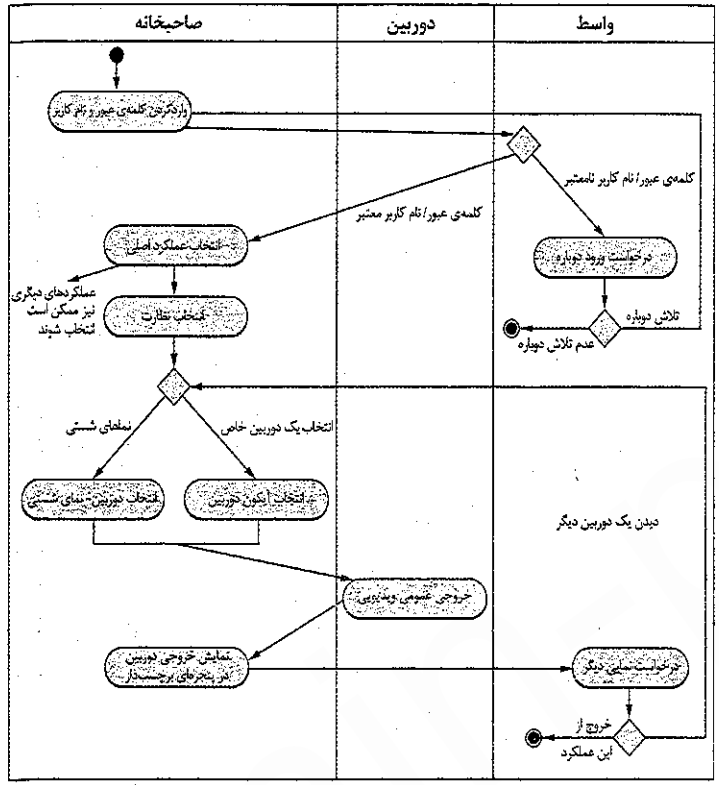
نکته‌ی کلیدی

نمودار بخش‌بندی UML، جریان کنش‌ها و تصمیم گیری‌ها را به نمایش می‌گذارد و نشان می‌دهد کدام کنش گران کدام کنش را انجام می‌دهند.

سازمان‌دهی‌شده است که فعالیت‌های مرتبط با یک کلاس تحلیل خاص، در بخش اختصاص داده شده به آن کلاس قرار گیرند. برای مثال، کلاس واسط نشان دهنده‌ی واسط کاربر از دید صاحبخانه است. در نمودار فعالیت‌ها دو پیام درخواست ذکر شده است که مسؤولیت واسط هستند- «درخواست برای ورود دوباره» و «درخواست برای نمای دیگر». این پیام‌های درخواست و تصمیم گیری‌های مرتبط با آنها در بخش واسط قرار می‌گیرند، ولی پیکان‌هایی از آن بخش به بخش **صاحبخانه** بر می‌گردند که در آن، کنش‌های صاحبخانه رخ می‌دهد.

«یک مدل خوب تفکر ما را هدایت می‌کند و مدل بد آن را منحرف می‌سازد.»

برایان ماریک



شکل ۶-۶ نمودار بخش‌بندی برای دستیابی به پایش دوربین‌ها از طریق اینترنت- عملکرد مشاهده دوربین‌ها.

use caseها، به همراه نمودارهای فعالیت و نمودارهای بخش بندی، به صورت روال‌ها عمل می‌کنند. این ابزارها شیوه‌ی فراخوانی عملکردهای خاص (یا سایر مراحل روالی) توسط کنش‌گران، برای برآورده‌شدن خواسته‌های سیستم را به نمایش در می‌آورند، ولی روالی بودن نمای خواسته‌ها تنها یک بُعدِ سیستم را نشان می‌دهد. در بخش ۴-۶ فضای اطلاعاتی و چگونگی به نمایش درآوردن خواسته‌های داده‌ای را بررسی خواهیم کرد.

۶-۴ مفاهیم مدل‌سازی داده‌ها

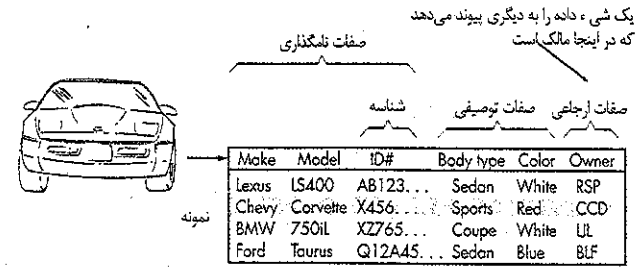
اگر خواسته‌های نرم‌افزار شامل ایجاد، بسط یا برقراری ارتباط با یک بانک اطلاعاتی شود یا مستلزم ساخت و دستکاری ساختار فایل‌های پیچیده باشد، تیم مهندسی نرم‌افزار ممکن است به‌عنوان بخشی از مدل‌سازی خواسته‌ها تصمیم به ایجاد *مدل داده‌ای* بگیرد. مهندس نرم‌افزار یا تحلیل‌گر، همه‌ی اشیای داده را که در سیستم پردازش می‌شوند، روابط میان اشیای داده و سایر اطلاعات مرتبط با این روابط را تعیین می‌کند. *نمودار موجودیت- ارتباط* (ERD)[1] به این مسائل می‌پردازد و همه‌ی اشیای داده را که در یک برنامه‌ی کاربردی، وارد، ذخیره، تبدیل و تولید می‌شوند، به نمایش می‌گذارند.

۶-۴-۱ اشیای داده

شیء داده نمایشی از اطلاعات مرکب است که باید نرم‌افزار آنها را بفهمد. منظور از *اطلاعات مرکب*، چیزی است که دارای چند صفت یا خاصیت متفاوت باشد. بنابراین، پهنا (که تنها یک مقدار دارد)، شیء داده معتبری نیست، ولی **dimensions** (که شامل درازا، پهنا و ارتفاع می‌شود) به‌عنوان یک شیء قابل تعریف است.

یک شیء داده می‌تواند نهادی خارجی (مثلاً هر چیزی که اطلاعات را تولید یا مصرف کند)؛ یک چیز (مثلاً گزارش یا صفحه نمایش)، یک رخداد (مثلاً تماس تلفنی) یا رویداد (مثلاً هشدار)، یک نقش (مثلاً فروشنده)، یک واحد سازمانی (مثلاً بخش حسابداری)، یک مکان (مثلاً انبار)، یا یک ساختار (مثلاً فایل) باشد. برای مثال، **car** یا **person** را می‌توان به‌عنوان یک شیء داده در نظر گرفت از این لحاظ که هر دو آنها را می‌توان بر حسب مجموعه‌ای از صفات تعریف کرد. توصیف شیء داده خود آن شیء داده و همه‌ی صفات آن را در بر می‌گیرد.

شیء داده تنها داده‌ها را پنهان‌سازی می‌کند- در داخل یک شیء داده هیچ آدرسی برای عملیات قابل انجام روی داده‌ها وجود ندارد.[2] بنابراین، شیء داده را می‌توان به‌صورت جدول شکل ۶-۷ به نمایش گذاشت. عناوین ستون‌های جدول، نشان‌گر صفات شیء هستند. در این مورد، شیء خودرو بر حسب مارک، مدل، شماره پلاک، نوع بدنه، رنگ و صاحب خودرو تعریف می‌شود. متن جدول شامل چند نمونه‌ی خاص از شیء داده است. برای مثال، شورلت کوروت، نمونه‌ای از شیء داده **car** است.



شکل ۶-۷ نمایش جدول‌بندی شده‌ی اشیای داده.

مرجع وب

اطلاعات مفیدی درباره مدل‌سازی داده‌ها را در www.datamodel.org می‌توان یافت.

یک شیء داده چگونه خود را در حیطه‌ی یک برنامه‌ی کاربردی، اعلام می‌کند؟

نکته‌ی کلیدی

یک شیء داده نمایشی است از هر گونه اطلاعات مرکب که توسط نرم‌افزار پردازش می‌شود.

[1] Entity-Relationship Diagram

[2] همین وجه تمایز است که شیء داده‌ای را از کلاس یا شیء تعریف شده به‌عنوان بخشی از رویکرد شیءگرا متمایز می‌سازد (پیوست ۲).

۶-۴-۲ صفات داده‌ها

صفات داده‌ها، خواص یک شیء داده را تعریف می‌کنند و یکی از سه خصوصیت مهم را به خود می‌گیرند. از آنها می‌توان برای (۱) نامگذاری نمونه‌ای از شیء داده‌ای، (۲) توصیف نمونه یا (۳) ارجاع به نمونه‌ای دیگر در جدولی دیگر استفاده کرد. به‌علاوه، یک یا چند صفت را باید به‌عنوان شناسه تعریف کرد- یعنی هنگامی که بخواهیم نمونه‌ای از شیء داده را بیابیم، صفت شناسه به‌عنوان «کلید» عمل می‌کند. در برخی موارد، مقادیر مربوط به شناسه(ها) منحصر به فردند هر چند که این ضروری نیست. در مورد شیء داده **car**، یک شناسه منطقی می‌تواند شماره پلاک باشد.

مجموعه صفات مناسب برای یک شیء داده مفروض از طریق شناخت حیطه‌ی مسأله قابل تعیین است. صفات مربوط به خودرو ممکن است به خوبی برای یک برنامه مورد استفاده توسط بخش وسائط نقلیه موتوری عمل کند، ولی همین صفات ممکن است برای یک شرکت خودروسازی که نیاز به نرم‌افزار کنترل تولید دارد، بی‌فایده باشد. در این مورد اخیر، صفات مربوط به شیء خودرو می‌توانند شامل *شماره پلاک*، *نوع بدنه* و *رنگ* باشد، ولی صفات دیگری (مثل *کد داخلی*، *نوع رانش خودرو*، *نوع انتقال نیرو*) را نیز باید اضافه کرد. شیء خودرو در حیطه‌ی کنترل تولید معنا پیدا می‌کند.

اطلاعات

اشیای داده و کلاس‌های شیء گرا- آیا اینها یکسانند؟

هنگام بحث درباره اشیای داده یک پرسش رایج پیش می‌آید: آیا اشیای داده همان کلاس‌های شیءگرا هستند؟[1] پاسخ، منفی است. شیء داده یک موجودیت داده‌ای مرکب را تعریف می‌کند؛ یعنی شامل مجموعه‌ای از موجودیت‌های داده‌های منفرد (صفات) می‌شود و به این مجموعه یک نام اختصاص می‌دهد (که نام شیء داده است).

کلاس شیءگرا، صفات داده‌ها را پنهان‌سازی می‌کند، ولی شامل عملیات (متدهای) دستکاری آن داده‌ها نیز می‌شود. به‌علاوه، از تعریف کلاس‌ها چنین بر می‌آید که زیر ساختی فراگیر و جامع در رویکرد مهندسی نرم‌افزار شیءگرا باشند. کلاس‌ها از طریق پیام‌ها با هم ارتباط برقرار می‌کنند، می‌توان آنها را در یک سلسله مراتب سازمان دهی کرد و برای اشیایی که نمونه‌ای از یک کلاس هستند، خصوصیات وراثتی به همراه دارند.

۶-۴-۳ ارتباطات

اشیای داده به طرق گوناگون به هم متصل می‌شوند. دو شیء داده **person** و **car** را در نظر بگیرید. این اشیا را می‌توان با به‌کارگیری نمادگذاری ساده‌ی شکل ۶-۸(الف) به نمایش گذاشت. میان **person** و **car** اتصال برقرار شده است، چون این دو شیء با هم ارتباط دارند، ولی این ارتباطات چیستند؟ برای پاسخ گفتن به این پرسش، باید در حیطه‌ی نرم‌افزاری که قرار است ساخته شود، نقش اشخاص (در این مورد مالک) و خودروها را بدانید. می‌توانید مجموعه‌ای از روابط جفتی میان اشیا تعیین کنید که این ارتباط‌ها را مشخص کنند. برای مثال:

- *شخصی صاحب* یک خودرو است.
- *شخصی مجوز رانندگی* خودرو را دارد.

نکته‌ی کلیدی

صفات، یک شیء داده را نام می‌برند، خصوصیات آن را توصیف می‌کنند و در برخی موارد، به شیء دیگر ارجاع می‌دهند.

مرجع وب

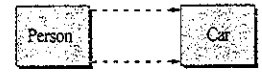
مفهومی به نام «بهنجارش» برای علاقمندان به مدل‌سازی داده‌ها اهمیت دارد. معرفی مفیدی از این مفهوم را در www.datamodel.org می‌توانید بیابید.

نکته‌ی کلیدی

رابطه‌ها نشان‌گر شیوه‌ی اتصال اشیای داده به یکدیگرند.

روابط صاحب بودن و *داشتن مجوز رانندگی*، ارتباط میان شخص و خودرو را تعیین می‌کنند. در شکل ۸-۶ب، این روابط جفتی میان اشیا به‌صورت گرافیکی نمایش داده شده است. پیکان‌های شکل ۸-۶ب، اطلاعات مهمی درباره جهت پذیری واسط در اختیار قرار می‌دهند و غالباً از ابهام و سوء تعبیر می‌کاهند.

(الف) یک شرط پایه میان اشیای داده



(ب) روابط میان اشیای داده

شکل ۸-۶ روابط میان اشیای داده.

ابزارهای نرم‌افزاری

مدل‌سازی داده‌ها

هدف: ابزارهای مدل‌سازی داده‌ها، توانایی نمایش اشیای داده، خصوصیات آنها و روابط میان آنها را در اختیار مهندس نرم‌افزار می‌گذارد. ابزارهای مدل‌سازی داده‌ها، که عمدتاً برای کاربردهای مربوط به بانک‌های اطلاعاتی بزرگ مورد استفاده قرار می‌گیرند، ایجاد نمودارهای ارتباط موجودیت‌ها، دیکشنری‌های اشیای داده، و مدل‌های مرتبط را خودکار می‌سازند.

مکانیک: ابزارهای این گروه، کاربر را در توصیف اشیای داده و روابط میان آنها یاری می‌دهند. در برخی موارد، این ابزارها از نمادگذاری ERD استفاده می‌کنند. در موارد دیگر، این ابزارها، روابط را با استفاده از سازوکاری دیگر مدل‌سازی می‌کنند. ابزارهای این گروه غالباً به‌عنوان بخشی از طراحی بانک اطلاعاتی به‌کار می‌روند و ایجاد یک مدل بانک اطلاعاتی را با تولید طرحی از بانک اطلاعاتی برای سیستم‌های مدیریت بانک اطلاعاتی (DBMS) میسر می‌سازد.

ابزارهای نمونه

AllFusion ERWin، که توسط Computer Associates توسعه یافته است (**www3.ca.com**)، به طراحی اشیای داده، ساختار مناسب و عناصر کلیدی برای بانک‌های اطلاعاتی کمک می‌کند.

ER/Studio که توسط Embarcadero Software توسعه یافته است (**www.embarcadero.com**) مدل‌سازی موجودیت-ارتباط را پشتیبانی می‌کند.

Oracle Designer، که توسط Oracle Systems توسعه یافته است (**www.oracle.com**) «فرایندهای تجاری، موجودیت‌های داده‌ای و روابط میان آنها را [که] به طراحی تبدیل می‌شوند و از آنها برنامه‌ها و بانک‌های اطلاعاتی کامل ایجاد می‌شوند، مدل‌سازی می‌کنند.»

Visible Analyst، که توسط Visible Systems توسعه یافته است (**www.visible.com**)، انواع عملیات مدل‌سازی تحلیل، از جمله مدل‌سازی داده‌ها را پشتیبانی می‌کند.

۶-۵ مدل‌سازی مبتنی بر کلاس‌ها

مدل‌سازی مبتنی بر کلاس‌ها، اشیایی را که سیستم دستکاری می‌کند، عملیاتی (متد یا سرویس) که در مورد اشیا به‌کار می‌رود تا این دستکاری‌ها انجام شوند، روابط میان اشیاء (که برخی از آنها سلسله مراتبی هستند) و همکاری‌هایی را که بین کلاس‌ها رخ می‌دهند، به نمایش می‌گذارند. عناصر یک مدل مبتنی بر کلاس‌ها شامل کلاس‌ها و اشیا، صفات، عملیات، مدل‌های همکاری- مسؤولیت کلاس‌ها (CRC)، نمودارهای همکاری و پکیج‌ها می‌شود. در بخش‌هایی که به‌دنبال خواهد آمد، یک سری دستورالعمل‌های غیر رسمی ارائه خواهد شد که به شناسایی و نمایش آنها کمک خواهد کرد.

۶-۵-۱ شناسایی کلاس‌های تحلیل

اگر نگاهی به اطراف یک اتاق بیندازید، مجموعه‌ای از اشیای فیزیکی وجود دارد که به راحتی می‌توانید آنها را شناسایی، طبقه‌بندی و تعریف کنید (بر حسب صفات و عملیات آنها)، ولی هنگامی که در فضای مسأله‌ی یک نرم‌افزار به اطراف نگاه می‌کنید، شناسایی کلاس‌ها (و اشیاء) ممکن است دشوارتر باشد.

«مسأله‌ی واقعاً دشوار، کشف اشیا [کلاس‌های] درست در وهله‌ی نخست است.»

کارل آرجیلا

می‌توانیم شناسایی کلاس‌ها را با بررسی سناریوهای کاربری آغاز کنیم که به‌عنوان بخشی از مدل خواسته‌ها توسعه می‌یابند و use caseهای تهیه شده برای سیستم را «تجزیه‌ی گرامری» کنیم [Abb83]. کلاس‌ها با خط کشیدن زیر اسم‌ها یا عبارت‌های اسمی و وارد کردن آن در یک جدول ساده تعیین می‌شوند. باید به اسم‌های مترادف توجه کرد. اگر کلاس (اسم) برای پیاده‌سازی یک راهکار مورد نیاز باشد، در آن صورت بخشی از فضای راهکار است؛ در غیر این صورت، اگر کلاسی تنها برای توصیف یک راهکار لازم باشد، بخشی از فضای مسأله است.

کلاس‌های تحلیل چگونه خود را به‌عنوان عناصر فضای راهکار اعلام می‌کنند؟

ولی هنگامی که همه‌ی اسم‌ها را جدا کردیم باید در جستجوی چه باشیم؟ *کلاس‌های تحلیل*، خود را به یکی از طرق زیر نشان می‌دهند:

- *موجودیت‌های خارجی* (مانند سایر سیستم‌ها، دستگاه‌ها، افراد) که اطلاعات مورد استفاده یک سیستم کامپیوتری را تولید یا مصرف می‌کنند.
- چیزهایی (مانند *گزارش‌ها*، صفحه نمایش‌ها، نامه‌ها، سیگنال‌ها) که بخشی از دامنه‌ی اطلاعاتی مسأله‌اند.
- رخدادها یا رویدادهایی (مانند انتقال یک خاصیت یا کامل‌شدن یک سری حرکات روبات) که در حیطه‌ی عملیاتی سیستم به وقوع می‌پیوندند.
- *نقش‌هایی* (مانند مدیر، مهندس، فروشنده) که توسط افراد در حال تعامل با سیستم ایفا می‌شود.
- *واحدهای سازمانی* (مانند بخش، گروه، تیم) که به‌کاربردی خاص مربوط می‌شوند.
- *مکان‌هایی* (مانند قسمت تولید یا بارانداز) که حیطه‌ی مسأله و عملکرد کلی سیستم را تعیین می‌کند.
- *ساختارهایی* (مانند حس‌گرها، وسایل چهار چرخ یا کامپیوترها) که کلاسی از اشیا یا کلاس‌های مرتبطی از اشیاء را تعریف می‌کنند.

این گروه‌بندی یکی از چند نوع گروه‌بندی پیشنهاد شده در متون است.[1] برای مثال، باد [Bud96] طبقه‌بندی دیگری برای کلاس‌ها پیشنهاد می‌کند که شامل *تولید کنندگان* (منابع) و *مصرف کنندگان* (چاهک‌های) داده‌ها، *مدیران داده‌ها*، *کلاس‌های مشاهده‌ای* و *کلاس‌های کمک رسان* می‌شوند.

همچنین شایان ذکر است که بدانیم چه چیزهایی کلاس یا شیء نیستند. به‌طور کلی، یک کلاس هرگز نباید دارای «نام روالی الزام‌آور» باشد [Cas89]. برای مثال، اگر سازندگان نرم‌افزار یک سیستم تصویربرداری پزشکی، شیء‌ای با نام **InvertImage** یا حتی **ImageInversion** (وارونه کردن تصویر) تعریف کرده باشند، مرتکب اشتباهی ظریف شده‌اند. **Image** (تصویر) به‌دست آمده از نرم‌افزار بدون شک می‌تواند یک شیء باشد (چیزی است که بخشی از دامنه‌ی اطلاعاتی است). وارونه‌کردن تصویر،عملی است که برای شیء **Image** تعریف می‌شود، ولی به‌عنوان یک کلاس مجزا برای دلالت

[1] یک گروه‌بندی مهم دیگر که در آن، کلاس‌های کنترل‌گر، موجودیت و مرزی تعریف می‌شود، در بخش ۶-۵-۴ بحث خواهد شد.

ضمنی بر «وارونه کردن تصویر» تعریف نمی شود. چنان که کشمن [Cas89] می‌گوید: «مقصود از شیءگرایی، بسته‌بندی داده‌ها و عملیاتی است که روی آنها انجام می‌شود، ولی جدایی میان آنها همچنان باید حفظ شود.»

برای اینکه نشان دهیم کلاس‌های تحلیل را چگونه می‌توان طی مراحل اولیه‌ی مدل‌سازی تعریف کرد، تجزیه‌ی گرامری روایت پردازش[1] قابلیت امنیتی در محصول *SafeHome* در نظر بگیرید (زیر اسم‌ها خط کشیده می‌شود، فعل‌ها به‌صورت ایتالیک نشان داده می‌شوند).

قابلیت امنیت در محصول *SafeHome* صاحبخانه را *قادر می‌سازد* که سیستم امنیتی را پس از *نصب کردن*، *پیکربندی کند*، همه‌ی حس‌گرهایی را که به سیستم امنیتی *متصل شده‌اند*، *پایش کند* و با صاحبخانه از طریق اینترنت، PC یا پانل کنترلی، *تعامل کند*.

در مدتی که *نصب انجام می‌شود*، از PC *SafeHome* برای *برنامه‌ریزی* و *پیکربندی* سیستم استفاده می‌شود. به هر حس‌گر یک عدد و نوع نسبت داده می‌شود، یک کلمه‌ی عبوری اصلی برای *فعال کردن* و *غیر فعال کردن* سیستم برنامه‌ریزی می‌شود و شماره تلفن(هایی) *وارد می‌شوند* تا در صورت رخ دادن یک رویداد حس‌گری این شماره‌ها *گرفته شوند*.

هنگامی که یک رویداد حس‌گری *تشخیص داده شد*، نرم‌افزار یک آژیر صوتی را به صدا *در می‌آورد* که به سیستم متصل است. پس از مشخص‌شدن زمان تأخیر توسط صاحبخانه در طول فعالیت‌های پیکربندی سیستم، نرم‌افزار شماره تلفن یک سرویس پایشی را می گیرد، اطلاعات مربوط به مکان را *ارائه می‌دهد*، و ماهیت رویداد تشخیص داده شده را *گزارش می‌کند*. این شماره تلفن هر ۲۰ ثانیه یک بار *از نو گرفته می‌شود* تا اینکه تماس تلفنی *برقرار شود*.

صاحبخانه، اطلاعات امنیتی را از طریق یک پانل کنترلی، PC یا مرورگر که در مجموع واسط گفته می‌شود، *دریافت می‌کند*. این واسط، پیام‌های *درخواستی* و اطلاعات وضعیتی را روی پانل کنترلی، PC یا پنجره مرورگر به نمایش می گذارد. تعامل صاحبخانه به شکل زیر خواهد بود...

اندرز

تجزیه‌ی گرامری، ضد خطا نیست، ولی می‌تواند نقطه پرسی عالی در اختیارتان بگذارد، اگر سعی در تعریف اشیای داده و تبدیلاتی دارید که روی آنها عمل می کنند.

با استخراج اسم‌ها، چند کلاس بالقوه می‌توان پیشنهاد کرد:

کلاس بالقوه	طبقه‌بندی کلی
صاحبخانه	نقش یا موجودیت خارجی
حس گر	نهاد خارجی
پانل کنترلی	نهاد خارجی
نصب	رویداد
سیستم (سیستم امنیتی)	چیز (thing)
شماره، نوع	غیر شیء، صفات حس گر
کلمه‌ی عبور اصلی	چیز
شماره تلفن	چیز
رویداد حس گری	رخداد
آژیر صوتی	نهاد خارجی
سرویس پایشی	واحد سازمانی یا موجودیت خارجی

[1] روایت پردازش، سبکی مشابه با use case دارد ولی هدف آن قدری متفاوت است. روایت پردازش، توصیفی کلی از قابلیت در حال توسعه ارائه می‌دهد. سناریویی نیست که از دیدگاه تنها یک کنش‌گر نوشته شده باشد. ولی لازم به توجه است که تجزیه گرامری را برای هر use case تهیه شده در جمع آوری خواسته‌ها نیز می‌توان به کاربرد.

این فهرست چندان ادامه می‌یابد که همه‌ی اسم‌های موجود در روایت پردازش در نظر گرفته شوند. توجه دارید که هر دارایه از این فهرست را یک شیء بالقوه می‌خوانیم. پیش از تصمیم‌گیری نهایی باید هر کدام را بیشتر در نظر بگیریم.

کود و یوردان [Coa91] شش خصوصیات انتخابی پیشنهاد می‌کند که در پرداختن به هر کدام از کلاس‌های بالقوه برای لحاظ کردن در مدل تحلیل، باید از آنها استفاده کرد:

چگونه تعیین کنم که آیا یک کلاس بالقوه واقعاً یک کلاس تحلیل است؟

۱. *اطلاعات نگهداری شده*. کلاس بالقوه، تنها در صورتی در تحلیل مفید واقع خواهد شد که به خاطر سپردن اطلاعات مربوط به آن، برای عملکرد سیستم ضروری باشد.

۲. *سرویس‌های لازم*. کلاس بالقوه باید دارای مجموعه‌ای از عملیات قابل شناسایی باشد که بتوانند مقدار صفات آن را به طریقی تغییر دهند.

۳. *صفات چندگانه*. طی تحلیل خواسته‌ها، اطلاعات «اصلی» را باید کانون توجه قرار داد؛ کلاسی با یک صفت به تنهایی ممکن است در طراحی واقعاً مفید واقع شود، ولی طی فعالیت تحلیل احتمالاً بهتر است در قالب صفتی از یک کلاس دیگر نمایش داده شود.

۴. *صفات مشترک*. مجموعه‌ای از صفات که برای کلاس بالقوه، قابل تعریف است و این صفات در همه‌ی نمونه‌های کلاس مصداق دارند.

۵. *عملیات مشترک*. مجموعه‌ای از عملیات‌ها که برای کلاس بالقوه، قابل تعریف است و این عملیات‌ها در همه‌ی نمونه‌های کلاس مصداق دارند.

۶. *خواسته‌های اساسی*. موجودیت‌های خارجی که در فضای مسأله ظاهر می‌شوند و اطلاعات ضروری جهت عملکرد هر راهکار برای سیستم را تولید یا مصرف می‌کنند، نیز در مدل خواسته‌ها همواره به‌عنوان کلاس تعریف می‌شوند.

برای اینکه یک شیء بالقوه به‌عنوان کلاسی قانونی در مدل خواسته‌ها در نظر گرفته شود، باید همه‌ی این خصوصیات (یا تقریباً همه‌ی آنها) را داشته باشد. تصمیم‌گیری برای لحاظ کردن کلاس‌های بالقوه در مدل تحلیل تا حدی ذهنی است و طی ارزیابی‌های بعدی ممکن است شیء‌ای حذف یا دوباره انتخاب شود، ولی نخستین مرحله در مدل‌سازی مبتنی بر کلاس‌ها، تعریف کلاس هاست و در این خصوص تصمیم‌گیری‌هایی (حتی آنها که ذهنی هستند) باید انجام شود. با در نظر داشتن این نکته، باید خصوصیات انتخاب را برای فهرست کلاس‌های بالقوه *SafeHome* به‌کار بگیرید:

«کلاس‌ها نقلا می‌کنند، بعضی پیروز می‌شوند و بقیه حذف می‌شوند.»

مائو زدونگ

کلاس بالقوه	عدد مشخصه‌ای که کاربرد دارد
صاحبخانه	رد: ۱ و ۲ درست نیست هرچند ۶ درست است
حس گر	قبول: همه درست است
پانل کنترلی	قبول: همه درست است
نصب	رد
سیستم (سیستم امنیتی)	قبول: همه درست است
شماره، نوع	رد: ۳ درست نیست؛ صفات حس گر
کلمه‌ی عبور اصلی	رد: ۳ درست نیست
شماره تلفن	رد: ۳ درست نیست
رویداد حس گری	قبول: همه درست است
آژیر صوتی	قبول: ۲، ۳، ۴، ۵، ۶ درست هستند
سرویس پایشی	رد: ۱ و ۲ درست نیست هرچند ۶ درست است

لازم به ذکر است که (۱) فهرست بالا شامل همه‌ی موارد نمی شود و برای کامل‌شدن مدل بالا کلاس‌های دیگری به آن افزوده شود؛ (۲) برخی کلاس‌های بالقوه رد شده به‌عنوان صفات کلاس‌های پذیرفته شده مطرح می‌شوند (مثلاً *شماره* و *نوع*، صفاتی از حس‌گر هستند و *کلمه‌ی عبور اصلی* و *شماره تلفن* ممکن است صفاتی از سیستم باشند)؛ (۳) بیان‌های متفاوتی از مسأله ممکن است به تصمیم گیری‌های متفاوتی برای «پذیرش یا رد» منجر شوند (مثلاً اگر هر صاحبخانه یک کلمه‌ی عبور فردی می‌داشت یا هویت او با تشخیص صدایش تأیید می‌شد، صاحبخانه، خصوصیات ۱ و ۲ را دارا می‌شد و به‌عنوان کلاس پذیرفته می‌شد).

۶-۵-۲ مشخص کردن صفات

صفات، کلاس انتخاب شده برای گنجاندن در مدل خواسته‌ها را توصیف می‌کنند. در اصل، همین صفات هستند که کلاس را تعریف می‌کنند- که مشخص می‌کنند منظور از کلاس در حیطه‌ی فضای مسأله چیست. برای مثال، اگر قرار بود سیستمی بسازیم که آمار بازی بیس بال را برای بازیگران حرفه‌ای پایش کند، صفات کلاس **Player** با صفات همان کلاس در صورت استفاده در حیطه‌ی سیستم و دستمزد در بیس بال حرفه‌ای کاملاً تفاوت می‌داشت. در اولی، صفاتی نظیر *نام*، *موقعیت*، *میانگین ضربه زنی*، *درصد حضور در میدان*، *سال‌های بازی* و *تعداد بازی‌های حضور یافته* ممکن است مرتبط به نظر برسند. برای دومی، برخی از این صفات مرتبط خواهند بود، ولی برخی دیگر جای خود را به صفاتی مثل *میانگین دستمزد*، *آدرس پستی* و *گزینه‌های حقوقی انتخاب شده* می‌دهند.

نکته‌ی کلیدی

صفات، مجموعه‌ای از اشیای داده هستند که یک کلاس را به‌طور کامل در حیطه‌ی مسأله تعریف می‌کنند.

برای توسعه‌ی مجموعه‌ای با معنی از صفات برای یک کلاس تحلیل، باید هر کدام از use caseها را مطالعه کنید و آن «چیزهایی» را انتخاب کنید که به‌طور منطقی به کلاس «تعلق» دارند. به‌علاوه، برای هر کلاس باید به پرسش زیر پاسخ داد: «کدام اقلام داده‌ای (مرکب و/یا پایه) به‌طور کامل این کلاس را در حیطه‌ی مسأله مورد نظر تعریف می‌کنند؟»

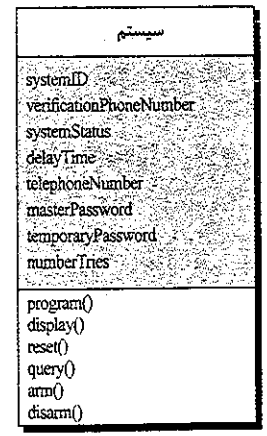
جهت روشن‌شدن مطلب، کلاس **System** را که برای *SafeHome* تعریف شده است، در نظر می‌گیریم. صاحبخانه می‌تواند قابلیت امنیتی را پیکربندی کند تا اطلاعات حس‌گر، اطلاعات پاسخ آژیر، اطلاعات فعال سازی/غیر فعال‌سازی، اطلاعات احراز هویت و غیره را منعکس سازد. می‌توانیم این اقلام داده‌ای مرکب را به شیوه‌ی زیر نمایش دهیم:

identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

هرکدام از اقلام داده‌ای واقع در طرف راست علامت تساوی را می‌توان باز هم تا یک سطح پایه‌ای تعریف کرد، ولی برای اهدافی که ما در پی آن هستیم، فهرستی منطقی از صفات برای کلاس سیستم تشکیل می‌دهند (بخش هاشور خورده از شکل ۹-۶).



شکل ۹-۶ نمودار کلاس‌ها برای کلاس System

حس‌گرها بخشی از کل سیستم *SafeHome* هستند و با این حال به‌عنوان اقلام داده‌ای یا صفات در شکل ۹-۶ فهرست نشده‌اند. حس‌گر قبلاً به‌عنوان یک کلاس تعریف شده است و اشیای حس‌گر با کلاس سیستم مرتبط خواهند بود. به‌طور کلی، از تعریف یک قلم به‌عنوان صفت پرهیز می‌کنیم اگر بیش از یک قلم قرار باشد با کلاس مرتبط شود.

SafeHome

مدل کلاس‌ها

صحنه: اتاقک اد؛ در شروع مدل‌سازی خواسته‌ها.

نقش‌آفرینان: جیمی، وینود و اد-همه‌ی اعضای تیم مهندسی نرم‌افزار *SafeHome*

گفتگو:

[اد روی استخراج کلاس‌ها از الگوی use case برای **ACS-DCV** (که قبلاً در این فصل ارائه شد) کار کرده است و کلاس‌های استخراج شده را به همکاران ارائه می‌دهد.]

اد: خلاصه، وقتی که صاحبخانه می‌خواهد یک دوربین انتخاب کند، باید آن را از یک نقشه ساختمانی انتخاب کند. من یک کلاس با نام نقشه ساختمان تعریف کرده ام. این هم نمودارش (آنها شکل ۱۰-۶ را نگاه می‌کنند.)

جیمی: پس **FloorPlan** شیءای است که با دیوارها، درها، پنجره‌ها و دوربین‌ها در ارتباط است. این خطوط نشان دار به همین معنی هستند، نه؟

اد: بله. به آنها «همبستگی» می‌گویند. ارتباط یک کلاس با کلاس دیگر طبق همبستگی‌هایی که نشان داده‌ام به هم مرتبط می‌شوند. [همبستگی‌ها در بخش ۵-۵-۶ بحث خواهند شد.]

وینود: پس نقشه ساختمان واقعی از دیوارها ساخته می‌شود و جاوی دوربین‌ها و حس‌گرهایی است که روی آن دیوارها قرار داده می‌شوند. نقشه ساختمان از کجا می‌داند که این اشیا را کجا باید بگذارد؟

اد: این را نمی‌داند. این، کار کلاس‌های دیگر است. صفات تحت کلاس **قطعه دیوار** را ببینید؛ این کلاس برای ساختن دیوارها به‌کار می‌رود. قطعه دیوار یک مختصات شروع و پایان دارد و عملیات ()draw بقیه کارها را انجام می‌دهد.

جیمی: و برای پنجره‌ها و درها هم همین وضعیت را داریم. ظاهراً دوربین‌ها صفات بیشتری دارند.

اد: بله، کاری کردم که اطلاعات مربوط به زوم و زاویه را هم شامل بشوند.

وینود: من یک سؤال دارم. چرا دوربین‌ها شماره شناسایی دارند، ولی بقیه ندارند؟ می‌بینم که یک صفت به نام *nextWall* داری. **WallSegment** از کجا بداند که دیوار بعدی چیست؟

اد: سؤال خوبی است، ولی همان طور که می‌گویند، این یک مسأله‌ی طراحی است و من هم آن را به بعد موکول.

جیمی: یک لحظه صبر کن- قول می‌دهم که قبلاً فکرش را کرده‌ای.

اد (محجوبانه لبخند می‌زند): درست است؛ از یک ساختار فهرستی استفاده می‌کنم که موقع طراحی، آن را مدل‌سازی می‌کنم. اگر تو درباره جداسازی تحلیل و طراحی تعصب داری، سطح جزئیاتی که اینجا دارم ممکن است ایجاد سوء ظن کند.

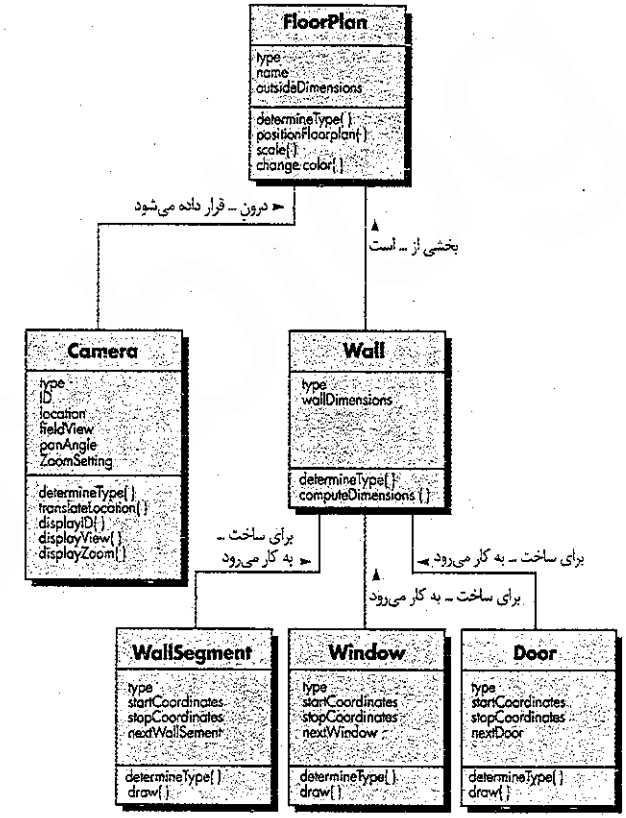
جیمی: به نظر من که عالی است، ولی چند تا سؤال دیگر هم دارم.

(جیمی پرسش‌هایی مطرح می‌کند که به اصطلاحات جزئی منجر می‌شود)

وینود: برای هر کدام از اشیا، کارهای CRC داری؟ اگر داری باید از طریق آنها با هم نقش بازی کنیم تا ببینیم چیزی از قلم نیفتاده باشد.

اد: خیلی از بابت نحوه انجام این کار مطمئن نیستم.

وینود: کار زیاد سختی نیست و واقعاً ارزش‌اش را دارد. به شما نشان می‌دهم.



شکل ۱۰-۶ نمودار کلاس‌ها برای FloorPlan.

۳-۵-۶ تعریف عملیات‌ها

عملیات‌ها، رفتار شیء را تعریف می‌کنند. گرچه انواع بسیار متفاوتی از عملیات وجود دارد، آنها را معمولاً در چهار گروه گسترده تقسیم می‌کنند: (۱) عملیاتی که داده‌ها را به طریقی دستکاری می‌کنند (مثل اضافه کردن، حذف کردن، فرمت‌بندی دوباره و انتخاب کردن)، (۲) عملیات‌هایی که محاسبه انجام می‌دهند، (۳) عملیات‌هایی که درباره‌ی حالت یک شیء تحقیق می‌کنند و (۴) عملیات‌هایی که یک شیء را برای وقوع یک رویداد کنترل کننده پایش می‌کنند. این قابلیت‌ها با عمل کردن روی صفات و/یا همبستگی‌ها (associations) قابل دستیابی خواهند بود (بخش ۵-۵-۶). بنابراین، عملیات باید از ماهیت همبستگی‌ها و صفات کلاس «آگاهی» داشته باشد.

به‌عنوان اولین دور تکرار در به‌دست آوردن مجموعه عملیات‌های یک کلاس تحلیل، می‌توانید دوباره یک روایت پردازش (یا use case) را مطالعه کنید و عملیاتی را انتخاب کنید که به‌طور منطقی به کلاس تعلق دارند. برای نیل به این مقصود، تجزیه‌ی گرامری دوباره مطالعه می‌شود و این بار، افعال جداسازی می‌شوند. برخی از این فعل‌ها عملیات قانونی بوده می‌توان به راحتی آنها را به کلاسی مشخص ربط داد. برای مثال، از روایت پردازشی که قبلاً در همین فصل ارائه شد، مشاهده می‌کنیم که

اندرز

هنگامی که عملیات‌های مربوط یک کلاس تحلیل را تعریف می‌کنید، به جای رفتارهای لازم برای پیاده سازی، توجه خود را به رفتارهای مسأله‌گرا معطوف کنید.

«به حس‌گر یک شماره *نوع نسبت داده می‌شود.*» یا «یک کلمه‌ی عبور اصلی برای *فعال کردن* و *غیر فعال کردن* سیستم، *برنامه‌ریزی می‌شود.*» این عبارت‌ها چند مورد را نشان می‌دهند:

- این که عملیاتی با عنوان *assign( )* با کلاس **Sensor** در ارتباط است.
- این که عملیاتی با عنوان *program( )* با کلاس **System** در ارتباط است.
- این که *arm( )* و *disarm( )* عملیات‌هایی هستند که برای کلاس **System** کاربرد دارند.

با بررسی بیشتر، این احتمال وجود دارد که عملیات *program( )* را به چند عملیات فرعی مشخص‌تر تقسیم کنیم که برای پیکربندی سیستم مورد نیازند. برای مثال، *program( )* به معنای مشخص کردن شماره تلفن‌ها، پیکربندی خصوصیات سیستم (مثلاً ایجاد جدول حس‌گرها، وارد کردن خصوصیات آژیر) و وارد کردن کلمه(ها)ی عبوراست، ولی فعلاً *program( )* را به‌عنوان یک عملیات واحد، مشخص می‌کنیم.

علاوه بر تجزیه‌ی گرامری می‌توانید با در نظر گرفتن ارتباطاتی که میان اشیا رخ می‌دهد، دید بیشتری از سایر عملیات به‌دست آورید. اشیا با تبادل پیام به یکدیگر با هم ارتباط برقرار می‌کنند. پیش از ادامه‌ی تعیین مشخصات عملیات‌ها، این موضوع را قدری بیشتر بررسی می‌کنیم.

۴-۵-۶ مدل‌سازی همکار مسؤولیت کلاس‌ها (CRD)

مدل‌سازی همکار- مسؤولیت کلاس‌ها (CRC) [Wir90] ابزاری ساده برای شناسایی و سازمان دهی کلاس‌های مرتبط با خواسته‌های سیستم یا محصول فراهم می‌سازند. امبلر [Amb95] مدل‌سازی CRC را به شیوه‌ی زیر توصیف می‌کند:

مدل CRC در واقع مجموعه‌ای از کارت‌های شاخص استاندارد است که کلاس‌ها را به نمایش می‌گذارند. این کارت‌ها به سه بخش تقسیم می‌شوند:

در بالای کارت، نام کلاس را می‌نویسید. در بدنه‌ی کارت، فهرست مسؤولیت‌های کلاس را در طرف راست و همکاران را در طرف چپ می‌نویسید.

در واقع، برای مدل CRC ممکن است از کارت‌های واقعی یا مجازی استفاده شود. هدف، بسط یک نمایش سازمان یافته از کلاس‌هاست. *مسؤولیت‌ها* صفات و عملیات مرتبط با کلاس هستند. به بیان ساده، مسؤولیت عبارت است از «هر چیزی که کلاس می‌داند یا انجام می‌دهد» [Amb95]. *همکاران*، کلاس‌هایی هستند که برای فراهم ساختن اطلاعات لازم برای کامل‌شدن یک مسؤولیت توسط یک کلاس، مورد نیازند. به‌طور کلی، هر *همکاری* یا به معنای درخواست برای اطلاعات یا تقاضای یک کنش است.

یک کارت شاخص ساده CRC برای کلاس نقشه ساختمان در شکل ۱۱-۶ نشان داده شده است. فهرست مسؤولیت‌های نشان داده شده روی کارت CRC، فهرستی مقدماتی است و ممکن است مواردی به آنها اضافه و اصلاح شود. کلاس‌های دیوار و دوربین در کنار مسؤولیتی ذکر می‌شوند که نیاز به همکاری آنها دارند.

کلاس‌ها، دستور العمل‌های اصلی برای شناسایی کلاس‌ها و اشیا قبلاً در این فصل ارائه شدند. طبقه‌بندی انواع کلاس‌های ارائه شده در بخش ۱-۵-۶ را می‌توان با در نظر گرفتن گروه‌های زیر بسط داد:

«یکی از اهداف کارت‌های CRC، شکست زودهنگام، شکست زیاد و شکست کم‌هزینه است. پاره کردن چند تکه کاغذ به مراتب کم خرج‌تر از سازمان دهی دوباره به مقادیر فراوانی از کد منبع است.»

سی. هورستمان

مرجع وب

بحثی عالی درباره این انواع کلاس‌ها را در وب سایت زیر می‌توانید بیابید:
www.theumlcafe.com/a0079.htm

کلاس: FloorPlan	
مسؤولیت:	همکار:
تعیین نام، نوع نقشه ساختمان	
مدیریت مکان نقشه ساختمان	
تغییر مقیاس نقشه ساختمان برای نمایش	
تغییر مقیاس نقشه ساختمان برای نمایش	
قراردادن دیوارها، درها و پنجره‌ها	Wall
نشان دادن موقعیت دوربین‌ها	Camera

شکل ۱۱-۶ یک کارت شاخص مدل CRC.

- *کلاس‌های موجودیت*، که کلاس‌های مدل یا تجاری نیز نامیده می‌شوند، مستقیماً از بیان مسأله استخراج می‌شوند (مثلاً **FloorPlan** و **Sensor**). این کلاس‌ها معمولاً چیزهایی را نمایش می‌دهند که قرار است در یک بانک اطلاعاتی ذخیره شوند و در سرتاسر مدت کاربرد ماندگار باشند (مگر اینکه مشخصاً حذف شوند).
- *کلاس‌های مرزی* در ایجاد واسط (مثلاً صفحه نمایش تعاملی یا گزارش‌های چاپی) به‌کار می‌روند که کاربر به هنگام استفاده از نرم‌افزار و تعامل با آن مشاهده می‌کند. اشیای موجودیت حاوی اطلاعاتی هستند که برای کاربران اهمیت دارند، ولی خودشان را نشان نمی دهند. کلاس‌های مرزی با مسؤولیت مدیریت کردن شیوه‌ی ارائه اشیای موجودیت به‌کاربران طراحی می‌شوند. برای مثال، یک کلاس مرزی با نام پنجره دوربین مسؤولیت نمایش خروجی دوربین‌ها را برای سیستم *SafeHome* برعهده خواهد داشت.
- *کلاس‌های کنترل گر*. یک «واحد کار» [UML03] را از ابتدا تا انتها مدیریت می‌کند. یعنی، کلاس‌های کنترل‌گر را می‌توان طوری طراحی کرد که (۱) ایجاد یا بهنگام‌سازی اشیای موجودیت ، (۲) معرفی اشیای مرزی به هنگام کسب اطلاعات از اشیای موجودیت، (۳) ارتباطات پیچیده میان مجموعه‌های اشیا، (۴) اعتبارسنجی داده‌های ارتباطی میان اشیا یا میان کاربر و برنامه را مدیریت کنند. به‌طور کلی، کلاس‌های کنترل‌گر تا شروع فعالیت طراحی در نظر گرفته نمی‌شوند.

مسؤولیت‌ها. دستور العمل‌های پایه برای شناسایی مسؤولیت‌ها (صفات و عملیات ها) در بخش‌های ۲-۵-۶ و ۳-۵-۶ ارائه شده‌اند ویرفس-براک و همکارانش [Wir90] پنج دستورالعمل برای تخصیص مسؤولیت‌ها به کلاس‌ها پیشنهاد می‌کنند.

۱. **هوش‌مندی سیستم باید طوری میان کلاس‌ها توزیع شود که به بهترین وجه پاسخ گوی نیازهای مسأله باشد.** هر برنامه‌ی کاربردی شامل سطح معینی از هوش‌مندی می‌شود؛ یعنی آنچه که سیستم می‌داند و آنچه قادر به انجام آن است. این هوش‌مندی به چند شیوه‌ی متفاوت در میان کلاس‌ها قابل توزیع است. کلاس‌های «خنگ» (کلاس‌هایی با تعداد معدودی از مسؤولیت‌ها) را می‌توان برای خدمت رساندن به کلاس‌های «زرنگ» (کلاس‌هایی با مسؤولیت‌های فراوان) مدل‌سازی کرد. گرچه این روش، جریان کنترل در یک سیستم به صراحت مشخص می‌شود،

> «اشیا را از نظر علمی به سه گروه اصلی می‌توان طبقه‌بندی کرد: آنها که کار نمی کنند، آنها که از کار می‌افتند و آنها که گم می‌شوند.»
>
> راسل بیکر

> چه دستورالعمل‌هایی را می‌توان برای تخصیص مسؤولیت‌ها به کلاس‌ها به کاربرد؟

معایبی هم دارد: همه‌ی هوش‌مندی را در چند کلاس معدود متمرکز می‌کند، اعمال تغییرات را دشوارتر می‌سازد و نیاز به ایجاد کلاس‌های بیشتر و در نتیجه کار و تلاش بیشتر دارد.

اگر هوش‌مندی سیستم به‌طور یکنواخت‌تر در میان کلاس‌های یک برنامه‌ی کاربردی توزیع شده باشد، هر شیء تنها از چند چیز اطلاعات دارد و همان چند چیز را انجام می‌دهد (که عموماً آنها را با توجه خوبی انجام خواهد داد) و یکپارچگی سیستم بهبود خواهد یافت.[1] این باعث بهبود قابلیت نگهداری سیستم شده از تأثیر اثرات جانبی ناشی از تغییر می‌کاهد.

برای اینکه بدانید آیا هوش‌مندی سیستم از توزیع مناسبی برخوردار هست یا خیر، مسؤولیت‌های ذکر شده در هر کارت شاخص مدل CRC را باید ارزیابی کنید تا معلوم شود که آیا کلاس(هایی) دارای فهرست مسؤولیت‌های بلندتر از حد معمول هستند یا خیر. به این شیوه، شاخصی از هوش‌مندی به‌دست می‌آید.[2] به‌علاوه، مسؤولیت هر کدام از کلاس‌ها باید سطح انتزاعی هم‌ردیف با سایر کلاس را از خود نشان دهد. برای مثال، در میان عملیات فهرست شده برای یک کلاس مجتمع با نام **CheckingAccount** در حین بازبینی به دو مسؤولیت برخورد می‌کنید: *موازنه حساب* و *چک‌های پاس شده*. عملیات (مسؤولیت) اول شامل یک روال ریاضی و منطقی پیچیده می‌شود. دومی یک فعالیت دفتری ساده است. چون این دو مسؤولیت در سطح انتزاع یکسان قرار ندارند، *چک‌های پاس شده* باید در مسؤولیت‌های کلاس **CheckEntry** قرار داده شوند، کلاسی که بخشی از کلاس مجتمع **CheckingAccount** است.

۲. **هر مسؤولیتی باید تا حد امکان به‌صورت کلی بیان شود.** این دستور العمل بدان مناسبت که مسؤولیت‌های کلی (هم صفات و هم عملیات) باید در بالای سلسله مراتب کلاس‌ها قرار داده شوند (چون عمومیت دارند، باید در مورد همه‌ی زیرکلاس‌ها کاربرد داشته باشند).

۳. **اطلاعات و رفتار مرتبط با آن باید در یک کلاس قرار داده شوند.** به این ترتیب، اصلی در رویکرد شیء‌گرا با عنوان *پنهان‌سازی* رعایت خواهد شد. داده‌ها و فرایندهایی که این داده‌ها را دستکاری می‌کنند باید به‌صورت یک واحد یکپارچه بسته‌بندی شوند.

۴. **اطلاعات مربوط به یک چیز باید تنها در یک کلاس قرار داده شوند و نباید در میان چند کلاس توزیع شوند.** یک کلاس به تنهایی باید مسؤولیت ذخیره‌سازی و دستکاری نوع مشخصی از اطلاعات را عهده دار گردد. به‌طور کلی، این مسؤولیت نباید در میان چند کلاس به اشتراک گذاشته شود. اگر اطلاعات توزیع شوند، نگهداری نرم‌افزار دشوارتر می‌شود و برای آزمون آن هم چالش بیشتری وجود خواهد داشت.

۵. **مسؤولیت‌ها را در صورت امکان باید در میان کلاس‌های مرتبط به اشتراک گذاشت.** موارد فراوانی وجود دارد که در آنها انواع اشیای مرتبط همگی باید در یک زمان، رفتاری مشابه از خود نشان دهند. به‌عنوان مثال، یک بازی کامپیوتری را در نظر بگیرید که کلاس‌های زیر را نشان می‌دهد: **Player**، **PlayerBody**، **PlayerArms**، **PlayerLegs**، **PlayerHead**. هر کدام از این کلاس‌ها دارای صفات خاص خود است (مثل *موقعیت*، *جهت گیری*، *رنگ*، *سرعت*) و

---

[1] یکپارچگی یک مفهوم طراحی است که در فصل ۸ بحث خواهیم کرد.

[2] در چنین مواردی، ممکن است نیاز باشد که کلاس به چند کلاس دیگر تقسیم گردد تا هوش‌مندی بهتر توزیع شود.

همه‌ی آنها باید با حرکت دادن دسته بازی توسط کاربر، بهنگام شوند و به نمایش درآیند. پس مسؤولیت‌های *update( )* و *display( )* باید در همه‌ی اشیای ذکرشده به‌طور مشترک موجود باشند. بازیکن می‌داند که چه هنگام چیزی تغییر کرده است و update( ) لازم است. این شیء با سایر اشیا همکاری می‌کند تا موقعیت و جهت‌گیری جدیدی اتخاذ شود، ولی هر شیء نمایش خود را کنترل می‌کند.

همکاری‌ها. کلاس‌ها به یکی از دو شیوه، مسؤولیت‌های خود را به انجام می‌رسانند: (۱) کلاس می‌تواند از عملیات‌های خودش برای دستکاری صفات خودش استفاده کند یا (۲) کلاس می‌تواند با سایر کلاس‌ها همکاری کند. ویرفس- براک و همکارانش [Wir90] همکاری را به شیوه‌ی زیر تعریف می‌کنند:

> همکاری‌ها نشان‌گر درخواست‌های یک کلاینت از سرور برای به انجام رساندن مسؤولیت یک کلاینت هستند. همکاری، تجسم هم پیمانی کلاینت و سرور است... می‌گوییم یک شیء با دیگری همکاری می‌کند اگر برای به انجام رساندن مسؤولیتی، نیاز به ارسال پیام به شیء دیگر داشته باشد. یک همکاری منفرد، تنها در یک جهت جریان پیدا می‌کند- که در خواست را از کلاینت به‌سرور نشان می‌دهد. از دید کلاینت، هرکدام از این همکاری‌ها با یک مسؤولیت خاص همراه است که توسط سرور پیاده‌سازی می‌شود.

برای شناسایی همکاری‌ها باید تعیین کرد کدام کلاس می‌تواند هر مسؤولیت را خودش به انجام برساند. اگر نتواند، ناگزیر از تعامل با کلاسی دیگر است. از این رو، یک همکاری به‌شمار می‌رود.

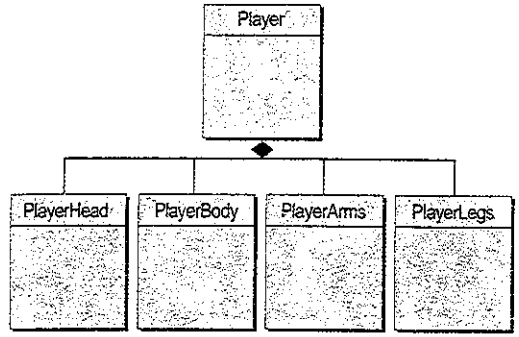
به‌عنوان مثال، قابلیت امنیتی *SafeHome* را در نظر بگیرید. شیء پانل کنترلی به‌عنوان بخشی از روال فعال‌سازی، باید تعیین کند که آیا حس‌گری باز هست. مسؤولیتی با نام *determine-sensors-tatus( )* تعریف می‌شود. اگر حس‌گرها باز باشند، پانل کنترلی باید صفت *وضعیت* را در حالت «غیر آماده» قرار دهد. اطلاعات حس‌گر از هر کدام از اشیای حس‌گر ممکن است به‌دست آید. بنابراین، مسؤولیت *determine-sensor-status( )* تنها در صورتی قابل انجام است که پانل کنترلی با حس‌گر همکاری کند.

برای کمک به شناسایی همکارها می‌توانید سه رابطه‌ی کلی میان کلاس‌ها را بررسی کنید [Wir90]: (۱) رابطه‌ی «*شمول*»، (۲) رابطه‌ی «*آگاهی داشتن از*» و (۳) «*بستگی داشتن به*». هر کدام از سه رابطه‌ی مذکور را به اختصار در پاراگراف‌های زیر شرح خواهیم داد.

همه‌ی کلاس‌هایی که بخشی از یک کلاس مجتمع هستند، از طریق رابطه‌ی شمول به آن کلاس مجتمع متصل‌اند. کلاس‌های مربوط به بازی کامپیوتری را که در بالا گفته شد، در نظر بگیرید: کلاس بدنه بازیکن بخشی از بازیکن است و همین طور، **PlayerArms**، **PlayerLegs** و **PlayerHead**. در نمادگذاری UML، این روابط به‌صورت مجتمع شکل ۱۲-۶ نمایش داده شده است.

هنگامی که یک کلاس باید اطلاعات را از کلاس دیگری کسب کند، رابطه‌ی «*آگاهی داشتن از*» برقرار می‌شود. مسؤولیت *determine-sensor-status( )* که قبلاً ذکر شد، مثالی از رابطه‌ی «*آگاهی داشتن از*» است.

رابطه‌ی «*بستگی داشتن به*» به این معنی است که دو کلاس دارای ارتباطی به جز «*آگاهی داشتن از*» و «*شمول*» هستند. برای مثال، کلاس **PlayerHead** باید همواره به کلاس **PlayerBody** متصل باشد (مگر اینکه در بازی کامپیوتری مورد بحث، خشونت زیادی در نظر گرفته شود)، و در عین حال هر



شکل ۱۲-۶ یک کلاس مجتمع مرکب.

شیء بدون آگاهی از دیگری وجود دارد. یک صفت از شیء **PlayerHead** با نام «*موقعیت-مرکز*» از روی موقعیت مرکز شیء **PlayerBody** تعیین می‌شود. این اطلاعات از طریق یک شیء سوم، **Player**، به‌دست می‌آید که آن را از **PlayerBody** کسب می‌کند پس **PlayerHead** به **PlayerBody** *وابسته* است.

در تمامی کلاس‌ها، نام کلاس همکار روی کارت شاخص مدل CRC در کنار مسؤولیتی نوشته می‌شود که آن همکاری را طلب می‌کند. پس کارت شاخص حاوی فهرستی از مسؤولیت‌ها و همکاری‌های متناظر می‌شود که انجام آن مسؤولیت را میسر می‌سازند (شکل ۱۱-۶).

هنگامی که یک مدل CRC کامل توسعه یافت، طرف‌های ذی‌نفع می‌توانند مدل را با استفاده از رویکرد زیر بازبینی کنند [Amb95]:

۱. به همه‌ی مشارکت کنندگان در بازبینی (مدل CRC) زیر مجموعه‌ای از کارت‌های شاخص مدل CRC داده می‌شود. کارت‌هایی که همکاری دارند، باید جدا شوند (یعنی هیچ کس نباید دو کارت داشته باشد که همکاری دارند).
۲. همه‌ی سناریوهای use case (و نمودارهای مربوط به use case) باید گروه‌بندی شوند.
۳. سرگروه تیم بازبینی، use case را به شیوه‌ای شمرده قرائت می‌کند. با رسیدن سرگروه به یک شیء نامگذاری شده، رشته سخن را به‌دست کسی می‌سپرد که کارت شاخص کلاس مربوط را دردست دارد. برای مثال، یک use case مربوط به محصول *SafeHome* حاوی متن روایی زیر است:

> صاحبخانه پانل کنترلی *SafeHome* را مشاهده می‌کند تا معلوم شود که آیا سیستم برای وارد کردن دستورات آماده است. اگر سیستم آماده نبود، صاحبخانه باید به‌صورت فیزیکی پنجره‌ها/درها را ببندد، به‌طوری که نشان‌گر آمادگی، روشن شود. [نشان‌گر not-ready مشخص می‌کند که حس‌گری باز است. یعنی در یا پنجره‌ای باز مانده است.]

هنگامی که سرگروه تیم مروز در متن روایی use case به «پانل کنترلی» رسید، رشته سخن به کسی سپرده می‌شود که کارت شاخص **پانل کنترلی** را در دست دارد. عبارت «مشخص می‌کند که حس‌گری باز است» ایجاب می‌کند که کارت شاخص حاوی مسؤولیتی باشد که این معنی را اعتبارسنجی کند (مسؤولیت *determine-sensor-status( )* این کار را انجام می‌دهد). در کنار این مسؤولیت روی کارت اندیس، همکار حس‌گر مشاهده می‌شود. اکنون رشته سخن به شیء حس‌گر سپرده می‌شود.

SafeHome

مدل‌های CRC

صحنه: اتاقک اد، در شروع مدل‌سازی خواسته‌ها.

نقش‌آفرینان: وینود و اد- اعضای تیم مهندسی نرم‌افزار *SafeHome*.

مکالمه:

[وینود می‌خواهد با نشان دادن یک مثال، نحوه‌ی توسعه‌ی کارت‌های CRC را به اد یاد بدهد.]

وینود: در جالی که تو داشتی روی سیستم پایش *SafeHome* کار می‌کردی و جیمی هم مشغول قابلیت امنیتی بود، من هم روی قابلیت مدیریت خانه کار می‌کردم.

اد: در چه وضعی است؟ بازاریابی مدام نظرش را عوض می‌کند.

وینود: بیا. این اولین برش از use case مربوط به کل این قابلیت است... ما یک قدری پالایش کردیم، ولی می‌تواند یک دید کلی بدهد...

use case: قابلیت خانه در محصول *SafeHome*

متن روایی: می‌خواهیم از واسط مدیریت خانه روی PC یا اتصال اینترنتی استفاده کنیم و دستگاه‌های الکترونیکی را که دارای کنترل‌گرهای واسط بی‌سیم هستند، کنترل کنیم. این سیستم باید به من این امکان را بدهد که چراغ‌های مشخصی را روشن و خاموش کنم، لوازم خانگی متصل به واسط بی‌سیم را کنترل کنم، سیستم گرمایش و تهویه‌ام را در دماهای معین، تنظیم کنم. برای این منظور، می‌خواهم دستگاه‌ها را از یک نقشه ساختمان منزل انتخاب کنم. هر دستگاه باید روی نقشه ساختمان تعیین گردد. به‌عنوان یک ویژگی اختیاری، می‌خواهم همه‌ی دستگاه‌های صوتی- تصویری- ضبط، تلویزیون، DVD، دوربین‌های دیجیتال و غیره- قابل کنترل باشند.

می‌خواهم قادر باشم با یک انتخاب ساده کل خانه را برای وضعیت‌های گوناگون تنظیم کنم. یکی با عنوان *home* یکی با عنوان *away*، سومی با عنوان *overnight travel* (سفر یک شبه) و چهارمی با عنوان *extended travel*. این وضعیت دارای تنظیماتی خواهد بود که روی همه‌ی دستگاه‌ها اعمال خواهند شد. در حالت‌های overnight travel و extended travel، سیستم باید چراغ‌ها را در فواصل زمانی تصادفی، روشن و خاموش کند (تا به نظر برسد که کسی در خانه است) و سیستم گرمایش و تهویه را کنترل کند. به‌علاوه باید بتوانم از طریق اینترنت با وارد کردن کلمه‌ی عبور مناسب، این تنظیمات را تغییر دهم...

اد: بچه‌های سخت‌افزار همه‌ی واسط‌های بی‌سیم را درست کرده‌اند؟

وینود (لبخند می‌زند): دارند روی آن کار می‌کنند؛ این مسأله مهمی نیست. به هر حال، من یک مشت کلاس برای مدیریت خانه استخراج کرده‌ام و می‌توانیم از آنها به‌عنوان مثال استفاده کنیم. از کلاس HomeManagementInterface استفاده می‌کنیم.

اد: باشد... پس مسؤولیت‌ها همان صفات و عملیات‌های کلاس هستند و همکارها، کلاس‌هایی که مسؤولیت‌ها به آنها اشاره دارند.

وینود: فکر کنم درست CRC را نفهمیدی.

اد: شاید یک کم، ولی شروع کن.

وینود: من کلاس‌های **HomeManagementInterface** را این طور تعریف کردم.

صفات:

Option Panel–حاوی اطلاعاتی درباره دکمه‌هایی ست که به‌کاربر امکان انتخاب قابلیت را می‌دهد.

Situation Panel–حاوی اطلاعاتی درباره دکمه‌هایی است که به‌کاربر امکان انتخاب وضعیت را می‌دهد.

FloorPlan–همانند شیء پایش است، ولی این یکی دستگاه‌ها را نشان می‌دهد.

Device Icons–اطلاعات مربوط به آیکون‌هایی که چراغ‌ها، لوازم خانگی، HVAC و غیره را نشان می‌دهند.

DevicePanels- پانل کنترلی برای شبیه‌سازی لوازم خانگی یا دستگاه‌ها، کنترل را میسر می‌سازد.

عملیات‌ها:

displayControl( )
displaySituation( )
accessFloorPlan( )
displayDevicePanel( )
selectControl( )
selectSituation( )
selectDeviceIcon( )
accessDevicePanel( )

کلاس: HomeManageMentInterface

همکار	مسؤولیت
OptionsPanel (کلاس)	*displayControl( )*
OptionsPanel (کلاس)	*selectControl( )*
SituationPanel (کلاس)	*displaySituation( )*
SituationPanel (کلاس)	*selectSituation( )*
FloorPlan (کلاس)	*accessFloorPlan*

اد: پس وقتی که *accessFloorPlan( )* فراخوانده شود، با شیء **FloorPlan** همکاری می‌کند، درست مثل همان که برای پایش توسعه دادیم. صبر کن، یک توصیف از آن در اینجا دارم (به شکل ۱۰-۶ نگاه می‌کنند.)

وینود: دقیقاً. و اگر می‌خواستیم کل مدل کلاس‌ها را مرور کنیم، می‌توانستیم با این کارت شاخص شروع کنیم بعد به کارت شاخص کلاس همکار برویم و از آنجا به همکارهای کلاس همکار و غیره.

اد: راه خوبی برای پیدا کردن خطاها یا جاافتادگی‌هاست.

وینود: بله.

۴. هنگام تحویل نشانه، از نگهدارنده‌ی کارت Sensor خواسته می‌شود که مسؤولیت‌های ذکر شده روی کارت را توصیف کند. گروه تعیین می‌کند که آیا یک (یا چند) مورد از مسؤولیت‌ها، خواسته‌ی مورد نظر use case را برآورده می‌سازد یا خیر.

۵. اگر مسؤولیت‌ها و همکاری‌های ذکر شده در کارت‌های شاخص نتوانند پاسخ گوی use case باشند، اصطلاحاتی در کارت‌ها به عمل می‌آید. این اصطلاحات ممکن است شامل تعریف کلاس‌های جدید (و کارت‌های شاخص CRC متناظر با آنها) یا تعیین مشخصات یک مسؤولیت یا همکاری بازبینی شده روی کارت‌های موجود شود.

این شیوه‌ی عمل خاص آن‌قدر ادامه پیدا می‌کند که use case تمام شود. هنگامی که همه‌ی use caseها بازبینی شدند، مدل‌سازی خواسته‌ها ادامه می‌یابد.

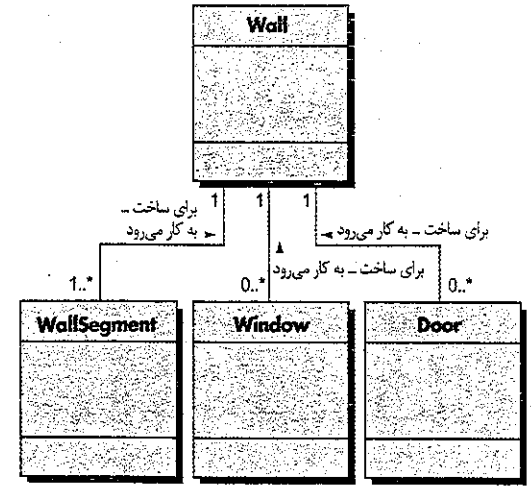
۶-۵-۵ اجتماع و وابستگی

در بسیاری از موارد، دو کلاس تحلیل، به شیوه‌ای بسیار مشابه با ارتباط دو شیء داده‌ای، با هم ارتباط دارند (بخش ۳-۴-۶). در UML، این روابط را *اجتماع* می‌نامند. توجه دوباره به شکل ۱۰-۶ می‌بینیم که کلاس **FloorPlan** با شناسایی مجموعه‌ای از اجتماع‌ها میان **FloorPlan** و دو کلاس دیگر، **Camera** و **Wall** تعریف می‌شوند. کلاس **Wall** با سه کلاس همبستگی دارد که به دیوار امکان ساخته‌شدن را می‌دهند و عبارتند از **Wall Segment**، **Window**، و **Door**.

نکته‌ی کلیدی

اجتماع، رابطه‌ی میان کلاس‌ها را مشخص می‌کند. چندگانگی مشخص می‌کند که چند کلاس با چند کلاس دیگر ارتباط دارند.

در برخی موارد، یک همبستگی ممکن است با ذکر *چندگانگی* (multiplicity) بهتر قابل تعریف باشد. با رجوع به شکل ۱۰-۶، شیء **Wall** از یک یا چند شیء **WallSegment** ساخته می‌شود. این قید و بندهای چندگانگی در شکل ۱۳-۶ نمایش داده شده‌اند که در آن «یک یا چند» با استفاده از *..1 و «صفر یا چند» با استفاده از *..0 نمایش داده می‌شود. در UML، ستاره نشان‌گر مرز بالایی نامحدود گسترده است.[1]



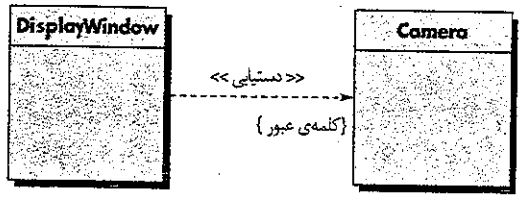
شکل ۱۳-۶ چندگانگی.

[1] سایر روابط چندگانگی-یک به یک، یک به چند، چند به چند، یک به گستره‌ای مشخص و مرزهای پایینی و بالایی و غیره- را می‌توان به‌عنوان بخشی از یک همبستگی ذکر کرد.

کلیشه چیست؟

در بسیاری موارد، میان دو کلاس تحلیل یک رابطه‌ی کلاینت- سرور وجود دارد. در این گونه موارد، کلاس کلاینت به نحوی به کلاس سرور وابسته است و یک *رابطه‌ی وابستگی* برقرار می‌شود. وابستگی‌ها توسط یک کلیشه تعریف می‌شوند. کلیشه یک «سازوکار بسط پذیری» [Arl02] در UML است که به شما امکان تعریف یک عنصر مدل‌سازی خاص را می‌دهد که معناشناسی آن مطابق میل شما قابل تعیین است. در UML، کلیشه‌ها در داخل براکت‌های دوتایی آورده می‌شوند مثلاً << کلیشه>>.

نمایشی از یک وابستگی ساده در داخل سیستم پایش *SafeHome*. یک شیء **Camera** (که در این مورد، کلاس سرور است) یک تصویر ویدیویی در اختیار شیء **DisplayWindow** قرار می‌دهد (که در این مورد، کلاس کلاینت است). رابطه‌ی میان این دو شیء یک همبستگی ساده نیست و در عین حال، یک همبستگی از نوع وابستگی وجود دارد. در یک use case نوشته شده برای پایش (که در اینجا نشان داده نشده است)، در می‌یابید که برای مشاهده مکان دوربین‌های مشخص، باید یک کلمه‌ی عبور خاص ارائه شود. یک راه برای نیل به این مقصود، آن است که **Camera** درخواست کلمه‌ی عبور کند و سپس اجازه تولید نمایش ویدیویی را به **Display Window** اعطا کند. این را می‌توان به‌صورت نشان داده شده در شکل ۱۴-۶ نمایش داد که در آن <<access>> بدان معناست که استفاده از خروجی دوربین توسط یک کلمه‌ی عبور خاص کنترل می‌شود.



شکل ۱۴-۶ وابستگی‌ها.

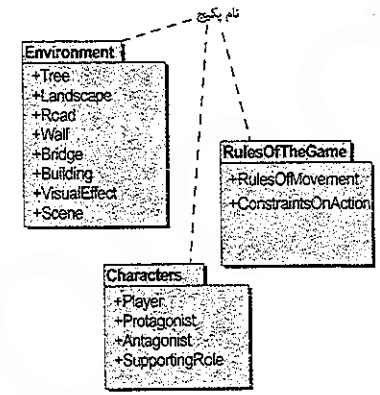
۶-۵-۶ پکیج‌های تحلیل

نکته‌ی کلیدی

پکیج برای بسته‌بندی مجموعه‌ای از کلاس‌های مرتبط به‌کار می‌رود.

بخش مهمی از مدل‌سازی تحلیل، گروه‌بندی است. یعنی عناصر گوناگون مدل تحلیل (مثل use caseها و کلاس‌های تحلیل) طوری گروه‌بندی می‌شوند که یک پکیج از آنها تشکیل شود- و به آنها پکیج *تحلیل* گفته می‌شود؛ به هر کدام از این پکیج‌ها یک نام مشخص داده می‌شود.

برای روشن‌شدن کاربرد پکیج‌های تحلیل، همان مثال بازی کامپیوتری را در نظر بگیرید که قبلاً معرفی شد. همچنان که بازی توسعه پیدا می‌کند، تعداد زیادی از کلاس‌ها به‌دست خواهد آمد. برخی از آنها بر محیط بازی تاکید دارند- منظور، صحنه‌های بصری است که کاربر هنگام نمایش بازی مشاهده می‌کند. کلاس‌هایی نظیر **Tree**، **Landscape**، **Road**، **Wall**، **Bridge**، **Building** و **VisualEffect** ممکن است در این گروه قرار گیرند. کلاس‌هایی نظیر **Player** (که قبلاً شرح داده شد)، **Protagonist**، **Antagonist** و **SupportingRoles** را نیز می‌توان تعریف کرد. به‌علاوه، کلاس‌های دیگری هستند که قواعد بازی را توصیف می‌کنند- اینکه بازیکن چگونه در محیط گشت و گذار کند. کلاس‌هایی نظیر **RulesOfMovement** و **ConstraintsOnAction** مثال‌هایی از این گروه به‌شمار می‌روند. گروه‌های بسیار دیگری نیز ممکن است وجود داشته باشند. این کلاس‌ها را می‌توان در پکیج‌های تحلیل مطابق با شکل ۱۵-۶ گروه‌بندی کرد.



شکل ۱۵-۶ پکیج‌ها.

علامت مثبت قبل از نام کلاس تحلیل در هر پکیج نشان‌گر آن است که این کلاس‌ها در معرض دید عموم هستند و بنابراین از طریق سایر پکیج‌ها قابل دستیابی‌اند. علامت‌های دیگری نیز ممکن است قبل از عناصر داخل یک پکیج قرار داده شوند. علامت منفی نشان می‌دهد که عنصر از همه‌ی پکیج‌های دیگر پنهان است و علامت # نشان می‌دهد که یک عنصر، تنها در دسترس پکیج‌های موجود در داخل پکیجی مشخص قرار دارد.

۶-۶ خلاصه

هدف از مدل‌سازی خواسته‌ها، ایجاد انواع نمایش هاست که نیازهای مشتری را توصیف کنند، بستری برای ایجاد طراحی نرم‌افزار فراهم سازند و مجموعه‌ای از خواسته‌ها را تعریف کنند که پس از ساخته‌شدن نرم‌افزار بتوان آنها را اعتبارسنجی کرد. مدل خواسته‌ها پلی است میان نمایش در سطح سیستمی (که کل سیستم و عملکردهای تجاری آن را توصیف می‌کند) و یک طراحی نرم‌افزار (که معماری کاربرد نرم‌افزار، واسط کاربر، و ساختار سطح- مؤلفه‌ای را توصیف می‌کند).

مدل‌های مبتنی بر سناریو، خواسته‌های نرم‌افزار را از دیدگاه کاربر به تصویر می‌کشند. use case- توصیفی روایی یا مبتنی بر یک الگوی مشخص از تعامل میان کنش‌گر و نرم‌افزار- عنصر اصلی مدل‌سازی به‌شمار می‌رود. use case، که طی استخراج خواسته‌ها به‌دست می‌آید، مراحل کلیدی مربوط به یک عملکرد یا تعامل مشخص را تعریف می‌کند. درجه‌ی رسمیت و جزئیات در use case، متغیر است، ولی نتیجه‌ی نهایی، ورودی لازم برای همه‌ی فعالیت‌های دیگر مدل‌سازی تحلیل را فراهم می‌سازد. سناریوها را با استفاده از نمودار فعالیت‌ها نیز می‌توان توصیف کرد- نمایش گرافیکی شبیه به نمودارهای گردشی که جریان پردازش را در داخل آن سناریو به تصویر می‌کشد. نمودارهای بخش بندی، چگونگی تخصیص دهی جریان پردازش به کنش گران یا کلاس‌های گوناگون را نشان می‌دهد.

مدل‌سازی داده‌ها در توصیف فضای اطلاعاتی که توسط نرم‌افزار ساخته یا دستکاری خواهد شد، به‌کار گرفته می‌شود. مدل‌سازی داده‌ها با نمایش دادن اشیای داده آغاز می‌شود- اطلاعات مرکبی که باید نرم‌افزار آنها را بفهمد. صفات هر شیء داده شناسایی و روابط میان اشیای داده توصیف می‌شود.

در مدل‌سازی مبتنی بر کلاس‌ها از اطلاعات به‌دست آمده از عناصر مدل‌سازی مبتنی بر سناریو یا مدل‌سازی داده‌ها استفاده می‌شود. برای استخراج کلاس‌ها، صفات و عملیات‌های کاندیدا از روایت‌های متنی از تجزیه‌ی گرامری ممکن است استفاده شود. ملاک‌های تعریف یک کلاس مشخص می‌شوند.

برای تعریف روابط میان کلاس‌ها می‌توان از یک مجموعه کارت شاخص CRC استفاده کرد. به‌علاوه، انواع نمادهای مدل‌سازی UML را می‌توان برای تعریف سلسله مراتب‌ها، روابط، همبستگی‌ها، اجتماع‌ها و وابستگی‌ها در میان کلاس‌ها به‌کار گرفت. پکیج‌های تحلیل برای گروه‌بندی کلاس‌ها به‌کار می‌روند تا در سیستم‌های بزرگ، بهتر بتوان آنها را مدیریت کرد.

مسائل و نکاتی برای تعمق

۱-۶ آیا شروع کد نویسی بلافاصله پس از ایجاد مدل تحلیل امکان‌پذیر است؟ بر پاسخ خود توضیح دهید و برای نظر مخالف دلیل بیاورید.

۲-۶ طبق یک قاعده ساده در تحلیل «مدل باید خواسته‌هایی را کانون توجه قرار دهد که در دامنه‌ی مسأله یا کسب و کار قابل مشاهده باشند.» کدام خواسته‌ها هستند که در این دامنه‌ها قابل مشاهده نیستند؟ چند مثال بیاورید.

۳-۶ هدف از تحلیل دامنه چیست؟ چه ارتباطی با مفهوم الگوی خواسته‌ها دارد؟

۴-۶ آیا توسعه‌ی یک مدل تحلیل اثربخش، بدون توسعه دادن هر چهار عنصر شکل ۳-۶ غیر ممکن است؟ توضیح دهید.

۵-۶ از شما خواسته شده است که یکی از سیستم‌های زیر را بسازید:

الف. یک سیستم ثبت نام واحدهای درسی تحت شبکه برای دانشگاه

ب. یک سیستم سفارش‌گیری مبتنی بر وب برای فروشگاه کامپیوتر

پ. یک سیستم صدور فاکتور برای شرکت‌های تجاری کوچک

ت. یک کتاب آشپزی اینترنتی که در اتاق مایکروویو تعبیه‌شده است.

۶-۶ سیستم مورد علاقه خود را انتخاب کنید و یک نمودار رابطه‌ی میان موجودیت‌ها تهیه کنید که اشیای داده، روابط و صفات را توصیف کند.

بخش کارهای عمومی برای یک شهر بزرگ تصمیم گرفته یک سیستم مبتنی بر وب برای ترمیم چاله‌های خیابان‌ها (PHTRS) ایجاد کند. شرح این سیستم به‌صورت زیر است:

> شهروندان می‌توانند وارد یک وب سایت شوند و مکان و شدت چاله را گزارش کنند. این چاله‌ها پس از گزارش‌شدن در یک «سیستم ترمیم چاله‌ها» ثبت می‌شوند و یک شماره شناسایی به آنها داده می‌شود، نشانی خیابان، اندازه چاله (در مقیاس ۱ تا ۱۰)، مکان (وسط خیابان، لبه پیاده رو،...)، ناحیه (از روی آدرس خیابان تعیین می‌شود) و اولویت ترمیم (از روی اندازه چاله) ذخیره می‌شود. داده‌های سفارش کار با هر چاله همراه می‌شوند و شامل مکان و اندازه چاله، شماره شناسایی گروه ترمیم گر، تعداد افراد گروه، تجهیزات لازم، ساعت‌های صرف شده برای ترمیم، وضعیت چاله (کار در حال انجام، ترمیم شده، ترمیم موقت، ترمیم نشده)، مقدار ماده پرکننده به‌کار رفته و هزینه‌ی ترمیم می‌شود (که از روی ساعت‌های کار شده، تعداد افراد، ماده و تجهیزات به‌کار رفته محاسبه می‌شود). سرانجام، یک فایل خسارت ایجاد می‌شود که اطلاعات مربوط به خسارت‌های ناشی از چاله را گزارش می‌کند و شامل نام شهروند، آدرس، شماره تلفن، نوع خسارت و مقدار خسارت برحسب دلار را در خود نگهداری می‌کند. PHTRS یک سیستم آنلاین است؛ همه‌ی پرسش و پاسخ‌ها باید به‌صورت تعاملی باشد.

الف. برای سیستم PHTRS یک نمودار use case UML رسم کنید. برای شیوه‌ی تعامل کاربر با این سیستم باید یک سری فرضیات داشته باشید.

ب. یک مدل کلاس برای سیستم PHTRS توسعه دهید.

۶-۷ یک use case مبتنی بر الگو برای سیستم مدیریت خانه در محصول *SafeHome* بنویسید که به‌طور غیر رسمی در کادر بخش ۴-۵-۶ توصیف شود.

۶-۸ مجموعه کاملی از کارت‌های شاخص مدل CRC برای سیستم یا محصول انتخاب شده در مسأله ۵-۶ تهیه کنید.

۶-۹ کارت‌های شاخص مدل CRC را با همکاران خودتان بازبینی کنید. در نتیجه‌ی این بازبینی چند کلاس، مسؤولیت و همکار دیگر اضافه می‌شود؟

۶-۱۰ پکیج تحلیل چیست و چگونه می‌توان از آن استفاده کرد؟

فصل ۷

مدل‌سازی خواسته‌ها: جریان، رفتار، الگوها و برنامه‌های تحت وب

نگاهی گذرا

مدل‌سازی خواسته‌ها چیست؟ مدل خواسته‌ها چند بُعد دارد. در این فصل، مطالبی درباره مدل‌های جریان‌گرا، مدل‌های رفتاری و ملاحظات خاص برنامه‌های تحت وب برای تحلیل خواسته‌ها خواهید آموخت.

هر کدام از این نمایش‌های مدل سازی، مکمل use case، مدل‌های داده‌ای و مدل‌های مبتنی بر کلاس بحث شده در فصل ۶ هستند.

چه کسی آن را انجام می‌دهد؟ مهندس نرم‌افزار (که گاهی «تحلیل گر» نامیده می‌شود) مدل را با به‌کارگیری خواسته‌های استخراج شده از طرف‌های ذی‌نفع گوناگون می‌سازد.

چرا اهمیت دارد؟ دید شما از خواسته‌های نرم‌افزار متناسب با تعداد ابعاد متفاوت مدل‌سازی خواسته‌ها رشد می‌کند. گرچه ممکن است وقت، منابع یا حتی تمایل به توسعه‌ی همه‌ی نمایش‌های پیشنهادی در این فصل و فصل ۶ را نداشته باشید، بدانید که هر رویکرد مدل‌سازی متفاوت، نگاه متفاوتی از مسأله به شما می‌دهد. در نتیجه، شما (و سایر طرف‌های ذی‌نفع) بهتر می‌توانید تشخیص دهید که آیا آن چه قرار است ساخته شود، به خوبی مشخص شده است یا خیر.

مراحل کار کدام است؟ مدل‌سازی جریان‌گرا چگونگی تبدیل اشیای داده‌ای توسط قابلیت‌های پردازش را نشان می‌دهد. در مدل‌سازی رفتاری، حالت‌های سیستم و کلاس‌های آن و تأثیر رویدادها بر این حالت‌ها به تصویر کشیده می‌شود. در مدل‌سازی مبتنی بر الگو، از دانش موجود درباره‌ی دامنه برای تسهیل در امر خواسته‌ها استفاده می‌شود. مدل‌های خواسته‌های مربوط به برنامه‌های تحت وب باید برای نمایش خواسته‌های مرتبط با محتوا، تعامل، عملکرد و پیکربندی مطابقت داده شوند.

محصول کار چیست؟ آرایه‌ی وسیعی از فرم‌های متنی و نموداری را می‌توان برای مدل‌سازی خواسته‌ها انتخاب کرد. هر کدام از این نمایش‌ها، دیدی از یک یا چند عنصر مدل ارائه می‌دهد.

چگونه مطمئن شوم که درست از عهده کار بر آمده‌ام؟ محصولات کاری مدل‌سازی خواسته‌ها را باید از نظر صحت، کمال و سازگاری بازبینی کرد. این محصولات باید نیازهای کلیه‌ی طرف‌های ذی‌نفع را منعکس سازد و بستری فراهم سازد تا طراحی در آن بستر اجرا گردد.

پس از بحث درباره use case، مدل‌سازی داده‌ای و مدل‌سازی مبتنی بر کلاس‌ها در فصل ۶، منطقی است که بپرسیم: «آیا این نمایش‌ها برای مدل‌سازی خواسته‌ها کفایت می‌کنند؟» تنها پاسخ منطقی که می‌توان داد این است که «بستگی دارد.»

برای برخی انواع نرم‌افزار، use case ممکن است تنها نمایش مورد نیاز برای مدل‌سازی باشد. برای بقیه، یک روش شیءگرا انتخاب می‌شود و مدل مبتنی بر کلاس‌ها را می‌توان برای آن‌ها توسعه داد. ولی در شرایط دیگر، ممکن است خواسته‌های کاربردی پیچیده، بررسی مواردی را که به دنبال خواهد آمد، طلب کند: یعنی چگونگی تبدیل اشیای داده‌ای را به هنگام حرکت در سیستم؛ چگونگی رفتار یک برنامه کاربردی در نتیجه‌ی رویدادهای خارجی؛ اینکه آیا آگاهی از دامنه‌ی موجود را می‌توان بر مسأله فعلی تطبیق داد؛ یا در مورد سیستم‌های مبتنی بر وب، محتوا و قابلیت‌های عملیاتی چگونه توانایی گشت وگذاری موفق در یک برنامه‌ی تحت وب را در اختیار کاربر می‌گذارند تا به اهداف خود دست پیدا کند.

۷-۱ راهبردهای مدل‌سازی خواسته‌ها

در یک دیدگاه از مدل‌سازی خواسته‌ها، که *تحلیل ساخت یافته* نامیده می‌شود، داده‌ها و فرایندهایی که این داده‌ها را تبدیل می‌کنند، موجودیت‌هایی مجزا در نظر گرفته می‌شوند. اشیای داده‌ای به شیوه‌ای مدل‌سازی می‌شوند که صفات و روابط میان آن‌ها را تعریف کند. فرایندهایی که اشیای داده‌ای را دستکاری می‌کنند، به شیوه‌ای مدل‌سازی می‌شوند که چگونگی تبدیل اشیای داده‌ای را به هنگام جریان یافتن آن‌ها در سیستم نشان دهند. رویکرد دوم برای مدل تحلیل، که تحلیل شیءگرا نامیده می‌شود، بر تعریف کلاس‌ها و شیوه همکاری آن‌ها با یکدیگر برای برآورده ساختن خواسته‌های مشتری تأکید دارد.

گرچه مدل تحلیلی که ما در این کتاب ارائه می‌دهیم، ویژگی‌های هر دو رویکرد را ترکیب می‌کند، تیم نرم‌افزار غالباً یک روش را انتخاب می‌کند و نمایش‌های مربوط به روش دیگر را طرد می‌کنند. مسأله این نیست که کدام روش بهتر است، بلکه مسأله این است که کدام ترکیب از نمایش‌ها بهترین مدل خواسته‌های نرم‌افزار را در اختیار طرف‌های ذی‌نفع قرار می‌دهد و کارامدترین پلی را به طراحی نرم‌افزار می‌زند.

۷-۲ مدل‌سازی جریان‌گرا (Flow-Oriented Modeling)

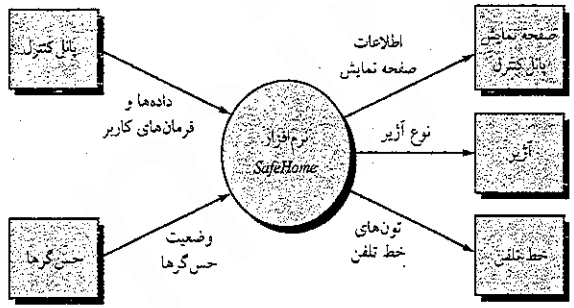
گرچه مدل‌سازی جریان‌گرا از نگاه بسیاری از مهندسان نرم‌افزار، خارج از رده به‌شمار می‌رود، همچنان یکی از پرکاربردترین نمادگذاری‌های تحلیل خواسته‌ها تاکنون بوده است.[1] *نمودار جریان داده‌ها* (DFD) و اطلاعات و نمودارهای مرتبط با آن، بخشی رسمی از UML به‌شمار نمی روند، ولی می‌توان از آن‌ها برای تکمیل نمودارهای UML بهره برد و دیدی اضافی از خواسته‌ها و جریان داده‌ها به‌دست آورد.

[1] در *تحلیل ساخت یافته* مدل‌سازی جریان داده‌ها یک فعالیت مدل‌سازی هسته‌ای به‌شمار می‌رود.

DFD نمایی از سیستم بر اساس ورودی- فرایند- خروجی به‌دست می‌دهد. یعنی اشیای داده‌ای به درون نرم‌افزار جریان پیدا می‌کنند، توسط عناصر پردازشی تبدیل می‌شوند و اشیای داده‌ای حاصل به بیرون نرم‌افزار جریان پیدا می‌کنند. اشیای داده‌ای با پیکان‌های برچسب‌دار (labeled arrows) و تبدیلات با دایره (که حباب هم نامیده می‌شوند) نمایش داده می‌شوند. DFD به شیوه‌ای سلسله مراتبی نمایش داده می‌شود. یعنی، اولین مدل در جریان داده‌ها (که گاهی DFD سطح صفر یا نمودار حیطه‌ای نامیده می‌شود) کل سیستم را نمایش می‌دهد. نمودارهای بعدی جریان داده‌ها، نمودار حیطه‌ای را پالایش می‌کنند و در هر سطح بعدی، جزئیات بیشتری افزوده می‌شود.

۷-۲-۱ ایجاد مدل جریان داده‌ها

با نمودار جریان داده‌ها می‌توانید مدل‌هایی از دامنه‌ی اطلاعاتی و دامنه عملیاتی را توسعه دهید. با پالایش DFD به سطوح بالاتری از جزئیات، می‌توانیم سیستم را به‌طور ضمنی از نظر عملیاتی تجزیه کنیم. در همان حال، پالایش DFD به پالایش داده‌ها در حین حرکت از میان فرایندهای در برگیرنده‌ی برنامه‌ی کاربردی منجر می‌شود. چند دستورالعمل ساده می‌تواند در به‌دست آوردن نمودار جریان داده‌ها کمک کند: (۱) نمودار جریان داده‌ها در سطح صفر باید نرم‌افزار/ سیستم را به‌عنوان یک حباب منفرد تصویر کند؛ (۲) ورودی و خروجی اولیه باید به دقت ذکر شود؛ (۳) پالایش باید با جداسازی فرایندهای کاندید، اشیای داده‌ای و مخزن‌های داده‌ای که قرار است در سطح بعد به نمایش در آیند، آغاز گردد؛ (۴) همه‌ی پیکان‌ها و حباب‌ها باید با نام‌های مناسب نشان‌گذاری شوند. (۵) پیوستگی جریان اطلاعات باید از سطحی به سطح دیگر حفظ گردد[1] و (۶) هر بار تنها یک حباب را باید پالایش کرد. اصولاً تمایل دارند که نمودار جریان داده‌ها را بیش از حد پیچیده کنند. این وضعیت هنگامی رخ می‌دهد که سعی کنید جزئیات زیاده از حد را خیلی زودتر از موعد نشان دهید، یا جنبه‌های روالی نرم‌افزار را به‌جای جریان اطلاعات به نمایش بگذارید.



شکل ۷-۱ DFD در سطح حیطه‌ای برای عملکرد امنیت در *SafeHome*

[1] یعنی اشیای داده‌ای که در سیستم یا در هر تبدیل در یک سطح جریان می‌یابند، باید همان اشیای داده‌ای (یا قطعات سازنده‌ی آن‌ها) باشند که در یک سطح پالایش یافته‌تر به درون تبدیل جریان می‌یابند.

اندرز

برخی پیشنهاد می‌کنند که DFD مکتبی قدیمی است و در کار مدرن جایی ندارد. این دیدگاه باعث می‌شود که از شیوه‌ی نمایشی مفید این روش در سطح تحلیل استفاده نشود. اگر DFD می‌تواند کمک کند، از آن استفاده کنید.

«هدف نمودارهای جریان داده‌ها فراهم ساختن یک پل معنایی میان کاربران و سازندگان سیستم است.»

کنت کوزار

برای نشان دادن کاربرد DFD و نمادگذاری مرتبط با آن، دوباره به قابلیت امنیت در محصول *SafeHome* می‌پردازیم. در شکل ۱-۷، نمودار DFD سطح صفر برای قابلیت امنیت نشان داده شده است. *درایه‌های خارجی* اولیه (چهارگوش‌ها) اطلاعات مورد نیاز سیستم را تولید و اطلاعات تولیدشده توسط سیستم را مصرف می‌کنند. پیکان‌های برچسب‌دار، اشیای داده‌ای یا سلسله مراتب‌هایی از اشیای داده‌ای را نشان می‌دهند. برای مثال، «**user commands and data**» شامل همه‌ی فرمان‌های پیکربندی، همه‌ی فرمان‌های فعال‌سازی/غیر فعال‌سازی، همه‌ی تعاملات متفرقه و همه‌ی داده‌هایی می‌شود که وارد می‌شوند تا فرمانی را بسط دهند یا آن را واجد شرایط لازم سازند.

اکنون DFD سطح صفر باید به یک مدل جریان داده‌ها در سطح یک، بسط داده شود. ولی چگونه باید پیش رفت؟ با دنبال کردن رویکرد پیشنهاد شده در فصل ۶ باید «تجزیه گرامری» [Abb83] را به‌کار گیرید تا به متن روایی use case‌ی که حباب سطح حیطه‌ای را توصیف می‌کند، برسید. یعنی همه‌ی اسم‌ها (و عبارت‌های اسمی) و فعل‌ها (و عبارت‌های فعلی) را در متن روایی به‌دست آمده از اولین جلسه‌ی جمع‌آوری خواسته‌های محصول *SafeHome* جدا می‌کنیم. با به خاطر آوردن متن روایی تجزیه شده در بخش ۱-۵-۶ داریم:

> قابلیت امنیت در محصول *SafeHome*، صاحبخانه را *قادر می‌سازد* که سیستم امنیتی را پس از *نصب کردن*، *پیکربندی کند*، همه‌ی حس‌گرهایی را که به سیستم امنیتی *متصل شده‌اند*، *پایش کند* و با صاحبخانه از طریق اینترنت، PC یا پانل کنترل، *تعامل کند*.
>
> در مدتی که *نصب انجام می‌شود*، از PC *SafeHome* برای *برنامه‌ریزی* و *پیکربندی* سیستم استفاده می‌شود. به هر حس‌گر یک عدد و نوع نسبت داده می‌شود، یک کلمه‌ی عبوری اصلی برای *فعال کردن* و *غیر فعال کردن* سیستم برنامه‌ریزی می‌شود و شماره تلفن(هایی) *وارد می‌شوند* تا در صورت رخ دادن یک رویداد حس‌گر، این شماره‌ها *گرفته شوند*.
>
> هنگامی که یک رویداد حس‌گر *تشخیص داده شد*، نرم‌افزار یک آژیر صوتی را به صدا *در می‌آورد* که به سیستم متصل است. پس از *مشخص‌شدن* زمان تأخیر توسط صاحبخانه در طول فعالیت‌های پیکربندی سیستم، نرم‌افزار شماره تلفن یک سرویس پایشی را می‌گیرد، اطلاعات مربوط به مکان را *ارائه می‌دهد*، و ماهیت رویداد تشخیص داده شده را *گزارش می‌کند*. این شماره تلفن هر ۲۰ ثانیه یک بار *از نو گرفته می‌شود* تا اینکه تماس تلفنی *برقرار شود*.
>
> صاحبخانه، اطلاعات امنیتی را از طریق یک پانل کنترل، PC یا مرورگر که در مجموع، واسط گفته می‌شود، *دریافت می‌کند*. این واسط، پیام‌های *درخواستی* و اطلاعات *وضعیتی* را روی پانل کنترل، PC یا پنجره مرورگر به نمایش می‌گذارد. تعامل صاحبخانه به شکل زیر خواهد بود...

در این تجزیه گرامری، فعل‌ها فرایندهای *SafeHome* هستند که می‌توان آن‌ها را با حباب نشان داد. اسم‌ها یا موجودیت‌های خارجی (چهارگوش‌ها) هستند یا اشیای داده‌ای و کنترل (پیکان ها)، یا مخزن داده‌ها (خطوط موازی). با توجه به بحثی که در فصل ۶ داشتیم، فعل‌ها و اسم‌ها را می‌توان به هم مرتبط کرد (مثلاً هر حس‌گر با یک شماره و نوع مرتبط است؛ بنابراین، *number* و *type* صفات شیء داده‌ای **sensor** هستند). بنابراین، با تجزیه گرامری متن روایی پردازش برای حبابی در هر سطح DFD، می‌توانید اطلاعات بسیار مفیدی درباره چگونگی پیشروی در پالایش بعدی ایجاد کنید. یک DFD

نکته‌ی کلیدی

پیوستگی جریان را باید با پالایش هر سطح از DFD حفظ کرد. این بدان معناست که ورودی و خروجی در یک سطح باید همانند ورودی و خروجی در سطح پالایش یافته باشد.

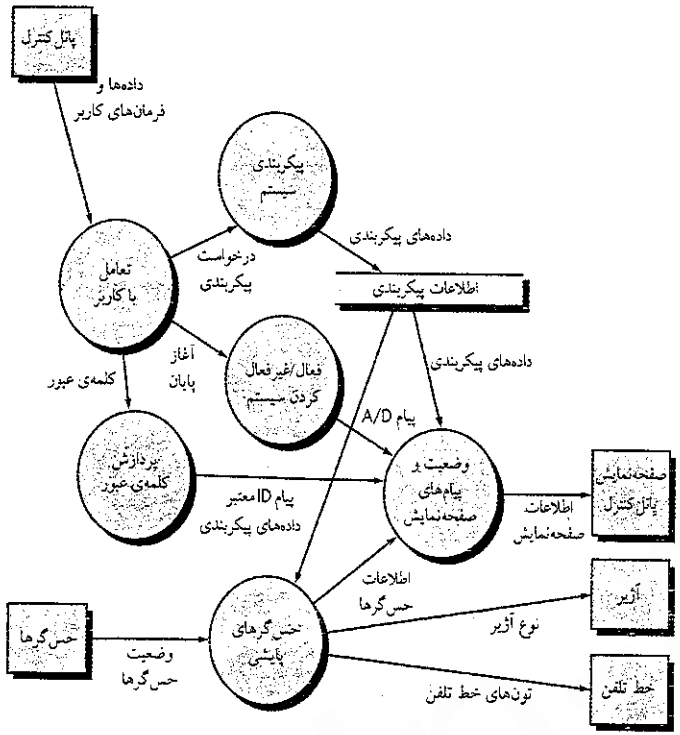
اندرز

در تجزیه‌ی گرامری امکان ارتکاب خطا وجود دارد، ولی اگر تلاش می‌کنید اشیای داده‌ای و تبدیلات روی آن‌ها را تعریف کنید می‌تواند نقطه پرش خوبی باشد.

اندرز

یقین حاصل کنید که متن روایی پردازشی که در صدد تجزیه گرامری آن هستید، همه جا در یک سطح انتزاع نوشته شده باشد.

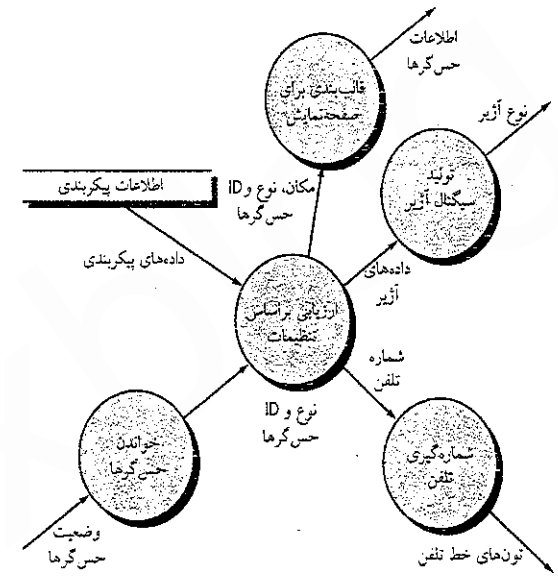
سطح یک که با استفاده از این اطلاعات تهیه شده است، در شکل ۲-۷ نشان داده شده است. فرایند سطح حیطه‌ای که در شکل ۱-۷ نشان داده شده است، به شش فرایند بسط داده شده است که از بررسی تجزیه گرامری به‌دست آمده‌اند. به‌طور مشابه، جریان اطلاعات میان فرایندها در سطح یک از این تجزیه به‌دست آمده است. علاوه بر آن، پیوستگی جریان اطلاعات بین سطوح صفر و یک حفظ شده است.



شکل ۲-۷ DFD سطح یک برای عملکرد امنیت در *SafeHome*

فرایندهای ارائه شده در DFD سطح یک را باز هم می‌توان به سطوح پایین‌تر پالایش کرد. برای مثال، فرایند *monitor sensors* را می‌توان به یک DFD سطح دو پالایش کرد (شکل ۳-۷). باز هم توجه داشته باشید که پیوستگی جریان اطلاعات بین سطوح حفظ شده است.

پالایش DFDها چندان ادامه می‌یابد که هر حباب تنها یک عملکرد را نشان می‌دهد. یعنی، تا هنگامی که فرایند نشان داده شده توسط حباب، عملی انجام دهد که به راحتی به‌عنوان یک مؤلفه‌ی برنامه قابل پیاده‌سازی باشد. در فصل ۸، مفهومی به نام *یکپارچگی* (cohesion) را مورد بحث قرار خواهیم داد که از آن می‌توان برای ارزیابی میزان توجه ویژه به یک عملکرد مفروض استفاده کرد. در حال حاضر تلاش می‌کنیم DFDها را پالایش کنیم تا اینکه هر حباب حاوی تنها یک فکر باشد.



شکل ۳-۷ DFD سطح دو که فرایند *پایش حس‌گرها* را پالایش می‌کند.

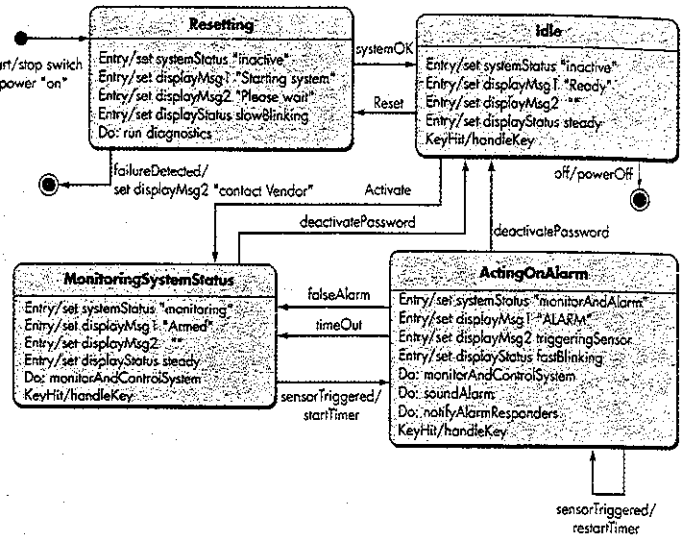
۷-۲-۲ ایجاد مدل جریان کنترل

برای برخی انواع برنامه‌های کاربردی، مدل داده‌ها و نمودار جریان داده‌ها، همه‌ی آن چیزی است که برای به‌دست‌آوردن دیدی مناسب از خواسته‌های نرم‌افزار لازم است. ولی چنان که پیش از این نیز گفته شد، گروه بزرگی از برنامه‌های کاربردی بیشتر توسط «رویدادها» اداره می‌شوند تا توسط داده‌ها، بیشتر اطلاعات کنترلی تولید می‌کنند تا گزارش و چیزهایی برای نمایش، و اطلاعات را با توجه جدی به زمان و کارایی پردازش می‌کنند. این گونه برنامه‌های کاربردی علاوه بر مدل‌سازی جریان داده‌ها به مدل‌سازی جریان کنترل هم نیاز دارند.

قبلاً گفتیم که یک آیتم کنترلی یا رویدادی به‌صورت مقداری بولی (مثلاً درست یا نادرست، خاموش یا روشن، 1 یا 0) یا فهرستی مجزا از شرط‌ها (مثلاً خالی، گیرکرده، پر) پیاده‌سازی می‌شود. برای انتخاب رویدادهای بالقوه‌ی کاندیدا، دستورالعمل‌های زیر پیشنهاد می‌شود:

- همه‌ی حس‌گرهایی را که نرم‌افزار می‌خواند، فهرست کنید.
- همه‌ی شرایط وقفه را فهرست کنید.
- همه‌ی «کلیدهایی» را که توسط اپراتور فعال می‌شوند، فهرست کنید.
- همه‌ی شرایط داده‌ها را فهرست کنید.
- با به‌کار بردن تجزیه اسمی/ فعلی که در متن روایی پردازش به‌کار برده شد، همه‌ی «آیتم‌های کنترلی» را به‌عنوان ورودی ها/خروجی‌های ممکن برای تعیین مشخصات کنترلی مرور کنید.
- رفتار سیستم را با شناسایی حالت‌ها، شناسایی چگونگی رسیدن به هر حالت و تعیین گذارهای میان حالت‌ها توصیف کنید.
- جاافتادگی‌های ممکن- خطایی بسیار رایج در مشخص کردن کنترل- را کانون توجه قرار دهید؛ برای مثال، بپرسید: «آیا راهی هست که بتوانم به این حالت برسم یا از آن خارج شوم؟»

رویدادهای بالقوه برای یک نمودار جریان کنترل، نمودار حالت یا CSPEC را چگونه انتخاب کنم؟



شکل ۴-۷ نمودار حالت برای عملکرد امنیت در *SafeHome*

از میان آیتم‌های کنترلی و رویدادی که بخشی از نرم‌افزار *SafeHome* به‌شمار می‌رود، می‌توان به **sensor event** (یعنی حس‌گری به دام افتاده است) **blink flag** (سیگنالی برای چشمک زدن روی صفحه نمایش) و **start/stop switch** (سیگنالی برای روشن یا خاموش کردن سیستم) اشاره کرد.

۷-۲-۳ مشخصات کنترل

مشخصات کنترل (CSPEC)، رفتار سیستم را (در سطحی که نسبت به آن سنجیده می‌شود) به دو شیوه به نمایش می‌گذارد.[1] CSPEC حاوی یک «نمودار حالت» است که رفتار را به‌صورت ترتیبی مشخص می‌کند. این مشخصات همچنین حاوی یک جدول فعال‌سازی برنامه‌ها- مشخصات ترکیبی رفتار- است.

در شکل ۴-۷ نمودار حالت مقدماتی[2] مدل کنترل فرایند سطح یک برای *SafeHome* به تصویر کشیده شده است. این نمودار چگونگی پاسخ دهی سیستم به رویدادها را به هنگام عبور از چهار حالت تعریف شده در این سطح نشان می‌دهد. با مرور این نمودار حالت می‌توانید رفتار سیستم را تعیین کنید و از آن مهم‌تر، ببینید که آیا در رفتار مشخص‌شده «حفره» وجود دارد یا خیر.

برای مثال، نمودار حالت (شکل ۴-۷) نشان می‌دهد که گذار از حالت **Idle** در صورتی رخ دهد که سیستم reset فعال یا خاموش شود. اگر سیستم فعال شود (یعنی سیستم آژیر روشن شود)، گذار به حالت **Monitoring System Status** رخ می‌دهد، پیام‌های صفحه نمایش، به‌صورت نشان داده شده تغییر می‌کند و فرایند *monitorAndControlSystem* فراخوانی می‌شود. دو گذار در خارج از حالت **MonitoringSystemStatus** رخ می‌دهد- (۱) هنگامی که سیستم غیر فعال شود،گذاری در بازگشت

[1] نمادهای دیگر مدل‌سازی رفتاری در بخش ۳-۷ ارائه خواهد شد.

[2] نمادگذاری نمودار حالت به کار رفته در این کتاب نمادگذاری UML همخوانی دارد. «نمودار گذار حالت» در تحلیل ساخت یافته در دسترس هست ولی قالب UML در محتوا و نمایش اطلاعات برتری دارد.

به حالت **Idle** رخ می‌دهد؛ (۲) هنگامی که حس‌گری به حالت **ActingOnAlarm** برده شود. همه‌ی گذارها و محتوای همه‌ی حالت‌ها طی بازبینی در نظر می‌شوند.

یک شیوه نسبتاً متفاوت برای نمایش رفتار، جدول **فعال‌سازی فرایندها** (PAT) است. PAT اطلاعات موجود در نمودار حالت را در حیطه‌ی فرایندها و نه حالت‌ها، نشان می‌دهد. یعنی، این جدول نشان می‌دهد که کدام فرایندها (حباب‌ها) در مدل جریان، هنگامی فراخوانی می‌شود که رویدادی رخ دهد. طراحی که باید یک فایل اجرایی ایجاد کند تا فرایندهای نشان داده شده در این سطح را بسازد، از PAT می‌تواند به‌عنوان دستورالعملی برای این منظور استفاده کند. در شکل ۵-۷، PAT مربوط به مدل جریان سطح یک برای نرم‌افزار *SafeHome* نشان داده شده است.

input events						
sensor event	0	0	0	0	1	0
blink flag	0	0	1	1	0	0
start stop switch	0	1	0	0	0	0
display action status complete	0	0	0	1	0	0
in-progress	0	0	1	0	0	0
time out	0	0	0	0	0	1
output						
alarm signal	0	0	0	0	1	0
process activation						
monitor and control system	0	1	0	0	1	1
activate/deactivate system	0	1	0	0	0	0
display messages and status	1	0	1	1	1	1
interact with user	1	0	0	1	0	1

شکل ۵-۷ جدول فعال‌سازی فرایند برای عملکرد امنیت در *SafeHome*

CSPEC، رفتار سیستم را توصیف می‌کند، ولی درباره کارکرد داخلی فرایندهایی که در نتیجه‌ی این رفتار فعال می‌شوند، هیچ اطلاعاتی نمی‌دهد. نمادگذاری مدل‌سازی که این اطلاعات را فراهم می‌سازد، در بخش ۴-۲-۷ بحث خواهد شد.

۴-۲-۷ مشخصات فرایندها

مشخصات فرایندها (PSPEC) در توصیف همه‌ی فرایندهای مدل جریان که در سطح نهایی پالایش ظاهر می‌شوند، کاربرد دارد. محتوای تعیین مشخصات فرایندها می‌تواند شامل متن روایی، توصیفی از زبان طراحی برنامه (PDL)[1] برای الگوریتم فرایند، معادلات ریاضی، جداول، یا نمودارهای فعالیت‌های UML باشد. با فراهم ساختن یک PSPEC برای همراه کردن با هر حباب در مدل جریان، می‌توانید مجموعه‌ای از ریزمشخصاتی ایجاد کنید که برای طراحی مؤلفه‌ای از نرم‌افزار، که آن حباب را پیاده‌سازی می‌کند، به‌عنوان دستورالعمل به‌کار می‌رود.

برای نشان دادن کاربرد PSPEC، تبدیل *process password* را در نظر بگیرید که در مدل جریان برای محصول *SafeHome* نشان داده شده است (شکل ۲-۷). PSPEC مربوط به عملکرد ممکن است به شکل زیر باشد:

[1] زبان طراحی برنامه‌ها (PDL) نحو و قالب زبان‌های برنامه‌نویسی را با متون روایی در هم می‌آمیزد تا جزئیات طراحی روالی فراهم گردد. درباره تفصیل در فصل ۱۰ بحث خواهیم کرد.

نکته‌ی کلیدی

PSPEC «ریزمشخصاتی» برای هر تبدیل در پایین‌ترین سطح پالایش در DFD است.

SafeHome

مدل‌سازی جریان داده‌ها

صحنه: اتاقک جیمی، پس از پایان آخرین جلسه جمع‌آوری خواسته‌ها

بازیگران: جیمی، وینود و اد-همه‌ی اعضای تیم نرم‌افزار SafeHome

گفتگو:

(جیمی مدل‌های نشان داده شده در شکل‌های ۱-۷ تا ۵-۷ را رسم کرده است و در حال نشان دادن آن‌ها به اد و وینود است.)

جیمی: وقتی در دانشکده درس مهندسی نرم‌افزار را گرفتم این‌ها را به ما یاد دادند. استان می‌گفت یک قدری قدیمی شده ولی راستش را بخواهید، به من در روشن کردن اوضاع کمک می‌کند.

اد: عالی است. ولی من اینجا هیچ کلاس یا شیءای نمی‌بینم.

جیمی: نه... این فقط یک مدل جریان است که قدری چیزهای رفتاری هم چاشنی آن شده.

وینود: پس این DFDها یک دید IPO از نرم‌افزار ارائه می‌کنند. درست است؟

اد: IPO؟

وینود: ورودی- پردازش- خروجی. DFDها در واقع باید گویا باشند... اگر به آن‌ها نگاه کنید نحوه‌ی جریان پیدا کردن اطلاعات از میان سیستم تبدیل آن‌ها می‌بینید.

اد: انگار که هر حباب را می‌توانیم به یک مؤلفه اجرایی تبدیل کنیم... حداقل در پایین‌ترین سطح DFD این طور به نظر می‌رسد.

جیمی: قسمت جالب آن همین جاست. در واقع، راهی برای ترجمه‌ی DFDها به معماری طراحی وجود دارد.

اد: واقعاً؟

جیمی: بله، ولی اول باید یک مدل کامل از خواسته‌ها توسعه بدهیم و این آن مدل کامل نیست.

وینود: جف. این تازه قدم اول است ولی ما باید عناصر مبتنی بر کلاس و همچنین جنبه‌های رفتاری را در نظر بگیریم هر چند که نمودار حالت و PAT هم این کار را انجام می‌دهند.

اد: ما یک عالم کار داریم و وقت زیادی هم نمانده. (داگ- مدیر مهندسی نرم‌افزار- وارد اتاقک می‌شود.)

داگ: خب. چند روز آینده را صرف توسعه‌ی مدل خواسته‌ها می‌کنید، نه؟

جیمی (مغرور به نظر می‌رسد): ما قبلاً کار را شروع کرده ایم.

داگ: خوب است، یک عالم کار داریم و وقت زیادی هم باقی نمانده.

(سه مهندس نرم‌افزار به هم نگاهی می‌کنند و لبخند می‌زنند.)

PSPEC: پردازش کلمه‌ی عبور (در قاب کنترل). تبدیل process password اعتبارسنجی کلمه‌ی عبور در قاب کنترل را برای قابلیت امنیت در *SafeHome* انجام می‌دهد. *process password* یک کلمه‌ی عبور چهار رقمی از تابعی به نام interact with user دریافت می‌کند. این کلمه‌ی عبور نخست با کلمه‌ی

عبور نگهداری شده در داخل سیستم مقایسه می‌شود. اگر کلمه‌ی عبور اصلی درست باشد، <valid id message=true> به تابع message and status display تحویل می‌شود. اگر کلمه‌ی عبور درست نبود، چهار رقم با جدول کلمات عبور ثانویه مقایسه می‌شود (که ممکن است به مهمانان خانه و/یا کارگرانی که در غیاب صاحبخانه نیاز به ورود به خانه دارند، داده شده باشد). اگر کلمه‌ی عبور با درایه‌ای از این جدول همخوانی داشت، <valid id message=true> به تابع *message and status display* تحویل می‌شود. اگر همخوانی وجود نداشته باشد، <valid id message=false> به تابع *message and status display* تحویل خواهد شد.

اگر در این مرحله، جزئیات الگوریتمی بیشتری مورد نیاز باشد، نمایشی از زبان طراحی برنامه نیز ممکن است به‌عنوان بخشی از PSPEC گنجانده شود. ولی بسیاری بر این باورند که نسخه‌ی PDL باید تا شروع‌شدن طراحی قطعه به تعویق افتد.

ابزارهای نرم‌افزاری

تحلیل ساخت یافته

هدف: مهندس نرم‌افزار به کمک ابزارهای تحلیل ساخت یافته می‌تواند مدل‌های داده ای، مدل‌های جریان و مدل‌های رفتاری را به شیوه‌ای ایجاد کند که سازگاری و چک کردن پیوستگی و بسط و ویرایش آسان را میسر سازد. مدل‌های ایجاد شده با استفاده از این ابزارها دیدی از نمایش تحلیل می‌دهند و به حذف خطاها قبل از انتشار یافتن آن‌ها در طراحی یا در پیاده‌سازی، کمک می‌کنند.

مکانیک: ابزارهای این گروه از یک «دیکشنیری داده‌ها» به‌عنوان بانک اطلاعاتی اصلی برای توصیف تمامی اشیای داده‌ای استفاده می‌کنند. پس از اینکه درایه‌های دیکشنیری تعریف شدند، نمودارهای موجودیت- ارتباط را می‌توان ایجاد کرد و سلسله مراتب‌های اشیا را توسعه داد. ابزارهای ایجاد نمودارهای جریان داده‌ها ایجاد آسان این مدل گرافیکی را میسر می‌سازند و همچنین ویژگی‌هایی برای ایجاد PSPECها و CSPECها فراهم می‌سازند. ابزارهای تحلیلی همچنین به مهندس نرم‌افزار در ایجاد مدل‌های رفتاری با استفاده از نمودار حالت به‌عنوان نمادگذاری عملیاتی کمک می‌کنند.

ابزارهای نمونه

MacA&D، *WinA&D*، که توسط نرم‌افزاری Excel (www.excelsoftware.com) توسعه یافته است، مجموعه‌ای از ابزارهای ساده و ارزان برای تحلیل و طراحی را برای ماشین‌های Mac و Windows فراهم می‌سازد.

MetaCase Workbench که توسط شرکت MetaCase Consulting توسعه یافته است (www.metacase.com) شبه ابزاری است که در تعریف روش تحلیل یا طراحی (از جمله تحلیل ساخت یافته) و مفاهیم، قواعد، نمادها و مولدهای آن به‌کار می‌رود.

System Architect که توسط Popkin Software (www.popkin.com) توسعه یافته است، گستره وسیعی از ابزارهای تحلیل و طراحی از جمله ابزارهای مربوط به مدل‌سازی داده‌ها و تحلیل ساخت یافته را فراهم می‌سازد.

واکنش نرم‌افزار به یک رویداد خارجی را چگونه مدل‌سازی کنم؟

۷-۳ ایجاد مدل رفتاری

نمادگذاری مدل‌سازی که تا این نقطه بحث شد، عناصر ایستای مدل خواسته‌ها را نمایش می‌دهد. اکنون زمان آن فرا رسیده است که به رفتار پویای سیستم یا محصول گذار کنیم. برای این منظور، می‌توانیم رفتار سیستم را به‌صورت تابعی از زمان و رویدادهای مشخص نمایش دهیم.

مدل‌های رفتاری نشان می‌دهد که نرم‌افزار چگونه به رویدادها یا محرک‌های خارجی پاسخ می‌دهند. برای ایجاد این مدل، باید مراحل زیر را اجرا کنید:

۱. ارزیابی همه‌ی موارد برای درک کامل تعامل‌های داخل سیستم.
۲. شناسایی رویدادهایی که این تعامل‌ها را اداره می‌کنند و درک چگونگی ارتباط این رویدادها با اشیای مشخص.
۳. ایجاد یک دنباله یا توالی برای هر use case.
۴. ساخت یک نمودار حالت برای سیستم.
۵. مرور مدل رفتاری برای نشان دادن درستی و سازگاری.

هر کدام از این مراحل را در بخش‌های بعدی به تفصیل شرح خواهیم داد.

۷-۳-۱ شناسایی رویدادها به کمک use case

در فصل ۶ دانستید که use case، دنباله‌ای از فعالیت‌ها را نشان می‌دهد که کنش‌گر و سیستم را شامل می‌شوند. به‌طور کلی، هر گاه که سیستم و کنش‌گری به تبادل اطلاعات بپردازند، یک رویداد رخ می‌دهد. در بخش ۷-۲-۳ نشان دادیم که رویداد، اطلاعات تبادل‌شده نیست بلکه این حقیقت است که، اطلاعات تبادل‌شده است.

use case برای نقاط تبادل اطلاعات بررسی می‌شود. برای روشن‌شدن مطلب، use case بخشی از قابلیت امنیت در *SafeHome* را در نظر می‌گیریم.

صاحبخانه از صفحه کلید استفاده می‌کند و کلمه‌ی عبور چهار رقمی را وارد می‌کند. این کلمه‌ی عبور با کلمه‌ی عبور معتبر نگهداری شده در سیستم مقایسه می‌شود. اگر کلمه‌ی عبور نادرست باشد، قاب کنترل یک بار به‌صدا در می‌آید و خود را برای ورودی جدید، reset می‌کند. اگر کلمه‌ی عبور درست بود، قاب کنترلی برای کنش بعدی منتظر می‌ماند.

بخش‌هایی از use case که زیر آن‌ها خط کشیده شده است، رویدادها را نشان می‌دهند. کنش‌گر باید برای هر رویدادی شناسایی شود؛ اطلاعاتی که تبادل می‌شود باید ذکر شود و هر شرط یا قید وبندی باید فهرست شود.

به‌عنوان مثالی از یک رویداد رایج، عبارت «صاحبخانه از صفحه کلید استفاده می‌کند و کلمه‌ی عبور چهار رقمی را وارد می‌کند» را در نظر بگیرید که زیر آن خط کشیده شده است. در حیطه‌ی مدل خواسته‌ها، شیء **Homeowner**[1] رویدادی را به شیء **ControlPanel** مخابره می‌کند. این رویداد را می‌توان *password entered* نام نهاد. اطلاعاتی که در اینجا مخابره می‌شود، همان چهار رقم تشکیل دهنده‌ی کلمه‌ی عبور است ولی این، بخش ضروری مدل رفتاری نیست. ذکر این نکته حائز اهمیت

[1] در این مثال، فرض می‌کنیم که هر کاربر (صاحبخانه) که با *SafeHome* تعامل می‌کند، دارای کلمه‌ی عبوری برای شناسایی است و لذا شیئی قانونی است.

است که برخی رویدادها تأثیری صریح و روشن بر جریان کنترل در use case می‌گذارند. برای مثال، رویداد *password entered* به‌طور صریح جریان کنترل use case را تغییر نمی‌دهد، ولی نتایج رویداد *password compared* (که از تعامل «این کلمه‌ی عبور با کلمه‌ی عبور معتبر نگهداری شده در سیستم مقایسه می‌شود» حاصل شده است) بر جریان کنترل و اطلاعات در نرم‌افزار *SafeHome* تأثیری آشکار و صریح دارد.

زمانی که همه‌ی رویدادها شناسایی شدند، به اشیای موجود تخصیص داده می‌شوند. اشیاء می‌توانند مسؤول ایجاد رویدادها باشند (مثل **Homeowner** که رویداد *password entered* را ایجاد می‌کند) یا رویدادهایی را که در جای دیگر رخ می‌دهند، شناسایی کنند (مثل **ControlPanel** که نتیجه دودویی رویداد *password compared* شناسایی می‌کند).

۲-۳-۷ نمایش حالت‌ها (State Representation)

در حیطه‌ی مدل‌سازی رفتاری، حالت‌ها را از دو نظر باید مشخص کرد: (۱) حالت هر کلاس در زمانی که به وظیفه خود عمل می‌کند و (۲) حالت سیستم از دید ناظر خارجی در زمانی که به وظیفه خود عمل می‌کند.[1]

حالت یک کلاس هر دو خصوصیت انفعالی (passive) و فعال (active) را به خود می‌گیرد [cha93]. حالت انفعالی صرفاً حالت فعلی همه‌ی صفت‌های یک شیء است. برای مثال، حالت انفعالی کلاس **Player** (در بازی کامپیوتری فصل ۶) شامل صفت‌های فعلی *position* و *orientation* برای **Player** و نیز سایر ویژگی‌های **Player** می‌شود که به بازی مربوط می‌شوند (مثل صفتی که *magic wishes remaining* را نشان می‌دهد). حالت *فعال شیء*، وضعیت فعلی شیء را به هنگام قرار گرفتن در معرض تبدیل یا پردازش مستمر نشان می‌دهد. کلاس **Player** ممکن است حالت‌های فعالی را داشته باشد که به دنبال خواهد آمد: در *حال حرکت*، در *حال سکون*، *مجروح*، در *حال مداوا*، *به‌دام‌افتاده*، *گم‌شده* و غیره. رویدادی باید رخ دهد تا گذار از یک حالت فعال به حالت فعال دیگر انجام شود.

نکته‌ی کلیدی

سیستم، حالت‌هایی دارد که رفتار خاص قابل مشاهده از بیرون را به نمایش می گذارد؛ کلاس دارای حالت‌هایی است که رفتار آن را به هنگام اجرای وظایف به نمایش می گذارد.

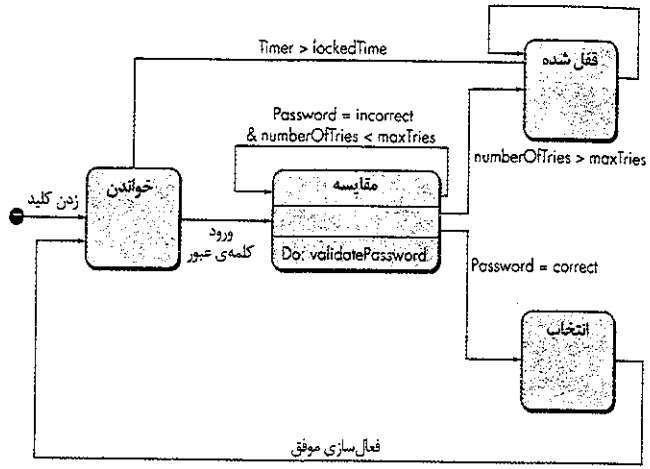
دو نمایش رفتاری متفاوت در پاراگراف‌های زیر بحث خواهد شد. اولی چگونگی تغییر یک کلاس را بر اساس رویدادهای خارجی نشان می‌دهد و دومی نشان‌گر رفتار نرم‌افزار به‌عنوان تابعی از زمان است.

نمودارهای حالت برای کلاس‌های تحلیل. یک مؤلفه از مدل رفتاری، نمودار حالت UML است[2] که حالت‌های فعال هر کلاس و رویدادهایی را نشان می‌دهد که باعث تغییر در این حالت‌های فعال می‌شوند. در شکل ۶-۷ نمودار حالت برای شیء **ControlPanel** در عملکرد امنیت ***SafeHome*** نشان داده شده است.

هرکدام از پیکان‌های شکل ۶-۷، گذاری از یک حالت فعال شیء به حالت فعال دیگر را نشان می‌دهد. برچسب‌های روی هر پیکان، رویدادی را نشان می‌دهند که گذار را آغاز می‌کنند. گرچه مدل

[1] نمودارهای حالت ارائه شده در فصل ۶ و در بخش ۲-۳-۷، حالت سیستم را به تصویر می‌کشند. بحث ما در این بخش بر حالت هر کلاس در مدل تحلیل تأکید دارد.

[2] اگر با UML آشنایی ندارید، معرفی مختصری از این نمادگذاری مهم مدل‌سازی در پیوست ۱ ارائه شده است.



شکل ۶-۷ نمودار حالت برای کلاس ControlPanel.

حالت‌های فعال، دیدی مفید از «تاریخچه‌ی حیات» شیء می‌دهد، می‌تواند اطلاعات دیگری را هم مشخص کند تا از رفتار شیء درکی عمیق‌تر فراهم گردد. علاوه بر تعیین مشخصات رویدادی که باعث رخ دادن گذار می‌شود، می‌توانید یک نگهبان (guard) و کنش مشخص کنید [Cha93]. *نگهبان* یک شرط بولی است که باید برآورده شود تا گذار رخ دهد. برای مثال، نگهبان گذار از حالت «reading» به حالت «comparing» در شکل ۶-۷ را می‌توان با بررسی use case تعیین کرد:

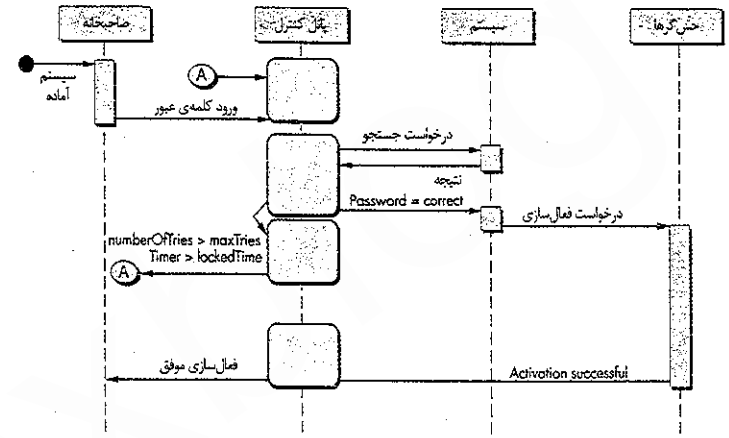
if (password input = 4 digits) then compare to stored password

به‌طور کلی، نگهبان مربوط به یک گذار، معمولاً به مقدار یک یا چند صفت از شیء بستگی دارد. به بیان دیگر، نگهبان به حالت انفعالی شیء بستگی دارد.

هر کنش، همزمان با ساختار گذار یا به‌عنوان نتیجه‌ای از آن رخ می‌دهد و عموماً شامل یک یا چند عملیات (مسؤولیت) از شیء می‌شود. برای مثال کنش متصل به رویداد *password entered* (شکل ۶-۷) عملیاتی است با نام *validatepassword()* که به شیء **password** دستیابی دارد و مقایسه‌ای رقم به رقم انجام می‌دهد تا کلمه‌ی عبور واردشده را اعتبارسنجی کند.

نمودارهای ترتیب (Sequence Diagram). دومین نوع نمایش رفتار، که نمودار ترتیب در UML نامیده می‌شود، نشان می‌دهد که رویدادها چگونه باعث گذار یک شیء به شیء دیگر می‌شوند. پس از آن که رویدادها با بررسی use case شناسایی شدند، مدل‌ساز، یک نمودار ترتیب ایجاد می‌کند که نشان می‌دهد رویدادها چگونه باعث ایجاد جریان از یک شیء به شیء دیگر به‌عنوان تابعی از زمان می‌شوند. در اصل، نمودار ترتیب، نسخه‌ای خلاصه شده از use case است. این نمودار، کلاس‌های کلیدی و رویدادهایی را نشان می‌دهد که باعث جریان یافتن رفتار از کلاسی به کلاس دیگر می‌شوند.

در شکل ۷-۷ بخشی از یک نمودار ترتیب برای قابلیت امنیت در *SafeHome* نشان داده شده است. هر کدام از پیکان‌ها نشان‌گر یک رویداد (به‌دست‌آمده از use case) بوده چگونگی کانال‌زدن رویداد بین اشیا برای جریان یافتن رفتار را نشان می‌دهد. زمان به‌صورت عمودی (بالا به پایین) سنجیده می‌شود و مستطیل‌های باریک عمودی، زمان صرف شده در پردازش یک فعالیت را نشان می‌دهند. حالت‌ها را می‌توان در راستای یک خط زمانی عمودی نمایش داد.



شکل ۷-۷ بخشی از نمودار ترتیب برای عملکرد امنیت در *SafeHome*.

رویداد نخست، *system ready*، از محیط خارجی به‌دست می‌آید و جریان رفتار را به سوی شیء **Homeowner** هدایت می‌کند. صاحبخانه کلمه‌ی عبوری را وارد می‌کند. یک رویداد *request lookup* به **System** تحویل می‌شود که کلمه‌ی عبور را در یک بانک اطلاعاتی ساده جستجو کرده نتیجه‌ای را (*موفق* یا *ناموفق*) به **ControlPanel** بر می‌گرداند (که اکنون در حالت *comparing* قرار دارد). کلمه‌ی عبور معتبر به رویداد *password=correct* برای **System** منجر می‌شود که آن هم به نوبه‌ی خود، **Sensors** را با رویداد *request activation* فعال می‌کند. سرانجام، کنترل با رویداد *request activation* دوباره به صاحبخانه باز گردانده می‌شود.

پس از این‌که نمودار ترتیب کاملی ایجاد شد، همه‌ی رویدادهایی را که باعث گذار میان اشیای سیستم می‌شوند، می‌توان در مجموعه‌ای از رویدادهای ورودی و رویدادهای خروجی جمع‌آوری کرد. این اطلاعات در ایجاد طراحی مؤثر برای سیستمی که قرار است ساخته شود، مفید واقع می‌شوند.

نکته‌ی کلیدی

برخلاف نمودار حالت که رفتار را بدون توجه به کلاس‌های موجود نشان می‌دهد، نمودار ترتیب، رفتار را با توصیف چگونگی حرکت کلاس‌ها از حالتی به حالت دیگر به نمایش می‌گذارد.

۷-۴ الگوهایی برای مدل‌سازی خواسته‌ها

الگوهای نرم‌افزاری، سازوکارهایی هستند برای کسب آگاهی از دامنه، به شیوه‌ای که بتوان هنگام مواجهه با مسأله‌ای جدید، آن‌ها را دوباره به‌کار برد. در برخی موارد، دانش حاصل از یک دامنه، برای مسأله‌ای جدید در همان دامنه به‌کار گرفته می‌شود. در سایر موارد، دانش حاصل از دامنه توسط یک الگو را می‌توان از طریق مقایسه با یک دامنه‌ی کاربرد متفاوت به‌کاربرد.

نویسنده‌ی اصلی یک الگوی تحلیل، الگو را «ایجاد نمی کند» بلکه آن را به موازات اجرای کار مهندسی خواسته‌ها، کشف می‌کند. هنگامی که الگو کشف شد، با توصیف صریح مسأله‌ی کلی که الگو در مورد آن کاربرد دارد، راهکار تجویزی، فرضیات و قیدوبندهای استفاده از الگو در عمل و غالباً اطلاعات دیگری درباره الگو، نظیر ایجاد انگیزه و نیروهای محرکه برای استفاده از الگو، بحث درباره مزایا و معایب الگو و ارجاع به مثال‌های شناخته شده‌ای از به‌کارگیری آن الگو در کاربردهای عملی، مستندسازی می‌شود.» [Dev01]

ابزارهای نرم‌افزاری

مدل‌های تحلیل عمومی در UML

هدف: ابزارهای مدل‌سازی تحلیل، توانایی توسعه‌ی مدل‌های مبتنی بر سناریو، مدل‌های مبتنی بر کلاس و مدل‌های رفتاری را با استفاده از نمادگذاری UML فراهم می‌آورند.

مکانیک: ابزارهای این گروه گستره‌ی وسیعی از نمودارهای UML مورد نیاز برای ساخت مدل تحلیل را پشتیبانی می‌کنند (این ابزارها مدل‌سازی طراحی را نیز پشتیبانی می‌کنند). این ابزارها علاوه بر ایجاد نمودار، (۱) همه‌ی نمودارهای UML را از نظر سازگاری و صحت، چک می‌کنند، (۲) پیوندهایی برای طراحی و کدنویسی فراهم می‌سازند، (۳) یک بانک اطلاعاتی می‌سازند که مدیریت و ارزیابی مدل‌های بزرگ UML برای سیستم‌های پیچیده را امکان پذیر می‌سازد.

ابزارهای نمونه

ابزارهای زیر، گستره‌کاملی از نمودارهای UML لازم برای مدل‌سازی تحلیل را پشتیبانی می‌کنند.

Argo UML ابزاری با منبع باز است که در **argouml.tigris.org** می‌توان آن را یافت.

Enterprise Architect که توسط SparxSystem (**www.sparxsystem.com.au**) توسعه یافته است.

Power Designer که توسط Sybase توسعه یافته است (**www.sybase.com**)

Rational Rose که توسط IBM توسعه یافته است (**www01.ibm.com/softwarerational**)

System Architect که توسط Popkin Software توسعه یافته است (**www.popkin.com**)

UML Studio که توسط Pragsoft Corporation توسعه یافته است (**www.pragsoft.com**)

Visio که توسط Microsoft توسعه یافته است (**www.microsoft.com**)

Visual UML که توسط Visual Object Modelers (**www.visualuml.com**) توسعه یافته‌است

در فصل ۵، مفهوم الگوهای تحلیل را معرفی کردیم و خاطر نشان ساختیم که این الگوها راهکاری را نشان می‌دهند که غالباً شامل یک کلاس، یک عملکرد یا رفتار در داخل دامنه‌ی کاربرد است. این الگو را می‌توان هنگام اجرای مدل‌سازی خواسته‌ها برای برنامه‌ی کاربردی در حیطه‌ی یک دامنه‌ی معین، دوباره استفاده کرد.[1] الگوهای تحلیل در یک مخزن ذخیره می‌شوند، به‌طوری که اعضای تیم نرم‌افزار بتوانند با به‌کارگیری تسهیلات جستجو، آن‌ها را بیابند و دوباره استفاده کنند. هنگامی که الگوی مناسب انتخاب شد، با ارجاع به نام الگو، به مدل خواسته‌ها افزوده می‌شود.

۷-۴-۱ کشف الگوهای تحلیل

مدل خواسته‌ها از گستره وسیعی از عناصر تشکیل می‌شود: مبتنی بر سناریو (use case)، داده محور (مدل داده‌ها)، مبتنی بر کلاس، جریان‌گرا و رفتاری. هر کدام از این عناصر، مسأله را از دیدگاهی متفاوت بررسی می‌کنند و هر کدام برای کشف الگوهایی که ممکن است در سرتاسر یک دامنه‌ی کاربرد رخ دهند، یا بر اساس مقایسه، در میان دامنه‌های کاربردی متفاوت، رخ دهند.

[1] بحثی عمقی از به کارگیری الگوها در طی طراحی نرم‌افزار در فصل ۱۲ بحث خواهد شد.

اصلی‌ترین عنصر در توصیف مدل خواسته‌ها، use case است. در حیطه‌ی این بحث، مجموعه‌ای یکپارچه از use case می‌تواند به‌عنوان مبنا و اساسی برای کشف یک یا چند الگوی تحلیل عمل کند. *الگوی تحلیل معناشناختی* (SAP)[1] الگویی است که مجموعه کوچکی از use case یکپارچه را توصیف می‌کند که به همراه یکدیگر، یک کاربرد کلی و پایه‌ای را توصیف می‌کنند [Fer00].

use case مقدماتی زیر را در نظر بگیرید که برای کنترل و پایش دوربین نمای واقعی و حس‌گر مجاورتی (proximity) در یک خودرو ضروری است.

use case: پایش حرکت دنده عقب

توصیف: هنگامی که وسیله نقلیه در حالت دنده عقب قرار داده شد، نرم‌افزار کنترل‌کننده با استفاده از دوربینی که در پشت خودرو نصب شده است، تصویر ویدیویی آن محل را روی صفحه نمایش جلو داشبورد نشان می‌دهد. این نرم‌افزار کنترل علاوه بر آن، انواع خطوط تعیین فاصله و جهت‌یاب را روی صفحه به نمایش در می‌آورد تا راننده بتواند جهت خود را هنگام حرکت به طرف عقب، حفظ کند. نرم‌افزار کنترلی علاوه بر آن، یک «حس‌گر مجاورتی» را پایش می‌کند تا اگر شیء‌ای در فاصله‌ی سه‌متری آن مشاهده شد، وجود آن را تشخیص دهید. اگر حس‌گر، شیئی را در x‌متری پشت خودرو دید (که x بر اساس سرعت خودرو تعیین می‌شود) به‌طور خودکار ترمز می‌کند تا خودرو متوقف شود.

این use case شامل انواع قابلیت‌های عملیاتی‌ای می‌شود که طی جمع‌آوری و مدل‌سازی خواسته‌ها پالایش خواهند شد و جزئیات آن‌ها مشخص می‌شود (در قالب مجموعه‌ای از use caseهای یکپارچه). در هر سطحی از جزئیات که باشیم، use case یک SAP ساده و در عین حال با استفاده‌ای گسترده را پیشنهاد می‌کند- پایش و کنترل حس‌گر و محرک‌ها در یک سیستم فیزیکی به کمک نرم‌افزار. در این مورد، «حس‌گرها» اطلاعات مربوط به مجاورت و اطلاعات تصویری را فراهم می‌آورند. «محرک» سیستم ترمز خودکار خودرو است (که در صورت نزدیک‌شدن بیش از حد شیء به هنگام عبور از کنار آن فراخوانی می‌شود). ولی در یک مورد عمومی تر، الگویی با استفاده‌ی گسترده‌تر کشف می‌شود.

نرم‌افزار در بسیاری از دامنه‌های کاربرد متفاوت برای پایش حس‌گرها و کنترل محرک‌های فیزیکی ضروری است. از این رو، یک الگوی تحلیل که خواسته‌های کلی را برای این توانایی توصیف کند، به‌طور گسترده قابل استفاده خواهد بود. الگویی با نام **Actuator-Sensor** به‌عنوان بخشی از مدل خواسته‌ها برای *SafeHome* قابل استفاده است که در بخش ۲-۴-۷ بحث می‌شود.

۲-۴-۷ مثالی از الگوی خواسته‌ها: Actuator-Sensor[2]

یکی از خواسته‌های امنیتی در *SafeHome*، توانایی آن در پایش حس‌گرهای امنیتی (مثلاً حس‌گرهای نشان دهنده‌ی ورود غیرمجاز، آتش‌سوزی، دود یا گاز CO یا حس‌گرهای آب) است. شکل بسط‌یافته و اینترنتی *SafeHome*، توانایی کنترل حرکت دوربین‌های امنیتی (مثلاً زوم یا تغییر زاویه) از داخل منزل را دارد. این بدان معناست که نرم‌افزار SafeHome باید حس‌گرها و «محرک‌های» گوناگونی (مثلاً سازوکارهای کنترل دوربین) را مدیریت کند.

[1] Semantic Analysis Pattern

[2] این بخش از مرجع [Kon02] و با کسب اجازه از مؤلفان آن برگرفته شده است.

کنراد و چنگ [Kon02] یک الگوی خواسته‌ها با نام **Actuator-Sensor** پیشنهاد کرده‌اند که دستورالعملی مناسب برای مدل‌سازی این خواسته در نرم‌افزار *SafeHome* فراهم می‌سازد. نسخه‌ی خلاصه‌شده‌ای از الگوی **Actuator-Sensor** که در آغاز برای کاربردهای خودکارسازی توسعه یافت، به‌صورت زیر است:

نام الگو. Actuator-Sensor

هدف. تعیین مشخصات انواع گوناگون حس‌گرها و محرک‌ها در سیستم‌های تعبیه‌شده.

انگیزه. سیستم‌های تعبیه‌شده معمولاً دارای انواع حس‌گرها و محرک‌ها هستند. این حس‌گرها و محرک‌ها همگی به‌طور مستقیم یا غیرمستقیم به واحد کنترل متصل‌اند. گرچه بسیاری از حس‌گرها و محرک‌ها ظاهری کاملاً متفاوت دارند، رفتار آن‌ها به‌قدر کافی مشابه هست که در داخل یک الگو سازمان دهی شوند. این الگو چگونگی تعیین مشخصات حس‌گرها و محرک‌های یک سیستم، از جمله صفت‌ها و عملیات‌ها را نشان می‌دهد. الگوی **Actuator-Sensor** از سازوکار *واکنشی* (درخواست صریح برای اطلاعات) برای **PassiveSensors** (حس گرهای انفعالی) و از سازوکار انتشار (پخش اطلاعات) برای **Active Sensors** (حس گرهای فعال) استفاده می‌کند.

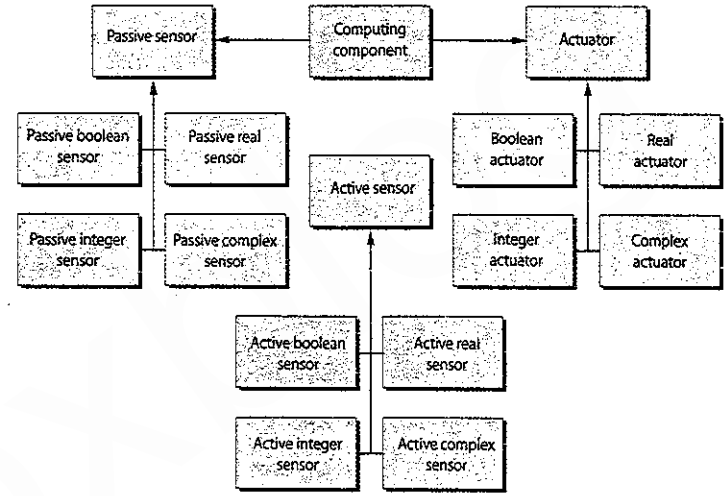
قید و بندها

- هر حس‌گر منفعلی باید یک روش برای خواندن ورودی حس‌گر و صفت‌های نشان دهنده‌ی مقدار حس‌گر داشته باشد.
- هر حس‌گر فعالی باید هنگام تغییر مقدار، توانایی پخش پیام‌های به‌هنگام‌سازی را داشته باشد.
- هر حس‌گر فعالی باید یک تیک حیاتی[1] (یعنی پیام وضعیتی که در یک چارچوب زمانی مشخص صادر می‌شود) برای آشکارسازی موارد سوء عملکرد ارسال کند.
- هر محرک باید یک روش برای فراخوانی پاسخ مناسبی داشته باشد که توسط **ComputingComponent** تعیین می‌شود.
- هر حس‌گر و محرک باید دارای تابعی باشد که برای چک کردن حالت عملیاتی خودش پیاده‌سازی شده باشد.
- هر حس‌گر و محرک باید قادر به آزمایش اعتبار مقادیر دریافتی یا ارسالی باشد و بتواند درصورت قرار گرفتن مقادیر در خارج از مشخصات، حالت عملیاتی خود را تعیین کند.

قابلیت استفاده (Applicability). در هر سیستمی که در آن حس‌گرها و محرک‌های چندگانه وجود دارند، مفید واقع می‌شود.

ساختار (Structure). نمودار کلاس‌های UML برای الگوی **Actuator-Sensor** در شکل ۸-۷ نشان داده شده است. **Actuator**، **PassiveSensor** و **ActiveSensor**، کلاس‌های انتزاعی‌اند و با حروف ایتالیک نشان داده شده‌اند. چهار نوع حس‌گر و محرک در این الگو وجود دارد. کلاس‌های **Boolean**، **Integer** و **Real** متداول‌ترین انواع حس‌گرها و محرک‌ها را نشان می‌دهند، کلاس‌های پیچیده، حس‌گرها یا محرک‌هایی هستند که از مقادیری استفاده می‌کنند که به آسانی برحسب انواع داده‌های اولیه قابل نمایش نیستند؛ مثلاً در یک دستگاه راداری. با این وجود، این دستگاه‌ها هنوز باید واسط را

[1] life tick



شکل ۸-۷ نمودار ترتیب برای الگوی Actuator-Sensor.

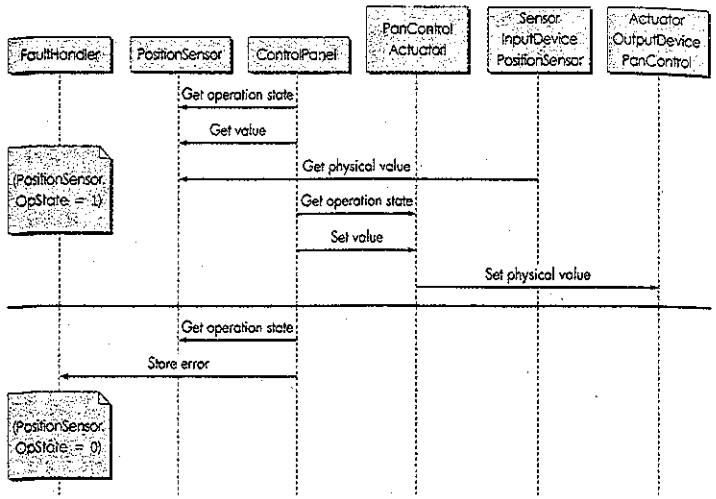
از کلاس‌های انتزاعی به ارث ببرند زیرا این کلاس‌ها دارای قابلیت‌های پایه‌ای از قبیل درخواست حالت‌های عملیاتی هستند.

رفتار (Behavior). در شکل ۹-۷ یک نمودار ترتیب UML برای مثالی از کاربرد الگوی **Actuator-Sensor** در قابلیت کنترل موقعیت دوربین‌های امنیتی در *SafeHome* داده شده است. در اینجا، **ControlPanel**[1] از یک حس‌گر (حس گری در موقعیت انفعالی) و یک محرک (کنترل زاویه دوربین) درخواست می‌کند که حالت عملیاتی را برای اهداف عیب‌یابی قبل از خواندن یا تعیین یک مقدار چک کند. پیام‌های *Set Physical Value* (تعیین مقدار فیزیکی) و *Get Physical Value* (گرفتن مقدار فیزیکی) پیام‌های بین دو شیء نیستند. درعوض، این پیام‌ها تعامل میان دستگاه‌های فیزیکی سیستم و همتاهای نرم‌افزاری آن‌ها را توصیف می‌کنند. در بخش زیرین نمودار، پایین خط افقی، **PositionSensor** گزارش می‌کند که حالت عملیاتی، صفر است. سپس **ComputingComponent** (که به‌عنوان **ControlPanel** نشان داده می‌شود) کد خطایی را برای شکست حس‌گر موقعیتی به **FaultHandler** ارسال می‌کند تا برای چگونگی تأثیرگذاری این خطا بر سیستم و کنش‌های مورد نیاز تصمیم‌گیری کند. این شیء، داده‌ها را از حس‌گر می‌گیرد و پاسخ لازم برای محرک‌ها را محاسبه می‌کند.

مشارکت‌کنندگان (Partners). در این بخش از توصیف الگوها «ریز کلاس‌های اشیای گنجانده شده در الگوی خواسته‌ها فهرست می‌شود» [Kon02] و مسؤولیت هر کلاس/ شیء توصیف می‌گردد (شکل ۸-۷). یک فهرست مختصر در زیر داده شده است:

- انتزاع **PassiveSensor**: واسطی برای حس‌گرهای منفعل تعریف می‌کند.
- **PassiveBooleanSensor**: حس‌گرهای بولی منفعل را تعریف می‌کند.

[1] در الگوی اولیه از عبارت کلی **ComputingComponent** استفاده می‌شود.



شکل ۹-۷ نمودار کلاس‌های UML برای الگوی Actuator-Sensor.

- **PassiveIntegerSensor**: حس‌گرهای عدد صحیح منفعل را تعریف می‌کند.
- **PassiveRealSensor**: حس‌گرهای عدد حقیقی منفعل را تعریف می‌کند.
- انتزاع **ActiveSensor**: واسطی برای حس‌گرهای فعال تعریف می‌کند.
- **ActiveBooleanSensor**: حس‌گرهای بولی فعال را تعریف می‌کند.
- **ActiveIntegerSensor**: حس‌گرهای عدد صحیح فعال را تعریف می‌کند.
- **ActiveRealSensor**: حس‌گرهای عدد حقیقی فعال را تعریف می‌کند.
- انتزاع **Actuator**: واسطی برای محرک تعریف می‌کند.
- **BooleanActuator**: محرک‌های بولی را تعریف می‌کند.
- **IntegerActuator**: محرک‌های عدد صحیح را تعریف می‌کند.
- **RealActuator**: محرک‌های عدد حقیقی را تعریف می‌کند.
- **ComputingComponent**: بخش مرکزی کنترل گر؛ داده‌ها را از حس‌گرها می‌گیرد و پاسخ لازم را برای محرک‌ها محاسبه می‌کند.
- **ActiveComplexSensor**: حس‌گرهای فعال پیچیده همان قابلیت‌های عملیاتی کلاس انتزاعی **ActiveSensor** را دارند ولی صفات و متدهای اضافی و با جزئیات بیشتری باید برای آن‌ها مشخص شود.
- **PassiveComplexSensor**: حس‌گرهای منفعل پیچیده، همان قابلیت‌های عملیاتی کلاس انتزاعی **PassiveSensor** را دارند ولی صفات و متدهای اضافی و با جزئیات بیشتری باید برای آن‌ها مشخص گردد.
- **ComplexActuator**: محرک‌های پیچیده، همان قابلیت عملیات‌های پایه‌ای کلاس **Actuator** را دارند ولی صفات و متدهای اضافی و با جزئیات بیشتری باید برای آن‌ها مشخص گردد.

همکاری‌ها (Collaborations). در این بخش چگونگی تعامل اشیا و کلاس‌ها با یک دیگر و چگونگی انجام مسؤولیت‌ها توصیف می‌شود.

- هنگامی که قرار باشد **ComputingComponent** مقدار یک **PassiveSensor** را بهنگام کند، با ارسال پیام مناسب و درخواست مقدار، وضعیت حس‌گرها را جویا می‌شود.
- وضعیت **ActiveSensor**ها مورد سؤال قرار نمی‌گیرد بلکه انتقال مقادیر حس‌گرها به واحد محاسبه‌کننده را با استفاده از روش مناسب برای تعیین مقدار در **ComputingComponent** آغاز می‌کنند. این اشیا در چارچوب زمانی مشخص‌شده، حداقل یک بار «تیک حیاتی» ارسال می‌کنند تا مُهر زمانی (time stamp) خود را با زمان ساعت سیستم بهنگام کنند.
- هنگامی که **ComputingComponent** نیاز به تعیین مقدار یک محرک داشته باشد، مقدار را به محرک ارسال می‌کند.
- **ComputingComponent** می‌تواند حالت عملیاتی حس‌گرها و محرک‌ها را با به‌کارگیری متدهای مناسب، پرسش و تنظیم کند. اگر حالت عملیاتی برابر با صفر باشد، خطایی به **FaultHandler** ارسال می‌شود، که کلاسی حاوی متدهای لازم برای کار با پیام‌های خطا از قبیل شروع به‌کار یک سازوکار بازیابی اثربخش‌تر یا دستگاه پشتیبان است. اگر بازیابی میسر نباشد، سیستم فقط می‌تواند از آخرین مقدار معلوم، برای حس‌گر یا مقدار پیش‌فرض استفاده کند.
- **ActiveSensors** متدها را برای اضافه یا حذف کردن آدرس‌ها یا گستره‌ای از آدرس مؤلفه‌هایی ارائه می‌دهد که می‌خواهند پیام‌ها را در مورد تغییر مقدار دریافت کنند.

پیامدها (Consequences)

۱. کلاس‌های حس‌گر و محرک واسط مشترک دارند.

۲. صفات کلاس‌ها تنها از طریق پیام‌ها قابل دستیابی‌اند و کلاس است که تصمیم می‌گیرد آیا پیام را بپذیرد یا خیر. برای مثال، اگر مقدار یک محرک بالای مقداری بیشینه قرار داده شود، کلاس محرک ممکن است پیام را نپذیرد یا ممکن است از یک مقدار بیشینه‌ی پیش‌فرض استفاده کند.

۳. پیچیدگی سیستم به‌طور بالقوه به دلیل یکنواختی واسط‌ها برای محرک‌ها و حس‌گرها کاهش می‌یابد.

این توصیف از الگوی خواسته‌ها ممکن است ارجاع‌هایی به سایر الگوهای طراحی و خواسته‌ها داشته باشد.

۷-۵ مدل‌سازی خواسته‌ها برای برنامه‌های تحت وب[1]

کسانی که در وب کار می‌کنند، غالباً به تحلیل خواسته‌ها برای برنامه‌های تحت وب، با دیده‌ی تردید می‌نگرند و استدلال آن‌ها هم این است که «گذشته از همه‌ی حرف‌ها، فرایند کار در وب باید چابک باشد و تحلیل، کاری است که زمان می‌برد. درست همان زمانی که باید به کار طراحی و ساخت برنامه‌های تحت وب بپردازیم، از سرعت ما کم می‌کند.»

شکی نیست که تحلیل خواسته‌ها زمان می‌برد، ولی حل مسأله‌ی اشتباهی، از آن هم بیشتر زمان می‌برد. پیش روی هر برنامه‌نویس تحت وب این پرسش ساده مطرح می‌شود: آیا مطمئنی که خواسته‌های مسأله را می‌دانی؟ اگر پاسخ به صراحت مثبت باشد، در آن صورت گذشتن از مدل‌سازی

[1] این بخش از منبع [Pre08] و با کسب اجازه آورده شده است.

خواسته‌ها امکان‌پذیر است، ولی اگر پاسخ منفی باشد، مدل‌سازی خواسته‌ها را باید انجام داد.

۷-۵-۱ چقدر تحلیل کافی است؟

میزان تأکید ورزیدن بر مدل‌سازی خواسته‌ها برای برنامه‌های تحت وب، به عوامل زیر بستگی دارد:

- اندازه و پیچیدگی نسخه‌ی برنامه‌های تحت وب.
- تعداد طرف‌های ذی‌نفع (تحلیل می‌تواند به شناسایی خواسته‌های متضاد از منابع متفاوت کمک کند).
- اندازه تیم برنامه‌های تحت وب.
- میزان همکاری قبلی اعضای تیم برنامه‌های تحت وب (تحلیل می‌تواند به درک مشترکی از پروژه کمک کند).
- میزان بستگی مستقیم موفقیت سازمان به موفقیت برنامه‌های تحت وب.

عکس نکات فوق از این قرار است که با کوچکترشدن پروژه، کم‌شدن تعداد ذی‌نفع‌ها، یکپارچگی بیشتر تیم توسعه و حیاتی نبودن پروژه، بهتر است تحلیل کمتری صورت گیرد.

گرچه تحلیل مسأله *قبل* از شروع طراحی خوب است، این درست نیست که *کل* تحلیل باید قبل از *کل* طراحی انجام شود. در واقع، طراحی بخش مشخصی از برنامه‌های تحت وب، مستلزم تحلیل آن دسته از خواسته‌هایی است که تنها بر آن بخش تأثیر می‌گذارند. به‌عنوان مثالی از پروژه‌ی *SafeHome* می‌توانید عناصر زیبایی‌شناختی کل وب سایت (چیدمان‌ها، رنگ‌بندی‌ها و غیره) را به‌طور معتبر طراحی کنید، بدون اینکه خواسته‌های عملیاتی را برای قابلیت‌های تجارت الکترونیکی، تحلیل کرده باشید. فقط کافی است آن بخش از مسأله را تحلیل کنید که با کار طراحی برای تحویل نسخه‌ای از پروژه در ارتباط است.

۷-۵-۲ ورودی در مدل‌سازی خواسته‌ها

نسخه‌ی چابکی از فرایند کلی نرم‌افزار را، که در فصل ۲ بحث شد، می‌توان در مهندسی برنامه‌های تحت وب به‌کاربرد. این فرایند شامل فعالیت برقراری ارتباط می‌شود که در آن گروه‌های ذی‌نفع و کاربر، حیطه‌ی تجاری، اهداف اطلاعاتی و کاربردی تعیین شده، خواسته‌های عمومی برنامه‌های تحت وب و سناریوهای کاربرد- اطلاعاتی که ورودی مدل‌سازی خواسته‌ها می‌شوند- شناسایی خواهند شد. این اطلاعات به شکل توصیف‌های زبان طبیعی، خلاصه‌ی طرح‌های تقریبی و سایر نمایش‌های رسمی و غیر رسمی ارائه می‌شوند.

تحلیل، این اطلاعات را می‌گیرد و با استفاده از یک الگوی نمایش معین، به آن ساختار می‌دهد و سپس مدل‌های محکم‌تری را به‌عنوان خروجی ایجاد می‌کند. مدل خواسته‌ها، ساختار واقعی مسأله را با جزئیات آن به نمایش می‌گذارد و دیدی از راهکار ارائه می‌دهد.

با قابلیت ACS-DCV در محصول *SafeHome* (پایش دوربین‌ها) در فصل ۶ آشنا شدیم. این قابلیت هنگام معرفی، نسبتاً واضح به‌نظر می‌رسید و به‌عنوان بخشی از یک use case با قدری تفصیل شرح داده شد (بخش ۱-۲-۶). ولی با بررسی دوباره use case، اطلاعاتی آشکار می‌شود که قبلاً جای آن‌ها خالی بوده است یا مبهم و ناواضح بوده‌اند.

برخی از جنبه‌های این اطلاعاتِ جاافتاده، به‌طور طبیعی طی طراحی نمایان می‌شوند. مثال‌ها می‌تواند شامل چیدمان مشخص دکمه‌های عملیاتی، شکل و شمایل زیبایی‌شناختی، اندازه نمای دوربین‌ها، قرار دادن نمای دوربین‌ها و نقشه ساختمان در صفحه، یا حتی موارد فرعی نظیر حداکثر و حداقل طول کلمات عبور شود. برخی از این جنبه‌ها، تصمیم‌گیری‌های طراحی (نظیر چیدمان دکمه‌ها) و سایرجنبه‌ها، خواسته‌هایی هستند (نظیر طول کلمات عبور)که تأثیری بنیادی برکار طراحی اولیه ندارند.

ولی ممکن است برخی از اطلاعات جاافتاده، واقعاً بر خودِ طراحی تأثیر بگذارند و بیشتر به درک واقعی خواسته‌ها مرتبط باشند. برای مثال،

پرسش ۱: خروجی دوربین‌های *SafeHome* از چه تفکیکی برخودار است؟

پرسش ۲: اگر شرایط هشدار در هنگام پایش دوربین‌ها پیش آید، چه خواهد شد؟

پرسش ۳: سیستم چگونه می‌تواند با دوربین‌هایی کار کند که زاویه و زوم آن‌ها قابل تغییر است؟

پرسش ۴: چه اطلاعاتی باید همراه با نمای دوربین فراهم آورده شود؟ (برای مثال، مکان؟ زمان/تاریخ؟ آخرین دستیابی قبلی؟)

هیچ یک از این پرسش‌ها در توسعه‌ی اولیه use case، شناسایی یا در نظر گرفته نمی شود و با این حال پاسخ‌ها اثری چشمگیر بر جنبه‌های متفاوت طراحی دارند.

بنابراین، منطقی است نتیجه بگیریم که گرچه فعالیت برقراری ارتباط، بستری مناسب برای درک و شناخت فراهم می‌سازد، تحلیل خواسته‌ها این درک و شناخت را با فراهم آوردن تفسیر اضافی، پالایش می‌کنند. با ترسیم ساختار مسأله به‌عنوان بخشی از مدل خواسته‌ها، ناگزیر پرسش‌هایی پیش می‌آید. همین پرسش‌ها هستند که شکاف‌ها را پر می‌کنند- یا در برخی موارد، ما را در پیدا کردن شکاف‌ها در وهله‌ی نخست یاری می‌دهند.

به‌طور خلاصه، ورودی‌های مدل خواسته‌ها، اطلاعاتی هستند که طی فعالیت برقراری ارتباط به‌دست می‌آیند-و شامل هر چیزی، از نامه‌های الکترونیکی غیر رسمی گرفته تا یک خلاصه پروژه‌ی مشروح با سناریوهای کاربرد جامع و مشخصات کامل محصول را در بر می‌گیرند.

۳-۵-۷ خروجی‌های مدل‌سازی خواسته‌ها

تحلیل خواسته‌ها یک سازوکار منضبط برای ارائه و ارزیابی محتوا و قابلیت‌های عملیاتی برنامه‌های تحت وب، شیوه‌های تعامل فراروی کاربر و محیط و زیرساخت قرار گرفتن برنامه‌های تحت وب فراهم می‌آورد.

هرکدام از این خصوصیات را می‌توان به‌عنوان بخشی از مدل‌هایی نشان داد که تحلیل خواسته‌های برنامه‌های تحت وب را به شیوه‌ای ساخت یافته میسر می‌سازند. در حالی‌که مدل‌های مشخص، بستگی چشمگیری به ماهیت برنامه‌ی تحت وب دارند، آن‌ها را به پنج گروه می‌توان طبقه‌بندی کرد:

- **مدل محتوا**- طیف کاملی از محتوایی را که قرار است برنامه‌ی تحت وب فراهم آورد، مشخص می‌کند. این محتوا عبارت است از داده‌های متنی،گرافیکی و تصاویر، ویدیو، و داده‌های صوتی.
- **مدل تعامل‌ها**- شیوه تعامل کاربران با برنامه‌ی تحت وب را توصیف می‌کند.
- **مدل عملیاتی**- عملیاتی را تعریف می‌کند که روی محتوای برنامه‌ی تحت وب انجام می‌شوند و قابلیت‌های پردازشی مستقل از محتوا و در عین حال ضروری برای کاربر را توصیف می‌کنند.

- **مدل گشت و گذار**- راهبرد کلی گشت و گذار را برای برنامه‌ی تحت وب تعریف می‌کند.
- **مدل پیکربندی**- محیط و زیرساختی را توصیف می‌کند که برنامه‌ی تحت وب در آن قرار داده می‌شود.

هر کدام از این مدل‌ها را می‌توانید با به‌کارگیری یک الگوی نمایش (که غالباً «زبان» نامیده می‌شود) توسعه دهید تا محتوا و ساختار آن را بتوان به راحتی به اطلاع اعضای تیم مهندسی وب و سایر طرف‌های ذی‌نفع رساند. در نتیجه، فهرستی از مسائل کلیدی (مانند خطاها، جاافتادگی‌ها، ناسازگاری‌ها، پیشنهادهایی برای بهسازی یا اصلاح، نقاط وضوح) شناسایی ورودی آن‌ها کار می‌شود.

۴-۵-۷ مدل محتوا برای برنامه‌های تحت وب

مدل محتوا حاوی عناصری ساختاری است که دیدی مهم از خواسته‌های محتوای برنامه‌های تحت وب در اختیار قرار می‌دهد. این عناصر ساختاری شامل اشیای محتوایی و همه‌ی کلاس‌های تحلیل می‌شوند- موجودیت‌های قابل مشاهده از دید کاربر که در تعامل کاربر بابرنامه‌های تحت وب، ایجاد یا دستکاری می‌شوند.[1]

محتوا را می‌توان قبل از پیاده‌سازی برنامه‌های تحت وب، به موازات ساخته‌شدن برنامه‌های تحت وب، یا مدت‌ها پس از عملیاتی‌شدن برنامه‌ی تحت وب توسعه داد. در هر حال، این محتوا از طریق مرجع گشت و گذار در ساختار کلی برنامه‌ی تحت وب گنجانده می‌شود. یک *شیء محتوایی* ممکن است توصیفی متنی از یک محصول، مقاله‌ای در توضیح یک رویداد خبری، عکسی از یک رویداد ورزشی، پاسخ یک کاربر در میزگرد، نمایشی پویانمایی شده از لوگوی یک شرکت، یک قطعه ویدیو از سخنرانی، یا صداگذاری روی مجموعه‌ای از اسلایدها باشد. اشیای محتوایی را می‌توان به‌عنوان فایل‌های مجزا نگهداری کرد، به‌طور مستقیم در صفحات وب تعبیه کرد، یا به‌صورت پویا از یک بانک اطلاعاتی به‌دست آورد. به عبارت دیگر، شیء محتوایی هر آیتمی از اطلاعات یکپارچه است که قرار است به‌کاربر نهایی ارائه شود.

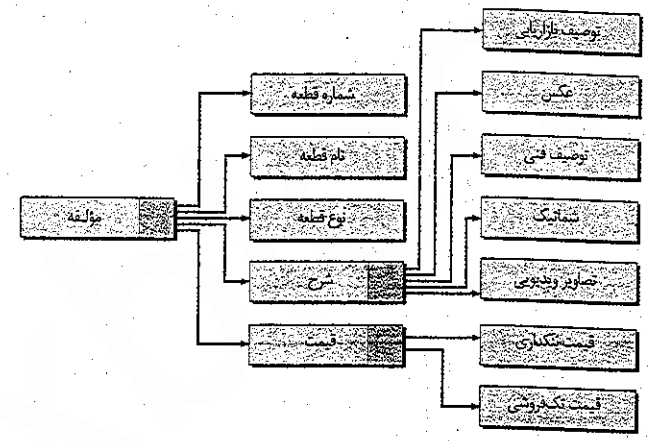
اشیای محتوایی را می‌توان به‌طور مستقیم از روی use case و با بررسی توصیف سناریو برای ارجاع‌های مستقیم و غیر مستقیم به محتوا تعیین کرد. برای مثال برنامه‌ی تحت وبی که *SafeHome* را پشتیبانی می‌کند، در **SafeHomeAssured.com** قرار داده می‌شود. یک use case با عنوان *خرید انتخابی مؤلفه‌های SafeHome* سناریوی لازم برای خرید یک مؤلفه *SafeHome* را توصیف می‌کند و حاوی جمله زیر است:

> من قادر به دریافت اطلاعات توصیفی و قیمت‌گذاری برای هر کدام از مؤلفه‌های محصول خواهم بود.

مدل محتوا باید قادر به توصیف شیء محتوای **Component** باشد. در بسیاری موارد، فهرست ساده‌ای از اشیای محتوایی، در کنار توصیف مختصر هر شیء، برای تعریف خواسته‌های مربوط به محتوایی که قرار است طراحی و پیاده‌سازی شوند، کفایت می‌کند. ولی در برخی موارد، مدل محتوا ممکن است از تحلیلی غنی‌تر بهرمند شود که به‌طور گرافیکی روابط میان اشیای محتوایی و/یا سلسه مراتب محتوای یک برنامه‌ی تحت وب را به نمایش می‌گذارد.

---

[1] کلاس‌های تحلیل در فصل ۶ بحث و بررسی شدند.



شکل ۱۰-۷ درخت داده‌ها برای مؤلفه‌ی SafeHomeAssured.com

برای مثال، درخت *داده‌های* ایجادشده برای مؤلفه‌های **SafeHomeAssured.com** را در نظر بگیرید (شکل ۱۰-۷) [Sri01]. این درخت، سلسله مراتب اطلاعاتی را نشان می‌دهد که در توصیف یک مؤلفه به‌کار می‌رود. آیتم‌های داده‌ای ساده یا مرکب (یک یا چند مقدار داده ای) به‌صورت مستطیل‌های هاشور نخورده نشان داده می‌شوند. اشیای محتوایی به‌صورت مستطیل‌های هاشورخورده نمایش داده می‌شوند. در این شکل *description* توسط پنج شیء محتوایی تعریف می‌شود (مستطیل‌های هاشورخورده). در برخی موارد، یک یا چند شیء از این اشیا با بسط یافتن درخت داده‌ها پالایش می‌شوند.

برای هر محتوایی که از چند شیء محتوایی و آیتم داده‌ای تشکیل می‌شود، یک درخت داده‌ها می‌توان ایجاد کرد. درخت داده‌ها برای تعریف روابط سلسله مراتبی میان اشیای محتوایی و فراهم ساختن ابزاری برای مرور محتوا توسعه می‌یابد به‌طوری که جاافتادگی‌ها و ناسازگاری‌ها قبل از شروع طراحی کشف شوند. به‌علاوه، درخت داده‌ها به‌عنوان مبنایی برای طراحی محتوا عمل می‌کند.

۵-۵-۷ مدل تعامل برای برنامه‌های تحت وب

گستره‌ی وسیعی از برنامه‌های تحت وب، «گفتگو» میان کاربر نهایی و قابلیت عملیاتی، محتوا یا رفتار برنامه را میسر می‌سازند. این گفتگو را می‌توان با به‌کارگیری یک *مدل* تعامل توصیف کرد که از یک یا چند عنصر زیر تشکیل می‌شود: (۱) use case، (۲) نمودارهای ترتیب، (۳) نمودارهای حالت[1] و/یا (۴) نمونه‌های اولیه‌ی واسط کاربری.

در بسیاری از نمونه‌ها، مجموعه‌ای از use case برای توصیف تعامل در سطحی تحلیلی کفایت می‌کند (پالایش جزئیات بیشتر طی مرحله‌ی طراحی وارد خواهد شد). با این وجود، هنگامی که ترتیب تعامل‌ها پیچیده و شامل کلاس‌های تحلیل چند گانه و وظایف فراوان باشد، گاهی به تصویر کشیدن آن با استفاده از یک شکل نموداری‌تر، ارزش‌مند است.

[1] نمودارهای ترتیب و نمودارهای حالت با استفاده از نمادگذاری UML مدل‌سازی می‌شوند. نمودارهای حالت در بخش ۳-۷ شرح داده شدند. برای جزئیات بیشتر، پیوست ۱ را ببینید.

چیدمان واسط کاربری، محتوایی که ارائه می‌دهد، سازوکارهای تعاملی که پیاده‌سازی می‌کند و زیبایی شناسی کلی ارتباطات میان برنامه‌ی تحت وب و کاربر، تأثیر زیادی بر رضایت کاربر و موفقیت کلی برنامه‌ی تحت وب دارد. گرچه می‌توان استدلال کرد که ایجاد نمونه‌ی اولیه‌ای از واسط کاربری، یک فعالیت طراحی به‌شمار می‌رود، اجرای آن طی مرحله‌ی ایجادِ مدل تحلیل، ایده‌ی خوبی به نظر می‌رسد. هر چه زودتر بتوان نمایش فیزیکی واسط کاربری را بازبینی کرد، احتمال رسیدن کاربران نهایی به آن چه می‌خواهند، بیشتر می‌شود. طراحی واسط‌های کاربری را به تفصیل در فصل ۱۱ بحث خواهیم کرد.

از آن‌جا که ابزارهای ساخت برنامه‌های تحت وب، بسیار زیاد، نسبتاً ارزان و دارای قدرت عملیاتی بالا هستند، بهترین کار، ایجاد نمونه‌ی اولیه واسط با استفاده از این گونه ابزارهاست. در این نمونه‌ی اولیه باید پیوندهای اصلی مربوط به گشت و گذار در برنامه‌های تحت وب پیاده‌سازی شود و چیدمان کلی صفحه به همان صورتی که قرار است ساخته شود، به نمایش در آید. برای مثال، اگر پنج عملکرد اصلی قرار است در اختیار کاربر نهایی قرار داده شود، نمونه‌ی اولیه باید آن‌ها را به همان صورتی نشان دهد که کاربر در نخستین بار ورود به برنامه‌ی تحت وب خواهد دید. آیا پیوندهای گرافیکی فراهم خواهد آمد؟ منوی گشت و گذار کجا نمایش داده خواهد شد؟ کاربر چه اطلاعات دیگری را خواهد دید؟ نمونه‌ی اولیه باید به پرسش‌هایی از این دست پاسخ دهد.

۶-۵-۷ مدل عملیاتی برای برنامه‌های تحت وب

بسیاری از برنامه‌های تحت وب، گستره‌ی وسیعی از قابلیت‌های محاسباتی و دستکاری داده‌ها را تحویل می‌دهند که می‌تواند به‌طور مستقیم با محتوا (چه استفاده از آن و چه تولید آن) همراه باشد و غالباً هدف اصلی تعامل میان کاربر و برنامه‌ی تحت وب است. به همین دلیل، خواسته‌های عملیاتی را باید تحلیل و در صورت نیاز مدل‌سازی کرد.

مدل *عملیاتی* به دو عنصر پردازشی در برنامه‌های تحت وب می‌پردازد که هر یک سطح متفاوتی از انتزاع روالی را نشان می‌دهد: (۱) قابلیت‌های عملیاتی که توسط برنامه‌ی تحت وب به‌کاربر نهایی ارائه می‌شوند و او قادر به دیدن آن‌هاست و (۲) عملیاتی که در داخل کلاس‌های تحلیل قرار دارند و رفتارهای مرتبط با هر کلاس را پیاده‌سازی می‌کنند.

قابلیت‌هایی که کاربر قادر به دیدن آنهاست، شامل کلیه‌ی قابلیت‌های پردازشی می‌شوند که به‌طور مستقیم توسط کاربر آغاز می‌شوند. برای مثال، یک برنامه‌ی تحت وب مالی ممکن است انواع قابلیت‌های مالی (از قبیل محاسبه‌ی پس انداز دانشجویی یا پس انداز بازنشستگی) را پیاده‌سازی کند. این قابلیت‌ها ممکن است واقعاً با استفاده از عملیات داخل کلاس‌های تحلیل پیاده‌سازی شوند، ولی از دید کاربر نهایی، عملکرد (یا به عبارت صحیح تر، داده‌هایی که این عملکرد ارائه می‌دهد)، پیامد قابل مشاهده است.

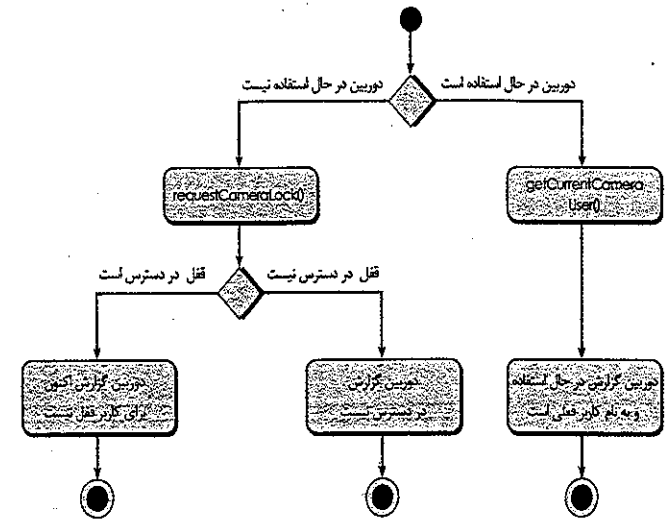
در سطح پایین‌تری از انتزاع روالی، مدل خواسته‌ها پردازشی را توصیف می‌کند که عملیات‌های کلاس تحلیل، آن را انجام می‌دهند. این عملیات‌ها صفات کلاس را دستکاری می‌کنند و به عنوان کلاس‌هایی که برای رسیدن به رفتار مورد نیاز همکاری می‌کنند، در نظر گرفته می‌شوند.

سطح انتزاع هر چه که باشد، از نمودار فعالیت‌های UML می‌توان برای نمایش دادن جزئیات پردازشی استفاده کرد. در سطح تحلیل، از نمودارهای فعالیت‌ها می‌توان تنها هنگامی استفاده کرد که

عملکرد نسبتاً پیچیده باشد. بیشتر پیچیدگی‌ها در بسیاری از برنامه‌های تحت وب، نه در قابلیتِ فراهم‌شده بلکه در ماهیت اطلاعاتی که قابل دستیابی‌اند و شیوه‌های دستکاری آن‌ها مشاهده می‌شود.

مثالی از یک عملکرد نسبتاً پیچیده برای **SafeHomeAssured.com** را می‌توان در use case با عنوان *«دریافت توصیه‌هایی برای چیدمان حس‌گرها برای فضای امن»* یافت. کاربر قبلاً برای فضایی که قرار است پایش شود، چیدمانی تهیه کرده است و در این use case، آن چیدمان را انتخاب و مکان‌هایی برای قرار دادن حس‌گرها در این چیدمان درخواست می‌کند. **SafeHomeAssured.com** با نمایش گرافیکی چیدمان و ارائه اطلاعات اضافی روی مکان‌های توصیه شده برای حس‌گرها به این درخواست پاسخ می‌دهد. تعامل بسیار ساده است، محتوا قدری پیچیده‌تر است ولی عملکرد نهفته در پس آن بسیار پیچیده است. سیستم باید تحلیل نسبتاً پیچیده‌ای روی چیدمان انجام دهد تا تعیین کند که آیا چیدمان حس‌گرها بهینه هست یا خیر. باید ابعاد اتاق‌ها و مکان درها و پنجره‌ها را بررسی کند و این‌ها را با قابلیت‌ها و مشخصات حس‌گرها هماهنگ کند. این اصلاً وظیفه‌ی کوچکی نیست! مجموعه‌ای از نمودارهای فعالیت را می‌توان برای توصیف پردازش برای این use case به‌کار برد.

مثال دوم، use case *کنترل دوربین‌ها* است. در این use case تعامل نسبتاً ساده است، ولی با توجه به این که این عملیات «ساده» به ارتباطات پیچیده با دستگاه‌های دور دست و قابل دستیابی از طریق اینترنت نیاز دارد، پتانسیلی برای پیچیدگی وجود دارد. یک پیچیدگی دیگر ممکن است هنگامی رخ دهد که چند نفر همزمان بخواهند یک حس‌گر را پایش و/یا کنترل کنند.



شکل ۱۱-۷ نمودار فعالیت‌ها برای عملیات *takeControlOfCamera()*

در شکل ۱۱-۷ نمودار فعالیت‌ها برای عملیات *takeControlOfCamera()* نشان داده شده است که بخشی از کلاس تحلیل **Camera** است و در داخل use case کنترل دوربین‌ها استفاده شده است. لازم به ذکر است که دو عملیات اضافی در جریان روالی فراخوانده می‌شود: *requestCameraLock()* که سعی می‌کند دوربین را برای کاربر قفل کند و *getCurrentCameraUser()* که نام کاربری را بازیابی می‌کند که هم‌اکنون در حال کنترل دوربین است. جزئیات ساختِ نشان دهنده‌ی این عملیات‌ها

فراخوانی می‌شوند و پرداختن به جزئیات واسط برای هر کدام از این عملیات‌ها تا شروع طراحی به تعویق می‌افتد.

۷-۵-۷ مدل‌های پیکربندی برای برنامه‌های تحت وب

در برخی موارد، مدل پیکربندی چیزی بیش از فهرست صفات مربوط به کلاینت و صفات مربوط به سرور نیست. ولی برای اکثر برنامه‌های تحت وب پیچیده، انواع پیچیدگی‌های پیکربندی (مثلاً توزیع بار در میان چند سرور، قراردادن معماری‌ها در نهان گاه‌ها، بانک‌های اطلاعاتی دور دست، سرورهای چندگانه‌ای که به اشیای گوناگون موجود در یک صفحه وب سرویس می‌دهند) ممکن است بر تحلیل و طراحی تأثیر بگذارد. در وضعیت‌هایی که باید معماری پیچیده‌ای برای پیکربندی در نظر گرفته شود، می‌توان از نمودار استقرار UML استفاده کرد.

برای **SafeHomeAssured.com**، عملکرد و محتوای عمومی را باید طوری مشخص کرد که در میان همه‌ی مرورگرهای وب (یعنی آن‌ها که بیش از یک درصد از سهم بازار را در اختیار دارند[1]) قابل دستیابی باشد. برعکس، می‌توان پذیرفت که عملکرد پایشی و کنترل پیچیده‌تر (که تنها در دسترس کاربران **Homeowner** است) به مجموعه کوچک‌تری از مرورگرها محدود گردد. این مدل پیکربندی برای **SafeHomeAssured.com**، عملیات متقابل میان بانک‌های اطلاعاتی موجود و برنامه‌های پایش‌گر را نیز مشخص می‌سازد.

۷-۵-۸ مدل‌سازی گشت و گذار

در مدل‌سازی گشت و گذار، چگونگی سیاحت گروه‌های کاربری از یک عنصر برنامه‌ی تحت وب به عنصر دیگر در نظر گرفته می‌شود. مکانیک این گشت و گذار به‌عنوان بخشی از طراحی تعریف می‌شود. در این مرحله، باید خواسته‌های کلی گشت و گذار را کانون توجه قرار دهید. در این راستا پرسش‌های زیر را می‌توانید بپرسید:

- آیا دستیابی به یک سری عناصر معین آسان‌تر از بقیه باشد (یعنی به مراحل کمتری نیاز داشته باشند)؟ اولویت ارائه عناصر به چه صورت است؟
- آیا باید بر عناصر خاصی تأکید ورزید تا کاربران، گشت و گذار خود را به آن جهت متمایل سازند؟
- با خطاهای گشت و گذار چه باید کرد؟
- آیا باید گشت و گذار به گروه مرتبطی از عناصر، بر گشت و گذار به یک عنصر مشخص اولویت داشته باشد؟
- آیا گشت و گذار باید از طریق پیوندها قابل انجام باشد، از طریق دستیابی مبتنی بر جستجو یا از طریقی دیگر؟
- آیا عناصر معین باید بر اساس حیطه‌ی کنش‌های گشت وگذاری قبلی به‌کاربران ارائه شوند؟

---

[1] تعیین سهم بازار برای مرورگرهای وب بسیار دشوار است و بسته به تحقیق و نظر خواهی مورد استفاده، نتایج متفاوتی به دست می‌آید. با این حال، هنگام نوشته شدن این کتاب، دو مرورگر Internet Explorer و Firefox تنها مرورگرهایی بودند که سهم آن‌ها بالغ بر ۳۰٪ گزارش شده بود و Mozilla، Opera و Safari هر یک فقط بالای یک درصد سهم داشتند.

- آیا برای گشت و گذار هر کاربر، باید یک فایل ایجاد کارنامه تهیه شود؟
- آیا یک منو یا نقشه گشت و گذار کامل (در مقابل پیوند ساده «بازگشت» یا اشاره‌گر جهت دار) باید در هر نقطه تعامل کاربر در دسترس باشد؟
- آیا طراحی گشت و گذار باید بر اساس رفتار مورد انتظار برای اکثر کاربران صورت پذیرد یا بر اساس اهمیت عناصر تعیین شده‌ای از برنامه‌ی تحت وب ؟
- آیا کاربری می‌تواند گشت و گذار خود را از طریق برنامه‌ی تحت وب «ضبط» کند تا در آینده بتواند آن را به‌کار ببرد؟
- برای کدام گروه کاربران باید گشت و گذار بهینه را طراحی کرد؟
- با پیوندهای خارجی منتهی به برنامه‌ی تحت وب چگونه باید کار کرد؟ روی پنجره فعلی مرورگر با شود؟ به‌صورت پنجره‌ای جدید؟ یا یک کادر جداگانه؟

این پرسش‌ها و پرسش‌های بسیار دیگر را باید به‌عنوان بخشی از تحلیل گشت و گذار مطرح کرد و به آن‌ها پاسخ گفت.

همچنین شما و سایر طرف‌های ذی‌نفع باید خواسته‌های کلی را برای گشت و گذار تعیین کنید. برای مثال، آیا «نقشه سایتی» فراهم خواهد آمد تا به‌کاربر دیدی اجمالی از کل ساختار برنامه‌ی تحت وب بدهد؟ آیا کاربر می‌تواند از یک «تور راهنمایی شده» استفاده کند که مهمترین عناصر در دسترس (قابلیت‌ها و اشیای محتوایی) را به او معرفی کند؟ آیا کاربر می‌تواند بر اساس صفات تعیین شده برای آن عناصر به قابلیت‌ها یا اشیای محتوایی دست پیدا کند (مثلاً کاربری ممکن است بخواهد به همه‌ی عکس‌های یک ساختمان مشخص یا همه‌ی قابلیت‌هایی که محاسبه‌ی وزن را امکان پذیر می‌سازند، دستیابی داشته باشد)؟

۶-۷ خلاصه

در مدل‌های جریان‌گرا، جریان اشیای داده‌ای به هنگام تبدیل‌شدن توسط عملیات پردازشی کانون توجه قرار می‌گیرند. مدل‌های جریان‌گرا که از تحلیل ساخت یافته به‌دست می‌آیند، از نمودار جریان داده‌ها استفاده می‌کنند، در این نمادگذاری مدل‌سازی چگونگی تبدیل ورودی به خروجی با به حرکت در آمدن اشیای داده‌ای در سیستم به تصویر کشیده می‌شود. هر عملکرد نرم‌افزار که داده‌ها را تبدیل می‌کند، با متن روایی فرایند توصیف می‌شود. این عنصر مدل‌سازی علاوه بر جریان داده‌ها، جریان کنترل را هم به تصویر می‌کشد- نمایشی که چگونگی تأثیرگذاری رویدادها بر یک سیستم را نشان می‌دهد.

مدل‌سازی رفتاری، رفتار پویای سیستم را به تصویر می‌کشد. مدل رفتاری از ورودی به‌دست آمده از عناصر مبتنی بر سناریو، جریان‌گرا و مبتنی بر کلاس به‌عنوان ورودی استفاده می‌کند و حالت کلاس‌های تحلیل و کل سیستم را به نمایش می‌گذارد. برای نیل به این مقصود، حالت‌ها، شناسایی می‌شوند، رویدادهایی که باعث می‌شوند یک کلاس (یا سیستم) از حالتی به حالت دیگر گذار کند، تعیین می‌شوند و کنش‌هایی که با انجام گذار رخ می‌دهند نیز شناسایی می‌شوند. نمادگذاری مورد استفاده برای مدل‌سازی رفتاری، نمودارهای حالت و نمودارهای ترتیب هستند.

مهندس نرم‌افزار به کمک الگوهای تحلیل می‌تواند از اطلاعات دامنه‌ای موجود برای تسهیل ایجاد مدل خواسته‌ها استفاده کند. الگوی تحلیل یک ویژگی نرم‌افزاری خاص یا قابلیتی را توصیف می‌کند که توسط مجموعه یکپارچگی از use case شرح داده می‌شود. در این الگو، هدف ایجاد آن، انگیزه‌ی استفاده از آن، قید و بندهایی که استفاده از آن را محدود می‌سازند، کاربرد آن در دامنه‌های گوناگون مسأله، ساختار کلی الگو، رفتار و همکاری‌های آن و سایر اطلاعات مکمل گنجانده می‌شود.

در مدل‌سازی خواسته‌ها برای برنامه‌های تحت وب می‌توان اکثر (یا شاید همه) عناصر مدل‌سازی بحث شده در این کتاب را به‌کاربرد. به هر حال این عناصر در مجموعه‌ای از مدل‌های تخصص یافته به‌کار برده می‌شوند که به محتوا، تعامل، قابلیت عملیاتی، گشت و گذار و پیکربندی سرور-کلاینت برای برنامه‌ی تحت وب می‌پردازند.

مسائل و نکاتی برای تعمق

۷-۱ اختلاف بنیادی میان راهبردهای تحلیل ساخت یافته و شیءگرا برای تحلیل خواسته‌ها در چیست؟

۷-۲ در یک نمودار جریان داده‌ها، آیا پیکان، نشان‌گر جریان کنترل است یا چیزی دیگر؟

۷-۳ «پیوستگی جریان اطلاعات» چیست و در پالایش نمودار جریان داده‌ها چگونه به‌کار گرفته می‌شود؟

۷-۴ تجزیه گرامری چگونه در ایجاد DFD به‌کار می‌رود؟

۷-۵ تعیین مشخصات کنترل چیست؟

۷-۶ آیا PSPEC و use case یک چیزند؟ اگر نیستند، تفاوت‌های آن‌ها را شرح دهید.

۷-۷ دو نوع «حالت» متفاوت وجود دارند که مدل‌های رفتاری می‌توانند آن‌ها را به نمایش در آورند. آن دو نوع حالت کدام‌اند؟

۷-۸ نمودار ترتیب چه تفاوتی با نمودار حالت دارد؟ چه شباهتی با هم دارند؟

۷-۹ سه الگوی خواسته‌ها برای یک تلفن همراه مدرن پیشنهاد کنید و شرح مختصری از هر کدام بنویسید. آیا این الگوها را می‌توان برای سایر دستگاه‌ها به‌کار برد؟ مثالی بیاورید.

۷-۱۰ یکی از الگوهایی را که در تمرین ۷-۹ توسعه دادید، انتخاب کنید و توصیف کاملی از الگو را مشابه با آن چه که در بخش ۷-۴-۲ ارائه شده ارائه دهید (مشابه از نظر محتوا و سبک).

۷-۱۱ تصور می‌کنید چه مقدار مدل‌سازی تحلیل برای **SafeHomeAssured.com** لازم است؟ آیا هر کدام از انواع مدل‌های توصیف شده در بخش ۷-۵-۳ مورد نیاز خواهد بود؟

۷-۱۲ هدف مدل تعامل برای یک برنامه‌ی تحت وب چیست؟

۷-۱۳ می‌توان استدلال کرد که مدل عملیاتی یک برنامه‌ی تحت وب را می‌توان تا طراحی به تعویق انداخت. درباره مزایا و معایب این استدلال بحث کنید.

۷-۱۴ هدف مدل پیکربندی چیست؟

۷-۱۵ مدل گشت و گذار چه تفاوتی با مدل تعامل دارد؟

فصل ۸

مفاهیم طراحی

نگاهی گذرا

طراحی چیست؟ طراحی، آن چیزی است که تقریباً هر مهندسی می‌خواهد انجام دهد. جایی است که در آن خلاقیت حاکم است- جایی که خواسته‌های ذی‌نفع‌ها، نیازهای تجاری، و ملاحظات فنی، همگی در کنار یکدیگر به تدوین محصول یا سیستم کمک می‌کنند. طراحی، نمایش یا مدلی از نرم‌افزار ایجاد می‌کند، ولی بر خلاف مدل خواسته‌ها (که توصیف داده‌ها، قابلیت‌ها و رفتار مورد نیاز در کانون توجه آن قرار دارد)، مدل طراحی جزئیات مربوط به معماری نرم‌افزار، ساختمان داده‌ها، واسط‌ها و مؤلفه‌های لازم برای پیاده‌سازی سیستم را فراهم می‌کند.

چه کسی آن را انجام می‌دهد؟ تمامی وظایف طراحی بر عهده‌ی مهندس نرم‌افزار است.

چرا اهمیت دارد؟ طراحی به شما این امکان را می‌دهد تا سیستم یا محصولی را که قرار است ساخته شود، مدل‌سازی کنید. این مدل را می‌توان پیش از تولید کد، اجرای آزمون‌ها و درگیرشدن تعداد کثیری از کاربران از لحاظ کیفیت مورد ارزیابی قرار داد و بهبود بخشید. طراحی جایی است که در آن کیفیت نرم‌افزار تثبیت می‌شود.

مراحل کار کدام است؟ طراحی، نرم‌افزار را به چند شیوه‌ی متفاوت به تصویر می‌کشد. نخست، معماری سیستم یا محصول باید نمایش داده شود. سپس، واسط‌هایی که نرم‌افزار را به‌کاربران نهایی، به سایر سیستم‌ها و دستگاه‌ها و همچنین به مؤلفه‌های سازنده خودش مرتبط می‌سازند، مدل‌سازی می‌شوند. سرانجام، مؤلفه‌های نرم‌افزار که در ساخت سیستم به‌کار می‌روند، مدل‌سازی می‌شوند. هر کدام از این نماها یک کنش طراحی متفاوت را نشان می‌دهند، ولی همه باید از مجموعه‌ای مفاهیم طراحی اصلی پیروی کنند که راهنمای کار طراحی نرم‌افزار هستند.

محصول کار چیست؟ یک مدل طراحی که شامل نمایش‌هایی از معماری و واسط، نمایش در سطح مؤلفه‌ها و نمایش‌های استقرار می‌شود، محصول کاریِ اصلی است که طی طراحی نرم‌افزار ساخته می‌شود.

چگونه اطمینان حاصل کنم که درست از عهده کار بر آمده‌ام؟ مدل طراحی توسط تیم نرم‌افزار ارزیابی می‌شود تا معلوم شود که آیا حاوی خطا، ناسازگاری یا جاافتادگی هست؛ آیا جایگزین‌های بهتری برای آن وجود دارد؛ و آیا این مدل را می‌توان در قید و بندها، زمان‌بندی و یا هزینه‌ی تعیین شده پیاده‌سازی کرد یا خیر.

طراحی نرم‌افزار، شامل مجموعه‌ای از اصول، مفاهیم و کارها می‌شود که به تولید محصول یا سیستمی با کیفیت بالا منجر می‌گردد. اصول طراحی، فلسفه‌ای را پایه‌ریزی می‌کنند که شما را در کار طراحی راهنمایی خواهد کرد. پیش از پیاده‌سازی کار طراحی، مفاهیم طراحی را باید به‌خوبی بشناسید و کار طراحی، خود به ایجاد نمایش‌های متنوعی از نرم‌افزار می‌انجامد که به‌عنوان راهنمایی برای فعالیت بعدی، یعنی ساخت، عمل می‌کنند.

طراحی در موفقیت مهندسی نرم‌افزار اهمیت محوری دارد. در اوایل دهه‌ی ۱۹۹۰، میچ کاپور (پدیدآورنده‌ی Lotus 1-2-3) در مجله‌ی دکتر دابز[1]، «بیانیه‌ی طراحی نرم‌افزار» را این‌گونه منتشر ساخت:

> طراحی چیست؟ طراحی برزخ میان دو دنیاست (دنیای فن‌آوری و دنیای آدمیان و اهداف انسانی) و شما باید این دو دنیا را به هم نزدیک سازید.
>
> معمار و منتقد روم باستان، ویترو ویوس، پیشگام این مفهوم است که ساختمان‌هایی از طراحی خوب برخوردارند که استحکام، تناسب و لذت را به نمایش بگذارند. برای نرم‌افزار خوب هم می‌توان چنین چیزی گفت. *استحکام*: برنامه نباید اشکال و ایرادی داشته باشد که از عملکرد آن ممانعت کند. *تناسب*: برنامه باید برای اهدافی که برای آن نوشته شده است، مناسب باشد. *لذت*: تجربه‌ی به‌کارگیری برنامه باید تجربه‌ای دلپذیر باشد. اینجاست که نظریه‌ی طراحی برای نرم‌افزار آغاز می‌شود.

هدف طراحی، ایجاد مدل یا نمایشی است که استحکام، تناسب و *لذت* از خود نشان دهد. برای رسیدن به این مقصود، باید تنوع و سپس همگرایی را عملی سازید. بلادی [Bel81] می‌گوید: «تنوع عبارت است از در اختیار داشتن فهرستی از منابع متفاوت، مواد خام طراحی: مؤلفه‌ها، راهکارهای مؤلفه‌ای و آگاهی، که همگی در کاتالوگ‌ها، کتاب‌های درسی و در ذهن افراد موجودند.» هنگامی که این مجموعه اطلاعات گوناگون در کنار هم قرار داده شود، باید عناصری از این فهرست را انتخاب کنید که نیازهای مشخص‌شده در مهندسی خواسته‌ها و مدل تحلیل را برآورده سازند (فصل‌های ۵ تا ۷). با رخ‌دادن این اتفاقات، راه‌های متفاوتی آزموده می‌شوند تا این که به «پیکربندی خاصی از مؤلفه‌ها همگرا شوید و محصول نهایی را ایجاد کنید» [Bel 81].

تنوع و همگرایی، بصیرت و قضاوت را بر اساس تجربه به‌دست‌آمده در ساخت موجودیت‌های مشابه، مجموعه‌ای از اصول و/یا ابتکارات راهنمای شیوه تکامل مدل، مجموعه‌ای از ملاک‌هایی که قضاوت درباره طراحی پایانی را امکان پذیر می‌سازند و یک فرایند تکرار که سرانجام به نمایش طراحی نهایی می‌انجامد، با هم ترکیب می‌کند.

طراحی نرم‌افزار با تکامل روش‌های جدید، تحلیل بهتر و افزایش شناخت پیوسته تغییر می‌کند.[2] حتی امروز، اکثر روش‌شناسی‌های طراحی نرم‌افزار فاقد عمق، انعطاف‌پذیری و ماهیتی کیفی هستند که معمولاً در رشته‌های سنتی‌تر طراحی مهندسی مشاهده می‌شود. به هر حال، روش‌هایی برای طراحی نرم‌افزار وجود دارند، ملاک‌هایی برای کیفیت طراحی در دسترس هستند و نمادگذاری طراحی را می‌توان به‌کار گرفت. در این فصل، اصول و مفاهیم بنیادی قابل استفاده در کل طراحی نرم‌افزار، عناصر

[1] Dr. Dobbs Journla

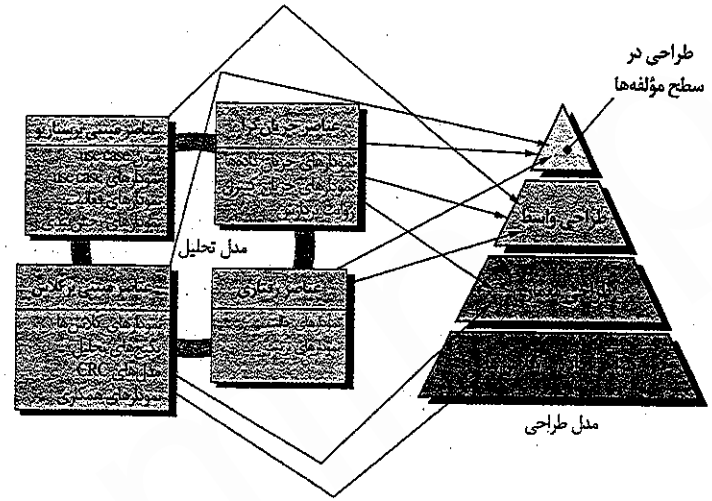
[2] خوانندگانی که به فلسفه طراحی نرم افزار علاقه بیشتری دارند ممکن است بحث فیلیپ کروچن درباره طراحی پست مدرن را مفید بیابند.

مدل طراحی و تأثیر الگوها بر فرایند طراحی را بررسی خواهیم کرد. در فصل‌های ۹ تا ۱۳، انواع روش‌های طراحی و کاربرد آن‌ها در طراحی معماری، طراحی واسط‌ها و طراحی در سطح مؤلفه‌ها و همچنین روش‌های طراحی مبتنی بر الگو و مبتنی بر وب به‌کار برده خواهند شد.

۸-۱ طراحی در حیطه‌ی مهندسی نرم‌افزار

طراحی نرم‌افزار هسته اصلی نرم‌افزار را تشکیل می‌دهد و به‌کارگیری آن مستقل از نوع مدل فرایند نرم‌افزار مورد استفاده است. طراحی نرم‌افزار که پس از تحلیل و مدل‌سازی خواسته‌ها آغاز می‌شود، آخرین کنش مهندسی نرم‌افزار در فعالیت مدل‌سازی است که صحنه را برای **ساخت** (تولید و آزمایش کد) آماده می‌کند.

هر کدام از عناصر مدل تحلیل (فصل‌های ۶ و ۷) اطلاعاتی را فراهم می‌آورد که برای ایجاد چهار مدل طراحی مورد نیاز جهت تعیین مشخصات کامل طراحی ضروری هستند. جریان اطلاعات طی طراحی نرم‌افزار در شکل ۸-۱ نشان داده شده است. مدل خواسته‌ها، که توسط عناصر مبتنی بر سناریو، مبتنی بر کلاس، جریان‌گرا و رفتاری نمود پیدا می‌کند، وظیفه‌ی طراحی را تغذیه می‌کند. در طراحی، با استفاده از نمادگذاری طراحی و روش‌های طراحی که در فصل‌های آینده بحث خواهد شد، طراحی داده‌های کلاس‌ها، طراحی معماری، طراحی واسط و طراحی مؤلفه‌ها ایجاد می‌شود.



شکل ۸-۱ برگردان مدل خواسته‌ها به مدل طراحی.

طراحی داده‌ها/کلاس‌ها، مدل‌های کلاس‌ها (فصل ۶) به کلاس‌های طراحی و ساختمان داده‌های لازم برای پیاده‌سازی نرم‌افزار تبدیل می‌شوند. اشیا و روابط تعریف شده در نمودار CRC و جزئیات محتویات داده‌های تصویر شده توسط صفات کلاس و سایر نمادگذاری‌ها، مبنایی برای کنش طراحی داده‌ها فراهم می‌آورد. بخشی از طراحی کلاس‌ها ممکن است مرتبط با طراحی معماری نرم‌افزار رخ دهد. طراحی مفصل‌تر کلاس‌ها با طراحی هر مؤلفه از نرم‌افزار صورت می‌پذیرد.

> «متداول‌ترین معجزه‌ی مهندسی نرم‌افزار، گذار از تحلیل به طراحی و گذار از طراحی به کد است.»
>
> **ریچارد دو**

اندرز

طراحی نرم‌افزار همواره باید با در نظر گرفتن داده‌ها- بنیادی برای همه‌ی عناصر دیگر طراحی- آغاز شود. پس از این که بنیاد مذکور گذاشته شد، معماری باید به‌دست آورده شود. تنها در آن صورت است که باید سایر وظایف طراحی را اجرا کرد.

SafeHome

طراحی در مقایسه با کد نویسی

صحنه: اتاقک جیمی، هنگامی که تیم آماده ترجمه خواسته‌ها به طراحی می‌شود.

نقش‌آفرینان: جیمی، وینود و اد- همه‌ی اعضای تیم نرم‌افزار *SafeHome*

گفتگوها:

جیمی: داگ [مدیر تیم] از طراحی خوشش نمی آید. رو راست بگویم، چیزی که دوست دارم انجام بدهم، کد نویسی است. C++ یا جاوا که باشد، خوشحالم می‌کند.

اد: نه... از طراحی خوشت می‌آید.

جیمی: گوش نمی کنی؛ کد نویسی کار اصلی است.

وینود: فکر کنم منظور اد این است که تو واقعاً کد نویسی را دوست نداری بلکه طراحی و بیان آن به‌صورت کد را دوست داری. کد زبانی است برای نمایش طراحی.

جیمی: و مشکل آن چی هست؟

وینود: سطح انتزاع.

جیمی: چی؟

اد: زبان برنامه‌نویسی برای نشان دادن جزئیاتی مثل ساختمان داده‌ها و الگوریتم‌ها خوب است، ولی برای به نمایش در آوردن معماری یا همکاری میان مؤلفه‌ها... یا چیزهایی از این دست خوب نیست.

وینود: و یک معماری بد می‌تواند حتی بهترین کدها را خراب کند.

جیمی (لحظه‌ای به فکر فرو می‌رود): یعنی می‌گویی که من نمی توانم معماری را در قالب کدها نشان دهم... نه خیر درست نیست.

وینود: قطعاً می‌توانی معماری را در قالب کد ارائه بدهی، ولی در اکثر زبان‌های برنامه‌نویسی، به‌دست آوردن تصویری واضح از معماری یا بررسی کدها خیلی سخت است.

اد: و ما قبل از شروع کدنویسی چه می‌خواهیم؟

جیمی: بسیار خب، شاید طراحی و کدنویسی دو تا کار متفاوت باشند، ولی من هنوز کد نویسی را بیشتر دوست دارم.

در طراحی معماری، رابطه‌ی میان عناصر ساختاری اصلی نرم‌افزار، سبک‌های معماری و الگوهای معماری تعریف می‌شود که از آن‌ها می‌توان در دستیابی به خواسته‌های تعریف شده برای سیستم و قید و بندهای مؤثر بر شیوه‌ی پیاده‌سازی معماری استفاده کرد [Sha96]. نمایش طراحی معماری- که چارچوب سیستم کامپیوتری است- از مدل خواسته‌ها به‌دست می‌آید.

در طراحی واسطها چگونگی برقراری ارتباط نرم‌افزار با سیستم‌هایی که با آن همکاری متقابل دارند و با افرادی که از آن‌ها استفاده می‌کنند، توصیف می‌شود. واسط، جریان اطلاعات را (مثلاً کنترل و/یا داده‌ها) و نوع مشخصی از رفتار را نشان می‌دهد. بنابراین، مدل‌های رفتاری و سناریوهای کاربری، اکثر اطلاعات لازم برای طراحی واسط را فراهم می‌سازند.

طراحی در سطح مولفه‌ها، عناصر ساختاری معماری نرم‌افزار را به توصیف روالی از مؤلفه‌های نرم‌افزار تبدیل می‌کند. اطلاعات به‌دست آمده از مدل‌های مبتنی بر کلاس‌ها، مدل‌های جریان و مدل‌های رفتاری به‌عنوان مبنایی برای طراحی مؤلفه‌ها عمل می‌کنند.

در طول طراحی، تصمیم‌هایی می‌گیرند که در نهایت بر موفقیت ساخت نرم‌افزار و به همان اندازه از اهمیت، بر سهولت نگهداری نرم‌افزار تأثیر می‌گذارند. ولی چرا طراحی این همه اهمیت دارد؟

اهمیت طراحی نرم‌افزار را در یک کلمه می‌توان خلاصه کرد- *کیفیت*. طراحی جایی است که کیفیت از آن وارد مهندسی نرم‌افزار می‌شود. طراحی نمایش‌هایی از نرم‌افزار را در اختیارتان قرار خواهد داد که برای کیفیت می‌توانید آن‌ها را مورد ارزیابی قرار دهید. طراحی تنها راهی است که می‌توانید از طریق آن‌ها خواسته‌های طرف‌های ذی‌نفع را به درستی به محصول یا سیستم نهایی ترجمه کنید. طراحی نرم‌افزار به‌عنوان بستری برای همه‌ی فعالیت‌های مهندسی و پشتیبانی نرم‌افزار که به دنبال خواهند آمد، عمل می‌کند. بدون طراحی، ساخت سیستمی ناپایدار را همراه با ریسک می‌پذیرید- سیستمی که با اعمال تغییرات کوچک، به شکست می‌انجامد؛ سیستمی که آزمایش آن ممکن است دشوار باشد؛ سیستمی که کیفیت آن تا اواخر فرایند قابل ارزیابی نیست، یعنی زمانی که وقت تنگ است و مبالغ هنگفتی هزینه شده است.

> دو راه برای بنا نهادن طراحی نرم‌افزار وجود دارد. یکی آن که طراحی را چنان ساده کنیم که به‌وضوح هیچ کمبودی نشان ندهند و راه دیگر آن است که طراحی را چنان پیچیده کنیم که هیچ کمبود آشکاری وجود نداشته باشد. روش اول به مراتب دشوارتر است.
>
> **سی. ای. آر. هوار**

۸-۲ فرایند طراحی

طراحی نرم‌افزار، فرایندی مبتنی بر تکرار است که از طریق آن خواسته‌ها به «نقشه‌ای» برای ساخت نرم‌افزار ترجمه می‌شوند. در آغاز، این نقشه نمایی کلی از نرم‌افزار را به تصویر می‌کشد. یعنی، طراحی در سطح بالایی از انتزاع نمایش داده می‌شود- سطحی که به‌طور مستقیم تا هدف خاصی از سیستم و جزئیات بیشتری از خواسته‌های داده ای، عملیاتی و رفتاری قابل پیگیری است. با تکرارشدن طراحی، پالایش‌های بعدی به نمایش‌های طراحی در سطوح پایین تری از انتزاع منجر می‌شود. این‌ها را نیز می‌توان تا خواسته‌ها پیگیری کرد، ولی ارتباطات ظریف‌ترند.

۸-۲-۱ دستور العمل‌ها و صفات کیفیت نرم‌افزار

در سرتاسر فرایند طراحی، کیفیت طراحی در حال تکامل، با یک سری بازبینی‌های فنی قابل ارزیابی است که در فصل ۱۵ بحث خواهد شد. مک گلافلین [McG91] سه خصوصیت پیشنهاد می‌کند که به‌عنوان راهنمایی برای تکامل طراحی خوب به‌شمار می‌روند.

- طراحی باید همه‌ی خواسته‌های صریح موجود در مدل خواسته‌ها را پیاده‌سازی کند و باید همه‌ی خواسته‌های ضمنی مطلوب طرف‌های ذی‌نفع را پاسخ گو باشد.
- طراحی باید یک راهنمای خوانا و قابل فهم (۱) برای کسانی باشد که کدها را تولید می‌کنند و (۲) برای کسانی که نرم‌افزار را آزمایش و بعداً پشتیبانی می‌کنند.
- طراحی باید تصویر کاملی از نرم‌افزار باشد که دامنه‌های داده‌ای، عملیاتی و رفتاری را از دیدگاه پیاده‌سازی به نمایش بگذارد.

در واقع، هر کدام از این خصوصیت‌ها فرایند طراحی است. ولی هر کدام از این اهداف چگونه قابل حصول است؟

> «... نوشتن قطعه کدی هوشمندانه یک چیز است؛ طراحی چیزی که بتوان تجارتی را در درازمدت پشتیبانی کند چیزی کاملاً متفاوت است.»
>
> **سی. فرگوسن**

اطلاعات

دستیابی به کیفیت طراحی- مرور فنی

طراحی مهم است چون به تیم نرم‌افزار امکان ارزیابی کیفیت نرم‌افزار را پیش از پیاده‌سازی آن می‌دهد- یعنی در زمانی که جاافتادگی‌ها، خطاها یا ناسازگاری‌ها را با هزینه بسیار کم می‌توان تصحیح کرد. ولی کیفیت را چگونه می‌توان در طول طراحی ارزیابی کرد؟ نرم‌افزار را نمی‌توان آزمایش کرد چون هنوز اصلاً نرم‌افزاری وجود ندارد که آزمایش شود. چه باید کرد؟ در طول طراحی، کیفیت با اجرای یک سری بازبینی‌های فنی ارزیابی می‌شود. مرور فنی را در فصل ۱۵ به تفصیل مورد بحث قرار خواهیم داد، ولی در این مرحله، ارائه خلاصه‌ای از این تکنیک مناسب به نظر می‌رسد. مرور فنی، جلسه‌ای است که توسط اعضای تیم نرم‌افزار برگزار می‌شود. معمولاً بسته به دامنه‌ی اطلاعات طراحی که باید مرور شود، دو، سه یا چهار نفر در این جلسه شرکت می‌کنند. هر شخص نقشی دارد: رهبر مرور، برنامه‌ریزی جلسه را بر عهده دارد، دستور کاری تنظیم می‌کند و جلسه را اداره می‌کند؛ منشی جلسه یادداشت بر می‌دارد تا چیزی از قلم نیفتد؛ تولیدکننده کسی است که محصول کاری او (مثلاً طراحی مؤلفه‌ای از نرم‌افزار) قرار است مرور شود. پیش از جلسه، به هر یک از افراد حاضر در تیم مرور یک نسخه از محصول کاریِ طراحی داده می‌شود و از او خواسته می‌شود آن را بخواند و به دنبال خطاها، جاافتادگی‌ها یا ابهام‌ها بگردد. با شروع جلسه، هدف، ذکر همه‌ی مشکلات محصولات کاری است به‌طوری که بتوان آن‌ها را قبل از شروع پیاده‌سازی تصحیح کرد. مرور فنی معمولاً نود دقیقه تا دو ساعت زمان می‌برد. در پایان جلسه، تیم مرور تعیین می‌کند که قبل از این که محصول کاریِ طراحی را بتوان به‌عنوان بخشی از مدل طراحی نهایی به تصویب رساند، آیا به کنش‌های بیشتری نیاز هست یا خیر.

دستور العمل‌های کیفیتی. شما و سایر اعضای تیم نرم‌افزار به منظور تعیین کیفیت نمایش طراحی باید ملاک‌هایی فنی برای طراحی خوب وضع کنید. در بخش ۳-۸ درباره مفاهیم طراحی‌ای بحث خواهیم کرد که به‌عنوان ملاک‌های کیفیت نرم‌افزار نیز عمل می‌کنند. در حال حاضر، دستور العمل‌های زیر را در نظر می‌گیریم:

۱. طراحی باید معماری‌ای را نشان دهد که (۱) با استفاده از سبک‌ها یا الگوهای معماری شناخته شده ایجاد شده باشند، (۲) از مؤلفه‌هایی تشکیل شده باشد که خصوصیات طراحی خوبی از خود به نمایش بگذارند (این خصوصیات را بعداً در همین فصل مورد بحث قرار خواهیم داد.)، و (۳) به شیوه‌ای تکاملی قابل پیاده‌سازی باشند[1] تا به این ترتیب، پیاده‌سازی و آزمایش تسهیل گردد.

۲. طراحی باید پیمانه‌بندی شده باشد؛ یعنی نرم‌افزار به طرزی منطقی به عناصر یا زیر سیستم هایش افراز شده باشد.

۳. طراحی باید حاوی نمایش‌های متمایزی از داده‌ها، معماری، واسط‌ها و مؤلفه‌ها باشد.

خصوصیات طراحی خوب کدام‌اند؟

[1] برای سیستم‌های کوچکتر، طراحی را گاهی می‌توان به‌صورت خطی نیز توسعه داد.

۴. طراحی باید به ساختمان داده‌ای منجر گردد که برای کلاس‌هایی که قرار است پیاده‌سازی شوند و از الگوهای داده‌ای قابل تشخیص بیرون کشیده می‌شوند، مناسب باشند.

۵. نتیجه‌ی طراحی باید مؤلفه‌هایی باشد که از خود خصوصیات عملیاتی مستقل به نمایش بگذارند.

۶. طراحی باید به واسط‌هایی بینجامد که پیچیدگی ارتباطات میان مؤلفه‌ها و ارتباط آن‌ها با محیط خارجی را کاهش دهند.

۷. طراحی باید با استفاده از روشی تکرار پذیر به‌دست آید که خود با اطلاعات حاصل از تحلیل خواسته‌های نرم‌افزار به‌دست می‌آید.

۸. طراحی باید با به‌کارگیری نمادهایی ارائه شود که معنا و مفهوم را به خوبی برساند.

رسیدن به این اهداف با بخت و اقبال میسر نخواهد بود. برای رسیدن به آن‌ها باید یک سری اصول بنیادی طراحی، روش شناسی سیستماتیک و مرور کامل را به‌کار برد.

«کیفیت، چیزی نیست که مثل گذاشتن یک پولک و زرق و برق روی درخت کریسمس به‌دست آید.»

رابرت پیرسیگ

اندرز

طراحان نرم‌افزار تمایل دارند مسائلی را کانون توجه قرار دهند که قرار است حل شود. فقط فراموش نکنید که صفات FURPS همواره بخشی از مسأله‌اند. آن‌ها را باید در نظر گرفت.

صفات کیفیتی. هیولت- پاکارد [Gra 87] یک مجموعه صفات کیفیتی برای نرم‌افزار توسعه داده است که به اختصار به‌عنوان FURPS (قابلیت عملیاتی، قابلیت کاربرد، قابلیت اطمینان، کارایی و قابلیت پشتیبانی) شناخته می‌شوند. صفات کیفیتی FURPS نشان‌گر هدفی برای همه‌ی طراحی نرم‌افزارند.

- *قابلیت عملیاتی* (Functionality) با تعیین مجموعه ویژگی‌ها و قابلیت‌های برنامه، عملیات کلی که تحویل می‌شوند و امنیت کل سیستم ارزیابی می‌شود.
- *قابلیت کاربرد* (Usability) با اندازه‌گیری عوامل انسانی (فصل ۱۱)، زیبایی شناسی کلی، سازگاری و مستندسازی سنجیده می‌شود.
- *قابلیت اطمینان* (Reliability) با اندازه‌گیری فراوانی و شدت شکست‌ها، صحت نتایج خروجی، میانگین زمان شکست (MTTF)، توانایی خلاصی یافتن از شکست و قابلیت پیش بینی برنامه تعیین می‌شود. *کارایی* با در نظر گرفتن سرعت پردازش، زمان پاسخ دهی، مصرف منابع، توان عملیاتی و بازدهی سنجیده می‌شود.
- *قابلیت پشتیبانی* (Supportability) ترکیبی است از توان بسط برنامه (بسط‌پذیری)، قابلیت انطباق، قابلیت سرویس- این سه صفت، در کل نشان‌گر صفتی متداول‌تر، موسوم به قابلیت نگهداری هستند- و علاوه بر آن، قابلیت آزمایش، سازگاری، قابلیت پیکربندی (توانایی سازمان دهی و کنترل عناصر پیکربندی نرم‌افزار، فصل ۲۲)، سهولت نصب سیستم و سهولت پیداکردن مسائل.

در توسعه‌ی طراحی نرم‌افزار، همه‌ی صفات کیفیتی نرم‌افزار به یک میزان اهمیت ندارند. در یک برنامه‌ی کاربرد ممکن است قابلیت عملیاتی با تأکیدی خاص بر امنیت مورد توجه باشد ولی در برنامه‌ی کاربردی دیگر کارایی همراه با تأکیدی خاص بر سرعت پردازش مورد توجه باشد. در سومی ممکن است قابلیت اطمینان مد نظر باشد. این میزان اهمیت هر چه که باشد، شایان ذکر است که این صفات کیفیتی را باید همان زمان که طراحی شروع می‌شود، مورد توجه قرار داد نه پس از کامل‌شدن طراحی و شروع ساخت.

۲-۲-۸ تکامل طراحی نرم‌افزار

تکامل طراحی نرم‌افزار، فرایندی پیوسته است که اکنون نزدیک به شش دهه را پشت سر گذاشته است.

کار طراحی اولیه بر ملاک‌هایی برای توسعه‌ی برنامه‌های پیمانه‌بندی شده [Den73] و روش‌هایی برای پالایش ساختارهای نرم‌افزار به‌شیوه‌ای از بالا به پایین (top-down) [Wir71] متمرکز است. جنبه‌های روالی (procedural aspects) تعریف، به فلسفه‌ای موسوم به برنامه‌نویسی ساخت‌یافته ([Mil72]، [Dah72]) تکامل پیدا کرد. در کارهای بعدی روش‌هایی برای ترجمه‌ی جریان داده‌ها [Ste74] یا ساختمان داده‌ها ([Jac75]، [War74]) به تعریف طراحی پیشنهاد شد. در رویکردهای جدیدتر طراحی ([Jac92]،[Gsm95]) روشی شیءگرا برای رسیدن به طراحی پیشنهاد شد. در رویکردهای جدیدتر، در طراحی نرم‌افزار، معماری نرم‌افزار [Kru06] و الگوهای طراحی قابل استفاده در پیاده‌سازی معماری نرم‌افزار و سطوح پایین تری از انتزاع در طراحی مورد تأکید بوده‌اند ([Hol06]، [Sha05]). تأکید فزاینده بر روش‌های جنبه‌گرا ([Cla05]، [Jac04])، باعث شده است تا توسعه‌ی مبتنی بر مدل [Sch 06] و توسعه مبتنی بر آزمون [Ast04]، بر تکنیک‌های دستیابی به پیمانه‌بندی و ساختار معماری اثربخش‌تر در طراحی‌های ایجادشده، بیش از پیش مورد توجه قرار گیرد.

چند روش طراحی که از میان کارهای ذکر شده در بالا متبلور شده‌اند، در سرتاسر صنعت نرم‌افزار مورد استفاده قرار گرفته‌اند. همانند روش‌های تحلیلی که در فصل‌های ۶ و ۷ ارائه شدند، هر روش طراحی حاوی نمادگذاری و ابتکاری منحصر به فرد، با دیدی نسبتاً محدود از آن چیزی است که کیفیت طراحی را مشخص می‌کند. با این وجود، همه‌ی این روش‌ها یک سری خصوصیات مشترک دارند: (۱) سازوکاری برای ترجمه مدل خواسته‌ها به نمایش طراحی، (۲) یک نمادگذاری برای نمایش مؤلفه‌های عملیاتی و واسط‌های آن‌ها، (۳) ابتکاراتی برای پالایش و افراز و (۴) دستور العمل‌هایی برای ارزیابی کیفیت.

هر روش طراحی که به‌کار برده شود، باید مجموعه‌ای از مفاهیم طراحی داده ای، طراحی معماری، طراحی واسط‌ها و طراحی در سطح مؤلفه‌ها را به‌کار ببرید، که در بخش بعدی به این مفاهیم خواهیم پرداخت.

«طراح خوب می‌داند که وقتی به کمال رسیده است که چیزی برای دور انداختن وجود نداشته باشد نه اینکه چیزی برای اضافه کردن وجود نداشته باشد.»

آنتوان دوسنت اگزوپری

چه خصوصیاتی در همه‌ی روش‌های طراحی، مشترک هستند؟

۸-۳ مفاهیم طراحی

در تاریخ مهندسی نرم‌افزار، مجموعه‌ای از مفاهیم بنیادی نرم‌افزار تکامل یافته است. گرچه میزان توجه به هر مفهوم طی این سال‌های تغییر کرده است، هر کدام از آن‌ها از امتحان زمان سر بلند بیرون آمده‌اند. هر کدام از این مفاهیم برای طراح نرم‌افزار، بستری فراهم می‌سازد که روش‌های پیچیده‌تر طراحی را بر اساس آن می‌تواند به‌کار ببرد. هر کدام از این مفاهیم شما را در پاسخ گفتن به پرسش‌های زیر یاری می‌دهد:

- برای افراز نرم‌افزار به مؤلفه‌های جداگانه از چه ملاک‌هایی می‌توان استفاده کرد؟
- جزئیات قابلیت عملیاتی یا ساختمان داده‌ها چگونه از نمایش مفهومی نرم‌افزار جدا می‌شود؟
- کدام ملاک‌های یکنواخت، کیفیت فنی طراحی نرم‌افزار را تعیین می‌کنند؟

ام. ای. جکسون [Jac75] زمانی گفته است: «شروع خرد برای مهندس نرم‌افزار زمانی است که اختلاف میان به‌کار انداختن یک برنامه و درست انجام دادن آن را تشخیص دهد.» مفاهیم بنیادی طراحی نرم‌افزار، چارچوب لازم برای «درست انجام دادن» را فراهم می‌آورند.

در بخش‌هایی که به دنبال خواهد آمد، مروری مختصر بر مفاهیم مهم طراحی نرم‌افزار خواهیم داشت که هر دو شیوه سنتی و شیءگرا برای توسعه‌ی نرم‌افزار را شامل می‌شوند.

مجموعه وظایف

مجموعه وظایف کلی مربوط به طراحی

۱. مدل دامنه‌ی اطلاعاتی و ساختمان داده‌ای مناسب را برای اشیای داده‌ای و صفات آن‌ها بررسی کنید.
۲. با استفاده از مدل تحلیل، یک سبک معماری انتخاب کنید که مناسب نرم‌افزار باشد.
۳. مدل تحلیل را به زیر سیستم‌های طراحی، افراز و وظیفه‌ی هر زیر سیستم را در معماری مشخص کنید.
هر زیر سیستم از نظر عملیاتی باید یکپارچگی داشته باشد.
رابط‌های میان زیرسیستم‌ها را طراحی کنید.
به هر زیر سیستم، کلاس‌های تحلیل یا قابلیت‌های عملیاتی تخصیص دهید.
۴. مجموعه‌ای از کلاس‌های طراحی یا مؤلفه‌ها ایجاد کنید:
توصیف کلاس تحلیل را به کلاس طراحی ترجمه کنید.
هر کلاس طراحی را در مقابل ملاک‌های طراحی چک کنید؛ مسائل وراثتی را مد نظر داشته باشید.
متدها و پیام‌های مرتبط با هر کلاس طراحی را تعریف کنید.
الگوهای طراحی را برای یک کلاس یا زیرکلاس طراحی، ارزیابی و انتخاب کنید.
کلاس‌های طراحی را مرور و در صورت نیاز بازنویسی کنید.
۵. هرگونه واسط لازم برای سیستم‌ها یا دستگاه‌های خارجی را طراحی کنید.
۶. واسط کاربری را طراحی کنید:
نتایج تحلیل وظایف را مرور کنید.
سلسله کنش‌ها را بر اساس سناریوهای کاربری مشخص سازید.
مدل رفتاری واسط را ایجاد کنید.
اشیای واسط و سازوکارهای کنترلی را تعریف کنید.
طراحی واسط‌ها را مرور و در صورت نیاز، بازنویسی کنید.
۷. طراحی را در سطح مؤلفه‌ها را انجام دهید.
همه‌ی الگوریتم‌ها را در سطح نسبتاً پایین از انتزاع مشخص کنید.
واسط هر مؤلفه پالایش کنید.
هر مؤلفه را مرور و همه‌ی خطاهای آشکارشده را تصحیح کنید.
۸. مدلی برای استقرار تهیه کنید.

۸-۳-۱ انتزاع (Abstraction)

هنگامی که راهکاری پیمانه‌ای را برای مسأله در نظر می‌گیرید، سطوح انتزاع متعددی ممکن است پیش آید. در بالاترین سطح انتزاع، راهکار در قالب عبارت‌هایی کلی و با استفاده از زبان محیط مسأله، بیان می‌شود. در سطوح پایین انتزاع، توصیف مشروح‌تری از راهکار ارائه می‌شود. برای بیان راهکار، اصطلاح‌شناسی مسأله با اصطلاح‌شناسی پیاده‌سازی، تلفیق می‌شود. سرانجام، در پایین‌ترین سطح از انتزاع، راهکار به شیوه‌ای بیان می‌شود که به‌طور مستقیم قابل اجرا و پیاده‌سازی باشد.

«انتزاع یکی از شیوه‌های بنیادی است که ما انسان‌ها از طریق آن با پیچیدگی کار می‌آییم.»

گرادی بوچ

با توسعه‌یافتن سطوح متفاوت انتزاع، روی ایجاد هر دو نوع انتزاع فرایندی و داده‌ای کار می‌کنید. منظور از انتزاع فرایندی، دستورالعمل‌هایی است که وظیفه‌ای مشخص و محدود دارند. از نام انتزاع فرایندی، این وظایف پیداست، ولی جزئیات خاصی کنار گذاشته می‌شود. مثالی از انتزاع فرایندی، واژه *باز کردن* برای درهاست. *باز کردن* به معنای سلسله‌ای طولانی از مراحل روالی است (مثلاً رفتن به طرف در، دراز کردن دست و گرفتن دستگیره، چرخاندن دستگیره، کشیدن در و دورشدن از در)[1].

انتزاع داده‌ای مجموعه‌ای از داده‌ها با نام مشخص است که شیء داده‌ای را توصیف می‌کند. در حیطه‌ی انتزاع فرایندی برای *باز کردن در*، می‌توانیم یک انتزاع داده‌ای با نام **door** تعریف کنیم. همانند هر شیء داده‌ای دیگر، انتزاع داده‌ای برای **door** شامل مجموعه‌ای از صفات خواهد شد که در را توصیف می‌کنند (مثل نوع در، جهت بسته‌شدن آن، سازوکار باز شدن، وزن، ابعاد). لازم می‌آید که «*انتزاع فرایندی باز کردن در*» از اطلاعات موجود در صفات انتزاع داده‌ای **door** استفاده کند.

اندرز

به‌عنوان طراح، سخت کار کنید تا هر دو نوع انتزاع فرایندی و داده‌ای را که به مسأله‌ی مورد نظر کمک می‌کند، به‌دست آورید. اگر آن‌ها بتوانند به‌عنوان دامنه‌ی کامل مسائل عمل کنند، حتی بهتر خواهد بود.

مرجع وب

بحثی عمیق درباره معماری نرم‌افزار را می‌توان در www.sei.cmu.edu/ata/ata_init.html یافت.

۸-۳-۲ معماری (Architecture)

معماری نرم‌افزار به «ساختار کلی نرم‌افزار و شیوه‌هایی مربوط می‌شود که این ساختار باعث یکپارچگی مفهومی در سیستم می‌گردد.» [Sha 95a]. معماری در ساده‌ترین شکل خود، ساختار یا سازمان‌دهی مؤلفه‌های برنامه، شیوه‌ی تعامل این مؤلفه‌ها و ساختمان داده‌های قابل استفاده توسط این مؤلفه‌هاست. ولی از یک دیدگاه گسترده‌تر، مؤلفه‌ها را می‌توان طوری تعمیم بخشید که عناصر اصلی سیستم و تعامل‌های آن‌ها را نشان دهند.

یکی از اهداف مهندسی نرم‌افزار، به‌دست آوردن یک نمای معماری از سیستم است. این نما به‌عنوان چارچوبی عمل می‌کند که از طریق آن فعالیت‌های مشروح‌تر طراحی اجرا می‌شوند. مجموعه‌ای از الگوهای معماری، مهندس نرم‌افزار را قادر به حل مسائل رایج در طراحی می‌سازند.

شا و گارلان [Sha 95a] مجموعه‌ای از خواص را توصیف می‌کنند که خوب است به‌عنوان بخشی از طراحی معماری در نظر گرفته شوند:

خواص ساختاری. در این جنبه از نمایش طراحی معماری، مؤلفه‌های سیستم (مثل پیمانه‌ها، اشیاء، فیلترها) و شیوه‌ی بسته‌بندی و تعامل آن‌ها با یکدیگر تعیین می‌شود. برای مثال، اشیا بسته‌بندی می‌شوند تا هم داده‌ها و هم پردازش‌های عمل‌کننده روی این داده‌ها را پنهان‌سازی کنند و از طریق فراخوانی متدها با یکدیگر تعامل می‌کنند.

خواص عملیاتی اضافی. در توصیف طراحی معماری باید چگونگی بر آورده‌شدن خواسته‌های کارایی، ظرفیتی، قابلیت اطمینان، امنیت، انطباق پذیری، و سایر خصوصیات سیستم در معماری طراحی در نظر گرفته شود.

خانواده‌های سیستم‌های مرتبط. در طراحی معماری باید الگوهایی تکرار پذیر به‌دست آورده شود که به‌طور متداول در طراحی خانواده‌هایی از سیستم‌های مرتبط با آن مواجه می‌شویم. در اصل، طراحی باید توانایی استفاده‌ی مجدد از قطعات سازنده‌ی معماری را داشته باشد.

«معماری نرم‌افزار محصول کاری‌ای است که بیشترین غایدی را در خصوص سرمایه‌گذاری از نظر کیفیت، زمان‌بندی و هزینه می‌دهد.»

لن باس و سایرین

[1] به هر حال، لازم به ذکر است که مجموعه‌ای از عملیات را می‌توان جایگزین مجموعه‌ای دیگر کرد، مشروط بر آن‌که وظیفه‌ی مورد نظر در انتزاع فرایندی بدون تغییر بماند. بنابراین، مراحل لازم برای پیاده‌سازی روال *باز کردن در* یا خودکاربودن در یا متصل‌شدن آن به یک حس‌گر، به‌طور چشمگیری تغییر خواهد کرد.

با توجه به خواص مذکور، طراحی معماری را می‌توان با به‌کارگیری یک یا چند مدل متفاوت به‌نمایش در آورد [Gar95]. در *مدل‌های ساختاری*، معماری به‌صورت مجموعه‌ای سازمان یافته از مؤلفه‌های برنامه نمایش داده می‌شود. *مدل‌های چارچوبی* با تلاش برای شناسایی چارچوب‌های طراحی معماری تکرارپذیری که در انواع مشابه‌کاربردها مشاهده می‌شوند، سطح انتزاع را در طراحی بالا می‌برند. *مدل‌های پویا* به جنبه‌های رفتاری معماری برنامه می‌پردازند و چگونگی تغییر پیکربندی سیستم یا ساختار را به‌عنوان تابعی از رویدادهای خارجی مشخص می‌کنند. در *مدل‌های فرایندی*، طراحی فرایند تجاری یا فنی‌ای که سیستم باید دربرگیرد، کانون توجه قرار می‌گیرد. سرانجام، مدل‌های عملیاتی را می‌توان برای به نمایش در آوردن سلسله مراتب عملیات‌ها در یک سیستم به‌کار برد.

چند زبان توصیف معماری (ADL) متفاوت برای نشان دادن این مدل‌ها توسعه یافته است [Sha95b]. گرچه ADLهای متفاوت فراوانی پیشنهاد شده است، اکثریت آن‌ها سازوکارهایی برای توصیف مؤلفه‌های سیستم و شیوه‌ی اتصال آن‌ها به یکدیگر فراهم می‌آورند.

باید توجه داشته باشید که درباره نقش معماری در طراحی، بحث و جدل وجود دارد. برخی پژوهشگران چنین استدلال می‌کنند که به‌دست آوردن معماری نرم‌افزار را باید از طراحی جدا کرد و آن را بین کنش‌های مهندسی خواسته‌ها و کنش‌های سنتی‌تر طراحی انجام داد. عده‌ای دیگر بر این باورند که به‌دست آوردن معماری، بخشی جدایی ناپذیر از فرایند طراحی است. شیوه مشخص کردن معماری و نقش آن در فصل ۹ بحث خواهد شد.

۸-۳-۳ الگوها (Patterns)

براد اپلتون، *الگوی طراحی* را به‌صورتی که به‌دنبال خواهد آمد، تعریف می‌کند: «الگو عبارت است از کتیبه‌ای دارای نام که حامل جوهره‌ی راهکار اثبات شده برای مسأله‌ای تکراری در حیطه‌ی معین و در میان دغدغه‌های گوناگون است.» [App00]. به بیان دیگر، الگوی طراحی، توصیفی است از یک ساختار طراحی که یک مسأله طراحی را در حیطه‌ای خاص حل می‌کند و «نیروهایی» بینا بینی که ممکن است بر شیوه به‌کارگیری و استفاده از این الگو تأثیرگذار باشند.

هدف هر الگوی طراحی فراهم‌ساختن توصیفی است که طراح به کمک آن بتواند تعیین کند که (۱) آیا این الگو برای کار فعلی قابل استفاده هست، (۲) آیا این الگو قابل استفاده‌ی مجدد هست (تا در زمان طراحی صرفه‌جویی شود) و (۳) آیا این الگو می‌تواند به‌عنوان راهنمایی برای توسعه‌ی یک الگوی مشابه ولی با ساختار و عملکرد متفاوت عمل کند. الگوهای طراحی را در فصل ۱۲ به تفصیل بحث خواهیم کرد.

۸-۳-۴ جداسازی دغدغه‌ها (Separation of Concerns)

جداسازی دغدغه‌ها، یک مفهوم طراحی است [Dij82] که پیشنهاد می‌کند هر مسأله پیچیده‌ای را می‌توان بهتر حل کرد اگر به قطعاتی تقسیم گردد که هر یک را بتوان به‌طور مستقل، حل و/یا بهینه‌سازی کرد. هر *دغدغه*، ویژگی یا رفتاری است که به‌عنوان بخشی از مدل خواسته‌ها برای نرم‌افزار مشخص می‌شود. با جداسازی دغدغه‌ها به قطعات کوچکتر (و بنابراین با قابلیت اداره‌ی بهتر)، زمان و تلاش کمتری صرف حل مسأله می‌شود.

برای دو مسأله P_1 و P_2، اگر پیچیدگی P_1 بیشتر از پیچیدگی P_2 باشد، لازم می‌آید که تلاش به عمل آمده برای حل کردن P_1 بزرگتر از تلاش به عمل آمده برای حل کردن P_2 باشد. به‌عنوان یک حالت کلی، این نتیجه بدیهی است، چون حل مسائل مشکل‌تر زمان بیشتری طلب می‌کند.

اندرز

اجازه ندهید که معماری خودش اتفاق بیفتد. اگر چنین کنید، بقیه‌ی پروژه را صرف این خواهید کرد که طراحی را به اجبار در آن بگنجانید. معماری را به صراحت تعیین کنید.

«هر الگو مسأله‌ای را توصیف می‌کند که بارها و بارها در محیط ما رخ می‌دهد و سپس هسته‌ی راهکار آن مسأله را شرح می‌دهد به گونه‌ای که بتوانید از این راهکار هزاران بار استفاده کنید بدون این که نیاز به دوباره‌کاری داشته باشید.»

کریستوفر الکساندر

همچنین لازم می‌آید که پیچیدگی تلفیق دو مسأله، غالباً بیشتر از مجموع پیچیدگی‌های هر یک از دو مسأله به تنهایی باشد. این به یک راهبرد، تقسیم و حل منجر می‌گردد- حل یک مسأله‌ی پیچیده با تقسیم آن به قطعات کوچکتر قابل مدیریت، آسان‌تر خواهد شد. این واقعیت، در خصوص پیمانه‌بندی نرم‌افزار، اهمیتی چشمگیر دارد.

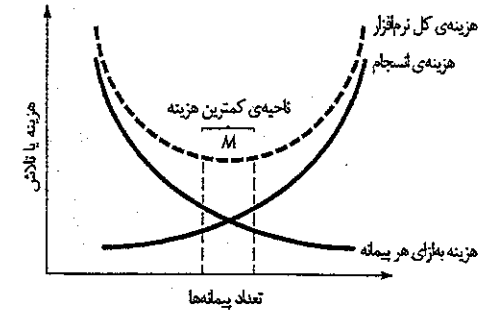
جداسازی دغدغه‌ها در سایر مفاهیم طراحی مرتبط نیز نمود می‌یابد: پیمانه‌بندی، جنبه‌ها، استقلال عملیاتی، و پالایش. هر کدام از این مفاهیم را در بخش‌های بعدی مورد بحث قرار خواهیم داد.

۸-۳-۵ پیمانه‌بندی (Modularity)

پیمانه‌بندی، متداول‌ترین نمود جداسازی دغدغه‌هاست. نرم‌افزار به مؤلفه‌های جداگانه و دارای نام تقسیم می‌شود که گاه از آن‌ها با عنوان پیمانه یاد می‌شود؛ از انسجام این پیمانه‌ها، خواسته‌های مسأله برآورده می‌شود.

گفته شده است که «پیمانه‌بندی تنها صفت نرم‌افزار است که اداره‌ی هوشمندانه‌ی برنامه را میسر می‌سازد» [Mye 78]. نرم‌افزار یکپارچه (یعنی برنامه بزرگی متشکل از تنها یک پیمانه) را مهندس نرم‌افزار نمی تواند به آسانی در اختیار داشته باشد. تعداد مسیرهای کنترلی، گستردگی ارجاع‌ها، تعداد متغیرها و پیچیدگی کلی، شناخت برنامه را تقریباً غیر ممکن می‌سازد. تقریباً در همه‌ی نمونه‌ها، باید طراحی را به چندین پیمانه تقسیم کنید، به این امید که درک و فهم مسأله آسان‌تر شود و در نتیجه هزینه لازم برای ساخت نرم‌افزار کاهش یابد.

با توجه به بحثی که در خصوص جداسازی دغدغه‌ها داشتیم، می‌توان نتیجه گرفت که اگر نرم‌افزار را به بی نهایت قطعه تقسیم کنید، تلاش لازم برای توسعه آن بسیار بسیار کوچک خواهد شد! متأسفانه، نیروهای دیگری وارد صحنه می‌شوند که باعث می‌شوند این نتیجه‌گیری (متأسفانه) نادرست از آب در آید. شکل ۲-۸ نشان می‌دهد که تلاش (هزینه) لازم برای توسعه‌ی یک پیمانه نرم‌افزار، با افزایش تعداد پیمانه‌ها کاهش می‌یابد. اگر مجموعه یکسانی از خواسته‌ها را در نظر بگیریم، بیشتربودن تعداد پیمانه‌ها به معنای کوچک‌ترشدن اندازه‌ی هر پیمانه خواهد بود. ولی با رشد تعداد پیمانه‌ها، تلاش (هزینه) مرتبط با انسجام‌بخشیدن به این پیمانه‌ها نیز رشد می‌کند. این خصوصیات به یک منحنی هزینه (تلاش) کل می‌انجامد که در شکل مشاهده می‌کنید. یک تعداد مشخص M از پیمانه‌ها وجود دارد که به کمترین هزینه‌ی توسعه منجر می‌شود، ولی پیش بینی این مقدار M کار آسانی نیست.



شکل ۲-۸ پیمانه‌بندی و هزینه نرم‌افزار.

اندروز

استدلال مربوط به دغدغه‌ها را می‌توان بسیار بیش از اینها پیش برد. اگر مسأله‌ای را به یک مجموعه مسائل کوچک‌تر فرودستی تقسیم کنید، حل هر کدام از آن‌ها ساده‌تر می‌شود، ولی کنار هم گذاشتن و منسجم ساختن آن‌ها نیز می‌تواند بسیار دشوار باشد.

منحنی‌های نشان داده شده در شکل ۲-۸ راهنمای کیفی مفیدی برای در نظر گرفتن پیمانه‌بندی فراهم می‌آورند. پیمانه‌بندی را باید انجام دهید، ولی باید احتیاط کنید که تعداد پیمانه‌ها در همان حدود M باشد. از پیمانه‌بندی بیش از حد یا کمتر از حد مناسب باید پرهیز کرد. ولی حدود M را چگونه مشخص خواهید کرد؟ نرم‌افزار شما تا چه حد باید پیمانه‌بندی شود؟ پاسخ گویی به این پرسش‌ها به شناخت سایر مفاهیم طراحی نیاز دارد که بعداً در همین فصل به آن‌ها خواهیم پرداخت.

تعداد صحیح پیمانه‌ها برای یک سیستم مفروض کدام است؟

طراحی (و برنامه‌ی حاصل از آن) را باید طوری پیمانه‌بندی کنید که توسعه را بتوان به آسانی برنامه ریزی کرد؛ نسخه‌های نرم‌افزار را بتوان تعیین کرد و تحویل داد؛ تغییرات را بتوان به آسانی امکان داد؛ آزمایش و اشکال زدایی را با بازدهی بیشتر بتوان اجرا نمود و نگهداری دراز مدت را بتوان بدون اثرات جانبی اجرا کرد.

۸-۳-۶ پنهان‌سازی اطلاعات (Information Hiding)

مفهوم پیمانه‌بندی، شما را به این پرسش بنیادی رهنمون می‌شود: «چگونه یک راهکار نرم‌افزاری را برای به‌دست آوردن بهترین مجموعه از پیمانه‌ها تجزیه کنیم؟» از اصل پنهان کردن اطلاعات [Par72] چنین بر می‌آید که پیمانه‌ها باید «با تصمیم‌گیری‌های طراحی مشخص شوند که (هر) کدام از دید بقیه پنهان هستند.» به عبارت دیگر، پیمانه‌ها باید طوری مشخص و طراحی شوند که اطلاعات (الگوریتم‌ها و داده‌های) موجود در یک پیمانه نتواند در دسترس پیمانه‌های دیگری قرار گیرد که به این اطلاعات نیاز ندارند.

پنهان کردن به این معناست که پیمانه‌بندی اثربخش از طریق تعریف یک مجموعه پیمانه‌های مستقل قابل انجام است. این مجموعه پیمانه تنها با پیمانه‌ی دیگری ارتباط برقرار می‌کند که حاوی اطلاعات لازم برای دستیابی به عملکرد نرم‌افزار است. انتزاع به تعریف موجودیت‌های روالی (یا اطلاعاتی) کمک می‌کند که نرم‌افزار را تشکیل می‌دهند. پنهان‌سازی، قید و بندهای دستیابی به جزئیات روالی در داخل یک پیمانه و در هر ساختمان داده‌ی محلی مورد استفاده‌ی آن پیمانه را تعریف و ایجاب می‌کند [Ros75].

نکته‌ی کلیدی

هدف از پنهان‌سازی اطلاعات، پنهان کردن جزئیات ساختمان داده‌ها و پردازش روالی در پس واسط یک پیمانه است. کاربران پیمانه نیازی به آگاهی از این جزئیات ندارند.

استفاده از پنهان کردن اطلاعات به‌عنوان ملاک طراحی برای سیستم‌های پیمانه‌ای، هنگامی بیشترین مزایا را به همراه خواهد داشت که اصلاحاتی طی آزمایش و سپس طی نگهداری نرم‌افزار لازم باشد. از آن‌جا که اکثر جزئیات روالی و داده‌ای از دید سایر بخش‌های نرم‌افزار پنهان هستند، احتمال انتشار خطاهای سهوی طی انجام اصلاحات در سایر نقاط نرم‌افزار، کمتر می‌شود.

۸-۳-۷ استقلال عملیاتی (Functional Independence)

مفهوم استقلال عملیاتی، نتیجه‌ی مستقیم جداسازی دغدغه‌ها، پیمانه‌بندی و مفاهیم پنهان‌سازی اطلاعات و انتزاع است. ویرث [Wir71] و پارناس [Par72] طی یک سری مقالات برجسته درباره طراحی نرم‌افزار، به تکنیک‌های پالایشی اشاره می‌کنند که استقلال پیمانه‌ها را بهبود می‌بخشند. کارهای بعدی که توسط استیونز، مایرز و کنستانتین [Ste74] انجام شد، این مفهوم را بهتر شکل داد.

استقلال عملیاتی با توسعه‌ی پیمانه‌هایی با عملکرد «یگانه» و «دوری جستن» از تعامل بیش از حد با سایر پیمانه‌ها به‌دست می‌آید. به بیان دیگر، باید نرم‌افزار را طوری طراحی کنید که هر پیمانه، به زیر مجموعه‌ی مشخصی از خواسته‌ها بپردازد و از نگاه سایر بخش‌های ساختار برنامه، دارای واسطی ساده باشد. عادلانه است که بپرسید چرا استقلال اهمیت دارد.

توسعه‌ی نرم‌افزارهایی با پیمانه‌بندی اثربخش، یعنی پیمانه‌های مستقل، راحت‌تر است، چون قابلیت‌های عملیاتی را می‌توان به واحدهای کوچک‌تر تقسیم کرد و واسط‌ها ساده می‌شوند (به انشعاب‌هایی فکر کنید که هنگام اجرای توسعه‌ی نرم‌افزار توسط یک تیم ممکن است رخ دهد). پیمانه‌های مستقل را آسان‌تر می‌توان نگهداری (و آزمایش) کرد چون اثرات ثانویه‌ی ناشی از اصلاح کد یا طراحی محدود می‌شوند، انتشار خطا کاهش می‌یابد و ایجاد پیمانه‌هایی با قابلیت استفاده‌ی مجدد میسر می‌شود. به‌طور خلاصه، استقلال عملیاتی کلید طراحی خوب و طراحی کلید کیفیت نرم‌افزار است.

چرا باید در ایجاد پیمانه‌های مستقل بکوشید؟

استقلال با استفاده از دو ملاک کیفیتی قابل ارزیابی است: یکپارچگی (cohesion) و اتصال (coupling). *یکپارچگی*، نشان از قدرت عملیاتی نسبی یک پیمانه دارد. *اتصال*، نشان از استقلال در میان پیمانه‌ها دارد.

نکته‌ی کلیدی
یکپارچگی: شاخصی کیفی از میزان تمرکز یک پیمانه بر تنها یک چیز است.

یکپارچگی بسط طبیعی مفهوم پنهان‌سازی اطلاعات است که در بخش ۶-۳-۸ شرح داده شد. پیمانه‌ی یکپارچه، وظیفه‌ای منفرد بر عهده دارد و به این ترتیب به تعامل اندک با سایر مؤلفه‌های موجود در بخش‌های دیگر سیستم نیاز دارد. به بیان ساده، پیمانه‌های یکپارچه باید (به‌طور ایده آل) تنها یک کار انجام دهند. گرچه همواره باید برای یکپارچگی بالا تلاش کنید، غالباً لازم است و توصیه می‌شود که مؤلفه‌ای از نرم‌افزار چند وظیفه بر عهده داشته باشد. ولی، از مؤلفه‌های «جنون آمیز» (پیمانه‌هایی که وظایف نا مربوط متعدد انجام می‌دهند) باید پرهیز کرد تا طراحی خوبی حاصل شود.

نکته‌ی کلیدی
اتصال: شاخصی کیفی از میزان ارتباط یک پیمانه با سایر پیمانه‌ها و جهان خارج است.

اتصال، شاخصی از ارتباط میان پیمانه‌های موجود در ساختار نرم‌افزار است. اتصال به پیچیدگی واسط‌های میان پیمانه‌ها، نقطه‌ای که در آن ورود یا ارجاع به یک پیمانه انجام می‌شود و داده‌هایی که از واسط عبور می‌کنند، بستگی دارد. در طراحی نرم‌افزار باید تلاش کنید به کمترین اتصال ممکن برسید. اتصال و ارتباط ساده میان پیمانه‌ها به نرم‌افزاری منجر می‌شود که درک آن آسان‌تر است و کمتر در معرض «اثر تَمَوُّجی» [Ste74] قرار دارند؛ اثر تموجی از رخ‌دادن خطا در یک نقطه و انتشار آن در سرتاسر سیستم ناشی می‌شود.

۸-۳-۸ پالایش (Refinement)

پالایش مرحله‌ای (stepwise refinement) یک راهبرد طراحی از بالا به پایین است که اولین بار توسط نیکولاس ویرث [Wir71] پیشنهاد شد. برنامه با سطوح پالایش پیاپی از جزئیات روالی توسعه داده می‌شود. یک سلسله مراتب، با تجزیه‌ی بیان ماکروسکوپی قابلیت عملیاتی (یک انتزاع فرایندی) به شیوه‌ای مرحله‌ای توسعه می‌یابد تا اینکه به دستورات زبان برنامه‌نویسی برسیم.

اندرز
تمایلی برای حرکت فوری به جزئیات کامل، با عبور گذرا از مراحل پالایش، وجود دارد. این مسأله به خطاها و جاافتادگی‌ها می‌انجامد و باعث می‌شود که مرور طراحی دشوارتر شود. پالایش را به‌صورت مرحله به مرحله انجام دهید.

پالایش در واقع همان فرایند تعیین جزئیات است. با توصیف عملکرد (یا توصیف اطلاعات) که در سطح بالایی از انتزاع تعریف می‌شود، کار خود را شروع می‌کنید. یعنی، این بیان، قابلیت عملیاتی یا اطلاعاتی را به‌صورت مفهومی شرح می‌دهد، ولی هیچ اطلاعاتی درباره کارکرد داخلی قابلیت عملیاتی یا ساختار داخلی اطلاعات نمی‌دهد. سپس جزئیات مربوط به بیان اولیه را تعیین می‌کنید و با هر بار پالایش پیاپی، جزئیات بیشتر و بیشتری به آن اضافه می‌کنید.

پالایش و انتزاع، مفاهیمی مکمل یکدیگرند. به کمک انتزاع می‌توانید روال و داده‌ها را از نظر داخلی مشخص کنید، ولی نیاز «افراد متفرقه» به داشتن آگاهی از جزئیات سطح پایین را بر طرف می‌سازد. پالایش به شما کمک می‌کند تا با پیشرفت طراحی، جزئیات سطح پایین را آشکار کنید. هر دو مفهوم شما را در ایجاد یک مدل طراحی کامل، یاری می‌دهند.

۹-۳-۸ جنبه‌ها (Aspects)

در همان حال که تحلیل خواسته‌ها رخ می‌دهد، مجموعه‌ای از «دغدغه‌ها» آشکار می‌شود. این دغدغه‌ها (شامل خواسته‌ها، use case ویژگی‌ها، ساختمان‌های داده‌ها، مسائل مربوط به کیفیت سرویس، شکل‌های گوناگون، مرزهای عقلانی، همکاری‌ها، الگوها و قراردادها می‌شوند.» [AOS07]. در حالت ایده آل، مدل خواسته‌ها را می‌توانید چنان سازمان دهی کنید که هر کدام از دغدغه‌ها (خواسته‌ها) جداسازی شود، به‌صورتی که بتوان به‌طور مستقل به آن پرداخت. ولی در عمل، برخی از این دغدغه‌ها کل سیستم را در بر می‌گیرند و به آسانی نمی توان آن‌ها را به قطعات کوچک‌تر تقسیم کرد.

با شروع طراحی، خواسته‌ها به نمایش طراحی پیمانه‌ای پالایش می‌شود. دو خواسته‌ی A و B را در نظر بگیرید. خواسته‌ی A، پیش‌نیاز خواسته‌ی B است اگر تجزیه (پالایشی) از نرم‌افزار انتخاب شده باشد که در آن، B را نتوان بدون در نظر گرفتن A برآورده ساخت.» [Ros04].

برای مثال، دو خواسته را برای برنامه تحت وب **SafeHomeAssured.com** در نظر بگیرید. خواسته‌ی A از طریق **ACS-DCV use case** توصیف می‌شود که در فصل ۶ بحث شد. در پالایش طراحی، آن دسته از پیمانه‌هایی کانون توجه قرار می‌گیرند که کاربر ثبت شده را قادر می‌سازند تا به ویدیوی دوربین‌های کار گذاشته شده در سرتاسر یک فضا دسترسی داشته باشد. خواسته‌ی B یک خواسته امنیتی عمومی است که بیان می‌کند، اعتبار کاربر ثبت شده باید قبل از به‌کارگیری **SafeHomeAssured.com** تایید شده باشد این خواسته برای همه‌ی قابلیت‌های عملیاتی که در دسترس کاربران ثبت شده‌ی *SafeHome* قرار دارند، مصداق پیدا می‌کند. با رخ دادن پالایش طراحی، A^* نمایش طراحی برای خواسته‌ی A و B^* نمایش طراحی برای خواسته‌ی B است. بنابراین، A^* و B^* نمایش‌هایی از دغدغه‌ها هستند و B^* پیش‌نیاز A^* است.

جنبه، نمایشی از یک دغدغه‌ی پیش‌نیاز است. بنابراین، نمایش طراحی B^* از این خواسته که «*اعتبار کاربر ثبت شده قبل از به‌کارگیری* **SafeHomeAssured.com** *باید تایید شود*» جنبه‌ای از برنامه تحت وب در *SafeHome* است. شناسایی جنبه‌ها به‌طوری که طراحی بتواند به‌صورت مناسب آن‌ها را ضمن انجام پالایش و پیمانه‌بندی، دربرگیرد، اهمیت دارد. در حالت ایده‌آل، هر جنبه به‌صورت پیمانه‌ای جداگانه (مؤلفه) پیاده‌سازی می‌شود نه به‌صورت تکه‌هایی از نرم‌افزار که در سرتاسر چندین مؤلفه «پراکنده» و «در هم و بر هم» شده باشند [Ban06]. برای دستیابی به این مقصود، معماری طراحی باید از سازوکاری برای تعریف یک جنبه پشتیبانی کند- یعنی پیمانه‌ای که پیاده‌سازی یک دغدغه را در سرتاسر همه‌ی دغدغه‌هایی دیگری که پیش‌نیاز آن باشند، میسر سازد.

«اگر کتابی درباره اصول سحر و جادو می‌خوانید، باید هر از چندگاه نظری به جلد آن بیفکنید تا مطمئن شوید که این کتاب به طراحی نرم‌افزار مربوط نمی‌شود.»
بروس تونیاتزینی

نکته‌ی کلیدی
دغدغه‌ی پیش‌نیاز، خصوصیتی از سیستم است که در میان خواسته‌های متفاوت کاربرد دارد.

۱۰-۳-۸ بازآرایی (Refactoring)

یک فعالیت مهم طراحی که برای بسیاری از روش‌های چابک (فصل ۳) پیشنهاد شده است، *بازآرایی* است که تکنیکی برای سازمان‌دهی مجدد به‌شمار می‌رود و طراحی (یا کد) یک مؤلفه را ساده می‌کند، بدون اینکه قابلیت عملیاتی رفتار آن را تغییر دهد. فاولر [Fow00]، بازآرایی را به این صورت تعریف می‌کند: «بازآرایی، عبارت است از فرایند تغییر دادن سیستم نرم‌افزاری به گونه‌ای که رفتار خارجی کد [طراحی] تغییر نکند و در عین حال، ساختار درونی آن بهبود یابد.»

مرجع وب
منابع عالی برای بازآرایی را می‌توان در وب‌سایت زیر مشاهده کرد:
www.refactoring.com

SafeHome

مفاهیم طراحی

صحنه: اتاقک وینود، شروع مدل‌سازی طراحی

نقش‌آفرینان: وینود، جیمی و اد- اعضای تیم نرم‌افزار *SafeHome* شکیرا، عضو جدید تیم نیز حضور دارد.

گفتگوها:

[هر چهار عضو تیم هم‌اکنون از یک سمینار صبح گاهی تحت عنوان «به‌کارگیری مفاهیم پایه‌ی طراحی» برگشته‌اند که یکی از استادان محلی علوم کامپیوتر ارائه داده است.]

وینود: چیزی از این سمینار دستگیرتان شد؟

اد: بیشتر مطالب اش را می‌دانستم، ولی شنیدن دوباره آن هم بد فکری نیست.

جیمی: دانشجو که بودم، هیچ وقت واقعاً نفهمیدم چرا پنهان‌سازی اطلاعات این قدر که گفته می‌شود، اهمیت دارد.

وینود: چون- قضیه اصلی- این است که انتشار خطا در برنامه کاهش پیدا کند. در واقع، استقلال عملیاتی هم همین هدف را دارد.

شکیرا: من فارغ‌التحصیل علوم کامپیوتر نیستم و به همین علت هم خیلی از مطالبی که استاد گفت برایم تازگی داشت. من می‌توانم کدهای خوبی را به سرعت ایجاد کنم. نمی فهمم چرا این حرف‌ها این قدر باید مهم باشد.

جیمی: من کارهای تو را دیده‌ام شکیرا، و راستش را بخواهی، تو خیلی از این چیزها را به‌صورت طبیعی انجام می‌دهی- برای همین هم طراحی‌ها و کدهایت جواب می‌دهند.

شکیرا (با لبخند): خب، من همیشه واقعاً سعی می‌کنم که کدهایم را افراز کنم. طوری که هر افراز به یک چیز اختصاص پیدا کند، واسطهای ساده داشته باشم و هر وقت که امکان داشته باشد، از کدها دوباره استفاده کنم- از این جور چیزها.

اد: پیمانه‌بندی، استقلال عملیاتی، پنهان سازی، الگوها... همین است دیگر!

جیمی: من هنوز اولین دوره برنامه‌نویسی را که گذراندم، یادم هست... به ما یاد می‌دادند که کدها را به‌صورت تکراری پالایش کنیم.

وینود: یک کاری هست که می‌توان روی طراحی انجام داد و من قبلاً نشنیده بودم؛ آن هم «بازآرایی» بود و البته چیزی از «جنبه» هم نشنیده بودم.

شکیرا: فکر کنم که استاد گفت در برنامه‌نویسی حدی (xp) از آن استفاده می‌شود.

اد: بله. تفاوت زیادی با پالایش ندارد فقط آن را پس از تمام‌شدن طراحی و کد نویسی انجام می‌دهند. اگر از من بپرسید، می‌گویم یک جور بهینه‌سازی نرم‌افزار است.

جیمی: خب حالا برگردیم به طراحی *SafeHome*. فکر کنم در توسعه‌ی مدل طراحی برای *SafeHome* باید این مفاهیم را هم به «چک‌لیست» مرور خودمان اضافه کنیم.

وینود: موافقم. ولی این هم نکته مهمی است که موقع توسعه‌ی طراحی، همگی ما باید درباره آن‌ها فکر کنیم.

مرجع وب

انواع الگوهای بازآرایی را در آدرس زیر می‌توانید بیابید.
http://c2.com/cgi/wiki?RefactorigPatterns

هنگامی که نرم‌افزار بازآرایی می‌شود، طراحی موجود برای زوائد، عناصر استفاده نشده‌ی طراحی، الگوریتم‌های ناکارامد یا غیر ضروری، ساختمان‌های داده‌ای ضعیف یا نامناسب، یا هر گونه شکست طراحی دیگر که قابل اصلاح باشد، بررسی می‌شود تا طراحی بهتری به‌دست آید. برای مثال، در اولین دور تکرار طراحی ممکن است مؤلفه‌ای به‌دست آید که یکپارچگی بالایی از خود نشان ندهد (یعنی مثلاً سه وظیفه انجام دهد که رابطه‌ی چندانی با هم ندارند). پس از ملاحظه‌ی دقیق، می‌توانید تصمیم بگیرید که مؤلفه را باید به سه مؤلفه جداگانه بازآرایی کنید که هر کدام یکپارچگی بالایی از خود نشان می‌دهد. نتیجه، نرم‌افزاری خواهد بود که راحت‌تر می‌توان آن را انسجام بخشید، آسان‌تر می‌توان آزمایش کرد و ساده‌تر می‌توان نگهداری کرد.

نتیجه، نرم‌افزاری خواهد بود که راحت‌تر می‌توان به آن انسجام بخشید و آزمودن و نگهداری آن آسان‌تر است.

۸-۳-۱۱ مفاهیم طراحی شیءگرا

الگوی شیءگرا (OO) در مهندسی نرم‌افزار نوین، کاربردی گسترده دارد. در پیوست ۲ برای آنان که با مفاهیمی از قبیل کلاس و شیء، وراثت، پیام، چند ریختی و غیره آشنایی ندارند، مفاهیم طراحی شیءگرا ارائه شده است.

۸-۳-۱۲ کلاس‌های طراحی (Design Classes)

در مدل خواسته‌ها، مجموعه‌ای از کلاس‌های تحلیل (analysis classes) تعریف می‌شود (فصل ۶) که هر کدام، عنصری از دامنه‌ی مسأله را با توجه خاص به جنبه‌هایی از مسأله توصیف می‌کند که در معرض دید کاربر قرار دارند. سطح انتزاع یک کلاس تحلیل، نسبتاً بالاست.

به موازاتی که مدل طراحی تکامل پیدا می‌کند، مجموعه‌ای از *کلاس‌های طراحی* را تعریف می‌کنید که کلاس‌های تحلیل را با فراهم آوردن جزئیات طراحی پالایش می‌کنند (این جزئیات به کلاس‌ها امکان پیاده‌سازی می‌دهند) و یک زیرساخت نرم‌افزاری را پیاده‌سازی می‌کنند که راهکار تجاری را پشتیبانی می‌کند. پنج نوع متفاوت از کلاس‌های طراحی می‌توان توسعه داد که هر کدام لایه متفاوتی از معماری طراحی را نمایش می‌دهند [Amb01]:

طراح چه نوع کلاس‌هایی ایجاد می‌کند؟

- *کلاس‌های واسط کاربری*، همه‌ی انتزاع‌های لازم برای تعامل میان انسان و کامپیوتر (HCI) را تعریف می‌کنند. در بسیاری موارد، HCI در حیطه‌ی یک *استعاره* (دسته چک، فرم سفارش، ماشین فکس) ظاهر می‌شود و ممکن است کلاس‌های طراحی برای واسط نمایشی از عناصر این استعاره باشند.
- *کلاس‌های دامنه‌ی تجاری*، غالباً شکل پالایش یافته‌ی کلاس‌های تحلیلی هستند که قبلاً تهیه شده‌اند. این کلاس‌ها صفات و سرویس‌ها (متد هایی) را مشخص می‌کنند که برای پیاده‌سازی عنصری از دامنه‌ی تجاری مورد نیازند.
- *کلاس‌های پردازش*، انتزاع‌های تجاری سطح پایین لازم برای مدیریت کامل کلاس‌های دامنه‌ی تجاری را پیاده‌سازی می‌کنند.
- *کلاس‌های ماندگار*، انباره‌های داده‌ها (مثلاً بانک‌های اطلاعاتی) را نشان می‌دهند که پس از اجرای نرم‌افزار، ماندگار می‌شوند.

- *کلاس‌های سیستمی*، عملیات‌های مدیریتی و کنترلی را پیاده‌سازی می‌کنند که کارکرد سیستم را میسر می‌سازند و ارتباط میان درون و بیرون محیط کامپیوتری را بر قرار می‌سازند.

با شکل‌گیری معماری، سطح انتزاع با تبدیل هر کلاس تحلیل به یک نمایش طراحی، بیشتر کاهش می‌یابد. یعنی کلاس‌های تحلیل، اشیای داده‌ای (و سرویس‌های مرتبط به‌کار رفته در آن‌ها) را با استفاده از اصطلاحات رایج در دامنه‌ی تجاری مورد نظر به نمایش در می‌آورند. کلاس‌های طراحی، به‌طور چشمگیر، جزئیات فنی بیشتری به‌عنوان راهنمای پیاده‌سازی فراهم می‌سازند.

آرلو و نویشتات [Arl02] پیشنهاد می‌کنند که هر کلاس طراحی مرور شود تا اطمینان حاصل آید که از شکل خوبی برخوردار است. آن‌ها چهار مشخصه برای کلاس طراحی «خوش‌فرم» بر می‌شمرند:

کلاس طراحی «خوش‌فرم» چیست؟

کامل و کافی (Complete and Sufficient). طراحی باید پنهان‌سازی کاملی از همه‌ی صفات و متدهایی باشد که به‌طور منطقی برای آن کلاس انتظار می‌رود (بر اساس تفسیری قابل فهم از نام کلاس). برای مثال، کلاس **Scene** که برای نرم‌افزار ویرایش تصاویر ویدیویی تعریف می‌شود، تنها در صورتی کامل است که حاوی همه‌ی صفات و متدهایی باشد که به‌طور منطقی برای ایجاد یک صحنه‌ی ویدیویی انتظار می‌رود. *کافی بودن* به آن معناست که کلاس تنها حاوی متدهایی باشد که برای دستیابی به هدف کلاس کفایت می‌کنند، نه کمتر و نه بیشتر.

سادگی (Primitiveness). متدهای مرتبط با یک کلاس طراحی باید انجام یک سرویس برای کلاس را کانون توجه قرار دهند. هنگامی که آن سرویس با یک متد پیاده‌سازی شد، کلاس نباید راه دیگری برای دستیابی به همان هدف فراهم سازد. برای مثال، کلاس **VideoClip** برای نرم‌افزار ویرایش تصاویر ویدیویی ممکن است دارای صفاتی از قبیل *start-point* و *point-end* باشد تا نقاط شروع و پایان کلیپ را مشخص کند (ویدیوی دانلودشده در سیستم ممکن است بلندتر از کلیپ مورد استفاده باشد). متدهای *setStartPoint()* و *setEndPoint()* تنها روش ممکن برای تعیین نقاط شروع و پایان کلیپ هستند.

یکپارچگی بالا (High Cohesion). یک کلاس طراحی یکپارچه دارای مجموعه‌ای کوچک و متمرکز از مسؤولیت‌هاست که صفات و متدها را برای پیاده‌سازی همان مسؤولیت‌ها به‌کار می‌برد. برای مثال، کلاس **VideoClip** ممکن است حاوی مجموعه‌ای از متدها برای ویرایش کلیپ ویدیویی باشد. مادامی که هر متد تنها صفات مرتبط با کلیپ ویدیویی را مورد توجه قرار دهد، یکپارچگی حفظ خواهد شد.

اتصال پایین (Low Coupling). در مدل طراحی، کلاس‌های طراحی باید با یکدیگر همکاری کنند. ولی این همکاری باید در یک سطح کمینه‌ی قابل قبول حفظ گردد. اگر در یک مدل طراحی، میزان اتصال بالا باشد (همه‌ی کلاس‌های طراحی با همه‌ی کلاس‌های طراحی دیگر همکاری کنند)، پیاده‌سازی سیستم، آزمایش آن و نگهداری آن در گذر زمان دشوار می‌شود. به‌طور کلی، کلاس‌های طراحی در داخل یک زیر سیستم فقط باید آگاهی محدودی از سایر کلاس‌ها داشته باشد. این محدودیت، که قانون دمتر نامیده می‌شود[Lie03]، پیشنهاد می‌کند که یک متد فقط باید به متدهای موجود در کلاس‌های همسایه پیام ارسال کند.[1]

[1] یک روش کمتر رسمی برای بیان قانون دمتر به این صورت است که: «هر واحد تنها باید با دوستان خود صحبت کند و صحبتی با غریبه‌ها نداشته باشد.»

SafeHome

پالایش کلاس تحلیل به کلاس طراحی

صحنه: اتاقک اد، در شروع مدل‌سازی طراحی.

نقش‌آفرینان: وینود و اد- اعضای تیم نرم‌افزار *SafeHome*.

گفتگوها:

[اد در حال کار روی کلاس **FloorPlan** است (بخش ۳-۵-۶ و شکل ۱۰-۶) و آن را برای مدل طراحی اصلاح کرده است.]

اد: کلاس **FloorPlan** را که یادت هست؟ به‌عنوان بخشی از قابلیت‌های مدیریت خانه و پایش از آن استفاده می‌شد.

وینود (سرش را تکان می‌دهد): آره، فکر کنم موقع بحث CRC برای مدیریت منزل از این کلاس استفاده کردیم.

اد: درست است. به هر حال، دارم آن را برای طراحی پالایش می‌کنم. می‌خواهم نشان بدهم که کلاس **FloorPlan** چطور واقعاً پیاده‌سازی شود. من آن را به‌صورت یک مجموعه فهرست‌های مرتبط [یک ساختمان داده‌ی خاص] پیاده‌سازی می‌کنم. خلاصه می‌بایستی کلاس تحلیل **FloorPlan** (شکل ۱۰-۶) را پالایش می‌کردم و در واقع یک جورهایی آن را ساده می‌کردم.

وینود: کلاس تحلیل، چیزها را فقط در دامنه‌ی مسأله نشان می‌داد، یعنی در واقع روی صفحه‌ی کامپیوتر، که برای کاربر نهایی قابل مشاهده بودند. درست است؟

اد: بله، ولی برای کلاس طراحی **FloorPlan**، باید چیزهایی اضافه کنم که خاص پیاده‌سازی هستند. لازم بود که نشان بدهم **FloorPlan** مجموعه‌ای از یک سری قطعات است (پس کلاس **Segment** است) و نشان بدهم کلاس **Segment** از فهرست‌هایی برای قطعات دیوار، پنجره‌ها، درها و غیره تشکیل می‌شود. کلاس **Camera** با **FloorPlan** همکاری دارد و بدیهی است که دوربین‌های فراوانی در نقشه ساختمان وجود دارد.

وینود: اووه، بگذار ببینم این کلاس طراحی **FloorPlan** چه شکلی است. [اد شکل ۳-۸ را به وینود نشان می‌دهد.]

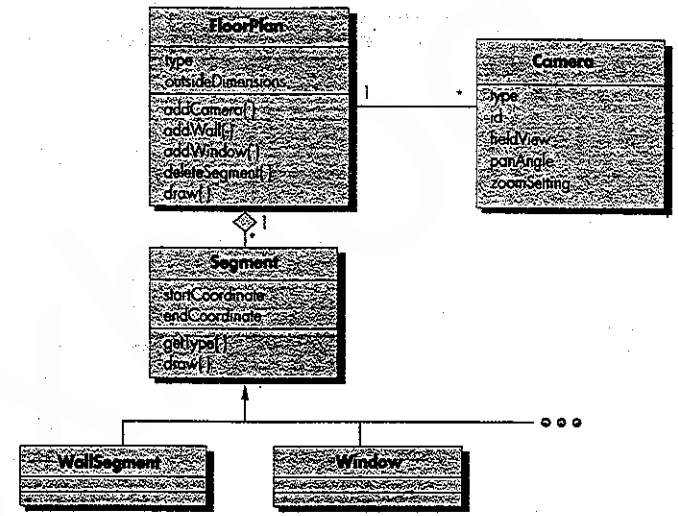
وینود: بسیار خب، حالا می‌فهمم سعی داری چه کار کنی. به این ترتیب می‌توانی نقشه ساختمان را به‌راحتی اصلاح کنی، چون آیتم‌های جدید را می‌شود به فهرست اضافه کنی یا از آن کم کنی، بدون این که مشکلی پیش بیاید.

اد (سری تکان می‌دهد): بله، فکر می‌کنم جواب بدهد.

وینود: من هم فکر می‌کنم.

۸-۴ مدل طراحی (Design Model)

همان طور که از شکل ۴-۸ پیداست، مدل طراحی را از دو بُعد متفاوت می‌توان در نظر گرفت. *بُعد فرایندی*، تکامل مدل طراحی را به موازات اجرای وظایف طراحی به‌عنوان بخشی از فرایند نرم‌افزار نشان می‌دهد. *بُعد انتزاعی*، سطح جزئیات را به موازات تبدیل هر عنصر از مدل تحلیل به یک مدل

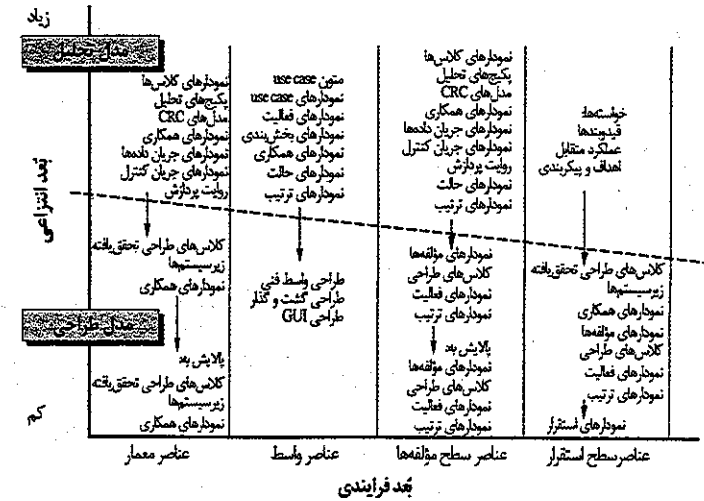


شکل ۳-۸ کلاس طراحی برای FloorPlan (نقشه ساختمان) و مجموعه مرکب برای کلاس.

طراحی و سپس پالایش تکراری، نمایش می‌دهد. همان طور که از شکل ۴-۸ پیداست، خط چین، مرز میان مدل‌های تحلیل و طراحی را نشان می‌دهد. در برخی موارد، تمایز روشن میان مدل‌های تحلیل و طراحی امکان پذیر است. در مواردی، مدل تحلیل به آهستگی با طراحی درآمیخته می‌شود و تمایز میان آن‌ها چندان آشکار نیست.

نکته‌ی کلیدی

مدل طراحی چهار عنصر اصلی دارد: داده‌ها، معماری، مؤلفه‌ها و واسط



شکل ۴-۸ ابعاد مدل طراحی.

در عناصر مدل طراحی، بسیاری از همان نمودارهای UML به‌کار گرفته می‌شود[1] که قبلاً در مدل تحلیل به‌کار برده شدند. اختلاف آن‌ها در این است که نمودارهای مذکور به‌عنوان بخشی از طراحی، پالایش می‌شوند و جزئیاتی به آن‌ها افزوده می‌شود؛ جزئیات بیشتری که در خصوص پیاده‌سازی فراهم می‌آید و سبک و ساختار معماری، مؤلفه‌هایی که در داخل معماری قرار می‌گیرند و واسط‌هایی میان این مؤلفه‌ها و با دنیای خارج که بر همه‌ی آن‌ها تأکید خواهد شد.

به هر حال، لازم به ذکر است که عناصر مدل نشان داده شده در راستای محور افقی، همواره به شیوه‌ای ترتیبی توسعه نمی یابند. در اکثر موارد، طراحی معماری مقدماتی، صحنه را آماده می‌کند و پس از آن نوبت به طراحی واسط‌ها و طراحی در سطح مؤلفه‌ها می‌رسد، که غالباً به‌صورت موازی رخ می‌دهند. مدل استقرار معمولاً تا توسعه کامل طراحی به تأخیر می‌افتد.

می‌توانید الگوهای طراحی (فصل ۱۲) را در هر نقطه از طراحی به‌کار ببرید. با این الگوها می‌توانید آگاهی‌های طراحی را در مسائل خاص دامنه‌ای که دیگران دیده و حل کرده‌اند، به‌کار بگیرید.

> «پرسش‌هایی درباره اینکه آیا طراحی لازم یا قابل انجام است، کاملاً بی‌مورد است. طراحی اجتناب ناپذیر است. گزینه دیگر در مقابل طراحی کردن، طراحی نکردن نیست، بد طراحی کردن است.»
>
> داگلاس مارتین

۸-۴-۱ عناصر طراحی داده‌ها

طراحی داده‌ها (که گاهی از آن به‌عنوان معماری داده‌ها یاد می‌شود) همانند فعالیت‌های دیگر مهندسی نرم‌افزار، یک مدل از داده‌ها و/یا اطلاعات ایجاد می‌کند که در سطح بالایی از انتزاع نمایش داده می‌شود (دیدگاه مشتری/کاربر نسبت به داده‌ها). این مدل داده‌ها سپس به نمایش‌هایی پالایش می‌شود که به تدریج جزئیات خاص پیاده‌سازی بر آن‌ها افزوده می‌شود و با سیستم کامپیوتری قابل پردازش هستند. در بسیاری از کاربردهای نرم‌افزاری، معماری داده‌ها تأثیری بنیادی بر معماری نرم‌افزاری دارد که باید آن داده‌ها را پردازش کند.

نکته‌ی کلیدی

طراحی داده‌ها در سطح معماری، قابل‌ها یا بانک‌های اطلاعاتی را کانون توجه قرار می‌دهد؛ طراحی داده‌ها در سطح مؤلفه‌ها، به ساختمان‌های داده‌ای مورد نیاز برای پیاده‌سازی اشیای داده‌ای محلی توجه دارد.

ساختمان داده‌ها همواره بخش مهمی از طراحی نرم‌افزار بوده است. در سطح مؤلفه‌های برنامه، طراحی ساختمان‌های داده‌ها و الگوریتم‌های مورد نیاز برای دستکاری آن‌ها در ایجاد برنامه‌های کاربردی با کیفیت بالا، اهمیت اساسی دارد. در سطح برنامه کاربردی، برگردان یک مدل داده‌ای (که به‌عنوان بخشی از مهندسی خواسته‌ها به‌دست می‌آید) به یک بانک اطلاعاتی، اساس دستیابی به اهداف تجاری سیستم است. در سطح تجاری، مجموعه اطلاعات ذخیره شده در بانک‌های اطلاعاتی نامتجانس و سازمان‌دهی شده در یک «انبار داده»، کشف دانش یا داده‌کاوی‌ای را امکان‌پذیر می‌سازند که می‌توانند بر موفقیت خود شرکت تجاری تأثیر گذار باشند. در هر مورد، طراحی داده‌ها نقشی مهم دارد. طراحی داده‌ها را با تفصیل بیشتر در فصل ۹ بحث خواهیم کرد.

۸-۴-۲ عناصر طراحی معماری

طراحی معماری برای نرم‌افزار، هم‌ارز نقشه برای ساختمان است. نقشه‌ی ساختمان، چیدمان کلی اتاق‌ها؛ اندازه‌ی آن‌ها، شکل آن‌ها و واسط آن‌ها با یکدیگر را به تصویر می‌کشد؛ و درها و پنجره‌هایی که حرکت به درون و بیرون خانه را امکان‌پذیر می‌سازند. نقشه ساختمان، دیدی کلی از ساختمان به ما می‌دهد. عناصر طراحی معماری هم دیدی کلی از نرم‌افزار به‌دست می‌دهند.

> «می‌توانید از تخته پاک کن روی تخته سیاه استفاده کنید یا در سایت ساختمانی از پتک استفاده کنید.»
>
> فرانک لوید رایت

[1] اگر با UML آشنایی ندارید، معرفی مختصری از این نمادگذاری مهم مدل‌سازی در پیوست ۱ ارائه شده است.

مدل معماری [Sha96] از سه منبع به‌دست می‌آید: (۱) اطلاعات مربوط به دامنه‌ی کاربرد برای نرم‌افزاری که قرار است ساخته شود؛ (۲) عناصر خاصی از مدل خواسته‌ها از قبیل نمودارهای جریان داده‌ها یا کلاس‌های تحلیل، روابط و همکاری‌های میان آن‌ها برای مسأله‌ی مورد نظر؛ و (۳) قابلیت دسترسی به سبک‌های معماری (فصل ۹) و الگوهای معماری (فصل ۱۲).

عنصر طراحی معماری معمولاً به‌عنوان مجموعه‌ای از زیر سیستم‌های متصل به هم تصویر می‌شود، که غالباً از پکیج‌های تحلیل در مدل خواسته‌ها به‌دست می‌آیند. هر کدام از این زیر سیستم‌ها ممکن است معماری خاص خود را داشته باشد (مثلاً ساختار واسط گرافیکی کاربر ممکن است مطابق با سبک موجود برای واسط‌های کاربری تعیین شود). تکنیک‌های مربوط به کسب عناصر خاص از مدل معماری در فصل ۹ ارائه خواهند شد.

> «عوام با طراحی بد آشنایی بیشتری دارند تا طراحی خوب. درواقع طوری بار آمده‌اند که طراحی بد را ترجیح دهند زیرا با آن زندگی می‌کنند. نو تهدید است و کهنه، قوت قلبی دوباره.»
>
> **پاول راند**

۸-۴-۳ عناصر طراحی واسط‌ها

طراحی واسط‌ها برای نرم‌افزار، مشابه با مجموعه‌ای از ترسیم‌های مشروح (و مشخصات) برای درها، پنجره‌ها و سایر امکانات خارجی برای یک خانه است. این ترسیم‌ها، اندازه و شکل درها و پنجره‌ها، شیوه‌ی عملکرد آن‌ها، و چگونگی اتصالات مربوط به امکانات خارجی (لوله کشی آب و گاز، سیم کشی برق و تلفن) و توزیع این امکانات در میان اتاق‌های ترسیم شده در نقشه ساختمان را به تصویر می‌کشند. به ما می‌گویند که زنگ خانه کجا قرار دارد، آیا آیفونی برای اعلام حضور مهمان لازم هست، و سیستم امنیتی چگونه باید نصب شود. در اصل، این ترسیم‌های مشروح (و مشخصات) برای درها، پنجره‌ها و امکانات خارجی به ما می‌گویند که چیزها و اطلاعات چگونه به درون و بیرون خانه و در داخل اتاق‌هایی که بخشی از نقشه ساختمان هستند، جریان پیدا می‌کنند. عناصر طراحی واسط‌ها برای نرم‌افزار، جریان‌های اطلاعات به درون و بیرون سیستم و چگونگی برقراری ارتباط میان مؤلفه‌های تعریف شده به‌عنوان بخشی از معماری را به تصویر می‌کشند.

> **نکته‌ی کلیدی**
>
> عنصر طراحی واسط سه بخش دارد: واسط کاربری، واسط‌هایی با سیستم خارج از برنامه کاربردی و واسط‌هایی با مؤلفه‌های داخل برنامه کاربردی.

سه عنصر مهم در طراحی واسط‌ها وجود دارد: (۱) واسط کاربری (UI)؛ (۲) واسط‌های خارجی با سایر سیستم‌ها، دستگاه‌ها، شبکه‌ها، یا سایر تولید کنندگان یا مصرف کنندگان اطلاعات؛ و (۳) واسط‌های داخلی میان مؤلفه‌های طراحی گوناگون. این عناصر طراحی واسط‌ها به نرم‌افزار امکان می‌دهند که ارتباط خارجی بر قرار کند و همکاری و ارتباطات داخلی میان مؤلفه‌های تشکیل‌دهنده‌ی معماری نرم‌افزار را میسر می‌سازند.

> «هر بار دورشدن از کار و برگشتن دوباره، قدری آسایش به همراه دارد زیرا آن گاه که به کار برمی‌گردید، در قضاوت خویش مطمئن‌تر خواهید شد. قدری دورتر بروید تا کار کوچک‌تر شود و بخش بزرگ‌تری از آن را بتوان در یک نگاه دید؛ در این صورت، ناهماهنگی و عدم تناسب راحت‌تر به چشم خواهد آمد.»
>
> **لئوناردو داوینچی**

طراحی UI (که اکنون به‌طور فزاینده‌ای از آن به‌عنوان *طراحی قابلیت استفاده* یاد می‌شود) یک کنش اصلی در مهندسی نرم‌افزار به‌شمار می‌رود که به تفصیل در فصل ۱۱ بحث خواهد شد. طراحی واسط کاربری، شامل عناصر زیبایی‌شناسی (نظیر چیدمان، رنگ، گرافیک، سازوکارهای تعامل)، عناصر ارگونومی (نظیر چیدمان و طرز قرار گرفتن اطلاعات، استعاره‌ها، گشت و گذار در UI) و عناصر فنی (نظیر الگوهای UI، مؤلفه‌های قابل استفاده‌ی مجدد) می‌شود. به‌طور کلی، UI یک زیر سیستم منحصر به فرد در داخل معماری کاربرد کلی است.

طراحی واسط‌های خارجی به اطلاعات قطعی درباره موجودیتی نیاز دارد که اطلاعات به آن ارسال یا از آن دریافت می‌شود. در هر حال، این اطلاعات را باید طی مهندسی خواسته‌ها (فصل ۵) جمع‌آوری و هنگام شروع طراحی واسط‌ها اعتبارسنجی کرد.[1] طراحی واسط‌های خارجی باید، شامل چک کردن خطاها و (در صورت نیاز) ویژگی‌های امنیتی مناسب باشد.

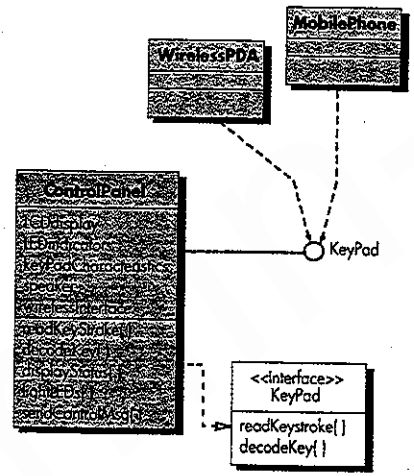
[1] خصوصیات واسط ممکن است با زمان، تغییر کند. بنابراین، طراح باید اطمینان حاصل کند که مشخص‌سازی واسط به درستی و کامل انجام شده است.

طراحی واسط‌های داخلی، رابطه‌ای تنگاتنگ با طراحی در سطح مؤلفه‌ها دارد (فصل ۱۰). تبدیل کلاس‌های تحلیل به کلاس‌های طراحی، همه‌ی عملیات‌ها و الگوهای پیام رسانی لازم برای برقراری ارتباط و همکاری میان عملیات‌های موجود در کلاس‌های گوناگون را به نمایش می‌گذارد. هر پیام باید طوری طراحی شود که انتقال اطلاعات ضروری و خواسته‌های خاص عملیات درخواست شده را پاسخ گو باشد. اگر رویکرد کلاسیک «ورودی- پردازش- خروجی» برای طراحی انتخاب شده باشد، واسط هر مولفه نرم‌افزار بر اساس نمایش‌های جریان داده‌ها و قابلیت عملیاتی توصیف شده در یک روایت پردازش، طراحی می‌شود.

در برخی موارد، واسط تا حد زیادی به شیوه‌ی یک کلاس مدل‌سازی می‌شود. به زبان UML، واسط به‌صورتی که به دنبال خواهد آمد، تعریف می‌شود [OMG03a]: «رابط، مشخص کننده‌ای برای عملیات‌های قابل مشاهده از بیرون یک کلاس، مؤلفه، یا سایر طبقه بندی‌ها (شامل زیر سیستم‌ها) بدون تعیین مشخصات ساختار داخلی است.» به بیان ساده تر، واسط به مجموعه‌ای از عملیات گفته می‌شودکه بخشی از رفتار یک‌کلاس را توصیف می‌کند و دستیابی به این عملیات‌ها را فراهم می‌آورد.

برای مثال، در قابلیت امنیتی در محصول *SafeHome* از یک پانل کنترل استفاده می‌شود که به صاحبخانه این امکان را می‌دهد تا جنبه‌های معینی از قابلیت امنیتی را کنترل کند. قابلیت‌های عملیاتی پانل‌کنترل در نسخه‌ی پیشرفته‌ای از این سیستم از طریق PDA بی سیم یا تلفن همراه قابل پیاده‌سازی است.

کلاس **ControlPanel** (شکل ۵-۸) رفتار مرتبط با یک صفحه کلید را فراهم می‌آورد و بنابراین، باید عملیات‌های *readKeyStroke()* و *decodeKey()* را پیاده‌سازی کند.



شکل ۵-۸ نمایش واسط‌ها برای پانل کنترل.

اگر قرار باشد که این دو عملیات در اختیار سایر کلاس‌ها نیز قرار داده شوند (که در این مورد خاص، دو کلاس **WirelessPDA** و **MobilePhone** خواهند بود)، تعریف یک واسط به‌صورت نشان داده شده در شکل، مفید خواهد بود. واسط، که **KeyPad** نامیده می‌شود، به‌صورت نمونه‌ی اولیه‌ی <<interface>> یا یک دایره کوچک برچسب‌دار مشخص می‌شود که با یک خط به کلاس متصل

> **مرجع وب**
>
> اطلاعاتی بسیار باارزش را درباره طراحی می‌توان در www.useit.com یافت.

> «یک اشتباه رایج که افراد هنگام طراحی چیزهای کاملاً ضد خطا مرتکب می‌شوند، دست کم گرفتن استعداد آدم‌های نادان است.»
>
> **داگلاس آدامز**

می‌گردد. واسط، بدون صفات و مجموعه عملیات‌های لازم برای دستیابی به رفتار یک صفحه کلید تعریف می‌شود.

خط‌چین با مثلث توخالی در انتهای آن (شکل ۵-۸) نشان می‌دهد که کلاس **ControlPanel** عملیات‌های **KeyPad** را به‌عنوان بخشی از رفتار آن فراهم می‌سازد. در زبان UML، این را با تحقق‌بخشی (realization) مشخص می‌کنیم. یعنی بخشی از رفتار **ControlPanel** با تحقق بخشیدن به عملیات‌های **KeyPad** پیاده‌سازی خواهند شد. این عملیات‌ها برای سایر کلاس‌هایی که به این واسط دستیابی دارند نیز ارائه می‌شوند.

۸-۴-۴ عناصر طراحی در سطح مؤلفه‌ها

طراحی در سطح مؤلفه‌ها برای نرم‌افزار، هم ارز مجموعه‌ای از ترسیم‌های مشروح (و مشخصات) مربوط به هر اتاق در خانه‌اند. این ترسیم‌ها، سیم کشی و لوله کشی به هر کدام از اتاق‌ها، محل قرار گرفتن پریزها و کلیدهای دیواری، شیر آلات، دستشویی‌ها، دوش‌ها، وان‌ها، چاه‌ها، کابینت‌ها و کمدها را به تصویر می‌کشند. توصیف کف پوش‌ها، قالب‌هایی که باید استفاده شوند و هرگونه جزئیات دیگر مربوط به اتاق نیز در همین ترسیم‌ها آورده می‌شود. طراحی در سطح مؤلفه‌ها برای نرم‌افزار، توصیف کاملی است از جزئیات داخلی هر مؤلفه‌ی نرم‌افزار. برای نیل به این مقصود، طراحی در سطح **مؤلفه‌ها**، ساختمان داده‌ها برای کلیه اشیای داده‌ای محلی و جزئیات الگوریتمی برای کلیه پردازش‌هایی که در داخل مؤلفه رخ می‌دهد و واسطی که دستیابی به همه‌ی عملیات‌ها (رفتارهای) مؤلفه را امکان پذیر می‌سازد، تعریف می‌کند.

> جزئیات، جزئیات نیستند بلکه طراحی را می‌سازند.
> چارلز ایمز

در حیطه‌ی مهندسی نرم‌افزار شیءگرا، هر مؤلفه‌ی نمودار UML، به‌صورت شکل ۶-۸ نمایش داده می‌شود. در این شکل، مؤلفه‌ای با نام SensorManagement (بخشی از قابلیت امنیتی در *SafeHome*) نمایش داده شده است. پیکان خط‌چین، این مؤلفه را به کلاسی با نام **Sensor** متصل می‌کند که به آن نسبت داده شده است. مؤلفه‌ی **SensorManagement** همه‌ی وظایف مرتبط با حسگرهای *SafeHome*، از جمله پایش و پیکربندی آن‌ها را اجرا می‌کند. بحث بیشتر درباره نمودارهای مؤلفه‌ها در فصل ۱۰ ارائه خواهد شد.

شکل ۶-۸ یک نمودار مؤلفه در UML.

جزئیات طراحی یک مؤلفه را می‌توان در سطوح انتزاع متفاوت فراوان مدل‌سازی کرد. از نمودار فعالیت‌های UML می‌توان برای نمایش منطق پردازش بهره برد. «جریان روالی مشروح»[1] برای یک مؤلفه را می‌توان با استفاده از شبه کد (نمایشی شبیه به زبان برنامه‌نویسی که در فصل ۱۰ شرح داده خواهد شد) یا یک شکل نموداری دیگر (مثلاً نمودار گردشی یا نمودار کادری) نمایش داد. ساختارهای الگوریتمی از همان قواعد وضع‌شده برای برنامه‌نویسی ساخت‌یافته (یعنی مجموعه‌ای از

[1] Detailed Procedural Flow

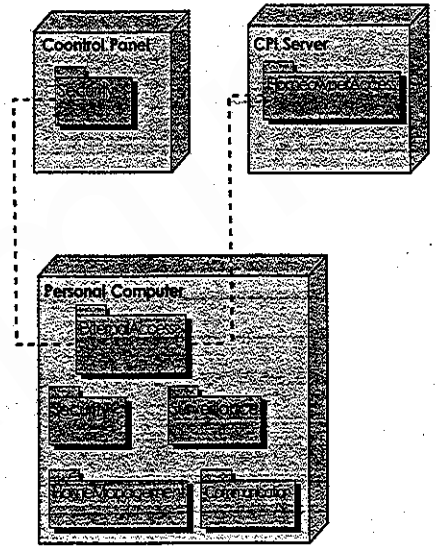
ساخت‌های روالی مقید) پیروی می‌کنند. ساختمان داده‌ها، که بر اساس ماهیت اشیای داده‌ای پردازش شونده انتخاب می‌شوند، معمولاً با استفاده از شبه کد یا زبان برنامه‌نویسی به‌کاررفته در پیاده‌سازی، مدل‌سازی می‌شوند.

۸-۴-۵ عناصر طراحی در سطح استقرار

عناصر طراحی در سطح استقرار چگونگی تخصیص یافتن زیر سیستم‌ها و قابلیت‌های عملیاتی نرم‌افزار را در داخل محیط کامپیوتری‌ای نشان می‌دهند که نرم‌افزار را پشتیبانی می‌کند. برای مثال، عناصر محصول *SafeHome* طوری پیکربندی شده‌اند که در سه محیط کامپیوتری اولیه- PC خانگی، پانل کنترل *SafeHome* و کارگزاری که داخل شرکت نصب شده است و دستیابی اینترنتی را فراهم می‌آورد- عمل کنند.

نکته‌ی کلیدی
نمودارهای استقرار در شکل توصیفی آغاز می‌شوند؛ در این شکل، محیط استقرار در عبارت‌هایی کلی توصیف می‌شود. بعداً، از شکل نمونه‌ای استفاده می‌شود و عناصر پیکربندی به صراحت نمایش داده می‌شوند.

در طی طراحی، یک نمودار استقرار UML توسعه داده می‌شود و سپس به‌صورت نشان داده شده در شکل ۷-۸ پالایش می‌یابد. در این شکل، سه محیط کامپیوتری مذکور نشان داده شده‌اند (در واقع محیط‌های بیشتر وجود خواهند داشت از جمله حسگرها، دوربین‌ها و غیره). زیر سیستم‌ها (قابلیت‌های عملیاتی) قرار داده شده در هر عنصر از محیط کامپیوتری مشخص شده‌اند. برای مثال، کامپیوتر شخصی، زیر سیستم‌هایی را در خود جای می‌دهد که ویژگی‌های مربوط به امنیت، پایش محیط، مدیریت منزل و ارتباطات را پیاده‌سازی می‌کنند. به‌علاوه، یک زیر سیستم دستیابی خارجی هم طراحی شده است که همه‌ی تلاش‌های به عمل آمده جهت دستیابی به سیستم *SafeHome* از یک منبع خارجی را مدیریت کند. هر زیر سیستم باید تجزیه و تحلیل شود تا مؤلفه‌هایی که پیاده‌سازی می‌کند، مشخص گردد.



شکل ۷-۸ یک نمودار استقرار در UML.

نمودار شکل ۷-۸ به صورت یک *توصیف گر* ارائه شده است، به این معنی که نمودار استقرار، محیط کامپیوتری را نشان می‌دهد، ولی جزئیات پیکربندی را به صراحت مشخص نمی کند. برای مثال، «کامپیوتر شخصی» دیگر بیش از این مشخص نمی شود. می‌تواند کامپیوتری با سیستم عامل Mac یا Windows باشد، یک ایستگاه کاری Sun باشد یا با سیستم عامل Linux راه اندازی شده باشد. این جزئیات هنگامی ارائه خواهد شد که نمودار استقرار به *شکل نمونه‌ای* و طی مراحل طراحی یا در آغاز ساخت مرور شود. هر نمونه از استقرار (که یک پیکربندی مشخص و دارای نام است) تعیین می‌شود.

۸-۵ خلاصه

طراحی نرم‌افزار با به پایان رسیدن اولین دور تکرار مهندسی خواسته‌ها آغاز می‌شود. هدف از طراحی نرم‌افزار، به‌کارگیری مجموعه‌ای از اصول، مفاهیم و کارهاست که به توسعه‌ی محصول یا سیستمی با کیفیت بالا می‌انجامد. هدف از طراحی، ایجاد مدلی از نرم‌افزار است که همه‌ی خواسته‌های مشتری را به‌طور صحیح پیاده‌سازی کند و برای کاربران حسی دلپذیر ایجاد کند. طراحی نرم‌افزار باید از میان گزینه‌های فراوان طراحی که پیش روی خود دارد، بهترین‌ها را انتخاب کند تا به راهکاری برسد که به بهترین وجه با نیازهای طرف‌های ذی‌نفع پروژه همخوانی دارند. فرایند طراحی، گذاری است از یک نمای «تصویر بزرگ» از نرم‌افزار به نمایی ریزتر که جزئیات لازم برای پیاده‌سازی را فراهم می‌سازد. این فرایند با بذل توجه فراوان به معماری آغاز می‌گردد. زیر سیستم‌ها تعریف می‌شوند؛ سازوکارهای ارتباطی میان زیر سیستم‌ها برقرار می‌شوند؛ مؤلفه‌ها شناسایی می‌شوند و توصیف مشروحی از هر مؤلفه توسعه می‌یابد. به‌علاوه، واسط‌های خارجی، داخلی و کاربری نیز طراحی می‌شوند.

مفاهیم طراحی طی شصت سال اول مهندسی نرم‌افزار تکامل پیدا کردند. نرم‌افزار کامپیوتری، صرف نظر از فرایند مهندسی نرم‌افزار انتخاب شده، روش‌های طراحی به‌کار گرفته شده یا زبان برنامه‌نویسی مورد استفاده، صفاتی را باید از خود نشان دهند که آن‌ها را با همین مفاهیم می‌توان شناخت. در اصل، مفاهیم طراحی بر مواردی که به دنبال آمد، تأکید دارند: نیاز به انتزاع به‌عنوان سازوکاری برای ایجاد مؤلفه‌های قابل استفاده‌ی مجدد؛ اهمیت معماری به‌عنوان راهی برای درک بهتر ساختار کلی سیستم؛ مزایای مهندسی مبتنی بر الگو به‌عنوان تکنیکی بر طراحی نرم‌افزار با قابلیت‌های به اثبات رسیده؛ ارزش جداسازی دغدغه‌ها و پیمانه‌بندی اثربخش به‌عنوان شیوه‌ای برای قابل فهم‌تر کردن نرم‌افزار، بالا بردن قابلیت آزمایش نرم‌افزار و افزودن بر قابلیت نگهداری آن؛ پیامدهای پنهان‌سازی اطلاعات به‌عنوان سازوکاری برای کاهش انتشار اثرات جانبی در صورت رخ دادن خطا؛ تأثیر استقلال عملیاتی به‌عنوان ملاکی برای ساخت پیمانه‌های اثربخش؛ کاربرد پالایش به‌عنوان سازوکاری برای طراحی؛ در نظر گرفتن جنبه‌هایی که پیش‌نیاز خواسته‌های سیستم هستند؛ به‌کارگیری بازآرایی برای بهینه کردن طراحی به‌دست آمده؛ و اهمیت کلاس‌های شیءگرا و خصوصیات مرتبط با آن‌ها.

مدل طراحی شامل چهار عنصر متفاوت می‌شود. با توسعه یافتن هر کدام از این عناصر، دید کامل تری از طراحی تکامل پیدا می‌کند. عنصر معماری از اطلاعات به‌دست آمده از دامنه‌ی کاربرد، مدل خواسته‌ها و کاتالوگ‌های در دسترس برای الگوها و سبک‌ها استفاده می‌کند تا نمایش ساختاری کاملی از نرم‌افزار، زیر سیستم‌ها و مؤلفه‌های آن به‌دست آورد. عناصر طراحی در سطح مؤلفه‌ها، هر کدام از

پیمانه‌ها (مؤلفه‌های) تشکیل دهنده معماری را تعریف می‌کنند. سرانجام، عناصر طراحی در سطح استقرار، معماری، مؤلفه‌های آن و واسط‌های آن با پیکربندی فیزیکی‌ای که نرم‌افزار را در خود جای می‌دهد، تخصیص می‌دهند.

مسائل و نکاتی برای تعمق

۸-۱ آیا هنگامی که برنامه‌ای «می‌نویسید»، طراحی هم می‌کنید؟ چه چیز، طراحی نرم‌افزار را از کدنویسی متمایز می‌سازد؟

۸-۲ اگر طراحی نرم‌افزار، برنامه نیست (که نیست) پس چیست؟

۸-۳ کیفیت طراحی یک نرم‌افزار را چگونه ارزیابی می‌کنیم؟

۸-۴ مجموعه وظایف ارائه شده برای طراحی را بررسی کنید. در این مجموعه وظایف، کیفیت کجا ارزیابی می‌شود؟ چگونه این کار انجام می‌شود؟ صفات کیفیتی بحث شده در بخش ۸-۲-۱ چگونه قابل دستیابی اند؟

۸-۵ مثال‌هایی از سه انتزاع داده‌ای و سه انتزاع فرایندی، که در دستکاری آن‌ها قابل استفاده باشند، ارائه دهید.

۸-۶ معماری نرم‌افزار را به زبان ساده شرح دهید.

۸-۷ یک الگوی طراحی پیشنهاد کنید که در گروهی از چیزهای روزمره (مثلاً دستگاه‌های الکترونیکی، خودروها یا لوازم منزل) به آن برخورد کرده اید. این الگو را به اختصار شرح دهید.

۸-۸ جداسازی دغدغه‌ها را به زبان ساده شرح دهید. آیا موردی هست که راهبرد تقسیم و غلبه مناسب نباشد؟ چنین مورد چگونه می‌تواند بر پیمانه‌بندی تأثیر بگذارد؟

۸-۹ طراحی پیمانه‌ای چه هنگام باید به‌عنوان نرم‌افزاری یکپارچه پیاده‌سازی شود؟ چگونه می‌توان به آن رسید؟ آیا کارایی تنها توجیه برای پیاده‌سازی نرم‌افزار یکپارچه است؟

۸-۱۰ درباره رابطه‌ی میان مفهوم پنهان‌سازی اطلاعات به‌عنوان صفتی از پیمانه‌بندی اثربخش و مفهوم استقلال عملیاتی توضیح دهید.

۸-۱۱ مفاهیم اتصال و حمل‌پذیری نرم‌افزار چه ارتباطی با هم دارند؟ مثال‌هایی در تأیید بحث خود بیاورید.

۸-۱۲ برای یک یا چند برنامه‌ای که به دنبال می‌آیند، با به‌کارگیری «رویکرد پالایش مرحله‌ای» سه سطح انتزاع فرایندی متفاوت ایجاد کنید: (الف) برنامه‌ای برای نوشتن چک که با گرفتن وجه چک به‌صورت عددی، مقدار آن را به‌صورت حروفی روی چک چاپ می‌کند. (ب) حل ریشه‌های یک معادله متعالی به روش مبتنی بر تکرار، (پ) توسعه یک الگوریتم ساده برای زمان‌بندی وظایف روی یک سیستم عامل ساده.

۸-۱۳ نرم‌افزار مورد نیاز برای پیاده‌سازی قابلیت کامل ناوبری (با استفاده از GPS) در تلفن همراه را در نظر بگیرید. دو یا سه دغدغه پیش‌نیاز موجود را در نظر بگیرید. چگونه یکی از این دغدغه‌ها را به‌عنوان یک جنبه در نظر می‌گیرید؟ در این مورد بحث کنید.

۸-۱۴ آیا «بازآرایی» به معنی اصلاح کل طراحی به‌صورت تکراری است؟ اگر خیر، چه معنایی دارد؟

۸-۱۵ هر کدام از چهار عنصر مدل طراحی را شرح دهید.

فصل ۹

طراحی معماری

نگاهی گذرا

طراحی معماری چیست؟ طراحی معماری نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساخت یک سیستم کامپیوتری مورد نیازند. سبک معماری که سیستم به خود می‌گیرد، ساختار و خواص مؤلفه‌های تشکیل دهنده سیستم و روابط میان همه‌ی مؤلفه‌های معماری سیستم، در این طراحی معماری در نظر گرفته می‌شود.

چه کسی آن را انجام می‌دهد؟ گرچه مهندس نرم‌افزار قادر به طراحی داده‌ها و معماری است، در صورت بزرگ و پیچیده بودن سیستم، این وظیفه به افراد متخصص سپرده می‌شود. طراح بانک اطلاعاتی یا انبار داده‌ها، معماری داده‌های سیستم را ایجاد می‌کند. «معمار سیستم» سبک معماری مناسبی را با توجه به خواسته‌های به‌دست آمده در حین تحلیل خواسته‌های نرم‌افزار انتخاب می‌کند.

چرا اهمیت دارد؟ شما تلاش نمی‌کنید که بدون داشتن نقشه، ساختمان بسازید، درست است؟ ترسیم نقشه را هم با چیدمان لوله‌کشی خانه شروع نمی‌کنید. اول باید به تصویر بزرگ توجه کنید- خود خانه- و بعد به جزئیات بپردازید. این کاری است که در طراحی معماری انجام می‌شود- تصویر بزرگ را در اختیارتان قرار می‌دهد تا اطمینان کنید که کار درست پیش می‌رود.

مراحل کار کدام است؟ طراحی معماری با طراحی داده‌ها شروع می‌شود و سپس با به‌دست آوردن یک یا چند نمایش از ساختار معماری سیستم ادامه می‌یابد. سبک‌ها یا الگوهای معماری متفاوت تحلیل می‌شوند تا ساختاری به‌دست آید که بیشترین مناسبت را با صفات کیفیتی و خواسته‌های مشتری داشته باشد. پس از این که این سبک معماری انتخاب شد، جزئیات این معماری با استفاده از یک روش طراحی معماری تعیین خواهد شد.

محصول کار چیست؟ طی طراحی معماری، یک مدل معماری شامل معماری داده‌ها و ساختار برنامه ایجاد می‌شود. به‌علاوه، خواص مؤلفه‌ها و روابط (تعامل‌ها) نیز توصیف می‌شوند.

چگونه اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ در هر مرحله، محصولات کاری طراحی نرم‌افزار از نظر وضوح، صحت، کامل بودن و سازگاری با خواسته‌ها و با یکدیگر بازبینی می‌شوند.

طراحی به‌عنوان یک فرایند چند مرحله‌ای توصیف شده است که در آن، نمایش‌هایی از ساختار برنامه و داده‌ها، خصوصیات واسط و جزئیات روال‌ها از روی خواسته‌های اطلاعاتی ساخته می‌شوند. فریمن این توصیف را به‌صورت زیر بسط داده است [Fre80]:

> طراحی، فعالیتی است مرتبط با تصمیم‌گیری‌های عمده که غالباً ماهیتی ساختاری دارند. وجه اشتراک آن با برنامه‌نویسی در انتزاع نمایش اطلاعات و توالی‌های پردازشی است، ولی سطح جزئیات در حالت‌های حدی کاملاً متفاوت است. طراحی، نمایش‌هایی یکپارچه و خوش ترکیب از برنامه‌ها ارائه می‌دهد که بر روابط میان اجزا در سطحی بالاتر و عملیات‌های منطقی در سطوح پایین‌تر تمرکز دارند.

چنان که در فصل ۸ گفته شد، طراحی یک فعالیت اطلاعات- محور است. روش‌های طراحی نرم‌افزار با در نظر گرفتن هر کدام از سه دامنه مدل تحلیل به‌دست می‌آیند. دامنه‌های داده‌ای، عملیاتی و رفتاری به‌عنوان راهنمایی برای ایجاد طراحی نرم‌افزار عمل می‌کنند.

روش‌های مورد نیاز برای ایجاد «نمایش‌های یکپارچه و خوش ترکیب از لایه‌های معماری و داده‌های مدل طراحی در این فصل ارائه خواهند شد. هدف، فراهم آوردن روشی سیستماتیک برای به‌دست‌آوردن طراحی معماری- نقشه مقدماتی که نرم‌افزار بر اساس آن ساخته می‌شود- است.

۹-۱ معماری نرم‌افزار

شا و گارلان [Sha96] درکتاب برجسته‌ی خود در این باب، معماری نرم‌افزار را چنین توصیف می‌کنند:

> از آن زمان که اولین برنامه به چند پیمانه تقسیم شد، سیستم‌های نرم‌افزاری دارای معماری شدند و مسؤولیت تعامل میان پیمانه‌ها و خواص کلی سرهم‌بندی این پیمانه‌ها بر دوش برنامه نویسان قرار گرفت. به لحاظ تاریخی، معماری‌ها نقشی ناپیدا داشته اند- پیاده‌سازی تصادفی یا سیستم‌های قدیمی به جا مانده از گذشته. نرم‌افزارنویسان خوب، غالباً یک یا چند الگوی معماری را به‌عنوان راهبردهایی برای سازمان‌دهی به سیستم پذیرفته‌اند، ولی از این الگوها به شیوه‌ای غیر رسمی استفاده می‌کنند و راهی برای ابراز صریح آن‌ها در سیستم حاصل ندارند.

امروزه، معماری نرم‌افزار اثربخش، همراه با نمایش و طراحی صریح آن، به زمینه‌های غالب در مهندسی نرم‌افزار تبدیل شده‌اند.

۹-۱-۱ معماری چیست؟

هنگام[illegible] ماری یک ساختمان را در نظر می‌گیرید، صفات فراوانی به ذهن خطور می‌کند. در [illegible] طوح، به شکل کلی ساختار فیزیکی آن می‌اندیشید. ولی در واقعیت، معماری چیزی [illegible] است. معماری، روش انسجام‌بخشی اجزای ساختمان برای تشکیل کلیتی [illegible] فتن ساختمان در محیط و هماهنگی آن با سایر ساختمان‌های هم‌جوار [illegible] دن هدف توسط ساختمان و برآوردن نیازهای مالک آن. حس [illegible] ساختمان - و شیوه ترکیب بافت‌ها، رنگ‌ها و مواد برای ایجاد [illegible] ست. جزئیات ریز است- طراحی نور پردازی، نوع سقف، [illegible] ست همچنان ادامه دارد. و سرانجام این که، معماری،

شکل ۹-۴ ابعاد مدل [illegible]

> «معماری سیستم، یک چارچوب جامع است که شکل و ساختار آن سیستم را توصیف می‌کند- مؤلفه‌های آن و این که چگونه با هم مطابقت می‌کنند»
>
> جرولد گروچو

معماری، چیز دیگری نیز هست: «هزاران تصمیم‌گیری بزرگ و کوچک» [Tyr05]. برخی از این تصمیم‌ها در ابتدای طراحی گرفته می‌شوند و می‌توانند تأثیری عمیق بر کلیه کنش‌های دیگر طراحی بگذارند. سایر تصمیم‌گیری‌ها به تعویق می‌افتند تا این که قیدوبندهای بیش از حد محدود کننده‌ای که ممکن است به پیاده‌سازی ضعیف سبک معماری منجر شوند، از سر راه برداشته شده باشند.

ولی معماری نرم‌افزار چیست؟ باس، کلمنتس و کازمان [Bas03] این عبارت را چنین تعریف می‌کند.

> معماری نرم‌افزار برای یک برنامه یا سیستم برنامه نویسی، ساختار یا ساختارهایی از سیستم است که از مؤلفه‌های نرم‌افزار، خواص بیرونی قابل مشاهده‌ی این مؤلفه‌ها و روابط میان آن‌ها تشکیل می‌شود.

معماری، نرم‌افزار عملیاتی نیست بلکه نمایشی است که به کمک آن می‌توانید (۱) میزان اثربخشی طراحی را در برآورده ساختن خواسته‌های بیان شده تحلیل کنید، (۲) در مرحله‌ای که اعمال تغییرات طراحی هنوز آسان است، آلترناتیوهایی برای معماری در نظر بگیرید و (۳) خطرات مرتبط با ساخت نرم‌افزار را کاهش دهید.

در این تعریف، بر نقش «مؤلفه‌های نرم‌افزار» در هر نمایش معماری تأکید می‌شود. در حیطه‌ی طراحی معماری، مؤلفه نرم‌افزار می‌تواند چیزی به سادگی یک پیمانه از برنامه یا یک کلاس شیءگرا باشد و در عین حال می‌تواند چنان بسط یابد که شامل یک بانک اطلاعات یا «میان افزاری» باشد که پیکربندی شبکه‌ای از سرورها و کلاینت‌ها را میسر سازد. خواص مؤلفه‌ها همان ویژگی‌هایی هستند که برای درک چگونگی تعامل مؤلفه‌ها با یکدیگر ضرورت دارند. در سطح معماری، خواص درونی (از قبیل جزئیات الگوریتم) مشخص نمی‌شوند. روابط میان مؤلفه‌ها می‌تواند به سادگی فراخوانی روالی از یک پیمانه به پیمانه دیگر یا به پیچیدگی پروتکل دستیابی به یک بانک اطلاعاتی باشد.

برخی اعضای جامعه نرم‌افزاری (مثل [Kaz03]) بین واکنش‌های مرتبط با به‌دست آوردن معماری نرم‌افزار (که آن را «طراحی معماری» می‌خوانیم) و کنش‌های اعمال شده برای به‌دست آوردن طراحی نرم‌افزار، تفاوت قائل می‌شوند. یکی از کسانی که این ویرایش را مرور کرده است، چنین می‌نویسد:

> تفاوت متمایزی میان واژه‌های طراحی و معماری وجود دارد. طراحی نمونه‌ای از یک معماری است، مشابه با این که یک شیء نمونه‌ای از یک کلاس است. برای مثال، معماری کلاینت-سرور را در نظر بگیرید. من می‌توانم یک سیستم نرم‌افزار کلاینت- سرور را به شیوه‌های متفاوت، از روی این معماری و با استفاده از سکوی جاوا (Java EE) یا سکوی مایکروسافت (.NET framework) طراحی کنم. پس تنها یک معماری وجود دارد، ولی طراحی‌های فراوانی را براساس آن معماری می‌توان ایجاد کرد. از این رو، نمی‌توانید «معماری» و «طراحی» را با هم مخلوط کنید.

گرچه من نیز موافقم که یک طراحی نرم‌افزار، نمونه‌ای از یک معماری نرم‌افزار مشخص است، عناصر ساختارهایی که به‌عنوان بخشی از معماری تعریف می‌شوند، ریشه‌ی هر طراحی‌ای هستند که از آن متکامل می‌شود. طراحی با در نظر گرفتن معماری آغاز می‌شود.

در این کتاب، در طراحی معماری نرم‌افزار به دو سطح از هرم طراحی (شکل ۱-۸) خواهیم پرداخت- طراحی داده‌ها و طراحی معماری. در حیطه‌ی بحث قبلی، در طراحی داده‌ها می‌توانید مؤلفه داده‌ای معماری را در سیستم‌های مرسوم و تعاریف کلاس‌ها (شامل صفات و عملیات‌ها) در

نکته‌ی کلیدی

معماری نرم‌افزار باید ساختار یک سیستم و شیوه‌ی همکاری داده‌ها و مؤلفه‌های عملیاتی با یکدیگر را مدل‌سازی کند.

> «به شتاب با معماری ازدواج کنید و در فراغ خیال از آن دل بکنید»
>
> بری بوهم

مرجع وب

اشاره‌های مفیدی به بسیاری از سایت‌های معماری را در نشان زیر خواهید یافت:
www2.umassd.edu/SECenter/SAResources.html

سیستم‌های شیءگرا نمایش دهید. در معماری طراحی، آن چه که کانون توجه قرار می‌گیرد عبارت است از نمایش ساختار مؤلفه‌های نرم‌افزار، خواص آن‌ها و تعامل‌های میان این مؤلفه‌ها.

۹-۱-۲ اهمیت معماری در چیست؟

باس و همکاران در کتابی که به معماری نرم‌افزار اختصاص دارد [Bas03] سه دلیل مهم برای اهمیت معماری نرم‌افزار ذکر می‌کنند.

- نمایش‌های معماری نرم‌افزار، برقراری ارتباط بین طرف‌های ذی‌نفع در توسعه‌ی یک سیستم کامپیوتری را میسر می‌سازند.
- معماری، تصمیم‌گیری‌های زود هنگامی را که بر همه‌ی کارهای بعدی مهندسی نرم‌افزار تأثیر عمیق می‌گذارند، برجسته می‌سازد و با همان میزان از اهمیت، موفقیت نهایی سیستم را به‌عنوان یک موجودیت عملیاتی نمایان می‌سازد.
- معماری «یک مدل نسبتاً کوچک و قابل درک از چگونگی ساخت یافتگی سیستم و چگونگی همکاری مؤلفه‌های آن تشکیل می‌دهد.» [Bas03].

مدل طراحی معماری و الگوهای معماری موجود در آن، قابل انتقال هستند. یعنی، ژانرها (genres)، سبک‌ها و الگوهای معماری (بخش‌های ۲-۹ تا ۴-۹) را می‌توان در طراحی سایر سیستم‌ها به کاربرد و مجموعه‌ای از انتزاع‌ها را نمایش داد که مهندس نرم‌افزار را در توصیف معماری به شیوه‌های قابل پیش‌بینی یاری دهند.

۹-۱-۳ توصیف‌های معماری

هر کدام از ما تصویری ذهنی از معنا و مفهوم واژه‌ی *معماری* در ذهن دارد. ولی در واقعیت، معماری برای افراد متفاوت، معانی متفاوت دارد. می‌خواهیم بگوییم که طرف‌های ذی‌نفع متفاوت، معماری را از دیدگاه‌های متفاوتی می‌بینند که علت این تفاوت دیدگاه، تفاوت در مجموعه دغدغه‌های متفاوت است. این بدان معناست که یک توصیف معماری در واقع مجموعه‌ای از محصولات کاری است که نماهای متفاوتی از سیستم ارائه می‌دهد.

برای مثال، معمار یک ساختمان اداری بزرگ باید با انواع متفاوتی از طرف‌های ذی‌نفع کار کند. دغدغه‌ی اصلی مالک ساختمان (یکی از طرف‌های ذی‌نفع) حصول اطمینان از این بابت است که ساختمان به لحاظ زیبایی‌شناختی، نمایی دلپذیر داشته باشد و فضای اداری کافی و زیرساخت لازم فراهم شود تا از بابت سودهی خاطرش آسوده شود. بنابراین، معمار باید با استفاده از ساختمان که به دغدغه‌های مالک می‌پردازد، توصیفی ارائه دهد. دیدگاه‌های مورد استفاده، ترسیم‌هایی سه بعدی از ساختمان (برای ارائه جنبه‌های زیبایی‌شناختی) و مجموعه‌ای از نقشه‌های ساختمانی دوبعدی برای پرداختن به دغدغه‌های مالک در خصوص فضای اداری و زیرساخت‌ها خواهد بود.

ولی ساختمان اداری طرف‌های ذی‌نفع دیگری هم دارد که از آن جمله‌اند، اسکلت‌ساز. اسکلت‌ساز به اطلاعات معماری در خصوص تیرآهن‌ها و میل گردهای فولادی برای پشتیبانی ساختمان نیاز دارد؛ این اطلاعات عبارتند از نوع تیرآهن‌ها، ابعاد آن‌ها، شیوه اتصال میان آن‌ها، نوع مواد و بسیاری جزئیات دیگر. به این دغدغه‌ها در محصولات کاری متفاوتی پاسخ گفته می‌شود که نماهای متفاوتی از معماری ارائه می‌دهند. در نقشه‌های تخصصی اسکلت‌بندی فولادی ساختمان، تنها به یکی از دغدغه‌های اسکلت‌ساز پرداخته می‌شود.

> «معماری به مراتب مهم‌تر از آن است که تنها به یک نفر واگذار گردد هر قدر هم که آن فرد، قابل باشد.»
>
> اسکات امبلر

نکته‌ی کلیدی

مدل معماری نمایی کلی از سیستم فراهم می‌آورد به‌طوری که مهندس نرم‌افزار می‌تواند آن را به‌عنوان یک کل بررسی کند.

اندرز

تلاش شما باید بر نمایش‌های معماری‌ای متمرکز باشد که راهنمایی برای کلیه جنبه‌های دیگر طراحی باشند. زمانی را صرف مطالعه‌ی دقیق معماری کنید. اشتباهی در این جا تأثیرات منفی دراز مدت خواهد داشت.

توصیف معماری یک سیستم نرم‌افزاری نیز باید خصوصیاتی را از خود نشان دهد که مشابه با خصوصیات ذکر شده برای ساختمان اداری است. تایری و آکرمن [Tyr05] به این نکات چنین اشاره کرده‌اند: «سازندگان، به راهنمایی واضح و راسخ برای چگونگی پیشروی در کار طراحی نیاز دارند. مشتریان به درک روشنی از تغییرات محیطی که باید رخ دهند و ارائه تضمین در خصوص بر آورده شدن خواسته‌های خود از سوی معماری نیاز دارند. معماران دیگر نیز درکی واضح و برجسته از جنبه‌های کلیدی معماری می‌خواهند.» هر کدام از این «خواستن ها» در نمای متفاوتی منعکس می‌شود که با استفاده از دیدگاهی متفاوت نمایش داده می‌شود.

جامعه کامپیوتری IEEE استاندارد IEEE-Std-1471-2000 را تحت عنوان *کارهای توصیه شده برای توصیف معماری سیستم‌های نرم‌افزاری* [IEE00] پیشنهاد کرده است؛ این استاندارد اهداف زیر را دنبال می‌کند: (۱) ساخت چارچوبی مفهومی و واژگان مربوط به طراحی معماری نرم‌افزار، (۲) فراهم آوردن دستور العمل‌های مشروح برای نمایش یک توصیف معماری و (۳) تشویق به طراحی معماری منطقی.

در این استاندارد IEEE، *توصیف معماری* (AD) به‌عنوان «مجموعه‌ای از محصولات برای مستندسازی یک معماری» تعریف شده است. این توصیف، خود با استفاده از چند نما ارائه می‌شود که در آن هر *نما* «ارائه‌ای است از کل سیستم از دیدگاه یک مجموعه دغدغه‌های مرتبط (طرف ذی‌نفع).» هر *نما* مطابق با قواعد و قراردادهای تعریف شده در یک *دیدگاه* تعریف می‌شود- «مشخصه‌ای از قراردادها برای ساختن و به‌کارگیری یک نما [IEE00]. چند محصول کاری متفاوت در بسط دادن نماهای متفاوتی از معماری نرم‌افزار به‌کار می‌روند که در این فصل بحث خواهد شد.

۹-۱-۴ تصمیم‌گیری‌های معماری

هر کدام از نماهای بسط یافته به‌عنوان یک توصیف معماری به دغدغه معینی از یک طرف ذی‌نفع می‌پردازد. برای بسط دادن هر نما (و توصیف معماری به‌عنوان یک کلیت) معماری سیستم انواع آلترناتیوها را در نظر می‌گیرد و سرانجام درباره ویژگی‌های معماری خاصی که بیشترین همخوانی را با آن دغدغه دارد، تصمیم‌گیری می‌کند. بنابراین، خود تصمیم‌گیری‌های معماری را می‌توان یک نما از معماری در نظر گرفت. دلایلی که تصمیم‌گیری‌ها بر اساس آن‌ها انجام می‌شوند، دیدی از ساختار سیستم و همخوانی آن با دغدغه‌های طرف ذی‌نفع به‌دست می‌دهند.

شما به‌عنوان معمار سیستم می‌توانید از قالب پیشنهاد شده در کادر صفحه‌ی بعد برای مستندسازی هر تصمیم‌گیری استفاده کنید. به این ترتیب، توجیهی برای کار خود فراهم می‌آورید. سوابقی ایجاد می‌کنند که هنگام انجام اصلاحات روی طراحی می‌تواند مفید واقع شود.

۹-۲ ژانرهای معماری

گرچه اصول بنیادی طراحی معماری در تمامی انواع معماری کاربرد دارند، ژانر معماری غالباً رویکرد معماری مشخصی را در ساختاری که قرار است ساخته شود، دیکته می‌کند. در حیطه‌ی طراحی معماری، *ژانر* به معنای گروهی خاص در دامنه کلی نرم‌افزار است. در هر گروه، با چند زیرگروه مواجه می‌شوید. برای مثال، در ژانر ساختمان‌ها، سبک‌های عمومی خانه، آپارتمان، مجتمع مشاع، ساختمان اداری، ساختمان صنعتی، انبار و غیره را خواهید دید. در هر سبک عمومی، سبک‌های

اطلاعات

قالب برای توصیف تصمیم‌گیری‌های معماری

هر تصمیم‌گیری معماری عمده را می‌توان برای بازبینی طرف‌های ذی‌نفع آینده مستندسازی کرد تا توصیف معماری پیشنهاد شده را درک کنند. قالب ارائه شده در این کادر، نسخه‌ای خلاصه شده از قالب پیشنهادی توسط تایری و آکرمن [Tyr05] است.

مسأله طراحی: توصیف مسائل طراحی معماری که قرار است به آن‌ها پرداخته شود.

تحلیل (resolution): بیان رویکرد انتخابی برای پرداختن به مسأله طراحی.

گروه: تعیین گروه طراحی که مسأله طراحی و تحلیل به آن می‌پردازند (مثلاً طراحی داده‌ها، ساختار محتویات، ساختار مؤلفه‌ها، انسجام، ارائه).

فرض‌ها: ذکر هر فرضی که به اتخاذ تصمیم کمک می‌کند.

قیدوبندها: مشخص کردن هرگونه قیدوبند محیطی که به اتخاذ تصمیم کمک می‌کند (مثلاً استانداردهای فن آوری، الگوهای در دسترس، مسائل مرتبط با پروژه).

آلترناتیوها: توصیف مختصر سایر طراحی‌های معماری که در نظر گرفته می‌شوند و دلیل رد آن‌ها.

استدلال: بیان دلیل انتخاب یک رویکرد از میان سایر رویکردها.

دلالت: ذکر پیامدهای طراحی تصمیم‌گیری. تحلیل چگونه بر سایر مسائل طراحی معماری تأثیر خواهد گذاشت؟ آیا تحلیل، طراحی را به نحوی با قیدوبند مواجه می‌کند؟

تصمیم‌گیری‌های مرتبط: کدام تصمیم‌گیری‌های مستندسازی شده‌ی دیگر با این تصمیم‌گیری در ارتباط هستند؟

دغدغه‌های مرتبط: کدام خواسته‌ها با این تصمیم‌گیری در ارتباط هستند؟

محصولات کاری: ذکر این که تصمیم‌گیری در کجای توصیف معماری منعکس می‌شود.

یادداشت‌ها: ارجاع به هرگونه یادداشت‌های تیمی یا سایر مستنداتی که برای تصمیم‌گیری به‌کار گرفته شده است.

مشخص‌تری ممکن است کاربرد پیدا کنند (بخش ۳-۹). هر سبک دارای ساختاری است که با به‌کارگیری مجموعه‌ای از الگوهای قابل پیش‌بینی قابل توصیف است.

گرادی بوچ در کتاب خود با عنوان *راهنمای معماری نرم‌افزار* [Boo08]، ژانرهای معماری زیر را برای سیستم‌های کامپیوتری پیشنهاد می‌کند:

- **هوش مصنوعی**- سیستم‌هایی که شناخت انسانی، حرکت یا سایر فرایندهای آلی را شبیه‌سازی یا تکمیل می‌کنند.

نکته‌ی کلیدی

چند سبک معماری متفاوت ممکن است برای یک ژانر مشخص قابل استفاده باشد.

- **تجاری و غیر انتفاعی**- سیستم‌هایی که در راه اندازی شرکت‌های تجاری اهمیت بنیادی دارند.
- **ارتباطاتی**- سیستم‌هایی که زیرساخت لازم برای انتقال و مدیریت داده‌ها، برای متصل کردن کاربران آن داده‌ها یا برای ارائه داده‌ها در لبه‌ی یک زیرساخت فراهم می‌آورد.
- **پردازش محتویات**- سیستم‌هایی که برای ایجاد یا دستکاری کارهای متنی یا چند رسانه‌ای به‌کار می‌روند.
- **دستگاه‌ها**- سیستم‌هایی که با جهان فیزیکی تعامل دارند و نقاط سرویس‌دهی را برای افراد فراهم می‌آورند.
- **ورزش و تفریح**- سیستم‌هایی که رویدادهای عمومی را مدیریت می‌کنند یا گروه بزرگی از کاربران را سرگرم می‌کنند.
- **مالی**- سیستم‌هایی که زیرساخت لازم برای انتقال دادن و مدیریت پول و سایر موارد امنیتی را فراهم می‌سازند.
- **بازی‌ها**- سیستم‌هایی که تجربه سرگرمی برای افراد یا گروه فراهم می‌سازند.
- **دولتی**- سیستم‌هایی که هدایت و عملیات یک موجودیت سیاسی محلی، ایالتی، فدرال یا جهانی را پشتیبانی می‌کنند.
- **صنعتی**- سیستم‌هایی که فرایندهای فیزیکی را تقویت یا کنترل می‌کنند.
- **حقوقی**- سیستم‌هایی که موجودیت‌های حقوقی را پشتیبانی می‌کنند.
- **پزشکی**- سیستم‌هایی که به معاینه یا درمان کمک می‌کنند یا در پژوهش‌های پزشکی سهم دارند.
- **نظامی**- سیستم‌هایی برای مشاوره، ارتباطات، فرماندهی، کنترل و جاسوسی (C4I) و نیز سلاح‌های دفاعی و تهاجمی.
- **سیستم‌های عامل**- سیستم‌هایی که صرفاً روی سخت افزار نصب می‌شوند و سرویس‌های نرم‌افزاری پایه ارائه می‌دهند.
- **سکوها**- سیستم‌هایی که صرفاً روی سیستم عامل قرار می‌گیرند و سرویس‌های پیشرفته ارائه می‌دهند.
- **علمی**- سیستم‌هایی که برای پژوهش و کاربردهای علمی به‌کار می‌روند.
- **ابزارها**- سیستم‌هایی که در توسعه‌ی سایر سیستم‌ها به‌کار می‌روند.
- **حمل و نقل**- سیستم‌هایی که وسایل نقلیه‌ی آبی، زمینی، هوایی یا فضایی را کنترل می‌کنند.
- **برنامه‌های کمکی**- سیستم‌هایی که با سایر نرم‌افزارها تعامل می‌کنند تا نقاط سرویس‌دهی را فراهم آورند.

از دیدگاه طراحی معماری، هر ژانر نشان‌گر چالشی منحصر به فرد است. به‌عنوان مثال، معماری نرم‌افزار برای یک سیستم بازی را در نظر بگیرید. سیستم‌های بازی، که گاهی از آن‌ها به‌عنوان *برنامه‌های کاربردی تعاملی مجازی* نیز یاد می‌شود، به محاسبه الگوریتم‌های پرکار، گرافیک‌های پیچیده، منابع داده‌ای چند رسانه‌ای جریان‌دار (streaming)، تعامل زمان حقیقی از طریق ورودی‌های متداول و نامتداول و انواع دغدغه‌های تخصصی دیگر نیاز دارند.

«برنامه‌نویسی بدون در ذهن داشتن کل معماری یا طراحی مثل کاوش در غار با تنها یک چراغ قوه است: نمی‌دانید کجا بوده اید؛ نمی‌دانید به کجا می‌روید و درست نمی‌دانید کجا هستید.»

دنی توروپ

الکساندر فرانسواز [Fra03] برای برنامه‌های کاربردی تعاملی[1] یک معماری نرم‌افزار پیشنهاد می‌کند که در محیط بازی قابل استفاده است. او این معماری را چنین توصیف می‌کند:

> معماری نرم‌افزار برای برنامه‌های کاربردی تعاملی مجازی، یک مدل معماری نرم‌افزار جدید برای طراحی، تحلیل و پیاده‌سازی برنامه‌های کاربردی است که پردازش موازی، ناهمگام و توزیع شده‌ی جریان‌هایی از داده‌های کلی را بر عهده دارند. هدف آن، فراهم ساختن چارچوبی جهانی برای پیاده‌سازی توزیع شده‌ی الگوریتم‌ها و انسجام بخشی آسان به آن‌ها در سیستم‌های پیچیده است... مدل داده‌ای قابل بسط زیر بنایی و مدل پردازش موازی ناهمگام توزیع شده (مخزن مشترک و تبادل پیام) امکان دستکاری طبیعی و اثربخش روی جریان‌های داده‌ای کلی را با استفاده از کتابخانه‌های موجود و کدهای مشابه فراهم می‌سازد. پیمانه‌بندی سبک باعث تسهیل در توسعه‌ی کد توزیع شده، آزمون و استفاده مجدد شده علاوه بر آن، طراحی سیستم، انسجام، نگهداری و تکامل آن نیز با سرعت بیشتری قابل انجام است.

بحث مفصلی در این خصوص، خارج از حوصله‌ی این کتاب است. ولی دانستن این نکته حائز اهمیت است که به ژانر سیستم بازی می‌توان با یک سبک معماری پرداخت (بخش ۹-۳) که مشخصاً برای پرداختن به دغدغه‌های مربوط به سیستم‌های بازی طراحی شده است. در صورت علاقه بیشتر، [Fra03] را ببینید.

۹-۳ سبک‌های معماری

هنگامی که معمار از عبارت «عمارت ویلایی» برای توصیف یک خانه استفاده می‌کند، اکثر افراد آشنا با انواع خانه‌ها می‌توانند تصویری کلی از چنین خانه‌ای در ذهن داشته باشند و می‌دانند که این خانه چه شکل و ظاهری دارد. معمار از یک سبک معماری به‌عنوان یک سازوکار توصیفی استفاده کرده است تا این خانه را از سایر سبک‌ها (مثلاً آپارتمانی، حیاط دار، و ...) متمایز سازد. ولی مهمتر این که سبک معماری، قالبی برای ساخت فراهم می‌آورد. جزئیات بیشتر خانه باید اضافه شود، ابعاد نهایی آن تعیین گردد، یک سری ویژگی‌ها به سفارش صاحب منزل به آن‌ها اضافه شود، نوع مصالح ساختمانی تعیین شود، ولی سبک- عمارت ویلایی- معمار را در کارش یاری می‌دهد.

نرم‌افزاری که برای سیستم‌های کامپیوتری ساخته می‌شود نیز یکی از چند سبک معماری را از خود نشان می‌دهد. هر سبک، گروهی از سیستم‌ها را توصیف می‌کند که شامل موارد زیر می‌شود:

۱. مجموعه‌ای از مؤلفه‌ها (مثلاً بانک اطلاعاتی و پیمانه‌های محاسباتی) که وظیفه‌ای برای سیستم به انجام می‌رسانند؛
۲. مجموعه‌ای از کانکتورها که «برقراری ارتباط، هماهنگ‌سازی و همکاری» میان مؤلفه‌ها را امکان‌پذیر می‌سازند؛
۳. قیدوبندهایی که تعیین می‌کنند مؤلفه‌ها را چگونه می‌توان با هم منسجم ساخت و سیستم را ایجاد کرد؛
۴. مدل‌های معناشناختی که طراح به کمک آن‌ها می‌تواند خواص کلی سیستم را با تحلیل خواص بخش‌های سازنده آن درک کند.

«در پس ذهن هر هنرمندی یک الگو یا نوعی معماری وجود دارد.»
ج. ک. چسترتون

سبک معماری چیست؟

[1] فرانسواز برای این برنامه‌ها از اصطلاح immersipresence استفاده می‌کند.

اطلاعات

ساختارهای معماری کانونیک

معماری نرم‌افزار در اصل نشان‌گر ساختاری است که در آن مجموعه‌ای از موجودیت‌ها (که غالباً مؤلفه خوانده می‌شوند) توسط مجموعه‌ای از روابط (که غالباً کانکتور خوانده می‌شوند) به هم متصل می‌شوند. مؤلفه‌ها و کانکتورها هر دو با مجموعه‌ای از خواص همراه هستند که به طراح امکان می‌دهند تا میان انواع مؤلفه‌ها و کانکتورهای در دسترس، تمایز قائل شود. ولی در توصیف یک معماری از چه نوع ساختارهایی (مؤلفه‌ها، کانکتورها و خواص) می‌توان استفاده کرد؟ باس و کازمان [Bas03] پنج ساختار معماری کانونیک یا بنیادی پیشنهاد می‌کنند:

ساختار عملیاتی. مؤلفه‌ها نشان‌دهنده‌ی موجودیت‌های پردازشی یا عملیاتی هستند. کانکتورها، واسط‌هایی را نشان می‌دهند که توانایی «استفاده» یا «تحویل داده‌ها به» یک مؤلفه را فراهم می‌سازند. خواص، ماهیت مؤلفه‌ها و سازمان‌دهی واسط‌ها را توصیف می‌کنند.

ساختار پیاده‌سازی. «مؤلفه‌ها می‌توانند پکیج، کلاس، شیء، روال، تابع، متد و غیره باشند که همه‌ی آن‌ها به‌عنوان ابزاری برای بسته‌بندی قابلیت عملیاتی در سطوح گوناگونی از انتزاع به‌کار می‌روند [Bas03]. کانکتورها شامل توانایی تحویل و کنترل داده‌ها، به‌اشتراک گذاشتن داده‌ها، «استفاده» و «نمونه‌ای است از» می‌شوند. خواص، ویژگی‌های کیفیتی را که کانون توجه قرار می‌دهند (مثل قابلیت نگهداری، قابلیت استفاده دوباره)که هنگام پیاده‌سازی ساختار نتیجه می‌شوند.

ساختار همروند. مؤلفه‌ها «واحدهای همروند» را نشان می‌دهند که به‌عنوان نخ‌ها یا وظایف موازی سازمان‌دهی می‌شوند. «روابط [کانکتورها] عبارتند از «همگام می‌شود با ...»، «اولویت بیشتری از...» دارد، «داده‌ها را به... ارسال می‌کند»، «بدون... قابل اجرا نیست» و «با... قابل اجرا نیست». خواص مرتبط با این ساختار شامل اولویت، قابلیت قبضه‌کردن و زمان اجرا می‌شود [Bas03].

ساختار فیزیکی. این ساختار مشابه با مدل استقراری است که به‌عنوان بخشی از طراحی، توسعه می‌یابد. مؤلفه‌ها، سخت افزارهای فیزیکی‌ای هستند که نرم‌افزارها روی آن‌ها اسکان می‌یابند. کانکتورها، واسط‌های میان مؤلفه‌های سخت افزاری هستند، و خواص به چیزهایی از قبیل ظرفیت، پهنای باند، کارایی و سایر صفات می‌پردازند.

ساختار توسعه‌ای. این ساختار، مؤلفه‌ها، محصولات کاری و سایر منابع اطلاعاتی را تعریف می‌کند که با پیشرفت مهندسی نرم‌افزار به آن‌ها نیاز خواهیم داشت. کانکتورها، روابط میان محصولات کاری را نشان می‌دهند و خواص، ویژگی‌های هر آیتم را مشخص می‌سازند.

هر کدام از این ساختارها نمای متفاوتی از معماری نرم‌افزار را نشان می‌دهند و اطلاعاتی را در معرض دید قرار می‌دهند، و به این ترتیب، تیم نرم‌افزاری را در مدل‌سازی و ساخت یاری می‌دهند.

سبک معماری، تبدیلی است که بر طراحی کل یک سیستم اعمال می‌شود. هدف از آن، ایجاد ساختاری برای کلیه مؤلفه‌های سیستم است. در مواردی که یک معماری موجود قرار است دوباره مهندسی شود (فصل ۲۹)، اعمال سبک معماری، به تغییرات بنیادی در ساختار نرم‌افزار منجر خواهد شد که از آن جمله می‌توان به تعیین مجدد قابلیت‌های عملیاتی مؤلفه‌ها اشاره کرد [Bos00].

الگوی معماری، همانند سبک معماری، تبدیل طراحی معماری را باعث می‌شود. ولی الگو به چند طریق بنیادی با سبک‌ها تفاوت دارد: (۱) دامنه الگو از وسعت کمتری برخوردار است و به جای آن که کل معماری را کانون توجه قرار دهد تنها به یک جنبه از آن توجه دارد؛ (۲) الگو قانونی را در معماری وضع می‌کند که شرح می‌دهد نرم‌افزار چگونه باید جنبه‌ای از قابلیت عملیاتی خودش را در سطح زیرساختی سامان بخشد [Bos00]؛ (۳) در الگوهای معماری (بخش ۴-۹) تمایل بر این است که به مسائل رفتاری خاص در حیطه‌ی معماری پرداخته شود (مثلاً این که سیستم‌های بی‌درنگ چگونه به همگام‌سازی یا وقفه‌ها سامان می‌دهند). از الگوها می‌توان در ارتباط با یک سبک معماری استفاده کرد و به ساختار کلی سیستم شکل داد. در بخش ۱-۳-۹، به سبک‌ها و الگوهای معماری رایج در نرم‌افزار خواهیم پرداخت.

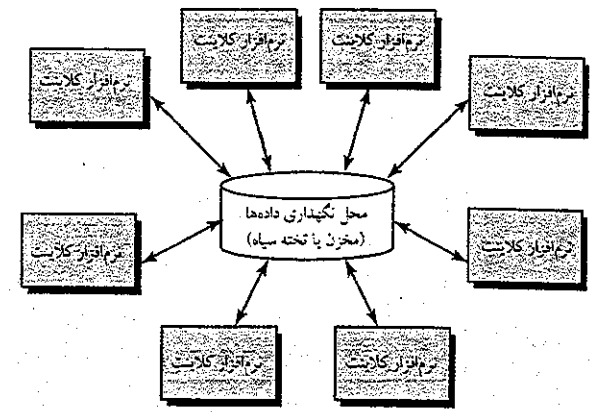
مرجع وب

سبک‌های معماری مبتنی بر صفات (ABAS) را می‌توان به‌عنوان قطعات سازنده برای معماری نرم‌افزار به کاربرد. اطلاعات مربوط به آن‌ها را در وب سایت زیر می‌یابید:
www.sei.edu/architecture/abas.html

۱-۳-۹ طبقه‌بندی مختصر سبک‌های معماری

گرچه طی شصت سال اخیر، میلیون‌ها سیستم کامپیوتری ایجاد شده است، اکثر آن‌ها را می‌توان در یکی از چند سبک معماری زیر خلاصه کرد:

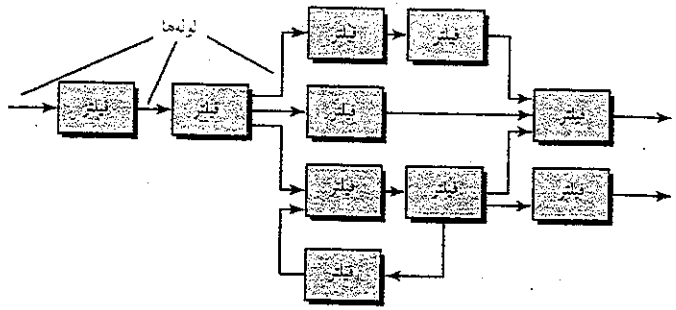
معماری‌های داده محور (Data-centered). محور این نوع معماری را یک انبار داده‌ها (فایل یا بانک اطلاعاتی) تشکیل می‌دهد که دستیابی به آن غالباً توسط مؤلفه‌های دیگری صورت می‌پذیرد که داده‌های موجود در این انبار را به هنگام، اضافه، حذف یا به طریقی دیگر، اصلاح می‌کنند. در شکل ۱-۹ نمونه‌ای از یک سبک معماری داده محور نشان داده شده است. نرم‌افزار کلاینت به یک مخزن مرکزی دستیابی دارد. در برخی موارد، این مخزن داده‌ها منفعل است به این معنی که نرم‌افزار کلاینت مستقل از هرگونه تغییراتی در داده‌ها یا کنش‌های سایر نرم‌افزارهای کلاینت، به این داده‌ها دستیابی دارد. با تغییر این رویکرد، مخزن به یک «تخته سیاه» تغییر ماهیت می‌دهد که هرگاه داده‌های مورد نظر کلاینت تغییر کند، کلاینت را از آن آگاه می‌سازد.



شکل ۱-۹ معماری داده محور.

استفاده از الگوها و سبک‌ها در رشته‌های معماری، امری فراگیر است.
مری شا و دیوید گارلان

معماری داده محور، یکپارچگی و انسجام را ارتقا می‌دهند [Bas03]. یعنی، مؤلفه‌های موجود را می‌توان تغییر داد و مؤلفه‌های کلاینت جدیدی را به معماری افزود، بی آن که نیازی باشد نگران کلاینت‌های دیگر باشیم (زیرا مؤلفه‌های کلاینت، مستقل از هم عمل می‌کنند). به‌علاوه، داده‌ها را می‌توان با استفاده از سازوکار تخته سیاه میان کلاینت‌ها تبادل کرد (یعنی مؤلفه‌ی تخته سیاه به هماهنگ کردن انتقال اطلاعات میان کلاینت‌ها کمک می‌کند). مؤلفه‌های کلاینت، فرایندها را مستقل از هم اجرا می‌کنند.



لوله‌ها و فیلترها

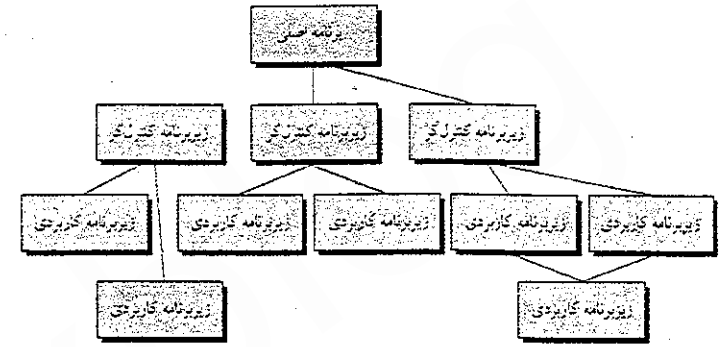
شکل ۲-۹ معماری جریان داده‌ها.

معماری‌های جریان داده‌ها (data-flow). این معماری هنگامی به‌کار برده می‌شود که قرار باشد داده‌های ورودی از طریق یک سری مؤلفه‌های محاسباتی و دستکاری، به داده‌های خروجی تبدیل شوند. الگوی لوله (pipe) و فیلتر (شکل ۲-۹) شامل مجموعه‌ای از مؤلفه‌ها، موسوم به فیلتر، می‌شود که توسط یک سری لوله به هم متصل می‌شوند؛ این لوله‌ها داده‌ها را از مؤلفه‌ای به مؤلفه‌ی بعدی ارسال می‌کنند. هر فیلتر مستقل از مؤلفه‌های فوقانی و زیرین خود عمل می‌کند، طوری طراحی می‌شود که داده‌های ورودی را در شکلی معین پذیرا باشد و داده‌های خروجی را با شکلی خاص تولید می‌کند (تا در اختیار فیلتر بعدی قرار گیرد). به هر حال، فیلتر نیازی ندارد که از عملکرد فیلترهای مجاور خود اطلاع داشته باشد.

اگر جریان داده‌ها تنها به یک خط از تبدیلات منجر شود، به آن، *ترتیب دسته‌ای* (batch sequential) گفته می‌شود. این ساختار، دسته‌ای از داده‌ها را می‌پذیرد سپس یک سری مؤلفه‌های ترتیبی (فیلترها) را به‌کار می‌گیرد تا آن دسته از داده‌ها را تبدیل کند.

معماری‌های فراخوانی و بازگشت. به کمک این سبک معماری می‌توانید به ساختاری برای برنامه دست پیدا کنید که اصلاح و تغییر دادن ابعاد آن نسبتاً آهسته باشد. در این گروه چند سبک فرعی نیز وجود دارد [Bas03]:

- *معماری‌های برنامه اصلی/ زیر برنامه.* در این ساختار کلاسیک برنامه، تابع به یک سلسله مراتب کنترلی تجزیه می‌شود که در آن یک برنامه «اصلی» چند مؤلفه از برنامه را فرا می‌خواند که هر یک به نوبه خود ممکن است مؤلفه‌های دیگری را فراخوانی کند. در شکل ۳-۹، یک معماری از این نوع نشان داده شده است.

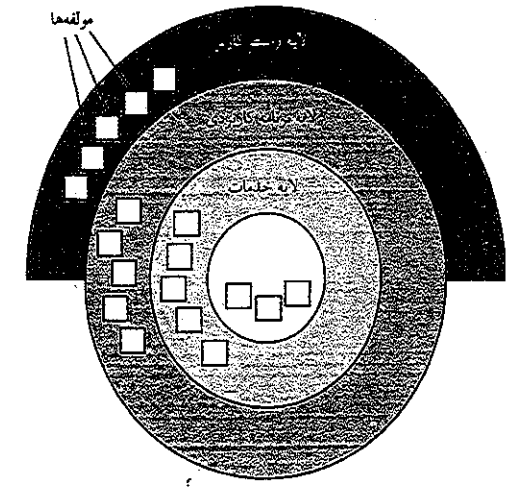


شکل ۳-۹ معماری برنامه اصلی / زیربرنامه.

- *معماری‌های فراخوانی روال‌های راه دور.* مؤلفه‌های معماری «برنامه اصلی / زیربرنامه» در میان چندین کامپیوتر روی یک شبکه توزیع می‌شوند.

معماری شیءگرا. مؤلفه‌های این سیستم، داده‌ها و عملیاتی را که باید برای دستکاری آنها اجرا شوند، کپسوله (encapsulate) می‌کنند. برقراری ارتباط و هماهنگ‌سازی میان مؤلفه‌ها از طریق مبادله پیام انجام می‌شود.

معماری لایه‌ای. ساختار اصلی یک معماری لایه‌ای در شکل ۴-۹ نشان داده شده است. تعدادی لایه‌های متفاوت تعریف می‌شود که هر یک عملیاتی را انجام می‌دهند و به‌طور تدریجی به دستورات ماشین نزدیک‌تر می‌شود. در لایه خارجی، مؤلفه‌ها به عملیات واسط کاربر سرویس می‌دهند. در لایه داخلی، مؤلفه‌ها، ارتباط با سیستم‌عامل را برقرار می‌کنند. لایه‌های میانی خدمات و عملکردهای اصلی نرم‌افزار را فراهم می‌آورند.



شکل ۴-۹ معماری لایه‌ای.

سبک‌های معماری فوق‌الذکر فقط زیرمجموعه کوچکی از سبک‌های در دسترس هستند[1]. هنگامی که در مهندسی خواسته‌ها، ویژگی‌ها و قیدوبندهای حاکم بر سیستم معلوم شد، می‌توان سبک معماری و/یا ترکیبی از الگوها را انتخاب نمود که بهترین مناسبت را با این ویژگی‌ها و قیدوبندها داشته باشند. در بسیاری موارد، بیش از یک الگو ممکن است مناسب باشد و می‌توان سبک‌های معماری متفاوتی را طراحی و ارزیابی کرد.

۹-۳-۲ الگوهای معماری

به موازاتی که مدل خواسته‌ها توسعه می‌یابد، متوجه خواهید شد که نرم‌افزار باید به چند مسأله عمده پاسخ‌گو باشد که کل برنامه کاربردی را در بر می‌گیرند. برای مثال، مدل خواسته‌ها برای هر برنامه کاربردی در زمینه‌ی تجارت الکترونیک با مسأله‌ای مواجه است که به دنبال خواهد آمد: *چگونه تعداد زیادی از کالاها را به آرایه‌ی گسترده‌ای از مشتریان ارائه دهیم و امکان خرید آنلاین کالاهای خود را برای این مشتریان فراهم سازیم؟*

مدل خواسته‌ها همچنین حیطه‌ای را تعریف می‌کند که این پرسش در آن باید پاسخ داده شود. برای مثال، یک شرکت تجارت الکترونیکی که تجهیزات گلف می‌فروشد، در حیطه‌ای متفاوت با شرکت دیگری کار می‌کند که تجهیزات صنعتی گران قیمت به شرکت‌های بزرگ یا میانه می‌فروشد. به‌علاوه، یک مجموعه محدودیت‌ها و قیدوبندها ممکن است برای شیوه‌ی رویارویی شما با مسأله تأثیر بگذارد.

الگوهای معماری به مسأله‌ای با کاربرد خاص در حیطه‌ای مشخص و تحت مجموعه‌ای از محدودیت‌ها و قیدوبندها می‌پردازند. این الگو، یک راهکار معماری پیشنهاد می‌کند که می‌تواند به‌عنوان مبنایی برای طراحی معماری عمل کند.

پیش‌تر در همین فصل متذکر شدیم که اکثر کاربردها در یک دامنه یا ژانر خاص می‌گنجند و یک یا چند سبک معماری ممکن است مناسب آن ژانر باشد. برای مثال، سبک معماری کلی مربوط به یک برنامه کاربردی ممکن است فراخوانی و بازگشت یا شیءگرا باشد. ولی در آن سبک، با مجموعه‌ای از مسائل مشترک مواجه خواهید شد که ممکن است به بهترین وجه با الگوهای معماری مشخص بتوان به آنها پرداخت. برخی از این مشکلات و بحث کامل‌تری درباره‌ی الگوهای معماری در فصل ۱۲ ارائه شده‌است.

۹-۳-۳ سازمان‌دهی و پالایش

از آنجا که فرایند طراحی، غالباً چند گزینه مختلف معماری را فرا روی شما قرار می‌دهد، ایجاد مجموعه‌ای از ملاک‌های طراحی که بتوان از آنها در ارزیابی طراحی معماری استفاده کرد، اهمیت دارد. پرسش‌های زیر [Bas03] دیدی از یک سبک معماری به‌دست می‌دهند.

کنترل. کنترل در داخل معماری چگونه مدیریت می‌شود؟ آیا سلسله مراتب کنترلی متمایزی وجود دارد و اگر چنین است، نقش مؤلفه‌ها در این سلسله مراتب کنترلی چیست؟ مؤلفه‌ها چگونه کنترل را در داخل سیستم منتقل می‌کنند؟ کنترل چگونه در میان مؤلفه‌ها به اشتراک گذاشته می‌شود؟ توپولوژی کنترل (یعنی شکل هندسی‌ای که کنترل خود می‌گیرد) چیست؟ آیا کنترل همگام شده است یا مؤلفه‌ها به‌صورت ناهمگام عمل می‌کنند؟

[1] برای بحث مفصلی درباره سبک‌ها و الگوهای معماری، [Bus07]، [Gor06]، [Bas03]، [Bos00] یا [Hof00] را ببینید.

> «شاید در زیر زمین باشد، بگذار بروم نگاهی به طبقه دوم بیندازم.»
> **موریس اشر**

سبک معماری به‌دست آمده را چگونه ارزیابی کنیم؟

SafeHome

انتخاب سبک معماری

صحنه: کابین جیمی، ابتدای مدل‌سازی طراحی.

نقش‌آفرینان: جیمی و اد- اعضای تیم نرم‌افزاری *SafeHome*.

گفتگو:

اد (اخم کرده است): ما قابلیت امنیتی را با UML مدل‌سازی کردیم- منظورم کلاس‌ها، روابط و از این جور چیزهاست. خلاصه، فکر می‌کنم معماری شیءگرا راه درست باشد.

جیمی: ولی...؟

اد: ولی... من نمی‌توانم تجسم کنم معماری شیءگرا چطور باید باشد. معماری فراخوانی و بازگشت را می‌فهمم، یک جورهایی همان سلسله مراتب سنتی فرایندهاست، ولی شیءگرا- نمی‌دانم، بی شکل به نظر می‌رسد.

جیمی (با لبخند): بی شکل؟

اد: بله- منظورم این است که نمی‌توانم یک ساختار واقعی برای آن تصور کنم. فقط کلاس‌های طراحی شناور در فضا.

جیمی: خب، این درست نیست. کلاس‌ها هم سلسله مراتب دارند- سلسله مراتبی را که برای شیء **FloorPlan** (شکل ۳-۸) داشتیم، یادت هست؟ معماری شیءگرا ترکیبی از آن ساختار و ارتباطات- همکاری‌ها- میان کلاس‌هاست. می‌توانیم این را با توصیف کامل صفات و عملیات‌ها، پیام‌هایی که ارسال می‌شوند و ساختار کلاس‌ها نشان دهیم.

اد: من یک ساعتی را صرف ترسیم معماری فراخوانی و بازگشت می‌کنم، بعداً بر می‌گردم و معماری شیءگرا را در نظر می‌گیرم.

جیمی: داگ مشکلی با این قضیه ندارد. گفته باید معماری‌های مختلفی را در نظر بگیریم. در ضمن، اصلاً دلیلی وجود ندارد که نتوان این دو معماری را با هم به کار برد.

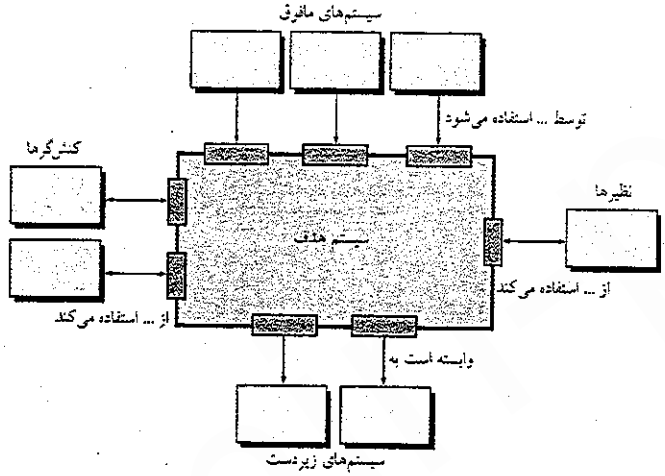
اد: خوب است، پس شروع می‌کنم.

داده‌ها. داده‌ها چگونه میان مؤلفه‌ها مبادله می‌شوند؟ آیا جریان داده‌ها پیوسته است یا اشیای داده‌ای، گاه و بی‌گاه به سیستم تحویل می‌شوند؟ شیوه‌ی انتقال داده‌ها چیست (یعنی آیا داده‌ها از مؤلفه‌ای به مؤلفه‌ی دیگر تحویل می‌شوند یا داده‌ها به‌طور سرتاسری در دسترس قرار دارند تا در میان مؤلفه‌های سیستم به اشتراک گذاشته شوند)؟ آیا مؤلفه‌های داده‌ها (مثلاً تخته سیاه یا مخزن) وجود دارند و اگر وجود دارند، نقش آن‌ها چیست؟ مؤلفه‌های عملیاتی چگونه با مؤلفه‌های داده‌ای تعامل دارند؟ آیا مؤلفه‌های داده‌ای فعال هستند یا منفعل (یعنی آیا مؤلفه داده‌ای با سایر مؤلفه‌های سیستم تعامل دارد)؟ تعامل داده‌ها و کنترل در سیستم به چه شیوه‌ای است؟

طراح با این پرسش‌ها می‌تواند کیفیت طراحی را مورد ارزیابی اولیه قرار دهد و بستری برای تحلیل مشروح‌تر معماری فراهم سازد.

۹-۴ طراحی معماری

به موازاتی که طراحی آغاز می‌شود، نرم‌افزاری که قرار است توسعه یابد، باید در حیطه‌ی کاری قرار داده شود- یعنی طراحی باید موجودیت‌های خارجی (سایر سیستم‌ها، دستگاه‌ها، افراد) را که نرم‌افزار با آن‌ها تعامل دارد و نیز ماهیت تعامل را تعریف کند. این اطلاعات را به‌طور کلی می‌توان از مدل خواسته‌ها و همه‌ی اطلاعات دیگری به‌دست آورد که طی مهندسی خواسته‌ها جمع آوری می‌شوند. هنگامی که حیطه‌ی کار مدل‌سازی شد و همه‌ی واسط‌های خارجی نرم‌افزار توصیف شدند، می‌توانید مجموعه‌ای از نمونه‌های اولیه معماری را شناسایی کنید. *نمونه اولیه*، انتزاعی (شبیه به کلاس) است که یک عنصر از رفتار سیستم را نمایش می‌دهد. این مجموعه نمونه‌های اولیه، مجموعه‌ای از انتزاع‌ها را فراهم می‌سازد که باید از نظر معماری مدل‌سازی شوند تا سیستم ساخته شود، ولی خود نمونه‌های اولیه جزئیات پیاده‌سازی کافی فراهم نمی‌آورند. بنابراین، طراح، با تعریف و پالایش مؤلفه‌هایی که هر نمونه اولیه را پیاده‌سازی می‌کنند، ساختار سیستم را مشخص می‌کند. این فرایند به تکرار چندان ادامه می‌یابد که یک ساختار معماری کامل به‌دست آید. در بخش‌هایی که به دنبال خواهد آمد، هر کدام از این وظایف معماری را با جزئیات بیشتر بررسی خواهیم کرد.



شکل ۹-۵ نمودار حیطه‌ی معماری.

۹-۴-۱ نمایش سیستم در حیطه‌ی کاری

در سطح طراحی معماری، معماری نرم‌افزار برای مدل‌سازی شیوه‌ی تعامل نرم‌افزار با موجودیت‌های خارج از مرزهای خود از نمودار حیطه‌ی معماری (ACD) استفاده می‌کند. ساختار کلی نمودار حیطه‌ی معماری در شکل ۹-۵ نشان داده شده است.

همان‌طور که این شکل نشان می‌دهد، سیستم‌هایی که با سیستم *هدف* (سیستمی که باید برای آن یک طراحی معماری توسعه داده شود) همکاری متقابل دارند، به‌صورت‌های زیر نشان داده می‌شوند:

> پزشک می‌تواند اشتباهات خود را دفن کند، ولی معمار تنها می‌تواند به مشتری خود بگوید که تاک و پیچک بکارد.»
>
> **فرانک لوید رایت**

نکته‌ی کلیدی

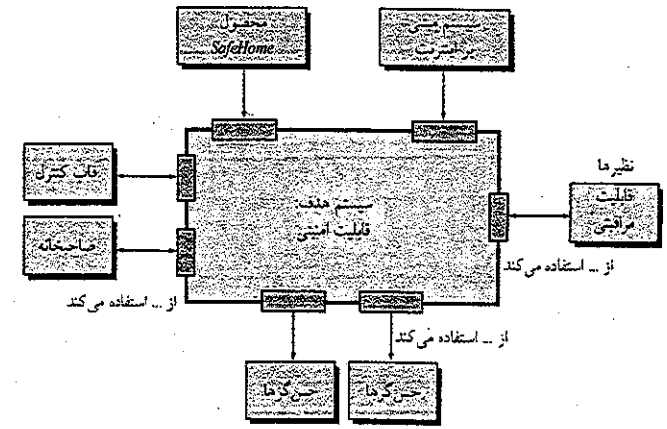
حیطه‌ی معماری چگونگی تعامل نرم‌افزار با موجودیت‌های خارج از مرزهای آن را نمایش می‌دهد.

- *سیستم‌های مافوق (superordinate systems)*- سیستم‌هایی که از سیستم هدف به‌عنوان بخشی از الگوی پردازش سطح بالاتر استفاده می‌کنند.
- *سیستم‌های زیردست (subordinate systems)*- سیستم‌هایی که توسط سیستم هدف به‌کار گرفته می‌شوند و داده‌ها یا پردازش مورد نیاز برای کامل کردن قابلیت‌های عملیاتی سیستم هدف را فراهم می‌سازند.
- *سیستم‌های سطح نظیر (peer-level systems)*- سیستم‌هایی که در سطح نظیر به نظیر تعامل دارند (یعنی اطلاعات توسط سیستم هدف و نظیرها، تولید یا مصرف می‌شود).
- *کنش‌گرها (actors)*- مرجودیت‌ها (افراد، دستگاه‌هایی) که با تولید یا مصرف اطلاعات مورد نیاز برای پردازش‌های ضروری، با سیستم هدف تعامل دارند.

هر کدام از این موجودیت‌های خارجی از طریق یک واسط، با سیستم هدف ارتباط برقرار می‌کنند (واسط‌ها با مستطیل‌های هاشور خورده‌ی کوچک مشخص شده‌اند).

برای نشان دادن کاربرد ACD، قابلیت امنیتی منزل در محصول *SafeHome* را در نظر بگیرید. کنترل‌گر جامع در محصول *SafeHome* و سیستم مبتنی بر اینترنت، هر دو زیردست قابلیت امنیتی به شمار می‌روند و از این رو، در شکل ۶-۹ در بالای سیستم هدف قرار دارند. قابلیت عملیاتی پایش منزل یک سیستم *نظیر* بوده در نسخه‌های بعدی محصول از قابلیت امنیتی منزل استفاده می‌کند (توسط آن استفاده می‌شود). صاحبخانه و قاب‌های کنترل، کنش گرهایی هستند که هم تولید کننده و هم مصرف کننده‌ی اطلاعاتی هستند که توسط نرم‌افزار امنیتی استفاده/ تولید می‌شوند. سرانجام، حس‌گرها توسط نرم‌افزار امنیتی استفاده می‌شوند و به‌عنوان زیردست آن نشان داده می‌شوند.

به‌عنوان بخشی از طراحی معماری، جزئیات هر واسط نشان داده شده در شکل ۶-۹ باید مشخص گردد. همه‌ی داده‌هایی که به درون و بیرون سیستم هدف جریان می‌یابند باید در این مرحله شناسایی شوند.



شکل ۶-۹ نمودار حیطه‌ی معماری برای قابلیت امنیتی منزل در محصول *SafeHome*.

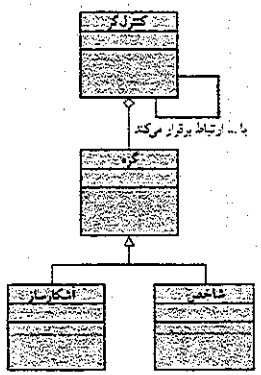
سیستم‌ها چگونه با یکدیگر همکاری می‌کنند؟

۹-۴-۲ تعریف نمونه‌های اولیه

نمونه اولیه، کلاس یا الگویی است که یک انتزاع هسته‌ای را نشان می‌دهد که در طراحی معماری برای سیستم هدف، اهمیت حیاتی دارد. به‌طور کلی، حتی برای طراحی سیستم‌های نسبتاً پیچیده به مجموعه نسبتاً کوچکی از نمونه‌های اولیه نیاز است. معماری سیستم هدف، از این نمونه‌های اولیه تشکیل می‌شود که عناصر پایدار معماری را نشان می‌دهند، ولی ممکن است بر اساس رفتار سیستم، نمونه‌های بعدی به شیوه‌های گوناگون از روی آن‌ها ساخته شود.

نکته‌ی کلیدی
نمونه‌های اولیه، قطعات سازنده انتزاعی در یک طراحی معماری‌اند.

در بسیاری موارد، نمونه‌های اولیه را می‌توان با بررسی کلاس‌های تحلیل تعریف شده به‌عنوان بخشی از مدل خواسته‌ها به‌دست آورد. با ادامه‌ی بحث قابلیت امنیتی منزل، می‌توانید نمونه‌های اولیه زیر را تعریف کنید:



شکل ۷-۹ روابط UML برای نمونه‌های اولیه قابلیت امنیتی منزل.

- **گره (node).** مجموعه‌ای یکپارچه از عناصر ورودی و خروجی قابلیت امنیتی منزل را نشان می‌دهد. برای مثال، یک گره ممکن است از (۱) حس‌گرهای گوناگون و (۲) انواع شاخص‌های آژیر (خروجی) تشکیل شده باشند.
- **آشکارساز (detector).** انتزاعی که شامل همه‌ی تجهیزات حس‌گر می‌شود و اطلاعات را به درون سیستم هدف تغذیه می‌کند.
- **شاخص (indicator).** انتزاعی که همه‌ی سازوکارها (مانند آژیر خطر، نور فلاش، زنگ اخبار) را نشان می‌دهد تا مشخص کند که شرایط هشدار رخ داده است.
- **کنترل‌گر (controller).** انتزاعی که نشان‌دهنده‌ی سازوکاری برای مسلح کردن یا غیر مسلح کردن یک گره است. اگر کنترل‌گر روی شبکه مستقر باشد، این توان را دارد که با دیگران ارتباط برقرار کند.

هر کدام از نمونه‌های اولیه با استفاده از نمادگذاری UML نمایش داده می‌شوند (شکل ۷-۹). به‌خاطر بسپارید که نمونه‌های اولیه، اساس و مبنایی برای معماری تشکیل می‌دهند، ولی انتزاع‌هایی هستند که باید باز هم با پیشرفت طراحی معماری پالایش شوند. برای مثال، **Detector** را شاید بتوان به سلسله مراتبی از حس‌گرها پالایش کرد.

۹-۴-۳ پالایش معماری به مؤلفه‌ها

با پالایش معماری نرم‌افزار به مؤلفه‌های آن، ساختار سیستم نمایان خواهد شد. ولی این مؤلفه‌ها چگونه انتخاب می‌شوند؟ به منظور پاسخ دادن به این پرسش، با کلاس‌هایی شروع می‌کنیم که به‌عنوان بخشی از مدل خواسته‌ها توصیف شد.[1] این کلاس‌های تحلیل، نشان‌گر موجودیت‌هایی در دامنه‌ی کاربرد (تجارت) هستند که باید در معماری نرم‌افزار به آن‌ها پرداخت. از این رو، دامنه کاربرد، یک منبع برای به‌دست آوردن و پالایش مؤلفه‌ها به شمار می‌رود. منبع دیگر، دامنه‌ی زیرساختی است. معماری باید مؤلفه‌های زیرساختی فراوانی را در بر گیرد که مؤلفه‌های کاربردی را فعال می‌سازند، ولی هیچ ارتباط تجاری با دامنه کاربردی ندارند. برای مثال، مؤلفه‌های مدیریت حافظه، مؤلفه‌های مخابراتی، مؤلفه‌های بانک اطلاعاتی و مؤلفه‌های مدیریت وظایف غالباً در معماری نرم‌افزار گنجانده می‌شوند.

ساختار یک سیستم نرم‌افزاری، سوم‌شناختی را مشخص می‌سازد که کد برنامه در آن زاده می‌شود، رشد می‌کند و می‌میرد. نومی که خوب طراحی شده باشد، امکان تکامل موفق را برای همه‌ی مؤلفه‌های مورد نیاز در یک سیستم نرم‌افزاری فراهم می‌سازد.
آر. پاتیس

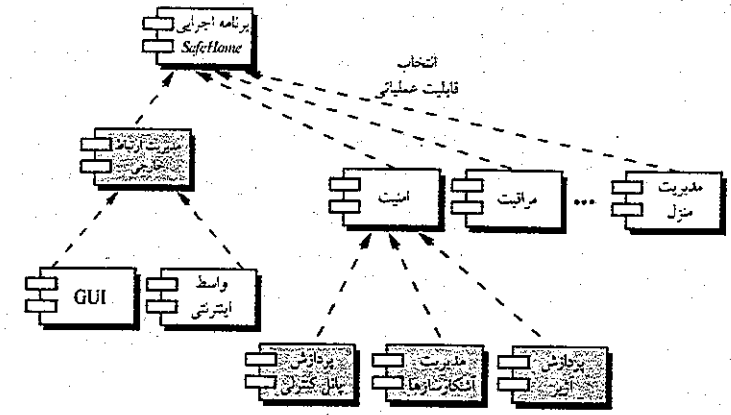
[1] اگر یک رویکرد ستی (غیر شیءگرا) انتخاب شود، مؤلفه‌ها از مدل جریان داده‌ها به‌دست می‌آیند. درباره این رویکرد به اختصار در بخش ۶-۹ بحث خواهیم کرد.

واسط‌های تصویر شده در نمودار حیطه‌ی معماری (بخش ۱-۴-۹) نشان‌گر یک یا چند مؤلفه‌ی تخصص‌یافته‌ترند که داده‌های جریان یافته از میان واسط هستند. در برخی موارد (مانند واسط گرافیکی کاربر)، یک معماری زیر سیستم کامل، با مؤلفه‌های بسیار زیاد باید طراحی شود.

با ادامه‌ی مثال قابلیت امنیتی منزل در محصول *SafeHome* می‌توانید مجموعه‌ای از مؤلفه‌های سطح بالا را تعریف کنید که به قابلیت عملیاتی زیر بپردازد:

- **مدیریت ارتباطات خارجی**- ارتباطات قابلیت امنیتی را با موجودیت‌های خارجی از قبیل سیستم‌های اینترنتی دیگر و سیستم هشدار خارجی هماهنگ می‌کند.
- **پردازش پانل کنترلی**- همه‌ی عملکردهای پانل کنترلی را مدیریت می‌کند.
- **مدیریت آشکارسازها**- دستیابی به همه‌ی آشکارسازهای متصل به سیستم را هماهنگ می‌کند.
- **پردازش هشدارها**- همه‌ی شرایط هشدار را وارسی و روی آن‌ها عمل می‌کند.

به هر کدام از این مؤلفه‌های سطح بالا باید به‌طور تکراری جزئیات بیشتری افزود و سپس در کل معماری *SafeHome* مستقر نمود. کلاس‌های طراحی (با صفت‌ها و عملیات‌های مناسب) برای هر کدام تعریف خواهند شد. ولی ذکر این نکته حائز اهمیت است که جزئیات طراحی همه‌ی صفات و عملیات‌ها، تا طراحی در سطح مؤلفه‌ها مشخص نخواهد شد (فصل ۱۰).



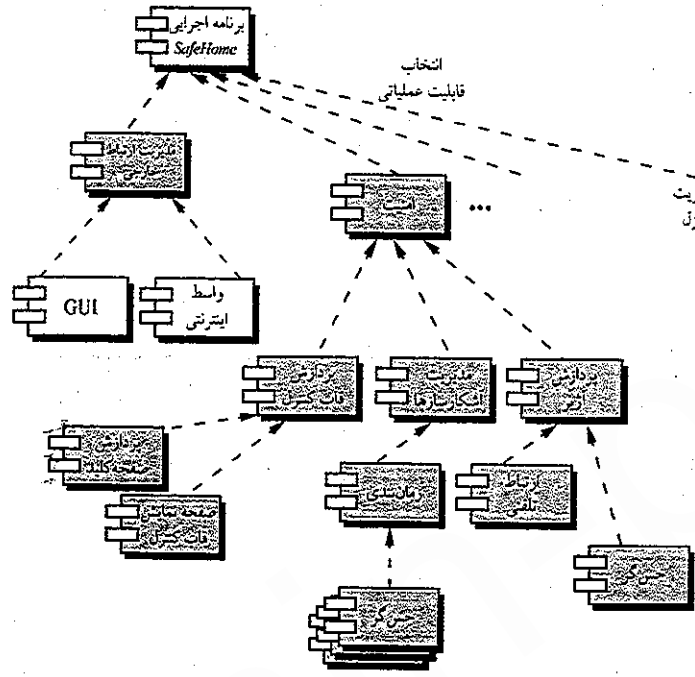
شکل ۸-۹ کل ساختار معماری برای *SafeHome* با مؤلفه‌های سطح بالا.

کل ساختار معماری (که به‌صورت نمودار مؤلفه‌های UML نمایش داده می‌شود) در شکل ۸-۹ نشان داده شده است. تراکنش‌ها به وسیله‌ی *مدیریت ارتباطات خارجی* و به موازات حرکت آن‌ها از مؤلفه‌هایی به‌دست می‌آیند که *SafeHome* GUI و واسط اینترنتی را پردازش می‌کنند. این اطلاعات توسط یک مؤلفه‌ی اجرایی *SafeHome* مدیریت می‌شود که قابلیت عملیاتی مناسب محصول (در این مورد، امنیت منزل) را انتخاب می‌کند. مؤلفه‌ی *پردازش پانل کنترلی* با صاحبخانه تعامل می‌کند تا قابلیت امنیتی منزل را مسلح یا غیر مسلح کند. مؤلفه *مدیریت آشکارسازها*، به حس‌گرها سر می‌زند تا شرایط هشدار را رؤیت کند و در صورت لزوم، مؤلفه‌ی *پردازش هشدارها* خروجی را تولید می‌کند.

۹-۴-۴ توصیف ساخت نمونه‌ای از سیستم

طراحی معماری که تا این نقطه مدل‌سازی شده است، هنوز از سطح نسبتاً بالایی برخوردار است. حیطه‌ی سیستم نشان داده شده است، نمونه‌های اولیه‌ای که انتزاع‌های مهم را در دامنه‌ی مسأله نشان می‌دهند، تعریف شده‌اند، ساختار سیستم مشخص شده است و مؤلفه‌های اصلی نرم‌افزار شناسایی شده‌اند. ولی، هنوز به پالایش بیشتر (به‌خاطر دارید که همه‌ی طراحی، تکراری است) نیاز است.

برای دستیابی به این منظور، نمونه‌ای واقعی از معماری توسعه می‌یابد. هدف، به‌کار گرفتن معماری در یک مسأله‌ی خاص است تا نشان دهیم که ساختار و مؤلفه‌ها مناسب هستند.



شکل ۹-۹ نمونه‌ای برگرفته از قابلیت امنیتی منزل با جزئیات افزوده شده به مولفه‌ها.

در شکل ۹-۹، ساخت نمونه‌ای از معماری *SafeHome* برای سیستم امنیت منزل آمده است. به مؤلفه‌های نشان داده شده در شکل ۸-۹ جزئیات بیشتری افزوده می‌شود تا جزئیات اضافی نشان داده شود. برای مثال، مؤلفه‌ی *مدیریت آشکارسازها* با مؤلفه‌ی زیرساختی *زمان‌بندی*، تعامل می‌کند، که تمام اشیای حس‌گر موجود در سیستم امنیت منزل را چک می‌کند. به هر کدام از مؤلفه‌های نشان‌داده‌شده در ۸-۹ جزئیاتی افزوده می‌شود.

۹-۵ تعیین طراحی‌های معماری متفاوت

کلمتس و همکاران وی [Cle03] در کتابی راجع به ارزیابی معماری‌های نرم‌افزار چنین اظهار کرده‌اند:

ابزارهای نرم‌افزاری

طراحی معماری

هدف: ابزارهای طراحی معماری، کل ساختار نرم‌افزار را با نشان دادن واسط مؤلفه‌ها، وابستگی‌ها و روابط، و تعامل‌ها مدل‌سازی می‌کنند.

مکانیک: مکانیک این ابزارها متفاوت است. در اکثر موارد، قابلیت طراحی معماری بخشی از قابلیت عملیاتی فراهم شده توسط ابزارهای خودکار برای مدل‌سازی طراحی و تحلیل است.

ابزارهای نمونه

Adalon، که توسط شرکت Synhtis (**www.synthis.com**) توسعه یافته است، یک ابزار طراحی تخصص یافته برای طراحی و ساخت معماری مؤلفه‌های مبتنی بر وب است.

ObjectiF، که توسط microTOOL GmbH (**www.microtool.de/objectiF/en/**) توسعه یافته است، یک ابزار طراحی مبتنی بر UML است که به معماری‌های مناسب برای مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها (مثل Coldfusion، J2EE Fusebox) مربوط می‌شود (فصل ۲۹).

Rational Rose، که توسط Rational (**www-360.ibm.com/software/rational/**) توسعه یافته است، یک ابزار طراحی مبتنی بر UML است که همه‌ی جنبه‌های طراحی معماری را پشتیبانی می‌کند.

بی پرده بگوییم، معماری قماری است بر سر موفقیت یک سیستم. به جای این که صبر کنید تا سیستم تقریباً کامل شود و هنوز ندانید که آیا خواسته‌ها را برآورده می‌کند یا خیر، آیا بهتر نیست که از قبل بدانید روی کارت برنده شرط بسته‌اید؟ اگر سیستمی می‌خرید یا برای توسعه‌ی آن پولی می‌پردازید، دوست ندارید تضمینی داشته باشید که در مسیر درست حرکت کند؟ اگر خودتان معمار هستید، دوست ندارید راه خوبی برای اعتبارسنجی تجربه و بصیرت خود داشته باشید به‌طوری که بتوانید شب با خیال آسوده بخوابید در حالی که می‌دانید طراحی را به خوبی ایجاد کرده‌اید.

در واقع پاسخ به این پرسش‌ها ارزش‌مند است. طراحی به چند معماری متفاوت منجر می‌گردد که با ارزیابی هر کدام می‌توان تعیین کرد کدام یک برای مسأله‌ای که باید حل شود، بیش از دیگران مناسب است. در بخش‌هایی که به دنبال خواهد آمد، دو رویکرد متفاوت برای ارزیابی طراحی‌های معماری متفاوت ارائه خواهد شد. روش نخست، از روش تکراری برای ارزیابی طراحی‌های معماری متفاوت یا توازن‌ها (trade-offs) استفاده خواهد کرد. در رویکرد دوم از یک تکنیک شبه کمّی برای ارزیابی کیفیت طراحی استفاده می‌شود.

۹-۵-۱ روش تحلیل توازن‌های معماری

مؤسسه مهندسی نرم‌افزار (SEI) یک روش تحلیل توازن‌های معماری توسعه داده است [Kaz98] که برای معماری‌های نرم‌افزار، یک فرایند ارزیابی تکراری تدارک می‌بیند. فعالیت‌های تحلیل طراحی که در زیر ذکر می‌شوند، به‌صورت تکراری به اجرا درمی‌آیند:

۱. *جمع‌آوری سناریوها.* مجموعه‌ای از use caseها (فصل‌های ۵ و ۶) برای نشان دادن سیستم از دیدگاه کاربر تهیه می‌شود.

مرجع وب

اطلاعاتی عمیق در خصوص ATAM را در سایت زیر می‌توان یافت:

www.sei.cmu.edu/activities/architecture/ata_method.html

۲. *روشن کردن خواسته‌ها*، قیدوبندها و توصیف محیط. این اطلاعات به‌عنوان بخشی از مهندسی خواسته‌ها موردنیاز است و بدین منظور مورد استفاده قرار می‌گیرند که اطمینان حاصل شود که مشتری، کاربر و کلیه افراد ذی‌نفع در نظر گرفته شده‌اند.

۳. *توصیف سبک‌ها / الگوهای معماری که برای پرداختن به سناریوها و خواسته‌ها انتخاب شده‌اند.* سبک(ها) باید با استفاده از نماهای معماری زیر توصیف شوند:

- *نمای پیمانه (module view).* برای تحلیل تخصیص کارها به مؤلفه‌ها و میزان پنهان کردن اطلاعات.
- *نمای پردازش (process view).* برای تحلیل کارایی سیستم.
- *نمای جریان داده‌ها (data-flow view).* برای تحلیل میزان برآورده شدن خواسته‌های عملیاتی در معماری.

۴. *ارزیابی صفات کیفیتی با در نظر گرفتن هر صفت به‌طور مجزا.* تعداد صفات کیفیتی که برای تحلیل انتخاب شد، تابعی از زمان لازم برای بازبینی و میزان ارتباط صفات کیفیتی با سیستم موردنظر است. صفات کیفیتی برای ارزیابی طراحی معماری عبارتنداز: قابلیت اطمینان، کارآیی، امنیت، قابلیت نگهداری، انعطاف‌پذیری، آزمون‌پذیری، قابلیت حمل، قابلیت استفاده مجدد و قابلیت کار متقابل.

۵. *شناسایی حساسیت صفات کیفیتی در مقابل صفات معماری گوناگون برای یک سبک معماری مشخص.* برای این منظور می‌توان تغییرات کوچکی در معماری اعمال کرد و تعیین کرد که یک صفت کیفیتی، مثلاً کارآیی، تا چه حد نسبت به این تغییر حساسیت دارد. هر صفتی که از این تغییر تأثیر زیادی بپذیرد، به‌عنوان نقطه حساسیت در نظر گرفته می‌شود.

۶. *نقد معماری‌های کاندیدا (که در مرحله ۳ توسعه داده شده‌اند) با استفاده از تحلیل حساسیت که در مرحله ۵ اجرا می‌شود.* SEI این روش را به شیوه زیر توصیف می‌کند [Kaz98]:

هنگامی که نقاط حساسیت معماری تعیین شد، یافتن نقاط توازن صرفاً عبارت از شناسایی عناصری از معماری است که چند صفت نسبت به آنها حساسیت دارند. برای مثال، کارایی یک معماری کلاینت/سرور ممکن است نسبت به تعداد سرورها بسیار حساس باشد (در یک گستره معین، کارایی با افزایش تعداد سرورها افزایش می‌یابد). قابلیت دستیابی چنین معماری‌ای نیز ممکن است مستقیماً با تعداد سرورها تغییر کند. ولی، امنیت سیستم ممکن است با تعداد سرورها به‌طور معکوس تغییر کند (زیرا تعداد نقاط حمله‌ی بالقوه سیستم بیشتر می‌شود). در این صورت، تعداد سرورها، یک نقطه توازن برای این معماری به شمار می‌رود. این یکی از چند عنصر بالقوه‌ای است که در آنجا توازن معماری چه به‌صورت خودآگاه و چه ناخودآگاه، صورت می‌پذیرد.

شش مرحله‌ای که در بالا ذکر شد، اولین دور تکرار ATAM را تشکیل می‌دهد. براساس نتایج مراحل ۵ و ۶ ممکن است چند مورد از معماری‌ها حذف شود و یک یا چند معماری باقیمانده را با جزئیات بیشتری اصلاح کرد و به نمایش درآورد و سپس مراحل ATAM را دوباره به اجرا گذاشت.[1]

[1] *روش تحلیل معماری نرم‌افزار* (SAAM) روشی مشابه با ATAM است. بد نیست خوانندگان علاقه‌مند به تحلیل معماری به آن نیز نگاهی داشته باشند. از نشانی www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html می‌توانید مقاله‌ای درباره SAAM دانلود کنید.

SafeHome

ارزیابی معماری‌ها

صحنه: دفتر داگ میلر؛ پیشرفت مدل‌سازی طراحی معماری.

نقش‌آفرینان: وینود، جیمی و اد- اعضای تیم مهندسی نرم‌افزار *SafeHome* و داگ میلر، مدیر گروه مهندسی نرم‌افزار.

گفتگو:

داگ: خبر دارم که شماها دارید دو معماری متفاوت برای محصول *SafeHome* در می‌آورید و این خوب است. فکر می‌کنم سؤال من این باشد که چطور می‌خواهیم گزینه بهتر را انتخاب کنیم.

اد: من دارم روی سبک قراخوانی و بازگشت کار می‌کنم و بعد یا خودم یا جیمی یک معماری شیءگرا در خواهیم آورد.

داگ: بسیار خب و چطور انتخاب کنیم؟

جیمی: من سال آخر تحصیلم یک دوره CS در طراحی گذراندم و یادم هست که چند روش برای این انتخاب وجود دارد.

وینود: بله هست، ولی قدری دانشگاهی هستند. ببینید من فکر می‌کنم می‌توانیم ارزیابی و انتخاب خودمان را با استفاده از پرونده‌های کلرد و سناریوها انجام بدهیم.

داگ: این همان مورد نیست؟

وینود: نه وقتی دارید درباره ارزیابی معماری‌ها صحبت می‌کنید. ما از قبل یک مجموعه use case داریم. بنابراین هر کدام را برای هر دو معماری به کار می‌بریم تا ببینیم سیستم چگونه واکنش نشان می‌دهد و مؤلفه‌ها و کانکتورها در حیطه‌ی use caseها چطور کار می‌کنند.

اد: ایده‌ی خوبی است. مطمئن می‌شویم که چیزی از قلم نیفتاده است.

وینود: درست است، ولی این را هم به ما می‌گوید که آیا طراحی معماری پیچیده است. آیا سیستم باید خودش را بیچ و تاب بدهد تا کار انجام شود؟

جیمی: سناریوها دقیقاً همان use caseها هستند؟

وینود: نه، در این مورد، سناریو چیزی متفاوت است.

داگ: تو داری درباره سناریوی کیفیتی یا سناریوی تغییر صحبت می‌کنی، درست است؟

وینود: بله. کاری که می‌کنیم این است که به طرف‌های ذی‌نفع رجوع کنیم و از آن‌ها بپرسیم *SafeHome* بعداً (مثلاً سه سال بعد) احتمالاً چطور تغییر می‌کند. منظورم، نسخه‌ها و ویژگی‌های جدید است و از این چیزها. ما یک مجموعه سناریوهای تغییر می‌سازیم. یک مجموعه سناریوهای کیفیتی هم توسعه می‌دهیم که صفات مورد نظر در معماری نرم‌افزار را تعریف می‌کنند.

جیمی: و آن‌ها را در معیاری به کار می‌گیریم.

وینود: دقیقاً. سبکی که بهترین مناسبت را با use caseها و سناریوها داشته باشد، سبک مورد انتخاب خواهد بود.

۹-۵-۲ پیچیدگی معماری

یک تکنیک سودمند برای ارزیابی پیچیدگی کلی یک معماری پیشنهادی، عبارت است از در نظر گرفتن وابستگی‌های میان مؤلفه‌های موجود در معماری. این وابستگی‌ها توسط جریان اطلاعات / کنترل موجود در سیستم به‌دست می‌آیند. ژائو [Zha98] سه نوع وابستگی را پیشنهاد می‌کند:

وابستگی‌های اشتراکی (sharing dependencies)، نشان‌گر روابط میان مصرف‌کننده‌هایی هستند که از یک منبع مشترک استفاده می‌کنند یا تولید کننده‌هایی که برای مصرف کننده‌ای مشترک تولید می‌کنند. به‌عنوان مثال، برای دو مؤلفه **u** و **v**، اگر **u** و **v** هر دو به یک سری داده‌های سرتاسری رجوع کنند، بین **u** و **v** وابستگی اشتراکی وجود دارد.

وابستگی‌های جریانی (flow dependencies)، نشان دهنده روابط میان تولیدکننده‌ها و مصرف کننده‌های منابع هستند. به‌عنوان مثال، برای دو مؤلفه **u** و **v** اگر **u** باید پیش از انتقال کنترل به **v** کامل شود (پیش‌نیاز)، یا اگر **u** توسط پارامترها با **v** ارتباط برقرار کند، در آن صورت یک وابستگی جریانی بین **u** و **v** وجود دارد.

وابستگی‌های مقید (constrained dependencies)، نشان‌گر قیدوبندهای حاکم بر جریان نسبی کنترل در میان مجموعه‌ای از فعالیت‌ها هستند. به‌عنوان مثال، برای دو مؤلفه **u** و **v**، اگر **u** و **v** را نتوان همزمان اجرا کرد (انحصار متقابل)، در آن صورت یک رابطه‌ی وابستگی حدی بین **u** و **v** وجود دارد.

وابستگی‌های اشتراکی و جریانی که ژائو ذکر می‌کند، مشابه مفهوم اتصال است که در فصل ۸ بحث شد. معیارهای ساده‌ای برای ارزیابی این وابستگی‌ها در فصل ۲۳ بحث شد.

۹-۵-۳ زبان‌های توصیف معماری

معماری یک خانه، دارای مجموعه‌ای از ابزارهای استاندارد است که ارائه و نمایش طراحی را به‌شیوه‌ای قابل فهم و عاری از ابهام امکان‌پذیر می‌سازد. گرچه معمار نرم‌افزار می‌تواند از نمادگذاری UML، سایر شکل‌های نموداری، و چند ابزار دیگر استفاده کند، برای مشخص‌سازی طراحی معماری به چند رویکرد رسمی‌تر نیاز است.

زبان توصیف معماری (ADL) ابزاری معنایی و نحوی را برای توصیف معماری نرم‌افزار فراهم می‌آورد. هوفمان و همکاران وی [Hof01] پیشنهاد می‌کنند که یک ADL باید توانایی تجزیه مؤلفه‌های معماری، تجزیه هر کدام از مؤلفه‌ها به قطعات معماری بزرگتر و نمایش واسط‌ها (سازوکارهای اتصال) میان مؤلفه‌ها را در اختیار طراح قرار دهد. هنگامی که تکنیک‌های توصیفی و زبانی برای طراحی معماری تعیین شوند، احتمال برقراری روش‌های ارزیابی اثربخش به موازات تکامل طراحی افزایش می‌یابد.

۹-۶ نگاشت معماری‌ها با به‌کارگیری جریان داده‌ها

سبک‌های معماری بحث شده در بخش ۹-۳-۱، معماری‌هایی کاملاً متفاوت را نشان می‌دهند و از این رو تعجبی ندارد که یک نگاشت، گذار از مدل خواسته‌ها به مدل طراحی را انجام دهد. درواقع، برای برخی سبک‌های معماری هیچ نگاشتی وجود ندارد و طراح باید برای تبدیل خواسته‌ها به طراحی برای این سبک‌ها، از فنون بحث شده در بخش ۹-۴ استفاده کند.

ابزارهای نرم‌افزاری

زبان‌های توصیف معماری

در زیر، خلاصه‌ای از چند ADL مهم توسط ریچارد لند [Lan02] فراهم آمده است که با کسب اجازه از وی، عیناً در این جا آورده شده است. لازم به ذکر است که پنج ADL نخست برای اهداف پژوهشی توسعه یافته‌اند و محصولات تجاری نیستند.

Rapide **(http://poset.stanford.edu/rapide)** بر اساس نمادگذاری مجموعه‌هایی با نظم جزئی ساخته شده است و از همین رو، ساختارهای کاملاً جدیدی برای برنامه‌نویسی معرفی می‌کند.

UniCon **(www.cs.cmu.edu/~UniCon)** یک زبان توصیف معماری است که برای کمک به طراحان در تعریف معماری نرم‌افزار بر حسب انتزاع‌هایی که مفید می‌یابند، ساخته شده است.

Aesop **(www.cs.cmu.edu/~able/aesop/)** به مسأله‌ی استفاده مجدد از سبک‌ها می‌پردازد. با Aesop، تعریف سبک‌ها و استفاده از آن‌ها هنگام ایجاد یک سیستم واقعی امکان‌پذیر است.

Wight **(www.cs.cmu.edu/~able/wright/)** یک زبان رسمی شامل عناصر زیر است: مؤلفه‌ها با پورت‌ها، کانکتورها با نقش‌ها و چسب برای متصل کردن نقش‌ها به پورت‌ها. سبک‌های معماری را می‌توان با یک سری گزاره‌ها در زبان تدوین کرد و از این رو، با چک کردن‌های ایستا می‌توان سازگاری و کامل بودن معماری را تعیین کرد.

Acme **(www.cs.cmu.edu/~acme/)** را می‌توان به‌عنوان یک ADL نسل دوم در نظر گرفت، زیرا به منظور شناسایی نوعی مخرج مشترک برای ADLها توسعه یافته است.

UML **(www.uml.org/)** شامل بسیاری از محصولات مورد نیاز برای توصیفات معماری- فرایندها، گره‌ها، نماها و غیره می‌شود. برای توصیفات غیر رسمی، UML بسیار مناسب است، چون استانداردی است که به‌طور گسترده پذیرفته شده است. به هر حال، فاقد قدرت کامل مورد نیاز برای توصیف معماری کافی است.

برای نشان دادن یکی از روش‌های مربوط به نگاشت معماری، معماری فراخوانی و بازگشت (یکی از متداول‌ترین ساختارها برای انواع بسیاری از سیستم‌ها) را در نظر می‌گیریم. معماری فراخوانی و بازگشت را می‌توان در سایر معماری‌های پیچیده‌تر بحث شده در این فصل جای داد. برای مثال، معماری یک یا چند مؤلفه از یک معماری کلاینت- سرور می تواند فراخوانی و بازگشت باشد.

یک تکنیک نگاشت موسوم به طراحی ساخت‌یافته [You79]، غالباً به‌عنوان روشی یک مبتنی بر جریان داده‌ها شناخته می شود، زیرا به کمک آن می‌توان به‌راحتی از نمودار جریان داده‌ها (فصل ۷) به معماری نرم‌افزار رسید.[1] گذار از جریان اطلاعات (که در قالب DFD ارائه می‌شود) به ساختار برنامه، به‌عنوان بخشی از یک فرایند شش مرحله‌ای انجام می‌شود: (۱) نوع جریان اطلاعات تعیین می‌شود؛ (۲) مرزهای جریان اطلاعات مشخص می‌شود؛ (۳) DFD به یک ساختار برنامه‌ای نگاشت می‌شود؛ (۴) سلسله مراتب کنترلی تعریف می‌شود؛ (۵) ساختار حاصل با استفاده از موازین و اصول طراحی مورد پالایش قرار می‌گیرد و (۶) توصیف معماری پالایش شده، تعیین می‌شود.

[1] شایان ذکر است که سایر عناصر مدل خواسته‌ها نیز طی روش نگاشت‌برداری به‌کار گرفته می‌شوند.

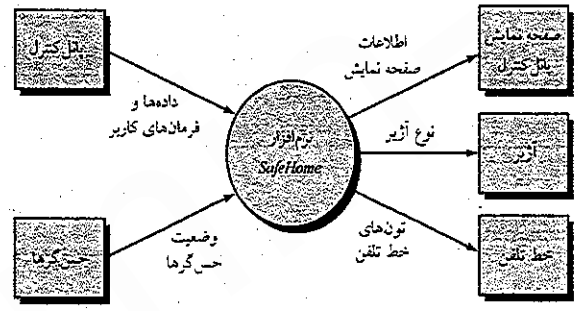
به‌عنوان مثال مختصری از نگاشت جریان داده‌ها، نگاشت مرحله به مرحله‌ای را برای بخش کوچکی از عملکرد *SafeHome* ارائه می‌کنیم.[1] به منظور اجرای حمل نگاشت، نوع جریان اطلاعات باید تعیین گردد. یک نوع جریان اطلاعات، *جریان تبدیل* نام دارد و کیفیت خطی را نشان می‌دهد. داده‌ها در راستای یک مسیر *جریان ورودی* به درون سیستم جریان پیدا می‌کند که در آن از نمایش جهان خارج به شکل نمایش داخلی تبدیل می‌شود. هنگامی که به شکل داخلی درآمد، در یک مرکز *تبدیل*، پردازش می‌شود. سرانجام در راستای یک مسیر *جریان* خروجی به بیرون سیستم جریان پیدا می‌کند که داده‌ها را به شکل جهان خارج تبدیل می‌کند.[2]

۹-۶-۱ نگاشت تبدیل

نگاشت تبدیل (transform mapping) به مجموعه‌ی مراحلی گفته می‌شود که نگاشت از DFD با خصوصیات جریان تبدیل به یک سبک معماری مشخص را امکان‌پذیر می‌سازد. برای نشان دادن این رویکرد، دوباره قابلیت امنیتی منزل در محصول *SafeHome* را در نظر بگیرید.[3]

یک عنصر مدل تحلیل، مجموعه‌ای از نمودارهای جریان داده‌هاست که جریان اطلاعات را داخل قابلیت امنیتی منزل توصیف می‌کند. برای نگاشت این نمودارهای جریان داده‌ها به یک معماری نرم‌افزار، باید مراحل طراحی زیر را آغاز کرد:

مرحله ۱. بازبینی مدل سیستم بنیادی. مدل سیستم بنیادی یا نمودار حیطه‌ی سیستم، قابلیت امنیتی را به‌صورت یک تبدیل مشخص می‌کند که نشان‌دهنده‌ی مصرف‌کننده‌ها و تولیدکننده‌های خارجی جریان داده‌های ورودی و خروجی این قابلیت امنیتی است. در شکل‌های ۹-۱۰ و ۹-۱۱، جریان داده‌ها در سطح صفر و در سطح ۱ برای نرم‌افزار *SafeHome* نشان داده شده است.

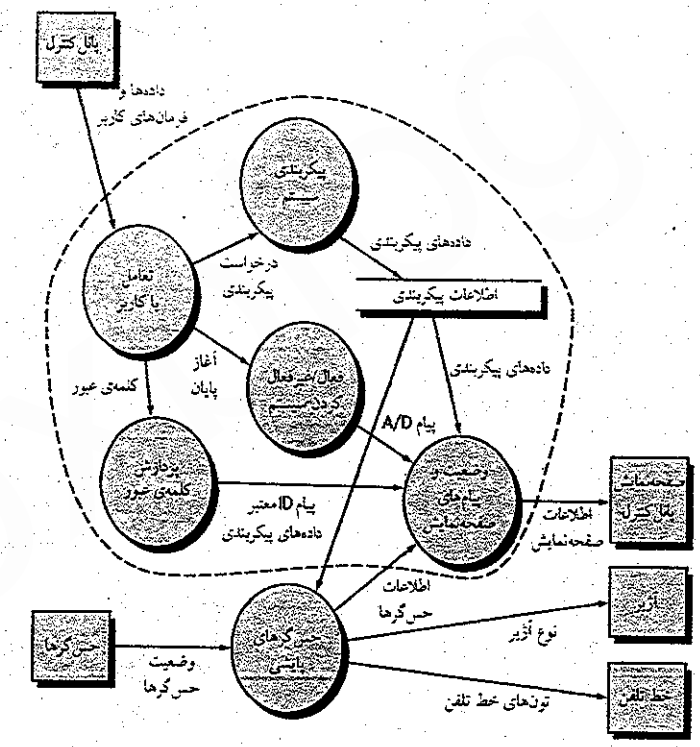


شکل ۹-۱۰ DFD در سطح حیطه‌ای برای قابلیت امنیتی منزل در محصول *SafeHome*

[1] بحث مشروح‌تری درباره طراحی ساخت‌یافته در وب سایت این کتاب ارائه شده است.

[2] یک نوع مهم دیگر جریان اطلاعات، موسوم به *جریان تراکنش*، در این مثال در نظر گرفته نمی‌شود ولی در مثال مربوط به طراحی ساخت یافته در وب سایت این کتاب ارائه شده است.

[3] تنها بخشی از قابلیت عملیاتی امنیت منزل را در نظر خواهیم گرفت که از قاب کنترل استفاده می‌کند. سایر ویژگی‌های بحث‌شده در سرتاسر این کتاب در این‌جا در نظر گرفته نخواهد شد.



شکل ۱۱-۹ DFD سطح یک برای قابلیت امنیتی منزل در محصول *SafeHome*

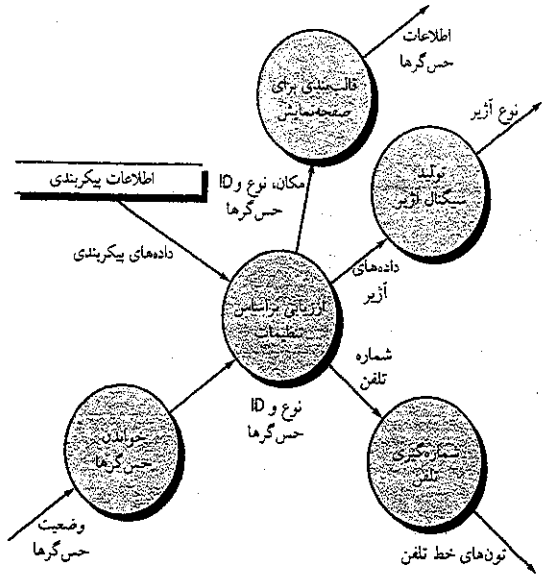
مرحله ۲. بازبینی و پالایش نمودار جریان داده‌ها برای نرم‌افزار. اطلاعات به‌دست آمده از مدل‌های تحلیل موجود در مشخصات خواسته‌های نرم‌افزار، پالایش می‌شود تا جزئیات بیشتری تولید شود. برای مثال، DFDسطح ۲ برای حس‌گرهای پایشی (شکل ۱۲-۹) مورد بررسی قرار می‌گیرد و یک DFD سطح ۳ مطابق شکل ۱۳-۹ به‌دست می‌آید. در سطح ۳، هر تبدیل در نمودار جریان داده‌ها نشان‌گر یک انسجام نسبتاً بالاست (فصل ۸). یعنی، فرایندی که از یک تبدیل ناشی می‌شود، یک عمل متمایز و یگانه انجام می‌دهد که در نرم‌افزار *SafeHome* به‌عنوان یک پیمانه قابل پیاده‌سازی است. بنابراین، DFD شکل ۱۳-۹ حاوی جزئیات کافی برای «نخستین برش» در طراحی زیرسیستم حس‌گرهای نظارتی است و دیگر نیازی به پالایش بیشتر نیست.

مرحله ۳. تعیین اینکه آیا DFD دارای ویژگی‌های جریان تراکنشی است یا جریان تبدیلی.[1] با بررسی DFD (شکل ۱۳-۹)، می‌بینیم که داده‌ها در راستای یک مسیر ورودی به نرم‌افزار وارد می‌شوند و در راستای سه مسیر خروجی از آن خارج می‌شوند. بنابراین، یک ویژگی تبدیل کلی برای جریان اطلاعات در نظر گرفته می‌شود.

اندرز
اگر DFD این بار باز هم پالایش شود، درصدد به‌دست آوردن حباب‌هایی باشید که یکپارچگی بالایی نشان دهند.

نکته‌ی کلیدی
غالباً تا هر دو نوع دیگر جریان داده‌ها در یک مدل جریان‌گرا مواجه خواهید شد. جریان‌ها افراز می‌شوند و ساختار برنامه با استفاده از نگاشت مناسب به‌دست می‌آید.

[1] در جریان تبدیلی، یک آیتم داده‌ای منفرد، موسوم به *تراکنش*، باعث انشعاب جریان داده‌ها در راستای یکی از چند مسیر جریانی می‌شود که ماهیت تراکنش آن‌ها را تعیین می‌کند.



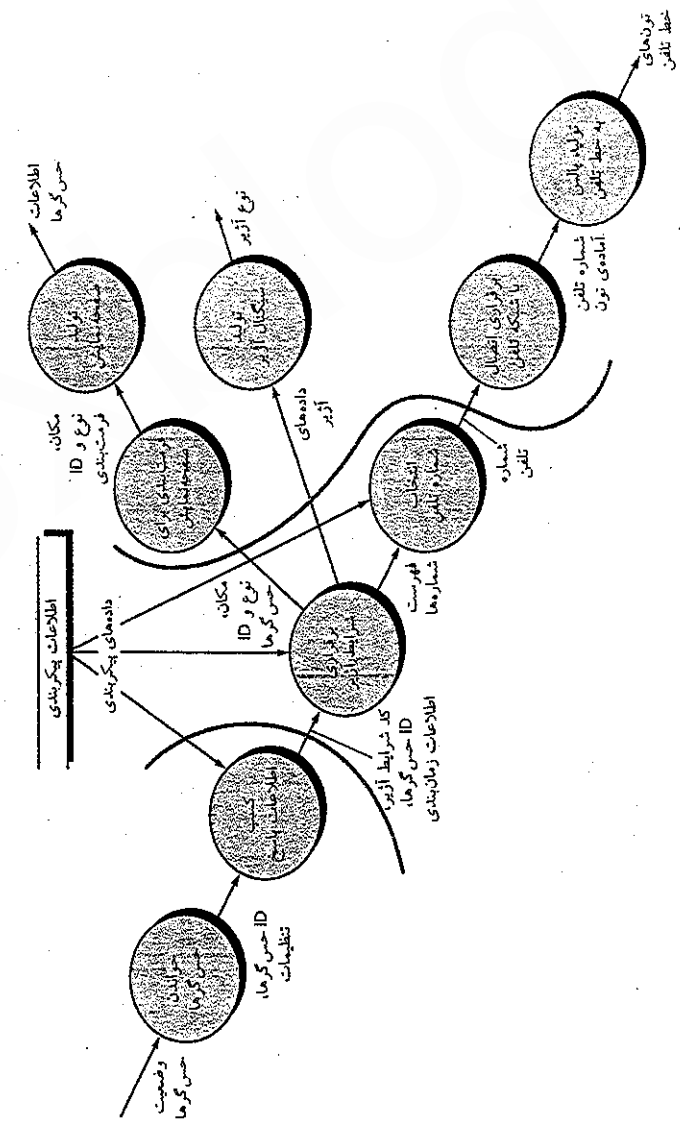
شکل ۱۲-۹ DFD سطح دو که تبدیل حس‌گرهای پایشی را پالایش می‌کند.

اندرز
مکان مرزهای جریان را تغییر دهید تا تولید ساختارهای برنامه‌ای متفاوت را کاوش کنید. این کار زمان‌بر است، کسی نمی‌گوید ولی دید خوبی به‌دست می‌دهد.

مرحله ۴. جداسازی مرکز تبدیل با مشخص کردن مرزهای جریان ورودی و خروجی. در بخش قبلی، جریان ورودی به‌عنوان مسیری توصیف شد که در آن، اطلاعات از شکل خارجی به شکل داخلی تبدیل می‌شود؛ جریان خروجی شکل داخلی را به شکل خارجی تبدیل می‌کند. مرزهای ورودی و خروجی ممکن است به شیوه‌های مختلفی تفسیر شوند. یعنی، طراحان متفاوت ممکن است تقاطی با اندکی تفاوت در جریان را به‌عنوان مکان‌های مرزی در نظر بگیرند. گرچه هنگام انتخاب مرزها باید دقت به عمل آورد، تغییر یک حباب (تبدیل) در راستای مسیر جریان عموماً تأثیر اندکی بر ساختار برنامه نهایی دارد.

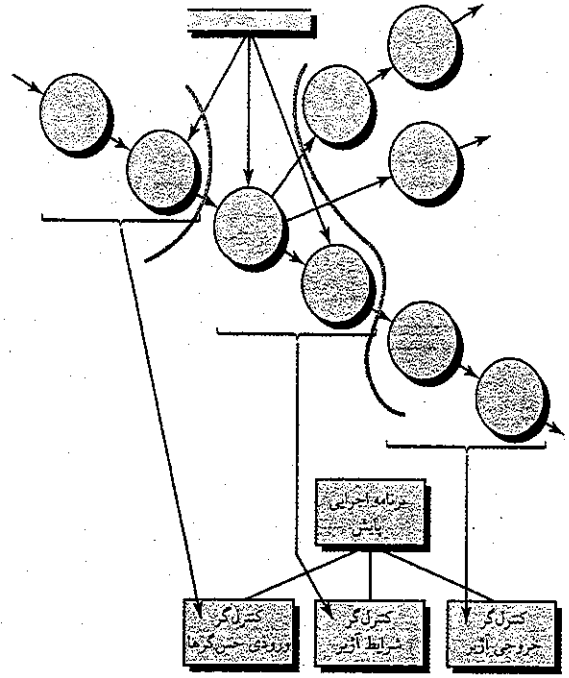
مرزهای جریان برای این مثال به‌صورت منحنی‌های سایه‌داری نشان داده شده‌اند که به‌طور عمودی از جریان عبور می‌کنند (شکل ۱۳-۹). تبدیلات (حباب‌هایی) که مرکز تبدیل را تشکیل می‌دهند در دو مرز سایه‌دار قرار می‌گیرند که از بالا به پایین در این شکل ادامه می‌یابند. تبدیلاتی (حباب‌هایی) که مرکز تبدیل را تشکیل می‌دهند، بین دو مرز سایه‌دار قرار می‌گیرند که در این شکل از بالا به پایین ادامه می‌یابند. برای تنظیم دوباره یک مرز می‌توان استدلالی انجام داد (مثلاً یک مرز جریان ورودی که خواندن حس‌گرها و به‌دست آوردن اطلاعات پاسخ را از هم جدا می‌کند، قابل پیشنهاد است). در این مرحله از طراحی، باید بر انتخاب مرزهای منطقی تأکید شود، نه بر تکرار طولانی تقسیمات.

مرحله ۵. اجرای «فاکتوربندی سطح اول» (first-level factoring). معماری برنامه، که با استفاده از این نگاشت به‌دست می‌آید، توزیع کنترل را از بالا به پایین نشان می‌دهد. *فاکتوربندی*، منجر به ساختار برنامه‌ای می‌شود که در آن، مؤلفه‌های سطح بالا، تصمیم‌گیری‌ها را انجام می‌دهند و مؤلفه‌های سطح پایین کارهای ورودی، محاسباتی و خروجی را انجام می‌دهند. مؤلفه‌های سطح میانی قدری کنترل و قدری کار انجام می‌دهند.



شکل ۱۳-۹ DFD سطح سه برای حس‌گرهای پایشی با مرزهای جریان.

هنگامی که یک جریان تبدیلی به چشم خورد، DFD به ساختاری مشخص (معماری فراخوانی و بازگشت) نگاشت می‌شود که کنترل را برای پردازش اطلاعات ورودی، تبدیل و خروجی فراهم می‌آورد. اولین سطح فاکتوربندی برای حس‌گرهای پایشی در شکل ۱۴-۹ آمده است. یک کنترل‌گر اصلی (موسوم به مدیریت حس‌گرهای ناظر) در بالای ساختار برنامه قرار دارد و به هماهنگ کردن عملیات کنترلی زیردست کمک می‌کند:



شکل ۱۴-۹ فاکتوربندی سطح یک برای حس‌گرهای پایشی.

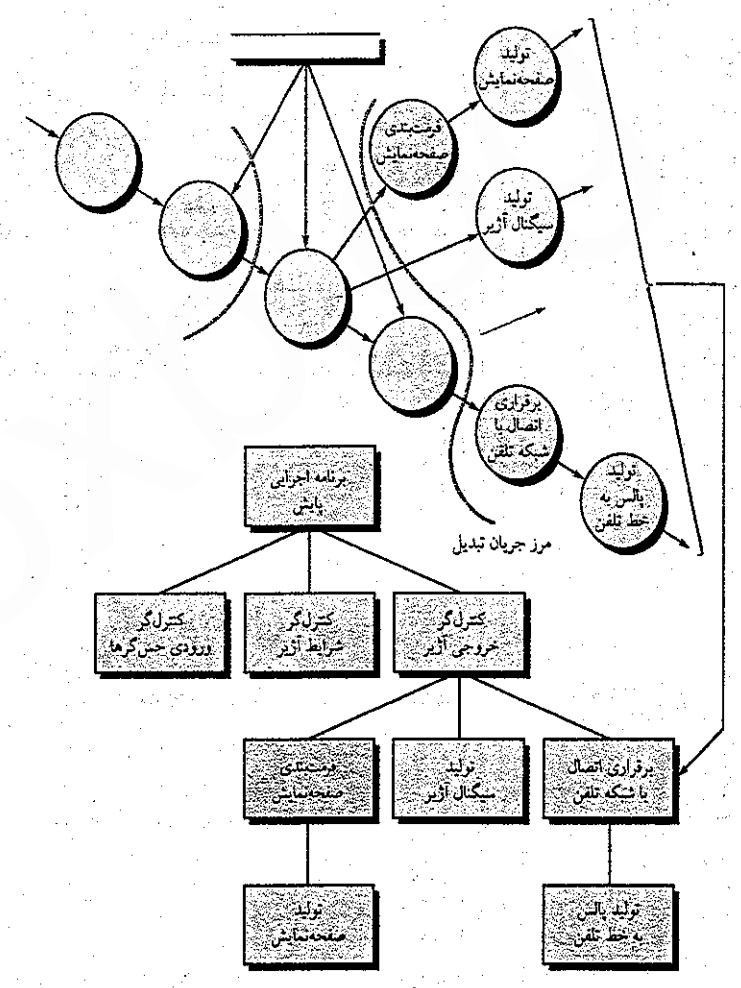
- یک کنترل‌گر پردازش اطلاعات ورودی موسوم به کنترل‌گر ورودی حس‌گر، دریافت کلیه‌ی داده‌های ورودی را هماهنگ می‌کند؛
- یک کنترل‌گر جریان تبدیل موسوم به کنترل‌گر شرایط آژیر، بر انجام کلیه عملیات روی داده‌ها به شکل داخلی آن نظارت دارد (مثلاً مدولی که روال‌های گوناگون تبدیل داده‌ها را فراخوانی می‌کند)؛
- کنترل‌گر پردازش اطلاعات خروجی، که کنترل‌گر خروجی آژیر نیز نامیده می‌شود، تولید اطلاعات خروجی را هماهنگ می‌کند.

گرچه ساختاری سه شاخه از شکل ۹-۱۴ نتیجه می‌شود، در سیستم‌های بزرگ، جریان‌های پیچیده می‌توانند دو یا چند پیمانه کنترل را برای هر یک از عملیات‌های کنترلی دیکته کنند. تعداد پیمانه‌ها در سطح اول باید به حداقلی محدود شود که بتواند کارهای کنترلی را انجام دهد و هنوز ویژگی‌های استقلال عملیاتی را حفظ نماید.

مرحله ۶ اجرای «فاکتوربندی سطح دوم». فاکتوربندی در سطح دوم، با نگاشت تک‌تک تبدیلات (حباب‌های) یک DFD در پیمانه‌های مناسب موجود در معماری انجام‌پذیر است. با شروع از مرز مرکزی تبدیل و حرکت به طرف خارج در راستای مسیرهای ورودی و خروجی، تبدیلات در سطوح زیرین در ساختار نرم‌افزار نگاشت می‌شوند. روش کلی در فاکتوربندی سطح دوم برای جریان داده‌ها در شکل ۱۵-۹ نشان داده شده است.

اندرز

در این مرحله، تعصب به خرج نده. ممکن است بر اساس پیچیدگی سیستمی که قرار است ساخته شود بیش از یک برقراری دو یا چند کنترل‌گر برای پردازش یا محاسبه‌ی ورودی‌ها باشد. اگر عقل سلیم این روش را حکم کرد، چنین کنید.

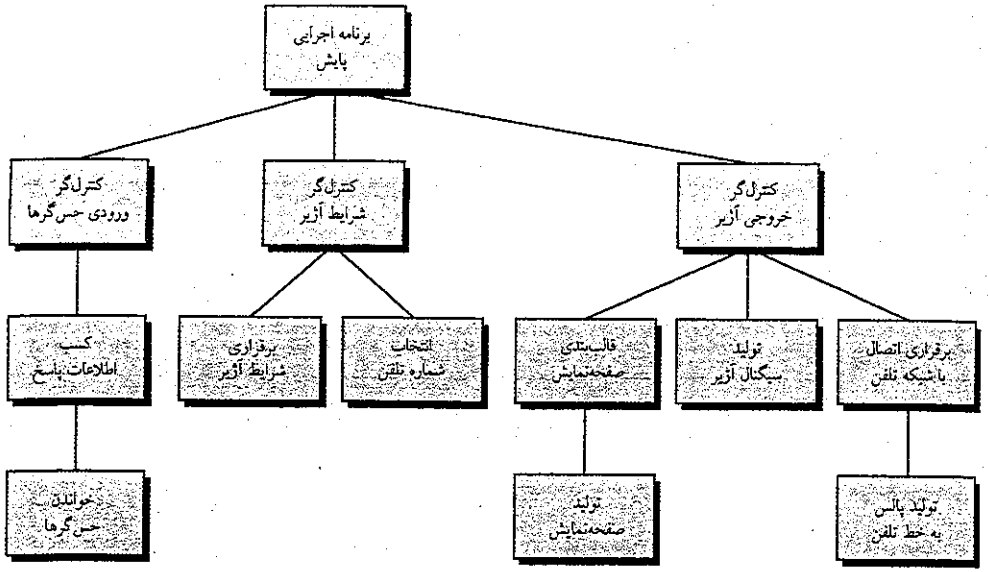


شکل ۱۵-۹ فاکتوربندی سطح دو برای حس‌گرهای پایشی.

گرچه شکل ۱۵-۹ نشان‌دهنده‌ی نگاشت یک به یک میان تبدیلات DFD و پیمانه‌های نرم‌افزار است، نگاشت‌های متفاوت به کرّات رخ می‌دهد. دو یا حتی سه حباب را می‌توان ترکیب کرد و به‌صورت پیمانه‌ای واحد نمایش داد (یا به‌خاطرسپردن مشکلات انسجام) یا یک حباب منفرد را می‌توان به دو یا چند پیمانه بسط داد. ملاحظات عملی و موازین کیفیت طراحی، پیامدهای فاکتوربندی سطح دوم را تعیین می‌کنند. بازبینی و پالایش ممکن است منجر به تغییراتی در ساختار شود، ولی می‌تواند به‌عنوان طراحی «تکرار اول» عمل کند.

فاکتوربندی سطح دوم برای جریان ورودی، به همان شیوه پیش می‌رود. فاکتوربندی مجدد، با حرکت از مرز مرکز *تبدیل* به طرف خارج و روی جریان ورودی انجام می‌شود. مرکز تبدیل نرم‌افزار زیرسیستم حس‌گرهای پایشی تا حدی به شکل متناوب نگاشت می‌شود. هر یک از تبدیلات داده‌ای یا تبدیلات محاسباتی مربوط به بخش تبدیلی DFD به یک پیمانه‌ی زیردستِ کنترل‌گر تبدیل، نگاشت می‌شود. معماری کامل در نخستین دور تکرار، در شکل ۱۶-۹ نشان داده شده است.

اندرز
پیمانه‌های کنترلی را حذف کنید. یعنی اگر یک پیمانه کنترلی، کاری جز کنترل یک پیمانه‌ی دیگر انجام نمی‌دهد، وظیفه‌ی کنترلی آن باید به یک پیمانه سطح بالاتر محول شود.



شکل ۱۶-۹ ساختار دور اول تکرار برای حس‌گرهای پایشی.

مؤلفه‌هایی که بدین ترتیب نگاشت می‌شوند و در شکل ۱۶-۹ نشان داده شده‌اند، طراحی اولیه معماری نرم‌افزار را مشخص می‌کنند. گرچه پیمانه‌ها به شیوه‌ای نامگذاری می‌شوند که حاکی از عملکرد آنها باشد، برای هر یک باید شرح پردازشی مختصری (که از PSPEC تهیه شده در هنگام مدل‌سازی تحلیلی برگرفته شده است) نوشته شود. در این شرح موارد زیر بیان می‌شود: اطلاعاتی که به پیمانه تحویل داده می‌شود و از آن تحویل گرفته می‌شود (توصیف واسط)؛ اطلاعاتی که توسط پیمانه حفظ می‌شوند، مثل داده‌های نگهداری شده در یک ساختمان داده محلی؛ یک نسخه عملکردی که نشانگر وظایف و نقاط تصمیم‌گیری اصلی است؛ بحث مختصری از محدودیت‌ها و ویژگی‌های خاص (مثل I/O فایل‌ها، ویژگی‌های وابسته به سخت‌افزار، خواسته‌های زمان‌بندی خاص).

اندرز
پیمانه‌های «کارگر» را در ساختار برنامه در سطحی پایین حفظ کنید. این کار به معماری‌ای منجر خواهد شد که نگهداری از آن آسان‌تر است.

مرحله ۷. پالایش نخستین تکرار معماری با استفاده از اصول طراحی برای بهبود بخشیدن به کیفیت نرم‌افزار. نخستین تکرار معماری را همواره می‌توان با استفاده از مفاهیم استقلال پیمانه‌ها (فصل ۱۳) پالایش کرد. پیمانه‌ها، انبساط یا انقباض می‌یابند تا فاکتوربندی معقول، انسجام خوب، اتصال کمینه و مهمتر از همه، ساختاری ایجاد شود که بدون مشکل پیاده‌سازی شود، بدون ایجاد سردرگمی آزموده شود و بدون دردسر بتوان آن را نگهداری کرد.

پالایش‌ها توسط روش‌های تحلیل و سنجشی که به اختصار در بخش ۹-۵ شرح داده شدند، و نیز ملاحظات عملی و عقلایی دیکته می‌شوند. برای مثال، مواردی پیش می آید که کنترل‌گر مربوط به جریان داده‌های ورودی، کاملاً غیرضروری است، پردازش ورودی در پیمانه زیردستِ کنترل‌گر ضروری است، اتصال بالا ناشی از داده‌های سراسری، قابل پرهیز نیست، یا ویژگی‌های ساختاری بهینه قابل دستیابی نیست. خواسته‌های نرم‌افزار همراه با داوری بشر، آخرین انتخاب است.

«تا آن حد که امکان دارد، ساده کنید. ولی ساده‌تر نه.»
آلبرت اینشتین

SafeHome

پالایش معماری در نخستین یُرش

صحنه: کابین جیمی، شروع مدل‌سازی طراحی.

نقش‌آفرینان: جیمی و اد- اعضای تیم مهندسی نرم‌افزار *SafeHome*.

گفتگو:

[اد به تازگی طراحی زیر سیستم حس‌گرهای پایشی را به پایان برده است. او به سراغ جیمی می‌رود تا نظر او را جویا شود.]

اد: خلاصه، این هم معماری‌ای که به‌دست آوردم.

[اد شکل ۹-۱۶ را به جیمی نشان می‌دهد و او هم چند لحظه‌ای آن را بررسی می‌کند.]

جیمی: عالی است، ولی فکر کنم می‌توانیم چند تا کار بکنیم تا ساده‌تر و بهتر شود.

اد: مثلاً؟

جیمی: حتم، چرا از مؤلفه‌ی «*کنترل‌گر ورودی حس‌گرها*» استفاده کردی؟

اد: چون برای نگاشت به یک کنترل‌گر نیاز داریم.

جیمی: نه واقعاً. کنترل‌گر، کار زیادی انجام نمی‌دهد چون برای داده‌های ورودی یک مسیر منفرد را مدیریت می‌کنیم. می‌توانیم کنترل‌گر را حذف کنیم بدون این که اثر سوئی بگذارد.

اد: مشکلی نیست. این تغییر را می‌دهم و...

جیمی (با لبخند): دست نگه دار! به‌علاوه می‌توانیم مؤلفه‌های «*برقراری شرایط آژیر* و *انتخاب شماره تلفن*» را هم کنار بگذاریم. کنترل‌گر تبدیلی که نشان می‌دهی، واقعاً ضرورتی ندارد و کاهش کوچکی در یکپارچگی قابل تحمل است.

اد: ساده‌سازی، نه؟

جیمی: اره. و در حالی که این پالایش‌ها را انجام می‌دهیم، بد نیست مؤلفه‌های *فرمت‌بندی صفحه‌نمایش* و *تولید صفحه‌نمایش* را کنار بگذاریم. می‌توانیم یک پیمانه‌ی جدید به نام *ایجاد صفحه‌نمایش* تعریف کنیم.

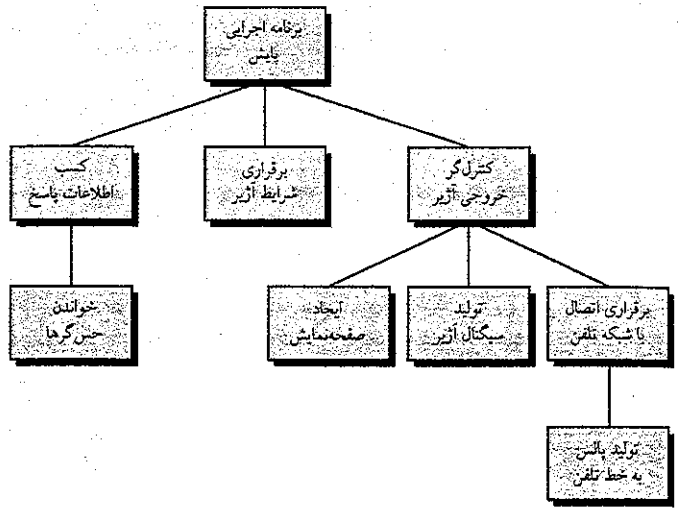
اد (در حال ترسیم): پس فکر می‌کنی باید این طوری باشد؟

[شکل ۹-۱۷ را نشان می‌دهد]

جیمی: شروع خوبی است.

هدف هفت مرحله‌ی فوق، توسعه یک نمایش معماری از نرم‌افزار است. یعنی، هنگامی که ساختار تعیین شد، می‌توانیم معماری نرم‌افزار را با در نظر گرفتن آن به‌عنوان یک کلّیت، مورد ارزیابی و پالایش قرار دهیم. اصلاحاتی که در این زمان انجام می‌شوند، نیاز به کار اضافی چندانی ندارند و در عین حال می‌توانند بر کیفیت نرم‌افزار تأثیر زیادی بگذارند.

خوب است اندکی مکث کنید و به اختلاف میان روش طراحی فوق‌الذکر و فرایند «نوشتن برنامه» توجه کنید. اگر کُد تنها نمایش نرم‌افزار باشد، سازنده در ارزیابی یا پالایش آن در سطح سرتاسری با مشکلات زیادی مواجه خواهد شد و درواقع به دلیل وجود درختان، جنگل را به سختی خواهد دید.



شکل ۹-۱۷ ساختار پالایش شده‌ی برنامه برای حس‌گرهای پایشی.

۹-۶-۲ پالایش طراحی معماری

پیش از هرگونه بحث درباره پالایش طراحی باید توضیح زیر آورده شود:

«به‌خاطر داشته باشید که در شایستگی «طراحی بهینه‌ای» که نتیجه‌بخش نباشد، تردید وجود دارد.» دغدغه‌ی شما باید توسعه‌ی نمایشی از نرم‌افزار باشد که واجد همه‌ی شرایط عملیاتی و کارایی باشد.

پالایش معماری نرم‌افزار طی مراحل طراحی باید تشویق شود. چنان که قبلاً در این فصل بحث شد، سبک‌های معماری متفاوتی ممکن است به‌دست آید، پالایش شود و برای «بهترین» رویکرد، مورد ارزیابی قرار گیرد. این رویکرد بهینه‌سازی، یکی از مزایای توسعه‌ی نمایش معماری نرم‌افزار است.

شایان ذکر است که سادگی ساختاری، غالباً انعکاسی از ظرافت و بازدهی است. پالایش طراحی باید به دنبال کوچکترین تعداد مؤلفه‌هایی باشد که با پیمانه‌بندی اثربخش سازگار باشد و نیز به دنبال ساختمان‌های داده‌ای با کمترین پیچیدگی باشد که خواسته‌های اطلاعاتی را به طرزی مناسب، پاسخ‌گو باشند.

پس از ایجاد معماری چه اتفاقی رخ می‌دهد؟

۹-۷ خلاصه

از یک دیدگاه کلی‌نگر، طراحی معماری با استفاده از چهار گام متمایز قابل انجام است. نخست، سیستم باید در حیطه‌ی مناسب ارائه شود. یعنی طراح باید موجودیت‌های خارجی را که نرم‌افزار با آن‌ها تعامل دارد و نیز ماهیت این تعامل را تعریف کند. هنگامی که حیطه مشخص شد، طراح باید مجموعه‌ای از انتزاع‌های سطح بالا، موسوم به نمونه اولیه، را شناسایی کند که عناصر اصلی رفتار یا عملکرد سیستم را نشان می‌دهند. پس از تعریف انتزاع‌ها، طراح شروع به نزدیک‌تر شدن به دامنه‌ی پیاده‌سازی می‌کند. مؤلفه‌ها در حیطه‌ی معماری‌ای که آن‌ها را پشتیبانی می‌کند، شناسایی و نمایش داده می‌شوند. سرانجام، نمونه‌برداری مشخصی از معماری انجام می‌شود تا طراحی در حیطه جهان واقعی «تصویب» گردد.

به‌عنوان مثالی از طراحی معماری، روش نگاشت ارائه شده در این فصل، از خصوصیات جریان داده‌ها برای به‌دست آوردن یک سبک معماری رایج استفاده می‌کند. یک نمودار جریان داده‌ها با استفاده از رویکرد نگاشت تبدیل، به ساختار برنامه نگاشت می‌یابد. نگاشت تبدیل برای جریان اطلاعاتی به‌کار می‌رود که مرزهای متمایزی میان داده‌های وارد شونده و خارج شونده از خود نشان می‌دهند. DFD به ساختاری نگاشت می‌شود که کنترل را از طریق سلسله مراتب فاکتوربندی شده‌ی جداگانه، به ورودی، پردازش و خروجی تخصیص می‌دهد. هنگامی که معماری‌ای به‌دست آمد، به آن جزئیاتی افزوده می‌شود و سپس با استفاده از ملاک‌های کیفیت مورد ارزیابی قرار می‌گیرد.

معماری نرم‌افزار، از سیستمی که قرار است ساخته شود، یک نمای کلی فراهم می‌آورد. معماری، ساختار و سازمان‌دهی مؤلفه‌های نرم‌افزار، خواص آنها، و ارتباطات میان آنها را مجسم می‌کند. قطعات نرم‌افزار شامل مؤلفه‌های برنامه‌ای و نمایش‌های گوناگونی از داده‌ها است که توسط برنامه دستکاری می‌شوند. بنابراین، طراحی داده‌ها بخشی تفکیک ناپذیر از معماری نرم‌افزار به‌شمار می‌رود. معماری، تصمیم‌گیری‌های طراحی اولیه را برجسته و نمایان کرده راهکاری برای در نظر گرفتن مزایای ساختارهای سیستمی متفاوت فراهم می‌آورند.

چند سبک و الگوی معماری متفاوت در اختیار مهندس نرم‌افزار هست که می تواند در یک ژانر معماری مفروض به کار گیرد. هر سبک، گروهی از سیستم‌ها را توصیف می‌کند که موارد زیر را در بر می گیرد: مجموعه‌ای از مؤلفه‌ها که وظیفه‌ی مورد نیاز سیستم را انجام می‌دهند، مجموعه‌ای از کانکتورها که ارتباطات را میسر می‌سازند، هماهنگ‌سازی‌ها و همکاری میان مؤلفه‌ها، قیدوبندهایی که تعیین می‌کنند مؤلفه‌ها را چگونه می‌توان منسجم کرد تا سیستم را بسازند و مدل‌های معنایی که طراح را قادر به درک خواص کلی سیستم می‌سازند.

مسائل و نکاتی برای تعمق

۹-۱ با استفاده از معماری ساختمان خانه به‌عنوان استعاره، نظیرهایی برای معماری نرم‌افزار بیابید. دو رشته‌ی معماری کلاسیک و معماری نرم‌افزار چه شباهت‌هایی دارند؟ چه تفاوت‌هایی دارند؟

۹-۲ دو یا سه مثال از کاربردهای هر یک از سبک‌های معماری ذکر شده در بخش ۹-۳-۱ ذکر کنید.

۹-۳ برخی از سبک‌های معماری ذکر شده در بخش ۹-۳-۱، ماهیتی سلسله مراتبی دارند و برخی دیگر خیر. از هر دو نوع، فهرستی تهیه کنید. سبک‌هایی که سلسله مراتبی نیستند چگونه پیاده‌سازی می‌شوند؟

۹-۴ اصطلاح‌های *سبک معماری*، *الگوی معماری* و *چارچوب* (که در این کتاب بحث نشده است) غالباً در بحث‌های معماری نرم‌افزار مشاهده می‌شوند. پژوهشی انجام دهید و شرح دهید که این اصطلاحات چه تفاوتی با همتای خود دارند؟

۹-۵ یک برنامه کاربردی را که با آن آشنا هستید، انتخاب کنید. هر یک از پرسش‌های مطرح شده برای کنترل و داده‌ها (بخش ۹-۳-۳) را پاسخ دهید.

۹-۶ درباره ATAM پژوهشی انجام دهید (با استفاده از [Kaz98]) و بحث مفصلی از شش مرحله‌ی ارائه شده در بخش ۹-۵-۱ ارائه دهید.

۹-۷ اگر مسأله‌ی ۹-۶ را حل نکرده اید، این کار را انجام دهید. با استفاده از روش‌های طراحی توصیف شده در این فصل، یک معماری نرم‌افزار برای PHTRS توسعه دهید.

۹-۸ با استفاده از نمودار جریان داده‌ها و روایت پردازش، یک سیستم کامپیوتری توصیف کنید که خصوصیات جریان تبدیل متمایز داشته باشد. مرزهای جریان را تعریف کنید و DFD را با استفاده از تکنیک توصیف شده در بخش ۹-۶-۱ به معماری نرم‌افزار نگاشت کنید.

فصل ۱۰

طراحی در سطح مؤلفه‌ها

نگاهی گذرا

طراحی در سطح مولفه‌ها چیست؟ طی طراحی معماری، مجموعه‌ی کاملی از مؤلفه‌های نرم‌افزاری تعریف می‌شوند. ولی ساختارهای داخلی داده‌ها و جزئیات پردازش هر مؤلفه، در سطحی از انتزاع نشان داده نمی‌شود که به کُد نزدیک باشد. در طراحی در سطح مولفه‌ها، ساختمان داده‌ها، الگوریتم‌ها، سازوکارهای ارتباطی و خصوصیات واسط‌های هر مؤلفه از نرم‌افزار تعریف می‌شوند.

چه کسی آن را انجام می‌دهد؟ طراحی در سطح مؤلفه‌ها را مهندس نرم‌افزار انجام می‌دهد.

چرا اهمیت دارد؟ باید بتوانید پیش از ساخت نرم‌افزار تعیین کنید که آیا کار می‌کند یا خیر. طراحی در سطح مؤلفه‌ها، نرم‌افزار را به شیوه‌ای نشان می‌دهد که به کمک آن بتوان جزئیات طراحی را برای صحت و سازگاری با سایر نمایش‌های طراحی مورد بازبینی قرار داد (یعنی، معماری داده‌ها و طراحی واسط‌ها). به این ترتیب، ابزاری به‌دست می‌آید که با استفاده از آن می‌توانیم ارزیابی کنیم که آیا ساختمان داده‌ها، واسط‌ها و الگوریتم کار می‌کنند یا خیر.

مراحل کار کدام است؟ نمایش‌های طراحی داده‌ها، معماری و واسط‌ها، بستری برای طراحی در سطح مؤلفه‌ها تشکیل می‌دهند. تعریف کلاس‌ها یا توصیف پردازش برای هر مؤلفه، به یک طراحی مشروح تبدیل می‌شود که برای مشخص‌کردن ساختمان داده‌های داخلی، جزئیات واسط‌های محلی و منطق پردازش، از شکل‌های نموداری یا متنی استفاده می‌کنند. نمادگذاری طراحی، شامل نمودارهای UML و شکل‌های مکمل می‌شود. طراحی رویه‌ای با استفاده از یک مجموعه‌ای از ساختارهای برنامه‌نویسی ساخت‌یافته مشخص می‌شود. غالباً به‌جای ساخت مؤلفه‌های جدید می‌توان از مؤلفه‌های موجود استفاده کرد.

محصول کار چیست؟ طراحی برای هر مؤلفه، که در قالب‌های گرافیکی، جدول یا نمادهای متنی ارائه می‌شود، محصول کاری تولید شده طی طراحی در سطح مؤلفه‌هاست.

چگونه اطمینان حاصل کنم که کار را درست انجام داده‌ام؟ بازبینی روی طراحی انجام می‌شود. طراحی بررسی می‌شود تا تعیین گردد که آیا ساختمان داده‌ها، واسط‌ها، ترتیب پردازش‌ها و شرط‌های منطقی درست هستند یا خیر، و آیا تبدیل کنترلی یا داده‌ای مناسب تخصیص یافته به هر مؤلفه را طی مراحل اولیه‌ی طراحی تولید می‌کنند یا خیر.

طراحی در سطح مؤلفه‌ها، پس از طراحی داده‌ها، طراحی معماری و طراحی واسط انجام می‌شود. هدف از این فعالیت، ترجمه‌ی مدل طراحی به نرم‌افزار است. ولی سطح انتزاع در مدل طراحی موجود، نسبتاً بالا و سطح انتزاع برنامه‌ای که کار کند، پایین است. این ترجمه می‌تواند مشکل‌آفرین باشد و باعث وارد شدن خطاهای ظریفی شود که یافتن و تصحیح آنها در مراحل بعدی فرایند نرم‌افزار دشوار است. ادزگار دیکسترا [Dij72] که سهم عمده‌ای در شناخت ما از طراحی دارد، در یک سخنرانی مشهور گفته است:

> به‌نظر می‌رسد نرم‌افزارها با بسیاری از محصولات دیگری که در آنها کیفیت بالاتر به معنای قیمت بالاتر است، تفاوت دارند. آنها که نرم‌افزارهای واقعاً قابل اطمینان می‌خواهند، پی خواهند برد که باید وسیله‌ای برای جلوگیری از اکثر اشکال‌ها بیابند و درنتیجه، فرایند برنامه‌نویسی ارزان‌تر تمام می‌شود ... برنامه‌نویسان کارامد ... نباید وقت خود را برای اشکال‌زدایی هدر دهند - آنها نباید تولید اشکال کنند.

این سخنان سال‌ها قبل ایراد شده‌اند، ولی امروزه کماکان صحت خود را حفظ کرده‌اند. هنگامی که مدل طراحی به کد منبع ترجمه شد، باید مجموعه‌ای از اصول طراحی را دنبال کنیم که نه تنها ترجمه را انجام می‌دهند، بلکه «ایجاد اشکال هم نمی‌کنند».

این امکان وجود دارد که طراحی در سطح مؤلفه‌ها را با استفاده از یک زبان برنامه‌نویسی نشان دهیم. در اصل، برنامه با استفاده از مدل طراحی به عنوان راهنما ایجاد می‌شود. یک روش دیگر برای نشان دادن طراحی در سطح مؤلفه‌ها، استفاده از یک نمایش واسطه (مثلاً گرافیکی، جدولی و متنی) است که به‌راحتی به کد منبع قابل ترجمه باشد. سازوکار به‌کاررفته برای نمایش طراحی در سطح مؤلفه‌ها هرچه که باشد، ساختمان داده‌ها، واسط‌ها و الگوریتم‌هایی که تعریف می‌شوند، باید از انواع متفاوتِ دستورالعمل‌های طراحی کاملاً رویه‌ای پیروی کنند که به پرهیز از خطا در اثنای تکامل طراحی رویه‌ای کمک می‌کند. در این فصل به بررسی این دستورالعمل‌ها و روش‌های طراحی در دسترس برای رسیدن به این اهداف می‌پردازیم.

۱-۱۰ مؤلفه چیست؟

مؤلفه به قطعات سازنده‌ی پیمانه‌ای نرم‌افزار کامپیوتری گفته می‌شود. به‌طور رسمی‌تر، در مشخصه‌ی زبان مدل‌سازی یکپارچه OMG [OMG03] مؤلفه به این صورت تعریف می‌شود: «...بخش پیمانه‌ای، قابل استقرار و قابل تعویض از یک سیستم که جزئیات پیاده‌سازی را در خود دارد و مجموعه‌ای از واسط‌ها را ارائه می‌دهد.»

چنان که در فصل ۹ بحث شد، مؤلفه‌ها معماری نرم‌افزار را تشکیل می‌دهند و در نتیجه در دست‌یابی به اهداف و خواسته‌های سیستمی که قرار است ساخته شود، نقش دارند. از آن جا که مؤلفه‌ها در داخل معماری نرم‌افزار جای دارند، باید قادر به برقراری ارتباط و همکاری با سایر مؤلفه‌ها و با موجودیت‌های خارجی (مانند سیستم‌های دیگر، دستگاه‌ها و آدم‌ها) باشند که خارج از مرزهای نرم‌افزار قرار دارند.

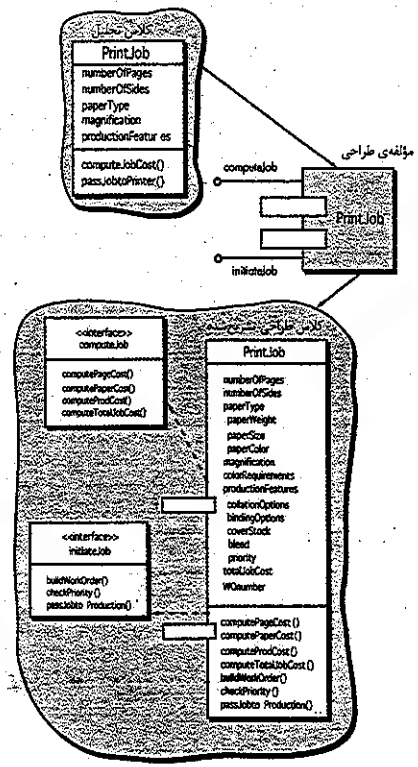
معنای واقعی مؤلفه، به دیدگاه مهندس نرم‌افزاری بستگی دارد که از آن استفاده می‌کند. در بخش‌های بعدی، به بررسی سه دیدگاه مهم درباره ماهیت مؤلفه‌ها و چگونگی استفاده از آن‌ها در پیشرفت مدل‌سازی طراحی خواهیم پرداخت.

> «جزئیات، جزئیات نیستند بلکه طراحی را می‌سازند.»
>
> چارلز ایمز

۱-۱-۱۰ دیدگاه شیءگرا

در حیطه‌ی مهندسی نرم‌افزار شیءگرا، مؤلفه حاوی مجموعه از کلاس‌هاست که با یکدیگر همکاری دارند.[1] هر کلاس در داخل یک مؤلفه دارای جزئیات کامل است به‌طوری که شامل کلیه صفات و عملیات‌ها مرتبط با پیاده‌سازی آن کلاس می‌شود. به‌عنوان بخشی از پرداختن به جزئیات طراحی، همه‌ی واسط‌هایی که کلاس‌ها را قادر به برقراری ارتباط و همکاری با سایر کلاس‌های طراحی می‌کنند نیز باید تعریف شوند. برای نیل به این مقصود، با مدل خواسته‌ها شروع می‌کنید و بر جزئیات کلاس‌های تحلیلی (برای مؤلفه‌هایی که با دامنه‌ی مسأله مرتبط هستند) و کلاس‌های زیرساختی (برای مؤلفه‌هایی که سرویس‌ها را برای دامنه مسأله فراهم می‌سازند) می‌افزایید.

برای نشان دادن این فرایند، پرداختن به جزئیات طراحی، نرم‌افزار پیچیده‌ای را در نظر بگیرید که قرار است برای یک چاپخانه ساخته شود. هدف کلی این نرم‌افزار، جمع آوری خواسته‌های مشتری در پیشخوان، تعیین هزینه کار چاپی و سپس تحویل کار چاپی به یک بخش تولید خودکار است. طی مهندسی خواسته‌ها، کلاس تحلیلی با نام **PrintJob** به‌دست آمده است. صفات و عملیات‌های تعیین‌شده در طول فرایند تحلیل، در بالای شکل ۱-۱۰ ذکر شده‌اند.



شکل ۱-۱۰ تشریح یک مؤلفه طراحی.

[1] در برخی موارد، مؤلفه ممکن است تنها حاوی یک کلاس باشد.

> **نکته‌ی کلیدی**
>
> از دیدگاه شیءگرا، مؤلفه به مجموعه‌ای از کلاس‌ها گفته می‌شود که با یکدیگر همکاری دارند.

طراحی معماری، **PrintJob** به‌عنوان مؤلفه‌ای در داخل معماری نرم‌افزار تعریف می‌شود و با استفاده از نمادگذاری UML[1]، در میانه‌ی سمت راست شکل نمایش داده می‌شود. توجه دارید که **PrintJob** دو واسط دارد، *computeJob* که قابلیت محاسبه هزینه‌ها را فراهم می‌سازد و *initiateJob* که کار را به بخش تولید تحویل می‌دهد. این‌ها به‌صورت نمادهای «آب نبات چوبی»، در طرف چپ چارگوش نشان دهنده‌ی مؤلفه، نمایش داده می‌شوند.

طراحی در سطح مؤلفه‌ها از همین نقطه آغاز می‌شود. جزئیات مؤلفه‌ی **PrintJob** باید تعیین شود تا اطلاعات کافی برای هدایت پیاده‌سازی فراهم گردد، جزئیات کلاس تحلیل اولیه افزوده می‌شود تا همه‌ی صفات و عملیات‌های مورد نیاز برای پیاده‌سازی کلاس به‌صورت مؤلفه‌ی **PrintJob** تعیین شود. با رجوع به بخش پایین و سمت راست شکل ۱-۱۰، کلاس طراحی **PrintJob** که اکنون جزئیات آن تعیین شده است، حاوی اطلاعات مشروح‌تری درباره صفات و همچنین توصیف مبسوطی از عملیات‌های مورد نیاز برای پیاده‌سازی آن مؤلفه است. واسط‌های *computeJob* و *initiateJob* ارتباط و همکاری با سایر مؤلفه‌ها را (که در این جا نشان داده نشده‌اند) بیان می‌کنند. برای مثال، عملیات *computePageCost( )* (بخشی از واسط *computeJob*) ممکن است با مؤلفه‌ی *pricingTable*، که حاوی اطلاعات تعیین قیمت چاپ است، همکاری کند. عملیات *checkPriority( )* (بخشی از واسط *initialJob*) ممکن است برای تعیین نوع و اولویت کارهایی که در حال حاضر منتظر چاپ هستند، با مؤلفه‌ی **JobQueue** همکاری کند.

اندرز

به‌خاطر داشته باشید که مدل‌سازی تحلیل و مدل‌سازی طراحی هر دو، کشف‌هایی مبتنی بر تکرارند. تشریح کلاس‌های تحلیل ممکن است نیاز به مراحل تحلیل اضافی داشته باشد که مراحل مدل‌سازی طراحی از پس آن می‌آید تا کلاس طراحی تشریح‌شده (جزئیات مؤلفه) نمایش داده شود.

این فعالیتِ پرداختن به جزئیات، برای هر کدام از مؤلفه‌های تعریف شده به‌عنوان بخشی از طراحی معماری به‌کار می‌رود و پس از این که کامل شد، به هر صفت، عملیات و واسط نیز جزئیات بیشتری افزوده می‌شود. ساختمان داده‌های مناسب برای هر صفت باید مشخص شود. به‌علاوه، جزئیات الگوریتم مورد نیاز برای پیاده‌سازی منطق پردازش در هر عملیات، طراحی می‌شود. این فعالیت طراحی رویه‌ای را بعداً در همین فصل مورد بحث قرار می‌دهیم. سرانجام، سازوکارهای لازم برای پیاده‌سازی واسط، طراحی می‌شود. برای نرم‌افزار شیءگرا، این ممکن است شامل توصیفی از همه‌ی پیام‌های لازم برای برقراری ارتباط میان اشیای داخلی سیستم شود.

۲-۱-۱۰ دیدگاه سنتی

مؤلفه در حیطه‌ی مهندسی نرم‌افزار سنتی، یک عنصر عملیاتی از برنامه است که منطق پردازش، ساختمان داده‌های داخلی که برای پیاده‌سازی منطق پردازش لازم‌اند و واسطی را در بر می‌گیرند که فراخوانی مؤلفه‌ها و تحویل داده‌ها به آن را میسر می‌سازد. مؤلفه‌های سنتی که به آن‌ها، *پیمانه* نیز گفته می‌شود، در داخل معماری نرم‌افزار جای دارند و به‌عنوان یکی از سه نقش مهم عمل می‌کنند: (۱) *مؤلفه‌ی کنترلی*، (۲) *مؤلفه‌ی دامنه مسأله* (که یک قابلیت عملیاتی کامل یا بخشی از آن را که مورد نیاز مشتری است، پیاده‌سازی می‌کند) یا (۳) *مؤلفه‌ی زیرساختی* (که مسؤول قابلیت‌های عملیاتی پشتیبان برای پردازش لازم در دامنه‌ی مسأله است).

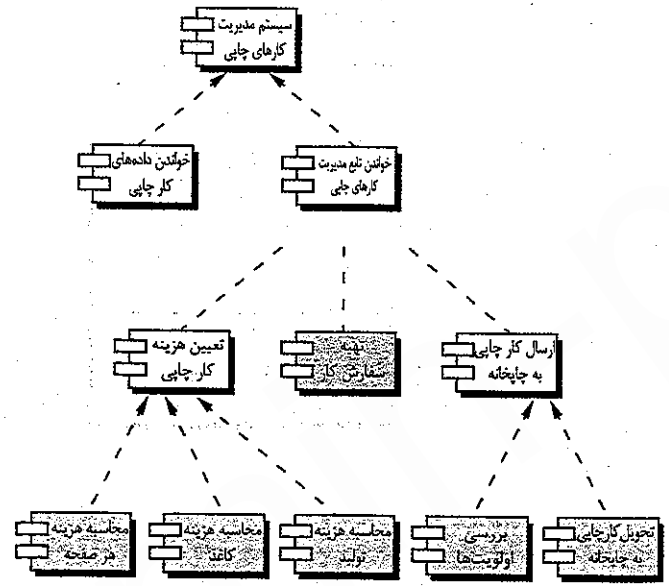
مؤلفه‌های نرم‌افزاری سنتی، همانند مؤلفه‌های شیءگرا از مدل تحلیل به‌دست می‌آیند. ولی در این مورد، عنصر جریان‌گرای مدل تحلیل به‌عنوان مبنایی برای به‌دست آوردن مؤلفه‌ها عمل می‌کند. هر

> «سیستم پیچیده‌ای که کار می‌کند بدون شک از سیستم ساده‌ای که کار می‌کرده است، تکامل پیدا کرده است.»
>
> جان گال

[1] خوانندگان ناآشنا با نمادگذاری UML باید به پیوست ۱ رجوع کنند.

تبدیل (حباب) که در پایین‌ترین سطح از نمودار جریان داده‌ها نمایش داده می‌شود، به سلسله مراتبی از پیمانه‌ها نگاشت می‌شود (بخش ۶-۹). مؤلفه‌ها (پیمانه‌های) کنترلی در نزدیکی بالای سلسله مراتب (معماری برنامه) و مؤلفه‌های دامنه‌ی مسأله بیشتر در پایین سلسله مراتب قرار می‌گیرند. برای دستیابی به پیمانه‌بندی اثربخش، مفاهیم طراحی از قبیل استقلال عملیاتی (فصل ۸) به‌عنوان یک مؤلفه بسط پیدا می‌کنند و جزئیات آن‌ها تعیین می‌شود.

برای نمایش این فرایندِ پرداختن به جزئیات طراحی برای مؤلفه‌های سنتی، دوباره همان نرم‌افزاری را در نظر بگیرید که قرار است برای چاپخانه ساخته شود. مجموعه‌ای از نمودارهای جریان داده‌ها طی مدل‌سازی خواسته‌ها به‌دست می‌آید. فرض کنید این نمودارها به یک معماری نگاشت می‌شود که در شکل ۲-۱۰ نشان داده شده است. هر چارگوش نشان‌گر مؤلفه‌ای از نرم‌افزار است. توجه دارید که چارگوش‌های خاکستری از نظر وظیفه هم‌ارز با عملیات‌های تعریف شده برای کلاس **PrintJob** هستند که در بخش ۱-۱-۱۰ بحث شد. ولی در این مورد، هر عملیات به‌عنوان یک پیمانه‌ی جداگانه نمایش داده می‌شود که به‌صورت نشان داده شده در شکل قابل فراخوانی است. سایر پیمانه‌ها برای کنترل پردازش به‌کار می‌روند و از این رو، مؤلفه‌های کنترلی هستند.



شکل ۲-۱۰ نمودار ساختاری برای یک سیستم سنتی.

طی طراحی در سطح مؤلفه‌ها، هر پیمانه در شکل ۲-۱۰ بسط داده می‌شود و بر جزئیات آن افزوده می‌شود. واسط پیمانه به صراحت تعریف می‌شود. یعنی، هر شیء داده‌ای یا کنترلی که از واسط جریان پیدا می‌کند، به نمایش در می‌آید. ساختمان داده‌های مورد استفاده در داخل پیمانه‌ها نیز تعریف می‌شوند. الگوریتمی که به پیمانه امکان می‌دهد تا وظیفه مورد نظر را انجام دهد، با استفاده از روش پالایش مرحله‌ای بحث شده در فصل ۸ طراحی می‌شود. رفتار پیمانه گاهی با استفاده از نمودار حالت نشان داده می‌شود.

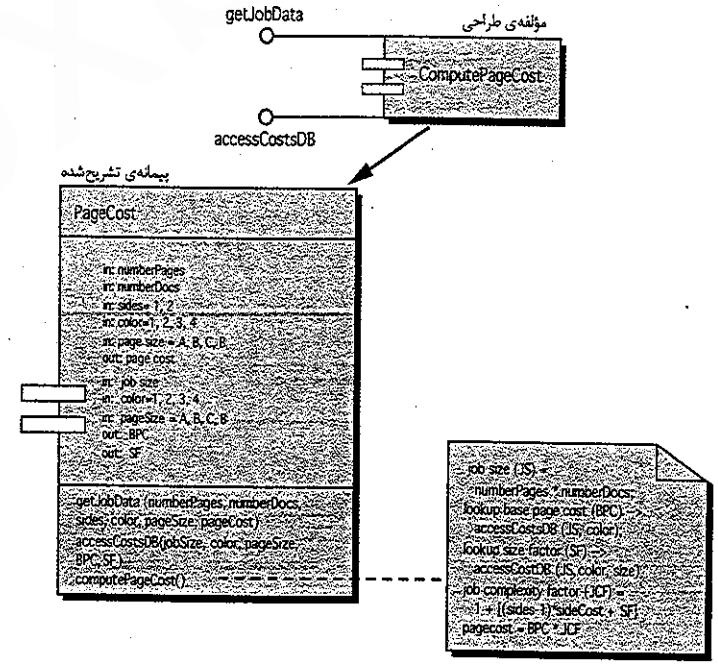
اندرز

به‌موازاتی که طراحی برای هر مؤلفه‌ی نرم‌افزار تشریح می‌شود، کانون توجه به سمت طراحی ساختمان داده‌ها و طراحی روال‌ها جابه‌جا می‌شود تا ساختمان داده‌ها دستکاری شود. ولی فراموش نکنید معماری‌ای که باید این مؤلفه‌ها با ساختمان داده‌های سازگاری راحت خود جای دهد، ممکن است به مؤلفه‌های فراوان سرویس دهد.

برای نشان دادن این فرایند، پیمانه‌ی *ComputePageCost* را در نظر بگیرید. هدف این پیمانه، محاسبه‌ی هزینه‌ی چاپ به ازای هر صفحه بر اساس مشخصات ارائه شده از سوی مشتری است. داده‌های مورد نیاز برای اجرای این وظیفه عبارتند از:

Number of pages in document ،total number of document to be produced ،one-or two-side printing ،color requirements و size requirements.

این داده‌ها از طریق واسط پیمانه به *computePageCost* تحویل می‌شوند. *computePageCost* از این داده‌ها برای تعیین هزینه صفحات بر اساس اندازه صفحه و پیچیدگی کار استفاده می‌کند- تابعی از همه‌ی این داده‌ها از طریق واسط به پیمانه تحویل می‌شود. هزینه‌ی صفحه با اندازه کار نسبت عکس و با پیچیدگی آن نسبت مستقیم دارد.



شکل ۳-۱۰ طراحی در سطح مؤلفه‌ها برای *computerPageCost*.

در شکل ۳-۱۰ طراحی در سطح مؤلفه‌ها با استفاده از نمادگذاری اصلاح شده‌ی UML نمایش داده شده است. پیمانه‌ی *computePageCost* با فرا خواندن پیمانه‌ی *getJobData* (که تحویل همه‌ی داده‌های مرتبط را به مؤلفه امکان پذیر می‌سازد) و یک واسط بانک اطلاعاتی *accessCostDB* (که دستیابی پیمانه به بانک اطلاعاتی حاوی تمامی هزینه‌های چاپی را فراهم می‌آورد) به داده‌ها دست پیدا می‌کند. با ادامه یافتن طراحی، پیمانه‌ی *computePageCost* حاوی جزئیات بیشتری در خصوص الگوریتم و واسط می‌شود (شکل ۳-۱۰). جزئیات الگوریتم را می‌توان به‌وسیله‌ی شبه کدهای متنی نشان داده شده در شکل یا با نمودارهای فعالیت UML به نمایش گذاشت. واسط‌ها به‌صورت مجموعه‌ای از اشیای ورودی و خروجی نشان داده می‌شود. بسط طراحی آن‌قدر ادامه پیدا می‌کند که جزئیات کافی برای ساخت مؤلفه‌ی مورد نظر فراهم آید.

۳-۱-۱۰ دیدگاه فرایندی

در دیدگاه‌های شیءگرا و سنتی طراحی در سطح مؤلفه‌ها که در بخش‌های ۱-۱-۱۰ و ۲-۱-۱۰ ارائه شد، فرض بر این است که مؤلفه از نقطه‌ی صفر ساخته می‌شود. یعنی، باید بر اساس مشخصه‌های به‌دست آمده از مدل خواسته‌ها، مؤلفه جدیدی ایجاد کنید. البته یک روش دیگر نیز وجود دارد.

طی دو دهه گذشته، جامعه‌ی مهندسی نرم‌افزار، بر ساخت سیستم‌هایی تأکید داشته است که از مؤلفه‌های نرم‌افزاری یا الگوهای طراحی موجود استفاده می‌کنند. در اصل، کاتالوگی از مؤلفه‌ها در سطح طراحی یا کد در حین طراحی در اختیار شما قرار داده می‌شود. با توسعه‌ی معماری نرم‌افزار، مؤلفه‌ها یا الگوهای طراحی را از کاتالوگ انتخاب می‌کنید و آن‌ها را در معماری خود جای می‌دهید. از آن جا که این مؤلفه‌ها با در نظر داشتن قابلیت استفاده‌ی مجدد ایجاد شده‌اند، توصیف کاملی از واسط آن‌ها، وظیفه (هایی) که انجام می‌دهند و ارتباطات و همکاری‌هایی که مورد نیاز آن‌هاست، در اختیار شما است. در بخش ۶-۱۰ به برخی جنبه‌های مهم مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها (CBSE) خواهیم پرداخت.

اطلاعات

چارچوب‌ها و استانداردهای مبتنی بر مؤلفه‌ها

یکی از عناصر کلیدی که به موفقیت یا شکست CBSE می‌انجامد، قابلیت دسترسی استانداردهای مبتنی بر مؤلفه‌هاست که گاهی میان‌افزار نامیده می‌شوند. میان‌افزار، مجموعه‌ای از مؤلفه‌های زیرساختی است که مؤلفه‌های دامنه‌ی مسأله را قادر می‌سازد تا روی یک شبکه یا داخل یک سیستم پیچیده امکان برقراری ارتباط می‌دهد. مهندسان نرم‌افزار که مایل به استفاده از توسعه‌ی مبتنی بر مؤلفه‌ها به‌عنوان فرایند نرم‌افزار خود هستند، می‌توانند از میان استانداردهای زیر یکی را انتخاب کنند:

OMG CORBA-www.corba.org/
Microsoft COM-www.microsoft.com/com/tech/complus.asp
Microsoft.NET-http://msdn2.microsoft.com/en-us/netframework/default.as/x
Sun Java Beans-http://java.sun.com/products/ejb/

وب‌سایت‌های ذکرشده، آرایه‌ی وسیعی از مطالب آموزشی، ابزارها، گزارش‌ها و منابع عمومی درخصوص این استانداردهای میان‌افزار ارائه شده است.

۲-۱۰ طراحی مؤلفه‌های مبتنی بر کلاس

چنان که قبلاً گفتیم، طراحی در سطح مؤلفه‌ها، از اطلاعات فراهم شده به‌عنوان بخشی از مدل خواسته‌ها (فصل‌های ۶ و ۷) و اطلاعات نمایش داده شده به‌عنوان بخشی از مدل معماری (فصل ۹)، بهره می‌برد. هنگامی که یک روش مهندسی نرم‌افزار شیءگرا انتخاب می‌شود، آن چه در طراحی در سطح مؤلفه‌ها کانون توجه قرار می‌گیرد، پرداختن به جزئیات کلاس‌های ویژه‌ی دامنه مسأله و تعریف و پالایش کلاس‌های زیرساختی موجود در مدل خواسته‌هاست. توصیف مشروحی از صفات، عملیات‌ها و واسط‌های مورد استفاده‌ی این کلاس‌ها، جزئیات طراحی لازم برای فعالیت ساخت شیء را فراهم می‌آورد.

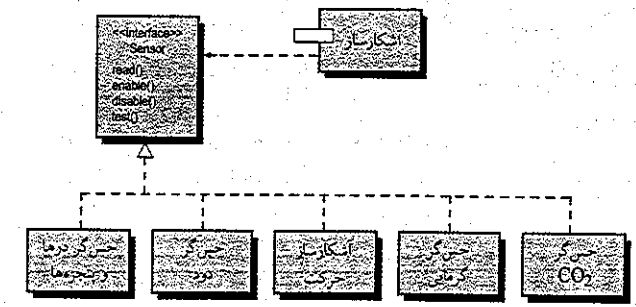
۱۰-۲-۱ اصول پایه‌ی طراحی

در طراحی در سطح مؤلفه‌ها از چهار اصل پایه طراحی استفاده می‌شود که هنگام به‌کارگیری مهندسی نرم‌افزار شیءگرا به‌طور گسترده پذیرفته می‌شوند. انگیزه اصلی برای به‌کارگیری این اصول، ایجاد طراحی‌هایی است که بیشتر مستعد تغییر باشند و انتشار اثرات جانبی ناشی از تغییرات را کاهش دهند. از این اصول می‌توانید به‌عنوان راهنمایی در توسعه هر مؤلفه‌ی نرم‌افزار استفاده کنید.

اصل باز-بسته (OCP). «یک پیمانه [مؤلفه] باید برای عمل بسط، باز و برای عمل اصلاح، بسته باشد.» [Mar00]. این عبارت ممکن است تناقض‌آمیز به‌نظر برسد، ولی یکی از مهمترین خصوصیات طراحی در سطح مؤلفه‌ها را نشان می‌دهد. به بیان ساده، باید مؤلفه را به قسمی مشخص کنید که بتوان آن را بسط داد (در دامنه‌ی عملیاتی مربوط)، بدون این که نیازی به انجام اصطلاحات داخلی (در سطح کدها یا منطق) در خود مؤلفه باشد. برای دستیابی به این هدف، انتزاع‌هایی ایجاد می‌کنید که میان قابلیت عملیاتی که احتمالاً باید بسط و گسترش داده شود و خود کلاس طراحی، به‌عنوان میانجی عمل می‌کند.

برای مثال، فرض کنید قابلیت امنیتی منزل در محصول *SafeHome* از کلاس **Detector** استفاده می‌کند که باید وضعیت هر نوع حس‌گر امنیتی را چک کند. این احتمال وجود دارد که با گذر زمان، تعداد و انواع حس‌گرهای امنیتی رشد پیدا کند. اگر منطق پردازش داخلی به‌صورت دنباله‌ای از ساختارهای if-then-else پیاده‌سازی شود و هر کدام به یک نوع حس‌گر متفاوت بپردازد، افزودن نوع جدیدی از حس‌گر مستلزم منطق داخلی اضافی (هنوز یک if-then-else) خواهد بود.

یک راه دستیابی به اصل OCP برای کلاس **Detector** در شکل ۴-۱۰ نشان داده شده است. واسط *sensor* نشان‌گر دیدگاهی سازگار از حس‌گرها در مؤلفه‌ی **Detector** است. اگر نوع جدیدی از حس‌گر افزوده شود، تغییری برای کلاس (مؤلفه) **Detector** مورد نیاز نیست. پس OCP برقرار است.



شکل ۴-۱۰ پیروی از OCP.

اصل جایگزینی لیسکوف (LSP). «*زیرکلاس‌ها باید با کلاس‌های پایه‌ی خود جایگزین‌پذیر باشند.*» [Mar00]. طبق این اصل طراحی، که اولین بار توسط باربارا لیسکوف [Lis88] پیشنهاد شد، مؤلفه‌ای که از یک کلاس پایه استفاده می‌کند، اگر به‌جای کلاس پایه، کلاس مشتق آن به مؤلفه ارسال شود، مؤلفه باید به‌درستی عمل کند. LSP حکم می‌کند که هر کلاس مشتق باید به قراردادهای میان کلاس پایه و مؤلفه‌ای که از آن استفاده می‌کند، بها دهد. «قرارداد» در این بحث، *پیش‌شرطی* است که باید درست باشد تا مؤلفه از یک کلاس پایه استفاده کند و *پس‌شرطی* است که باید پس از این که مؤلفه از

یک کلاس پایه استفاده می‌کند، درست باشد. هنگامی که کلاس‌های مشتق را ایجاد می‌کنید، اطمینان حاصل کنید که با پیش‌شرط‌ها و پس‌شرط‌ها همخوانی دارند.

SafeHome

OCP در عمل

صحنه: کابین وینود.

نقش‌آفرینان: وینود و شکیرا- اعضای تیم مهندسی نرم‌افزار *SafeHome*

گفتگو:

وینود: همین الان داگ [مدیر تیم] با من تماس گرفت. می‌گوید بخش بازاریابی می‌خواهد یک حس‌گر جدید اضافه کند.

شکیرا (با پوزخند): خدایا، دوباره؟

وینود: بله... و باورت نمی‌شود چه چیزی درست کرده‌اند.

شکیرا: دوست دارم بشنوم.

وینود (با خنده): اسمش را گذاشته‌اند *حس‌گر نگرانی سگی*.

شکیرا: یعنی چی؟

وینود: برای آدم‌هایی است که حیوان خانگی‌شان را در آپارتمان یا خانه‌هایی می‌گذارند که به خانه‌ها و آپارتمان‌های دیگر نزدیک است. سگ شروع به پارس کردن می‌کند. همسایه عصبانی می‌شود و شکایت می‌کند. ولی با این حس‌گرها، اگر سگ بیشتر از مثلاً یک دقیقه پارس کند، حس‌گر یک حالت هشدار را فعال می‌کند که به تلفن همراه صاحبخانه زنگ می‌زند.

شکیرا: شوخی می‌کنی، نه؟

وینود: نخیر. داگ می‌خواهد بداند چقدر وقت می‌گیرد تا یک قابلیت امنیتی دیگر اضافه کنیم.

شکیرا (لحظه‌ای می‌اندیشد): وقت زیادی نمی‌گیرد. ببین. [شکل ۴-۱۰ را به وینود نشان می‌دهد.] ما کلاس‌های واقعی حس‌گر را پشت واسط **Sensor** جدا کرده‌ایم. هر وقت مشخصات مربوط به حس‌گر سگی را داشته باشیم، اضافه کردن آن باید مثل آب خوردن باشد. تنها کاری که باید بکنم، ایجاد یک مؤلفه‌ی مناسب است... یعنی یک کلاس. مؤلفه‌ی **Detector** به هیچ وجه نباید تغییر کند.

وینود: پس به داگ می‌گویم که مشکل بزرگی وجود ندارد.

شکیرا: با شناختی که از داگ دارم، او از ما می‌خواهد که به کارمان ادامه بدهیم و این حس‌گر سگی را در اولویت بعدی تحویل دهیم.

وینود: این چیز بدی نیست، ولی اگر بخواهد الان هم می‌توانی آن را پیاده‌سازی کنی؟

شکیرا: بله، طراحی واسط طوری بوده که می‌توانم بدون هیچ مشکل خاصی این کار را انجام دهم.

وینود (لحظه‌ای می‌اندیشد): تا حالا از اصل باز-بسته چیزی شنیدی؟

شکیرا (شانه‌اش را بالا می‌اندازد): نه نشنیدم.

وینود (با لبخند): مشکلی نیست.

اصل وارونگی وابستگی (DIP). «*به انتزاع‌ها متکی باشید، به عینیت‌ها* (concretions) *متکی نباشید.*» [Mar00]. چنان‌که در بحث مربوط به OCP دیدیم، انتزاع‌ها نقاطی هستند که طراحی را از آن‌ها بدون پیچیدگی زیاد، بسط و گسترش می‌یابد. هرچه وابستگی یک مؤلفه به سایر مؤلفه‌های عینیت‌یافته بیشتر باشد، بسط و گسترش آن دشوارتر خواهد بود.

اصل جداسازی واسط‌ها (ISP). «*داشتن واسط‌های خاص کلاینت بسیار بهتر از یک واسط چندمنظوره است.*» [Mar00]. واسط‌های فراوانی وجود دارد که در آن‌ها چند مؤلفه‌ی کلاینت از عملیات‌هایی استفاده می‌کنند که توسط یک کلاس سرور منفرد فراهم می‌آیند. طبق اصل ISP باید برای سرویس‌دهی به هر دسته‌ی عمده از کلاینت‌ها یک واسط تخصص‌یافته ایجاد کنید. تنها آن دسته از عملیات‌هایی که به‌دسته‌ی خاصی از کلاینت‌ها مربوط می‌شوند باید در واسط مربوط به آن کلاینت مشخص شوند. اگر چند کلاینت به عملیات‌های یکسانی نیاز داشته باشند، این را باید در هر کدام از واسط‌های تخصص‌یافته مشخص کرد.

برای مثال، کلاس **FloorPlan** را در نظر بگیرید که برای قابلیت‌های عملیاتی امنیت منزل در محصول *SafeHome* به‌کار برده می‌شود (فصل ۶). برای قابلیت‌های پایشی و امنیتی، **FloorPlan** تنها طی فعالیت‌های پیکربندی به‌کار برده می‌شود و از عملیات‌های *placeDevice( )*، *showDevice( )*، *groupDevice( )* و *removeDevice( )* برای قرار دادن، نشان دادن، گروه‌بندی و حذف حسگرها از پلان همکف استفاده می‌کند. قابلیت عملیاتی پایش منزل از چهار عملیات ذکرشده برای امنیت استفاده می‌کند، ولی به عملیات‌های خاص برای مدیریت دوربین‌ها نیز نیاز دارد: *showFOV( )* و *showDevice ID( )*. از این رو، بنا به اصل ISP، مؤلفه‌های کلاینت از دو قابلیت عملیاتی *SafeHome* دارای واسط‌های تخصص یافته‌ای هستند که برای آن‌ها تعریف شده است. واسط مربوط به امنیت فقط شامل عملیات‌های *placeDevice( )*، *showDevice( )*، *groupDevice( )* و *removeDevice( )* می‌شود. واسط مربوط به پایش، علاوه بر عملیات‌های *placeDevice( )*، *ShowDevice( )*، *groupDevice( )* و *removeDevice( )*، عملیات‌های *showFOV( )* و *showDevice ID( )* را نیز در بر می‌گیرد.

گرچه اصول طراحی در سطح مؤلفه‌ها، راهنمای مفیدی فراهم می‌سازند، خود مؤلفه‌ها در خلاء وجود ندارند. در بسیاری موارد، تک تک مؤلفه‌ها یا کلاس‌ها در قالب چند زیرسیستم یا پکیج سازمان‌دهی می‌شوند. منطقی است که بپرسیم این فعالیت پکیج‌سازی چگونه باید انجام پذیرد. با پیشرفت طراحی، دقیقاً مؤلفه‌ها را چگونه باید سازمان‌دهی کرد؟ مارتین [Mar00] اصول پکیج‌سازی دیگری را پیشنهاد می‌کند که برای طراحی در سطح مؤلفه‌ها قابل استفاده‌اند.

اصل هم‌ارزی استفاده‌ی مجدد از نسخه‌ها (REP). «*استفاده‌ی مجدد، سنگ بنای ارائه‌ی نسخه‌های جدید است.*» [Mar00]. هنگامی که کلاس‌ها یا مؤلفه‌ها برای استفاده‌ی مجدد طراحی می‌شوند، میان سازندگان موجودیتی با قابلیت استفاده‌ی مجدد و کسانی که از آن استفاده می‌کنند، قرارداد نانوشته‌ای وجود دارد. سازنده متعهد می‌شود که یک سیستم کنترلی ایجاد کند که از نسخه‌های قدیمی‌تر آن موجودیت، پشتیبانی و نگهداری کند، در حالی که کاربران به آهستگی به آخرین نسخه‌ی موجود ارتقا پیدا می‌کنند. به‌جای پرداختن به هر کلاس به‌صورت انفرادی، غالباً توصیه می‌شود کلاس‌های قابل استفاده‌ی مجدد در قالب پکیج‌هایی گروه‌بندی شوند که به‌موازات تکامل نسخه‌های جدیدتر بتوان آن‌ها را مدیریت و کنترل کرد.

اندرز

اگر طراحی را توزیع و کدها را به قطعات تقسیم می‌کنید، فقط به‌خاطر داشته باشید که کدها «عینیت» نهایی‌اند. از DIP غلول نکنید.

نکته‌ی کلیدی

طراحی مؤلفه‌ها برای استفاده‌ی مجدد به چیزی بیش از طراحی خوب نیاز دارد. علاوه بر آن به سازوکارهای کنترل پیکربندی از بخش هم نیاز است (فصل ۲۲).

اصل بستار مشترک (CCP). «*کلاس‌هایی که با هم تغییر می‌کنند به هم تعلق دارند.*» [Mar00]. کلاس‌ها باید به‌طور مناسب در پکیج قرار گیرند. یعنی هنگامی که کلاس‌ها به‌عنوان بخشی از طراحی در پکیج‌ها قرار می‌گیرند، باید نواحی رفتاری و عملکردی یکسانی را اداره کنند. وقتی قرار باشد ویژگی‌های آن ناحیه تغییر کند، نباید به تغییر کلاس‌های موجود در آن پکیج نیازی باشد. به‌این ترتیب، کنترل و مدیریت نسخه‌ها بهتر انجام می‌گیرد.

اصل استفاده‌ی مجدد مشترک (CCP). «*کلاس‌هایی که با هم دوباره استفاده نمی‌شوند، نباید در یک پکیج قرار گیرند.*» [Mar00]. هنگامی که یک یا چند کلاس در یک پکیج تغییر می‌کنند، شماره‌ی نسخه‌ی پکیج تغییر می‌کند. همه‌ی کلاس‌ها یا پکیج‌های وابسته به پکیج تغییریافته، اکنون باید به آخرین نسخه‌ی آن پکیج بهنگام شوند و مورد آزمون قرار گیرند تا اطمینان حاصل شود که نسخه‌ی جدید بدون هیچ اتفاق خاصی عمل می‌کند. اگر کلاس‌ها به‌صورت یکپارچه گروه‌بندی نشده باشند، این امکان وجود دارد که کلاسی که با کلاس‌های دیگر موجود در پکیج رابطه ندارد، تغییر کند. این کار باعث آزمون‌های بیهوده می‌شود. به همین دلیل، تنها کلاس‌هایی که با یکدیگر استفاده می‌شوند باید در یک پکیج گنجانده شوند.

۲-۲-۱۰ دستورالعمل‌های طراحی در سطح مؤلفه‌ها

علاوه بر اصول بحث شده در بخش ۱-۲-۱۰، یک سری دستورالعمل‌های طراحی را نیز می‌توان به‌موازات پیشرفت طراحی در سطح مؤلفه‌ها به‌کار گرفت. این دستورالعمل‌ها برای مؤلفه‌ها، واسط‌های آن‌ها و خصوصیات وراثتی و وابستگی‌ای به‌کار برده می‌شوند که بر طراحی حاصل تأثیر دارند. امبلر [Amb2b] دستورالعمل‌های زیر را پیشنهاد می‌کند.

مؤلفه‌ها. برای مؤلفه‌هایی که به‌عنوان بخشی از مدل معماری مشخص می‌شوند و سپس به‌عنوان بخشی از مدل‌سازی در سطح مؤلفه‌ها پالایش می‌شوند و بر جزئیات آن‌ها افزوده می‌شود، قراردادهای نامگذاری مورد نیاز است. نام‌های مولفه‌های معماری باید از دامنه مسأله مشتق شوند و برای همه‌ی طرف‌های ذی‌نفع که مدل معماری را مشاهده می‌کنند، معنی داشته باشد. برای مثال، نام کلاس **FloorPlan** برای هر کس که آن را بخواند با هر دانش فنی که داشته باشد، معنی دارد. از طرف دیگر، مؤلفه‌های زیرساختی یا کلاس‌های سطح مؤلفه‌ای با جزئیات کافی باید طوری نامگذاری شوند که معنی مرتبط با پیاده‌سازی را انعکاس دهند. اگر قرار باشد فهرستی مرتبط به‌عنوان بخشی از پیاده‌سازی **FloorPlan** مدیریت شود، عملیاتی با نام *manageList( )* مناسب است، حتی اگر این امکان وجود داشته باشد که یک فرد فنی آن را سوء تعبیر کند.[1]

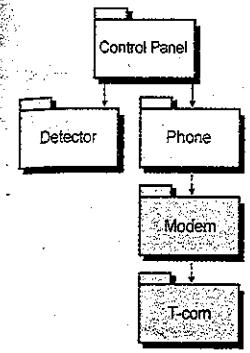
می‌توانید از یک سری کلیشه برای کمک به شناسایی ماهیت مؤلفه‌ها در سطح طراحی مشروح استفاده کنید. برای مثال، <<infrastructure>> را می‌توان برای شناسایی یک مؤلفه زیرساختی استفاده کرد، از <<database>> می‌توان برای شناسایی بانک اطلاعاتی‌ای استفاده کرد که به یک یا چند کلاس طراحی یا کل سیستم، سرویس می‌دهد؛ از <<table>> می‌توان برای شناسایی جدولی در یک بانک اطلاعاتی استفاده کرد.

هنگام نامگذاری مؤلفه‌ها چه نکاتی را باید مد نظر داشت؟

[1] احتمال این که کسی از سازمان بازاریابی یا مشتریان (یک نوع غیر فنی) اطلاعات طراحی مشروح را بررسی کند، زیاد نیست.

واسطها. واسطها اطلاعات مهمی درباره ارتباطات و همکاری فراهم می‌آورند (و همچنین ما را در دستیابی به OCP یاری می‌دهند). ولی، نمایش لجام گسیخته‌ی واسطها باعث پیچیده شدن نمودارهای مؤلفه‌ها می‌شود. امبلر [Amb02c] توصیه می‌کند که (۱) در صورت رشد کردن و پیچیده شدن نمودار باید از نمایش آب نبات چوبی به‌جای نمایش رسمی‌تر چهار گوش و پیکان مقطع UML برای واسطها استفاده کرد؛ (۲) برای سازگاری، واسطها باید از طرف چپ، وارد چهارگوش مؤلفه شوند؛ (۳) تنها آن واسطهایی که به مؤلفه‌ی مورد نظر مربوط می‌شوند، باید نشان داده شوند، حتی اگر واسطهای دیگری در دسترس باشند. این توصیه‌ها به منظور ساده‌سازی ماهیت بصری نمودارهای مؤلفه‌های UML ارائه شده‌اند.

وابستگی‌ها و وراثت. برای بهبود قابلیت خواندن، مدل‌سازی وابستگی‌ها از چپ به راست و مدل‌سازی وراثت از پایین (کلاس‌های مشتق) به بالا (کلاس‌های پایه) ایده‌ی خوبی است. به‌علاوه، وابستگی‌های میان مؤلفه‌ها را باید از طریق واسطها نمایش داد نه با نمایش وابستگی مؤلفه به مؤلفه. دنبال کردن فلسفه‌ی OCP، به شما کمک می‌کند سیستم را با قابلیت نگهداری بیشتری بسازید.



شکل ۵-۱۰ یکپارچگی لایه‌ای

۳-۲-۱۰ یکپارچگی (Cohession)

در فصل ۸، یکپارچگی را به‌عنوان «وحدت رأی» مؤلفه توصیف کردیم. در حیطه‌ی طراحی در سطح مؤلفه‌ها برای سیستم‌های شیءگرا، یکپارچگی بدان معناست که یک مؤلفه یا کلاس فقط صفات و عملیات‌هایی را در خود پنهان‌سازی می‌کند که رابطه‌ای تنگاتنگ با یکدیگر و با خود کلاس یا مؤلفه دارند. لتبرج و لاگانیه [Let01] چند نوع متفاوت از یکپارچگی را تعریف می‌کنند (که در زیر به‌ترتیب سطح یکپارچگی فهرست شده‌اند[1]):

عملیاتی: این سطح از یکپارچگی که اساساً به وسیله‌ی عملیات‌ها نشان داده می‌شود، هنگامی رخ می‌دهد که مؤلفه‌ای یک محاسبه‌ی هدف‌دار انجام دهد و سپس نتیجه‌ای را برگرداند.

لایه‌ای. این نوع یکپارچگی، که به‌وسیله‌ی پکیج‌ها، مؤلفه‌ها و کلاس‌ها به نمایش گذاشته می‌شود، هنگامی رخ می‌دهد که یک لایه بالایی به سرویس‌های لایه پایانی دستیابی دارد، ولی لایه پایینی به لایه بالایی دستیابی ندارد. برای مثال، این خواسته را برای قابلیت امنیتی منزل در محصول *SafeHome* در نظر بگیرید که در صورت فعال شدن یک حس‌گر، با بیرون تماس می‌گیرد. ممکن است تعریف مجموعه‌ای از پکیج‌های لایه‌بندی شده به‌صورت نشان داده شده در شکل ۵-۱۰ امکان‌پذیر باشد. پکیج‌های خاکستری حاوی مؤلفه‌های زیرساختی‌اند. دستیابی از پکیج ControlPanel به طرف پایین است.

ارتباطاتی. همه‌ی عملیات‌هایی که به داده‌های یکسان دستیابی دارند، تنها در یک کلاس تعریف می‌شوند. به‌طور کلی، در این گونه کلاس‌ها فقط داده‌های مورد نظر، دستیابی به آن‌ها و ذخیره‌سازی آن‌ها کانون توجه قرار می‌گیرد.

پیاده‌سازی، آزمون و نگهداری کلاس‌ها و مؤلفه‌هایی که یکپارچگی عملیاتی، لایه‌ای و ارتباطاتی را از خود به نمایش می‌گذارند، نسبتاً آسان است. باید درصورت امکان تلاش کنید به این سطوح

اندرز

گرچه درک سطوح مختلف یکپارچگی، آموزنده است، مهمتر این است که حین طراحی مؤلفه‌ها از این مفهوم کلی آگاه باشید. یکپارچگی را تا حد امکان در سطحی بالا حفظ کنید.

[1] به‌طور کلی، هر چه سطح هم بندی بالاتر باشد، پیاده‌سازی مؤلفه، آزمودن آن و نگهداری از آن آسان‌تر است.

SafeHome

یکپارچگی در عمل

صحنه: کابین جیمی

نقش‌آفرینان: جیمی و اد- اعضای تیم مهندسی نرم‌افزار *SafeHome* که روی قابلیت عملیاتی پایش منزل کار می‌کنند.

گفتگو:

اد: من اولین دور طراحی مولفه‌ی **camera** را تمام کردم.

جیمی: دوست داری نگاهی به آن بیندازیم؟

اد: گمان کنم به اظهار نظرت احتیاج دارم-

(جیمی اشاره می‌کند که ادامه دهد)

اد: ما اول پنج عملیات برای **camera** تعریف کردیم. ببین-

Determine() نوع دوربین را به من می‌گوید.

translateLocation() به من این امکان را می‌دهد که دوربین را حول نقشه منزل حرکت بدهم.

displayID() شماره‌ی ID دوربین را می‌گیرد و آن را نزدیک به آیکون دوربین نمایش می‌دهد.

displayView() میدان دید دوربین را به شیوه‌ی گرافیکی به من نشان می‌دهد.

displayZoom() درشت‌نمایی دوربین را به شیوه‌ی گرافیکی به من نشان می‌دهد.

اد: من هر کدام را جداگانه طراحی کرده‌ام و این‌ها عملیات‌های بسیار ساده‌ای هستند. بنابراین، فکر کردم شاید بد نباشد همه‌ی عملیات‌ها را فقط در یک عملیات به نام *displayCamera()* نشان بدهم- که ID نما و درشت‌نمایی را نشان دهد. نظر تو چی است؟

جیمی: مطمئن نیستم. فکر خوبی باشد.

اد (با اخم): چرا؟ همه‌ی این عملیات‌های کوچک باعث سر درد می‌شوند.

جیمی: مشکل ترکیب آن‌ها این است که یکپارچگی را از دست می‌دهیم. یعنی عملیات *displayCamera()* وحدت رأی ندارد.

اد (قدری برآشفته است): خوب که چی؟ کل این‌ها حداکثر صد خط می‌شود. فکر کنم پیاده‌سازی‌اش آسان‌تر باشد.

جیمی: و اگر بازاریابی تصمیم بگیرد که روش میدان دید را تغییر بدهیم؟

اد: کافی است *displayCamera()* را بیاوریم و اصلاح لازم را انجام بدهیم.

جیمی: اثرات جانبی چه می‌شود؟

اد: منظورت چیست؟

جیمی: خب، مثلاً تغییر را اعمال می‌کنی، ولی در نمایش ID یک مشکل پیش می‌آید.

اد: من آنقدرها هم بی‌دقت نیستم.

جیمی: ممکن است، ولی اگر یک نفر دو سال بعد بخواهد این تغییر را اعمال کند؟ ممکن است این عملیات را مثل تو نفهمد و چه کسی می‌داند، شاید او بی‌دقت باشد.

اد: پس مخالفی؟

جیمی: طراح، تو هستی... تصمیم‌گیری با خودت است... فقط مطمئن شو که عواقب یکپارچگی را می‌دانی.

اد (لحظه‌ای می‌اندیشد): شاید عملیات‌های نمایشی را جدا کنیم.

جیمی: تصمیم خوبی است.

یکپارچگی برسید. به هر حال، شایان ذکر است که اصول عملی طراحی و پیاده‌سازی، شما را گاهی وادار به گزینش سطوح پایین‌تر یکپارچگی می‌کنند.

۱۰-۲-۴ اتصال (Coupling)

در بحث قبلی درباره تحلیل و طراحی، متذکر شدیم که ارتباطات و همکاری، عناصر اساسی هر سیستم شیءگرا هستند. ولی این خصوصیت مهم (و ضروری) یک وجه تاریک نیز دارد. با افزایش مقدار ارتباطات و همکاری‌ها (یعنی با بالا رفتن درجه‌ی «اتصال» کلاس‌ها)، بر پیچیدگی سیستم نیز افزوده می‌شود. با افزایش پیچیدگی، پیاده‌سازی، آزمون و نگهداری نرم‌افزار نیز دشوارتر می‌شود.

اتصال، میزانی کیفی از درجه اتصال کلاس‌ها به یکدیگر است. با وابستگی بیشتر کلاس‌ها (و مؤلفه‌ها) به یکدیگر، اتصال افزایش پیدا می‌کند. یک هدف مهم در طراحی در سطح مؤلفه‌ها، حفظ اتصال در حداقل سطح ممکن است.

اتصال کلاس‌ها می‌تواند خود را به شیوه‌های گوناگون نشان دهد. لتبریج و لاگانیه [Let01] گروه‌های زیر را برای اتصال تعریف می‌کنند:

اتصال محتوا: هنگامی رخ می‌دهد که یک مؤلفه «در خفا داده‌هایی را اصلاح می‌کند که در داخل مؤلفه‌ای دیگر قرار دارند» [Let01]. این امر، عدول از پنهان‌سازی اطلاعات است که مفهومی اساسی در طراحی به شمار می‌رود.

اتصال مشترک. هنگامی رخ می‌دهد که چند مؤلفه، همگی از یک متغیر سراسری استفاده کنند. گرچه این گاهی ضرورت پیدا می‌کند (مثلاً برای برقراری مقادیر پیش‌فرض که در سراسر یک برنامه‌ی کاربردی قابل استفاده اند)، اتصال مشترک می‌تواند به انتشار خطای کنترل نشده و اثرات جانبی پیش‌بینی نشده در هنگام اعمال تغییرات بینجامد.

اتصال کنترل. هنگامی رخ می‌دهد که عملیات *A()* عملیات *B()* را فراخوانی کند و یک نشانه‌ی کنترل را به *B* تحویل می‌دهد. از این رو، نشانه کنترلی جریان منطقی را در داخل *B* «هدایت می‌کند». مشکل این نوع اتصال آن است که تغییر بی‌ربطی در *B* می‌تواند به تغییر در معنی نشانه‌ی کنترلی‌ای بینجامد که *A* به آن تحویل می‌دهد. اگر به این امر توجه کافی نشود، خطایی رخ خواهد داد.

اتصال مُهری (Stamp Coupling). هنگامی رخ می‌دهد که **ClassB** به‌عنوان نوع آرگومان یکی از عملیات‌های **ClassA** اعلام شود. چون **ClassB** اکنون بخشی از تعریف **ClassA** است، اصلاح سیستم، پیچیده‌تر می‌شود.

اندرز

به‌موازاتی که طراحی برای هر مؤلفه‌ی نرم‌افزار تشریح می‌شود، کانون توجه به سمت طراحی ساختمان داده‌ها و طراحی روال‌ها جابه‌جا می‌شود تا ساختمان داده‌ها دستکاری شود. ولی فراموش نکنید معماری‌ای را که باید این مؤلفه‌ها یا ساختمان داده‌های سراسری را در خود جای دهد، ممکن است به مؤلفه‌های فراوان سرویس دهد.

SafeHome

اتصال در عمل

صحنه: کابین شکیرا

نقش‌آفرینان: شکیرا و وینود- اعضای تیم نرم‌افزار *SafeHome* که روی قابلیت امنیتی منزل کار می‌کنند.

گفتگو:

شکیرا: فکر می‌کردم ایده خیلی خوبی به ذهنم رسیده... بعدش کمی درباره آن فکر کردم و به‌نظرم رسید که انگار خیلی هم ایده خوبی نیست. دست آخر هم ردش کردم، ولی گفتم قبلاً نظر تو را هم بدانم.

وینود: حتماً ایده‌ات چه بود؟

شکیرا: خب، هر کدام از حسگرها یک نوع شرایط هشدار را تشخیص می‌دهد، نه؟

وینود (با لبخند): به همین خاطر هم به آن‌ها حسگر می‌گویند.

شکیرا (برآشفته): طعنه وینود تو باید یک کم روی مهارت‌های اجتماعی‌ات کار کنی.

وینود: داشتی می‌گفتی...

شکیرا: بسیار خب، به هر حال من به این نتیجه رسیدم که چرا در داخل هر شیء حسگر، یک عملیات با نام *makeCall()* ایجاد نکنیم که به‌طور مستقیم با مؤلفه‌ی **OutgoingCall** کار کند و خب، یک واسط هم با مؤلفه **OutgoingCall** داشته باشد.

وینود (اندیشناک): منظورت این است که به‌جای این که همکاری خارج از مؤلفه‌ی مثل **ControlPanel** رخ بدهد؟

شکیرا: بله... ولی بعدش به خودم گفتم این یعنی این که هر شیء حسگری به مؤلفه **OutgoingCall** متصل خواهد شد و... خب، فکر کردم این خودش باعث پیچیده شدن اوضاع می‌شود.

وینود: در این مورد خاص، ایده‌ی بهتر همین است که بگذاریم واسط هر حسگر اطلاعات را به **ControlPanel** تحویل دهد و تماس با خارج را به عهده آن بگذارد. به‌علاوه، حسگرهای متفاوت ممکن است به تماس‌های تلفنی با شماره‌های متفاوت نیاز داشته باشند، تو که نمی‌خواهی حسگر این اطلاعات را در خودش ذخیره کند، چون اگر تغییر کند...

شکیرا: احساس می‌کردم درست در نیاید.

وینود: ابتکار در طراحی برای اتصال، به ما می‌گوید که درست نیست.

شکیرا: حالا هر چی...

اتصال داده‌ای. هنگامی رخ می‌دهد که عملیات‌ها، رشته‌های طولانی از آرگومان‌ها را ارسال می‌کنند. «پهنای باند» ارتباطات میان کلاس‌ها و مؤلفه‌ها رشد می‌کند و پیچیدگی واسط افزایش می‌یابد. آزمون و نگهداری دشوارتر می‌شود.

اتصال فراخوانی روال‌ها. هنگامی رخ می‌دهد که عملیاتی یک عملیات دیگر را فراخوانی می‌کند. این سطح اتصال، رایج و غالباً لازم است. به هر حال، میزان اتصال را در سیستم بالا می‌برد.

اتصال استفاده از نوع داده. هنگامی رخ می‌دهد که مولفه‌ی A از نوع داده‌ی تعریف‌شده در مؤلفه‌ی B استفاده می‌کند (مثلاً هنگامی پیش می‌آید که «یک کلاس، یک متغیر نمونه یا متغیر محلی را از نوع کلاس دیگری اعلان می‌کند» [Let01]). اگر نوع تعریف تغییر کند، هر مؤلفه‌ای که از این تعریف استفاده می‌کند نیز باید تغییر کند.

اتصال واردات یا شمول (Inclusion or Import Couplings). هنگامی رخ می‌دهد که مؤلفه‌ی A پکیج یا محتوای مؤلفه‌ی B را وارد کند یا شامل آن می‌شود.

اتصال خارجی (External Coupling). هنگامی رخ می‌دهد که مؤلفه‌ای با مؤلفه‌های زیرساخت (مثلاً، توابع سیستم عامل، قابلیت بانک اطلاعاتی، توابع مخابراتی) ارتباط برقرار کند یا همکاری داشته باشد. گرچه این نوع اتصال ضروری است، باید به تعداد کوچکی از مؤلفه‌ها یا کلاس‌های درون یک سیستم محدود گردد.

نرم‌افزار باید دارای ارتباط داخلی و خارجی ارتباط باشد. بنابراین، اتصال یک واقعیت زندگی است. ولی، طراح باید بکوشد تا هرگاه که امکان داشت، اتصال را کاهش دهد و هرگاه امکان پرهیز از آن وجود نداشت، پیامدهای ناگوار آن را بشناسد.

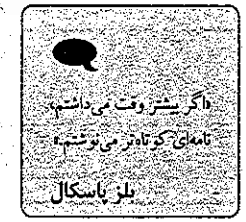
۱۰-۳ اجرای طراحی در سطح مولفه‌ها

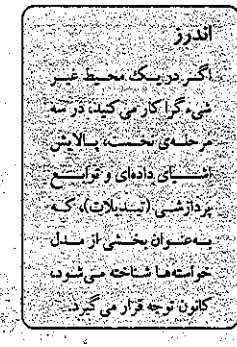
پیش از این، در همین فصل متذکر شدیم که طراحی در سطح مؤلفه‌ها ماهیتی پیچیده دارد. شما باید اطلاعات را از مدل‌های خواسته‌ها و معماری، به یک نمایش طراحی تبدیل کنید که جزئیات کافی برای راهنمایی در فعالیت ساخت (کدنویسی و آزمون) فراهم می‌سازد. مراحلی که به دنبال خواهد آمد، مجموعه‌ای از وظایف متداول برای طراحی در سطح مؤلفه‌ها در سیستم‌های شیءگراست.

مرحله ۱. همه‌ی کلاس‌های متناظر با دامنه‌ی مسأله را شناسایی کنید. با استفاده از مدل خواسته‌ها و مدل معماری، به هر کلاسِ تحلیل و مؤلفه‌ی معماری، جزئیات شرح داده شده در بخش ۱-۱-۱۰ افزوده می‌شود.

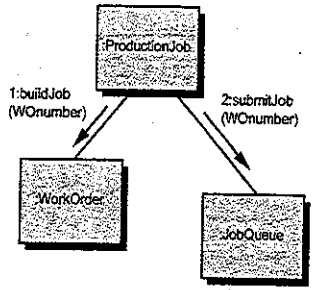
مرحله ۲. همه‌ی کلاس‌های طراحی متناظر با دامنه زیرساختار را شناسایی کنید. این کلاس‌ها در مدل خواسته‌ها توصیف نمی‌شوند و غالباً جای آن‌ها در مدل معماری نیز خالی است، ولی باید در این نقطه آن‌ها را توصیف کرد. چنان که قبلاً متذکر شدیم، کلاس‌ها و مؤلفه‌های این گروه شامل مؤلفه‌های GUI (که غالباً به‌صورت مؤلفه‌های قابل استفاده‌ی مجدد در دسترس قرار دارند)، مؤلفه‌های سیستم عامل و مؤلفه‌های مدیریت داده‌ها و اشیا می‌شوند.

مرحله ۳. جزئیات همه‌ی کلاس‌هایی را که به‌عنوان مؤلفه‌های قابل استفاده‌ی مجدد به‌دست نمی‌آیند، تعیین کنید. تعیین جزئیات ایجاب می‌کند که همه‌ی واسط‌ها، صفات و عملیات‌های لازم برای پیاده‌سازی کلاس به تفصیل توصیف شوند. ابتکار طراحی (مثلاً یکپارچگی و اتصال بالا) را باید در انجام این وظیفه مدّ نظر داشت.





مرحله‌ی ۳ الف. جزئیات پیام‌ها را برای همکاری کلاس‌ها و مؤلفه‌ها مشخص کنید. مدل خواسته‌ها برای نشان دادن چگونگی همکاری کلاس‌های تحلیل با یکدیگر، از یک نمودار همکاری استفاده می‌کند. با پیشرفت طراحی در سطح مؤلفه‌ها، گاهی نشان دادن جزئیات این همکاری‌ها با مشخص کردن ساختار پیام‌هایی که بین اشیای موجود در یک سیستم تبادل می‌شوند، مفید واقع می‌شود. گرچه این فعالیت طراحی، اختیاری است، از آن می‌توان به‌عنوان پیش ماده‌ای برای مشخص کردن واسط‌هایی استفاده کرد که نشان می‌دهند مؤلفه‌های داخل سیستم چگونه با هم ارتباط برقرار می‌کنند و همکاری دارند.



شکل ۶-۱۰ نمودار همکاری همراه با مبادله‌ی پیام.

شکل ۶-۱۰ یک نمودار همکاری ساده برای سیستم چاپی را نشان می‌دهد که قبلاً بحث شد. سه شیء، **ProductionJob**، **WorkOrder** و **JobQue** با یکدیگر همکاری می‌کنند تا کار چاپی را برای ارائه به خط تولید آماده کنند. پیام‌ها با پیکان‌های موجود در شکل مبادله می‌شوند. طی مدل‌سازی خواسته‌ها، پیام‌ها به‌صورت نشان داده شده در شکل مشخص می‌شوند. ولی با پیشرفت طراحی، هر پیام با بسط دادن قالب نحوی آن، به شیوه‌ی زیر جزئیات بیشتری کسب می‌کند [Ben02]:

[guard condition] sequence expression (return value):=

message name (argument list)

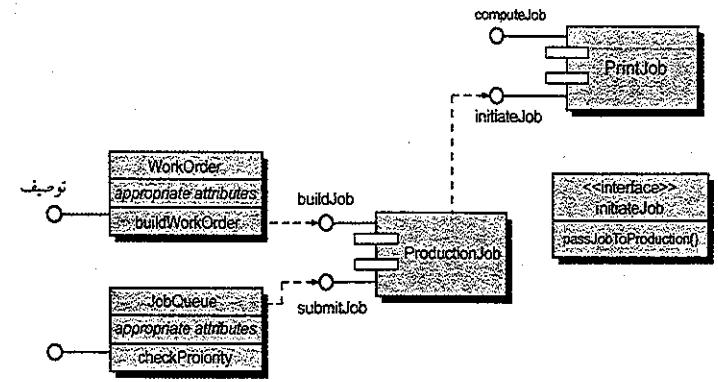
که *[guard condition]* به زبان قیدوبند اشیا (OCL)[1] نوشته می‌شود و هر مجموعه از شرایطی را که باید پیش از امکان ارسال پیام برقرار باشد، مشخص می‌کند؛ *sequence expression* یک مقدار عددی صحیح (یا هر شاخص ترتیب دیگر مثل 3.1.2) است که ترتیب ارسال پیام را مشخص می‌سازد؛ *(return value)* نام اطلاعاتی است که عملیات فراخوانده شده توسط پیام، آن را بر می‌گرداند؛ *message name* عملیاتی را مشخص می‌کند که باید فراخوانده شود و *(argument list)* فهرست صفاتی است که به عملیات ارسال می‌شوند.

مرحله ۳ب. برای هر مؤلفه، واسط‌های مناسب مشخص کنید. در حیطه‌ی طراحی در سطح مؤلفه‌ها، واسط UML «گروهی از عملیات‌هاست که از بیرون (یعنی از دید عموم) قابل مشاهده است. این واسط حاوی هیچ ساختار داخلی نیست، هیچ صفتی ندارد، و با چیزی وابستگی ندارد...» [Ben02].

[1] Object Constraint Language به اختصار در پیوست ۱ شرح داده شده است.

به بیان رسمی‌تر، واسط هم‌ارز، یک کلاس انتزاعی است که در شکل ۱-۱۰ نشان داده شد. در اصل، عملیات‌های تعریف شده برای کلاس طراحی، در یک یا چند کلاس انتزاعی گروه‌بندی می‌شوند. هر عملیات در کلاس انتزاعی (واسط) باید یکپارچه باشد؛ یعنی باید پردازشی را نشان دهد که یک تابع یا زیرتابع محدود شده را مورد توجه قرار می‌دهد.

با رجوع به شکل ۱-۱۰، می‌توان استدلال کرد که واسط *initiateJob* یکپارچگی کافی از خود نشان نمی‌دهد. در واقع، این رابطه سه زیر تابع متفاوت تعریف می‌کند- ساخت ترتیب کاری، چک‌کردن اولویت کارها و تحویل کار به خط تولید. طراحی واسط باید بازآرایی شود. یک روش می‌تواند بررسی دوباره‌ی کلاس‌های طراحی و تعریف کلاس جدید **WorkOrder** باشد که همه‌ی فعالیت‌های مرتبط با ترتیب کارها را بر عهده می‌گیرد. عملیات *buildWorkOrder( )* بخشی از آن کلاس می‌شود. به‌طور مشابه، می‌توانیم کلاسی با نام **JobQueue** را تعریف کنیم که شامل عملیات *checkPriority( )* می‌شود. کلاس **ProductionJob** شامل همه‌ی اطلاعات مرتبط با یک کار تولیدی می‌شود که باید به خط تولید تحویل شوند. سپس واسط *initiateJob* به‌صورتی در می‌آید که در شکل ۷-۱۰ نشان داده شده است. اکنون واسط *initiateJob* یکپارچه است و تنها یک قابلیت عملیاتی را مورد توجه قرار می‌دهد. واسط‌های مرتبط با **ProductionJob**، **WorkOrder** و **JobQueue** به‌طور مشابه وحدت رأی دارند.



شکل ۷-۱۰ بازآرایی واسط‌ها و تعاریف کلاس‌ها برای PrintJob.

مرحله ۳پ. جزئیات صفت‌ها و انواع داده‌ها و ساختمان داده‌های مورد نیاز برای پیاده‌سازی آن‌ها تعریف می‌شوند. به‌طور کلی، ساختمان داده‌ها و انواع مورد استفاده برای تعریف صفات، در زبان برنامه‌نویسی‌ای تعیین می‌شوند که قرار است برای پیاده‌سازی از آن استفاده شود. نوع داده‌ی یک صفت در UML با فرمت نحوی زیر تعریف می‌شود:

Name: type-expression= initial value {property string}

که *name* نام صفت، *type expression* نوع داده، *initial value* مقدار صفت هنگام ایجاد شیء و *property string* خاصیت یا کمیتی از صفت را تعریف می‌کند.

طی نخستین دور تکرار طراحی در سطح مؤلفه‌ها، صفات معمولاً با نام خود توصیف می‌شوند. یک بار دیگر با رجوع به شکل ۱-۱۰، می‌بینید که فهرست صفات PrintJob تنها حاوی نام صفات است. ولی با پیشرفت تعیین جزئیات طراحی، هر صفت با استفاده از فرمت نحوی UML تعریف خواهد شد. برای مثال، *paperType-weight* به‌شیوه‌ی زیر تعریف خواهد شد:

paperType-weight: string= "A" {contains 1 of 4 values-A,B,C, or D}

در اینجا *paperType-weight* به‌عنوان یک متغیر رشته‌ای تعریف می‌شود که مقدار اولیه‌ی A به آن داده شده است و می‌تواند یکی از چهار مقدار را از مجموعه {A,B,C,D} به خود بگیرد.

اگر صفتی به‌صورت مکرر در چند کلاس طراحی ظاهر شود و ساختار نسبتاً پیچیده‌ای داشته باشد، بهترین راه، ایجاد یک کلاس مجزا برای آن صفات است.

مرحله ۳ ت. توصیف مشروح جریان پردازش در هر عملیات. برای این منظور می‌توان از شبه کدهای مبتنی بر یک زبان برنامه‌نویسی یا نمودار فعالیت UML استفاده کرد. هر مؤلفه‌ی نرم‌افزار از طریق چند دور تکرار بسط داده می‌شود که در آن‌ها از مفهوم پالایش مرحله‌ای (فصل ۸) استفاده خواهد شد.

در نخستین دور تکرار، هر عملیات به‌عنوان بخشی از کلاس طراحی تعریف می‌شود. در هر حال، این عملیات باید به گونه‌ای مشخص شود که مشوق یکپارچگی بالا باشد؛ یعنی عملیات باید یک تابع یا زیر تابع با هدفی یگانه باشد. در دور بعدی تکرار، کاری بیش از بسط دادن نام عملیات انجام می‌شود. برای مثال، عملیات *computePaperCost( )* ذکر شده در شکل ۱-۱۰ را می‌توان به شیوه‌ی زیر بسط داد:

computePaperCost (weight,size,color): numeric

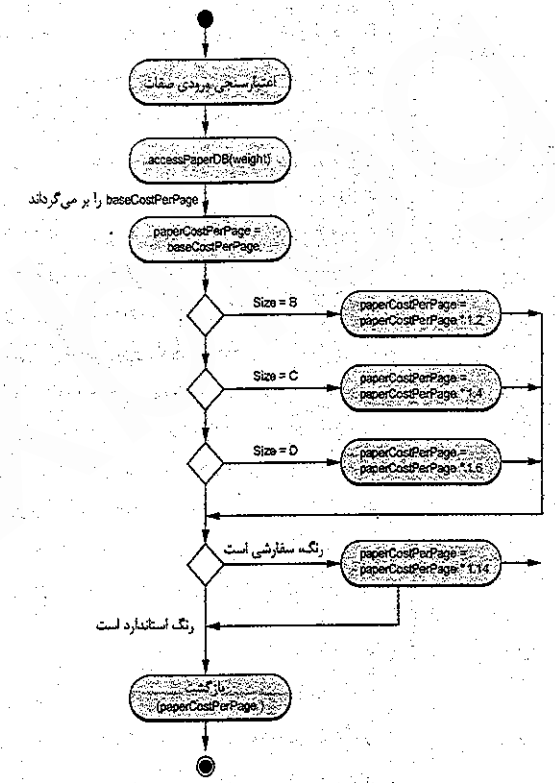
این نشان می‌دهد که *computePaperCost( )* نیاز به ورودی‌های weight، size و color دارد تا مقداری عددی (قیمت بر حسب دلار) را به‌عنوان خروجی باز گرداند.

اگر الگوریتم مورد نیاز برای پیاده‌سازی *computePaperCost( )* ساده باشد و همگان قادر به درک آن باشند، دیگر به جزئیات بیشتری برای طراحی نیاز نیست. مهندس نرم‌افزار که کدنویسی را انجام می‌دهد، جزئیات لازم برای پیاده‌سازی عملیات را فراهم می‌سازد. ولی اگر الگوریتم، پیچیده‌تر یا محرمانه باشد، در این مرحله جزئیات طراحی بیشتری مورد نیاز است. در شکل ۸-۱۰ یک نمودار فعالیت UML برای *computePaperCost( )* تصویر شده است. هنگامی که نمودارهای فعالیت برای مشخص کردن طراحی در سطح مؤلفه‌ها استفاده می‌شوند، به‌طور کلی، در سطحی از انتزاع بیان می‌شوند که قدری بالاتر از کد منبع است. یک روش دیگر- استفاده از شبه کد برای مشخص کردن طراحی- در بخش ۳-۵-۱۰ بحث می‌شود.

مرحله ۴. منابع داده‌ای پایدار (فایل‌ها و بانک‌های اطلاعاتی) را توصیف و کلاس‌های لازم برای مدیریت آن‌ها را تعریف کنید. فایل‌ها و بانک‌های اطلاعاتی معمولاً فراتر از توصیف طراحی یک مؤلفه به شمار می‌روند. در اکثر موارد، این انبارهای داده‌ای پایدار، ابتدا به‌عنوان بخشی از طراحی معماری مشخص می‌شود. به هر حال، با افزوده شدن بر جزئیات طراحی، فراهم آوردن جزئیات اضافی درباره ساختار و سازمان‌دهی این منابع داده‌ای پایدار، مفید واقع می‌شود.

آندرز

به موازاتی که طراحی مؤلفه‌ها را پالایش می‌کنید، از تشریح مرحله به مرحله استفاده کنید. همواره از خود بپرسید «آیا راهی وجود دارد که بتوان این روال را آسان‌تر کرد و در عین حال به همان نتیجه رسید؟»



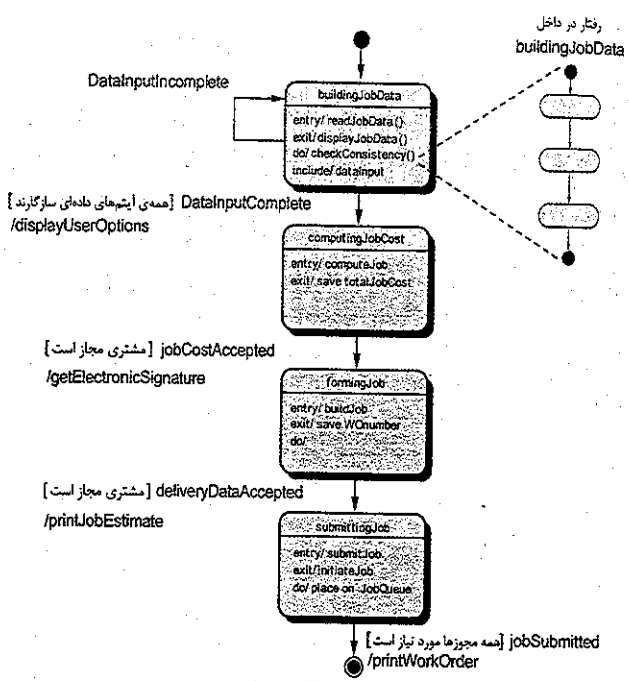
شکل ۸-۱۰ نمودار فعالیت UML برای *computePaperCost()*.

مرحله ۵. نمایش‌های رفتاری مربوط به یک کلاس یا مؤلفه را بسط و توسعه دهید. نمودارهای حالت UML به‌عنوان بخشی از مدل خواسته‌ها برای نمایش رفتار بیرونی سیستم و نیز رفتار محلی هر یک از کلاس‌های تحلیل به‌کار برده شدند. در اثنای طراحی در سطح مؤلفه‌ها، گاهی مدل‌سازی رفتار کلاس‌های طراحی ضرورت پیدا می‌کند.

رفتار پویای یک شیء (نمونه‌ای از یک کلاس طراحی در اجرای برنامه) از رویدادهایی که بیرون از آن به‌وقوع می‌پیوندد و از حالت فعلی (شیوه‌ی رفتار) شیء تأثیر می‌پذیرد. برای درک رفتار پویای یک شیء، باید کلیه‌ی use caseهای مرتبط با کلاس طراحی را در سراسر عمر آن بررسی کنید. این use caseها اطلاعاتی فراهم می‌سازند که شما را در ترسیم رویدادهای تأثیر گذار بر شیء و حالت‌هایی که شیء با گذر زمان و به وقوع پیوستن رویدادها در آن‌ها به سر می‌برد، یاری می‌دهند. **گذارهای میان حالت‌ها** (که رویدادها سبب به وقوع پیوستن آن‌ها می‌شوند) با به‌کارگیری یک نمودار حالت UML [Ben02] نمایش داده می‌شوند (شکل ۹-۱۰).

گذار از یک حالت (که با مستطیل گوشه گرد نشان داده شده است) به دیگری در نتیجه‌ی رویدادی به‌شکل زیر رخ می‌دهد:

Event-name (parameter-list) [guard-condition]/action expression



شکل ۹-۱۰ بخشی از نمودار حالت برای کلاس PrintJob.

که *event-name* رویداد را مشخص می‌کند، *parameter-list* شامل داده‌هایی می‌شود که به رویداد مربوط می‌شوند، *guard-condition* به زبان OCL نوشته می‌شود و شرطی را مشخص می‌کند که باید قبل از به وقوع پیوستن رویداد برقرار باشد و *action expression*، کنشی را تعریف می‌کند که با وقوع رویداد رخ می‌دهد.

در شکل ۹-۱۰ مشاهده می‌شود که هر حالت ممکن است کنش‌های */entry* و */exit* را تعریف کند که با *گذار به یک حالت* و *گذار از یک حالت* رخ می‌دهند. در اکثر موارد، این کنش‌ها متناظر با عملیات‌هایی هستند که به کلاس در حال مدل‌سازی تعلق دارند. شاخص */do* نشان‌دهنده‌ی فعالیت‌هایی است که در حالت رخ می‌دهد و شاخص */include* ابزاری برای پرداختن به جزئیات رفتار فراهم می‌آورد. برای این منظور، جزئیات بیشتری از نمودارهای حالت را در تعریف یک حالت به‌کار می‌گیرد.

ذکر این نکته اهمیت دارد که مدل رفتاری غالباً حاوی اطلاعاتی است که بلافاصله در مدل‌های طراحی دیگر آشکار نمی‌شود. برای مثال، بررسی دقیق نمودار حالت‌ها در شکل ۹-۱۰ نشان می‌دهد که رفتار پویای کلاس **PrintJob** وابسته به دو بار تصویب مشتری، یکی برای قیمت و یکی برای زمان‌بندی تحویل است. بدون این تصویب‌ها (*شرط نگهبان* تضمین می‌کند که مشتری مجاز به تصویب است)، کار چاپی را نمی‌توان ارائه کرد زیرا هیچ راهی برای رسیدن به حالت *submittingJob* وجود ندارد.

مرحله ۶. نمودارهای استقرار را بسط دهید تا جزئیات پیاده‌سازی اضافی فراهم آید. نمودارهای استقرار (فصل ۸) به‌عنوان بخشی از طراحی معماری به‌کار برده می‌شوند و به شکل توصیف‌گر ارائه می‌گردند. در این شکل، قابلیت‌های اصلی سیستم (که غالباً به‌صورت زیر سیستم نشان داده می‌شوند) در حیطه‌ی محیط محاسباتی‌ای که آن‌ها را در خود جای می‌دهد، نمایش داده شده‌اند.

در اثنای طراحی در سطح مؤلفه‌ها، نمودارهای استقرار را می‌توان بسط داد و بر جزئیات آن‌ها افزود؛ تا مکان پکیج‌های کلیدی مؤلفه‌ها را نمایش دهند. ولی، مؤلفه‌ها عموماً در نمودار مؤلفه‌ها به‌طور انفرادی نمایش داده نمی‌شوند. دلیل آن، پرهیز از پیچیدگی نموداری است. در برخی موارد، در این زمان بر جزئیات نمودارهای استقرار افزوده می‌شود تا به شکل نمونه‌ی اولیه در آیند. این بدان معناست که سخت‌افزارها و محیط(های) سیستم عامل ویژه‌ای که استفاده خواهند شد، مشخص می‌شوند و مکان پکیج‌های مؤلفه در این محیط خاطرنشان می‌شود.

مرحله ۷. نمایش طراحی در سطح مؤلفه‌ها را بازآرایی کنید و همواره راه‌های دیگر را مد نظر داشته باشید. در سراسر این کتاب تأکید کرده ایم که طراحی، فرایندی مبتنی بر تکرار است. نخستین مدل طراحی که در سطح مؤلفه‌ها ایجاد می‌کنید به اندازه‌ی *n* امین دور تکراری که روی مدل به‌کار می‌برید، کامل، سازگار یا صحیح نیست.

به‌علاوه، نباید از چشم‌انداز تونل متأثر شوید. همواره راهکارهای طراحی دیگری وجود دارد و بهترین طراحان، همه‌ی (یا اکثر) آن‌ها را قبل از پرداختن به مدل طراحی نهایی مد نظر قرار می‌دهند. این راه‌های دیگر را توسعه دهید و هر یک را با استفاده از اصول طراحی و مفاهیم ارائه شده در فصل ۸ و در این فصل به دقت در نظر بگیرید.

۱۰-۴ طراحی در سطح مؤلفه برای برنامه‌های تحت وب

مرز میان محتوا و قابلیت عملیاتی در خصوص سیستم‌ها و برنامه‌های کاربردی تحت وب، غالباً تیره و تار است. بنابراین، منطقی است بپرسیم: مؤلفه‌های برنامه‌ی تحت وب چه هستند؟

در حیطه‌ی این فصل، مؤلفه‌های برنامه‌های تحت وب (۱) توابع یکپارچه و تعریف‌شده‌ای هستند که محتوا را دستکاری می‌کنند یا پردازش محاسباتی یا داده‌ای را برای کاربر نهایی فراهم می‌سازند یا (۲) پکیج‌هایی از محتوا و توابع هستند که قدری از توانایی لازم را در اختیار کاربر نهایی قرار می‌دهند.

۱۰-۴-۱ طراحی محتوا در سطح مؤلفه‌ها

در طراحی محتوا در سطح مؤلفه‌ها، آنچه که کانون توجه قرار می‌گیرد، اشیای داده‌ای و شیوه‌ی بسته‌بندی آن‌ها برای ارائه به کاربر نهایی برنامه‌ی تحت وب است. برای مثال، قابلیت پایش ویدیویی مبتنی بر وب در داخل **SafeHomeAssured.com** را در نظر بگیرید. از جمله قابلیت‌های فراوان، کاربر می‌تواند هر کدام از دوربین‌های نقشه‌ی منزل را انتخاب و کنترل کند و تصاویر ویدیویی را از هر کدام از دوربین‌ها به نمایش در آورد. به‌علاوه، کاربر می‌تواند با استفاده از آیکون‌های کنترلی مناسب، زاویه و درشت‌نمایی دوربین را کنترل کند.

چند مؤلفه‌ی محتوایی بالقوه را می‌توان برای قابلیت پایش ویدیویی تعریف کرد: (۱) اشیای محتوایی که چیدمان فضایی (نقشه منزل) را با آیکون‌های اضافی نشان می‌دهد؛ این آیکون‌های اضافی،

مکان حس‌گرها و دوربین‌ها را نشان می‌دهند، (۲) مجموعه‌ای از تصاویر ویدیویی به‌صورت شمایل‌های کوچک (thumbnail) (هر کدام یک شیء داده‌ای جداگانه) و (۳) پنجره‌ی نمایش تصاویر زنده‌ی ویدیویی برای یک دوربین مشخص. هر کدام از این مؤلفه‌ها را می‌توان جداگانه به‌صورت پکیج، نامگذاری و دستکاری کرد.

نقشه‌ی منزلی را در نظر بگیرید که چهار دوربین مستقر در سراسر یک خانه را به تصویر می‌کشد. با درخواست کاربر، یک قاب ویدیویی از هر دوربین به نمایش در می‌آید و به‌عنوان شیئی با محتوای پویا به نام **VideoCapture*N*** شناخته می‌شود که N دوربین‌های ۱ تا ۴ را مشخص می‌کند. یک مؤلفه‌ی محتوایی با نام Thumbnail-Images، هر چهار شیء محتوایی **VideoCapture*N*** را با هم ترکیب کرده آن‌ها را روی صفحه‌ی پایش ویدیویی به نمایش در می‌آورد.

رسمیت طراحی محتوایی در سطح مؤلفه‌ها را باید مطابق با خصوصیات برنامه‌ی تحت وبی که قرار است ساخته شود، تنظیم کرد. در بسیاری موارد، لازم نیست اشیای محتوایی به‌عنوان یک مؤلفه، سازمان‌دهی شوند و می‌توان آن‌هارا به‌طور انفرادی پیاده‌سازی کرد. به هر حال، با رشد اندازه و پیچیدگی (برنامه‌ی تحت وب، اشیای محتوایی، و روابط میان آن‌ها)، ممکن است به سازمان‌دهی محتوا نیاز باشد تا دستکاری طراحی آسان‌تر شود.[1] به‌علاوه، اگر محتوا بسیار پویا باشد (مثل محتوای یک سایت حراج اینترنتی)، تعیین یک مدل ساختاری واضح که شامل مؤلفه‌های محتوایی باشد، اهمیت دارد.

۱۰-۴-۲ طراحی عملیاتی در سطح مؤلفه‌ها

برنامه‌های تحت وب نوین، حاوی قابلیت‌های پردازشی‌ای هستند که پیوسته بر پیچیدگی آن‌ها افزوده می‌شود؛ این قابلیت‌ها عبارتند از (۱) اجرای پردازش محلی برای تولید محتوا و قابلیت گشت‌وگذار به‌شیوه‌ای پویا، (۲) فراهم ساختن توانایی پردازش یا محاسبه داده‌ها که برای دامنه تجاری برنامه‌ی تحت وب مناسب باشد، (۳) فراهم ساختن امکان مراجعه به بانک‌های اطلاعاتی پیچیده و دستیابی به آن‌ها، یا (۴) برقراری واسط‌های داده‌ای با سیستم‌های خارجی. برای دستیابی به این قابلیت‌ها (و بسیاری قابلیت‌های دیگر) مؤلفه‌های عملیاتی‌ای برای برنامه‌ی تحت وب طراحی خواهید کرد که شکل آن‌ها مشابه مؤلفه‌های نرم‌افزاری برای نرم‌افزارهای سنتی است.

عملکردهای برنامه‌ی تحت وب به‌صورت یک سری مؤلفه تحویل داده می‌شوند که به موازات معماری اطلاعات توسعه می‌یابند تا از سازگار بودن آن‌ها اطمینان حاصل شود. در اصل با در نظر گرفتن مدل خواسته‌ها و همچنین معماری اطلاعات اولیه شروع می‌کنید و سپس به بررسی چگونگی تأثیرگذاری عملکردها بر تعامل کاربر با برنامه‌ی کاربردی، اطلاعاتی که ارائه می‌شوند و وظایفی که کاربر انجام می‌دهد، خواهید پرداخت.

طی طراحی معماری، محتوا و عملکردهای برنامه‌ی تحت وب با هم ترکیب می‌شوند تا یک معماری عملیاتی ایجاد گردد. *معماری عملیاتی*، نمایشی از دامنه‌ی عملیاتی برنامه‌ی تحت وب است و مؤلفه‌های عملیاتی کلیدی موجود در برنامه‌ی تحت وب و چگونگی تعامل این مؤلفه‌ها با یکدیگر را توصیف می‌کند.

---

[1] مؤلفه‌های محتوایی را نیز می‌توان در برنامه تحت وب‌های دیگر دوباره به‌کار برد.

برای مثال، قابلیت‌های درشت‌نمایی و تغییر زاویه دوربین برای پایش ویدیویی SafeHomeAssured.com به‌صورت بخشی از مؤلفه‌ی CameraControl پیاده‌سازی می‌شوند. به‌طریق دیگر، درشت‌نمایی و تغییر زاویه را می‌توان به‌صورت عملیات‌های *zoom( )* و *pan( )* پیاده‌سازی نمود که بخشی از کلاس Camera هستند. در هر حال، عملکردهایی که درشت‌نمایی و تغییر زاویه را ارائه می‌دهند، باید به‌صورت پیمانه‌هایی در داخل SafeHomeAssured.com پیاده‌سازی شوند.

۱۰-۵ طراحی مؤلفه‌های سنتی

مبانی طراحی در سطح مؤلفه‌ها در اوایل دهه ۱۹۶۰ شکل گرفت و با کارهای ادزگار دیکسترا و همکاران وی استحکام یافت [Boh66, Dij65, Dij76][1]. در اواخر دهه ۱۹۶۰، دیکسترا و دیگران، استفاده از ساختارهایی منطقی را پیشنهاد کردند که هر برنامه‌ای را با آنها می‌توان نوشت. این ساختارها بر «نگهداری از دامنه‌ی عملیاتی تأکید داشتند» یعنی هر ساختار دارای پیکربندی منطقی قابل پیش‌بینی بود که ورود به آن از بالا و خروج از آن از پایین رخ می‌داد به‌طوری که خواننده با سهولت بیشتری می‌توانست جریان رویه‌ای را دنبال کند.

نکته‌ی کلیدی
برنامه‌نویسی ساخت‌یافته یک تکنیک طراحی است که حاوی جریان منطقی برای سه نوع ساختمان است: ترتیب، شرط، تکرار

این ساختارها عبارتند از ترتیب (sequence)، شرط و تکرار. ترتیب، مراحلی از پردازش را پیاده‌سازی می‌کند که در تعیین مشخصات برنامه ضروری است. شرط، تسهیلات مربوط به پردازش انتخاب‌شده را براساس یک رخداد منطقی فراهم می‌آورد و تکرار، ایجاد حلقه را میسر می‌سازد. این سه ساختار در برنامه‌نویسی ساخت‌یافته - که تکنیک مهمی در طراحی در سطح مؤلفه‌ها به شمار می‌رود - اهمیت اساسی دارد.

پیشنهاد شده است که ساختارهای ساخت‌یافته، در تعدادی از عملیات قابل پیش‌بینی به‌کار گرفته شوند. معیارهای پیچیدگی (فصل ۲۳) نشان می‌دهد که استفاده از ساختارهای ساخت‌یافته، از پیچیدگی برنامه می‌کاهد و لذا خوانایی، آزمون‌پذیری و قابلیت نگهداری آن را افزایش می‌دهد. کاربرد تعداد محدودی از ساختارهای منطقی، در فرایند درک بشری سهم دارد که روان‌شناسان آن را *قطعه‌بندی* (chunking) می‌نامند. برای درک این فرایند، شیوه‌ی خواندن این صفحه در نظر بگیرید. شما حروف را تک به تک نمی‌خوانید بلکه الگوها یا قطعه‌هایی از حروف را می‌خوانید که واژه‌ها یا عبارت‌ها را تشکیل می‌دهند. ساختارهای ساخت‌یافته، قطعاتی منطقی هستند که به خواننده اجازه می‌دهند تا عناصر رویه‌ای یک پیمانه را به‌جای خواندن طراحی یا کد، به‌صورت خط به خط شناسایی کند. میزان درک، هنگامی بهبود می‌یابد که الگوهای منطقی قابل شناسایی وجود داشته باشند.

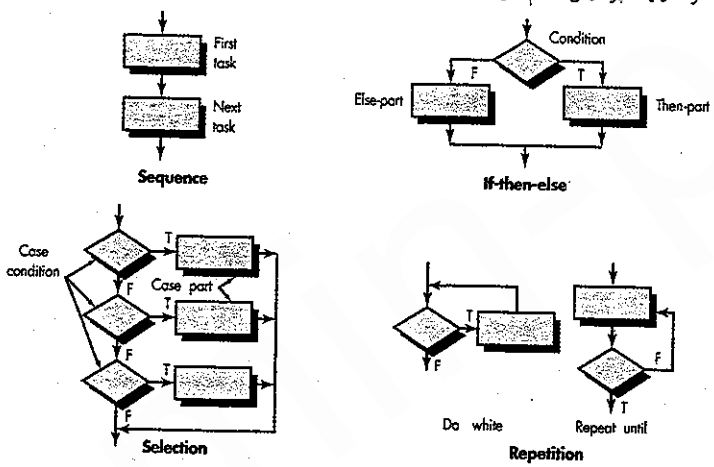
۱۰-۵-۱ طراحی با ابزارهای گرافیکی

«یک تصویر، گویاتر از هزار حرف است» ولی این که کدام تصویر و کدام هزار حرف، قدری اهمیت دارد. شکی نیست که ابزارهای گرافیکی مثل نمودار فعالیت UML یا نمودار گردشی، الگوهای

[1] مؤلفه‌ی سنتی، عنصری از پردازش پیاده‌سازی می‌شود که به تابع یا زیرتابعی موجود در دامنه مسأله یا قابلیتی موجود در دامنه زیرساخت می‌پردازد. مؤلفه سنتی که غالباً از آن با عنوان‌هایی چون پیمانه، روال یا زیرروال یاد می‌شود، داده‌ها را به آن صورت که در مؤلفه های شیءگرا پنهان‌سازی می‌شود، پنهان‌سازی نمی‌کند.

تصویری مفیدی به‌دست می‌دهند که به‌راحتی جزئیات رویه‌ای را منعکس می‌کنند. ولی، اگر از ابزارهای گرافیکی استفاده‌ی نادرست به عمل آید، تصویر نادرست ممکن است به نرم‌افزاری نادرست منتهی شود.

به کمک نمودار فعالیت می‌توانید ترتیب، شرط و تکرار (همه‌ی عناصر برنامه‌نویسی ساخت‌یافته) را به‌نمایش در آورید؛ نمودار فعالیت یک نمایش طراحی تصویری قدیمی‌تر موسوم به *نمودار گردشی* است (که هنوز هم کاربردی گسترده دارد). نمودار گردشی، همانند نمودار فعالیت، از نظر تصویری کاملاً ساده است. برای نشان دادن یک مرحله‌ی پردازش، از مستطیل استفاده می‌شود. لوزی نشان‌گر شرط‌های منطقی و پیکان‌ها نشان دهنده جریان کنترل هستند. در شکل ۱۰-۱۰ سه ساختار ساخت‌یافته را می‌بینید. ترتیب به‌صورت دو مستطیل پردازش نشان داده می‌شود که توسط یک خط (پیکان) کنترل به هم متصل می‌شوند. شرط، که به آن *if-then-else* هم می‌گویند، به‌صورت یک لوزی نشان داده می‌شود که اگر درست باشد، باعث پردازش بخش *then* می‌شود و اگر نادرست باشد، بخش *else* پردازش می‌شود. تکرار، با استفاده از دو شکل نسبتاً متفاوت نشان داده می‌شود. ساختار *do while* شرطی را چک می‌کند و حلقه را مادامی که آن شرط برقرار باشد، تکرار می‌کند. ساختار *repeat until* ابتدا حلقه را اجرا می‌کند، سپس‌شرطی را چک می‌کند و حلقه را آن قدر تکرار می‌کند تا آن شرط دیگر برقرار نباشد. ساختار انتخاب که در شکل نشان داده شده است، درواقع شکل بسط یافته‌ای از *if-then-else* است. پارامتری با تصمیم‌گیری‌های پیاپی چک می‌شود تا اینکه یک شرط درست برقرار گردد و یک پردازش انجام شود.



شکل ۱۰-۱۰ ساختارهای نمودار گردشی.

به‌طور کلی، اگر به‌جای مجموعه‌ای ازحلقه‌ها یا شرط‌های تودرتو، از ساختارهای ساخت‌یافته به‌وفور استفاده شود، بازدهی کاهش می‌یابد. مهم‌تر اینکه پیچیدگی اضافی کلیه آزمون‌های منطقی می‌تواند جریان کنترل نرم‌افزار را نامشخص کرده امکان خطا را بالا ببرد و تأثیری منفی بر خوانایی و قابلیت نگهداری آن بگذارد. پس چه باید کرد؟

دو گزینه فراروی طراح باقی می‌ماند: (۱) نمایش رویه‌ای، دوباره طراحی می‌شود تا در یک مکان تودرتو از جریان کنترل، نیاز به انشعاب نباشد؛ (۲) از ساختارهای ساخت‌یافته به شیوه‌ای کنترل شده صرفنظر می‌شود؛ یعنی یک انشعاب محدود به خارج از جریان کنترل طراحی می‌شود. واضح است که گزینه ۱ روشی ایده‌آل است، ولی گزینه ۲ را می‌توان بدون عدول از جوهره برنامه‌نویسی ساخت‌یافته، عملی کرد.

۲-۵-۱۰ نمادگذاری طراحی به‌روش جدولی

در بسیاری از کاربردهای نرم‌افزاری، ممکن است برای ارزیابی ترکیب پیچیده‌ای از شرط‌ها و انتخاب کنش‌های مناسب براساس این شرط‌ها، به یک پیمانه نیاز باشد. *جدول‌های تصمیم‌گیری* [Hur83]، نمادگذاری مربوط به ترجمه‌ی این کنش‌ها و شرط‌ها را (که در روایت پردازش با use case آمده‌اند) به شکل جدول فراهم می‌سازند. احتمال تفسیر نادرست این جدول بسیار کم است و حتی از آن می‌توان به عنوان ورودی برای یک الگوریتم جدولی استفاده کرد که ماشین قادر به خواندن آن باشد.

برخی از ابزارها و تکنیک‌های کهنه نرم‌افزار به خوبی با ابزارها و تکنیک‌های جدید مهندسی نرم‌افزار جور درمی‌آیند. جدول‌های تصمیم‌گیری مثالی عالی از این مدعایند. جداول تصمیم‌گیری تقریباً یک دهه قبل از ظهور مهندسی نرم‌افزار به وجود آمده‌اند، ولی به خوبی با مهندسی نرم‌افزاری که ممکن است برای آن هدف طراحی شده باشند، جور درمی‌آیند.

اندرز

هنگامی که به مجموعه‌ی پیچیده‌ای از شرایط و کنش‌ها در یک مؤلفه برخورد کنید، از جدول تصمیم‌گیری استفاده کنید.

قواعد

شرط‌ها	1	2	3	4		
	T	T				
			T	T		
					T	T
	F	T	F	T	F	T
کنش‌ها						
	✓					
			✓	✓		
					✓	✓
		✓		✓		✓

شکل ۱۱-۱۰ نامگذاری جدول تصمیم‌گیری

سازمان‌دهی جدول تصمیم‌گیری در شکل ۱۱-۱۰ نشان داده شده است. جدول به چهار بخش تقسیم شده است. ربع بالا سمت چپ، حاوی فهرستی از کلیه شرط‌هاست. ربع پایین سمت چپ، حاوی فهرستی از کلیه عملیاتی است که براساس ترکیبات شرط‌ها امکان‌پذیرند. ربع‌های دست راستی، ماتریسی را تشکیل می‌دهند که نشان دهنده ترکیبات شرطی و عملیات متناظر با این ترکیبات است.

چگونه یک جدول تصمیم‌گیری می‌سازیم؟

بنابراین، هر ستون از ماتریس را می‌توان به عنوان یک *قاعده‌ی پردازش* تفسیر کرد. مراحل زیر برای توسعه یک جدول تصمیم‌گیری اجرا می‌شوند:

۱. فهرست کردن کلیه عملیاتی که می‌توان به یک رویه (یا پیمانه) مشخص ربط داد؛
۲. فهرست کردن کلیه شرط‌ها (یا تصمیمات اخذ شده) در اثنای اجرای رویه؛
۳. ربط دادن مجموعه‌های مشخصی از شرط‌ها به عملیات مشخصی که شرط‌های ناممکن را حذف می‌کنند؛ به طریق دیگر، توسعه هر جایگشتِ ممکن از شرط‌ها؛
۴. تعریف قواعد با مشخص کردن اینکه چه کنش(هایی) برای یک مجموعه از شرایط رخ می‌دهد.

برای نشان دادن کاربرد جدول تصمیم‌گیری، قطعه‌ی زیر را در نظر بگیرید که از شرح پردازش یک سیستم قبض‌نویسی برای برق منازل گرفته شده است.

> سه نوع مشتری تعریف می‌شود: مشتری عادی، مشتری نقره‌ای و مشتری طلایی (این انواع متناسب با مقدار کار تجاری‌ای که مشتری طی ۱۲ ماه با چاپخانه انجام می‌دهد، به او نسبت داده می‌شود). به مشتری عادی سرعت چاپ و تحویل عادی اختصاص داده می‌شود. مشتری نقره‌ای تخفیف هشت درصدی می‌گیرد و جلوی مشتریان عادی در صف قرار داده می‌شود. مشتری طلایی، پانزده درصد تخفیف می‌گیرد و در صف جلوی مشتریان عادی و نقره‌ای قرار می‌گیرد. علاوه بر سایر تخفیف‌ها یک تخفیف x درصدی خاص نیز روی هر قیمت بنا به اراده‌ی مدیریت قابل اعمال خواهد بود.

در شکل ۱۱-۱۰، نمایشی از جدول تصمیم‌گیری این روایت پردازش نشان داده شده است. هر یک از این شش قاعده، یکی از شش شرط ممکن را نشان می‌دهند. به عنوان قاعده‌ای کلی، جدول تصمیم‌گیری را می‌توان به طور مؤثر برای تکمیل نمادگذاری طراحی رویه‌ای به‌کار برد.

۳-۵-۱۰ زبان طراحی برنامه

زبان طراحی برنامه (PDL)، که به آن *انگلیسی ساخت‌یافته* یا *شبه‌کد* نیز گفته می‌شود، ساختار منطقی زبان برنامه‌نویسی را با توانایی بیانی یک زبان طبیعی (مثلا انگلیسی) در هم می‌آمیزد. متن روایی (مثلاً انگلیسی) در قالب نحوی زبان برنامه‌نویسی ادغام می‌شود. برای بهبود بخشیدن به PDL می‌توان از ابزارهای خودکار (مثلاً [Cai03]) استفاده کرد.

نحو اصلی در PDL باید شامل ساختارهایی برای تعریف زیربرنامه‌ها، توصیف واسط، اعلان داده‌ها، تکنیک‌هایی برای سازمان‌دهی بلوک‌ها، ساختارهای شرطی، ساختارهای تکرار و ساختارهای I/O باشد. لازم به ذکر است که PDL را می‌توان بسط داد تا واژه‌های کلیدی مربوط به پردازش چند کار، و/یا پردازش همروند، مدیریت وقفه‌ها، همگام‌سازی بین فرایندها و بسیاری از ویژگی‌های دیگر را نیز در بر گیرد. طراحی کاربردهایی که PDL باید برای آنها استفاده می‌شود، شکل نهایی زبان طراحی را دیکته می‌کند. فرمت و معنای برخی از این ساختارهای PDL در مثال زیر ارائه می‌شود.

برای نشان دادن کاربرد PDL ، مثالی از یک طراحی رویه‌ای را برای نرم‌افزار سیستم امنیتی *SafeHome* ارائه می‌دهیم. سیستم *SafeHome* موردنظر برای دود، آتش‌سوزی، دزدی، آب و دما (مثلاً وقتی که صاحبخانه در زمستان در منزل نیست و شوفاژخانه خراب می‌شود) هشدار می‌دهد؛ آژیر را به صدا درمی‌آورد و با تولید پیام صدای ضبط شده سرویس پایشی را به کمک فرا می‌خواند.

به‌خاطر بسپارید که PDL، زبان برنامه‌نویسی نیست. طراح می‌تواند بنا به نیاز خود عمل کند و نگران خطاهای نحوی نباشد. ولی، طراحی برای نرم‌افزار پایش‌گر باید مورد بازبینی قرار گیرد (مشکلی احساس نمی‌کنید؟) و پیش از آنکه به کد تبدیل شود، باز هم باید پالایش شود. PDL زیر[1] به تشریح طراحی رویه‌ای برای نسخه ی اولیه‌ای از مؤلفه ی مدیریت هشدارها (**alarmManagemet**) کمک می‌کند.

```
Component alarmManagement;
The intent of this component is to manage control panel switches
and input from sensors by type and to act on any alarm condition
that is encountered.
   set default values for systemStatus(returned value),all data items
   initialize all system ports and reset all hardware
   check controlPanelSwitches(cps)
      if cps = "test" then invoke alarm set to "on"
      if cps = "alarmOFF" then invoke alarm set to "off"
      if cps = "newBoundingValue" then invoke keyBoardInput
      if cps = "burglarAlarmOFF" then invoke deactivateAlarm
      .
      .
      .
      default for cps = none
   reset all singalValues and switches
   do for all sensors
      invoke checkSensor procedure returning singalValue
      if singalValue > bound [alarmType]
         then phoneMessage = message [alarmType]
            set alarmBell to "on" for alarmTimeSeconds
            set system status = "alarmCondition"
            parbegin \parallel block,all tasks are executed in parallel.\
               invoke alarm procedure with "on", alarmTimeSeconds
               invoke alarm procedure set to alarmType, phoneNumber
            endpar
         else skip
      endif
   enddofor
end alarmManagement
```

[1] سطح جزئیات ارائه‌شده توسط PDL به‌صورت محلی تعریف می‌شود. برخی افراد، توصیف‌های زبانی را بیشتر ترجیح می‌دهند درحالی که عده‌ای دیگر، چیزی شبیه به کد را می‌پسندند.

توجه دارید که طراح برای مؤلفه **alarmManagemet** از یک ساختار جدید یعنی *parbegin... parend* استفاده کرده است که بلوکی موازی را مشخص می‌کند. همه‌ی وظایف مشخص شده در داخل بلوک *parbegin* به طور موازی اجرا می‌شوند. در این مورد، به جزئیات پیاده‌سازی کاری نداریم.

۱۰-۶ توسعه مبتنی بر مؤلفه‌ها

در حیطه‌ی مهندسی نرم‌افزار، استفاده‌ی مجدد ایده‌ای است جدید و در عین حال قدیمی. برنامه‌نویسان از اولین روزهای کار با کامپیوتر از ایده‌ها، انتزاع‌ها و پردازش‌ها چندین بار استفاده کردند، ولی رویکرد اولیه به استفاده‌ی مجدد، از روی برنامه‌ریزی نبود. امروزه، سیستم‌های کامپیوتری پیچیده با کیفیت بالا باید در دوره‌های زمانی بسیار کوتاه ساخته شوند و برای استفاده‌ی مجدد، به رویکردی سازمان یافته‌تر نیاز دارند.

مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها (CBSE) فرایندی است که بر طراحی و ساخت سیستم‌های کامپیوتری با به‌کارگیری «مؤلفه های» نرم‌افزار با قابلیت استفاده‌ی مجدد تاکید دارد. کلمنتس [Cle95] مفهوم CBSE را چنین توصیف می‌کند:

> [CBSE] تجسمی است از فلسفه‌ی «بخر و نسازه» که فرد بروکز و سایرین بر آن تاکید بسیار داشتند. به همان شیوه‌ای که زیر روال‌های اولیه، برنامه‌نویس را از اندیشیدن به جزئیات رها می‌ساختند، [CBSE] تأکید را از برنامه‌نویسی نرم‌افزار به ساخت سیستم‌های نرم‌افزاری جابجا می‌کند. کانون توجه از پیاده‌سازی به انسجام‌بخشی تغییر یافته است.

ولی چند سؤال مطرح می‌شود. آیا ساخت سیستم‌های پیچیده با سرهم‌کردن مؤلفه‌های قابل استفاده‌ی موجود در یک کاتالوگ امکان‌پذیر هست؟ آیا می‌توان آن را به شیوه‌ای انجام داد که از نظر هزینه و زمان، بازدهی داشته باشد؟ آیا می‌توان سازوکارهایی برقرار کرد که مهندسان نرم‌افزار را به استفاده‌ی مجدد به جای ابداع دوباره تشویق کند؟ آیا مدیریت، مشتاق پرداخت هزینه‌های اضافی برای ایجاد مؤلفه‌هایی با قابلیت استفاده‌ی مجدد هست؟ آیا کتابخانه‌ی مورد نیاز برای دستیابی به استفاده‌ی مجدد وجود دارد؟ آیا کسانی که به مؤلفه‌های موجود نیاز دارند، قادر به یافتن آنها هستند؟

پاسخ هر کدام از این پرسش به‌طور فزاینده‌ای مثبت است. در باقیمانده‌ی این فصل، به بررسی برخی مسائل خواهیم پرداخت که در موفقیت CBSE در یک سازمان مهندسی نرم‌افزار باید مد نظر داشت.

۱۰-۶-۱ مهندسی دامنه (Domain Engineering)

هدف از *مهندسی دامنه*، شناسایی، پیاده‌سازی، کاتالوگ‌بندی و توزیع مجموعه‌ای از مؤلفه‌های نرم‌افزاری است که در یک دامنه‌ی کاربرد خاص در نرم‌افزار فعلی و نرم‌افزارهای آینده، کاربرد دارد.[1] هدف کلی، برقراری سازوکارهایی است که مهندس نرم‌افزار به کمک آنها بتواند این مؤلفه‌ها را طی کار روی سیستم‌های جدید و سیستم‌های موجود به اشتراک بگذارد – تا از آنها استفاده‌ی مجدد به عمل آید. مهندسی دامنه شامل سه فعالیت اصلی می‌شود – تحلیل، ساخت و توزیع.

رویکرد کلی برای *تحلیل دامنه* غالباً در حیطه‌ی مهندسی نرم‌افزار شیءگرا مشخص می‌شود. مراحل این فرایند به‌صورت زیر تعریف می‌شوند:

[1] در فصل ۹ از ژانرهای معماری سخن به میان آمد که دامنه‌های کاربرد مشخص را تعیین می‌کنند.

> «مهندسی دامنه به یافتن وجوه مشترک میان سیستم‌ها مربوط می‌شود به طوری که بتوان مؤلفه‌های قابل استفاده در چندین سیستم را شناسایی کرد.»
>
> **پل کلمنتس**

۱. تعریف دامنه‌ای که قرار است بررسی شود.
۲. دسته‌بندی اقلامی که باید از دامنه استخراج شوند.
۳. گرفتن یک نمونه‌ی نماینده از برنامه‌های کاربردی موجود در آن دامنه.
۴. تحلیل هر کاربرد نمونه و تعریف کلاس‌های تحلیل.
۵. توسعه‌ی مدل خواسته‌ها برای کلاس‌ها.

لازم به ذکر است که تحلیل دامنه برای هر الگوی مهندسی نرم‌افزار قابل استفاده است و برای توسعه نرم‌افزار به روش سنتی نیز می‌توان از آن استفاده نمود.

۲-۶-۱۰ صلاحیت، تطبیق و ترکیب

مهندسی دامنه، کتابخانه‌ای از مؤلفه‌های قابل استفاده‌ی مجدد فراهم می‌سازد که برای مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها مورد نیاز است. برخی از این مؤلفه‌های قابل استفاده‌ی مجدد به‌صورت داخلی توسعه داده می‌شوند، برخی از این مؤلفه‌های قابل استفاده‌ی مجدد به‌صورت داخلی توسعه داده می‌شوند، برخی دیگر از برنامه‌های کاربردی موجود قابل استخراج هستند و عده‌ای را نیز می‌توان از یک شرکت سازنده دیگر تأمین نمود.

متاسفانه، وجود مؤلفه‌های قابل استفاده‌ی مجدد این را ضمانت نمی‌کند که مولفه‌های مذکور را می‌توان به سهولت یا به‌طرزی اثربخش در معماری برگزیده شده برای برنامه‌ی کاربردی جدید منسجم ساخت. به همین دلیل هنگام پیشنهاد یک مؤلفه، باید دنباله‌ای از کنش‌های توسعه مبتنی بر مولفه‌ها را اعمال نمود.

صلاحیت مؤلفه (Component Qualification). صلاحیت مؤلفه تضمین می‌کند که مؤلفه‌ی مورد نظر، وظیفه‌ی مورد نیاز را انجام می‌دهد، به‌طور مناسب در سبک معماری مشخص‌شده برای سیستم (فصل ۹) می‌گنجد، و ویژگی‌های کیفی (نظیر کارایی، قابلیت اعتماد و قابلیت استفاده) مورد نیاز برای برنامه‌ی کاربردی را فراهم می‌سازد.

توصیف واسط، اطلاعات مفیدی درباره‌ی عملیات و استفاده از یک مؤلفه نرم‌افزار در اختیار می‌گذارد، ولی برای تعیین اینکه آیا از یک مؤلفه‌ی پیشنهادی واقعاً در کاربردی جدید می‌توان استفاده کرد، همه‌ی اطلاعات لازم در اختیار قرار نمی‌دهد. از میان عوامل فراوانی که باید طی تعیین صلاحیت مؤلفه‌ها در نظر گرفت می‌توان به موارد ذیل اشاره نمود:

- واسط برنامه‌ی کاربردی (API)
- ابزارهای توسعه و انسجام بخشی مورد نیاز مؤلفه
- خواسته‌های زمان اجرا، از جمله استفاده از منابع (مثل حافظه یا فضای ذخیره سازی)، زمان‌بندی یا سرعت و پروتکل شبکه.
- خواسته‌های سرویس، از جمله واسط‌های سیستم عامل و پشتیبانی از سایر مؤلفه ها.
- ویژگی‌های امنیتی، از جمله کنترل‌های دستیابی و پروتکل تایید.
- فرض‌های پذیرفته‌شده در طراحی، از جمله استفاده از الگوریتم‌های عددی یا غیرعددی.
- مدیریت استثناها

اگر مؤلفه‌هایی که در داخل سازمان ساخته شده‌اند، به‌عنوان مؤلفه‌های قابل استفاده‌ی مجدد پیشنهاد شده باشند، ارزیابی هر کدام از این عوامل، نسبتاً آسان خواهد بود. اگر طی توسعه‌ی یک

اندرز

فرایند تحلیل که در این بخش بحث می‌کنیم، مؤلفه‌های قابل استفاده‌ی مجدد را مورد توجه قرار می‌دهد. ولی، تحلیل سیستم‌های COTS کامل (مثل برنامه‌های کاربردی تجارت الکترونیک، برنامه‌های خودکارسازی فروش) نیز می‌تواند بخشی از تحلیل دامنه باشد.

طی تعیین صلاحیت مؤلفه‌ها چه عواملی در نظر گرفته می‌شود؟

مؤلفه، کار مهندسی نرم‌افزار به خوبی انجام شده باشد، به این سؤالات می‌توان پاسخ داد. ولی تعیین عملکرد داخلی مؤلفه‌های COTS (Components Off The Shelf) یا مؤلفه‌های تأمین شده از منابع دیگر دشوارتر است، چون تنها اطلاعات در دسترس ممکن است خود مشخصات واسط باشد.

تطبیق مؤلفه‌ها (Component Adaptation). در شرایط ایده‌آل، با مهندسی دامنه، مؤلفه‌هایی ایجاد می‌شود که می‌توان به آسانی آنها را در معماری یک برنامه‌ی کاربردی انسجام بخشید. منظور از «انسجام آسان» این است که (۱) روش‌های سازگار مدیریت منابع برای همه‌ی مؤلفه‌های موجود در کتابخانه پیاده‌سازی شده باشند، (۲) فعالیت‌های رایج از قبیل مدیریت داده‌ها برای تمامی مؤلفه‌ها موجود باشد و (۳) واسط‌های داخل معماری با محیط خارجی به شیوه‌ای سازگار پیاده‌سازی شده باشند.

در واقع، حتی پس از تایید صلاحیت مؤلفه برای استفاده در معماری یک برنامه‌ی کاربردی خاص، تضادهایی ممکن است در یک یا چند مورد از موارد ذکر شده رخ دهد. برای پرهیز از این تضادها، گاهی یک تکنیک تطبیق موسوم به «لفاف‌بندی مؤلفه‌ها» [Bro96] به‌کار برده می‌شود. هنگامی که تیم نرم‌افزار به طراحی داخلی و کدهای مربوط به یک مؤلفه دسترسی داشته باشد (که معمولاً چنین نیست مگر اینکه از مؤلفه‌های COTS منبع-باز استفاده شود)، از تکنیک لفاف‌بندی جعبه‌ی سفید، استفاده می‌شود. لفاف‌بندی جعبه‌ی سفید، همانند همتای خود در آزمون نرم‌افزار (فصل ۱۸) جزئیات پردازشی داخلی مؤلفه را بررسی می‌کند و برای برطرف ساختن هر گونه تضاد، اصلاحاتی در سطح کدها به عمل می‌آورد. لفاف‌بندی جعبه‌ی خاکستری هنگامی به‌کار برده می‌شود که کتابخانه‌ی مؤلفه‌ها یک زبان بسط مؤلفه‌ها یا API فراهم می‌آورد که برطرف ساختن یا پوشاندن تضادها را میسر می‌سازد. لفاف‌بندی جعبه‌ی سیاه نیاز به واردکردن پیش پردازش و پس پردازش در واسط مؤلفه دارد تا بتواند تضادها را برطرف کند یا بپوشاند. شما باید تعیین کنید که آیا تلاش لازم برای لفاف‌بندی کافی یک مؤلفه، توجیه دارد یا اینکه یک مؤلفه‌ی سفارشی (طراحی‌شده به‌منظور حذف تضادهای مشاهده شده) باید دوباره مهندسی شود.

ترکیب مؤلفه‌ها (Component Composition). وظیفه‌ی ترکیب مؤلفه‌ها عبارت است از مونتاژ مؤلفه‌های تایید صلاحیت شده، تطبیق یافته و مهندسی شده جهت تشکیل معماری تعیین شده برای یک برنامه‌ی کاربردی. برای دستیابی به این منظور، باید زیرساختی فراهم آید که این مؤلفه‌ها را در قالب یک سیستم عملیاتی به هم پیوند دهد. این زیرساخت (معمولاً کتابخانه‌ای از مؤلفه‌های تخصص یافته) مدلی برای هماهنگی مؤلفه‌ها و سرویس‌های خاص فراهم می‌آورد که مؤلفه‌ها به کمک آن با یکدیگر هماهنگ شده، وظایف مشترک را اجرا می‌کنند.

از آن‌جا که تأثیر بالقوه‌ی استفاده‌ی مجدد و CBSE بر صنعت نرم‌افزار بسیار بزرگ است، چند کنسرسیوم شرکتی و صنعتی، استانداردهایی را برای نرم‌افزارهای مؤلفه‌ای پیشنهاد کرده‌اند.[1]

OMG/CORBA. گروه مدیریت اشیاء یک معماری واسطه‌ی درخواست اشیای مشترک (OMG/CORBA) منتشر ساخته است. واسطه‌ی درخواست اشیا (ORB) سرویس‌های متنوعی فراهم می‌سازد که برقراری ارتباط مؤلفه‌ها (اشیای) قابل استفاده‌ی مجدد با سایر مؤلفه‌ها را فارغ از مکان آنها در یک سیستم میسر می‌سازد.

[1] گرگ اولسن [Ols06] بحثی عالی از تلاش‌های گذشته و حال برای تحقق بخشیدن به CBSE فراهم ساخته است.

اندرز

علاوه بر ارزیابی این که آیا تطبیق مؤلفه‌ها برای استفاده‌ی مجدد، توجیه اقتصادی دارد، باید این را هم ارزیابی کند که آیا دستیابی به قابلیت عملیاتی و کارایی مورد نیاز از نظر هزینه‌ها بازدهی کافی را دارد یا خیر.

مرجع وب

آخرین اطلاعات در خصوص CORBA را در www.omg.org می‌توانید بیابید.

COM و .NET مایکروسافت. مایکروسافت یک مدل اشیای مؤلفه‌ای (COM) توسعه داده است که برای به‌کارگیری مؤلفه‌های تولید شده توسط شرکت‌های گوناگون فعال در یک برنامه‌ی کاربرد که تحت سیستم عامل Windows اجرا می‌شود، مشخصات لازم را تعیین می‌کند. از دیدگاه برنامه‌ی کاربردی، «آنچه که کانون توجه قرار می‌گیرد، فقط چگونگی پیاده‌سازیِ [اشیای COM] نیست، بلکه این واقعیت نیز مورد توجه است که شیء دارای واسطی است که با سیستم ثبت می‌شود و از سیستم مؤلفه‌ها برای برقراری ارتباط با سایر اشیای COM استفاده می‌کند.» [Har98a]. چارچوب .NET مایکروسافت شامل COM می‌شود و کتابخانه‌ای از کلاس‌های قابل استفاده‌ی مجدد را فراهم می‌آورد که آرایه‌ی وسیعی از دامنه‌های کاربرد را پوشش می‌دهند.

مؤلفه‌های JavaBeans سیستم مؤلفه‌های JavaBeans یک زیرساخت مستقل از سکو برای CBSE است که با به‌کارگیری زبان برنامه‌نویسی جاوا توسعه یافته است. سیستم مؤلفه‌های JavaBeans شامل مجموعه‌ای از ابزارها موسوم به **کیت توسعه‌ی دانه‌ها** (BDK) می‌شود که به سازندگان امکان می‌دهد تا (۱) چگونگی کارکردن دانه‌ها (مولفه های) موجود را تحلیل کنند، (۲) رفتار و ظاهر آنها را مطابق سلیقه خود تغییر دهند، (۳) سازوکارهایی برای هماهنگ‌سازی و برقراری ارتباط میان آنها وضع کنند، (۴) برای استفاده در برنامه‌های کاربردی خاص، دانه‌های سفارشی بسازند و (۵) رفتار دانه‌ها را آزمون و ارزیابی کنند.

هیچ یک از این استانداردها، به تنهایی در صنعت غالب نشده است. گرچه بسیاری از سازندگان، استانداردهای خود را براساس یکی از آنها وضع کرده اند، این احتمال وجود دارد که سازمان‌های بزرگ نرم‌افزاری، براساس گروه‌های برنامه‌ی کاربردی و سکوهای انتخاب شده، استانداردی را برگزینند.

مرجع وب

آخرین اطلاعات در خصوص COM و .NET را از وب سایت زیر می‌توانید به‌دست آورید:
www.microsoft.com/COM
و
msdn2.microsoft.com/en-us/netframework/default.aspx

مرجع وب

آخرین اطلاعات در خصوص JavaBeans را در
java.sun.com/products/javabeans/docs
می‌توانید بیابید.

۳-۶-۱۰ تحلیل و طراحی برای استفاده‌ی مجدد

گرچه فرایند CBSE مشوق استفاده از مؤلفه‌های نرم‌افزاری موجود است، گاهی چاره‌ای جز ایجاد مؤلفه‌های نرم‌افزاری جدید و ترکیب آن‌ها با COTSها و مؤلفه‌های داخلی سازمان وجود ندارد. از آن‌جا که این مؤلفه‌های جدید، عضو کتابخانه‌ی داخلی مؤلفه‌های قابل استفاده‌ی مجدد می‌شوند، باید برای استفاده‌ی مجدد مهندسی شوند.

مفاهیم طراحی از قبیل انتزاع، پنهان‌سازی، استقلال عملیاتی، پالایش و برنامه‌نویسی ساخت‌یافته همراه با روش‌های شیءگرا، آزمون، تضمین کیفیت نرم‌افزار (SQA) و روش‌های وارسی (فصل ۲۱) همگی در ایجاد مؤلفه‌های نرم‌افزاری با قابلیت استفاده‌ی مجدد سهم دارند. در این بخش، به مسائل مختص استفاده‌ی مجدد خواهیم پرداخت که مکمل کار مهندسی نرم‌افزار مستحکم است.

مدل خواسته‌ها تحلیل می‌شود تا عناصری تعیین گردند که به مؤلفه‌های قابل استفاده‌ی مجدد موجود اشاره دارند. طی فرایندی که گاه از آن به‌عنوان «*همخوانی مشخصات*» یاد می‌شود، عناصر مدل خواسته‌ها با توصیفات به عمل آمده از مؤلفه‌های قابل استفاده‌ی مجدد مقایسه می‌شوند [Bel95]. اگر همخوانی مشخصات نشان دهد که یک مؤلفه‌ی موجود برای نیازهای برنامه‌ی کاربردی فعلی مناسب است، می‌توانید مؤلفه را از کتابخانه‌ی (مخزن) استخراج کنید و آن را در طراحی سیستمی جدید به‌کار ببرید. اگر چنین مؤلفه‌هایی یافت نشوند (یعنی همخوانی مشاهده نشود)، مؤلفه‌ای جدید ایجاد

می‌گردد. در این نقطه است (یعنی هنگامی که شروع به ایجاد مؤلفه‌ای جدید می‌کنید) که *طراحی برای استفاده‌ی مجدد* (DFR) باید در نظر گرفته شود.

چنان که پیش از این نیز گفته شد، DFR شما را ملزم می‌سازد تا اصول و مفاهیم طراحی نرم‌افزار مستحکمی (فصل ۸) را به‌کار بندید. ولی خصوصیات دامنه‌ای کاربرد را نیز باید مد نظر داشت. بایندر [Bin93] چند مسأله‌ی کلیدی را مطرح می‌کند که بستری جهت طراحی برای استفاده‌ی مجدد تشکیل می‌دهند[1]:

اندرز

هنگامی که قرار باشد مؤلفه‌ها با سیستم‌های قدیمی یا با چند سیستم که معماری ناسازگاری دارند، منسجم شوند، DFR می‌تواند کاری دشوار باشد.

دامنه‌های استاندارد. دامنه کاربرد باید بررسی شود و ساختمان داده‌ها (مثلاً ساختار فایل‌ها یا یک بانک اطلاعاتی کامل) باید تعیین گردد. سپس همه‌ی مؤلفه‌های طراحی را می‌توان طوری مشخص کرد که از این ساختمان داده‌های استاندارد استفاده کنند.

پروتکل‌های واسط استاندارد. سه سطح از پروتکل واسط را باید وضع کرد: ماهیت واسط‌های بین پیمانه‌ها، طراحی واسط‌های فنی (غیرانسانی) خارجی و واسط‌های میان انسان و کامپیوتر.

قالب‌های برنامه. یک سبک معماری (فصل ۹) انتخاب می‌شود و می‌تواند به‌عنوان قالبی برای طراحی معماری نرم‌افزار جدید عمل کند.

هنگامی که داده‌های استاندارد، واسط‌ها و قالب‌های برنامه تعیین شدند، چارچوبی برای طراحی در اختیار دارید. مؤلفه‌های جدیدی که از این چارچوب پیروی می‌کنند، احتمال استفاده‌ی مجدد از آنها در آینده بیشتر خواهد بود.

۴-۶-۱۰ طبقه‌بندی و بازیابی مؤلفه‌ها

یک کتابخانه دانشگاهی بزرگ را در نظر بگیرید. صدها هزار کتاب، مجله و سایر منابع اطلاعاتی در دسترس قرار دارند. ولی برای دستیابی به این منابع، یک الگوی گروه‌بندی باید پی‌ریزی کرد. کتابدارها برای گشت و گذار در این حجم بزرگ اطلاعات، یک الگوی طبقه‌بندی تعریف کرده‌اند که شامل کد طبقه‌بندی کتابخانه کنگره آمریکا، واژه‌های کلیدی، نام مؤلفان و سایر مدخل‌های نمایه‌ای می‌شود. همه‌ی این اطلاعات به کاربر کمک می‌کنند تا منبع مورد نیاز را به سرعت و به راحتی بیابد.

اکنون یک مخزن بزرگ از مؤلفه‌ها را در نظر بگیرید. ده‌ها هزار مؤلفه‌ی نرم‌افزاری قابل استفاده‌ی مجدد در آن موجود است. ولی چگونه مؤلفه‌ای را که می‌خواهید، پیدا می‌کنید؟ برای پاسخ گفتن به این پرسش، یک سؤال دیگر مطرح می‌شود: مؤلفه‌های نرم‌افزاری را چگونه می‌توان به شیوه‌ای عاری از ابهام و قابل طبقه‌بندی توصیف کرد؟ این‌ها سؤالاتی دشوارند و هیچ پاسخ مشخصی هنوز به آنها داده نشده است. در این بخش، به بررسی دستورهایی می‌پردازیم که به مهندسان نرم‌افزار آینده امکان گشت و گذار در کتابخانه‌های مؤلفه را می‌دهند. یک مؤلفه‌ی نرم‌افزار قابل استفاده‌ی مجدد را به طرق گوناگون می‌توان توصیف کرد، ولی یک توصیف ایده‌آل شامل مدل 3C (*مفهوم، محتوا و حیطه*[2]) می‌شود [Tra95]. *مفهوم* مؤلفه‌ی نرم‌افزار «توصیفی است از آنچه که مؤلفه انجام می‌دهد» [Whi95]. واسط مؤلفه به‌طور کامل توصیف می‌شود و معناشناسی (که در حیطه‌ی پیش‌شرط‌ها و پس‌شرط‌ها ارائه می‌گردد) تعیین می‌شود. مفهوم مؤلفه باید با هدف و مقصد مؤلفه ارتباط برقرار کنند. *محتوای* مؤلفه، چگونگی تحقق یافتن مفهوم آن را توصیف می‌کند. در اصل، محتوا، اطلاعاتی است که از دید

[1] به‌طور کلی، مقدمات DFR باید به‌عنوان بخشی از مهندسی دامنه چیده شود.

[2] Concept, Content, Context

کاربران اتفاقی پنهان می‌ماند و تنها کسانی که قصد اصلاح یا آزمون مؤلفه را دارند، باید از آن مطلع باشند. *حیطه*، جایگاه مؤلفه‌ی قابل استفاده‌ی مجدد را در دامنه‌ی قابلیت کاربرد آن تعیین می‌کند. یعنی حیطه با مشخص کردن ویژگی‌های مفهومی، عملیاتی و پیاده‌سازی، به مهندس نرم‌افزار این امکان را می‌دهد تا مؤلفه‌ی مناسبی را بیابد که خواسته‌های او را برآورده سازد.

ابزارهای نرم‌افزاری

CBSE

هدف: کمک به مدل‌سازی، طراحی، بازبینی و انسجام بخشی به مؤلفه‌های نرم‌افزاری به‌عنوان بخشی از یک سیستم بزرگ‌تر.

مکانیک: مکانیک این ابزارها متنوع است. به‌طور کلی، ابزارهای CBSE به یک یا چند قابلیت زیر کمک می‌کنند: مشخص‌سازی و مدل‌سازی معماری نرم‌افزار؛ مرور و گزینش مؤلفه‌های نرم‌افزاری در دسترس؛ انسجام بخشیدن به مؤلفه‌ها.

ابزارهای نمونه[1]

Component Source (www.componentsource.com) آرایه وسیعی از ابزارها و مؤلفه‌های نرم‌افزاری COTS فراهم می‌آورد که در استانداردهای متفاوت بسیار پشتیبانی می‌شوند.

Component Manager که توسط Flashline (www.flashline.com) ساخته شده است و «یک برنامه‌ی کاربردی است که استفاده‌ی مجدد از مؤلفه‌های نرم‌افزاری را میسر ساخته امکان آن را اندازه‌گیری می‌کند.»

Select Component Factory که توسط Select Business Solutions (www.selectbs.com) توسعه یافته است و «مجموعه‌ای منسجم از محصولات برای طراحی نرم‌افزار، بازبینی طراحی، مدیریت خدمات/مؤلفه‌ها، مدیریت خواسته‌ها و کدنویسی است.»

Software Through Pictures-ACD که توسط Aonix توزیع می‌شود (www.aonix.com) مدل‌سازی جامعی با استفاده از UML برای معماری مدل‌گرای OMG -یک روش باز برای CBSE- فراهم می‌سازد.

برای اینکه مفهوم، محتوا و حیطه در عمل قابل استفاده باشند، باید به یک الگوی مشخصات مستحکم تبدیل شود. درباره‌ی الگوهای مشخصات مربوط به مؤلفه‌های قابل استفاده‌ی مجدد، ده‌ها مقاله نوشته شده است (برای مروری بر آنها می‌توانید [Cec06] را ببینید.)

به کمک طبقه‌بندی می‌توانید مؤلفه‌های قابل استفاده‌ی مجدد را بیابید و بازیابی کنید، ولی برای انسجام اثربخش این مؤلفه‌ها باید یک محیط استفاده‌ی مجدد وجود داشته باشد. محیط استفاده‌ی مجدد، خصوصیات زیر را از خود به نمایش می‌گذارد.

- یک بانک اطلاعاتی از مؤلفه‌ها که قادر به ذخیره‌سازی مؤلفه‌های نرم‌افزاری و طبقه‌بندی اطلاعات لازم برای بازیابی این مؤلفه‌ها باشد.

[1] ذکر نام این ابزارها در اینجا به معنای تأیید آنها نیست بلکه به‌عنوان نمونه‌هایی از این ابزارها به نام آنها اشاره شده است. در اکثر موارد، نام این ابزارها توسط سازندگان‌شان ثبت تجاری شده است.

- یک سیستم مدیریت کتابخانه، دستیابی بانک اطلاعاتی را فراهم سازد.
- سیستم بازیابی مؤلفه‌های نرم‌افزاری (مثلاً یک واسطه درخواست اشیا) که کلاینت را قادر به بازیابی مولفه‌ها و سرویس‌ها از سرور کتابخانه سازد.
- ابزارهای CBSE که انسجام مؤلفه‌های دوباره استفاده شده در یک طراحی یا پیاده‌سازی جدید را پشتیبانی کنند.

هر کدام از این وظایف با یک کتابخانه‌ی استفاده‌ی مجدد تعامل دارند یا در محدودیت‌های آن تجسم پیدا می‌کند.

کتابخانه‌ی استفاده‌ی مجدد، یکی از عناصر یک مخزن نرم‌افزار بزرگتر است (فصل ۲۲) و تسهیلات لازم برای ذخیره‌سازی مؤلفه‌های نرم‌افزاری و گستره‌ی وسیعی از محصولات کاری با قابلیت استفاده‌ی مجدد (مانند مشخصات، طراحی‌ها، الگوها، چارچوب‌ها، قطعات کد، موارد آزمون و راهنماهای کاربران) را فراهم می‌سازد. این کتابخانه شامل یک بانک اطلاعاتی و ابزارهایی می‌شود که برای ارجاع به بانک اطلاعاتی و بازیابی مولفه‌ها از آن لازم هستند. الگوی طبقه‌بندی مؤلفه‌ها به‌عنوان مبنایی برای درخواست‌های کتابخانه‌ای عمل می‌کند.

ارجاع به بانک اطلاعاتی غالباً با به‌کارگیری عنصر حیطه‌ای از مدل 3C مشخص می‌شود، که قبلاً در این بخش شرح داده شد. اگر یک درخواست اولیه، فهرستی بلند بالا از مؤلفه‌های پیشنهادی را ارائه کند، این درخواست پالایش می‌شود تا فهرست کوچکتر شود. سپس اطلاعات محتوایی و مفهومی استخراج می‌شوند (پس از یافته شدن مؤلفه‌های پیشنهادی) تا به انتخاب مؤلفه‌ی مناسب کمک کنند.

خصوصیات کلیدی محیط مناسب برای استفاده‌ی مجدد کدام است؟

مرجع وب

مجموعه‌ای جامع از منابع مربوط به CSBE را در www.cbd-hq.com/ می‌توانید بیابید.

۱۰-۷ خلاصه

طراحی در سطح مولفه‌ها شامل یک سری فعالیت‌های می‌شود که به آهستگی از سطح انتزاع نمایش نرم‌افزار می‌کاهد. طراحی در سطح مؤلفه‌ها در نهایت، نرم‌افزار را در سطحی از انتزاع به نمایش می‌گذارد که به کدهای برنامه نزدیک است.

بسته به ماهیت نرم‌افزاری که قرار است ساخته شود، سه دیدگاه مختلف نسبت به طراحی در سطح مؤلفه‌ها وجود دارد. در دیدگاه شیءگرا، آن چه کانون توجه قرار می‌گیرد، تشریح کلاس‌های طراحی است که از هر دو دامنه‌ی مسأله و زیرساخت نتیجه می‌شود. در دیدگاه سنتی، سه نوع مؤلفه یا پیمانه داریم: پیمانه‌های کنترلی، پیمانه‌های دامنه مسأله و پیمانه‌های زیرساختی. در هر دو مورد، اصول و مفاهیم پایه طراحی اعمال می‌شوند که به ایجاد نرم‌افزارهایی با کیفیت بالا منجر می‌گردند. طراحی در سطح مؤلفه‌ها از دیدگاه فرایندی، از مؤلفه‌های نرم‌افزاری قابل استفاده‌ی مجدد و الگوهای طراحی بهره می‌برد که عناصر مهم مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها هستند.

چند اصل و مفهوم مهم، طراح را در تشریح کلاس‌ها راهنمایی می‌کنند. ایده‌های نهفته در اصل *باز-بسته* و اصل *وارونگی وابستگی*، و مفاهیمی از قبیل اتصال و یکپارچگی، مهندس نرم‌افزار را در ساخت مؤلفه‌هایی با قابلیت آزمون، پیاده‌سازی و نگهداری یاری می‌دهند. برای اجرای طراحی در سطح مؤلفه‌ها در این حیطه، کلاس‌ها با مشخص کردن جزئیات پیام رسانی، شناسایی واسطهای مناسب، تعیین جزئیات صفات و تعریف ساختمان داده‌ها برای پیاده‌سازی این صفات، توصیف جریان پردازشی در داخل هر عملیات و نمایش رفتار در سطح کلاس یا مؤلفه، تشریح می‌شوند. در هر حال، تکرار طراحی (بازآرایی)، فعالیتی ضروری است.

طراحی درسطح مؤلفه‌ها به روش سنتی به نمایش ساختمان داده‌ها، واسط‌ها و الگوریتم‌های یک پیمانه با جزئیات کافی برای راهنمایی تولید کدهای منبع به زبان برنامه‌نویسی نیاز دارد. برای این منظور، طراح از یکی از چند نمادگذاری طراحی استفاده می‌کند که جزئیات در سطح طراحی را در قالب‌های گرافیکی، جدول‌بندی‌شده یا متنی ارائه می‌دهند.

در طراحی در سطح مؤلفه‌ها برای برنامه‌های تحت وب دو مقوله‌ی محتوا و قابلیت عملیاتی که توسط سیستم مبتنی بر وب به کاربر تحویل می‌شود باید مد نظر قرار داده شوند. طراحی محتوا در سطح مولفه‌ها، اشیای محتوایی و شیوه‌ی بسته‌بندی آن‌ها برای ارائه به کاربر نهایی مورد توجه قرار می‌گیرد. طراحی عملیاتی برای برنامه‌های تحت وب توابع پردازشی را کانون توجه قرار می‌دهد که محتوا را دستکاری می‌کنند، محاسبات را انجام می‌دهند، از بانک اطلاعاتی استفسار می‌کنند و به آن دستیابی دارند و با سایر سیستم‌ها رابطه ایجاد می‌کنند. همه‌ی اصول و دستورالعمل‌های طراحی در سطح مؤلفه‌ها در اینجا نیز کاربرد دارند.

برنامه‌نویسی ساخت‌یافته یک فلسفه‌ی طراحی رویه‌ای است که تعداد و نوع ساختمان‌های منطقی به‌کاررفته در نمایش جزئیات الگوریتمی را محدود می‌سازد. هدف از برنامه‌نویسی ساخت‌یافته، کمک به طراح در تعریف الگوریتم‌هایی است که پیچیدگی کمتری دارند و لذا خواندن، آزمودن و نگهداری آن‌ها آسان‌تر است.

مهندسی نرم‌افزار مبتنی بر مؤلفه‌ها، مجموعه‌ای از مؤلفه‌های نرم‌افزاری در یک دامنه‌ی کاربرد خاص، شناسایی، ساخته، کاتالوگ‌بندی و توزیع می‌شوند. این مؤلفه‌ها غالباً برای استفاده در یک سیستم جدید، تطبیق داده و منسجم می‌شوند تا واجد شرایط لازم برای این منظور گردند. مؤلفه‌های قابل استفاده‌ی مجدد باید در محیطی طراحی شوند که ساختمان داده‌های استاندارد، پروتکل‌های واسط و صفات برنامه را برای هر دامنه کاربرد برقرار سازد.

مسائل و نکاتی برای تعمق

۱۰-۱ تعریف واژه‌ی *مؤلفه*، گاه دشوار است. نخست، تعریفی کلی از این واژه ارائه دهید و سپس برای نرم‌افزار سنتی و شیءگرا، تعاریفی صریح‌تر بیاورید. سرانجام، سه زبان برنامه‌نویسی انتخاب کنید که با آن‌ها آشنایی دارید و نشان دهید که در هر کدام، مؤلفه چگونه تعریف می‌شود.

۱۰-۲ چرا مؤلفه‌های کنترلی در نرم‌افزارهای سنتی مورد نیازند، ولی به‌طور کلی در نرم‌افزارهای شیءگرا به آن‌ها نیازی نیست؟

۱۰-۳ OCP را به بیان ساده شرح دهید. چرا ایجاد انتزاع‌هایی که به‌عنوان واسط میان مؤلفه‌ها عمل می‌کنند ضروری است؟

۱۰-۴ DIP را به بیان ساده شرح دهید. اگر طراحی بیش از حد به سفت‌شدگی وابسته باشد چه اتفاقی رخ می‌دهد؟

۱۰-۵ سه مؤلفه‌ای را که به تازگی ساخته‌اید، انتخاب کنید و انواع یکپارچگی را که هر یک از خود به نمایش می‌گذارد، ارزیابی کنید. اگر قرار بود مزیت اصلی یکپارچگی بالا را تعیین کنید، آن مزیت چه می‌بود؟

۱۰-۶ سه مؤلفه انتخاب کنید که به تازگی ساخته‌اید و انواع اتصال را که هر یک از خود به نمایش می‌گذارد، ارزیابی کنید. اگر قرار بود، مزیت اصلی اتصال پایین را تعیین کنید، آن مزیت چه می‌بود؟

۱۰-۷ آیا منطقی است که بگوییم مؤلفه‌های دامنه مسأله هرگز نباید اتصال خارجی از خود نشان دهند؟ اگر موافق هستید، کدام نوع از مؤلفه‌ها، اتصال خارجی از خود به نمایش می‌گذارند؟

۱۰-۸ (۱) یک کلاس طراحی تشریح شده، (۲) توصیفات واسط، (۳) یک نمودار فعالیت برای یکی از عملیات‌های درون کلاس و (۴) یک نمودار حالت مشروح برای یکی از کلاس‌های *SafeHome* که در فصل‌های قبل بحث شد، توسعه دهید.

۱۰-۹ آیا پالایش مرحله‌ای و بازآرایی، مفاهیمی یکسان هستند؟ اگر خیر، تفاوت آن‌ها در چیست؟

۱۰-۱۰ مؤلفه‌ی برنامه‌ی تحت وب چیست؟

۱۰-۱۱ بخش کوچکی از یک برنامه موجود (تقریباً ۵۰ تا ۷۵ خط کد منبع) را انتخاب کنید. ساختارهای برنامه‌نویسی ساخت‌یافته را با رسم کادرهای چهارگوش حول آن‌ها مشخص سازید. آیا این قطعه برنامه، ساختارهایی دارد که از فلسفه‌ی برنامه‌نویسی ساخت‌یافته عدول کنند؟ اگر پاسخ مثبت است، کد را طوری دوباره طراحی کنید که از ساختارهای برنامه‌نویسی ساخت‌یافته پیروی کند. اگر خیر، درباره‌ی کادرهایی که رسم کرده‌اید چه چیزی توجه شما را جلب کرده است؟

۱۰-۱۲ همه‌ی زبان‌های برنامه‌نویسی نوین، ساختارهای برنامه‌نویسی ساخت‌یافته را پیاده‌سازی می‌کنند. از سه زبان برنامه‌نویسی مثال بیاورید.

۱۰-۱۳ یک مؤلفه‌ی کوچک را که کدهای نوشته شده باشد، انتخاب کنید و آن را یا به‌کارگیری (۱) نمودار فعالیت، (۲) نمودار گردشی، (۳) جدول تصمیم‌گیری و (۴) PDL نمایش دهید.

۱۰-۱۴ چرا «قطعه‌بندی» طی فرایند مرور بر طراحی در سطح مؤلفه‌ها اهمیت دارد؟

فصل ۱۱

طراحی واسط کاربر

نگاهی گذرا

واسط کاربر چیست؟ در طراحی واسط کاربر، یک رسانه ارتباطی مؤثر میان انسان و کامپیوتر ایجاد می‌شود. طراح با دنبال کردن یک مجموعه از اصول طراحی واسط، اشیای واسط و عملیات را شناسایی کرده سپس یک آرایش از صفحه‌نمایش ایجاد می‌کند که مبنایی برای نمونه‌ی اولیه واسط کاربر تشکیل می‌دهد.

چه می‌کند؟ مهندس نرم‌افزار، واسط کاربر را با اجرای یک فرایند تکرار طراحی می‌کند که از اصول طراحی از پیش تعیین شده پیروی می‌کند.

چرا اهمیت دارد؟ اگر استفاده از نرم‌افزار دشوار است، اگر شما را به اشتباه می‌اندازد یا شما را در رسیدن به اهدافتان با اشکال مواجه می‌سازد، و یا همه‌ی قدرت محاسباتی یا عملکرد بالایی که دارد، از آن خوشتان نمی‌آید، باید واسط آن را تغییر دهید.

مراحل کار کدام است؟ طراحی واسط کاربر با شناسایی کاربر، وظیفه و خواسته‌های محیطی آغاز می‌شود. هنگامی که وظایف کاربر مورد شناسایی قرار گرفت، سناریوهای کاربر پس از ایجاد، مورد تحلیل قرار می‌گیرند تا یک مجموعه عملیات و اشیای واسط تعریف شود. اینها مبنایی برای آرایش صفحه‌نمایش تشکیل می‌دهند که طراحی گرافیکی و قرار دادن آیکون‌ها، تعریف متون توصیفی صفحه‌نمایش، مشخصات و عنوان‌بندی پنجره‌ها و مشخصات عناصر اصلی و فرعی منوها در آن تصویر شده است. برای تهیه یک نمونه‌ی اولیه و پیاده‌سازی نهایی مدل طراحی، از تعدادی ابزار استفاده می‌شود و نتیجه نهایی از نظر کیفیت ارزیابی می‌شود.

محصول کار چیست؟ سناریوهای کاربر ایجاد و آرایش‌های صفحه‌نمایش تولید می‌شوند. یک نمونه‌ی اولیه از واسط تهیه و به شیوه‌ای تکراری اصلاح می‌شود.

چگونه اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ نمونه‌ی اولیه توسط کاربر مورد آزمون قرار می‌گیرد و از بازخورد آزمون، برای اصلاح تکراری نمونه‌ی اولیه استفاده می‌شود.

ما در جهانی از محصولات با فن‌آوری بالا زندگی می‌کنیم و در واقع همه‌ی آنها – دستگاه‌های الکترونیکی، تجهیزات صنعتی، سیستم‌های شرکتی، سیستم‌های نظامی، نرم‌افزارهای کامپیوترهای شخصی و برنامه‌های تحت وب – به مقیاسی کیفی برای سنجش سهولت و بازدهی نحوه‌ی به کارگیری قابلیت‌ها و ویژگی‌های ارائه شده توسط این محصولات نیاز دارند.

واسط مورد نظر چه برای یک دستگاه پخش موسیقی دیجیتال طراحی شده باشد چه برای سیستم کنترل سلاح‌های یک هواپیمای جنگنده، آنچه که اهمیت دارد، قابلیت استفاده است. اگر سازوکارهای واسط از طراحی خوبی برخوردار باشند، کاربر با استفاده از ریتمی ملایم به تعامل با دستگاه می‌پردازد که به او امکان می‌دهد تا بدون هیچ گونه تلاش زیادی به اهداف خود دست پیدا کند. ولی اگر واسط، خوب طراحی نشده باشد، کاربر سردرگم می‌شود و نتیجه‌ی نهایی چیزی جز ناراحتی و بازدهی ضعیف نخواهد بود.

طی سه دهه‌ی نخست عصر کامپیوترها، قابلیت استفاده در میان سازندگان نرم‌افزار دغدغه‌ی اصلی به‌شمار نمی‌رفت. دانلد نورمن در کتاب کلاسیک خود درخصوص طراحی [Nor88] استدلال می‌کند که زمان تغییر رویکرد فرارسیده بود:

> برای ساخت فن‌آوری‌هایی که برازنده‌ی انسان باشند، مطالعه انسان ضروری است. ولی اکنون ما فقط تمایل داریم فن‌آوری را مطالعه کنیم. در نتیجه، انسان‌ها ناگزیرند از فن‌آوری پیروی کنند. زمان آن فرا رسیده است که این روند وارونه شود، وقت آن رسیده که فن‌آوری را وادار به دنباله‌روی از انسان کنیم.

در نتیجه‌ی مطالعاتی که کارشناسان فن‌آوری روی تعامل‌های انسانی به عمل آوردند، دو مسأله غالب برملا شد. نخست، مجموعه‌ای از قواعد طلایی (بخش ۱-۱۱) شناسایی شد. این قواعد در کلیه‌ی تعامل‌های انسان با محصولات فن‌آوری به کاربرده شدند. دوم، مجموعه‌ای از *سازوکارهای تعاملی* تعریف شدند تا طراح نرم‌افزار به کمک آنها بتواند سیستم‌هایی بسازد که قواعد طلایی را به‌طرزی شایسته پیاده‌سازی کنند. این سازوکارهای تعاملی، که در مجموع واسط *گرافیکی کاربر* (GUI) نام نهاده شده‌اند، برخی از برجسته‌ترین مشکلات واسط‌های انسانی را برطرف ساخته‌اند. ولی حتی در یک «دنیای ویندوزی» همه‌ی ما به واسط‌هایی برخورده‌ایم که فراگیری آنها دشوار است، یا آنها سخت می‌شود کار کرد، گیج کننده‌اند، عاری از هوشمندی‌اند، ارتکاب اشتباه در آنها قابل بازگشت نیست و در بسیاری موارد کلاً خسته کننده‌اند. با این وجود، کسی وقت صرف این واسط‌ها کرده است و به‌نظر نمی‌رسد که سازنده این مشکلات را از قصد ایجاد کرده باشد.

۱-۱۱ قواعد طلایی

تئو مندل در کتاب خود [Man97] سه قاعده طلایی برای طراحی واسط‌ها عنوان می‌کند:

۱. سپردن کنترل به کاربر
۲. کاستن از بار حافظه کاربر
۳. سازگار ساختن واسط

این قواعد طلایی، درواقع مبنایی برای مجموعه‌ای از اصول طراحی واسط کاربر تشکیل می‌دهند که این جنبه‌ی مهم از طراحی نرم‌افزار را هدایت می‌کنند.

۱-۱-۱۱ سپردن کنترل به کاربر

طی یک جلسه جمع‌آوری خواسته‌ها برای یک سیستم اطلاعاتی جدید و بزرگ، از یکی از کاربران اصلی درباره صفات واسط گرافیکی پنجره‌ای سؤال شد.

آن کاربر گفت: «آنچه من واقعاً دوست دارم، سیستمی است که فکر مرا بخواند؛ پیش از آنکه نیاز به انجام کاری داشته باشم آن را به‌راحتی برایم انجام دهد. فقط همین.»

اولین واکنش من این بود که سرم را تکان دهم و لبخندی بزنم، ولی پس از قدری تأمل دریافتم که درخواست آن کاربر به هیچ وجه بیراه نبوده است. او سیستمی می‌خواست که به نیازهای وی پاسخ دهد و به او کمک کند تا کارها را درست انجام دهد. می‌خواست کامپیوتر در کنترل او باشد نه او در کنترل کامپیوتر.

اکثر قیدوبندها و محدودیت‌های حاکم بر واسط، به منظور تسهیل شیوه‌ی تعامل از سوی طراح تحمیل می‌شوند. ولی برای چه کسی؟

در اکثر موارد، به‌عنوان طراح ممکن است قیدوبندها و محدودیت‌هایی را وارد کنید تا پیاده‌سازی واسط را تسهیل کند. نتیجه ممکن است واسطی باشد که ساخت آن آسان ولی استفاده از آن ناراحت‌کننده است. مندل [Man97] چند اصل طراحی را تعریف می‌کند که کنترل را در اختیار کاربر قرار می‌دهد:

تعریف شیوه‌های تعامل به طریقی که کاربر را وادار به انجام عملیات غیرضروری یا نامطلوب نکند. شیوه تعامل، حالت فعلی واسط است. برای مثال، اگر در یکی از منوهای واژه‌پردازی، گزینه‌ی *spell check* (چک کردن املا) انتخاب شود، نرم‌افزار به حالت چک کردن املا می‌رود. دلیلی وجود ندارد که اگر کاربر طی این کار مایل به انجام یک ویرایش متنی باشد، مجبور باشد کماکان در حالت چک کردن املا باقی بماند. کاربر باید بدون انجام کار زیاد، از حالتی به حالت دیگر برود.

انعطاف‌پذیری در تعامل. چون کاربران متفاوت دارای ترجیحات متفاوتی در تعامل با کامپیوتر هستند، باید گزینه‌هایی برای این منظور فراهم آورده شود. برای مثال، نرم‌افزار ممکن است امکان تعامل از طریق فرمان‌های صفحه‌کلید، حرکت ماوس، قلم دیجیتالی یا فرمان‌های تشخیص گفتاری را به کاربر بدهد. ولی هر عملی برای همه‌ی راهکارهای تعاملی مناسب نیست. برای مثال، فکر کنید رسم یک شکل پیچیده از طریق صفحه‌کلید چقدر ممکن است دشوار باشد.

امکان توقف تعامل و خنثی کردن عملیات توسط کاربر. حتی وقتی که کاربر عملیاتی را انجام داد، باید بتواند این عملیات را به منظور انجام امر دیگری متوقف کند (بدون از دست رفتن نتیجه عملیات). کاربر باید قادر به خنثی کردن نتیجه عملیات نیز باشد.

تعامل روان با پیشرفت سطح مهارت و امکان سفارشی کردن نوع تعامل. کاربران غالباً تعدادی عملیات را مکرراً باید انجام دهند. بنابراین خوب است یک راهکار «ماکرو» طراحی شود که کاربران ماهر بتوانند واسط را سفارشی کنند تا تعامل آسان شود.

پنهان کردن جزئیات فنی از دید کاربران موردی. واسط کاربر باید کاربر را به دنیای مجازی برنامه کاربردی منتقل کند. کاربر نباید از سیستم‌عامل، عملیات مدیریت فایل‌ها، یا فن‌آوری‌های کامپیوتری دیگر آگاه باشد. در اصل، واسط نباید کاربر را وادار به تعامل در سطح «داخل» ماشین کند (مثلاً کاربر نباید مجبور به تایپ فرمان‌های سیستم‌عامل از داخل برنامه کاربردی باشد).

> «بهتر است تجربه‌ی کاربر طراحی شود تا اینکه اصلاح شود.»
>
> **جان میدز**

> «همیشه آرزو داشتم استفاده از کامپیوتر به آسانی استفاده از تلفن باشد. این آرزویم به تحقق پیوسته است. دیگر نمی‌دانم چطور از تلفن خود استفاده کنم.»
>
> **بیارن اشتروپ اشتروپ**
> (مبدع C++)

طراحی برای تعامل مستقیم با اشیایی که روی صفحه ظاهر می‌شوند. وقتی کاربر بتواند اشیای لازم برای انجام یک وظیفه را به شیوه‌ای فیزیکی دستکاری کند، احساس می‌کند کنترل بیشتری در اختیار او است. برای مثال، واسطی که به کاربر امکان می‌دهد تا شیءای را حرکت دهد (اندازه آن را تغییر دهد)، نمونه‌ای از دستکاری مستقیم است.

۲-۱-۱۱ کاستن از بار حافظه کاربر

هرچه کاربر مجبور به حفظ جزئیات بیشتری باشد، احتمال خطای او در تعامل با سیستم بالاتر می‌رود. به همین دلیل است که در یک طراحی واسط کاربر خوب، به حافظه کاربر بهایی داده نمی‌شود. در صورت امکان، سیستم باید اطلاعات مربوط را به خاطر بیاورد و به کاربر یادآوری کند که چه باید بکند. مندل [Man97] اصول طراحی‌ای را معین می‌کند که واسط را قادر به کاهش دادن بار حافظه می‌کنند:

کاهش تقاضا برای حافظه کوتاه مدت. هنگامی که کاربران درگیر وظایف پیچیده می‌شوند، تقاضا برای حافظه کوتاه مدت، ممکن است چشمگیر شود. واسط باید طوری طراحی شود که به‌خاطر سپردن عملیات و نتایج گذشته کاهش یابد. این منظور را می‌توان با فراهم آوردن سرنخ‌های عینی برآورده نمود، به این ترتیب که این سرنخ‌ها کاربر را قادر به تشخیص عملیات گذشته کند و دیگر مجبور نباشد آنها را به خاطر بیاورد.

برقراری پیش‌فرض‌های بامعنی. مجموعه‌ای از پیش‌فرض‌های اولیه حداقل باید برای نیمی از کاربران معنا داشته باشد، ولی کاربر باید قادر به مشخص کردن ترجیحات شخصی خود باشد. در هر حال، یک گزینه «تنظیم مجدد» باید در دسترس باشد تا در صورت لزوم بتوان همه‌ی پیش‌فرض‌های اولیه را اعمال کرد.

تعریف میانبرهای هوشمندانه. هنگامی که برای دستیابی به عملکردهای سیستم از کلیدهای میانبر استفاده می‌شود (مثل Alt + P برای عمل چاپ)، بین کلید میانبر و آن عمل باید ارتباطی منطقی وجود داشته باشد که به‌خاطر آوردن آن آسان باشد (مثلاً می‌توان از حرف اول آن عمل استفاده نمود).

چیدمان بصری و دیداری واسط باید مبتنی بر استعاره جهان واقعی باشد. برای مثال، سیستم پرداخت حقوق باید از استعاره دسته چک و ثبت چک استفاده کند تا کاربر را در فرایند پرداخت چک راهنمایی کند. به‌این ترتیب، کاربر می‌تواند به جای حفظ یک سری تعامل‌های اسرارآمیز، بر درک سرنخ‌های عینی تکیه کند.

فاش کردن اطلاعات به شیوه‌ای تدریجی. واسط باید دارای سازمان‌دهی سلسله مراتبی باشد. یعنی اطلاعات مربوط به یک وظیفه، شیء، یا یک رفتار را باید ابتدا در سطح بالایی از انتزاع ارائه داد. جزئیات بیشتر را باید پس از اظهار تمایل کاربر با کلیک ماوس ارائه کرد. یک مثال، که در بسیاری از برنامه‌های کاربردی واژه‌پرداز متداول است، عمل خط کشیدن زیر حروف است. خود این عمل یکی از چند عملی است که در منوی *text style* (شیوه متن) قرار دارند. ولی همه‌ی قابلیت‌های خط‌کشی زیر متن ذکر نشده است. کاربر باید این گزینه را انتخاب کند تا همه‌ی حالت‌های آن (مثل یک خط، دو خط، خط مقطع) در آن ارائه شود.

SafeHome

عدول از یک قاعده‌ی طلایی

صحنه: کابین وینود، شروع طراحی واسط کاربر.

نقش‌آفرینان: وینود و جیمی، اعضای تیم مهندسی نرم‌افزار *SafeHome*

گفتگو:

جیمی: داشتم به واسط پایش منزل فکر می‌کردم.

وینود (با لبخند): فکرکردن خوب چیزی است.

جیمی: فکر کنم شاید بشود یک چیزهایی را ساده کرد.

وینود: منظور؟

جیمی: خب، اگر نقشه‌ی منزل را به‌طور کامل حذف کنیم، چه می‌شود؟ خیلی خوب است، ولی کار بیشتری می‌برد. در عوض، فقط از کاربر می‌خواهیم که دوربین مورد نظرش را مشخص کند و بعد تصاویر ویدیویی آن دوربین را در یک پنجره نشان می‌دهیم.

وینود: صاحبخانه چطوری باید یادش باشد که چند تا دوربین نصب شده است و کجای خانه؟

جیمی (قدری آشفته می‌شود): خب او صاحب خانه است، باید بداند.

وینود: ولی اگر ندانست.

جیمی: نخیر باید بداند.

وینود: قضیه این نیست ... اگر فراموش کرد؟

جیمی: ... می‌توانیم فهرستی از دوربین‌های در حال کار و محل آنها ارائه بدهیم.

وینود: بهتر شد. حداقل مجبور نیست چیزهایی را که ما به او ارائه می‌دهیم، به‌خاطر بسپارد.

جیمی (لحظه‌ای فکر می‌کند): ولی تو همان نقشه‌ی منزل را بیشتر دوست داری، نه؟

وینود: بله.

جیمی: فکر می‌کنی بازاریابی از کدام بیشتر خوشش بیاید؟

وینود: شوخی می‌کنی، نه؟

جیمی: نخیر.

وینود: ببین آنها چیزی را می‌خواهند که برق بزند. مهم نیست که ساختن آن چقدر ساده‌تر باشد.

جیمی (آهی می‌کشد): باشد، شاید از هر دو یک نمونه‌ی اولیه بسازم.

وینود: فکر خوبی است و بعد به مشتری اجازه می‌دهیم تا تصمیم بگیرد.

۳-۱-۱۱ سازگارساختن واسط

واسط باید اطلاعات را به شیوه‌ای سازگار دریافت و ارائه کند. یعنی (۱) همه‌ی اطلاعات بصری براساس یک استاندارد طراحی سازمان‌دهی می‌شوند که در تمام صفحات نمایش رعایت می‌شود؛ (۲) راهکارهای ورودی، به مجموعه‌ای محدود خلاصه می‌شوند که به طور سازگار در سرتاسر برنامه کاربردی مورد استفاده قرار می‌گیرد و (۳) راهکارهایی برای رفتن از وظیفه‌ای به وظیفه دیگر به‌طور

چیزهایی که متفاوت به نظر می‌رسند باید متفاوت عمل کنند. چیزهایی که یکسان به نظر می‌رسند، باید یکسان عمل کنند.

لری مازین

اطلاعات

قابلیت استفاده (usability)

لری کنستانتین در مقاله‌ای پربار درخصوص قابلیت استفاده [Con98] پرسشی مطرح می‌کند که با این موضوع ارتباطی نزدیک دارد: «خلاصه، کاربر چه می‌خواهد؟» و چنین پاسخ می‌دهد:

آنچه کاربران واقعاً می‌خواهند، ابزارهای خوب است. همه‌ی سیستم‌های نرم‌افزاری، از سیستم‌های عامل و زبان‌های برنامه نویسی گرفته تا برنامه‌های پشتیبان تصمیم گیری و واردکردن داده‌ها، همگی فقط ابزار هستند. کاربران نهایی از ابزارهایی که برای آنها مهندسی می‌کنیم، همان چیزی را می‌خواهند که ما از ابزارها می‌خواهیم. آنها سیستم‌هایی می‌خواهند که یادگیری‌شان آسان باشد و آنها را در کارشان یاری دهد. آنها نرم‌افزارهایی می‌خواهند که سرعت کارشان را کم نکند، آنها را سردرگم نکند، ارتکاب خطا را آسان و تمام‌کردن کار را دشوار نکند.

کنستانتین استدلال می‌کند که قابلیت استفاده از زیبایی شناسی، سازوکارهای تعاملی بسیار پیشرفته یا هوشمندی واسط به‌دست نمی‌آید، بلکه هنگامی رخ می‌دهد که معماری واسط با نیازهای کسانی که از آن استفاده می‌کنند، همخوانی داشته باشد.

تعریف رسمی قابلیت استفاده قدری گمراه کننده است. دانیهیو و همکاران او [Don99] آن را چنین تعریف می‌کنند: «قابلیت استفاده میزانی است از اینکه سیستم کامپیوتری چقدر خوب یادگیری را تسهیل می‌کند؛ به یادگیرندگان کمک می‌کند تا آنچه را که یاد گرفته‌اند، به‌خاطر آورند؛ احتمال خطاها را کاهش می‌دهد؛ آنها را قادر می‌سازند تا اثربخش باشند و کاری کند که از سیستم راضی باشند.»

تنها راه برای تعیین اینکه آیا «قابلیت استفاده» در سیستمی که در حال ساخت آن هستید، وجود دارد یا خیر، این است که ارزیابی یا آزمون «قابلیت استفاده» را اجرا کنید. کاربران را در حال تعامل با سیستم مشاهده کنید و به پرسش‌های زیر پاسخ دهید [Con95]:

- آیا سیستم بدون کمک یا راهنمایی پیوسته قابل استفاده هست؟
- آیا قواعد تعامل به کاربران آگاه کمک می‌کنند تا بازدهی کارشان بالا برود؟
- آیا با آگاه‌ترشدن کاربران، سازوکارهای تعامل، انعطاف پذیرتر می‌شوند؟
- آیا سیستم مطابق با محیط فیزیکی اجتماعی که در آن استفاده خواهد شد، تنظیم شده است؟
- آیا کاربر از حالت سیستم آگاه است؟ آیا کاربر در همه حال می‌داند کجاست؟
- آیا واسط به شیوه‌ای منطقی و سازگار سازمان دهی شده است؟
- آیا سازوکارهای تعامل، آیکون‌ها و روال‌ها در سرتاسر واسط سازگارند؟
- آیا تعامل، خطاها را پیش بینی و به تصحیح کاربر کمک می‌کند؟
- آیا واسط تحمل خطاهای مرتکب شده را دارد؟
- آیا تعامل ساده هست؟

اگر پاسخ هر کدام از این پرسش‌ها «آری» است، این احتمال هست که قابلیت استفاده، محقق شده باشد.

مزایای قابل سنجش بسیاری که از یک سیستم قابل استفاده به‌دست می‌آید عبارتند از [Don99]: افزایش فروش و رضایت‌مندی مشتری، مزیت رقابتی، نقدهای بهتر در رسانه‌ها، خوش‌نامی، کاهش هزینه‌های پشتیبانی، بهبود بهره وری کاربران نهایی، کاهش هزینه‌های آموزش، کاهش هزینه‌های مستند سازی، کاهش احتمال شکایت از سوی مشتریان ناراضی.

سازگار تعریف و پیاده‌سازی می‌شوند. مندل [Man97] مجموعه‌ای از اصول طراحی را تعریف می‌کند که به سازگارکردن واسط کمک می‌کند:

به کاربر اجازه دهید تا وظیفه کنونی را در زمینه معنی‌دار قرار دهد. بسیاری از واسط‌ها، لایه‌های پیچیده‌ای از تعامل را با ده‌ها تصویر در صفحه‌نمایش پیاده‌سازی می‌کنند. فراهم آوردن نشان‌گرهایی (مثل عناوین پنجره‌ها، آیکون‌های گرافیکی، ترکیب رنگ سازگار) که کاربر را قادر به تشخیص زمینه کاری می‌سازند، مهم است. به‌علاوه، کاربر باید بتواند تعیین کند که از کجا آمده است و برای رفتن به یک وظیفه جدید، چه راه‌هایی در اختیار دارد.

در میان خانواده‌ای از برنامه‌های کاربردی، سازگاری را حفظ کنید. مجموعه‌ای از برنامه‌های کاربردی (یا محصولات) باید با قواعد طراحی یکسان پیاده شود، به قسمی که سازگاری برای همه‌ی تعامل‌ها حفظ شود.

اگر مدل‌های تعامل گذشته انتظارات کاربر را برآورده کرده است، تغییری اعمال نکنید مگر آنکه دلیل قانع‌کننده‌ای برای آن داشته باشید. هنگامی که تعدادی از تعامل خاص به استانداردی غیررسمی تبدیل می‌شود (مثل استفاده از Alt + S برای ضبط فایل)، کاربر انتظار دارد این را در هر نرم‌افزاری ببیند. یک تغییر (مثل استفاده از Alt + S برای تغییر مقیاس) باعث سردرگمی می‌شود.

اصول طراحی مورد بحث در این بخش‌ها و بخش‌های پیشین، راهنمایی برای مهندسی نرم‌افزار به دست می‌دهد. در بخش‌های آینده، خود فرایند طراحی واسط را مورد بحث قرار خواهیم داد.

۱۱-۲ تحلیل و طراحی واسط کاربر

فرایند کلی برای طراحی یک واسط کاربر با ایجاد مدل‌هایی متفاوت از عملکرد سیستم (آن طور که از خارج به نظر می‌رسد) آغاز می‌شود. سپس وظایف انسان و وظایف کامپیوتر برای تحقق عملکرد سیستم، معین می‌شوند؛ مسائل طراحی که در کلیه طراحی‌های واسط کاربرد دارند، در نظر گرفته می‌شود؛ ابزارهایی برای ساخت نمونه‌ی اولیه و سرانجام پیاده‌سازی مدل طراحی به کار می‌روند و نتیجه از نظر کیفیت ارزیابی می‌شود.

۱۱-۲-۱ مدل‌های تحلیل و طراحی واسط

هنگام طراحی واسط کاربر، چهار مدل متفاوت باید در نظر گرفته شود، مهندس نرم‌افزار یک مدل طراحی ایجاد می‌کند، یک مهندس انسانی (یا مهندس نرم‌افزار) *مدل کاربر* را می‌سازد، مهندس نرم‌افزار یک *مدل طراحی* ایجاد می‌کند، کاربر نهایی یک تصویر ذهنی ایجاد می‌کند که غالباً *مدل ذهنی* یا *ادراک* سیستم خوانده می‌شود و پیاده‌کنندگان سیستم یک *مدل پیاده‌سازی* ایجاد می‌کنند. متأسفانه هر کدام از این مدل‌ها ممکن است با بقیه تفاوت داشته باشد. نقش طراح واسط، آشتی دادن این اختلافات و به‌دست‌آوردن نمایشی سازگار از واسط است.

مدل کاربر، پروفایل کاربران نهایی سیستم را مشخص می‌کند. جف پاتون در خصوص «طراحی کاربر-محور» چنین می‌نویسد [Pat07]:

مرجع وب

یک منبع عالی برای اطلاعات طراحی UI را می‌توان در www.useit.com یافت.

«اگر کلکی در واسط کاربر باشد، حتماً شکست خواهد خورد.»

داگلاس اندرسن

حقیقت این است که، طراحان و سازندگان -از جمله خود من- زیاد به کاربران فکر می‌کنند. ولی در غیاب یک مدل ذهنی قوی از کاربران ویژه، ما خودمان را جای کاربران می‌گذاریم و این «کاربر-محوری» نیست، خودمحوری است.

برای ساخت یک واسط کاربر موثر، «همه‌ی طراحی باید با شناخت کاربران هدف شروع شود که شامل پروفایل سنی، جنسی، توانایی‌های فیزیکی، تحصیلات، زمینه‌های فرهنگی و قومی، انگیزش، اهداف و شخصیت می‌شود» [Shn04]. به‌علاوه، کاربران را می‌توان به‌صورت‌های زیر گروه‌بندی کرد:

- *تازه کار.* فاقد دانش نحوی[1] از سیستم و با اندکی دانش معنایی[2] از برنامه کاربردی یا کامپیوتر به‌طور عام؛
- *کاربران مطلع و متوسط.* با دانش معنایی متعارف از برنامه کاربردی، ولی اطلاعات نحوی نسبتاً پایین درخصوص استفاده از واسط؛
- *کاربران مطلع و دائمی.* دارای دانش معنایی و نحوی خوبی که غالباً منجر به «سندرم کاربران قوی» می‌شود، یعنی افرادی به دنبال میانبر و کوتاه کردن راه‌های تعامل می‌گردند.

اندرز

حتی یک کاربر تازه کار نیز خواهان میانبر است؛ حتی کاربران آگاه و آزموده نیز گاهی به راهنمایی نیاز دارند. آنچه را که نیاز دارند به آنها بدهید.

مدل ذهنی کاربر (ادراک سیستم)، تصویری از سیستم است که کاربر نهایی در ذهن خود دارد. برای مثال، اگر از کاربر یک واژه‌پرداز خاص، خواسته می‌شد که عمل آن واژه‌پرداز را توصیف کند، برداشت سیستم، راهنمایی برای پاسخ به این پرسش می‌بود. صحت توصیف، بستگی به سابقه کاربر (مثلاً تازه‌کاران در بهترین حالت یک پاسخ ناقص می‌دادند) و آشنایی کلی با نرم‌افزار در دامنه کاربرد دارد. اگر کاربری واژه‌پردازها را کاملاً بشناسد، ولی با واژه‌پرداز خاصی فقط یک بار کار کرده باشد، توصیفی که ارائه می‌کند، نسبت به کاربر تازه‌کاری که یک هفته با این واژه‌پرداز کار کرده است، کامل‌تر است.

نکته‌ی کلیدی

مدل ذهنی کاربر به چگونگی ادراک کاربر از واسط و اینکه آیا UI نیازهای کاربر را برآورده می‌سازد، شکل می‌دهد.

مدل پیاده‌سازی، نمای خارجی سیستم کامپیوتری (ظاهر و نمای واسط) را با اطلاعات پشتیبان (کتب، جزوات راهنما، نوارهای ویدیویی، فایل‌های راهنما) ترکیب کرده ساختار نحوی و واسط معنایی سیستم را توصیف می‌کند. هنگامی که تصویر سیستم و برداشت از سیستم بر هم انطباق یابند، کاربران احساس راحتی کرده از نرم‌افزار به‌طور اثربخش استفاده می‌کنند. برای دستیابی به این ترکیب از مدل‌ها، مدل طراحی باید طوری توسعه یافته باشد که اطلاعات موجود در مدل کاربر را در خود جای دهد و تصویر سیستم باید اطلاعات معنایی و نحوی مربوط به واسط را به‌درستی منعکس کند.

مدل‌های توصیف شده در این بخش «چکیده‌ای هستند از آن چیزی که کاربر دارد انجام می‌دهد یا تصور می‌کند که در حال انجام دادن آن است، یا چکیده‌ای از چیزی است که کاربری تصور می‌کند باید هنگام استفاده از سیستم تعاملی انجام دهد.» [Mon84] در اصل، این مدل‌ها، طراح واسط را قادر می‌سازند که یک عنصر کلیدی از مهمترین اصل طراحی واسط کاربر را برآورده سازد: «شناخت کاربر، شناخت وظایف».

«به آنچه که کاربران انجام می‌دهند، توجه کنید به آنچه که می‌گویند.»

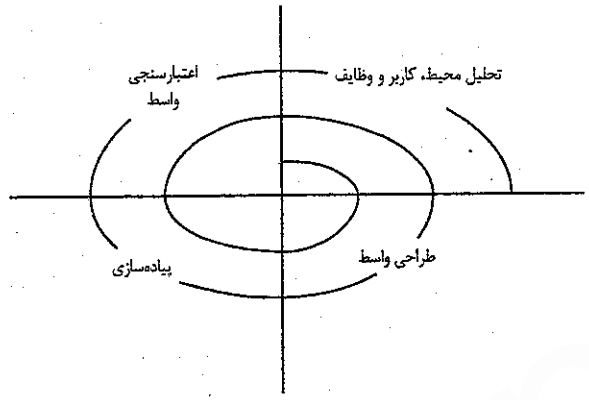
یاکوب نیلسن

[1] در این حیطه، منظور از *دانش نحوی* مکانیک تعاملی است برای استفاده ی اثربخش از واسط مورد نیاز است.

[2] منظور از *دانش معنایی* درک زیربنایی از کاربرد است، یعنی درک وظایفی که به انجام می‌رسد، معنای ورودی و خروجی و اهداف و مقاصد سیستم.

۲-۲-۱۱ فرایند

فرایند طراحی و تحلیل واسط‌های کاربر، طبیعتی تکراری دارد و با استفاده از مدل مارپیچی مشابه با آنچه که در فصل ۲ بحث شد، قابل ارائه است. رجوع به شکل ۱-۱۱ نشان می‌دهد که فرایند تحلیل و طراحی واسط کاربر از درون مارپیچ آغاز می‌شود و شامل چهار فعالیت چارچوبی متمایز می‌شود [Man97]: (۱) تحلیل و مدل‌سازی واسط، (۲) طراحی واسط (۳) پیاده‌سازی واسط (۴) اعتبارسنجی واسط. مارپیچ شکل ۱-۱۱ نشان می‌دهد که هر یک از وظایف بالا بیش از یک بار رخ می‌دهند و با هر بار دور زدن مارپیچ، خواسته‌های بیشتری آشکار می‌شود. در اکثر موارد، فعالیت پیاده‌سازی شامل ساخت نمونه‌ی اولیه - یعنی تنها راه عملی برای اعتبارسنجی آنچه که طراحی شده است - می‌شود.



شکل ۱-۱۱ فرایند طراحی واسط کاربر.

در *تحلیل واسط*، سابقه کاربرانی کانون توجه قرار می‌گیرد که با سیستم تعامل می‌کنند. سطح مهارت، شناخت تجارت و پذیرش سیستم ثبت می‌شود و گروه‌های کاربر متفاوت، تعریف می‌شوند. خواسته‌های هر یک از گروه‌های کاربران مشخص می‌شود. در اصل، مهندس نرم‌افزار کوشش می‌کند تا برداشت سیستم را برای هر طبقه از کاربران بشناسد (بخش ۱-۲-۱۱).

هنگامی که خواسته‌های کلی معین شد، تحلیل مفصل‌تری از وظایف اجرا می‌شود. آن دسته از وظایفی که کاربر انجام می‌دهد تا به اهداف سیستم دست پیدا کند، توصیف و پیاده‌سازی می‌شوند (البته با چند دور تکرار در مارپیچ). تحلیل وظایف را به طور مفصل‌تر در بخش ۳-۱۱ مورد بحث قرار می‌دهیم. سرانجام، در تحلیل محیط کاربر، محیط کار فیزیکی است که کانون توجه قرار می‌گیرد. از جمله پرسش‌هایی که باید مطرح شوند، عبارتند از:

- واسط از نظر فیزیکی در کجا قرار داده خواهد شد؟
- آیا کاربر به حالت نشسته کار می‌کند یا ایستاده، یا کارهای دیگری انجام می‌دهد که با واسط بی‌ارتباط هستند؟
- آیا سخت‌افزار واسط محدودیت‌های جا، نور و سروصدا را در برمی‌گیرد؟
- آیا ملاحظات خاصی درباره عوامل انسانی وجود دارد که عوامل محیطی سبب آنها بوده باشد؟

«بهتر است تجربه کاربر طراحی شود تا اینکه اصلاح شود.»

جان میدر

هنگامی که طراحی UI را آغاز می‌کنیم، درباره محیط چه باید بدانیم؟

اطلاعاتی که به‌عنوان بخشی از فعالیت تحلیل جمع‌آوری شده‌است، جهت ایجاد مدل تحلیل برای واسط به کار می‌رود. با استفاده از این مدل به‌عنوان مبنا، فعالیت طراحی شروع می‌شود.

هدف *طراحی واسط*، تعریف یک مجموعه از اشیا و عملیات واسط (و نمایش آنها در صفحه‌نمایش) است که کاربر را قادر به انجام کلیه وظایف تعریف شده می‌سازد، به طوری که همه‌ی اهداف قابلیت استفاده را برای یک سیستم برآورده سازد. درباره طراحی واسط به تفصیل بیشتر در بخش ۴-۱۱ بحث خواهیم کرد.

ساخت واسط معمولاً با ایجاد یک نمونه‌ی اولیه آغاز می‌شود که ارزیابی سناریوهای به‌کارگیری واسط را امکان‌پذیر می‌سازد. به موازاتی که فرایند طراحی تکراری ادامه می‌یابد، می‌توان از یک کیت‌ابزار واسط (بخش ۵-۱۱) برای کامل کردن ساختار واسط استفاده کرد.

اعتبارسنجی واسط بر این موارد تأکید دارد: (۱) توانایی واسط در پیاده‌سازی کلیه وظایف، انجام تمام وظایف و دستیابی به کلیه خواسته‌های عمومی کاربر؛ (۲) میزان سهولت استفاده و فراگیری واسط، و (۳) تلقی کاربران از واسط به‌عنوان یک ابزار مفید در کارهای آنها.

چنان که قبلاً نیز گفته شد، فعالیت‌های توصیف شده در این بخش به‌صورت تکراری رخ می‌دهند. بنابراین، نیازی به مشخص‌کردن همه‌ی جزئیات در همان گذر نخست (برای مدل تحلیل یا طراحی) نیست. با گذرهای بعدی از فرایند، جزئیات وظایف، اطلاعات طراحی و ویژگی‌های عملیاتی واسط بیشتر مشخص می‌شود.

۳-۱۱ تحلیل واسط[1]

یک اصل کلیدی در تمامی مدل‌های فرایند مهندسی نرم‌افزار این است: *درک مسأله قبل از این که اقدام به طراحی راهکار کنید*. در مورد طراحی واسط کاربر، درک مسأله به معنای شناخت (۱) افرادی (کاربران نهایی) است که از طریق واسط با سیستم تعامل می‌کنند، (۲) وظایفی که کاربران نهایی باید انجام دهند تا کار پیش برود، (۳) محتویاتی که به‌عنوان بخشی از واسط عرضه می‌شود و (۴) محیطی که این وظایف در آن اجرا خواهد شد. در بخش‌هایی که به دنبال خواهد آمد، هر کدام از این عناصر تحلیل واسط را به هدفِ بنانهادن بستری مستحکم برای وظایف طراحی آتی بررسی خواهیم کرد.

۱-۳-۱۱ تحلیل کاربران

عبارت «واسط کاربر» احتمالاً تنها توجیهی است که برای صرف زمان جهت شناخت کاربر قبل از پرداختن به موارد فنی لازم است. پیش از این گفتیم که هر کاربر، تصویری ذهنی از نرم‌افزار دارد که ممکن است با تصویر ذهنی سایر کاربران تفاوت داشته باشد. به‌علاوه، تصویر ذهنی کاربر ممکن است تفاوتی گسترده با مدل طراحی مهندس نرم‌افزار داشته باشد. تنها راه برای همگراکردن این تصویر ذهنی و مدل طراحی، این است که خود کاربران و نیز شیوه استفاده‌ی آنها از سیستم را درک کنید. اطلاعات به‌دست آمده از منابع گوناگون را می‌توان برای نیل به این مقصود به کاربرد:

[1] منطقی است که این بخش در فصل های ۵، ۶ یا ۷ آورده شود زیرا مسائل مربوط به تحلیل خواسته‌ها در آن فصل‌ها مطرح شد. ولی آن را در اینجا آوردیم چون تحلیل واسط و طراحی واسط ارتباطی تنگاتنگ با هم دارند و مرز میان این دو چندان مشخص نیست.

مصاحبه با کاربران. در مستقیم‌ترین روش، اعضای تیم نرم‌افزاری با کاربران نهایی ملاقات می‌کنند تا نیازها، انگیزش‌ها، فرهنگ کاری ایشان و بسیاری از مسائل دیگر را درک کنند. این هدف از طریق جلسات یک به یک یا گروه‌های کانونی (focus group) قابل حصول است.

ورودی فروش. کارمندان بخش فروش به‌صورت مرتب با کاربران ملاقات می‌کنند و می‌توانند اطلاعاتی جمع‌آوری کنند که به تیم نرم‌افزاری کمک می‌کند تا کاربران را گروه‌بندی کنند و خواسته‌های آنها را بهتر بشناسند.

ورودی بازاریابی. تحلیل بازار در تعریف بخش‌های مختلف بازار و درک اینکه هر کدام از این بخش‌ها چگونه از نرم‌افزار به شیوه‌هایی با تفاوت‌های ظریف استفاده می‌کنند، می‌تواند بسیار ارزشمند باشد.

ورودی پشتیبانی. صحبت‌های روزمره‌ی کارمندان بخش پشتیبانی با کاربران. این صحبت‌ها محتمل‌ترین منبع اطلاعات درباره مواردی است که به دنبال می‌آید: چیزهایی که کار می‌کنند و چیزهایی که کار نمی‌کنند، کاربران چه چیزهایی را دوست دارند و چه چیزهایی را دوست ندارند، چه ویژگی‌هایی تولید اشکال می‌کنند و استفاده از چه ویژگی‌هایی آسان است.

مجموعه پرسش‌های زیر، برگرفته‌شده از [Hac98]، شما را در شناخت بهتر کاربران سیستم یاری می‌دهد:

- آیا کاربران، افراد حرفه ای، فنی، کارمند یا کارگر تولید هستند؟
- کاربر متوسط از چه سطح تحصیلات برخوردار است؟
- آیا کاربران قادر به فراگیری از مطالب کتبی هستند یا تمایل به آموزش‌های کلاسی را از خود نشان داده‌اند؟
- آیا کاربران، تایپیست حرفه‌ای هستند یا از صفحه کلید خوششان نمی‌آید؟
- گستره سنی جامعه‌ی کاربران چگونه است؟
- آیا یک جنس (مونث یا مذکر) در میان کاربران غالب است؟
- شیوه پاداش‌دهی به کاربران برای آنچه انجام می‌دهند، چگونه است؟
- آیا کاربران در ساعات عادی کار می‌کنند یا آن قدر کار می‌کنند تا وظیفه‌شان را به‌انجام برسانند.
- آیا قرار است نرم‌افزار بخشی از کار روزمره کاربران باشد یا فقط هر از چندگاهی از آن استفاده خواهند کرد؟
- زبان اصلی که کاربران به آن تکلم می‌کنند، چیست؟
- اگر کاربری هنگام استفاده از سیستم، مرتکب خطا شود، پیامدهای آن چه خواهد بود؟
- آیا کاربران در موضوع کار سیستم کارشناس هستند؟
- آیا کاربران مایل هستند درباره فن‌آوری‌ای که ورای واسط قرارداد، اطلاعات داشته باشند؟

هنگامی که به این پرسش‌ها پاسخ داده شود، به هویت کاربران خود پی خواهید برد؛ خواهید دانست چگونه می‌توان در آنها ایجاد انگیزه کرد و محیط کار را برای آنها دلپذیر ساخت و چگونه آنها را در دسته‌های متفاوت کاربری گروه‌بندی کرد، مدل ذهنی آنها از سیستم چیست و واسط کاربر را چگونه می‌توان مشخص کرد به‌طوری که نیازهای آنها را برآورده سازد.

چگونه دریابیم که کاربر چه انتظاری از واسط دارد؟

اندرز

پیش از هر چیز، زمانی را صرف صحبت کردن با کاربران واقعی کنید، ولی مراقب باشید. یک ایده‌ی قوی الزاماً به این معنی نیست که اکثریت کاربران با آن موافق باشند.

نکته‌ی کلیدی

درباره‌ی جمعیت‌شناسی و خصوصیات کاربران نهایی چه می‌دانیم؟

۲-۳-۱۱ مدل‌سازی و تحلیل وظایف

هدف تحلیل وظایف، پاسخ دادن به پرسش‌های زیر است:

- کاربر در شرایط خاص چه کاری انجام می‌دهد؟
- همچنان که کاربر، کارش را انجام می‌دهد چه وظایف یا زیروظایفی انجام می‌شود؟
- کاربر هنگام انجام کار، کدام اشیای مربوط به دامنه‌ی مسأله‌ی خاص را دستکاری خواهد کرد؟
- ترتیب انجام وظایف کاری (جریان کار) چیست؟
- سلسله مراتب وظایف چیست؟

نکته‌ی کلیدی

هدف کاربر، انجام یک یا چند وظیفه از طریق UI است. برای این منظور، UI باید سازوکارهایی فراهم سازد که به کاربر امکان دهد تا به اهداف خود دست پیدا کند.

برای پاسخ گفتن به این پرسش‌ها، باید فنونی را به کار بگیرید که قبلاً در این کتاب بحث شد، ولی در این مورد، فنون ذکر شده برای واسط کاربر استفاده می‌شوند.

use case‌ها. در فصل‌های قبل آموختید که use case، شیوه‌های تعامل کنش‌گر (در حیطه‌ی طراحی واسط، کنش‌گر همواره انسان است) با سیستم را توصیف می‌کند. use case، در صورت استفاده به‌عنوان بخشی از تحلیل وظایف، طوری توسعه داده می‌شود که چگونگی انجام وظایف خاص توسط کاربر نهایی را نشان دهد. در اکثر موارد، use case به سبکی غیررسمی (در یک پاراگراف ساده) و از زبان اول شخص نوشته می‌شود. برای مثال، فرض کنید که یک شرکت نرم‌افزاری کوچک می‌خواهد یک سیستم طراحی به‌کمک کامپیوتر (CAD) بسازد که به صراحت برای طراحان داخلی کاربرد دارد. برای درک بهتر چگونگی انجام کار توسط طراحان داخلی از آنها خواسته می‌شود تا یک وظیفه‌ی طراحی خاص را توصیف کنند. وقتی از طراح داخلی پرسیده می‌شود: «چطور تصمیم می‌گیری که اثاثیه یک اتاق را کجا قرار دهی؟» او use case زیر را می‌نویسد:

> برای شروع، نقشه‌ی پلان اتاق را می‌کشم و ابعاد و موقعیت قرار گرفتن پنجره‌ها و درها را مشخص می‌کنم. نوری که از بیرون وارد اتاق می‌شود، برای من خیلی اهمیت دارد و همچنین نمای بیرونی پنجره‌ها (اگر نمای زیبایی باشد، می‌خواهم توجه را به آن جلب کنم)؛ و نیز طول دیوارهای بدون حائل و بالاخره جریان حرکت در اتاق. سپس به فهرست اثاثیه مشتری و اثاثیه‌ای که خودم انتخاب کرده‌ام، نگاه می‌کنم —میزها، صندلی‌ها، کاناپه‌ها، کابینت‌ها، و اثاثیه‌ای که خودم انتخاب کرده‌ام، فهرستی از اشیای تزئینی —لامپ‌ها، قالیچه‌ها، نقاشی‌ها، مجسمه‌ها، گیاهان، قطعات کوچکتر یادداشت‌هایی درخصوص خواسته‌های خاص مشتری برای قرار دادن اشیا. سپس هر آیتم را با استفاده از قالبی که در مقیاس نقشه پلان رسم شده است، از فهرست هایم ترسیم می‌کنم. هر آیتمی را که رسم می‌کنم نشان گذاری می‌کنم و از مداد استفاده می‌کنم چون همیشه جای اشیا را تغییر می‌دهم. چند وضعیت متفاوت برای قرار دادن هر کدام در نظر می‌گیرم و سپس بهترین آنها را انتخاب می‌کنم. بعد نمایی سه بعدی (راندو شده) از اتاق رسم می‌کنم تا مشتری از ظاهر و نمای اتاق تصویری به‌دست آورد.

این use case، توصیفی پایه‌ای از یک وظیفه کاری مهم برای سیستم «طراحی به‌کمک کامپیوتر» به‌دست می‌دهد که از آن می‌توانید وظایف، اشیا، و جریان کلی تعامل را استخراج کنید. به‌علاوه، سایر ویژگی‌های سیستم که طراح داخلی را خشنود می‌سازد نیز ممکن است لحاظ شود. برای مثال، یک عکس دیجیتالی از نمای بیرونی هر پنجره ممکن است گرفته شود. هنگامی که وضعیت اتاق مشخص شد، نمای خارجی هر پنجره را می‌توان مشاهده کرد.

SafeHome

UI*ها برای طراحی *usecase

صحنه: کابین وینود، ادامه طراحی واسط کاربر.

نقش‌آفرینان: وینود و جیمی، اعضای تیم نرم‌افزاری *SafeHome*

گفتگو:

جیمی: با واسطمان در بخش بازاریابی تماس گرفتم و از او خواستم برای واسط پایش، یک use case بنویسد.

وینود: از دیدگاه چه کسی؟

جیمی: صاحبخانه. مگر کس دیگری هم هست؟

وینود: نقش مدیر سیستم هم هست که حتی اگر صاحبخانه عهده‌دار آن باشد، دیدگاه متفاوتی است. «مدیر» سیستم را پیکربندی می‌کند، وضعیت اجزای آن را مشخص می‌کند، چیدمان نقشه‌ی پلان را تعیین می‌کند، محل قرار گرفتن دوربین‌ها را معین می‌کند.

جیمی: تنها کاری که ازش خواستم، این بود که نقش صاحبخانه را بازی کند که مثلاً دارد ویدیوها را ببیند.

وینود: خوب است. این یکی از رفتارهای اصلی است که واسط قابلیت پایش باید داشته باشد، ولی ما باید رفتار مدیر را هم بررسی کنیم.

جیمی (آشفته): درست می‌گویی.

[جیمی بیرون می‌رود که واسط بازاریابی را پیدا کند و چند ساعت بعد برمی‌گردد]

جیمی: خوشحالم بودم که او را پیدا کردم و با هم روی use case مدیر سیستم کار کردیم. اساساً می‌خواهیم «مدیر» را به‌عنوان یک قابلیت عملیاتی تعریف کنیم که در همه‌ی قابلیت‌های عملیاتی *SafeHome* قابل استفاده باشد. نتیجه‌ای که گرفتیم، این بود.

[جیمی use case غیررسمی زیر را به وینود نشان می‌دهد]

use case‌های غیررسمی: باید بتوانم چیدمان سیستم را در هر زمان که بخواهم پیکربندی کنم یا تغییر دهم. هنگامی که سیستم را پیکربندی می‌کنم، قابلیت عملیاتی با عنوان «مدیر» سیستم را انتخاب کنم که از من می‌پرسد آیا می‌خواهم پیکربندی جدیدی انجام دهم یا یک پیکربندی موجود را ویرایش کنم. اگر پیکربندی جدیدی را انتخاب کنم، سیستم یک صفحه ترسیم به نمایش در آورد که به کمک آن بتوانم نقشه پلان را روی یک شبکه شطرنجی رسم کنم. وجود آیکون‌هایی برای دیوارها، پنجره‌ها و درها، کار ترسیم را آسان می‌سازد. کافی است آیکون را به اندازه لازم استاد دهم. سیستم، طول‌ها را برحسب متر و فوت نشان خواهد داد. (بتوانم سیستم اندازه گیری را انتخاب کنم.) بتوانم از کتابخانه حس‌گرها و دوربین‌ها، مورد دلخواه را انتخاب کنم و آنها را در نقشه پلان قرار دهم. بتوانم تنظیمات حس‌گرها و دوربین‌ها را از منوهای مناسب انجام دهم. اگر حالت ویرایش را انتخاب کنم، بتوانم دوربین‌ها یا حس‌گرها را جابه جا کنم، دوربین‌ها یا حس‌گرهای جدید اضافه کنم یا اشیای موجود را حذف کنم. نقشه پلان را ویرایش کنم و تنظیمات حس‌گر و دوربین‌ها را اصلاح کنم. در هر حال، انتظار دارم سیستم، سازگاری‌ها را چک کند و مرا در پرهیز از خطا یاری دهد.

وینود (پس از خواندن سناریو): خوب است. احتمالاً الگوهای طراحی مفید [فصل ۱۲] یا مولفه‌های قابل استفاده‌ی مجدد برای GUIهای برنامه ترسیم موجود باشند. شک ندارم که می‌توانیم بخشی از واسط «مدیر» یا قسمت زیادی از آن را پیاده‌سازی کنیم.

جیمی: موافقت شد. من یک نگاهی به آن می‌کنم.

پرداختن به جزئیات وظایف. در فصل ۸ تشریح مرحله‌ای (یا تجزیه عملیاتی یا پالایش مرحله‌ای) را به‌عنوان راهکاری برای پالایش وظایف پردازشی مورد بحث قرار دادیم که برای دستیابی نرم‌افزار به یک عملکرد مطلوب، لازم‌اند. بعداً در این کتاب، تحلیل شیء‌گرا را به‌عنوان یک روش مدل‌سازی برای سیستم‌های کامپیوتری مورد بحث قرار خواهیم داد. در تحلیل وظایف مربوط به طراحی واسط، برای کمک به درک فعالیت‌های انسانی که واسط کاربر باید به آنها پاسخ‌گو باشد، از یک رویکرد تشریحی یا شیء‌گرا استفاده می‌شود.

تحلیل وظایف را به دو شیوه می‌توان اجرا کرد. چنان‌که پیش از این نیز گفتیم، یک سیستم کامپیوتری تعاملی به جای فعالیت‌های دستی یا نیمه دستی به کار می‌رود. برای درک وظایفی که باید اجرا شوند تا هدف فعالیت برآورده شود، مهندس نیروی انسانی باید وظایفی را بشناسد که انسان‌ها در حال حاضر انجام می‌دهند (هنگام استفاده از یک روش دستی) و سپس آنها را در یک مجموعه مشابه (ولی نه الزاماً همسان) از وظایف که در زمینه واسط کاربر پیاده‌سازی می‌شود، نگاشت کند. به طریق دیگر، مهندس نیروی انسانی می‌تواند مشخصه‌ی موجود برای راهکار کامپیوتری را مطالعه کند و مجموعه‌ای از وظایف کاربر را به دست آورد که مدل کاربر، مدل طراحی و برداشت از سیستم را در بر گیرد.

روش کلی تحلیل وظایف هر چه که باشد، مهندس نیروی انسانی باید ابتدا وظایف را تعریف و طبقه‌بندی کند. پیش از این متذکر شدیم که یک روش، تلاشی مرحله‌ای است. برای مثال، فرض کنید که یک شرکت نرم‌افزاری کوچک می‌خواهد یک سیستم طراحی کامپیوتری را برای طراحان داخلی بسازد. مهندس با مشاهده یک طرح داخلی در حال کار، متوجه می‌شود که طراحی داخلی از چند فعالیت اصلی تشکیل شده است: چیدن اثاثیه، انتخاب مواد و اجناس، انتخاب پوشش برای درها و پنجره‌ها، ارائه (به مشتری)، تعیین هزینه‌ها و خرید. هر یک از این وظایف اصلی را می‌توان به چند وظیفه فرعی تقسیم کرد. برای مثال، با استفاده از اطلاعات موجود در use case چیدن اثاثیه را می‌توان به وظایف زیر پالایش کرد: (۱) ترسیم طرح کف اتاق براساس ابعاد؛ (۲) قراردادن پنجره‌ها و درها در مکان‌های مناسب؛ (۳الف) استفاده از قالب‌های اثاثیه برای رسم آنها در مقیاس طرح کف اتاق؛ (۳ب) استفاده از قالب‌های تاکیدی برای رسم آنها در مقیاس طرح کف اتاق؛ (۴) جابجا کردن طرح اثاثیه‌ها برای رسیدن به بهترین حالت؛ (۵) نشانه‌گذاری طرح کلیه اثاثیه؛ (۶) رسم ابعاد برای نشان دادن موقعیت‌ها (۷) رسم نمای پرسپکتیو برای مشتری. برای هر یک از وظایف اصلی دیگر نیز می‌توان از روشی مشابه استفاده کرد.

وظایف فرعی ۱ تا ۷ را می‌توان باز هم پالایش کرد. وظایف فرعی ۱ تا ۶ با دستکاری اطلاعات و اجرای عملیات در واسط کاربر اجرا خواهند شد. از طرفی دیگر، وظیفه فرعی ۷ را می‌توان به طور خودکار در نرم‌افزار اجرا کرد که با کاربر تعامل دارد[1]. مدل طراحی واسط باید هر یک از این وظایف را به شیوه‌ای اسکان دهد که با مدل کاربر (سابقه یک طراحی داخلی «معمولی») و برداشت سیستم (آنچه که طراح از یک سیستم خودکار انتظار دارد)، سازگار باشد.

اندرز

پرداختن به جزئیات وظایف کاملاً مفید است، ولی می‌تواند خطرناک هم باشد. فقط به صرف اینکه جزئیات یک وظیفه را تعیین کرده‌اید، نمی‌توانید فرض کنید که راه دیگری برای انجام آن وجود ندارد و راه دیگر هنگامی انتخاب می‌شود که UI پیاده‌سازی می‌شود.

[1] به‌هرحال، ممکن است چنین نباشد. طراح داخلی ممکن است بخواهد پرسپکتیو را که قرار است ترسیم گردد، کاربرد رنگ‌ها، و سایر اطلاعات را مشخص سازد. پرونده کاربرد مرتبط با پرداخت ترسیم پرسپکتیو، اطلاعات مورد نیاز برای پرداختن به این وظیفه را فراهم می‌سازد.

پرداختن به جزئیات اشیا. به‌جای توجه به وظایفی که کاربر باید انجام دهد، می‌توانید استفاده از use case و سایر اطلاعات به‌دست آمده از کاربر را بررسی کنید و اشیای فیزیکی مورد استفاده در طراحی داخلی را استخراج کنید. این اشیا را می‌توان در قالب چند کلاس گروه‌بندی کرد. صفحات هر کلاس تعریف می‌شود و با انجام ارزیابی کنش‌های هر شیء، فهرستی از عملیات‌ها تهیه می‌شود. برای مثال، قالب اثاثیه را می‌توان به کلاسی به نام **Furniture** با صفاتی از قبیل location size shape و غیره تبدیل کرد. طراح داخلی، شیء را از کلاس **Furniture** انتخاب می‌کند، آن را به مکانی در نقشه پلان (شیء دیگری در این حیطه) منتقل می‌کند، طرح‌بندی اثاثیه را ترسیم می‌کند و ... وظایف *انتخاب کردن*، *منتقل ساختن و رسم کردن*، عملیات به شمار می‌روند. مدل تحلیلی واسط کاربر برای هر کدام از این عملیات‌ها یک پیاده‌سازی لفظی (literal) فراهم نمی‌آورد. ولی با افزوده شدن بر جزئیات طراحی، جزئیات هر کدام از عملیات‌ها تعیین می‌شود.

تحلیل جریان کاری. هنگامی که تعدادی کاربر متفاوت، هر یک در نقشی متفاوت، از یک واسط کاربر استفاده می‌کند، گاهی ضرورت پیدا می‌کند که از تحلیل وظایف و پرداختن به جزئیات اشیا فراتر رفته، تحلیل جریان کاری را نیز اعمال کنیم. به کمک این تکنیک، می‌توانید چگونگی به انجام رسیدن یک فرایند کاری را در صورت وجود چند نفر (و چند نقش) درک کنید. شرکتی را در نظر بگیرید که می‌خواهد پیچیدن و تحویل نسخه‌های دارو را به‌طور کامل خودکار کند. کل فرایند[1] حول یک برنامه‌ی تحت وب دور می‌زند که پزشکان، داروسازان و بیماران به آن دستیابی دارند. جریان کاری را می‌توان به‌طور موثر با یک نمودار بخش‌بندی UML (که شکل دیگری از نمودار فعالیت‌هاست) نمایش داد.

ما فقط بخش کوچکی از فرایند کاری را در نظر می‌گیریم: وضعیتی که در آن بیمار درخواست پیچیدن نسخه می‌کند. شکل ۲-۱۱ نمودار بخش‌بندی را نشان می‌دهد که وظایف و تصمیم گیری‌های مربوط به هر کدام از سه نقش ذکر شده‌ی قبلی را مشخص می‌سازد. این اطلاعات ممکن است از طریق مصاحبه یا از طریق use case های نوشته شده توسط هر کدام از کنش‌گران استخراج شده باشند. در هر حال، جریان رویدادها (که در شکل نشان داده شده است) شما را قادر می‌سازد تا چند خصوصیت مهم واسط را شناسایی کنید:

۱. هر کاربر، وظایف متفاوت را از طریق واسط پیاده‌سازی می‌کند؛ بنابراین، شکل ظاهری واسط طراحی‌شده برای بیمار با شکل ظاهری واسط طراحی شده برای داروساز یا پزشک متفاوت خواهد بود.
۲. طراحی واسط برای داروسازان و پزشکان باید دستیابی به اطلاعات از منابع ثانویه (مثلاً دستیابی به فهرست موجودی برای داروساز و دستیابی به اطلاعات مربوط به خواص دارویی برای پزشک) و به نمایش درآوردن این اطلاعات امکان‌پذیر باشد.
۳. بسیاری از فعالیت‌های نشان داده شده در نمودار بخش‌بندی را باز هم می‌توان با استفاده از تشریح اشیا و/یا تحلیل وظایف، ریزتر کرد و جزئیات بیشتری به آنها افزود (مثلاً Fill prescription می‌تواند به معنای تحویل سفارش پستی، بازدید از داروخانه یا بازدید از یک مرکز توزیع دارو باشد).

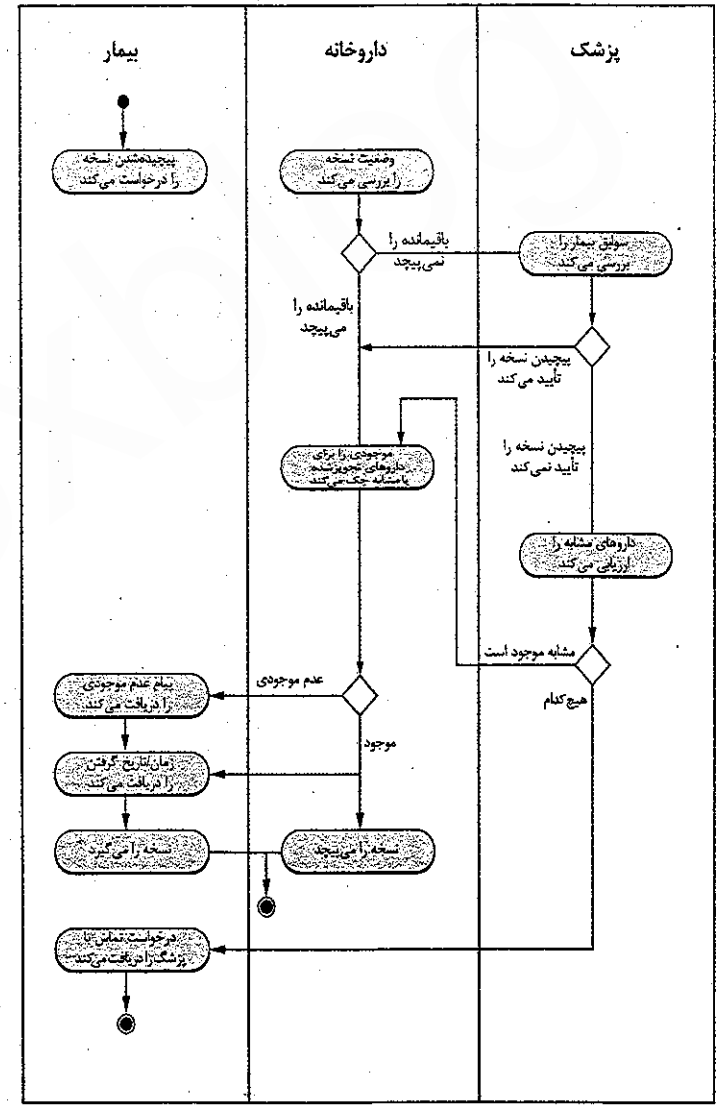
اندرز

گرچه پرداختن به جزئیات اشیا مفید است، نباید از آن به‌عنوان رویکردی قانع‌به‌ذات استفاده کرد. هنگام تحلیل وظایف، *صدای کاربر* نیز باید در نظر گرفته شود.

«به‌مراتب بهتر است فن‌آوری را بر کاربر تطبیق دهیم تا اینکه کاربر را بر فن‌آوری تطبیق دهیم.»

لری مارین

[1] این مثال از [Hac98] برگرفته شده است.



شکل ۲-۱۱ نمودار بخش‌بندی برای وظیفه‌ی پیچیدن نسخه.

نمایش سلسله مراتبی. با شروع تحلیل واسط، فرایند پرداختن به جزئیات رخ می‌دهد. هنگامی که جریان کاری تعیین شد، برای هر نوع کاربر یک سلسله مراتب از وظایف قابل تعریف است. این سلسله مراتب با تشریح مرحله‌ای هر کدام از وظایف شناسایی شده برای کاربر به‌دست می‌آید. برای مثال، سلسله مراتب وظایف و زیروظایف زیر را برای کاربر در نظر بگیرید.

وظیفه کاربر: درخواست برای پیچیدن یک نسخه

- فراهم ساختن اطلاعات هویتی
- مشخص‌کردن نام
- مشخص‌کردن نام کاربری
- مشخص‌کردن PIN و کلمه‌ی عبور
- مشخص‌کردن شماره نسخه
- مشخص‌کردن تاریخی که نسخه باید پیچیده شود.

برای به انجام رساندن این وظیفه، سه وظیفه فرعی تعریف می‌شود. یکی از این وظایف فرعی، یعنی *فراهم ساختن اطلاعات هویتی*، خود شامل سه وظیفه فرعی دیگر می‌شود.

۱۱-۳-۳ تحلیل محتوای صفحه نمایش

وظایف کاربر که در ۲-۳-۱۱ تعریف شدند، به ارائه‌ی انواع متفاوتی از محتوا منجر می‌شوند. برای برنامه‌های کاربردی مدرن، محتوای صفحه نمایش ممکن است از گزارش‌های نوشته شده با کاراکترها (مثلاً صفحات گسترده)، تصاویر گرافیکی (مثلاً بافت نگار، مدل سه بعدی یا تصویری از یک شخص) یا اطلاعات تخصص یافته (مثل فایل‌های صوتی یا تصویری) در تغییر باشد. تکنیک‌های مدل‌سازی تحلیل، که در فصل‌های ۶ و ۷ بحث شدند، اشیای داده‌ای خروجی را مشخص می‌کنند که توسط برنامه کاربردی تولید می‌شوند. این اشیای داده‌ای ممکن است (۱) توسط مولفه‌هایی (نامرتبط با واسط کاربر) در بخش دیگری از برنامه کاربردی تولید شده باشند، (۲) از داده‌های نگهداری شده در بانک اطلاعاتی‌ای به‌دست آیند که از برنامه کاربردی مورد نظر متشر شده باشند.

طی این مرحله از تحلیل واسط، فرمت‌بندی و زیبایی‌شناسی محتوا (آن گونه که توسط واسط به نمایش در می‌آید) مد نظر قرار خواهد گرفت. از جمله پرسش‌هایی که پرسیده و پاسخ داده می‌شوند، عبارتند از

- آیا انواع متفاوت داده‌ها به مکان‌های جغرافیایی مناسب در صفحه نمایش نسبت داده می‌شوند (مثلاً عکس‌ها همواره در گوشه‌ی بالا سمت راست ظاهر می‌شوند)؟
- آیا کاربر می‌تواند مکان صفحه نمایش را برای محتوا به سلیقه خود تغییر دهد؟
- آیا همه‌ی محتوا به‌طور مناسب روی صفحه نمایش شناسایی شده‌اند؟
- اگر قرار است گزارش بزرگی ارائه شود، چگونه باید تقسیم‌بندی شود تا بهتر قابل فهم باشد؟
- آیا در خصوص مجموعه‌های بزرگ داده‌ها سازوکارهایی برای حرکت مستقیم به اطلاعات خلاصه‌بندی شده وجود دارد؟
- آیا خروجی گرافیکی طوری مقیاس‌بندی خواهد شد که در مرزهای دستگاه نمایش بگنجد؟
- از رنگ چگونه در بالا بردن درک مطالب استفاده می‌شود؟
- پیام‌های خطا و هشدار چگونه به کاربر عرضه می‌شود؟

قالب و زیبایی‌شناسی محتوای نمایش داده‌شده به‌عنوان بخشی از UI را چگونه تعیین می‌کنیم؟

پاسخ این پرسش‌ها (و پرسش‌های دیگر) شما را در تعیین خواسته‌ها یاری خواهد داد.

۱۱-۳-۴ تحلیل محیط کار

هاکوس و ردیش [Hac98] اهمیت تحلیل محیط کار را چنین عنوان می‌کنند:

آدم‌ها در انزوا کار نمی‌کنند. آنها از فعالیت‌های اطراف خود، خصوصیات فیزیکی محل کار، نوع تجهیزاتی که به کار می‌برند و روابط کاری خود با دیگران تاثیر می‌پذیرند. اگر محصولاتی که طراحی می‌کنید، برازنده‌ی محیط نباشند، استفاده از آنها ممکن است دشوار یا حتی ناراحت کننده باشد.

در برخی برنامه‌های کاربردی، واسط کاربر برای سیستم‌های کامپیوتری در «مکانی کاربرپسند» (مثلاً نورپردازی مناسب، ارتفاع مناسب برای صفحه نمایش، دستیابی آسان به صفحه کلید) قرار داده می‌شود، ولی در برخی دیگر (مثلاً کف کارخانه یا کابین هواپیما)، نورپردازی ممکن است در حد بهینه نباشد، سروصدا وجود داشته باشد، امکان استفاده از صفحه کلید یا ماوس نباشد، طرز قرار گرفتن صفحه نمایش ایده آل نباشد. طراح واسط ممکن است تحت تاثیر عواملی باشد که سهولت استفاده را محدود می‌سازند.

علاوه بر عوامل محیطی فیزیکی، فرهنگ محیط کار نیز نقشی تعیین کننده دارد. آیا تعامل با سیستم به نوعی سنجیده می‌شود (مثلاً زمان لازم برای هر تراکنش یا صحت هر تراکنش)؟ آیا پیش از فراهم ساختن یک ورودی خاص، دو یا چند نفر باید اطلاعاتی را مشترک داشته باشند؟ این پرسش‌ها و پرسش‌های بسیار دیگر را باید پیش از شروع طراحی واسط پاسخ گفت.

۴-۱۱ مراحل طراحی واسط

هنگامی که تحلیل به پایان رسید، همه‌ی وظایف (یا اشیا و عملیات) مورد نیاز کاربر نهایی، به تفصیل مورد شناسایی قرار گرفته‌اند و فعالیت طراحی واسط شروع می‌شود. طراحی واسط، همانند همه‌ی بخش های طراحی نرم افزار، فرایندی مبتنی بر تکرار است. هر مرحله از طراحی واسط چند دور تکرار می‌شود و در هر دور تکرار، اطلاعات توسعه یافته در مرحله قبل پالایش و بر جزئیات آن افزوده می‌شود.

مدل های فراوانی برای طراحی واسط کاربر پیشنهاد شده است ولی در همه‌ی آنها ترکیبی از مراحل ذیل پیشنهاد می‌شود.

۱. استفاده از اطلاعات به دست آمده طی تحلیل واسط (بخش ۳-۱۱)، تعریف اشیا و کنش‌ها (عملیات‌های واسط).
۲. تعریف راهکارهای کنترلی، یعنی اشیا و عملیات در دسترس کاربر برای تغییردادن حالت سیستم.
۳. به تصویرکشیدن چگونگی تأثیرپذیرفتن حالت سیستم از راهکارهای کنترلی.
۴. نشان دادن چگونگی تفسیر حالت سیستم توسط کاربر با استفاده از اطلاعات به‌دست‌آمده ازطریق واسط.

در برخی موارد، می‌توانید با اتودهایی از هر حالت واسط شروع کنید (یعنی اینکه تحت شرایط گوناگون، واسط چه ظاهری خواهد داشت) و سپس رو به عقب کار کنید تا اشیا، کنش‌ها و سایر اطلاعات طراحی مهم را تعریف نمایید. ترتیب وظایف طراحی، هر چه که باشد، باید (۱) همواره قواعد طلایی بحث شده در بخش ۱-۱۱ را رعایت کنید، (۲) چگونگی پیاده‌سازی واسط را مدل‌سازی کنید و (۳) محیطی را که واسط در آن به کار گرفته می‌شود (مثلاً فن‌آوری صفحه نمایش، سیستم عامل، ابزارهای توسعه) مد نظر داشته باشید.

> «طراحی تعاملی، آمیزه‌ای همگن از هنرهای گرافیکی، فن‌آوری و روان‌شناسی است.»
> **براد وینزر**

۱-۴-۱۱ به کارگیری مراحل طراحی واسط

تعریف اشیا و عملیات واسط یک مرحله‌ی مهم در طراحی واسط، تعریف اشیای واسط و عملیاتی است که در آنها به کار گرفته می‌شوند. برای نیل به این مقصود، سناریوی کاربر تا حدّ زیادی شبیه شرح پردازشی در فصل ۱۲، مورد تجزیه قرار می‌گیرد. یعنی شرحی از سناریوی کاربر نوشته می‌شود. اسمها (اشیا) و فعلها (عملیات) جداسازی می‌شوند تا فهرستی از اشیای عملیات تهیه شود.

هنگامی که اشیا و عملیات تعیین شدند و در چند دور تکرار مورد شرح و تفسیر قرار گرفتند، از نظر نوع، گروه‌بندی می‌شوند. هدف، منبع، و اشیای کاربردی (Application Object) مورد شناسایی قرار می‌گیرد. یک شیء منبع (مثلاً یک آیکن گزارش) کشیده شده (drag) در یک شیء مقصد (مثلاً یک آیکن چاپ) رها می‌شود (drop). پیاده‌سازی این عمل، ایجاد یک گزارش روی کاغذ است. هر شیء کاربردی نشان‌گر داده‌های مشخص کاربردی است که به‌طور مستقیم به‌عنوان بخشی از تعامل با صفحه‌نمایش، دستکاری نمی‌شود. برای مثال برای نگهداری نام و آدرس، از یک فهرست آدرس‌های پستی استفاده می‌شود. خود فهرست ممکن است نگهداری شود، ادغام شود، یا سازمان‌دهی شود (عملیاتی که به کمک منو انجام می‌شوند) ولی از طریق تعامل‌های کاربر، کشیده و رها نمی‌شوند.

هنگامی که به این نتیجه رسیدید که کلیه اشیا و عملیات مهم تعیین شده‌اند (برای یک دور تکرار طراحی)، چیدمان صفحه‌نمایش اجرا می‌شود. برخلاف فعالیت‌های دیگر طراحی واسط، چیدمان واسط یک فرایند تعاملی است که در آن، طراحی گرافیکی و طرز قرار گرفتن آیکون‌ها، تعیین متون توصیفی صفحه‌نمایش، مشخص کردن پنجره‌ها و تعریف عناصر اصلی و فرعی منوها انجام می‌شود. اگر یک استعاره واقعی مناسب برای کاربر موردنظر وجود داشته باشد، در این هنگام مشخص می‌شود و چیدمان به شیوه‌ای سازمان‌دهی می‌شود که این استعاره را در خود جای دهد.

برای روشن کردن مراحل طراحی فوق‌الذکر، یک سناریوی کاربر برای نسخه پیشرفته‌ای از سیستم *SafeHome* در نظر می‌گیریم. در این نسخه پیچیده، *SafeHome* از طریق مودم یا اینترنت قابل دستیابی است. برنامه PC به صاحبخانه اجازه می‌دهد تا وضعیت منزل را از راه دور کنترل کند، پیکربندی *SafeHome* را به حالت اول برگرداند، سیستم را مسلح (arm) یا غیرمسلح (disarm) کند و اتاق‌های منزل را به صورت بصری مورد نظارت قرار دهد.

use case مقدماتی: می‌خواهم به سیستم *SafeHome* نصب‌شده در منزل خود دستیابی داشته باشم. با استفاده از نرم‌افزاری که روی یک PC راه دور قرار دارد (مثلاً یک کامپیوتر کیفی که در سفر یا محل کار خود به همراه دارد) وضعیت سیستم آژیر را تعیین می‌کنم، آن را مسلح یا غیرمسلح می‌کنم، نواحی امنیتی را دوباره پیکربندی می‌کنم و اتاق‌های متفاوت منزل را ازطریق دوربین‌های ویدیویی نصب‌شده مشاهده می‌کنم.

برای دستیابی از محل دور، یک شماره شناسایی و یک کلمه‌ی عبور ارائه می‌دهم. این دو مقدار، سطح دستیابی را تعیین می‌کنند (مثلاً همه‌ی کاربران ممکن است مجاز به پیکربندی دوباره سیستم نباشند) و ایمنی را فراهم می‌آورند. پس از آنکه اعتبار کاربر تأیید شد می‌توانم وضعیت سیستم را با مسلح کردن یا غیرمسلح کردن آن تغییر دهم. کاربر، سیستم را با نمایش یک نقشه پلان از منزل، مشاهده‌ی هر یک از حس‌گرهای امنیتی، به نمایش درآوردن هر یک از مناطق پیکربندی شده فعلی و اصلاح نواحی موردنظر، دوباره پیکربندی می‌کند. کاربر، داخل منزل را از طریق دوربین‌هایی که در محل‌های مهم نصب شده‌اند مشاهده می‌کند. می‌توانم هر یک از دوربین‌ها را زوم کنم و نماهای متفاوتی به‌دست آورم.

براساس use case، وظایف صاحبخانه، اشیا، و آیتم های داده‌ای زیر قابل شناسایی است:

- *دستیابی به سیستم SafeHome*
- *واردکردن یک* **شماره شناسایی و کلمه‌ی عبور** *برای اجازه دستیابی از راه دور*
- **چک کردن وضعیت سیستم**
- *مسلح کردن یا غیرمسلح کردن سیستم*
- *نمایش* **نقشه پلان و مکان حس‌گرها**
- *نمایش* **نواحی** *روی نقشه پلان*
- *تغییر* **نواحی** *روی نقشه پلان*
- *نمایش* **مکان دوربین‌ها** *روی نقشه پلان*
- *انتخاب* **دوربین‌ها** *برای مشاهده*
- *مشاهده* **تصاویر ویدیویی** *(چهار فریم بر ثانیه)*
- *زوم کردن* **دوربین‌ها**

اشیا (حروف ضخیم) و عملیات (حروف ایتالیک) از فهرست وظایف صاحبخانه که در بالا ذکر شد، استخراج می‌شود. اکثر اشیای ذکر شده، اشیای کاربردی‌اند. ولی **video camera lacation** (یک شیء منبع) کشیده می‌شود و روی **video camera** (شیء مقصد) رها می‌شود تا یک **video image** (پنجره‌ای با نمایش ویدیویی) ایجاد گردد.

یک طرح مقدماتی از چیدمان صفحه‌نمایش برای نظارت ویدیویی ایجاد می‌شود (شکل ۳-۱۱)[1]. برای فراخوانی تصویر ویدیویی، یک آیکن مکان دوربین (یک شیء منبع) که در نقشه پلان قرار دارد، در پنجره‌ی پایش انتخاب می‌شود. در این مورد، مکان دوربین در اتاق نشیمن، LR، کشیده می شود روی آیکن دوربین در گوشه بالا سمت چپ صفحه‌نمایش رها می‌شود. پنجره تصویر ویدیویی ظاهر شده، تصاویر زنده را از دوربین موجود در اتاق نشیمن (LR) به نمایش در می‌آورد. از لغزنده‌های Zoom و Pan برای کنترل بزرگ‌نمایی و جهت‌گیری تصویر ویدیویی استفاده می‌شود. برای انتخاب نمای حاصل از یک دوربین دیگر، کافی است کاربر یک آیکن مکان دوربین دیگر را کشیده در آیکن دوربین واقع در گوشه بالا سمت چپ صفحه‌نمایش رها کند.

چیدمان نشان داده شده باید با بسط دادن هر یک از عناصر منوی میله‌ای و نشان دادن عملیاتِ در دسترس برای هر حالت نظارت ویدیویی تکمیل شود. در اثنای طراحی واسط کاربر، مجموعه کاملی از طرح‌ها برای هر یک از وظایف صاحبخانه که در سناریوی کاربر ذکر شد، ایجاد خواهد گردید.

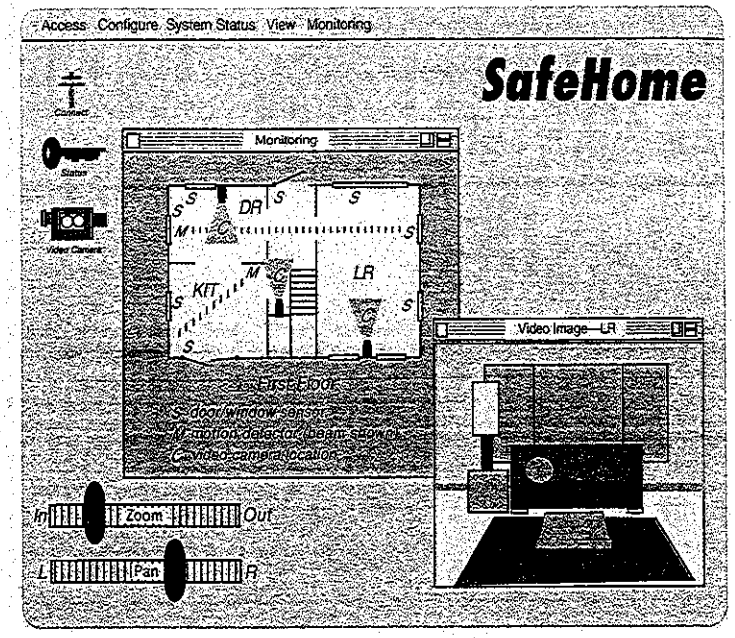
اندرز

گرچه ابزارهای خودکار می‌توانند در تهیه‌ی نمونه‌های اولیه‌ی چیدمان مفید باشند، گاهی مفید و کاغذ تنها چیزی است که لازم می‌شود.

۲-۴-۱۱ الگوهای طراحی واسط کاربر

واسط‌های گرافیکی کاربر چنان متداول شده‌اند که اکنون آرایه گسترده‌ای از الگوهای طراحی برای واسط کاربر پدید آمده‌اند. چنان که پیش‌تر نیز در این کتاب گفته شد، الگوی طراحی، انتزاعی است که برای یک مسأله‌ی طراحی ویژه و با مرزهای معین، راهکاری تجویز می‌کند.

[1] توجه دارید که این تا حدی با پیاده‌سازی این ویژگی‌ها در فصل‌های قبل تفاوت دارد. این را می‌توان به‌عنوان نخستین پیش‌نویس طراحی در نظر گرفت که یکی از راه‌های پیش رو را می‌تواند نشان دهد.



شکل ۳-۱۱ چیدمان صفحه‌نمایش مقدماتی.

مرجع وب

گستره وسیعی از الگوهای طراحی UI پیشنهاد شده است. برای معرفی آنها سایت www.hcipattern.org را ببینید.

به‌عنوان مثالی از مسأله‌ی طراحی واسط که به وفور مشاهده می‌شود، وضعیتی را در نظر بگیرید که کاربر در آن باید یک یا چند تاریخ تقویمی را، گاهی از ماه‌های قبل، وارد کند. راهکارهای بسیاری برای این مسأله ساده وجود دارد و چند الگوی متفاوت قابل پیشنهاد است. لاکسو [Laa00] الگویی به‌نام Calendar Strip (نوار تقویمی) را پیشنهاد می‌کند که تقویمی پیوسته با قابلیت جابه‌جایی تولید می‌کند که در آن، تاریخ فعلی برجسته است و تاریخ‌های آینده را می‌توان با انتخاب آنها از تقویم برگزید. استعاره‌ی تقویم نزد هر کاربری شناخته شده است و سازوکاری موثر برای قرار دادن تاریخ آینده در حیطه فراهم می‌آورد.

طی دهه‌ی گذشته آرایه گسترده‌ای از الگوهای طراحی پیشنهاد شده است. بحث مفصل‌تر الگوهای طراحی واسط در فصل ۱۲ ارائه شده است. به‌علاوه، اریکسون [Eri08] به مجموعه‌های تحت وب بسیار اشاره می‌کند.

۳-۴-۱۱ مسائل طراحی

به موازاتی که واسط کاربر تکامل پیدا می‌کند، همواره چهار مسأله طراحی متداول مطرح می‌شود: زمان پاسخ‌دهی سیستم، امکانات راهنمای کاربر، اداره‌کردن اطلاعات خطا، نشانه‌گذاری فرمان‌ها. متأسفانه بسیاری از طراحان تقریباً تا اواخر فرایند طراحی به این مسائل نمی‌پردازند (گاه این کار تا زمان آماده شدن اولین نسخه کاری به تعویق می‌افتد). نتیجه، غالباً تکرارهای بیهوده، تأخیر در پروژه و ناراحتی مشتری است. در نظر گرفتن هر یک از مسائل فوق در ابتدای طراحی نرم‌افزار، یعنی آنگاه که اِعمال تغییرات به‌راحتی و با هزینه کم امکان‌پذیر است، به مراتب بهتر است.

زمان پاسخ. زمان پاسخ سیستم مشکل اصلی بسیاری از برنامه‌های کاربردی تعاملی است. به طور کلی، زمان پاسخ سیستم، از نقطه‌ای اندازه‌گیری می‌شود که در آن کاربر یک عمل کنترلی را اجرا می‌کند (مثلاً کلید Enter را می‌زند یا ماوس را کلیک می‌کند) تا نرم‌افزار با خروجی یا عمل مطلوب، پاسخ بدهد.

پاسخ سیستم دو ویژگی مهم دارد: طول و تغییرپذیری. اگر *طول* پاسخ سیستم بیش از حد زیاد باشد، ناراحتی و استرس کاربر، نتیجه‌ای اجتناب‌ناپذیر خواهد بود. *تغییرپذیری* عبارت است از انحراف از زمان پاسخ میانگین، که مهمترین ویژگی زمان پاسخ‌دهی است. حتی اگر زمان پاسخ‌دهی نسبتاً طولانی باشد، تغییرپذیری اندک، کاربر را قادر به ایجاد ریتم در تکرار می‌کند. برای مثال، پاسخ یک ثانیه‌ای به یک فرمان، بر پاسخی که بین ۰/۱ ثانیه تا ۲/۵ ثانیه تغییر می‌کند، ترجیح داده می‌شود. کاربر همواره متحیر می‌ماند که آیا پشت پرده اتفاق «متفاوتی» می‌افتد.

تسهیلات کمکی (راهنما). تقریباً همه‌ی کاربران یک سیستم تعاملی و مبتنی بر کامپیوتر هر از گاهی به راهنمایی و کمک نیاز دارد. در برخی موارد، یک پرسش ساده که از همکاری پرسیده می‌شود، راهگشا خواهد بود. در سایر موارد، جستجوی مفصل در مجموعه‌ای چند جلدی از «جزوات راهنمای کاربران» ممکن است تنها گزینه پیش رو باشد. به هر حال، در اکثر موارد، نرم‌افزارهای مدرن امکانات راهنمای آنلاینی فراهم می‌سازند که به کاربر این امکان را می‌دهند تا بدون ترک واسط، پاسخ پرسش خود را بگیرد یا مشکلی را حل کند.

چند مسأله طراحی [Rub88] را هنگام پرداختن به تسهیلات راهنما باید در نظر داشت:

- آیا راهنما برای کلیه عملکردهای سیستم و در همه‌ی اوقات تعامل با سیستم در دسترس خواهد بود؟
- حالت‌های ممکن عبارتنداز: راهنما فقط برای زیرمجموعه‌ای از کلیه عملیات‌ها و عملکردها؛ راهنما برای کلیه عملکردها.
- کاربر چگونه درخواست کمک و راهنمایی می‌کند؟ حالت‌های ممکن عبارتنداز: یک منوی راهنما؛ یک کلید خاص؛ یک فرمان HELP.
- راهنما چگونه ارائه خواهد شد؟ حالت‌های ممکن عبارتنداز: یک پنجره جداگانه؛ ارجاع به یک سند چاپ شده (که چندان ایده‌آل نیست)؛ یک یا دو خط پیشنهادی که در مکان ثابتی از صفحه‌نمایش ظاهر می‌شود.
- کاربر چگونه به تعامل‌های عادی باز می‌گردد؟ حالت‌های ممکن عبارتنداز: دکمه برگشتی که بر روی صفحه قرار دارد، یک کلید تابعی یا دنباله کنترلی.
- اطلاعات راهنما چگونه ساختاری دارند؟ حالت‌های ممکن عبارتنداز: یک ساختار «همواره» که در آن همه‌ی اطلاعات از طریق یک واژه کلیدی قابل دستیابی است؛ یک ساختار سلسله مراتبی که با جلو رفتن کاربر در راستای این ساختار، جزئیات بیشتری را در اختیار قرار می‌دهد؛ یا استفاده از آبَرمتن.

مدیریت خطاها. پیام‌های خطا و هشدار «خبرهای ناگواری» هستند که هنگام خراب شدن اوضاع، تحویل کاربران سیستم‌های تعاملی می‌شود. پیام‌های خطا و هشدارها در بدترین حالت خود، اطلاعات بیهوده و گمراه‌کننده‌ای می‌دهند و فقط به افزایش ناراحتی کاربر کمک می‌کنند. کمتر کاربری است که

«یک اشتباه رایج که آدم‌ها هنگام طراحی چیزهای کاملاً ضد خطا مرتکب می‌شوند، دست کم گرفتن نوع افراد کاملاً نادان است.»

داگلاس آدامز

به خطاهایی از نوع زیر برنخورده باشد: «برنامه‌ی فلان اجبارا باید خاتمه پیدا کند، چون خطایی از نوع 1023 رخ داده است.» باید برای خطای 1023 توضیحی وجود داشته باشد؛ در غیر این صورت، چه لزومی داشت که طراحان این پیام را اضافه کنند؟ ولی، این پیام خطا واقعاً نشان نمی‌دهد که مشکل چیست یا برای به‌دست آوردن اطلاعات بیشتر، کجا را باید نگاه کرد. پیام خطایی که به شیوه بالا ارائه می‌شود به تسکین کاربر یا حل مشکل کمک نمی‌کند.

به‌طور کلی، هر پیام خطا یا هشدار تولید شده توسط یک سیستم تعاملی باید دارای ویژگی‌های زیر باشد:

- پیام باید مشکل را به زبانی شرح دهد که کاربر قادر به درک آن باشد.
- پیام باید حاوی یک توصیه سازنده برای رهایی از وضعیت خطا باشد.
- پیام باید هرگونه تبعات منفی خطا (مثلاً فایل‌های داده‌ای مخدوش شده) را خاطرنشان کند تا کاربر بتواند آنها را چک کند.
- پیام باید با یک نشانه سمعی یا بصری همراه باشد. یعنی یک صدای بوق یا نمایش پیام همراه شود، یا حالت چشمک زدن داشته باشد یا به رنگی ظاهر شود که به آسانی به‌عنوان «رنگ خطا» قابل تشخیص باشد.
- پیام نباید «قضاوت گونه» باشد. یعنی، لحن آن نباید طوری باشد که کاربر را مورد شماتت قرار دهد.

از آنجا که هیچ کس واقعاً از خبرهای ناگوار خوشش نمی‌آید، معدود کاربرانی هستند که پیام‌های خطا را دوست داشته باشند، هر چند که این پیام‌ها خیلی خوب طراحی شده باشند. ولی یک فلسفه مؤثر برای پیام‌های خطا می‌تواند تأثیر بسزایی در کیفیت یک سیستم تعاملی داشته باشد و هنگام رخ دادن مشکل، تا حدّ زیادی از ناراحتی کاربر بکاهد.

نشانه‌گذاری منوها و فرمان‌ها. زمانی متداول‌ترین شیوه تعامل میان کاربر و سیستم نرم‌افزاری، تایپ فرمان‌ها بود و از آنها برای انواع برنامه‌های کاربردی استفاده می‌شد. امروزه استفاده از واسط‌های پنجره‌ای، تکیه بر فرمان‌های تایپی را کاهش داده است، ولی بسیاری از کاربران قدرتمند همچنان شیوه تایپ فرمان‌ها را ترجیح می‌دهند. هنگام طراحی محیط تعامل از طریق تایپ فرمان‌ها، به مسائل زیر باید توجه داشت:

- آیا هر یک از گزینه‌های منو دارای یک فرمان متناظر هست؟
- فرمان‌ها چه شکلی به خود می‌گیرند؟ گزینه‌ها عبارتنداز: دنباله کنترلی (مثلاً Alt + P)؛ کلیدهای خاص؛ یا یک واژه تایپ شده.
- فراگیری و به خاطر سپردن فرمان‌ها چقدر دشوار خواهد بود؟ اگر فرمانی فراموش شد، چه می‌توان کرد؟
- آیا کاربر می‌تواند فرمان‌ها را به سلیقه خویش مختصر و کوتاه کند؟
- آیا نشانه‌های منوها در حیطه‌ی واسط، آن‌قدر واضح هستند که نیازی به توضیح نداشته باشند؟
- آیا زیرمنوها با قابلیتی که منوی اصلی بیان می‌کند، سازگاری دارند؟

«واسط از جهنم - برای تصحیح این خطا و ادامه کار هر عدد اول یازده رقمی را که می‌شناسید، وارد کنید ...»

ناشناس

پیام خطای «خوب» چه خصوصیاتی باید داشته باشد؟

ابزارهای نرم‌افزاری

توسعه واسط کاربر

هدف: این ابزارها به مهندس نرم‌افزار کمک می‌کنند تا یک GUI پیچیده را با توسعه نرم‌افزار نسبتاً اندک سفارشی ایجاد کند. این ابزارها، دستیابی به مولفه‌های قابل استفاده‌ی مجدد را فراهم ساخته ایجاد واسط را به انتخاب از میان قابلیت‌های از پیش تعیین شده‌ای تبدیل می‌کنند که با استفاده از آنها می‌توان این قابلیت‌ها را به هم مونتاژ کرد.

مکانیک: واسط‌های مدرن کاربر با به کارگیری مجموعه‌ای از مولفه‌های قابل استفاده‌ی مجدد ساخته می‌شوند که با چند مولفه‌ی سفارشی جفت می‌شوند و ویژگی تخصصی مورد نظر را فراهم می‌سازند. اکثر ابزارهای توسعه‌ی واسط به مهندس نرم‌افزار این امکان را می‌دهند تا با به کارگیری قابلیت «کشیدن و رها کردن» به ایجاد واسط بپردازد. یعنی، سازنده‌ی واسط از میان قابلیت‌های از پیش تعیین شده‌ی بسیار (فرم سازها، سازوکارهای تعامل، قابلیت پردازش فرمان) موارد دلخواه را بر می‌گزیند و این قابلیت‌ها را در داخل محتوا واسطی قرار می‌دهد که باید ساخته شود.

ابزارهای نمونه

Lega Suite GUI، که توسط *Seagull Software* (**www.seagullsoftware.com**) توسعه یافته است، ایجاد GUIهای مبتنی بر مرورگرها را میسر می‌سازد و تسهیلاتی برای مهندسی دوباره واسط‌های قدیمی فراهم می‌آورد. *Motif common Desktop Environment* که توسط Open Group (**www.osf.org/tech/desktop/cdel**) توسعه یافته است، یک واسط کاربر گرافیکی منسجم برای برنامه‌های کامپیوتری با سیستم‌های باز است. این ابزار، یک واسط گرافیکی استاندارد و یگانه برای مدیریت داده‌ها و فایل‌ها (رومیزی گرافیکی) و برنامه‌های کاربردی تحویل می‌دهد.

Altia Design 8.0 که توسط Altia (**www.altia.com**) توسعه یافته است، ابزاری برای ایجاد GUIها در انواع سکوهای متفاوت (مثلاً در خودروها، دستگاه‌های دستی و صنعتی) است.

چنان‌که پیش‌تر در همین فصل گفته شد، قرارداد مربوط به استفاده از فرمان‌ها باید در میان همه‌ی برنامه‌های کاربردی رعایت شود. اگر در برنامه‌ای فشاردادن ترکیب Alt+D باعث تکثیر یک شیء گرافیکی شود و در برنامه‌ای دیگر، همین ترکیب کلیدها به حذف شیء بینجامد، باعث سردرگمی کاربر شده احتمال خطا بالا می‌رود.

دسترس‌پذیری در برنامه کاربردی. با همگانی شدن برنامه‌های کاربردی کامپیوتری، مهندسان نرم‌افزار باید اطمینان حاصل کنند که طراحی واسط شامل سازوکارهایی است که برای افرادی با نیازهای خاص، دستیابی آسان را میسر می‌سازد. دسترس‌پذیری برای کاربران (و مهندسان نرم‌افزار) که ممکن است از نظر بدنی دچار مشکلاتی باشند، به دلایل اخلاقی، قانونی و تجاری، الزامی است. انواع دستورالعمل‌های مربوط به دسترس‌پذیری (مانند [W3C03])-که بسیاری از آنها برای برنامه‌های تحت وب طراحی شده‌اند، ولی غالباً برای همه‌ی انواع نرم‌افزار قابل اعمال هستند- برای طراحی واسط‌هایی که سطوح متغیری از دستیابی را امکان پذیر می‌سازند، پیشنهادهای مشروحی دارند. سایرین

مرجع وب

دستورالعمل‌هایی برای توسعه نرم‌افزارهای قابل دسترسی را می‌توان در نشانی زیر یافت:
www3.ibm.com/able/guidelines/software/accessoftware.html

(مثل [App08]، [Mic08]) دستورالعمل‌هایی برای «فن‌آوری کمک رسان» فراهم می‌سازند که به نیازهای افرادی با نارسایی‌های بینایی، شنوایی، حرکتی، گفتاری و یادگیری می‌پردازد.

جهانی‌سازی. مهندسان نرم‌افزار و مدیران آنها، مهارت‌ها و تلاش لازم برای ایجاد واسط‌هایی را که پاسخ‌گوی نیازهای زبان‌ها و مناطق متفاوت باشد، کوچک می‌شمارند. به وفور پیش می‌آید که واسط‌ها برای یک زبان و منطقه طراحی می‌شوند و سپس در کشورهای دیگر از آنها استفاده می‌شود. چالش برای طراحان واسط، ایجاد نرم‌افزارهای «جهانی» است. یعنی، واسط‌های کاربر باید طوری طراحی شوند که یک هسته‌ی کلی عملیاتی را در خود جای دهند که به همه‌ی کسانی که از نرم‌افزار استفاده می‌کنند قابل تحویل باشد. ویژگی‌های محلی سازی، کاربر را قادر می‌سازد تا واسط را مطابق با نیازهای یک بازار خاص، سفارشی کند.

انواع دستورالعمل‌های جهانی‌سازی (مانند [IBM03] در دسترس مهندسان نرم‌افزار قرار دارد. این دستورالعمل‌ها به مسائل طراحی گسترده (مانند اینکه چیدمان‌های صفحه نمایش ممکن است در بازارهای گوناگون با هم تفاوت داشته باشند) و مسائل پیاده‌سازی گسسته (مانند اینکه حروف الفبایی متفاوت ممکن است نشان گذاری تخصصی ایجاد کند) می‌پردازند. استاندارد یونیکد [Uni03] برای پرداختن به چالش مدیریتِ ده‌ها زبان طبیعی با صدها کاراکتر و نماد، توسعه یافته است.

۱۱-۵ طراحی واسط برنامه‌ی تحت وب

هر واسط کاربر، خواه برای یک برنامه‌ی تحت وب طراحی شده باشد خواه برای یک برنامه سنتی یا محصول مصرفی یا دستگاه صنعتی، باید خصوصیات «قابلیت استفاده» را که قبلاً در همین فصل بحث شد، از خود نشان دهد. دیکس [Dix99] استدلال می‌کند که یک واسط برنامه‌ی تحت وب را باید طوری طراحی کنید که به سه پرسش اولیه برای کاربر نهایی پاسخ بدهد:

من کجا هستم؟ واسط باید (۱) به نحوی نشان دهد که دستیابی به برنامه‌ی تحت وب صورت گرفته است[1] و (۲) کاربر را از موقعیت او در سلسله مراتب محتوا آگاه سازد.

اکنون چه می‌توانم بکنم؟ واسط همواره باید کاربر را در فهم گزینه‌های فعلی اش یاری دهد -اینکه چه قابلیت‌هایی در دسترس قرار دارند، کدام پیوندها فعال‌اند و کدام محتوای مناسبت دارند.

کجا بوده‌ام و به کجا خواهم رفت؟ واسط باید گشت‌وگذار را تسهیل کند. از این رو، باید «نقشه‌ای» فراهم آورد که نشان دهد کاربر کجا بوده است و چه مسیرهایی می‌تواند پیش بگیرد تا به‌جای دیگری از برنامه‌ی تحت وب برسد (این نقشه باید طوری پیاده‌سازی شود که قابل درک باشد).

یک واسط اثربخش برای برنامه‌ی تحت وب باید همچنان که کاربر نهایی در میان محتوا و قابلیت‌های عملیاتی گشت‌وگذار می‌کند، به هر کدام از این پرسش‌ها پاسخ دهد.

اندرز

اگر این احتمال وجود دارد که کاربران در مکان‌ها و سطوح گوناگونی از سلسله مراتب محتوا وارد برنامه‌ی تحت وب شما شوند، حتماً هر صفحه را با ویژگی‌های گشت‌وگذار طوری طراحی کنید که کاربر زاب سایر نقاط مورد نظر هدایت کند.

۱-۵-۱۱ دستورالعمل‌ها و اصول طراحی واسط

واسط کاربر برای یک برنامه‌ی تحت وب، جایی است که اولین تاثیر را بر کاربر می‌گذارد. برنامه‌ی تحت وب هر قدر هم که دارای محتوای باارزشی باشد، قابلیت‌های عملیاتی و سرویس‌های پیچیده

[1] هر کدام از ما صفحه وبی را نشان‌گذاری کرده‌ایم که بعداً دوباره از آن بازدید کنیم، ولی نشانی از حیطه‌ی صفحه نداریم (یا اینکه نمی‌توانیم به مکان دیگری از سایت حرکت کنیم).

داشته باشد و در کل مزایای زیادی در پی داشته باشد، اگر واسط آن از طراحی خوبی برخوردار نباشد، کاربر بالقوه را ناامید می‌کند و در واقع ممکن است باعث شود که کاربر به‌جای دیگری برود. به دلیل حجم بالای برنامه‌های تحت وب رقیب در هر زمینه و موضوع، واسط باید کاربر بالقوه را بلافاصله «به‌چنگ» آورد. بروس تونیوتزی [Tog01] مجموعه‌ای از خصوصیات بنیادی را تعریف می‌کند که همه‌ی واسط‌ها باید از خود نشان دهند و در این راه، فلسفه‌ای را بنیان می‌گذارد که هر طراح واسط برنامه‌ی تحت وب باید آن را دنبال کند:

> واسط‌های اثربخش به چشم می‌آیند و به کاربران خود حس کنترل القا می‌کنند. کاربران به سرعت گستردگی گزینه‌های پیش رو را می‌بینند، چگونگی رسیدن به هدف را درک می‌کنند و کار خود را انجام می‌دهند.
>
> واسط‌های اثربخش توجه کاربر را به کارکرد درونی سیستم جلب نمی‌کنند. کار به دقت انجام می‌شود و پیوسته ضبط می‌شود و کاربر اختیار کامل دارد که هر فعالیتی را در هر زمان بی اثر کند.
>
> سرویس‌ها و برنامه‌های کاربردی اثربخش، حداکثر کار را انجام می‌دهند در حالی که به حداقل اطلاعات از سوی کاربر نیاز دارند.

تونیوتزی [Tog01] به منظور طراحی واسط‌هایی برای برنامه‌ی تحت وب که این خصوصیات را از خود نشان می‌دهند، یک مجموعه اصول طراحی را مطرح می‌کند که باید بر سایر اصول برتری داد.[1]

پیش‌بینی. برنامه‌ی تحت وب *باید طوری طراحی شود که حرکت بعدی کاربر را پیش بینی کند.* برای مثال، یک برنامه‌ی تحت وب برای پشتیبانی مشتری را در نظر بگیرید که توسط سازنده‌ی چاپگرهای کامپیوتر تهیه شده است. کاربری یک شیء محتوایی را درخواست کرده است که اطلاعات مربوط به درایور چاپگر را برای سیستم عاملی که به تازگی وارد بازار شده است، در اختیار می‌گذارد. طراح برنامه‌ی تحت وب باید این را پیش‌بینی کند که کاربر ممکن است درخواست دانلودکردن درایور را داشته باشد و باید امکاناتی برای گشت‌وگذار فراهم سازد که این امر را میسر سازد بدون اینکه کاربر مجبور به جستجو به دنبال این قابلیت باشد.

ارتباطات. واسط *باید وضعیت هر کدام از فعالیت‌های آغاز شده توسط کاربر را انتقال دهد.* این انتقال و ارتباط می‌تواند آشکار باشد (مثلاً در قالب یک پیام متنی) یا با ظرافت انجام شود (مثلاً تصویر یک برگه کاغذ از میان چاپگر عبور می‌کند و نشان می‌دهد که عمل چاپ در حال انجام است). واسط همچنین باید وضعیت کاربر را (مثلاً هویت کاربر) و موقعیت او در سلسله مراتب محتوای برنامه‌ی تحت وب را به اطلاع برساند.

سازگاری. *استفاده از کنترل‌های گشت‌وگذار، منوها، آیکون‌ها و زیبایی‌شناسی (مثلاً، رنگ، شکل، چیدمان) باید در سرتاسر برنامه‌ی تحت وب سازگار باشد.* برای مثال، اگر متن آبی رنگ که زیر آن خط کشیده شده است، به معنای پیوند است، محتوا هرگز نباید حاوی متون آبی رنگ با خط زیرین باشد، در حالی که این متون هیچ پیوندی را مشخص نمی‌کنند. به‌علاوه، یک شیء، مثلاً مثلثی زرد رنگ، که برای نشان دادن پیام احتیاط قبل از فراخوانی تابع یا عملیاتی توسط کاربر استفاده می‌شود،

> «اگر سایتی کاملاً قابل استفاده باشد، ولی فاقد سبک طراحی مناسب و ظریف باشد، با شکست مواجه خواهد شد.»
>
> گرت کلونینگر

نکته‌ی کلیدی

یک واسط خوب برای برنامه‌ی تحت وب، قابل فهم است، خطاهای کاربر را نادیده می‌گیرد و به او حس کنترل می‌دهد.

آیا یک مجموعه اصول پایه وجود دارد که بتوان در طراحی GUI به کار برد؟

[1] اصول اولیه‌ی تونیوتزی برای استفاده در این کتاب، برگرفته و بسط داده شده‌اند. برای بحث بیشتر درباره این اصول، [Tog01] را ببینید.

نباید برای اهداف دیگر در جای دیگری از برنامه‌ی تحت وب استفاده شود. سرانجام اینکه هر ویژگی واسط باید به شیوه‌ای پاسخ دهد که با انتظارات کاربر سازگار باشد.[1]

خودمختاری کنترل شده. واسط *باید حرکت کاربر را در سرتاسر برنامه‌ی تحت وب تسهیل کند. ولی به‌طرزی که قراردادهای گشت‌وگذار وضع شده را تقویت نماید.* برای مثال، گشت‌وگذار به بخش‌های امن برنامه‌ی تحت وب باید با نام کاربری و کلمه‌ی عبور کنترل شود و نباید سازوکاری برای گشت‌وگذار وجود داشته باشد که کاربر را قادر به دور زدن این کنترل‌ها کند.

بازدهی. *طراحی برنامه‌ی تحت وب و واسط آن باید بازدهی کاربر را بهینه کند نه بازدهی سازنده‌ای که آن را طراحی می‌کند و می‌سازد یا بازدهی محیط کلاینت-سروری که آن را اجرا می‌کند.* تونیوتزی [Tog01] در این خصوص چنین می‌نویسد: «این حقیقت ساده، دلیلی است بر اینکه چرا هر کسی که در یک پروژه نرم‌افزار دخالت دارد باید بداند که هدف اول، بهره‌وری کاربر است و از تفاوت میان ساختن یک سیستم اثربخش و قدرت بخشیدن به کاربری اثربخش آگاه باشد.»

انعطاف‌پذیری. واسط *باید به‌قدر کافی انعطاف‌پذیر باشد تا برخی کاربران را قادر به انجام مستقیم وظایف و کاربران دیگر را قادر به کاوش در برنامه‌ی تحت وب به شیوه‌ای تصادفی سازد.* در هر مورد، باید کاربر را قادر سازد تا دریابد کجاست و قابلیت‌هایی را در اختیار کاربر بگذارد که بتواند اشتباهات خود را بی‌اثر کند و مسیرهای گشت‌وگذاری را که ضعیف انتخاب شده‌اند، دوباره دنبال کند.

کانون توجه. واسط *برنامه‌ی تحت وب (و محتویاتی که ارائه می‌دهد) باید وظایفی را کانون توجه قرار دهد که کاربر در دست دارد.* در همه‌ی ابررسانه‌ها، این تمایل وجود دارد که کاربر به محتویاتی هدایت شود که ممکن است ربط چندانی به کار او نداشته باشد. چرا؟ چون انجام این کار بسیار آسان است! مسأله این است که کاربر می‌تواند به سرعت در لایه‌های بسیاری از اطلاعات پشتیبان گم شود و اصلاً فراموش کند که از ابتدا به دنبال کدام محتوا بوده است.

قانون فیت (Fit's Law). «*زمان لازم برای رسیدن به یک هدف، تابعی است از فاصله تا آن هدف و اندازه آن*» [Tog01]. براساس مطالعه‌ای که در دهه ۱۹۵۰ اجرا شد [Fit54]، قانون فیت «روش مؤثری برای مدل‌سازی حرکت‌های سریع و هدف‌مند است که در آن یک دستگاه فرعی (مثلاً یک دست) از سکون در موقعیت شروع حرکت خود را آغاز می‌کند و در ناحیه‌ی هدف دوباره به سکون می‌رسد» [Zha02]. اگر یک سری انتخاب‌ها یا ورودی‌های استاندارد شده (با گزینه‌های متفاوت بسیار در آن سری) توسط یک وظیفه‌ی کاربری تعریف شود، انتخاب اول (مثلاً ماوس) باید از نظر فیزیکی به انتخاب بعدی نزدیک باشد. برای مثال، واسط صفحه اصلی یک برنامه‌ی تحت وب تجارت الکترونیکی را در نظر بگیرید که دستگاه‌های الکترونیکی را به‌فروش می‌رساند.

هر گزینه‌ی کاربر، به‌معنای مجموعه‌ای از انتخاب‌ها یا کنش‌هاست که دستورات کاربر را دنبال می‌کنند. برای مثال، گزینه «خرید یک محصول» ایجاب می‌کند که کاربر یک گروه محصول و سپس نام محصول را وارد کند. گروه محصولات (مثلاً تجهیزات صوتی، تلویزیون پخش DVD) به محض انتخاب «خرید یک محصول» به‌صورت یک منوی بازشونده ظاهر می‌شود. بنابراین، انتخاب بعدی

> «بهترین مقر، آن است که با چند گام انجام شود. فاصله میان کاربر و هدفش را کوتاه کنید.»
>
> ناشناس

مرجع وب

جستجو در وب، کتابخانه‌های بسیاری را آشکار خواهد کرد. مدل واسط‌ها، کلاس‌ها و پکیج‌های Java API در java.sun.com با COM، DCOM، TypeLibraries و msdn.Microsoft.com در

[1] تونیوتزی [Tog01] می‌گوید تنها راه برای حصول اطمینان از این‌که انتظارات کاربر به طور مناسب درک شده‌اند، آزمودن جامع کاربر است (فصل ۲۰).

بلافاصله آشکار می‌شود (دم دست است) و زمان به‌دست آوردن آن قابل چشم‌پوشی است. ولی اگر انتخاب روی منویی ظاهر شود که در طرف دیگر صفحه نمایش قرار دارد، زمان دستیابی کاربر به آنها (و انتخاب آن) بسیار طولانی خواهد بود.

اشیای واسط انسانی. کتابخانه گسترده‌ای از اشیای واسط انسانی که قابلیت استفاده‌ی مجدد دارند، برای برنامه‌های تحت وب توسعه یافته است. از آنها استفاده کنید. هر شیء واسطی که کاربر نهایی بتواند آن را «ببیند، بشنود، لمس کند یا به هر طریق دیگری ادراک کند» [Tog01] از هر کدام از چند کتابخانه‌ی اشیا قابل تامین است.

کاهش تأخیر (Latency Reduction). *به‌جای اینکه کاربر را وادار سازید تا منتظر شود یک عملیات درونی (مثلاً دانلود یک تصویر گرافیکی پیچیده) به پایان رسد، برنامه‌ی تحت وب باید به نحوی از قابلیت‌های چند کاره استفاده کند که کاربر بتواند به انجام کارهای خود ادامه دهد، طوری که انگار آن عملیات پایان یافته است.* علاوه بر کاهش تأخیر، آن را نیز باید به اطلاع کاربر رساند تا بداند چه اتفاقی در حال رخ‌دادن است. این شامل مواردی می‌شود که به‌دنبال خواهد آمد: (۱) فراهم‌آوردن بازخورد صوتی، هنگامی که انتخاب باعث کنش فوری توسط برنامه‌ی تحت وب نمی‌انجامد، (۲) به‌نمایش در آوردن یک ساعت شنی یا نوار پیشرفت که نشان دهد پردازش در حال انجام است و (۳) فراهم آوردن قدری سرگرمی (مثلاً یک پویانمایی یا متن نمایشی) در حالی که پردازش‌های طولانی رخ می‌دهد.

قابلیت یادگیری. واسط *برنامه‌ی تحت وب باید طوری طراحی شود که زمان یادگیری را به حداقل برساند و در صورت مرور برنامه‌ی تحت وب، نیاز به یادگیری دوباره به حداقل برسد.* به‌طور کلی، واسط باید بر یک طراحی ساده و خلاقانه تاکید ورزد که محتوا و قابلیت‌های عملیاتی را در گروه‌های آشکار در مقابل کاربر سازمان‌دهی کند.

استعاره‌ها (Metaphores). *واسطی که از یک استعاره تعاملی استفاده می‌کند، راحت‌تر قابل فراگیری و قابل به کارگیری است، مادامی که این استعاره برای برنامه و برای کاربر مناسب باشد.* استعاره باید تصاویر و مفاهیمی از تجربیات کاربر را به کمک بگیرد، ولی ضرورت ندارد که بازسازی دقیقی از تجربه‌ی کاربر از جهان واقعی باشد. برای مثال، یک سایت تجارت الکترونیک که پرداخت قبض برای موجودیتی مالی را انجام می‌دهد، از یک استعاره دسته چک برای کمک به کاربر استفاده می‌کند تا پرداخت قبض‌ها را زمان‌بندی کند. ولی هنگامی که کاربر، چکی «می‌کشد» ضرورتی ندارد که نام گیرنده‌ی وجه را کاملاً بنویسد چون می‌تواند آن را از فهرستی انتخاب کند یا براساس چند حرف اول تایپ شده پرداخت کند. استعاره بدون تغییر باقی می‌ماند، ولی کاربر از برنامه‌ی تحت وب کمک را دریافت می‌کند.

حفظ انسجام محصول کاری. *یک محصول کاری (مثلاً فرم تکمیل شده توسط کاربر یا فهرست مشخص‌شده به وسیله‌ی کاربر) باید به‌طور خودکار ضبط شود تا اگر خطایی رخ داد، محتوای آن از بین نرود.* هر کدام از ما این ناراحتی را تجربه کرده‌ایم که یک فرم طولانی را در برنامه‌ی تحت وبی پر کرده‌ایم و تنها به‌خاطر خطایی کوچک (که خود مرتکب شده‌ایم، یا برنامه‌ی تحت وب مرتکب شده است یا در انتقال از کلاینت به سرور رخ داده است) همه‌ی محتوای فرم از بین رفته است. برای

اندرز

استعاره‌ها، ایده‌های عالی به‌شمار می‌روند چون انعکاسی از تجربیات جهان واقعی‌اند. فقط اطمینان حاصل کنید که استعاره انتخابی شما به خوبی نزد کاربران نهایی شناخته شده باشد.

جلوگیری از این وضعیت، برنامه‌ی تحت وب باید طوری طراحی شود که همه‌ی داده‌های مشخص‌شده توسط کاربر را به‌صورت خودکار ضبط کند.

خوانایی. *همه‌ی اطلاعاتی که از طریق واسط ارائه می‌شوند باید برای پیر و جوان خوانا باشند.* طراح واسط باید بر سبک‌های خوانایی تایپ، اندازه‌ی فونت‌ها و انتخاب رنگ پس زمینه‌ای که تضاد را بهبود می‌بخشد، تاکید ورزد.

وضعیت پیگیری. *هر گاه که مناسب باشد، وضعیت تعامل کاربر باید پیگیری و ضبط شود، به‌طوری که کاربر بتواند از برنامه خارج شود و بعداً برگردد و از جایی که کار را رها کرده است، ادامه دهد.* به‌طور کلی، کوکی‌ها را می‌توان طوری طراحی کرد که اطلاعات وضعیتی را در خود نگهداری کنند. به هر حال، کوکی‌ها یک فن‌آوری بحث برانگیزند و برای برخی کاربران، سایر راهکارهای طراحی ممکن است خوشایندتر باشد.

گشت‌وگذار مرئی. یک واسط *برنامه‌ی تحت وب با طراحی خوب، این توهم را ایجاد می‌کند که کاربران درجای خود هستند و کار برای آن‌ها آورده می‌شود* [Tog01]. هنگامی که از این رویکرد استفاده شود، گشت‌وگذار، دغدغه‌ی کاربر نخواهد بود. در عوض، کاربر اشیای داده‌ای را بازیابی می‌کند و قابلیت‌هایی را برمی‌گزیند که از طریق واسط به نمایش در می‌آیند و اجرا می‌شوند.

نیلسن و واگنر [Nie96] چند دستورالعمل برای طراحی واسط (براساس طراحی دوباره یک برنامه‌ی تحت وب بزرگ) پیشنهاد می‌کنند که اصول پیشنهاد شده‌ی قبلی در این بخش را به خوبی تکمیل می‌کنند:

- سرعت خواندن مطالب روی مانیتور تقریباً ۲۵٪ آهسته‌تر از سرعت خواندن روی کاغذ است. بنابراین، خواننده را وادار نکنید که مقادیر انبوه متن را روی مانیتور بخواند به ویژه هنگامی که این متن، عملکرد برنامه‌ی تحت وب را توضیح می‌دهد یا به گشت‌وگذار کمک می‌کند.
- از پیام‌های «در دست احداث» بپرهیزید -پیوندی بیهوده که حاصلی جز ناامیدی ندارد.
- کاربران ترجیح می‌دهند که در پنجره حرکت نکنند. اطلاعات مهم را باید در ابعاد پنجره مرورگر بگنجانید.
- منوهای گشت‌وگذار و نوارهای عنوان مطالب باید به‌صورت سازگار طراحی شوند و باید در تمامی صفحات در دسترس، برای کاربر قابل دستیابی باشند. طراحی برای گشت‌وگذار نباید به قابلیت‌های مرورگر اتکا کند.
- زیبایی‌شناسی هرگز نباید بر قابلیت عملیاتی پیشی بگیرد. برای مثال، یک دکمه ساده ممکن است گزینه‌ی بهتری برای گشت‌وگذار باشد تا یک تصویر یا آیکون زیبا و دلپذیر که هدف و مقصود آن مبهم است.
- گزینه‌های گشت‌وگذار حتی برای کاربران عبوری باید واضح باشند. کاربر نباید مجبور باشد صفحه را بگردد تا بفهمد چگونه می‌تواند به سرویس یا محتوای دیگر متصل شود.

واسطی که خوب طراحی شده باشد، درک کاربر از محتوا یا سرویس‌های فراهم آمده توسط سایت را بهبود می‌بخشد. ضرورتی ندارد که پرزرق‌وبرق باشد بلکه همواره باید ساختار خوبی داشته باشد و از نظر ارگونومی مناسب باشد.

آدم‌ها در مقابل سایت‌های WWW ـ طراحی شده اصلاً صور نیست

یاکوب نیلسن و انت واگنر

SafeHome

مرور طراحی واسط

صحنه: دفتر داگ میلر.

نقش‌آفرینان: داگ میلر (مدیر گروه مهندسی نرم‌افزار *SafeHome*) وینود رامان، عضو تیم مهندسی نرم‌افزار محصول *SafeHome*.

گفتگو:

داگ: وینود، تو و تیمت فرصت پیدا کردید نمونه‌ی اولیه واسط تجارت الکترونیکی **SafeHomeAssured.com** را بازبینی کنید؟

وینود: بله ... همه‌ی ما از یک دیدگاه فنی به قضیه نگاه کردیم و چند تا یادداشت برداشتیم. من هم دیروز آنها را برای شارون آمدیر تیم برنامه‌ی تحت وب برای برون سپاری وظایف مربوط به وب سایت *SafeHome* فرستادم.

داگ: تو و شارون خودتان با هم در تماس باشید و درباره موارد جزئی با هم بحث کنید ... من یک خلاصه‌ای از مسائل مهم می‌خواهم.

وینود: در کل کارشان خوب بوده است. مشکل خاصی وجود نداشته است، ولی این یک واسط تجارت الکترونیکی معمولی است. گرافیک زیبا، چیدمان منطقی، همه‌ی قابلیت‌های مهم را در نظر گرفته‌اند ...

داگ (با ناراحتی لبخند می‌زند): ولی؟

وینود: خب، یک چیزهایی هست ...

داگ: مثلاً؟

وینود (در حالی که یک سری استوری بورد از نمونه‌ی اولیه واسط را به او نشان می‌دهد): اینها قابلیت‌های اصلی است که متوروی صفحه اصلی نمایش می‌دهد.

Learn about *SafeHome*
Describe your home
Get *SafeHome* component recommendations
Purchase a *SafeHome* system
Get technical support

این قابلیت‌ها مشکلی ندارند. همه‌ی آنها خوب هستند، ولی سطح انتزاع آنها درست نیست.

داگ: اینها قابلیت‌های اصلی‌اند، نه؟

وینود: درست است، ولی قضیه این است که ... می‌توانید سیستمی را با وارد کردن فهرست مولفه‌ها خرید کنید ... اگر نمی‌خواهید خانه را توصیف کنید. من فقط چهار گزینه را روی صفحه اصلی پیشنهاد می‌کنم.

Learn about *SafeHome*
Specify the *SafeHome* system you need
Purchase a *SafeHome* system
Get technical support

وقتی Specify the *SafeHome* system you need را انتخاب می‌کنید، دو گزینه‌ی

Select *SafeHome* components
Get *SafeHome* component recommendations

را دارید. اگر یک کاربر آگاه باشید، مولفه‌ها را از یک مجموعه منوهای گروه‌بندی شده برای حس‌گرها، دوربین‌ها، پانل‌های کنترل و غیره انتخاب خواهید کرد. اگر به کمک نیاز داشته باشید، تقاضای توصیه خواهید کرد و این شما را ملزم خواهد کرد که خانه را توصیف کنید. فکر می‌کنم این قدری منطقی‌تر باشد.

داگ: موافقم. در این مورد با شارون حرف زدی؟

وینود: نه، می‌خواهم اول با بازاریابی مطرح کنم؛ بعد با او تماس می‌گیرم.

۲-۵-۱۱ جریان کاریِ طراحی واسط برای برنامه‌های تحت وب

قبلاً در این فصل گفتیم که طراحی واسط کاربر با شناسایی خواسته‌های کاربری، وظیفه‌ای و محیطی آغاز می‌شود. هنگامی که وظایف کاربر مشخص شد، سناریوهای کاربری (use caseها) ایجاد و تحلیل می‌شوند تا مجموعه اشیا و کنش‌های واسط تعریف شوند.

اطلاعات موجود در مدل خواسته‌ها، مبنایی برای چیدمان صفحه نمایش تشکیل می‌دهد که طراحی گرافیکی و محل قرار گرفتن آیکون‌ها، تعریف متن نمایشی و توصیفی، مشخص‌کردن موقعیت پنجره‌ها و مشخص‌کردن آیتم‌های منوهای اصلی و فرعی را به تصویر می‌کشد.

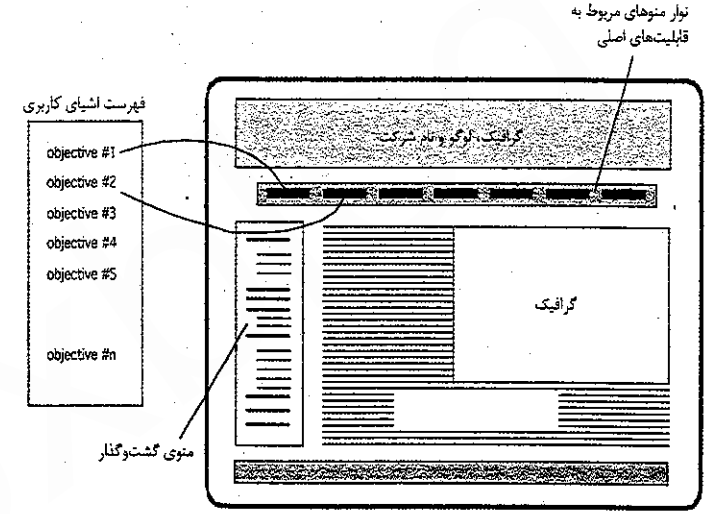
سپس از ابزارهایی برای تهیه‌ی نمونه‌ی اولیه و سرانجام پیاده‌سازی مدل طراحی واسط استفاده می‌شود. وظایف زیرنشان‌گر یک جریان کاری مقدماتی برای طراحی واسط برنامه‌ی تحت وب هستند:

۱. **مرور بر اطلاعات موجود در مدل خواسته‌ها و پالایش آن در صورت نیاز.**

۲. **تهیه‌ی اتودی اولیه از چیدمان واسط برنامه‌ی تحت وب.** نمونه‌ی اولیه یک واسط (که شامل چیدمان آن می‌شود) ممکن است به‌عنوان بخشی از فعالیت مدل‌سازی خواسته‌ها تهیه شود. اگر چیدمان از قبل موجود باشد، باید مرور و در صورت نیاز پالایش گردد. اگر چیدمان واسط توسعه داده نشده باشد، باید با طرف‌های ذی‌نفع کار کنید و آن را در این زمان توسعه دهید. طرحی از اتود اولیه چیدمان در شکل ۴-۱۱ نشان داده شده است.

۳. **نگاشت اهداف کاربر در کنش‌های ویژه واسط.** برای اکثریت برنامه‌های تحت وب ، کاربر یک مجموعه نسبتاً کوچک از اهداف اولیه دارد. این اهداف، چنان که در شکل ۴-۱۱ نشان داده شده است، باید در کنش‌های ویژه واسط نگاشت شوند. در اصل باید به این پرسش پاسخ گفته شود: «واسط چگونه به کاربر امکان می‌دهد که به هر کدام از اهداف خود برسد؟»

۴. **تعریف مجموعه‌ای از وظایف کاربر که به هر کنش مربوط می‌شود.** هر کنش واسط (مثلاً «خرید یک محصول») با مجموعه‌ای از وظایف کاربر در ارتباط است. این وظایف طی مدل‌سازی خواسته‌ها شناسایی شده‌اند. در طول طراحی، آنها را باید در تعامل‌های خاصی نگاشت که شامل مسائل گشت‌وگذار، اشیای محتوایی و قابلیت‌های عملیاتی برنامه‌ی تحت وب می‌شود.



شکل ۴-۱۱ نگاشت اهداف کاربر به کنش‌های واسط.

۵. **تهیه استوری بورد برای هر کنش واسط.** با در نظر گرفتن هر کدام از کنش‌های واسط، یک سری تصاویر استوری بورد باید تهیه شود که چگونگی پاسخ گویی به تعامل کاربر را مجسم می‌کنند. اشیای محتوایی باید شناسایی شوند (حتی اگر طراحی و تهیه نشده باشند)، قابلیت عملیاتی برنامه‌ی تحت وب باید نشان داده شود و پیوندهای گشت‌وگذار باید خاطرنشان شوند.

۶. **پالایش چیدمان واسط و استوری بوردها با استفاده از ورودی حاصل از طراحی زیبایی‌شناختی.** در اکثر موارد، مسؤولیت طراحی حدودی و تهیه استوری‌بورد بر عهده شماست، ولی شکل و شمایل زیبایی‌شناختی برای اکثر سایت‌های تجاری بزرگ غالباً وظیفه افراد هنرمند است نه افراد فنی. طراحی گرافیکی (فصل ۱۳) با کار اجرا شده توسط طراح واسط منسجم خواهد شد.

۷. **شناسایی اشیایی از واسط کاربر که برای پیاده‌سازی واسط ضروری‌اند.** این وظیفه ممکن است مستلزم جستجو در میان کتابخانه‌ای از اشیای موجود برای یافتن اشیا (کلاس‌های) قابل استفاده‌ی مجدد و مناسب برای واسط برنامه‌ی تحت وب باشد. به‌علاوه، همه‌ی کلاس‌های سفارشی در این زمان مشخص می‌شوند.

۸. **توسعه‌ی یک نمایش رویه‌ای از تعامل کاربر با واسط.** برای این وظیفه اختیاری، از نمودارهای ترتیبی و/یا نمودارهای فعالیت UML (پیوست ۱) برای به تصویر کشیدن جریان فعالیت‌ها (و تصمیم‌گیری‌هایی) استفاده می‌شود که هنگام تعامل کاربر با برنامه‌ی تحت وب رخ می‌دهند.

۹. **توسعه‌ی یک نمایش رفتاری از واسط.** در این وظیفه اختیاری، از نمودارهای حالت UML (پیوست ۱) برای نشان دادن گذارهای حالت و رویدادهایی استفاده می‌شود که باعث این گذارها می‌شوند. سازوکارهای کنترلی (یعنی اشیا و کنش‌های در دسترس کاربر برای تغییر دادن حالت یک برنامه‌ی تحت وب) تعریف می‌شوند.

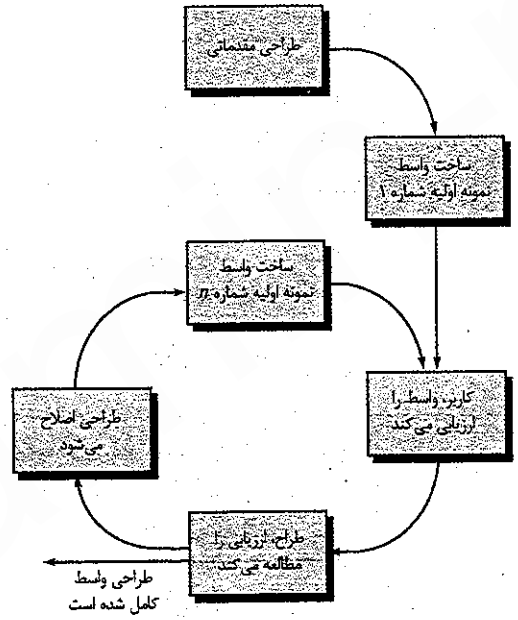
۱۰. **توصیف چیدمان واسط برای هر حالت.** با استفاده از طراحی توسعه یافته در وظایف ۲ و ۵، یک چیدمان خاص یا تصویر صفحه نمایش به هر کدام از حالت‌های برنامه‌ی تحت وب توصیف شده در وظیفه ۸ ربط داده می‌شود.

۱۱. **پالایش و مرور مدل واسط.** در مرور واسط «قابلیت استفاده» باید کانون توجه قرار گیرد.

لازم به ذکر است که مجموعه وظایف نهایی که انتخاب می‌کنید باید با خواسته‌های ویژه برنامه‌ای که قرار است ساخته شود، تطبیق داده شود.

۶-۱۱ ارزیابی طراحی

هنگامی که نمونه‌ی اولیه‌ی عملیاتی واسط کاربر را تهیه می‌کنید، باید آن را ارزیابی کنید تا معلوم شود که آیا نیازهای کاربر را برآورده می‌سازند یا خیر. ارزیابی می‌تواند شامل طیفی از رسمیت باشد که از یک «آزمایش رانندگی» غیررسمی (که در آن کاربر بازخوردی فی‌البداهه فراهم می‌سازد) تا یک مطالعه طراحی شده رسمی (که از روش‌های آماری برای ارزیابی پرسش نامه‌های پرشده توسط جمعیتی از کاربران نهایی استفاده می‌کند) در تغییر باشد.



شکل ۵-۱۱ چرخه‌ی ارزیابی طراحی واسط.

چرخه‌ی ارزیابی واسط کاربر، چیزی شبیه به شکل ۵-۱۱ خواهد بود. پس از کامل شدن مدل طراحی، یک نمونه‌ی اولیه سطح نخست ایجاد می‌شود. این نمونه‌ی اولیه توسط کاربر ارزیابی می‌شود[1] که او توضیحات مستقیمی درباره اثربخش بودن واسط در اختیار شما قرار می‌دهد. به‌علاوه، اگر از تکنیک‌های رسمی ارزیابی (پرسش نامه یا برگه‌های ارزش گذاری) استفاده شود، می‌توانید اطلاعات را از این داده‌ها استخراج کنید (مثلاً ۸۰٪ همه‌ی کاربران از سازوکار ضبط‌کردن فایل‌های داده‌ای راضی نبودند). براساس ورودی‌های کاربران، اصلاحاتی روی طراحی به عمل می‌آید و نمونه‌ی اولیه سطح بعدی تهیه می‌شود. این چرخه ارزیابی چندان ادامه می‌یابد که دیگر نیازی به اصلاحات بیشتر در طراحی واسط نباشد.

رویکرد تهیه نمونه‌ی اولیه موثر است، ولی آیا این امکان وجود دارد که کیفیت واسط کاربر را پیش از ساختن نمونه‌ی اولیه بسنجیم؟ اگر مسائل بالقوه را زودهنگام، شناسایی و تصحیح کنیم، تعداد چرخه‌های ارزیابی کاهش می‌یابد و زمان توسعه کوتاه می‌شود. اگر یک مدل طراحی برای واسط تهیه شده باشد، چند ملاک ارزیابی [Mor81] را می‌توان در اولین مرورهای طراحی به کار برد.

۱. طول و پیچیدگی مدل خواسته‌ها یا مشخصات نوشته‌شده‌ی سیستم و واسط آن، شاخصی از میزان یادگیری لازم برای کاربران سیستم به‌دست می‌دهد.
۲. تعداد وظایف کاربری مشخص شده و تعداد میانگین کنش‌ها به ازای هر وظیفه، شاخصی از زمان تعامل و بازدهی کلی سیستم به‌دست می‌دهد.
۳. تعداد کنش‌ها، وظایف و حالت‌های سیستم که توسط مدل طراحی مشخص می‌شود، بار حافظه‌ای را نشان می‌دهد که بر کاربران سیستم وارد می‌آید.
۴. سبک واسط، امکانات راهنمایی و پروتکل مدیریت خطا، شاخصی از پیچیدگی واسط و میزان پذیرش آن توسط کاربر فراهم می‌آورد.

هنگامی که نخستین نمونه‌ی اولیه ساخته شد، می‌توانید انواع داده‌های کیفی و کمی را جمع‌آوری کنید که به ارزیابی واسط کمک خواهند کرد. برای جمع‌آوری داده‌های کیفی می‌توانید پرسش نامه‌هایی را در میان کاربران این نمونه‌ی اولیه توزیع کنید. پرسش‌ها می‌توانند از این قرار باشند: (۱) پاسخ ساده آری/خیر، (۲) پاسخ عددی، (۳) پاسخ مقیاس‌بندی شده (موضوعی)، (۴) مقیاس‌های لیکرت (مثلاً کاملاً موافق، قدری موافق)، (۵) پاسخ درصدی (موضوعی) یا (۶) تشریحی.

اگر داده‌های کمّی مطلوب باشند، شکلی از تحلیل مطالعه زمانی را می‌توان اجرا کرد. کاربران در زمان تعامل مشاهده می‌شوند و داده‌هایی -از قبیل تعداد وظایفی که طی یک دوره زمانی استاندارد به درستی به انجام می‌رسند، فراوانی کنش‌ها، ترتیب کنش‌ها، زمان صرف شده برای «نگریستن» به صفحه‌نمایش، تعداد و نوع خطاها، زمان بیرون آمدن از خطا، زمان صرف شده برای استفاده از راهنما و تعداد مراجعات به راهنما به ازای دوره زمانی استاندارد- جمع‌آوری و به‌عنوان راهنمایی برای اصلاح واسط به کار برده می‌شوند.

بحث کاملی درباره روش‌های ارزیابی واسط کاربر از حوصله این کتاب خارج است. برای اطلاعات بیشتر [Hac98] و [Sto05] را ببینید.

[1] شایان ذکر است که کارشناسان ارگونومی و طراحی واسط نیز ممکن است واسط را مرور کنند. این مرورها، مرورهای اکتشافی یا کندوکاوهای شناختی نامیده می‌شوند.

۱۱-۷ خلاصه

واسط کاربر مهمترین عنصر سیستم‌ها یا محصولات کامپیوتری به شمار می‌رود. اگر طراحی واسط، ضعیف باشد، توانایی کاربر در استفاده از توان محاسباتی و محتوای اطلاعاتی یک برنامه کاربردی ممکن است با مشکل جدی مواجه شود. در واقع، یک واسط ضعیف ممکن است باعث شود برنامه‌ای که از نظر پیاده‌سازی کاربرد مورد نظر به خوبی طراحی شده است، با شکست مواجه گردد.

سه اصل مهم، راهنمایی طراحی واسط کاربر به طرزی اثربخش هستند: (۱) سپردن کنترل به کاربر، (۲) کاهش دادن بار حافظه‌ی کاربر و (۳) سازگارکردن واسط. برای دستیابی به واسطی که این سه اصل را در خود جای دهد، فرایند طراحی باید به شیوه‌ای سازمان یافته اجرا شود.

توسعه‌ی واسط کاربر با یک سری وظایف تحلیل آغاز می‌شود. با تحلیل کاربران، پروفایلی از انواع کاربران نهایی و منابع تجاری و فنی گوناگون به‌دست می‌آید. با تحلیل وظایف، وظایف و کنش‌های کاربر با استفاده از یک روش شیءگرا یا تشریحی و با به‌کارگیری use case ها، تشریح وظایف و اشیاء تحلیل جریان کاری و نمایش‌های سلسله مراتب وظایف برای درک کامل تعامل میان انسان و کامپیوتر، تعیین می‌شوند. با تحلیل محیطی، ساختارهای فیزیکی و اجتماعی که واسط باید در آنها کار کند، تعریف می‌شود.

هنگامی که وظایف شناسایی شد، سناریوهای کاربری تهیه و تحلیل می‌شوند تا مجموعه کنش‌ها و اشیای واسط تعریف شوند. به این ترتیب، مبنایی برای تعیین چیدمان که طراحی گرافیکی و محل قرار گرفتن آیکون‌ها را به تصویر می‌کشد، تعیین متون توصیفی، مشخص‌کردن ترتیب پنجره‌ها و مشخص‌کردن آیتم‌های منوهای اصلی و فرعی فراهم می‌سازد. مسائل طراحی از قبیل زمان پاسخ‌دهی، ساختار فرمان‌ها و کنش‌ها، مدیریت خطا و امکانات راهنمایی به موازات پالایش شدن مدل طراحی در نظر گرفته می‌شود. از انواع ابزارهای پیاده‌سازی در ساخت نمونه‌ی اولیه‌ای برای ارزیابی توسط کاربران استفاده می‌شود.

همانند طراحی واسط برای نرم‌افزارهای سنتی، طراحی واسط برای برنامه‌های تحت وب نیز ساختار و سازمان‌دهی واسط کاربر را توصیف می‌کند و شامل نمایشی از چیدمان صفحه نمایش، تعریف شیوه‌های تعامل و توصیف سازوکارهای گشت‌وگذار می‌شود. مجموعه‌ای از اصول طراحی و یک جریان کاری طراحی، طراح برنامه‌ی تحت وب را در طراحی سازوکارهای کنترلی واسط و چیدمان آن راهنمایی خواهند کرد.

واسط کاربر، پنجره‌ای فراروی نرم‌افزار است. واسط در بسیاری موارد، ادراک کاربر از کیفیت سیستم را شکل می‌دهد. اگر این «پنجره» غبار گرفته شود، موج‌دار شود یا شکسته شود، کاربر ممکن است سیستمی پرقدرت را پس بزند.

مسائل و نکاتی برای تعمق

۱-۱۱ بدترین واسطی را که تاکنون دیده‌اید، شرح دهید و آن را از نظر مفاهیمی که در این فصل معرفی شده نقد کنید. بهترین واسطی را که تاکنون دیده‌اید، شرح دهید و آن را از نظر مفاهیم معرفی شده در این فصل نقد کنید.

۲-۱۱ دو اصل طراحی دیگر توسعه دهید که «کاربر را در جایگاه کنترل» قرار می‌دهند.

۱۱-۳ دو اصل طراحی دیگر توسعه دهید که از «بار حافظه کاربر» می‌کاهند.

۱۱-۴ دو اصل طراحی دیگر توسعه دهید که «واسط را سازگار» می‌کنند.

۱۱-۵ یکی از برنامه‌های کاربردی تعاملی زیر (یا برنامه‌ای که استادتان تکلیف می‌کند) را در نظر بگیرید:
الف. یک سیستم نشر رومیزی
ب. یک سیستم طراحی به کمک کامپیوتر (CAD)
پ. یک سیستم طراحی داخلی
ت. یک سیستم ثبت‌نام خودکار برای دوره‌های دانشگاهی
ث. یک سیستم مدیریت کتابخانه
ج. یک سیستم رأی‌گیری انتخاباتی به کمک اینترنت
چ. یک سیستم بانکی خانوادگی؛ ح. یک برنامه کاربردی تعاملی.
مدل طراحی، مدل کاربر، تصویر سیستم و برداشت سیستم را برای هر کدام از سیستم‌های بالا توسعه دهید.

۱۱-۶ تحلیل مفصلی از وظایف را برای هر یک از سیستم‌های مسأله ۵-۱۱ اجرا کنید. از یک روش توضیحی یا شیءگرا استفاده کنید.

۱۱-۷ دست کم پنج پرسش به فهرست تهیه شده برای تحلیل محتویات در بخش ۳-۳-۱۱ اضافه کنید.

۱۱-۸ در ادامه مسأله ۵-۱۱، اشیا و کنش‌های واسط را برای هر برنامه‌ای که انتخاب کرده‌اید، تعیین کنید. نوع هر شیء را مشخص کنید.

۱۱-۹ یک مجموعه از چیدمان صفحه‌نمایش همراه با عناصر منوهای اصلی و فرعی برای برنامه‌ای که در مسأله ۵-۱۱ انتخاب کرده‌اید، توسعه دهید.

۱۱-۱۰ یک مجموعه از چیدمان صفحه‌نمایش همراه با عناصر منوهای اصلی و فرعی برای سیستم پیشرفته *SafeHome* توسعه دهید. می‌توانید به سلیقه خود، روش متفاوت با آنچه که در شکل ۳-۱۱ نشان داده شده است، برای چیدمان صفحه‌نمایش در نظر بگیرید.

۱۱-۱۱ روش خود را در قبال تسهیلات راهنمای کاربر جهت مدل طراحی و تحلیل وظایفی که در مسائل ۶ تا ۸ انجام‌دادید، شرح دهید.

۱۱-۱۲ چند مثال بیاورید که نشان دهد چرا تغییرپذیری در زمان پاسخ می‌تواند مسأله‌ساز شود.

۱۱-۱۳ روشی توسعه دهید که به طور خودکار پیام‌های خطا را با تسهیلات همراه منسجم کند. یعنی سیستم به طور خودکار نوع خطا را تشخیص داده یک پنجره راهنما با پیشنهادهایی جهت تصحیح آن فراهم آورد. یک طراحی نرم‌افزار کامل انجام دهید که در آن ساختمان داده‌ها و الگوریتم‌های مناسب در نظر گرفته شده باشند.

۱۱-۱۴ یک پرسش‌نامه ارزیابی واسط تهیه کنید که حاوی ۲۰ پرسش کلی باشد و در اکثر واسط‌ها قابل استفاده باشد. پرسش‌نامه را به ۱۰ نفر از همکلاسی‌های خود بدهید تا درباره یک سیستم تعاملی مورد استفاده همه‌ی آنها پر کنند. نتایج را خلاصه کرده به کلاس گزارش دهید.

فصل ۱۲

طراحی مبتنی بر الگوها

نگاهی گذرا

طراحی مبتنی بر الگوها چیست؟ در طراحی مبتنی بر الگوها، برنامه کاربردی جدید، با یافتن مجموعه‌ای از راهکارهای اثبات شده در یک مجموعه مسائل کاملاً مشخص ایجاد می‌شود. هر مسأله و راهکار آن به‌وسیله‌ی یک الگوی طراحی توصیف می‌شود که توسط سایر مهندسان نرم‌افزاری بررسی و فهرست‌بندی شده است که هنگام طراحی برنامه‌های کاربردی دیگر با این مسأله مواجه شده‌اند و راهکاری برای آن پیاده‌سازی کرده‌اند. هر الگوی طراحی برای بخشی از مسأله‌ای که قرار است حل شود، یک رویکرد و روش اثبات شده در اختیار شما می‌گذارد.

چه کسی آن را انجام می‌دهد؟ مهندس نرم‌افزار هر کدام از مسائل مشاهده شده برای یک برنامه کاربردی جدید را بررسی می‌کند و سپس می‌کوشد با جستجو در یک یا چند مخزن الگو، راهکاری مرتبط با آن بیابد.

چرا اهمیت دارد؟ آیا تاکنون اصطلاح «اختراع دوباره‌ی چرخ» را شنیده‌اید؟ این داستان در توسعه‌ی نرم‌افزارها همواره تکرار می‌شود و نتیجه‌ای جز اتلاف زمان و انرژی ندارد. با به‌کارگیری الگوهای طراحی موجود، می‌توانید برای یک مسأله‌ی مشخص، راهکاری اثبات شده به‌دست آورید. با به‌کارگیری هر الگو، راهکارها منسجم می‌شوند و برنامه‌ای که قرار است ساخته شود، به طراحی کامل نزدیک‌تر می‌شود.

مراحل کار کدام است؟ مدل خواسته‌ها بررسی می‌شود تا از مسائلی که قرار است حل شود، مجموعه‌ای سلسله مراتبی جدا شود. فضای مسأله به گونه‌ای تقسیم‌بندی می‌شود که زیر مجموعه‌های مسائل مرتبط با وظایف و ویژگی‌های نرم‌افزار را بتوان شناسایی کرد. مسائل را بر اساس نوع نیز می‌توان سازمان‌دهی کرد: معماری، در سطح مؤلفه‌ها، الگوریتمی، واسط کاربر و غیره. هنگامی که زیر مجموعه‌ای از مسائل تعریف شد، یک یا چند مخزن الگو مورد جستجو قرار می‌گیرد تا معلوم شود که آیا یک الگوی طراحی، که در سطح مناسبی از انتزاع نمایش داده می‌شود، وجود دارد یا خیر. الگوهای قابل استفاده به نیازهای خاص نرم‌افزاری که قرار است ساخته شود، تطبیق داده می‌شوند. در صورتی که نتوان الگویی برای مسأله پیدا کرد از روش حل سفارشی استفاده می‌شود.

محصول کاری چیست؟ یک مدل طراحی که ساختار معماری، واسط کاربر و جزئیات طراحی در سطح مؤلفه‌ها را به تصویر می‌کشد.

چگونه اطمینان حاصل کنم که درست از انجام کار بر آمده‌ام؟ با تبدیل هر الگوی طراحی به عنصری از مدل طراحی، محصولات کاری از نظر وضوح، درستی، کامل بودن و سازگاری با خواسته‌ها و با یکدیگر بازبینی می‌شوند.

برای هر کدام از ما پیش آمده که در مواجهه با یک مسأله‌ی طراحی از خود بپرسیم: آیا کسی برای این مسأله راهکاری پیدا کرده است؟ پاسخ تقریباً همیشه مثبت است! مسأله، یافتن راهکار است؛ حصول اطمینان از این که واقعاً در مسأله‌ی مورد نظر شما می‌گنجد؛ شناخت قیدوبندهایی که ممکن است شیوه‌ی به‌کارگیری راهکار را محدود سازند و سرانجام، تبدیل راهکار پیشنهادی به محیط طراحی شما.

ولی اگر راهکارها به نوعی تدوین شده باشند، چطور؟ اگر راهی برای توصیف مسأله وجود داشت (به‌طوری که بتوان به دنبال آن گشت) و روش سازمان یافته‌ای برای نمایش راهکار مسأله وجود می‌داشت، چطور؟ به نظر می‌رسد که مسائل نرم‌افزار با به‌کارگیری یک قالب استاندارد شده، تدوین و توصیف شده‌اند و راهکارهایی (همراه با قیدوبندهای مربوط) برای آن‌ها پیشنهاد شده است. این روش تدوین شده برای توصیف مسائل و راهکارهای آن‌ها، که *الگوهای طراحی* نامیده می‌شود، به جامعه‌ی مهندسی نرم‌افزار این امکان را می‌دهد که دانش طراحی را به شیوه‌ای به چنگ آورد که استفاده‌ی مجدد از آن میسر گردد.

تاریخچه‌ی اولیه‌ی الگوهای نرم‌افزار را نه یک دانشمند علم کامپیوتر بلکه یک معمار به نام کریستوفر الکساندر شروع کرده است؛ او بود که دریافت هرگاه یک ساختمان طراحی می‌شود، با مجموعه‌ای از مسائل تکراری روبه‌رو می‌شویم. او این مسائل تکراری و راهکارهای آن‌ها را *الگو* نامید و به شیوه‌ی زیر تعریف کرد [Ale77].

هر الگو، مسأله‌ای را توصیف می‌کند که بارها و بارها در محیط ما رخ می‌دهد و سپس هسته‌ی راهکار آن مسأله را طوری توصیف می‌کند که بتوان از آن راهکار هزاران بار استفاده کرد، بدون این که حتی دوبار به شیوه‌ای یکسان انجام شود.

ایده‌های الکساندر برای نخستین بار در کتاب‌های [Gam95]، بوشمان [Bus96] و بسیاری از همکاران ایشان وارد دنیای نرم‌افزار شدند.[1] امروزه، ده‌ها مخزن برای الگوها وجود دارد و از طراحی مبتنی بر الگوها می‌توان در دامنه‌های کاربردی متفاوت استفاده کرد.

۱-۱۲ الگوهای طراحی (Design Patterns)

الگوی طراحی را می‌توان «یک قاعده‌ی سه بخشی دانست که واسط میان یک حیطه معین، یک مسأله و یک راهکار را بیان می‌کند» [Ale79]. برای طراحی نرم‌افزار، حیطه به خواننده این امکان را می‌دهد تا محیطی را که مسأله در آن جای دارد، درک کند و دریابد چه راهکاری ممکن است در این محیط مناسب باشد. مجموعه‌ای از خواسته‌ها، از جمله محدودیت‌ها و قیدوبندها، به‌عنوان سیستم *نیروهای* تأثیرگذار بر شیوه‌ی تفسیر مسأله در حیطه‌اش و چگونگی به‌کارگیری مؤثر آن راهکار عمل می‌کند.

نکته‌ی کلیدی
نیروها آن مجموعه از خواص مسأله و صفات راهکار هستند که بر شیوه‌ی طراحی قیدوبند اعمال می‌کنند

به‌منظور درک بهتر این مفاهیم، وضعیتی را در نظر بگیرید[2] که در آن فردی باید میان نیویورک و لس‌آنجلس سفر کند. در این حیطه، سفر در کشوری صنعتی (ایالات متحده)، با استفاده از زیرساخت‌های حمل و نقلی موجود (جاده‌ها، خطوط راه آهن و خطوط هوایی) رخ می‌دهد. سیستم نیروهایی که بر شیوه‌ی حل مسأله‌ی سفر تأثیر می‌گذارد، عبارت خواهد بود از: این شخص می‌خواهد

[1] بحث‌های اولیه در خصوص الگوهای نرم‌افزار وجود دارند ولی در این دو کتاب کلاسیک، موضوع برای نخستین بار در قالبی منسجم ارائه شد.

[2] این مثال از [Cop05] برگرفته شده است.

با چه سرعتی از نیویورک به لس‌آنجلس برسد، آیا سفر شامل تماشای مناظر و اطراق بین راهی هم می‌شود، او چقدر پول می‌تواند خرج کند، آیا قصد خاصی از سفر دارد، و چه وسایل نقلیه شخصی در اختیار خود دارد. با توجه به این نیروها، مسأله‌ی (سفر از نیویورک به لس آنجلس) را بهتر می‌توان تعریف کرد. برای مثال، بررسی (جمع آوری خواسته‌ها) نشان می‌دهد که فرد، پول بسیار کمی دارد، تنها یک دوچرخه دارد (و به دوچرخه‌سواری مشتاق است)، دوست دارد این سفر را به منظور جمع آوری پول برای مؤسسه خیریه مورد علاقه اش انجام دهد و زمان زیادی هم در اختیار دارد. راهکار این مسأله، با توجه به حیطه و سیستم نیروها، ممکن است یک سفر سرتاسری با دوچرخه باشد. اگر نیروهای دیگری در کار بودند (مثلاً زمان سفر باید کمینه شود و هدف سفر، یک نشست تجاری باشد)، ممکن بود راهکاری دیگر مناسب‌تر باشد.

منطقی است استدلال کنیم که اکثر مسائل چندین راهکار دارند، ولی تنها یکی از آن‌هاست که در حیطه‌ی مسأله‌ی موجود، مناسب است و می‌تواند مؤثر واقع شود. سیستم نیروهاست که باعث می‌شود طراح، راهکاری خاص را برگزیند. هدف، فراهم آوردن راهکاری است که به بهترین نحو با سیستم نیروها همخوانی داشته باشد حتی هنگامی که این نیروها با هم در تناقض باشند. سرانجام، هر راهکار دارای پیامدهایی است که ممکن است بر سایر جنبه‌های نرم‌افزار تأثیر بگذارند و ممکن است خودشان برای سایر مسائلی که قرار است در سیستم بزرگتری حل شوند، بخشی از سیستم نیروها باشند.

کوپلین [Cop05] مشخصات الگوی طراحی اثربخش را چنین بر می‌شمرد:

- *مسأله‌ای را حل می‌کند.* الگوها، راهکارها را به چنگ می‌آورند، نه صرفاً اصول یا راهبردهای انتزاعی را.
- *مفهومی اثبات شده است.* الگوها راهکارهایی را به چنگ می‌آورند که دارای سابقه‌ی مشخص باشند نه بر اساس نظریه‌پردازی و گمانه‌زنی.
- *راهکار، واضح نیست.* بسیاری از تکنیک‌های حل مسأله (از قبیل روش‌ها یا الگوهای طراحی نرم‌افزار) سعی در به‌دست آوردن راهکارها از اصول نخست دارند. بهترین الگوها، به‌طور غیرمستقیم برای مسأله راهکاری *تولید می‌کنند*- یک رویکرد ضروری برای اکثر مسائل دشوار در طراحی.
- *یک رابطه را توصیف می‌کند.* الگوها پیمانه‌ها را توصیف نمی‌کنند، بلکه ساختارهای سیستمی و سازوکارهای عمیق‌تری را توصیف می‌کنند.
- *الگو دارای یک مؤلفه انسانی چشمگیر است (دخالت انسان را به حداقل می‌رساند).* همه‌ی نرم‌افزارها به رفاه و کیفیت زندگی انسان کمک می‌کنند؛ بهترین الگوها به صراحت، جذب زیبایی‌شناسی و رفاه می‌شوند.

به بیانی حتی عمل‌گراتر، الگوی طراحی خوب، دانش طراحی عملی را که به سختی به‌دست می‌آید، به شیوه‌ای در اختیار می‌گیرد که دیگران آن دانش را «هزاران بار دوباره به‌کار ببرند، بدون این که حتی دو بار این کار به‌صورت یکسان انجام شود.» الگوی طراحی شما را از «اختراع دوباره چرخ» یا حتی بدتر از آن، اختراع «چرخ جدیدی» که اندکی تاب دارد، برای به‌کارگیری در وضعیت مورد نظر شما بیش از حد کوچک است و برای زمینی که باید روی آن بچرخد بیش از حد باریک است، مصون نگه می‌دارد. الگوهای طراحی، اگر به‌طور مؤثر استفاده شوند، بدون تردید از شما یک طراح نرم‌افزار بهتر خواهند ساخت.

> «مسؤولیت ما عبارت است از انجام و آموختن آن‌چه که در توانمان است، بهبود بخشیدن به راهکارها و سپس واگذاری آن‌ها به دیگران.»
> **ریچارد پ. فاینمن**

۱-۱-۱۲ انواع الگوها

یکی از دلایلی که مهندسان به الگوهای طراحی علاقه دارند (و فریب آن را می‌خورند) آن است که انسان‌ها ذاتاً در تشخیص الگوها مهارت دارند. اگر چنین نبود، در فضا و زمان منجمد می‌شدیم- چون قادر به فراگیری از تجربیات پیشین نبودیم، به دلیل ناتوانی در تشخیص شرایطی که ممکن است به خطر بالا منجر شود، میلی به تهور و جسارت نداشتیم و در دنیایی که به نظر می‌رسد هیچ نظم یا سازگار منطقی بر آن حاکم نیست، سرگردان بودیم. خوشبختانه، هیچ کدام از این شرایط رخ نمی‌دهد چون در واقع در هر جنبه از زندگی ما الگوهایی به چشم می‌خورد.

در جهان واقعی، الگوهایی می‌شناسیم که با گذر زمان و از طریق تجربه به‌دست آمده‌اند. ما این الگوها را بلافاصله تشخیص می‌دهیم و ذاتاً می‌فهمیم که چه معنایی دارند و چگونه از آن‌ها می‌توان استفاده کرد. برخی از این الگوها دیدی از یک پدیده‌ی تکراری به‌دست می‌دهند. برای مثال، در بزرگراه مشغول حرکت از محل کار به سمت خانه‌اید که رادیو به اطلاع شما می‌رساند تصادفی در جهت مخالف رخ داده است. فاصله‌ی شما از محل تصادف، چهار کیلومتر است، ولی از هم اکنون می‌بینید که ترافیک کند شده است، الگویی که آن را **Rubbernecking** می‌نامیم. افرادی که در جهت شما حرکت می‌کنند، از سرعت خود می‌کاهند تا بهتر ببینند چه اتفاقی در طرف مقابل افتاده است. الگوی **Rubbernecking** نتایجی به‌دست می‌دهد که تا حد زیادی قابل پیش‌بینی است (گروه ترافیکی)، ولی کاری بیشتر از توصیف یک پدیده انجام نمی‌دهد. در فرهنگ اصطلاحات الگوها، می‌توان آن را الگوی *غیر مولد* (non generative) نامید زیرا حیطه و مسأله را توصیف می‌کند، اما هیچ راهکار قاطعی ارائه نمی‌دهد.

هنگام پرداختن به الگوهای طراحی در جستجوی شناسایی و مستندسازی الگوهای *مولد* (generative) هستیم. یعنی، الگویی را شناسایی می‌کنیم که جنبه‌ای مهم و تکرارپذیر از سیستم را توصیف می‌کند و شیوه‌ی ساخت آن جنبه را در سیستمی از نیروها که در یک حیطه‌ی مفروض، منحصر به فرد هستند، در اختیارمان قرار می‌دهد. در شرایط ایده‌آل، مجموعه‌ای از الگوهای طراحی مولد را می‌توان «ایجاد» یک سیستم کامپیوتری یا برنامه‌ی کاربردی دانست که معماری آن، تطبیق یافتن با تغییرات را میسر می‌سازد. «کاربرد پیاپی چند الگو، که هر یک حاوی مسأله و نیروهای خاص خود است، راهکار بزرگتری را بسط می‌دهد که به‌طور مستقیم در نتیجه‌ی راهکارهای کوچک نمود پیدا می‌کند» [App00] و گاهی *تولیدی* نامیده می‌شود.

نکته‌ی کلیدی

الگوی مولد، نه تنها مسأله، حیطه و نیروها را توصیف می‌کند بلکه راهکاری عملی برای این مسأله شرح می‌دهد.

الگوهای طراحی شامل طیف گسترده‌ای از انتزاع‌ها و کاربردها می‌شوند. *الگوهای معماری*، مسائل طراحی گسترده‌ای را توصیف می‌کنند که با به‌کارگیری یک رویکرد ساختاری حل می‌شوند. *الگوهای داده‌ای*، مسائل داده‌گرای تکراری و مسائل مدل‌سازی داده‌ها را توصیف می‌کنند که در حل این مسائل قابل استفاده‌اند. *الگوهای مؤلفه‌ای* (که از آن‌ها به‌عنوان *الگوهای طراحی* نیز یاد می‌شود) به مسائل مرتبط با توسعه‌ی زیر سیستم‌ها و مؤلفه‌ها، شیوه‌ی برقراری ارتباط آن‌ها با یکدیگر و تعیین مکان آن‌ها در یک معماری بزرگتر می‌پردازند. *الگوهای طراحی واسط*، مسائل واسط کاربری متداول و راهکار آن‌ها را با سیستمی از نیروها توصیف می‌کنند که شامل خصوصیات کاربران نهایی می‌شود. *الگوهای برنامه‌های تحت وب* به مجموعه مسائلی اختصاص دارند که هنگام ساخت این نوع برنامه‌ها مشاهده می‌شوند و غالباً خود شامل بسیاری از الگوهایی می‌شوند که هم‌اکنون ذکر شد. در سطح پایین‌تری

از انتزاع، *اصطلاحات* هستند که شیوه‌ی پیاده‌سازی کل یک الگوریتم خاص یا بخشی از آن یا ساختمان داده‌های مربوط به یک مؤلفه‌ی نرم‌افزاری را در حیطه‌ی یک زبان برنامه‌نویسی خاص توصیف می‌کنند.

گاما و همکاران[1] در کتابی مقدماتی درخصوص الگوهای طراحی [Gam95]، سه نوع الگو را کانون توجه قرار می‌دهند که به ویژه به طراحی شیءگرا مربوط می‌شود: الگوهای ایجادی (creational)، الگوهای ساختاری و الگوهای رفتاری.

در *الگوهای ایجادی*، آن‌چه که کانون توجه قرار می‌گیرد، «ایجاد، ترکیب و نمایش» اشیاست. گاما و همکاران [Gam95] خاطر نشان می‌سازند که الگوهای ایجادی «حاوی دانش مربوط به کلاس‌هایی است که سیستم از آن‌ها استفاده می‌کند.» الگوهای ایجادی سازوکارهایی فراهم می‌آورند که نمونه‌سازی از اشیا را در یک سیستم آسان‌تر کرده «درباره‌ی نوع و تعداد اشیایی که می‌توان در سیستم ایجاد کرد» قیدوبندهای را اِعمال می‌کنند [Maa07].

در *الگوهای ساختاری*، مسائل و راهکارهای مرتبط با چگونگی سازمان‌دهی و انسجام بخشیدن به اشیا برای ایجاد ساختاری بزرگتر، کانون توجه قرار می‌گیرد. در اصل، این الگوها به ایجاد روابطی میان موجودیت‌های موجود در یک سیستم کمک می‌کنند. برای مثال، الگوهای ساختاری‌ای که در آن‌ها مسائل کلاس‌گرا مورد توجه قرار می‌گیرند، ممکن است سازوکارهایی وراثتی فراهم سازند که نتیجه‌ی آن‌ها واسط‌های اثربخش‌تر برای برنامه باشد. الگوهای ساختاری که بر اشیا تأکید دارند، تکنیک‌هایی برای ترکیب اشیا در داخل سایر اشیا یا انسجام بخشیدن به اشیا در ساختاری بزرگتر فراهم می‌سازند.

الگوهای رفتاری به مسائل مرتبط با تقسیم مسؤولیت‌ها میان اشیا و شیوه‌ی تأثیرگرفتن ارتباطات میان اشیا می‌پردازند.

آیا راهی برای گروه‌بندی انواع الگوها وجود دارد؟

۲-۱-۱۲ چارچوب‌ها (Frameworks)

الگوها خود به تنهایی ممکن است برای توسعه‌ی یک طراحی کامل کافی نباشند. در برخی موارد ممکن است برای کار طراحی، نیاز به فراهم آوردن زیرساختار مختص یک پیاده‌سازی باشد که به آن *چارچوب* گفته می‌شود. یعنی می‌توانید یک «*ریزمعماری با قابلیت* استفاده‌ی مجدد انتخاب کنید که رفتار و ساختاری کلی را برای خانواده‌ای از انتزاع‌های نرم‌افزار، همراه با یک حیطه فراهم می‌سازد... که همکاری آن‌ها و کاربرد در دامنه‌ای مفروض را مشخص می‌کند.» [Amb98].

نکته‌ی کلیدی

چارچوب، «ریزمعماری‌ای» با قابلیت استفاده‌ی مجدد است و به‌عنوان بستری عمل می‌کند که سایر الگوهای طراحی از آن قابل اعمال خواهند بود.

چارچوب، الگوی معماری نیست، بلکه اسکلتی است با مجموعه‌ای از نقاط اتصال (یا قلاب‌ها) که به کمک آن‌ها می‌توان این اسکلت را بر یک دامنه‌ی مسأله‌ی خاص تطبیق داد. این نقاط اتصال (plug points) به شما امکان می‌دهند تا کلاس‌ها و قابلیت‌های عملیاتی خاص مسأله را در این اسکلت انسجام بخشید. در حیطه‌ای شیءگرا، چارچوب، مجموعه‌ای از کلاس‌هاست که با یکدیگر همکاری دارند. گاما و همکاران [Gam95] اختلاف میان الگوهای طراحی و چارچوب‌ها را چنین شرح می‌دهند:

۱. *الگوهای طراحی، انتزاعی از چارچوب‌ها هستند.* چارچوب‌ها را می‌توان در قالب کد ارائه داد، ولی فقط مثال‌هایی از الگوها را می‌توان به‌صورت کد ارائه کرد. یکی از نقاط قوت چارچوب‌ها این است که می‌توان آن‌ها را به زبان برنامه‌نویسی نوشت و نه تنها مطالعه کرد بلکه مستقیماً اجرا کرد و دوباره به‌کار گرفت.

[1] گاما و همکاران او [Gam95] را غالباً به عنوان باند چهار نفره می‌شناسند.

اطلاعات

الگوهای ایجادی، ساختاری و رفتاری

گستره‌ی وسیعی از الگوهای طراحی پیشنهاد شده‌اند که در گروه‌های ایجادی، ساختاری و رفتاری می‌گنجند و می‌توان آن‌ها را در وب یافت. در ویکی پیدیا (www.wikipedia.com) نمونه‌های زیر ذکر شده است:

الگوهای ایجادی

- **الگوی کارخانه‌ی انتزاعی:** متمرکز ساختن تصمیم‌گیری در خصوص کارخانه‌ای که از آن باید نمونه‌سازی شود.
- **الگوی روش کارخانه‌ای:** متمرکز ساختن ایجاد یک شیء از نوعی مشخص، با انتخاب یکی از چند پیاده‌سازی.
- **الگوی سازنده:** ساخت اشیای پیچیده را از نمایش آن‌ها جدا می‌کند، به‌طوری که با یک فرایند ساخت می‌توان نمایش‌های مختلفی ایجاد کرد.
- **الگوی نمونه اولیه:** هنگامی استفاده می‌شود که هزینه ذاتی ایجاد یک شیء به شیوه‌ی استاندارد (مثلاً با به‌کارگیری واژه کلیدی «جدید») چنان بالا باشد که نتوان از پس آن بر آمد.
- **الگوی یگانه (singleton):** نمونه‌سازی از یک کلاس برای ایجاد شیء را محدود می‌کند.

الگوهای ساختاری

- **الگوی تطبیق‌دهنده:** یک واسط مربوط کلاسی را به واسط مورد انتظار کلاینت تطبیق می‌دهد.
- **الگوی تجمیعی:** نسخه‌ای از الگوی مرکب همراه با روش‌هایی برای تجمیع فرزندها.
- **الگوی پل:** یک انتزاع را از پیاده‌سازی آن منفک می‌سازد به‌طوری که این دو بتوانند مستقل از هم تغییر کنند.
- **الگوی مرکب:** یک ساختار درختی از اشیا که در آن همه‌ی اشیا، واسط یکسان دارند.
- **الگوی ظرف (container):** اشیایی را برای هدف اساسی نگهداری و اداره‌ی سایر اشیا ایجاد می‌کند.
- **الگوی پروکسی:** کلاسی که به‌عنوان واسط برای شیء دیگر عمل می‌کند.
- **لوله‌ها و فیلترها:** زنجیره‌ای از فرایندها که در آن‌ها خروجی هر فرایند، ورودی فرایند بعدی است.

الگوهای رفتاری

- **الگوهای زنجیره‌ی مسؤولیت‌ها:** اشیای فرمان (command) به وسیله اشیای حاوی پردازش منطقی کنترل می‌شوند یا به سایر اشیا تحویل داده می‌شوند.
- **الگوی فرمان:** اشیای فرمانی که یک کنش و پارامترهای آن را کپسوله می‌کنند.
- **شنودگر رویدادها:** داده‌ها در میان اشیایی توزیع می‌شوند که نام آن‌ها برای دریافت این داده‌ها ثبت شده است.

- **الگوی مفسر:** یک زبان کامپیوتری تخصص‌یافته را برای حل کردن سریع مجموعه‌ای مشخص از مسائل، پیاده‌سازی می‌کند.
- **الگوی تکرارگر:** تکرارگرها در دستیابی ترتیبی به عناصر یک شیء متراکم به‌کار می‌روند، بدون این‌که نمایش واقعی آن را برملا سازند.
- **الگوی میانجی:** برای مجموعه‌ای از واسط‌های موجود در یک زیر سیستم، واسطی یکنواخت فراهم می‌آورد.
- **الگوی بازدیدگر:** راهی برای جداسازی الگوریتم از شیء.
- **الگوی بازدیدگر با سرویس‌دهی یگانه:** پیاده‌سازی بازدیدگری را بهینه می‌کند که تنها یک بار استفاده و سپس حذف می‌شود.
- **الگوی بازدیدگر سلسله مراتبی:** راهی برای بازدید هر گره در یک ساختار سلسله مراتبی مثلاً یک درخت فراهم می‌آورد.

توصیفات جامع درباره هر کدام از این الگوها را می‌توانید در ویکی‌پیدیا بیابید.

۲. *الگوهای طراحی، عناصر معماری کوچکتری از چارچوب‌ها هستند.* یک چارچوب حاوی چند الگوی طراحی است، ولی عکس این مطلب هرگز درست نیست.

۳. *تخصص یافتگی الگوهای طراحی کمتر از چارچوب هاست.* چارچوب‌ها همواره دارای دامنه کاربردی خاص هستند. برعکس، الگوهای طراحی را می‌توان تقریباً در هر برنامه کاربردی مورد استفاده قرار داد. در حالی که داشتن الگوهای طراحی با تخصص یافتگی بیشتر، قطعاً امکان‌پذیر است، حتی چنین الگوهایی نیز معماری یک برنامه کاربردی را تعیین نمی‌کنند.

در اصل، طراح چارچوب استدلال خواهد کرد که یک ریزمعماری با قابلیت استفاده‌ی مجدد، در همه‌ی نرم‌افزارهایی که قرار است در یک دامنه کاربرد محدود توسعه داده شوند، قابل به‌کارگیری است. چارچوب‌ها، برای این که بیشترین اثر را داشته باشند، بدون تغییر به‌کار برده می‌شوند. عناصر طراحی دیگر را می‌توان اضافه کرد، ولی تنها از طریق نقاط اتصال که به طراح امکان می‌دهند تا بر اسکلت چارچوبی، لایه‌ای عضلانی بکشد.

۳-۱-۱۲ توصیف الگوها

طراحی مبتنی بر الگوها با شناسایی الگوهای موجود در برنامه کاربردی‌ای که درصدد ساخت آن هستید، شروع می‌شود، با جستجو برای تعیین این که آیا دیگران به این الگو پرداخته‌اند، ادامه می‌یابد و با به‌کارگیری الگوی مناسب برای مسأله‌ی مورد نظر به پایان می‌رسد. دومین وظیفه از این سه وظیفه، غالباً از همه دشوارتر است. چگونه الگوهایی بیابیم که بر نیازهای ما منطبق باشند؟

پاسخ به این پرسش باید بر ارتباطات مؤثر الگویی که به مسأله می‌پردازد، حیطه‌ای که الگو در آن جای دارد، سیستم نیروهایی که این حیطه را شکل می‌دهد و راهکار پیشنهادی تکیه داشته باشد. برای برقراری ارتباط عاری از ابهام با این اطلاعات، شکل استاندارد یا قالبی برای توصیف الگوها مورد نیاز است. گرچه چند الگوی متفاوت پیشنهاد شده است، تقریباً همه‌ی آن‌ها حاوی زیر مجموعه‌ی عمده‌ای از محتویات پیشنهاد شده توسط گاما و همکاران [Gam95] هستند. یک قالب الگوی ساده در کادر زیر نشان داده شده است:

اطلاعات

قالب الگوی طراحی

نام الگو: توصیف جوهره الگو با نامی کوتاه، ولی شیوا.

مسأله: مسأله‌ای را توصیف می‌کند که الگو به آن می‌پردازد.

انگیزش: مثالی از مسأله فراهم می‌سازد.

حیطه: محیطی را توصیف می‌کند که مسأله در آن جای دارد و دامنه‌ی کاربرد را هم شامل می‌شود.

نیروها: سیستم نیروهای تأثیر گذار بر شیوه‌ی حل مسأله را فهرست می‌کند؛ شامل بحثی درباره‌ی محدودیت‌ها و قیدوبندهایی می‌شود که باید در نظر داشت.

راهکار: توصیف مشروحی از راهکار پیشنهادی برای مسأله فراهم می‌سازد.

هدف: الگو و آن‌چه را که انجام می‌دهد، توصیف می‌کند.

همکاری‌ها: چگونگی مشارکت سایر الگوها در راهکار را شرح می‌دهد.

پیامدها: توازن‌های بالقوه‌ای را که باید هنگام پیاده‌سازی الگو در نظر گرفت و نیز پیامدهای استفاده از آن را شرح می‌دهد.

پیاده‌سازی: مسائل خاصی را مشخص می‌کند که باید هنگام پیاده‌سازی الگو در نظر داشت.

موارد استفاده‌ی شناخته شده: مثال‌هایی از موارد استفاده‌ی واقعی الگوی طراحی را در برنامه‌های کاربردی واقعی به‌دست می‌دهد.

الگوهای مرتبط: الگوهای طراحی مرتبط را به هم ارجاع می‌دهد.

نام الگوهای طراحی باید با احتیاط انتخاب شود. یکی از مسائل فنی در طراحی مبتنی بر الگوها، ناتوانی یافتن الگوها در میان صدها یا هزاران الگو است. انتخاب نامی با معنا به جستجو به دنبال الگوی «درست» کمکی بسیار بزرگ می‌کند.

قالب الگو، ابزاری استاندارد برای توصیف الگوی طراحی فراهم می‌آورد. هر کدام از مدخل‌های قالب، خصوصیاتی از الگوی طراحی را فراهم می‌آورد که می‌توان آن را جستجو کرد (مثلاً از طریق بانک اطلاعاتی) به‌طوری که الگوی مناسب را بتوان به‌دست آورد.

> «الگوها زنده هستند یعنی همیشه باید آن‌ها را خودتان تمام کنید و بر محیط خودتان تطبیق دهید.»
>
> مارتین فاولر

۴-۱-۱۲ مخازن و زبان‌های الگوها (Pattern Languages and Repositories)

هنگامی که از واژه‌ی *زبان* استفاده می‌کنید، نخستین چیزی که به ذهن خطور می‌کند، یا زبانی طبیعی (مثل انگلیسی، اسپانیایی یا چینی) است یا یک زبان برنامه‌نویسی (مثل C++، جاوا). در هر دو مورد، زبان‌ها دارای قالب نحوی یا معناشناختی هستند که در تبادل ایده‌ها یا دستورالعمل‌های رویه‌ای به شیوه‌ای اثربخش به‌کار می‌روند.

هنگامی که واژه‌ی زبان در حیطه‌ی الگوهای طراحی به‌کار می‌رود، معنایی نسبتاً متفاوت به خود می‌گیرد. *زبان الگوها* شامل مجموعه‌ای از الگوها می‌شود که هر یک با به‌کارگیری یک قالب استاندارد شده (بخش ۳-۱-۱۲) توصیف می‌شود و با سایر الگوهای مجموعه ارتباط داده می‌شود تا مسائل

موجود در یک دامنه‌ی کاربرد را با همکاری یکدیگر حل کنند.[1]

در زبان‌های طبیعی، واژه‌ها در قالب یک سری جملات سازمان‌دهی می‌شوند که معنا را بیان می‌کنند. در زبان الگوها، الگوهای طراحی به شیوه‌ای سازمان‌دهی می‌شوند که «روش ساخت‌یافته‌ای برای توصیف طراحی خوب در دامنه‌ای خاص فراهم می‌سازند.»[2]

زبان الگوها از یک لحاظ مشابه با یک جزوه‌ی راهنمای ابر متنی است که برای حل مسأله در دامنه‌ای خاص به‌کار می‌رود. دامنه مسأله‌ی مورد نظر نخست به‌صورت سلسله مراتبی و با شروع از مسائل طراحی مرتبط با دامنه توصیف می‌شود و سپس هر کدام از مسائل به سطوح پایین‌تری از انتزاع پالایش می‌شوند. در حیطه‌ی نرم‌افزار، مسائل طراحی در مقیاس گسترده، ماهیتی معماری دارند و به ساختار کلی برنامه‌ی کاربردی و داده‌ها یا محتویاتی می‌پردازند که به آن سرویس می‌دهند. مسائل معماری به سطوح پایین‌تری پالایش می‌شوند که نتیجه‌ی آن حل مسائل فرعی و همکاری با یکدیگر در سطح مؤلفه‌ها (یا کلاس‌ها) است. زبان الگوها به‌جای یک فهرست ترتیبی از الگوها، مجموعه‌ای از اعضای مرتبط با هم را نشان می‌دهد که در آن کاربر می‌تواند با یک مسأله‌ی طراحی گسترده شروع کند و به‌طرف «پایین برود» تا مسائل مشخص و راهکار آن‌ها را کشف کند.

ده‌ها زبان الگو برای طراحی نرم‌افزار پیشنهاد شده است [Hil08]. در اکثر موارد، الگوهای طراحی که بخشی از زبان الگوها هستند، در مخزنی قرار داده می‌شوند که از طریق وب قابل دستیابی است (مثل [Boo08]، [Cha03]، [HPR02]). این مخزن، نمایه‌ای از همه‌ی الگوهای طراحی فراهم می‌آورد و حاوی پیوندهای فوق رسانه‌ای است که کاربر به کمک آن می‌تواند همکاری‌های میان الگوها را درک کند.

> **اندرز**
>
> اگر نمی‌توانید زبان الگوی مناسبی که مناسب دامنه‌ی مسأله‌ی شما باشد، به دنبال نشانه در مجموعه‌ی دیگری از الگوها بگردید.

> **مرجع وب**
>
> برای فهرستی از زبان‌های الگو، وب‌سایت **c2.com/ppr/titles.html** را ببینید. اطلاعات بیشتری را نیز می‌توانید در وب‌سایت **hillside.net/patterns/** بیابید.

۲-۱۲ طراحی نرم‌افزار بر اساس الگوها

بهترین طراح‌ها در هر زمینه‌ای این توانایی غیر عادی را دارند که الگوهای خاصی را می‌بینند. این الگوها مسأله و الگوهای متناظری را مشخص می‌سازند که می‌توان آن‌ها را برای ایجاد راهکار با هم ترکیب کرد. سازندگان نرم‌افزار در مایکروسافت [Mic04] این نکته را چنین مورد بحث قرار می‌دهند:

> در حالی که طراحی مبتنی بر الگوها مقوله‌ای نسبتاً جدید در زمینه‌ی توسعه‌ی نرم‌افزار به شمار می‌رود، فن‌آوری صنعتی از طراحی مبتنی بر الگوها به مدت چند دهه و شاید حتی چند قرن بهره برده است. کاتالوگ‌هایی از سازوکارها و پیکر بندی‌های استاندارد، عناصر طراحی به‌کاررفته در مهندسی خودروها، هواپیماها، ابزارآلات، و روبات‌ها را فراهم می‌سازند. به‌کارگیری طراحی مبتنی بر الگوها در توسعه‌ی نرم‌افزار، نویدبخش همان فواید صنعتی برای نرم‌افزار است: قابلیت پیش‌بینی، کاهش خطر، و افزایش بهره‌وری.

در سرتاسر فرایند طراحی، باید در جستجوی هر فرصتی باشید تا به‌جای ایجاد الگوهای جدید، الگوهای طراحی موجود را به‌کار گیرید (هنگامی که نیازهای طراحی را برآورده می‌سازند).

۱-۲-۱۲ طراحی مبتنی بر الگوها در حیطه (context)

طراحی مبتنی بر الگوها در خلأ به‌کار برده نمی‌شود. مفاهیم و تکنیک‌های بحث شده برای طراحی

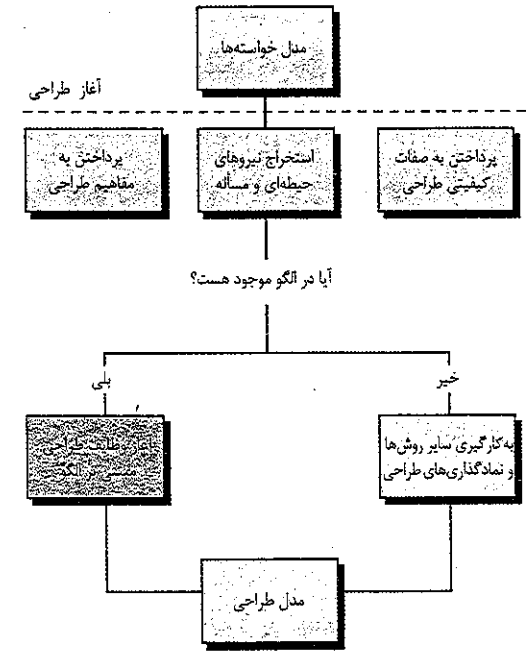
[1] کریستوفر الکساندر در آغاز برای معماری و نقشه کشی شهری، زبان‌های الگوها را پیشنهاد کرد. امروزه، زبان‌های الگوها برای هر چیزی از علوم اجتماعی گرفته تا فرائید مهندسی نرم‌افزار توسعه یافته‌اند.

[2] این توصیف ویکی‌پیدیا را می‌توان در http:/en.wikipedia.org/wiki/pattern-language یافت.

معماری، طراحی در سطح مؤلفه‌ها و طراحی واسط (فصل‌های ۹ تا ۱۱) همگی در ارتباط با رویکردی مبتنی بر الگو به‌کار برده می‌شوند.

در فصل ۸ متذکر شدیم که مجموعه‌ای از صفات دستورالعمل‌های کیفیتی به‌عنوان مبنایی برای همه‌ی تصمیم‌گیری‌های طراحی نرم‌افزار عمل می‌کند. این تصمیم‌گیری‌ها خود از یک مجموعه مفاهیم طراحی بنیادی (مثلاً جداسازی دغدغه‌ها، پالایش مرحله‌ای، استقلال عملیاتی) تأثیر می‌پذیرند که با به‌کارگیری ابتکارات به‌دست‌آمده در طی چند دهه و بهترین کارهایی (مثلاً تکنیک‌ها و نمادگذاری مدل‌سازی) که برای آسان‌تر کردن طراحی و مؤثرترکردن آن به‌عنوان مبنایی برای پیاده‌سازی پیشنهاد شده‌اند، حاصل می‌شوند.

نقش طراحی مبتنی بر الگوها در تمامی این فعالیت‌ها در شکل ۱-۱۲ نشان داده شده است. طراح نرم‌افزار با مدل خواسته‌ها (ضمنی یا صریح) شروع می‌کند که نمایش انتزاعی سیستم را نشان می‌دهد. مدل خواسته‌ها، مجموعه مسائل را توصیف می‌کند، حیطه را تعیین می‌کند و سیستم نیروها را مشخص می‌سازد. به این ترتیب ممکن است طراحی به شیوه‌ای انتزاعی نشان داده شود، ولی مدل خواسته‌ها کار زیادی برای نمایش صریح طراحی نمی‌کند.



شکل ۱-۱۲ طراحی مبتنی بر الگوها در حیطه

هنگامی که کار خود را به‌عنوان طراح شروع کردید، همواره باید صفات کیفیتی را مد نظر داشته باشید. این صفات (مثلاً طراحی باید همه‌ی خواسته‌های ذکر شده در مدل خواسته‌ها را پیاده‌سازی کند) راهی برای ارزیابی کیفیت نرم‌افزار تعیین می‌کنند، ولی در دستیابی به آن کمک چندانی به شما نمی‌دهند. طراحی‌ای که ایجاد می‌کنید، باید مفاهیم طراحی بنیادی بحث شده در فصل ۸ را دربرگیرد.

بنابراین، باید برای تبدیل انتزاع‌های موجود در مدل خواسته‌ها به شکلی مستحکم تر، که همان طراحی نرم‌افزار است، از تکنیک‌های اثبات شده بهره ببرید. برای این منظور، از روش‌ها و ابزارهای مدل‌سازی مربوط به طراحی معماری، طراحی در سطح مؤلفه‌ها و طراحی واسط استفاده خواهید کرد، ولی فقط هنگامی که با یک مسأله، حیطه، و سیستمی از نیروها مواجه می‌شوید که قبلاً راهکاری برای آن‌ها ارائه نشده است. اگر راهکاری از قبل موجود باشد، از آن استفاده کنید و این یعنی رویکرد طراحی مبتنی بر الگوها.

۲-۲-۱۲ اندیشیدن به الگوها

شالووی و ترات [Sha05] در کتابی بسیار خوب در باب طراحی مبتنی بر الگوها درباره «شیوه‌ی نوین تفکر» به هنگام استفاده از الگوها به‌عنوان بخشی از فعالیت طراحی، چنین می‌نویسند:

> می‌بایست ذهن خود را آماده‌ی شیوه‌ی جدید تفکر کنم و هنگامی که چنین کردم، شنیدم که [کریستوفر] الکساندر می‌گفت: «طراحی نرم‌افزار خوب را صرفاً با ترکیب بخش‌های کاری نمی‌توان به‌دست آورد.»

طراحی خوب با پرداختن به حیطه- تصویر بزرگ- آغاز می‌شود. همچنان که حیطه ارزیابی می‌شود، سلسله مراتبی از مسائل را استخراج می‌کنید که باید حل شود. برخی از این مسائل ماهیتی جهانی دارند، در حالی که بقیه به ویژگی‌ها و قابلیت‌های عملیاتی خاص نرم‌افزار می‌پردازند. همه‌ی این مسائل از سیستم نیروهایی تأثیر می‌پذیرند که بر ماهیت راهکار پیشنهادی تأثیر می‌گذارند.

شالووی و ترات [Sha05] رویکرد زیر را پیشنهاد می‌کنند[1] که طراح را قادر می‌سازد تا به الگوها بیندیشد:

۱. اطمینان حاصل کنید که تصویر بزرگ را درک کرده اید- حیطه‌ای که در آن نرم‌افزار قرار خواهد گرفت. مدل خواسته‌ها باید این امکان را فراهم کند.
۲. با بررسی تصویر بزرگ، الگوهایی را استخراج کنید که در آن سطح از انتزاع موجودند.
۳. طراحی خویش را با الگوهای «تصویر بزرگی» آغاز کنید که حیطه یا اسکلتی برای کار طراحی بیشتر وضع کند.
۴. « از حیطه به طرف داخل کار کنید» [Sha05] و در جستجوی الگوهایی در سطوح انتزاع پایین‌تر باشید که در راهکار طراحی سهم دارند.
۵. مراحل ۱ تا ۴ را چندان تکرار کنید که طراحی کامل بالنده شود.
۶. با تطبیق هر الگو بر خصوصیات نرم‌افزاری که قرار است بسازید، طراحی را پالایش کنید.

طراحی مبتنی بر الگو برای مسأله‌ای که باید حل کنم، جالب به نظر می‌رسد؛ از کجا شروع کنم؟

ذکر این نکته حائز اهمیت است که الگوها، موجودیت‌هایی مستقل نیستند. الگوهای طراحی که در سطح بالایی از انتزاع قرار دارند، تأثیری ناگزیر بر شیوه‌ی به‌کارگیری الگوهای دیگر در سطح پایین‌تر انتزاع دارد. به‌علاوه، الگوها غالباً با یکدیگر همکاری دارند. این بدان معناست که وقتی یک الگوی معماری انتخاب می‌کنید، ممکن است تأثیر بسیار خوبی بر الگوهای طراحی انتخاب شده در سطح مؤلفه‌ها بگذارد. به همین منوال، هنگامی که الگوی طراحی واسط مشخصی را انتخاب می‌کنید، گاهی ناگزیر از به‌کارگیری سایر الگوهایی هستید که با آن‌ها همکاری دارند.

[1] بر اساس کار کریستوفر الکساندر [Ale79].

برای روشن شدن مطلب، برنامه تحت وب SafeHomeAssured.com را در نظر بگیرید. اگر تصویر بزرگ را مد نظر قرار دهید، این برنامه‌ی تحت وب باید به چند مسأله‌ی بنیادی بپردازد که از آن جمله‌اند:

- چگونگی فراهم آوردن اطلاعات درباره محصولات و سرویس‌های *SafeHome*
- چگونگی فروش محصولات و سرویس‌های *SafeHome* به مشتریان
- چگونگی برقراری پایش و کنترل اینترنتی یک سیستم امنیتی نصب شده.

هر کدام از این مسائل را می‌توان باز هم به مجموعه‌ای از مسائل کوچکتر پالایش کرد. برای مثال، *چگونگی فروش از طریق اینترنت* یادآور یک الگوی **E-commerce** (تجارت الکترونیک) است که خود این الگو به معنای تعداد زیادی از الگوها در سطوح پایین‌تری از انتزاع است. الگوی **E-commerce** (که احتمالاً الگویی سلسله مراتبی است) به معنای سازوکارهایی برای ایجاد یک حساب کاربری، به نمایش درآوردن محصولات برای فروش، انتخاب محصولات برای خرید و غیره می‌شود. از این رو، اگر به الگوها بیندیشید، مهم است که تعیین کنید آیا برای ایجاد یک حساب کاربری، الگویی وجود دارد یا خیر. اگر **SetUpAccount** به‌عنوان الگویی ماندنی برای حیطه‌ی مسأله، در دسترس است، ممکن است با الگوهای دیگری از قبیل **BuildInputFrom**، **ManageFormsInput** و **ValidateFromEntry** همکاری کند. هر کدام از این الگوها، مسائلی را که باید حل شوند و راهکارهایی را که باید به‌کار برده شوند، ترسیم می‌کند.

۳-۲-۱۲ وظایف طراحی

وظایف طراحی زیر، هنگامی به‌کار برده می‌شوند که از فلسفه طراحی مبتنی بر الگوها استفاده شود:

۱. **مدل خواسته‌ها و توسعه‌ی سلسله مراتبی از مسائل را بررسی کنید.** هر مسأله و مسأله فرعی را با تفکیک مسأله، حیطه و سیستم نیروهای موجود توصیف کنید. از مسائل گسترده (سطح بالای انتزاع) به مسائل فرعی (سطح انتزاع پایین‌تر) کار کنید.

۲. **تعیین کنید آیا زبان الگوی مناسبی برای دامنه مسأله وجود دارد یا خیر.** چنان که در بخش ۴-۱-۱۲ گفته شد، زبان الگو به مسائل مرتبط با یک دامنه‌ی کاربردی خاص می‌پردازد. تیم نرم‌افزاری *SafeHome* به دنبال زبان الگویی است که مشخصاً برای محصولات امنیتی خانگی توسعه یافته‌اند. اگر زبان الگویی در آن سطح مشخص یافت نشود، تیم، مسأله نرم‌افزار *SafeHome* را به یک سری دامنه‌های مسأله‌ی کلی (مثلاً مسائل پایش دستگاه‌های دیجیتال، مسائل واسط کاربر، مسائل مدیریت تصاویر ویدیویی دیجیتال) افراز می‌کند و به دنبال زبان‌های الگوی مناسب می‌گردد.

۳. **با شروع از یک مسأله‌ی گسترده، تعیین کنید که آیا یک یا چند الگوی معماری برای آن در دسترس هست یا خیر.** اگر یک الگوی معماری در دسترس است، حتماً همه‌ی الگوهای همکار را بررسی کنید. اگر این الگو مناسب است، راهکار طراحی پیشنهاد شده را تطبیق دهید و یک عنصر مدل طراحی بسازید که آن را به طرزی مناسب نمایش دهد. چنان که در بخش ۲-۲-۱۲ گفته شد، یک مسأله‌ی گسترده برای برنامه تحت وب **SafeHomeAssured.com**، با الگوی **E-commerce** حل می‌شود. این الگو معماری مشخصی برای پرداختن به خواسته‌های تجارت الکترونیکی پیشنهاد می‌کند.

وظایف لازم برای طراحی مبتنی بر الگو کدام‌اند؟

۴. **با به‌کارگیری همکاری‌های فراهم آمده برای الگوی معماری، مسائل سطح مؤلفه‌ای یا زیرسیستم را بررسی کنید و به دنبال الگوهایی مناسب برای آن‌ها بگردید.** ممکن است جستجو در سایر مخازن الگوها و نیز فهرست الگوهای متناظر با راهکار معماری، ضرورت پیدا کند. اگر الگوی مناسب یافت شد، راهکار طراحی پیشنهادی را تطبیق دهید و یک عنصر مدل طراحی بسازید که به طرزی مناسب آن را نمایش دهد. حتماً مرحله ۷ را انجام دهید.

۵. **مراحل ۲ تا ۵ را چندان تکرار کنید که همه‌ی مسائل گسترده در نظر گرفته شوند.** منظور این است که با تصویر بزرگ شروع کنید و بکوشید با افزودن بر سطوح جزئیات، مسائل را حل کنید.

۶. **اگر مسائل طراحی واسط کاربر جداسازی شده‌اند (تقریباً همواره چنین است)، در میان مخازن الگوهای طراحی واسط کاربر، که تعداد زیادی از آن‌ها وجود دارد، به‌دنبال الگوهای مناسب بگردید.** به شیوه‌ای مشابه با مراحل ۳، ۴ و ۵ پیش بروید.

۷. **اگر یک زبان الگو و/ یا مخزن الگوها یا تنها یک الگوی امید بخش به نظر رسید، مسأله‌ای را که قرار است حل شود با الگو(های) ارائه شده‌ی موجود مقایسه کنید.** حتماً حیطه و نیروها را بررسی کنید و اطمینان حاصل کنید که الگو واقعاً راهکاری فراهم می‌آورد که مناسب مسأله است.

۸. **طراحی را به موازاتی که از الگوها به‌دست می‌آید، با به‌کارگیری ملاک‌های کیفیتی به‌عنوان راهنما، پالایش کنید.**

گرچه این رویکرد طراحی از بالا به پایین انجام می‌شود، در واقع راهکارهای طراحی قدری پیچیده‌ترند. گیلز [Gil08] در این باره چنین می‌گوید:

> الگوهای طراحی در مهندسی نرم‌افزار، به منظور استفاده به شیوه‌ای استنباطی و عقلایی‌تر پدید می‌آیند. اگر مسأله یا خواسته‌ی کلی X را دارید و الگوی طراحی Y، مسأله‌ی X را حل می‌کند، پس Y را به‌کار ببرید. اکنون هنگامی که در فرایند خودم تعمق می‌کنم- و دلیل دارم که باور کنم تنها نیستم- این را می‌فهمم که بیشتر ماهیتی ارگانیک دارد، بیشتر استقرایی است تا استنباطی و بیشتر از پایین به بالاست تا بالا به پایین.
>
> آشکار است که باید موازنه‌ای برقرار شود. هنگامی که پروژه‌ای در فاز اولیه‌ی شروع است و تلاش دارم از خواسته‌های انتزاعی به یک راهکار طراحی مستحکم جهش کنم، غالباً نوعی «جستجوی عرضی» (breadth-first) انجام می‌دهم... الگوهای طراحی را مفید یافته‌ام زیرا به من این امکان را می‌دهند که به سرعت، مسأله‌ی طراحی را به طرزی مستحکم حل کنم.

به‌علاوه، رویکرد مبتنی بر الگوها را باید همراه با سایر مفاهیم و تکنیک‌های طراحی به‌کار برد.

۴-۲-۱۲ ساخت جداول سازمان‌دهی الگوها

به موازاتی که طراحی مبتنی بر الگوها پیش می‌رود، ممکن است در سازمان‌دهی و گروه‌بندی الگوهای کاندیدا از چند مخزن و زبان الگو، با مشکل مواجه شوید. مایکرو سافت [Mic04] برای کمک به سازمان‌دهی الگوهایی که ارزیابی کرده اید، ایجاد یک *جدول سازمان‌دهی الگوها* را پیشنهاد می‌کند که صورت کلی آن در شکل ۲-۱۲ نشان داده شده است.

اندرز

درایه‌های موجود در جدول را با ذکر قابلیت استفاده‌ی مجدد الگو می‌توان کامل‌تر کرد.

				داده‌ها/محتویات
			نام الگو(ها)	سطح مؤلفه
		نام الگو(ها)		شرح مسأله
	نام الگو(ها)			معماری
		نام الگو(ها)		شرح مسأله
نام الگو(ها)				سطح مؤلفه
				شرح مسأله
				واسط کاربر
		نام الگو(ها)		شرح مسأله
				سطح مؤلفه
	نام الگو(ها)			شرح مسأله
		نام الگو(ها)		واسط کاربر
				شرح مسأله
	نام الگو(ها)			سطح مؤلفه
		نام الگو(ها)		شرح مسأله

شکل ۲-۱۲ جدول سازمان‌دهی الگوها.

این جدول سازمان‌دهی الگوها را می‌توان به‌صورت یک مدل صفحه گسترده و با به‌کارگیری نمونه‌ی شکل ۲-۱۲ پیاده‌سازی کرد. فهرست خلاصه‌ای از صورت مسائل، که براساس داده‌ها/محتوا، معماری، سطح مؤلفه‌ها و واسط کاربر سازمان‌دهی می‌شوند، در ستون سمت چپی (هاشورخورده) نشان داده می‌شود. چهار نوع الگو- بانک اطلاعاتی، کاربرد، پیاده‌سازی و زیرساخت- در بالای جدول ذکر می‌شوند. نام الگوهای کاندیدا در سلول‌های این جدول نوشته می‌شود.

برای فراهم ساختن مدخل‌های جدول سازمان دهی، باید در زبان‌های الگو و مخازن الگوها به‌دنبال الگوهایی بگردید که به‌صورت مسأله‌ی خاص می‌پردازند. هنگامی که یک یا چند الگوی کاندیدا پیدا شد، در ردیف متناظر با صورت مسأله و ستون متناظر با نوع الگو وارد می‌شود. نام الگو به‌صورت یک پیوند متهی به URL آدرس وب وارد می‌شود که حاوی توصیف کاملی از الگو است.

۱۲-۲-۵ اشتباهات متداول در طراحی

طراحی مبتنی بر الگوها می‌تواند از شما یک طراح نرم‌افزار بهتر بسازد، ولی اکسیر نیست. همانند همه‌ی روش‌های طراحی، باید با اصل اول شروع کنید: تأکید ورزیدن بر مبانی کیفیت نرم‌افزار و حصول اطمینان از این که طراحی، واقعاً، نیازهای بیان شده در مدل خواسته‌ها را پوشش می‌دهد.

چند اشتباه رایج در به‌کارگیری طراحی مبتنی بر الگوها رخ می‌دهد. در برخی موارد، زمان کافی صرف شناخت مسأله‌ی زیربنایی و خیطه و نیروهای آن صرف نمی‌کنید و در نتیجه، الگویی را بر می‌گزینید که درست به‌نظر می‌رسد، ولی برای راهکار مورد نیاز نامناسب است. هنگامی که الگوی نادرست انتخاب شود، از دیدن خطا سر باز می‌زنید و می‌کوشید که به زور الگو را مطابقت دهید. در موارد دیگر، مسأله دارای نیروهایی است که در الگوی انتخابی شما در نظر گرفته نشده است و در نتیجه، تطبیق ضعیف یا نادرستی خواهید داشت. یک الگو گاهی اوقات، بیش از حد لفظی به‌کار برده می‌شود و تطبیق‌های لازم برای فضای مسأله‌ی شما پیاده‌سازی نشده‌اند.

اندرز

یک الگو را به روز به کار نبرید حتی اگر مناسب مسأله‌ی مورد نظر باشد. اگر خیطه و نیروها مناسب نبودند، به‌دنبال الگوی دیگر باشید.

آیا از این اشتباهات می‌توان پرهیز کرد؟ در اکثر موارد، پاسخ «مثبت» است. هر طراح خوب به دنبال ایده دوم است و از نقد و بازبینی کارهایش استقبال می‌کند. تکنیک‌های بازبینی بحث شده در فصل ۱۵ می‌توانند به شما کمک کنند تا اطمینان حاصل کنید که طراحی مبتنی بر الگوهایی که توسعه داده‌اید، برای مسأله‌ی نرم‌افزاری که باید حل شود، به راهکاری با کیفیت بالا منجر گردد.

۱۲-۳ الگوهای معماری

اگر معماری تصمیم بگیرد که یک نوع خاص از عمارت را بسازد، تنها یک سبک معماری وجود دارد که می‌تواند به‌کار ببرد. جزئیات سبک (مثلاً تعداد شومینه، نمای خانه، محل قرار گرفتن درها و پنجره‌ها) ممکن است به‌طور چشمگیر تغییر کند، ولی هنگامی که تصمیم‌گیری درباره‌ی معماری کلی ساختمان به عمل آمد، سبک معماری در طراحی اعمال خواهد شد.[1]

الگوهای معماری قدری تفاوت دارند. برای مثال، در هر خانه (و در هر سبک معماری برای خانه‌ها) از یک الگوی **Kitchen** استفاده می‌شود. الگوی **Kitchen** و الگوهایی که با آن همکاری دارند، به مسائل مرتبط با نگهداری و تهیه غذا، ابزارهای لازم برای انجام این وظایف و قواعد مربوط به قرار دادن این ابزارها نسبت به جریان کاری در اتاق می‌پردازند. به‌علاوه، این الگو ممکن است به مسائل مربوط به رف‌ها، نور پردازی، کلیدها و پریزها، یک جزیره مرکزی، کف‌پوش و غیره نیز بپردازد. بدیهی است که بیش از یک طراحی برای آشپزخانه وجود دارد که غالباً حیطه و سیستم نیروها را دیکته می‌کند. ولی هر طراحی را می‌توان در حیطه‌ی «راهکار» پیشنهادشده توسط الگوی **Kitchen** تصور کرد.

چنان که قبلاً گفتیم، الگوهای معماری برای نرم‌افزار، رویکردی ویژه برای خصوصیتی از سیستم تعریف می‌کنند. بوش [Bos00] و بوچ [Boo08] چند دامنه‌ی الگوی معماری تعریف می‌کنند. چند مثال نمونه در پاراگراف‌های بعدی ارائه خواهد شد:

کنترل دستیابی. شرایط بسیاری وجود دارد که در آن‌ها دستیابی به داده‌ها، ویژگی‌ها و قابلیت‌هایی که برنامه تحویل می‌دهد به کاربران نهایی مشخص محدود می‌شود. از دیدگاه معماری، دستیابی به بخشی از معماری نرم‌افزار باید به شدت کنترل شود.

همروندی (Concurrency). بسیاری از برنامه‌های کاربردی باید وظایف چندگانه را به شیوه‌ای عهده‌دار شوند که موازی‌کاری را شبیه‌سازی کنند (یعنی زمانی که چند وظیفه یا مؤلفه‌ی «موازی» تنها توسط یک پردازنده مدیریت شوند). برنامه کاربردی به چند روش متفاوت می‌تواند همروندی را اداره کند که هر کدام را می‌توان با یک الگوی معماری متفاوت نشان داد. برای مثال، یک روش، استفاده از الگوی **OperatingSystemProcessManagement** است که ویژگی‌های توکاری از سیستم عامل را فراهم می‌آورند که همکاری همزمان مولفه‌ها را امکان‌پذیر می‌سازند. این الگو همچنین شامل قابلیت‌های عملیاتی سیستم عامل می‌شود که برقراری ارتباط میان فرایندها، زمان‌بندی و سایر توانایی‌های لازم برای دستیابی به همروندی را مدیریت می‌کنند. یک روش دیگر، ممکن است تعریف

نکته‌ی کلیدی

هر معماری نرم‌افزار ممکن است چند الگوی معماری داشته باشد که به مسائل ناهمگون از قبیل همروندی، مانیتورینگ و توزیع مربوط شوند.

چند نمونه از دامنه‌های الگوی معماری را نام ببرید.

[1] این بدان معناست که یک سرسرا و کریدور مرکزی وجود خواهد داشت. اتاق‌ها در طرف چپ و راست راهرو قرار داده می‌شوند، خانه دو (یا چند) طبقه دارد، اتاق خواب‌ها در طبقه بالا خواهند بود و غیره. این «قواعد» هنگامی اعمال می‌شوند که تصمیم بر استفاده از این سبک خاص گرفته شود.

واحدی برای زمان‌بندی وظایف در سطح برنامه کاربردی باشد. الگوی **TaskScheduler** حاوی مجموعه‌ای از اشیای فعال است که حاوی عملیات *( )tick* هستند [Bos00]. این واحد زمان‌بندی، به‌طور ادواری عملیات *( )tick* را برای هر شیء فراخوانی می‌کند که سپس وظایفی را که باید قبل از بازگرداندن کنترل به واحد زمان‌بندی انجام دهد، اجرا می‌کند و سپس عملیات *( )tick* را برای شیء همروند بعدی فراخوانی می‌کند.

توزیع (Distribution). مسأله‌ی توزیع به شیوه‌ی برقراری ارتباط سیستم‌ها یا مؤلفه‌های درون سیستم‌ها با یکدیگر در محیطی توزیع شده می‌پردازد. دو مسأله‌ی فرعی در نظر گرفته می‌شود: (۱) شیوه‌ی اتصال موجودیت‌ها به یکدیگر و (۲) ماهیت ارتباطی که برقرار می‌شود. متداول‌ترین الگوی معماری وضع شده برای پرداختن به مسأله‌ی توزیع، الگوی **Broker** است. میانجی به‌عنوان «واسطه‌ای» میان مؤلفه‌ی کلاینت و مؤلفه‌ی سرور عمل می‌کند. کلاینت، پیامی به میانجی ارسال می‌کند (که حاوی همه‌ی اطلاعات مناسب برای برقرارسازی ارتباط است) و میانجی، اتصال را کامل می‌کند.

ماندگاری (Persistence). داده‌ها در صورتی ماندگار خواهند بود که پس از اجرای فرایند ایجاد کننده‌ی آن‌ها باقی بمانند. داده‌های ماندگار در یک بانک اطلاعاتی یا فایل ذخیره می‌شوند و بعداً ممکن است توسط فرایندهای دیگری خوانده یا اصلاح شوند. در محیط‌های شیءگرا، ایده‌ی اشیای ماندگار، مفهوم ماندگاری را قدری وسعت می‌بخشد. مقادیر همه‌ی صفات اشیاء حالت کلی شیء و سایر اطلاعات مکمل برای بازیابی و استفاده‌ی بعدی ذخیره می‌شود. به‌طور کلی، دو الگوی معماری در دستیابی به ماندگاری به‌کار می‌رود- یک الگوی **DatabaseManagementSystem** که توانایی ذخیره‌سازی و بازیابی DBMS را در معماری برنامه به‌کار می‌گیرد یا الگوی **ApplicationLevelPersistence** که ویژگی‌های ماندگاری را در معماری برنامه قرار می‌دهد (مثل واژه پردازی که ساختار خاص خودش را برای مستندات مدیریت می‌کند).

پیش از این که بتوان هر کدام از نمونه الگوهای معماری ذکر شده در پاراگراف‌های قبل را انتخاب کرد، باید مناسب بودن آن را برای برنامه و سبک معماری کلی و نیز حیطه و سیستم نیروهایی که آن را مشخص می‌کند، مورد سنجش قرار داد.

۱۲-۴ الگوهای طراحی در سطح مؤلفه‌ها

الگوهای طراحی در سطح مؤلفه‌ها راهکارهایی اثبات شده در اختیار شما قرار می‌دهند که به یک یا چند مسأله‌ی فرعی استخراج شده از مدل خواسته‌ها می‌پردازند. در بسیاری موارد، الگوهای طراحی از این نوع یک عنصر عملیاتی از سیستم را کانون توجه قرار می‌دهند. برای مثال، برنامه تحت وب **SafeHomeAssured.com** باید به این مسأله فرعی طراحی بپردازد: *چگونه می‌توانیم مشخصات محصول و اطلاعات مرتبط را برای هر دستگاه SafeHome به‌دست آوریم؟*

اکنون با بیان زیر مسأله‌ای که قرار است حل شود، باید حیطه و سیستم نیروهای تأثیرگذار بر راهکار را در نظر بگیرید. با بررسی پرونده کاربرد مناسب در مدل خواسته‌ها در می‌یابید که مشتری برای یک دستگاه *SafeHome* (مثلاً حس‌گر یا دوربین) برای اهداف اطلاعاتی از مشخصات استفاده می‌کند. به هر حال، اطلاعات مرتبط با مشخصات (مثل قیمت گذاری) را می‌توان هنگام انتخاب قابلیت عملیاتی تجارت الکترونیکی به‌کار برد.

اطلاعات

مخازن الگوهای طراحی

برای الگوهای طراحی منابع فراوانی در وب در دسترس است. برخی از این الگوها را می‌توان از زبان‌های الگوی منتشر شده به‌دست آورد در حالی که عده‌ای دیگر به‌عنوان بخشی از مخازن الگوها قابل دستیابی‌اند. نگاه کردن به منابع زیر در وب مفید واقع می‌شود:

Hillside.net (**http://hillside.net/patterns/**)

یکی از جامع‌ترین مجموعه الگوها و زبان‌های الگو در وب .

Pattern Index Repository (**http://c2.com/ppr/index.html**)

حاوی آدرس انواع منابع الگوها و مجموعه الگوها.

Pattern Index (**http://c2.cpm/cgi/wiki?PatternIndex**)

«مجموعه ای گلچین از الگوها»

Booch Architecture Handbook (**www.booch.com/architecture/index.html**)

فهرستی از صدها الگوی طراحی معماری و مؤلفه‌ها.

مجموعه الگوهای واسط کاربر

UI/HCI Patterns (**www.hcipatterns.org/patterns.html**)

Jeniffer Tidwell's UI Patterns (**www.time-trippers.com/uipatterns.html**)

Mobile UI Design Patterns
(**http://patterns.littlespringsdesign.com/wikka.php?wakka=Mobile**)

الگوها

Patterns Language for *UI Design*
(**www.maplefish.com/todd/papers/Experiences.html**)

Interaction Design Library for Games
(**www.eelke.com/research/usability.html**)

UI Design Patterns (**www.cs.helsinki.fi/salaakso/usability/patterns/**)

الگوهای طراحی تخصص‌یافته

Aircraft Avionics (**www.g.oswego.edu/dl/acs/acs/acs.html**)

Business Information Systems (**www.objectarchitects.de/arcus/cookbook/**)

Distributed Processing (**www.cs.wustl.edu/~schmidt/**)

IBM Patterns for e-Business (**www128.ibm.com/developerworks/patterns**)

Yahoo! Design Pattern Library (**http://developers.yahoo.com/ypatterns**)

WebPatterns.org (**http://webpatterns.org/**)

راهکار این زیر مسأله شامل جستجو می‌شود. چون جستجو، مسأله‌ای بسیار متداول است، تعجبی ندارد که الگوهای بسیاری در خصوص جستجو وجود داشته باشد. با نگاه به چند مخزن، الگوهای زیر را می‌یابید که همراه هر کدام مسأله‌ی مربوط نیز ذکر می‌شود:

AdvancedSearch. کاربران باید یک آیتم مشخص را در مجموعه‌ی بزرگی از آیتم‌ها بیابند.

HelpWizard. کاربران به کمک درباره‌ی یک مبحث معین و مرتبط با وب‌سایت نیاز دارند، یا نیاز به یافتن صفحه‌ای خاص در سایت دارند.

SearchArea. کاربران باید یک صفحه را بیابند.

SearchTips. کاربران باید بدانند چگونه موتور جستجو را کنترل کنند.

SearchResults. کاربران باید فهرستی از نتایج جستجو را پردازش کنند.

SearchBox. کاربران باید یک آیتم یا اطلاعات مشخصی را بیابند.

برای **SafeHomeAssured.com** تعداد محصولات، خیلی زیاد نیست و هر کدام یک گروه‌بندی نسبتاً ساده دارد. لذا **AdvancedSearch** و **HelpWizard** احتمالاً ضروری نخواهند بود. به‌طور مشابه، جستجو به قدر کافی ساده نیست که **SearchTips** را الزامی کند. ولی توصیف **SearchBox** (تا حدی) به‌صورت زیر ارائه می‌شود:

Search Box

(بر گرفته از www.welie.com/patterns/show pattern.php?patternID=searchbox)

مسأله: کاربران باید یک آیتم یا اطلاعات مشخص را بیابند.

انگیزش: هر وضعیتی که در آن، جستجوی یک واژه‌ی کلیدی در میان مجموعه‌ای از اشیای محتوایی اعمال گردد که در قالب صفحات وب سازمان‌دهی شده‌اند.

حیطه: کاربر مایل است به‌جای گشت‌وگذار برای به‌دست آوردن اطلاعات یا محتویات، یک جستجوی مستقیم در میان محتویات موجود در چند صفحه‌ی وب انجام دهد. هر وب‌سایت دارای ابزارهای گشت‌وگذار ابتدایی است. کاربر ممکن است بخواهد آیتمی را در یک گروه جستجو کند. کاربر ممکن است بخواهد درخواستی را بیشتر مشخص کند.

نیروها: وب‌سایت از قبل دارای گشت‌وگذار اولیه است. کاربران ممکن است بخواهند آیتمی را در یک گروه جستجو کنند. کاربران ممکن است بخواهند درخواستی را با استفاده از عملگرهای بولی، بیشتر مشخص کنند.

راهکار: ارائه‌ی قابلیت عملیاتی جستجو که شامل یک برچسب جستجو، فیلدی برای واژه‌ی کلیدی، یک فیلتر (در صورت لزوم) و یک دکمه‌ی «شروع» می‌شود. همچنین فراهم ساختن **SearchTips** و مثال‌هایی در یک صفحه‌ی جداگانه. پیوندی منتهی به این صفحه در کنار قابلیت عملیاتی جستجو قرار داده می‌شود. کادر متنی به قدر کافی بزرگ هست که سه درخواست کاربری (معمولاً حدود بیست کاراکتر) را در خود جای دهد. اگر تعداد فیلترها بیش از دو باشد، برای انتخاب فیلترها از کادر کامبو استفاده می‌شود و در غیر این صورت، یک دکمه‌ی رادیویی به‌کار می‌رود.

نتایج جستجو در صفحه‌ی جدیدی ارائه می‌شود که با عنوانی نظیر «نتایج جستجو» مشخص می‌شود. قابلیت جستجوی واژه‌های کلیدی وارد شده در آن در بالای صفحه تکرار می‌شود تا کاربران بدانند واژه‌های کلیدی چه بوده‌اند.

توصیف این الگو با سایر مدخل‌ها در بخش ۳-۱-۱۲ ادامه می‌یابد.

این الگو ادامه می‌یابد تا چگونگی ارزیابی نتایج، ارائه‌ی آن‌ها، همخوانی نتایج و غیره را توصیف کند. بر این اساس، تیم **SafeHomeAssured.com** می‌تواند مؤلفه‌های لازم برای پیاده‌سازی جستجو را طراحی کند یا (به احتمال بیشتر) مؤلفه‌های موجود یا قابل استفاده‌ی مجدد را به‌دست آورد.

SafeHome

به‌کارگیری الگوها

صحنه: بحث غیر رسمی طی طراحی یک گام نرم‌افزار که کنترل حس‌گرها را از طریق اینترنت برای **SafeHomeAssured.com** پیاده‌سازی می‌کند.

نقش‌آفرینان: جیمی (مسؤول طراحی) و وینود (معمار ارشد سیستم **SafeHomeAssured.com**).

گفتگو:

وینود: طراحی واسط کنترل دوربین‌ها چطور پیش می‌رود؟

جیمی: خیلی بد نیست. من قسمت اعظم قابلیت اتصال به حس‌گرهای واقعی را بدون مشکل جدی طراحی کردم. به‌علاوه، درباره واسط کاربران برای حرکت دادن، تغییر زاویه و درشت نمایی دوربین‌ها از طریق صفحه‌ی وب فکر کرده‌ام، ولی مطمئن نیستم که راه درست را رفته باشم.

وینود: به کجا رسیدی؟

جیمی: خب، خواسته‌ها این است که کنترل دوربین باید بسیار تعاملی باشد- به محض این که کاربر کنترل را حرکت می‌دهد، دوربین باید هر چه سریع‌تر حرکت کند. بنابراین، داشتم فکر می‌کردم یک سری دکمه داشته باشیم مثل دکمه‌های دوربین واقعی که وقتی کاربر روی آن‌ها کلیک می‌کند، دوربین را کنترل کنند.

وینود: بله، جواب می‌دهد، ولی مطمئن نیستم کار درستی باشد- برای هر کلیک کنترلی باید منتظر بمانی تا یک ارتباط کلاینت- سرور کامل برقرار بشود و بنابراین، حس خوبی از بازخورد سریع به‌دست نمی‌آید.

جیمی: من هم همین فکر را می‌کردم- و به همین خاطر هم خیلی از این روش راضی نبودم، ولی نمی‌دانم چه کار دیگری می‌شود کرد.

وینود: خب چرا از الگوی **InteractiveDeviceControl** استفاده نمی‌کنی؟

جیمی: حالا چی هست؟ من که چیزی درباره‌اش نشنیده‌ام.

وینود: اساساً الگویی برای مسأله‌ای است که الان گفتی. راهکاری که پیشنهاد می‌کند، اساساً ایجاد یک اتصال کنترلی با سرور و دستگاه است که از طریق آن فرمان‌های کنترلی را می‌شود ارسال کرد. این طوری دیگر لازم نیست درخواست‌های HTTP را عادی ارسال کنی. و الگو حتی نشان می‌دهد که چطوری می‌توانی این را با یک تکنیک ساده‌ی AJAX پیاده‌سازی کنی. چند جاوا اسکریپت ساده‌ی کلاینت- سرور داری که مستقیماً با سرور ارتباط برقرار می‌کند و به محض این که کاربر عملی انجام داد، فرمان‌ها را ارسال می‌کنند.

جیمی: عالی است! این دقیقاً همان چیزی است که برای حل این مشکل لازم داشتم. کجا می‌توانم پیدایش کنم؟

وینود: از یک مخزن اینترنتی در دسترس است. این هم آدرس URL آن.

جیمی: الآن نگاه می‌کنم.

وینود: بله. اما یادت باشد که فیلد پیامدهای الگو را هم نگاه کنی. یادم هست که یک چیزهایی درباره مسائل امنیتی در آن نوشته شده باشد. فکر کنم دلیل‌اش این باشد که یک کانال کنترلی جداگانه ایجاد می‌کنی و بنابراین، سازوکارهای امنیتی وب را دور می‌زنی.

جیمی: نکته‌ی جالبی بود. احتمالاً من به این فکر نمی‌کردم. ممنون.

۵-۱۲ الگوهای طراحی واسط کاربر

صدها الگو برای واسط کاربر (UI) طی سال‌های اخیر پیشنهاد شده‌اند که اکثر آن‌ها در یکی از ده گروه الگوهای توصیف شده توسط تیدول [Tid02] و فان، ولی [Wel 01] قرار می‌گیرند (و با مثال‌های نمونه بحث شده‌اند[1]):

واسط کاربر کامل. راهنمایی برای طراحی ساختار سطح بالا، و گشت‌وگذار در سرتاسر واسط فراهم می‌آورد.

الگو: TopLevelNavigation

شرح مختصر: هنگامی استفاده می‌شود که یک سایت یا برنامه کاربردی چند قابلیت عملیاتی عمده را پیاده‌سازی می‌کند. یک منوی سطح بالا فراهم می‌آورد که غالباً با لوگو یا آرمی همراه است که گشت‌وگذار مستقیم در هر کدام از قابلیت‌های عمده را فراهم می‌سازد.

جزئیات. قابلیت‌های عملیاتی عمده (که عموماً به چهار تا هفت مورد محدود می‌شوند) در بالای صفحه نمایش به‌صورت افقی فهرست می‌شوند (فرمت ستونی عمودی نیز امکان‌پذیر است). هر نام، پیوندی منتهی به یک قابلیت عملیاتی یا منبع اطلاعات مناسب دارد. غالباً با الگوی **BreadCrumbs** استفاده می‌شود که بعداً بحث خواهد شد.

عناصر گشت‌وگذار: هر نام قابلیت عملیاتی/ محتویات، نشان‌گر پیوندی منتهی به قابلیت عملیاتی یا محتویات مناسب است.

چیدمان صفحه. به سازمان‌دهی کلی صفحات وب (برای وب‌سایت‌ها) یا صفحات نمایش متمایز (برای برنامه‌های تعاملی) می‌پردازد.

الگو: CardStack

شرح مختصر: هنگامی استفاده می‌شود که چند گروه از محتویات یا قابلیت‌های عملیاتی خاص مرتبط با یک ویژگی یا قابلیت عملیاتی، باید به طور تصادفی انتخاب شوند. ظاهری شبیه یک پشته از کارت‌های برگه‌دار (Tabbed cards) را ایجاد می‌کند که با کلیک کردن روی برگه‌ی هر صفحه، محتویات آن صفحه به نمایش در می‌آید.

جزئیات: کارت‌های برگه‌دار، استعاره‌ای شناخته‌شده هستند و کاربر به آسانی می‌تواند آن‌ها را دستکاری کند. فرمت هر کارت برگه‌دار ممکن است قدری متفاوت باشد. برخی ممکن است نیاز به ورودی داشته باشند و به دکمه‌ها یا سایر سازوکارهای گشت‌وگذار آراسته شده باشند؛ برخی دیگر ممکن است اطلاعاتی باشند. ممکن است با سایر الگوها از قبیل **DropDownList**، **Fill-in-the-Blanks** و غیره ترکیب شوند.

عناصر گشت‌وگذار: یک کلیک ماوس روی برگه باعث می‌شود تا کارت مناسب ظاهر گردد. ممکن است ویژگی‌های گشت‌وگذاری در هر کارت نیز موجود باشد، ولی به‌طور کلی، این‌ها باید باعث شروع به‌کار یک قابلیت عملیاتی مرتبط با داده‌های کارت شوند، نه این که پیوند واقعی به یک صفحه دیگر ظاهر گردد.

[1] در این‌جا از یک قالب الگوی مختصر استفاده می‌شود. توصیف کامل این الگوها (به همراه ده‌ها الگوی دیگر) را می‌توان در [Tid02] و [Wel01] دید.

فرم‌ها و ورودی. به انواع تکنیک‌های طراحی مربوط به تکمیل ورودی از طریق فرم‌ها می‌پردازد.

الگو: Fill-in-the-Blanks

شرح مختصر: امکان وارد کردن همه‌ی داده‌های حرفی- عددی در یک «کادر متنی» را فراهم می‌آورد.

جزئیات: داده‌ها را می‌توان در یک کادر متنی وارد کرد. به‌طور کلی، داده‌ها اعتبارسنجی شده پس از انتخاب یک شاخص گرافیکی یا متنی (مثلاً دکمه‌ی «go»، «submit» یا «next») پردازش می‌شوند. در بسیاری موارد، این الگو را می‌توان با یک منوی باز شونده ترکیب کرد (مثلاً <drop down list> FOR <fill-in-the-blanks>).

عناصر گشت‌وگذار: یک شاخص متنی یا گرافیکی که اعتبارسنجی و پردازش را آغاز می‌کند.

جدول‌ها. راهنمایی طراحی برای ایجاد و دستکاری داده‌های جدول‌بندی شده از همه نوع را فراهم می‌سازد.

الگو: SortableTable

شرح مختصر: فهرستی بلندبالا از رکوردها را نشان می‌دهد که می‌توان با انتخاب عنوان هر سرستون از جدول، آن را مرتب کرد.

جزئیات: هر ردیف جدول، نشان‌گر یک رکورد کامل است. هر ستون نشان‌گر یک فیلد از رکورد است. عنوان ستون در واقع یک دکمه‌ی قابل انتخاب است که با هر بار کلیک کردن روی آن از مرتب‌سازی صعودی به نزولی یا بالعکس تغییر وضعیت می‌دهد و فیلد مرتبط با ستون را برای همه‌ی رکوردها مرتب می‌کند. اندازه جدول عموماً قابل تغییر است و اگر تعداد رکوردها بزرگتر از فضای پنجره‌ی در دسترس باشد، می‌توان در آن پنجره حرکت کرد.

عناصر گشت‌وگذار. هر عنوان ستون، آغازگر مرتب‌سازی روی همه‌ی رکوردهاست. هیچ گونه گشت‌وگذار دیگری فراهم نمی‌آید هر چند که در برخی موارد، هر رکورد ممکن است خود حاوی پیوندهایی به سایر محتویات یا قابلیت‌های عملیاتی باشد.

دستکاری مستقیم داده‌ها. به ویرایش، اصلاح و تبدیل داده‌ها می‌پردازد.

الگو: BreadCrumbs

شرح مختصر: هنگامی که کاربر با سلسله مراتب پیچیده‌ای از صفحات وب یا صفحات نمایش کار می‌کند، یک مسیر کامل برای گشت‌وگذار ترسیم می‌کند.

جزئیات: به هر صفحه‌ی وب یا صفحه نمایش، یک شناسه منحصر به فرد داده می‌شود. مسیر گشت‌وگذار به موقعیت فعلی در یک مکان از پیش تعیین شده برای هر صفحه نمایش مشخص می‌گردد. این مسیر به شکل زیر است:

صفحه فعلی < صفحه خاص < صفحه مبحث فرعی < صفحه مبحث اصلی < خانه

عناصر گشت‌وگذار. هر کدام از مدخل‌های موجود در صفحه نمایش خرده نان[1] را می‌توان به‌عنوان اشاره گری به کاربرد که سطح بالاتری از سلسله مراتب را نشان می‌دهد.

[1] bread crumbs (خرده‌نان) اشاره به داستان هنسل و گرتل که دو کودک برای گم نکردن راه خانه، مسیر رفت را با خرده‌نان علامت‌گذاری کردند (م).

گشت‌وگذار. کاربر را در گشت‌وگذار میان منوهای سلسله مراتبی، صفحات وب و صفحات نمایش تعاملی یاری می‌دهد.

الگو: EditInPlace

شرح مختصر: قابلیت ویرایش ساده متون را برای انواع معینی از محتویات در مکان نمایش آن‌ها فراهم می‌آورد. لازم نیست کاربر یک تابع ویرایش متن را به صراحت وارد کند.

جزئیات: کاربر، محتویاتی را روی صفحه نمایش می‌بیند که باید تغییر داده شود. دو بار کلیک ماوس روی محتویات به سیستم نشان می‌دهد که ویرایش لازم است. محتویات به حالت برجسته در می‌آید تا نشان داده شود که حالت ویرایش در دسترس است و کاربر، تغییرات مناسب را اعمال می‌کند.

عناصر گشت‌وگذار: هیچ.

جستجو. جستجوهای خاص محتویات را از طریق اطلاعات موجود در یک وب‌سایت یا موجود در انبار داده‌های ماندگاری که از طریق یک برنامه تعاملی قابل دستیابی است، امکان‌پذیر می‌سازد.

الگو: SimpleSearch

شرح مختصر: توانایی جستجو به دنبال یک وب‌سایت یا منبع داده‌های ماندگار برای یک آیتم داده‌ای ساده که توسط رشته‌ای حرفی-عددی توصیف می‌شود.

جزئیات: توانایی جستجو به دنبال رشته‌ای از کاراکترها چه به‌صورت محلی (یک صفحه یا یک فایل) چه به‌صورت سرتاسری (کل سایت یا کل بانک اطلاعاتی) را فراهم می‌سازد. فهرستی از یافته‌ها را به ترتیب احتمال برآوردن نیازهای کاربر تولید می‌کند. جستجو به دنبال آیتم‌های چندگانه یا عملگرهای بولی خاص را فراهم نمی‌سازد (الگوی جستجوی پیشرفته را ببینید).

عناصر گشت‌وگذار. هر مدخل در فهرست یافته‌ها نشان‌گر پیوندی به داده‌هایی است که در آن مدخل به آن‌ها ارجاع داده شده است.

عناصر صفحه. پیاده‌سازی عناصر ویژه‌ای از یک صفحه‌ی وب یا صفحه نمایش.

الگو: Wizard

شرح مختصر: کاربر را در یک کار پیچیده گام به گام به پیش می‌برد و برای کامل شدن این کار از طریق یک سری صفحات نمایش پنجره‌ای ساده، راهنمایی لازم را فراهم می‌آورد.

عناصر گشت‌وگذار. گشت‌وگذار به طرف جلو و عقب به کاربر امکان می‌دهد تا هر مرحله از فرایند ویزارد را بازبینی کند.

تجارت الکترونیک. این الگوها، که خاص وب‌سایت‌ها هستند، عناصر تکراری برنامه‌های مربوط به تجارت الکترونیک را پیاده‌سازی می‌کنند.

الگو: ShoppingCart

شرح مختصر: فهرستی از آیتم‌های انتخاب شده برای خرید را فراهم می‌آورد.

جزئیات: آیتم، کمیت، کد محصول، موجود بودن، قیمت، اطلاعات تحویل، هزینه حمل و نقل و سایر اطلاعات مرتبط با خرید را فهرست می‌کند. همچنین توانایی ویرایش (مانند حذف، تغییر کمیت) را فراهم می‌آورد.

عناصر گشت‌وگذار: حاوی توانایی لازم برای پیش بردن عملیات خرید یا رفتن به بخش تسویه حساب است.

متفرقه. الگوهایی که به آسانی در یکی از گروه‌های فوق نمی‌گنجند. در برخی موارد، این الگوها مستقل از دامنه‌اند یا فقط برای طبقه‌های خاصی از کاربران رخ می‌دهند.

الگو: ProgressIndicator

شرح مختصر: هنگامی که عملیاتی بیش از n ثانیه به طول انجامد، پیشرفت کار را نشان می‌دهد.

جزئیات: به‌صورت یک آیکون پویا نمایی شده یا کادر پیام‌نما (message box) نشان داده می‌شود که به شکل بصری (مثلاً یک مارپیچ چرخان، یا لغزنده‌ای که درصد کامل شدن کار را نشان می‌دهد) پیشرفت کار را مشخص می‌کند. ممکن است حاوی یک نشان‌گر محتویات متنی از وضعیت پردازش نیز باشد.

عناصر گشت‌وگذار: غالباً حاوی دکمه‌ای است که به کاربر امکان متوقف کردن یا مکث کردن پردازش را می‌دهد.

هرکدام از الگوهای مثال قبلی (و همه‌ی الگوهای درون هر گروه) نیز یک طراحی در سطح مؤلفه‌ها خواهند داشت که شامل کلاس‌های طراحی، صفات طراحی، عملیات‌های طراحی و واسط‌های طراحی می‌شود.

بحث جامعی درباره الگوهای واسط کاربر خارج از حوصله این کتاب است. اگر بیشتر علاقه‌مند هستید، برای اطلاعات بیشتر، [Duy02]، [Bor01]، [Tid02] و [Wei 01] را ببینید.

۱۲-۶ الگوهای طراحی برای برنامه‌های تحت وب

در سرتاسر این فصل، آموختید که انواع متفاوتی از الگوها و شیوه‌های بسیاری برای گروه‌بندی آن‌ها وجود دارد. هنگام پرداختن به مسائل طراحی مرتبط با ساخت برنامه‌های تحت وب ، در نظر گرفتن گروه‌های الگوها با در نظر گرفتن دو بُعد می‌تواند مفید واقع گردد: **کانون طراحی** (Design focus) الگو و **سطح دانه‌بندی** (Granularity) آن. *کانون طراحی* مشخص می‌کند که کدام جنبه از مدل طراحی مورد نظر است (مثلاً معماری اطلاعاتی، گشت‌وگذار یا تعامل). *دانه‌بندی*، سطح انتزاعی را مشخص می‌کند که در نظر گرفته می‌شود (مثلاً این که آیا الگو در کل برنامه تحت وب کاربرد دارد، در یک صفحه از برنامه تحت وب، در یک زیر سیستم یا تنها در یک مؤلفه از برنامه تحت وب؟).

۱۲-۶-۱ کانون طراحی (Design focus)

در فصل‌های گذشته بر ترتیبی از پیشرفت طراحی تأکید ورزیدیم که با پرداختن به معماری، مسائل سطح مؤلفه و نمایش‌های واسط کاربر انجام می‌گیرد. در هر مرحله، مسائلی که به آن‌ها می‌پردازید و راهکارهایی که پیشنهاد می‌کنید، در سطح بالایی از انتزاع آغاز می‌شوند و به تدریج بر جزئیات و مشخصات آن‌ها افزوده می‌شود. به بیان دیگر، کانون طراحی با نزدیک‌تر شدن به طراحی، باریک‌تر می‌شود. مسائل (و راهکارهایی) که هنگام طراحی معماری اطلاعاتی برای یک برنامه تحت وب به آن‌ها بر می‌خورید، با مسائل (و راهکارهایی) که هنگام اجرای طراحی واسط مشاهده می‌کنید، تفاوت دارند. بنابراین، تعجبی ندارد که الگوهای مربوط به طراحی برنامه‌های تحت وب را می‌توانید

نکته‌ی کلیدی
هرچه در طراحی عمیق‌تر شوید، آن‌چه که کانون توجه قرار می‌گیرد، ریزتر خواهد بود.

در سطوح متفاوتی از کانون طراحی توسعه دهید، به‌طوری که می‌توانید به مسائل منحصر به فرد (و راهکارهای مربوطی) که در هر سطح می‌بینید، بپردازید. الگوهای برنامه تحت وب را می‌توان بر اساس سطوح کانون طراحی زیرگروه‌بندی کرد:

- **الگوهای معماری اطلاعات** به سطح کلی فضای اطلاعات و شیوه‌های تعامل کاربران با این اطلاعات مربوط می‌شود.
- **الگوهای گشت‌وگذار** ساختارهای پیوندهای گشت‌وگذار، از قبیل سلسله مراتب‌ها، حلقه‌ها، گردش‌ها و غیره را تعریف می‌کنند.
- **الگوهای تعامل** در طراحی واسط کاربر سهم دارد. الگوها در این گروه نشان می‌دهند که چگونه واسط، کاربر را از پیامدهای یک کنش خاص آگاه می‌سازند، چگونه یک کاربر محتویات را بر اساس حیطه‌ی کاربردی و تمایلات کاربرد گسترش می‌دهد، چگونه به بهترین نحو می‌توان مقصد یک پیوند را توصیف کرد، چگونه کاربر را می‌توان از وضعیت یک تعامل در حال وقوع و مسائل مرتبط با واسط آگاه کرد.
- **الگوهای ارائه** به ارائه‌ی محتویات از طریق واسط کمک می‌کنند. الگوهای موجود در این گروه، به چگونگی سازمان‌دهی قابلیت‌های عملیاتی کنترلی برای قابلیت استفاده‌ی بهتر، چگونگی نشان دادن رابطه‌ی میان یک کنش واسط و اشیای محتوایی که بر آن‌ها تأثیر می‌گذارد و چگونگی ایجاد سلسله مراتب محتوایی اثربخش می‌پردازد.
- **الگوهای عملیاتی** جریان‌های کاری، رفتارها، پردازش، ارتباطات و سایر عناصر الگوریتمی را تعریف می‌کنند.

در اکثر موارد، بررسی مجموعه‌ای از الگوهای معماری هنگام مواجهه با مسأله‌ای در طراحی واسط می‌تواند بیهوده باشد. الگوهای تعامل را بررسی می‌کنید، زیرا این کانون طراحی است که با کار در حال انجام مرتبط است.

۲-۶-۱۲ دانه‌بندی طراحی (Design granularity)

هنگامی که مسأله‌ای شامل «تصویر بزرگ» می‌شود باید بکوشید تا راهکارهایی توسعه دهید (و از الگوهای مرتبط و مناسبی استفاده کنید) که تصویر بزرگ را کانون توجه قرار می‌دهند. برعکس، هنگامی که کانون توجه، بسیار باریک باشد (مثلاً انتخاب منحصر به فرد یک آیتم از مجموعه کوچکتری حاوی پنج آیتم یا کمتر)، راهکار (و الگوی متناظر) در گستره‌ای بسیار باریک هدف گرفته می‌شود. از نظر سطح دانه‌بندی، الگوها را می‌توان در سطوح زیر توصیف کرد:

- **الگوهای معماری.** این سطح از انتزاع معمولاً به الگوهایی مربوط می‌شود که ساختار کلی برنامه‌ی تحت وب تعریف می‌کند، روابط میان مولفه‌ها یا گام‌ها را نشان می‌دهند و قواعد مربوط به مشخص کردن روابط میان عناصر (صفحات، پکیج‌ها، مؤلفه‌ها و زیر سیستم‌های) معماری را تعریف می‌کنند.
- **الگوهای طراحی.** این الگوها به عنصر خاصی از طراحی نظیر تجمیع مؤلفه‌ها برای حل یک مسأله‌ی طراحی، روابط میان عناصر روی صفحه، یا سازوکارهای مربوط به برقراری رابطه‌ی مؤلفه به مؤلفه می‌پردازند. یک مثال می‌تواند الگوی **Broadsheet** برای چیدمان صفحه‌ی اصلی برنامه تحت وب باشد.

اطلاعات

مخزن الگوهای طراحی ابررسانه‌ای

وب‌سایت IAWiki (**http://iawiki.net/websitepatterns**) یک فضای بحث و بررسی برای معماران اطلاعات که حاوی منابع مفید بسیار است. از آن جمله می‌توان به پیوندهای منتهی به چند کاتالوگ و مخزن الگوهای ابر رسانه‌ای مفید اشاره کرد. صدها الگوی طراحی عرضه شده است:

Hypermedia Design Pattern Repository Index
(www.designpattern.lu.unisi.ch/)

Interaction Patterns by Tom Erickson

(www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html)

Web Patterns forUI Design

(http://harbinger.sims.berkley.edu/ui_designpatterns/home.php)

Patterns for personal UI Websites

(www.rdrop.com/%7Ehalf/Creations/Writings/Web.patterns/index.html)

Improving Web Information Systems with Navigational Patterns

(http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html)

An HTML 2.0 Pattern Language
(www.anamorph.com/docs/patterns/default.html)

Common Ground – A Pattern Language for HCI Design

(www.mit.edu/~jtidwell/interactive_patterns.html)

Patterns for personal UI Websites

(www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html)

Indexing Pattern Language for HCI Design

(www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html)

- **الگوهای مؤلفه‌ها.** این سطح از انتزاع به تک تک عناصر یک برنامه تحت وب در مقیاس کوچک مربوط می‌شود. مثال‌ها عبارتند از تک تک عناصر تعاملی (مثل دکمه‌های رادیویی)، آیتم‌های گشت‌وگذار (مثلاً چگونه ممکن است پیوندها را قالب‌بندی کرد) یا عناصر عملیاتی (مثلاً الگوریتم‌های خاص).

تعریف ارتباط و همخوانی الگوهای متفاوت با کلاس‌های متفاوت کاربردها یا دامنه‌ها نیز امکان‌پذیر است. برای مثال، مجموعه‌ای از الگوها (در سطح متفاوتی از کانون طراحی و دانه بندی) ممکن است با تجارت الکترونیک ارتباط و همخوانی داشته باشد.

۷-۱۲ خلاصه

الگوهای طراحی، سازوکاری مدون برای توصیف مسائل و راهکار آن‌ها فراهم می‌آورند به‌گونه‌ای که جامعه‌ی نرم‌افزاری می‌تواند دانش طراحی برای استفاده‌ی مجدد را به چنگ آورد. الگو، مسأله را

توصیف می‌کند، حیطه‌ای را نشان می‌دهد که به کاربر کمک می‌کند تا محیط مسأله را درک کند و سیستمی از نیروها را توصیف می‌کند که نشان می‌دهد مسأله را چگونه می‌توان در حیطه‌ی آن تفسیر کرد و راهکار را چگونه می‌توان به کاربرد. در کار مهندسی نرم‌افزار، الگوهای زایایی را شناسایی و مستندسازی می‌کنیم که جنبه‌ای مهم و تکرارپذیر از یک سیستم را توصیف می‌کنند که سپس راهی برای ساختن آن جنبه در داخل سیستمی از نیروها فراهم می‌آورد که در آن حیطه‌ی مفروض، منحصر به فرد است.

الگوهای معماری مسائل طراحی گسترده‌ای را توصیف می‌کند که با استفاده از یک رویکرد ساختاری حل می‌شوند. الگوهای طراحی، مسائل داده‌گرای تکراری و راهکارهای مدل‌سازی داده‌ای را توصیف می‌کنند که می‌توان در حل آن‌ها به‌کار برد. الگوهای مؤلفه‌ای (که از آن‌ها به‌عنوان الگوهای طراحی نیز یاد می‌شود) به مسائل مربوط به توسعه‌ی زیر سیستم‌ها و مؤلفه‌ها، شیوه‌ی برقراری ارتباط میان آن‌ها و مکان قرار گرفتن آن‌ها در یک معماری بزرگتر می‌پردازند. الگوهای طراحی واسط، مسائل متداول در زمینه واسط کاربر و راهکار آن‌ها با سیستمی از نیروها را توصیف می‌کنند که شامل خصوصیات ویژه‌ی کاربران نهایی می‌شود. الگوهای مربوط به برنامه‌های تحت وب به مجموعه مسائلی می‌پردازند که هنگام ساخت برنامه‌های تحت وب مشاهده می‌شوند و غالباً شامل بسیاری از گروه‌های الگوها می‌شوند که هم اکنون ذکر شد. چارچوب، زیرساختی فراهم می‌سازد که الگوها ممکن است در آن قرار گیرند و علاوه بر این، اصطلاحات لازم برای توصیف جزئیات پیاده‌سازی مختص زبان برنامه‌نویسی را برای کل یا بخشی از یک الگوریتم یا ساختمان داده‌های خاص در اختیار می‌گذارد.

طراحی مبتنی بر الگوها در ارتباط با روش‌های معماری، سطح مؤلفه‌ای و واسط کاربر استفاده می‌شود. این رویکرد طراحی، با بررسی مدل خواسته‌ها برای جداسازی مسائل، تعریف حیطه و توصیف سیستم نیروها آغاز می‌شود. سپس، زبان‌های الگو برای دامنه‌ی مسأله جستجو می‌شوند تا تعیین شود که آیا برای مسائل جداسازی شده الگو وجود دارد. هنگامی که الگوهای مناسب به‌دست آمد، از آن‌ها به‌عنوان راهنمای طراحی استفاده می‌شود.

مسائل و نکاتی برای تعمق

۱-۱۲ درباره سه بخش از یک الگوی طراحی بحث کنید و برای هر کدام از این سه بخش، مثالی از زمینه‌های غیر نرم‌افزار بیاورید.

۲-۱۲ چه اختلافی میان الگوی مولد و غیرمولد هست؟

۳-۱۲ الگوهای معماری چه تفاوتی با الگوهای مؤلفه‌ای دارند؟

۴-۱۲ چارچوب چیست و چه تفاوتی با الگو دارد؟ اصطلاح چیست و چه تفاوتی با الگو دارد؟

۵-۱۲ با استفاده از قالب الگوی طراحی ارائه شده در بخش ۳-۱-۱۲، برای الگویی که مربی شما پیشنهاد می‌کند، توصیف کاملی از آن الگو ارائه دهید.

۶-۱۲ برای ورزشی که با آن آشنایی دارید یک زبان الگوی اسکلتی توسعه دهید. می‌توانید با پرداختن به حیطه، سیستم نیروها و مسأله‌ی گسترده‌ای که مربی و تیم او باید حل کنند، شروع کنید. فقط باید نام‌های الگو را مشخص کنید و یک توصیف تک جمله‌ای برای هر الگو ارائه دهید.

۷-۱۲ پنج مخزن الگو پیدا کنید و شرح مختصری از انواع الگوهای موجود در هر کدام ارائه دهید.

۸-۱۲ هنگامی که کریستوفر الکساندر می‌گوید: «طراحی خوب صرفاً با کنار هم قرار دادن بخش‌های کاری، قابل تحقق نیست»، چه منظوری دارد.

۹-۱۲ با استفاده از وظایف طراحی مبتنی بر الگوها که در بخش ۳-۲-۱۲ ذکر شد یک طراحی اسکلتی برای «سیستم طراحی درونی» توصیف شده در بخش ۲-۳-۱۱ توسعه دهید.

۱۰-۱۲ برای الگوهایی که در مسأله‌ی ۹-۱۲ به‌کار بردید، یک جدول سازمان‌دهی الگوها بسازید.

۱۱-۱۲ الگوی **Kitchen** ذکرشده در بخش ۳-۱۲ را با به‌کارگیری قالب الگوی طراحی ارائه شده در بخش ۳-۱-۱۲، به‌طور کامل توصیف کنید.

۱۲-۱۲ باند چهار نفره [Gam95] چند الگوی مؤلفه‌ای متنوع پیشنهاد کرده‌اند که در سیستم‌های شیءگرا قابل به‌کارگیری‌اند. یکی از آن‌ها را انتخاب کنید و درباره آن بحث کنید (این الگوها در وب موجودند).

۱۳-۱۲ سه مخزن الگو برای الگوهای واسط کاربر بیابید. از هر کدام یک الگو انتخاب کنید و شرح مختصری از آن ارائه دهید.

۱۴-۱۲ سه مخزن الگو برای الگوهای برنامه تحت وب بیابید. از هر کدام یک الگو انتخاب کنید و شرح مختصری از آن ارائه دهید.

فصل ۱۳

طراحی برنامه‌های تحت وب

نگاهی گذرا

طراحی برنامه‌های تحت وب چیست؟ طراحی برای برنامه‌های تحت وب شامل فعالیت‌های فنی و غیر فنی می‌شود که عبارتند از: تعیین ظاهر برنامه‌ی تحت وب، ایجاد چیدمان زیبایی‌شناختی واسط کاربر، تعریف ساختار معماری کلی، توسعه محتوا و قابلیت عملیاتی که در معماری جای داده می‌شود و طراحی گشت‌وگذاری که در داخل برنامه‌ی تحت وب رخ می‌دهد.

چه کسی آن را انجام می‌دهد؟ مهندسان وب، طراحان گرافیک، نویسندگان محتوا، و سایر ذی‌نفع‌ها. همگی در ایجاد مدل طراحی برنامه‌ی تحت وب دخالت دارند.

چرا اهمیت دارد؟ طراحی به شما این امکان را می‌دهد که مدلی قابل ارزیابی برای کیفیت بسازید و آن را پیش از تولید محتوا و کدهای برنامه، اجرای آزمون‌ها، و دخالت تعداد زیادی از کاربران، بهبود ببخشید. طراحی، جایی است که کیفیت برنامه‌ی تحت وب تعیین می‌شود.

مراحل کار کدام است؟ طراحی برنامه‌های تحت وب شامل شش مرحله اصلی می‌شود که با اطلاعات به‌دست آمده طی مدل‌سازی خواسته‌ها به پیش برده می‌شوند. در طراحی محتوا از مدل محتوا (که طی تحلیل به‌دست می‌آید) به‌عنوان مبنایی برای ایجاد طراحی اشیای محتوایی استفاده می‌شود. در طراحی زیبایی‌شناختی (که طراحی گرافیکی نیز نامیده می‌شود) شکل ظاهری برنامه تعیین می‌شود که کاربر نهایی آن را می‌بیند. طراحی معماری، ساختار ابر رسانه‌ای کلی همه‌ی اشیای محتوایی و قابلیت‌های عملیاتی را کانون توجه قرار می‌دهد. طراحی واسط، چیدمان و سازوکارهای تعاملی را مشخص می‌کند که واسط کاربر را تعریف می‌کنند. طراحی گشت‌وگذار تعیین می‌کند که کاربر نهایی چگونه می‌تواند در میان ساختار ابر رسانه‌ای گشت‌وگذار کند و طراحی مؤلفه‌ها ساختار درونی مشروح را برای عناصر عملیاتی برنامه‌ی تحت وب تعیین می‌کند.

محصول کار چیست؟ یک مدل طراحی که شامل نکات طراحی محتوایی، طراحی زیبایی‌شناختی، طراحی واسط، طراحی گشت‌وگذار و طراحی در سطح مؤلفه‌ها می‌شود. محصول کاری اصلی است که طی طراحی برنامه‌ی تحت وب به‌دست می‌آید.

چگونه اطمینان حاصل کنم که درست از عهده کار بر آمده‌ام؟ هر عنصر از مدل طراحی بازبینی می‌شود تا خطاها، ناسازگاری‌ها، یا جا افتادگی‌ها کشف شود. به علاوه، راهکارهای متفاوت در نظر گرفته می‌شوند و میزان منجر شدن مدل طراحی فعلی به پیاده‌سازی اثربخش، سنجیده می‌شود.

یاکوب نیلسن [Nie00] در کتاب خود در باب طراحی وب می‌گوید: «اساساً دو رویکرد پایه برای طراحی وجود دارد: *ایده‌آل هنرمندانه برای بیان خودتان و ایده‌آل مهندسی برای حل مشکلی از مشتری.*» طی اولین دهه از توسعه‌ی وب، ایده‌ی هنرمندانه، رویکردی بود که خیلی‌ها برگزیدند. طراحی به شیوه‌ای تک‌منظوره رخ می‌داد و معمولاً با تولید صفحات HTML اجرا می‌شد. طراحی از یک چشم‌انداز هنرمندانه تکامل پیدا کرد که خود با پیاده‌سازی برنامه‌های تحت وب تکامل می‌یافت.

حتی امروزه، بسیاری از توسعه‌دهندگان وب از برنامه‌های تحت وب به‌عنوان نشانه‌ای برای «طراحی محدود» استفاده می‌کنند. آن‌ها چنین استدلال می‌کنند که ناپایداری و بی‌واسطه بودن برنامه‌های تحت وب، عوامل پرقدرتی در مقابل طراحی رسمی به شمار می‌روند؛ که طراحی به موازات ساخته شدن برنامه کاربردی تکامل می‌یابد و این که به زمان نسبتاً اندکی برای ایجاد یک مدل طراحی مشروح نیاز است. این استدلال، محاسنی دارد، ولی تنها برای برنامه‌های تحت وب نسبتاً ساده. هنگامی که محتوا و عملکرد سیستم پیچیده باشد؛ هنگامی که برنامه‌های تحت وب شامل صدها یا هزاران شیء محتوایی، قابلیت عملیاتی و کلاس تحلیل می‌شود؛ و هنگامی که موفقیت برنامه‌ی تحت وب تأثیری مستقیم بر موفقیت تجاری دارد، طراحی را نمی‌توان و نباید سبک شمرد.

این واقعیت ما را به رویکرد دوم نیلسن رهنمون می‌شود- «ایده‌آل مهندسی برای حل مشکلی از مشتری.» در *مهندسی وب*[1] این فلسفه پذیرفته می‌شود و رویکردی سخت‌گیرانه‌تر به سازندگان این امکان را می‌دهد تا به آن دست پیدا کنند.

۱۳-۱ کیفیت طراحی برنامه‌های تحت وب

طراحی، یک فعالیت مهندسی است که به ایجاد محصول با کیفیت بالا می‌انجامد. این ما را به سؤالی تکراری رهنمون می‌شود که در همه‌ی رشته‌های مهندسی مطرح است: کیفیت چیست؟ در این بخش به بررسی پاسخ این سؤال در حیطه‌ی توسعه برنامه‌های تحت وب خواهیم پرداخت.

هر کس که در وب مروری کرده باشد یا از یک اینترنت شرکتی استفاده کرده باشد، از آن‌چه که برنامه‌ی تحت وب را «خوب» جلوه دهد، ایده‌ای در ذهن دارد. دیدگاه‌های فردی بسیار متنوع‌اند. برخی کاربران از تصاویر گرافیکی پر زرق و برق لذت می‌برند؛ عده‌ای دیگر خواهان متون ساده‌اند. برخی به دنبال اطلاعات فراوان هستند؛ دیگران به عرضه‌ی خلاصه‌ی مطالب علاقه دارند. برخی دستیابی به ابزارهای تحلیلی پیچیده یا بانک‌های اطلاعاتی را دوست دارند و برخی خواهان سادگی‌اند. در حقیقت، ادراک کاربران از «خوب بودن» (و پذیرش یا رد برنامه‌ی تحت وب در نتیجه‌ی این ادراک) ممکن است از هر بحث فنی درباره کیفیت برنامه‌ی تحت وب مهم‌تر باشد.

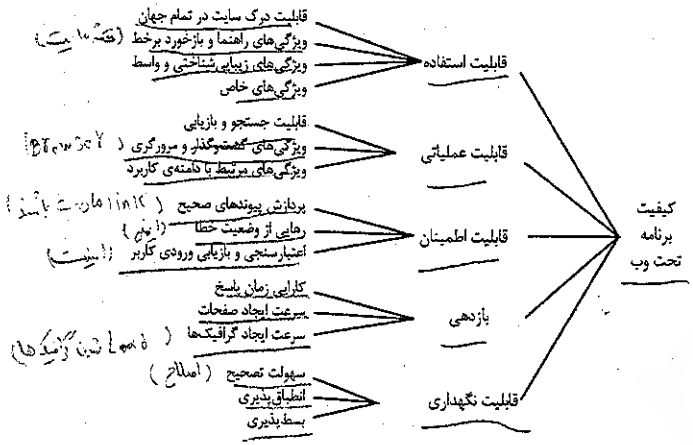
ولی کیفیت برنامه‌ی تحت وب را چگونه باید درک کرد؟ در برنامه‌ی تحت وب چه صفاتی باید وجود داشته باشد تا خوب بودن به چشم کاربران نهایی بیاید و در عین حال، خصوصیات فنی کیفیت را از خود نشان دهد که شما را در درازمدت قادر به تصحیح، تطبیق، به‌سازی و پشتیبانی از آن سازد؟

> «اگر محصولات طوری طراحی شوند که بهترنما تمایلات رفتاری طبیعی انسان‌ها همخوانی داشته باشند، در آن صورت رضایت‌مندی بیشتر می‌شود و بهره‌وری بالا می‌رود.»
>
> سوزان واین شنک

[1] *مهندسی وب* [Pre08] نسخه‌ی اصلاح‌شده‌ای از همان روش مهندسی نرم‌افزار است که در سرتاسر این کتاب ارائه شده است و در آن چارچوبی چابک و درعین حال منضبط برای سیستم‌ها و برنامه‌های مبتنی بر وب با کیفیت صنعتی پیشنهاد می‌شود.

در واقع، همه‌ی خصوصیات فنی کیفیت طراحی که در فصل ۸ بحث شدند و صفات کیفیتی کلی که در فصل ۱۴ ارائه خواهند شد، برای برنامه‌های تحت وب کاربرد دارند. ولی، مرتبط‌ترین این صفات- قابلیت استفاده، قابلیت عملیاتی، قابلیت اطمینان، بازدهی و قابلیت نگهداری- مبنایی مفید برای ارزیابی کیفیت سیستم‌های مبتنی بر وب فراهم می‌آورند.

اولسینا و همکاران [Ols99] یک «**درخت خواسته‌های کیفیتی**» تهیه کرده‌اند که مجموعه‌ای از صفات- قابلیت استفاده، قابلیت عملیاتی، قابلیت اطمینان، بازدهی و قابلیت نگهداری- را تعیین می‌کند که به کیفیت بالای برنامه‌های تحت وب منجر می‌گردد.[1] کار آن‌ها در شکل ۱-۱۳ خلاصه شده است. اگر قرار باشد برنامه‌های تحت وب را طی مدت زمان طولانی طراحی، پیاده‌سازی و نگهداری کنید، ملاک‌های ذکرشده در شکل مورد توجه ویژه خواهند بود.



شکل ۱-۱۳ درخت خواسته‌های کیفیتی.

آفوت [Off02] پنج صفت کیفیتی عمده‌ی ذکر شده در شکل ۱-۱۳ را با افزودن صفات زیر بسط می‌دهد:

امنیت. برنامه‌های تحت وب، همکاری و همبستگی سنگینی با بانک‌های اطلاعاتی شرکتی و دولتی مهم پیدا کرده‌اند. برنامه‌های کاربردی تجارت الکترونیک، اطلاعات حساس مشتریان را استخراج و سپس ذخیره می‌کنند. به این دلایل و دلایل بسیار دیگر، امنیت برنامه‌ی تحت وب در بسیاری از شرایط، اهمیت بنیادی دارد. راهکار کلیدی، توانایی برنامه‌ی تحت وب و محیط سرور آن در رد دستیابی غیر مجاز و/یا جلوگیری از حمله نفوذگران است. بحث مشروح درباره امنیت برنامه‌های تحت وب خارج از حوصله این کتاب است. در صورت علاقه به این موضوع، [Vac06]، [Kiz05] یا [Kal03] را ببینید.

> صفات اصلی کیفیتی برای برنامه‌ی تحت وب کدام‌اند؟

[1] این صفات کیفیتی بسیار مشابه با همان صفات ارائه شده در فصل‌های ۸ و ۱۴ هستند. نتیجه: خصوصیت‌های کیفیتی برای همه‌ی نرم‌افزارها یکسان و جهان شمول هستند.

اطلاعات

طراحی برنامه‌ی تحت وب-چک‌لیست کیفیت

چک‌لیست زیر، که از اطلاعات ارائه شده در Webrefrence.com برگرفته شده است، حاوی چند پرسش است که هم کاربران نهایی و هم طراحان وب را در ارزیابی کیفیت کلی برنامه‌ی تحت وب یاری می‌دهد:

- آیا گزینه‌های محتوایی و/یا قابلیتی و/یا گشت‌وگذاری بنا به سلیقه‌ی کاربر قابل تنظیم است؟
- آیا محتوا و/یا قابلیت‌ها بر اساس پهنای باندی که کاربر در اختیار دارد، قابل تغییر است؟
- آیا از رسانه‌های گرافیکی و سایر رسانه‌های غیر متنی به‌طور مناسب استفاده شده است؟ آیا اندازه فایل‌های گرافیکی برای بازدهی صفحه نمایش بهینه شده‌اند؟
- آیا جدول‌ها به شیوه‌ای سازمان‌دهی و اندازه‌بندی شده‌اند که بتوان آن‌ها را به‌طرزی اثربخش به نمایش در آورد و قابل درک باشند؟
- آیا HTML برای حذف ناکارآمدی‌ها بهینه شده است؟
- آیا طراحی کلی صفحه را به راحتی می‌توان خواند یا در آن گشت‌وگذار کرد؟
- آیا همه‌ی اشاره گرها پیوندی به اطلاعات مورد نظر کاربران فراهم می‌آورند؟
- آیا این احتمال وجود دارد که اکثر پیوندها روی وب ماندگار باشند؟
- آیا برنامه‌ی تحت وب به ابزارهای مدیریت مجهز هست که مواردی نظیر دنبال کردن میزان استفاده از سایت، آزمون پیوندها، جستجوی محلی و امنیت را امکان‌پذیر سازد؟

دسترس‌پذیری. حتی بهترین برنامه‌های تحت وب هم، اگر در دسترس نباشد، نمی‌تواند نیازهای کاربران را فراهم سازد . از دیدگاه فنی، دسترس‌پذیری، میزانی از درصد زمان در دسترس بودن برنامه‌ی تحت وب برای استفاده است. کاربر نهایی معمولی انتظار دارد که برنامه‌ی تحت وب، بیست و چهار ساعته/ هفت روز هفته و ۳۶۵ روز سال در دسترس باشد. هر چیزی کمتر از این، غیر قابل قبول در نظر گرفته می‌شود.[1] ولی «زمان برقراری» تنها شاخص برای در دسترس بودن نیست. افوت [Off02] معتقد است که «استفاده از ویژگی‌های در دسترس تنها روی یک مرورگر یا یک سکو» برنامه‌ی تحت وب را از دسترسی کسانی که مرورگر/ سکوی دیگری در اختیار دارند، خارج می‌سازد. پس کاربر ناگزیر به جای دیگر می‌رود.

گسترش‌پذیری (Scalabilty). آیا اندازه‌ی برنامه‌ی تحت وب و محیط سرور آن را می‌توان تغییر داد تا بتواند به ۱۰۰، ۱۰۰۰، ۱۰۰۰۰ یا ۱۰۰۰۰۰ کاربر سرویس بدهد؟ آیا برنامه‌ی تحت وب و سیستم‌هایی که با آن‌ها مرتبط است، قادرند از پس تغییرات حجم برآیند یا آیا پاسخ گویی به‌طور چشمگیری پایین می‌آید (یا اصلاً متوقف می‌شود)؟ ساخت یک برنامه‌ی تحت وب موفق کافی نیست. ساخت برنامه‌ی تحت وبی که بار موفقیت را تحمل کند (کاربران بسیار بیشتر) و موفق‌تر شود نیز به همان اندازه اهمیت دارد.

زمان عرضه به بازار. گرچه زمان عرضه به بازار از نظر فنی یک صفت کیفیتی واقعی به شمار نمی‌رود، از دیدگاه تجاری میزانی از کیفیت محسوب می‌شود. نخستین برنامه‌ی تحت وبی که یک

[1] این انتظار البته واقع‌بینانه نیست. برنامه‌های تحت وب بزرگ باید زمان‌هایی برای ترمیم و پهنگام‌سازی در نظر بگیرند.

بخش مشخصی از بازار را پاسخ دهد، غالباً تعداد شگفت‌انگیزی از کاربران نهایی را جذب می‌کند. برای آن‌ها که در جستجوی اطلاعات هستند، میلیاردها صفحه وب در دسترس است. حتی جستجوهای بسیار هدفمند در وب نیز به سیلی از محتوا منجر می‌شود. با وجود این تعداد زیاد منابع اطلاعات، کاربر چگونه می‌تواند به کیفیت (صداقت، صحت، کامل بودن، وقت شناسی) محتوای برنامه‌ی تحت وب دست پیدا کند؟ تیلمن [Til00] مجموعه‌ای از ملاک‌های مفید برای ارزیابی کیفیت محتوا ارائه می‌دهد:

هنگام ارزیابی محتوا چه چیزهایی را باید مد نظر داشت؟

- آیا حوزه و عمق محتوا را می‌توان به آسانی تعیین کرد و اطمینان یافت که نیازهای کاربر را برآورده می‌سازد؟
- آیا زمینه‌ی علمی و سوابق نویسندگان محتوا به راحتی قابل تعیین است؟
- آیا می‌توان از به‌روز بودن اطلاعات، آخرین به‌روزرسانی و آنچه که به‌روز شده است، آگاهی یافت؟
- آیا محتوا و مکان آن‌ها بایدارند (مثلاً آیا در URL ارجاع داده شده باقی خواهند ماند)؟
- علاوه بر این پرسش‌های مرتبط با محتوا، موارد زیر را نیز می‌توان افزود:
- آیا محتوا، قابل اطمینان هست؟
- آیا محتوا، منحصر به فرد است؟ یعنی، آیا برنامه‌ی تحت وب، مزایای منحصر به فردی برای کاربران خود به همراه دارد؟
- آیا محتوا برای جامعه کاربران هدف ارزش‌مند است؟
- آیا محتوا به خوبی سازمان‌دهی شده است؟

چک‌لیست ذکرشده در این بخش، تنها نمونه‌ی کوچکی از مسائل را نشان می‌دهد که باید در تکامل طراحی برنامه‌های تحت وب مد نظر داشت.

۲-۱۳ اهداف طراحی

جین کیسر [Kai02] در ستون منظم خود درباره طراحی وب، مجموعه‌ای از اهداف را تعریف می‌کند که در واقع در هر برنامه‌ی تحت وب فارغ از دامنه کاربرد، اندازه یا پیچیدگی قابل استفاده است.

«صرف توانایی، به معنای لزوم نیست.»
جین کیسر

سادگی (Simplicity). ممکن است این گفته قدیمی به‌نظر برسد، ولی در برنامه‌های تحت وب «همه چیز باید معتدل باشد.» برخی طراحان تمایل دارند «بیش از حد لازم» محتوا در اختیار کاربران قرار دهند- محتوای زیادی، تصاویر فراوان، پویانمایی‌های بیجا، صفحات وب بی‌شمار و این فهرست همچنان ادامه می‌یابد. بهتر است تلاش کنیم تا سادگی و اعتدال رعایت شود. محتوا باید حاوی اطلاعات مفید و موجز باشد و از شیوه‌ی تحویلی استفاده کند که با اطلاعات تحویلی (متون، تصاویر ویدیویی یا ثابت، صوت) سنخیت داشته باشد. ظاهر برنامه باید دلپذیر باشد و طوری نباشد که آزاردهنده شود (مثلاً بیش از حد رنگارنگ باشد که به جای بهتر کردن تعامل، در کاربر، ایجاد دافعه کند). معماری باید به ساده‌ترین شیوه‌ی ممکن، اهداف برنامه‌ی تحت وب را دست یافتنی کند. گشت‌وگذار باید صریح باشد و سازوکارهای گشت‌وگذار باید به‌طور حسی برای کاربر نهایی واضح باشد. استفاده از قابلیت‌ها باید آسان و درک آن‌ها باید آسان‌تر باشد.

سازگاری (Consistency). این هدف طراحی در واقع در هر عنصر از مدل طراحی اعمال می‌شود. محتوا باید به‌صورت سازگار ساخته شود (مثلاً فرمت‌بندی متون و سبک فونت‌ها باید در سرتاسر

مستندات متنی یکسان باشد؛ تصاویر گرافیکی از نظر الگوی رنگ و سبک هنری ظاهری هماهنگ داشته باشند). طراحی گرافیکی (زیبایی‌شناسی) باید ظاهری سازگار در میان همه‌ی بخش‌های برنامه‌ی تحت وب ارائه دهند. طراحی معماری باید قالب‌هایی را وضع کند که به یک ساختار ابررسانه‌ای سازگار منجر شوند. در طراحی واسط‌ها باید شیوه‌های سازگار تعامل، گشت‌وگذار، و نمایش محتوا تعریف گردد. سازوکارهای گشت‌وگذار را باید به‌طور سازگار در میان همه‌ی عناصر برنامه‌ی تحت وب به‌کار برد. چنان که کیسر [Kai02] می‌گوید: «به‌خاطر داشته باشید که نزد بازدیدکننده، وب‌سایت، یک مکان فیزیکی است. اگر صفحات داخل یک سایت از نظر طراحی سازگار نباشند، باعث سردگمی می‌شوند.»

هویت (Identity). طراحی زیبایی‌شناختی، واسط و گشت‌وگذار در یک برنامه‌ی تحت وب باید با دامنه کاربردی که برای آن ساخته می‌شود، سازگار باشد. وب‌سایتی که برای یک گروه موسیقی ساخته می‌شود، بدون شک ظاهری متفاوت با برنامه‌ی تحت وب طراحی‌شده برای شرکت خدمات مالی خواهد داشت. معماری برنامه‌ی تحت وب کاملاً متفاوت خواهد بود، واسط‌ها طوری ساخته می‌شوند که گروه‌های متفاوت کاربران را پاسخ گو باشند؛ گشت‌وگذار، طوری سازمان‌دهی می‌شود که اهداف متفاوت برآورده شود. شما (و سایر کسانی که در طراحی سهم دارند) باید بکوشید در سرتاسر طراحی، هویتی برای برنامه‌ی تحت وب برقرار کنید.

استحکام (Robustness). بر اساس هویتی که برقرار شده است، برنامه‌ی تحت وب غالباً «نویدی» ضمنی به‌کاربر می‌دهد. کاربر انتظار محتوا و قابلیت‌هایی مستحکم را دارد که با نیازهای او در ارتباط باشند. اگر جای این عناصر خالی باشد یا به قدر کافی وجود نداشته باشند، این احتمال وجود دارد که برنامه‌ی تحت وب به شکست بینجامد.

قابلیت گشت‌وگذار. قبلاً گفتیم که گشت‌وگذار باید ساده و سازگار باشد. همچنین طراحی باید مبتنی بر حس و قابل پیش‌بینی باشد. یعنی، کاربر باید بداند چگونه در برنامه‌ی تحت وب به گردش بپردازد، بدون این که مجبور باشد به دنبال پیوندها یا راهنمایی‌های مربوط بگردد. برای مثال، اگر مجموعه‌ای از آیکون‌ها یا تصاویر گرافیکی حاوی آیکون و تصاویر انتخاب شده‌ای است که به‌عنوان سازوکارهای گشت‌وگذار به‌کار می‌روند، آن‌ها را باید از نظر بصری مشخص کرد. هیچ چیز خسته‌کننده‌تر از این نیست که بکوشید یک پیوند فعال را در میان توده‌ای از تصاویر گرافیکی بیابید.

قراردادن پیوندهایی به محتوا و قابلیت‌های اصلی برنامه‌ی تحت وب در مکانی قابل پیش‌بینی نیز اهمیت بسیار دارد. اگر به جابه‌جایی در صفحه نیاز باشد (که معمولاً هست) قراردادن پیوندهایی در بالا و پایین صفحه، گشت‌وگذار را آسان‌تر می‌کند.

جاذبه‌ی بصری. از میان همه‌ی گروه‌های نرم‌افزار، برنامه‌های کاربردی تحت وب بی‌تردید بصری‌ترین، پویاترین و بی‌هیچ عذری زیبایی شناسانه‌ترین نوع نرم‌افزارها هستند. زیبایی (جاذبه‌ی بصری) بدون شک امری سلیقه‌ای است، ولی بسیاری از خصوصیات طراحی (مثل ظاهر محتوا؛ چیدمان واسط؛ هماهنگی رنگ‌ها؛ موازنه‌ی متون، گرافیک و سایر رسانه‌ها؛ سازوکارهای گشت‌وگذار) در جاذبه‌ی بصری سهم دارند.

همسازمندی (Compatibility). برنامه‌های تحت وب در محیط‌های متنوع (مثلاً سخت افزارها، انواع اتصال‌های اینترنتی، سیستم‌های عامل و مرورگرهای متفاوت) به‌کار گرفته خواهند شد و باید طوری طراحی شوند که با همه‌ی آن‌ها همساز باشند.

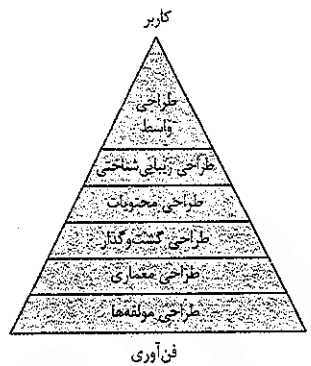
> «از نظر برخی، طراحی وب، شکل و شمایل بصری را کانون توجه قرار می‌دهد. از نظر برخی دیگر، طراحی وب به ساختاردهی اطلاعات و گشت‌وگذار در میان فضای مستندات مربوط می‌شود. سایرین حتی ممکن است طراحی وب را به فن‌آوری مربوط بدانند. در واقع، طراحی شامل همه‌ی این چیزها و حتی بیشتر از این می‌شود.»
>
> تامس پاول

۱۳-۳ هرم طراحی برای برنامه‌های تحت وب

طراحی برنامه‌ی تحت وب چیست؟ پاسخ‌دادن به این پرسش ساده، دشوارتر از آن چیزی است که بتوان باور کرد. من و دیوید لاو در کتاب خود در باب مهندسی وب [Pre08] در این مورد چنین نوشته ایم:

> ایجاد طراحی اثربخش معمولاً به مجموعه متنوع و گسترده‌ای از مهارت‌ها نیاز دارد. گاهی برای پروژه‌های کوچک، یک نفر به تنهایی باید چند مهارت داشته باشد. برای پروژه‌های بزرگتر، ممکن است بهره بردن از کارشناسان و متخصصان قابل توصیه و/یا امکان‌پذیر باشد: مهندسان وب، طراحان گرافیک، نویسندگان مطالب، برنامه نویسان، متخصصان بانک اطلاعاتی، معماران اطلاعات، مهندسان شبکه، کارشناسان امنیت و آزمون گران. بهره‌مندی از این مهارت‌های گوناگون، ایجاد مدلی را میسر می‌سازد که می‌توان کیفیت آن را ارزیابی کرد و *پیش از* تولید کدها و محتوا، آن را آزمود، آزمون‌ها را اجرا کرد و کاربران نهایی را به تعداد زیاد در این آزمون دخالت داد. اگر تحلیل جایی است که *کیفیت برنامه‌ی تحت وب تعیین می‌شود*، طراحی جایی است که *کیفیت حقیقتاً نهادینه می‌شود*.

آمیزه مناسبی از مهارت‌ها، بسته به ماهیت برنامه‌ی تحت وب، متغیر خواهد بود. در شکل ۲-۱۳، هرمی برای طراحی برنامه‌های تحت وب تصویر شده است. هر سطح از این هرم، یک کنش طراحی را نشان می‌دهد که در بخش‌های بعد به توصیف آن‌ها خواهیم پرداخت.



شکل ۲-۱۳ هرم طراحی برای برنامه‌های تحت وب.

۱۳-۴ طراحی واسط برنامه‌ی تحت وب

هنگامی که کاربری با سیستم کامپیوتری تعامل دارد، مجموعه‌ای از اصول بنیادی و دستورالعمل‌های طراحی مهم اعمال می‌شود که آن‌ها را در فصل ۱۱ بحث کردیم.[1] گرچه برنامه‌های تحت وب چند چالش خاص در خصوص طراحی واسط کاربر ایجاد می‌کنند، این اصول و دستورالعمل‌های بنیادی درباره برنامه‌های تحت وب نیز قابل اعمال هستند.

[1] بخش ۵-۱۱ به طراحی واسط برنامه‌های تحت وب اختصاص یافته است. اگر هنوز آن را نخوانده‌اید اکنون زمان خواندن آن است.

> «اگر سایتی کاملاً قابل استفاده باشد، ولی فاقد ظرافت و سبک طراحی مناسب باشد، شکست خواهد خورد.»
>
> کرت کلوئینگر

یکی از چالش‌های طراحی واسط برای برنامه‌های تحت وب، ماهیت نامعین نقطه ورود کاربر است. یعنی کاربر ممکن است از صفحه‌ی اصلی وارد برنامه‌ی تحت وب شود یا ممکن است پیوندی او را به یکی از سطوح پایین‌تر در معماری برنامه‌ی تحت وب هدایت کرده باشد. در برخی موارد، برنامه‌ی تحت وب را می‌توان طوری طراحی کرد که کاربر را مستقیماً به صفحه اصلی هدایت کند، ولی اگر این حالت مطلوب نبود، طراحی برنامه‌ی تحت وب باید ویژگی‌هایی برای گشت‌وگذار در واسط فراهم بیاورد که شامل همه‌ی اشیای محتوایی شوند و فارغ از چگونگی ورود کاربر به سیستم، در دسترس باشند.

اهداف واسط برنامه‌ی تحت وب عبارتند از: (۱) ساخت پنجره‌ای سازگار فراروی محتوا و قابلیت‌های فراهم آمده به‌وسیله‌ی واسط، (۲) راهنمایی کاربر از طریق یک سری تعامل با برنامه‌ی تحت وب و (۳) سازمان‌دهی گزینه‌های گشت‌وگذار و محتوای در دسترس کاربر. به منظور دستیابی به واسطی سازگار، نخست باید از طراحی زیبایی شناسانه (بخش ۵-۱۳) برای رسیدن به «سیمایی» یکپارچه استفاده کنید. این کار شامل خصوصیات بسیار می‌شود، ولی چیدمان و شکل گشت‌وگذار باید مورد تأکید قرار گیرد. برای راهنمایی تعامل کاربران، ممکن است از استعاره‌های مناسبی بهره ببرید[1] که کاربر را قادر می‌سازند تا درکی درست از واسط به‌دست آورد. برای پیاده‌سازی گزینه‌های گشت‌وگذار، می‌توانید یکی از چند سازوکار تعاملی زیر را انتخاب کنید:

- ***منوهای گشت‌وگذار***- منوهای کلید واژه‌ای (که به‌طور افقی یا عمودی سازمان‌دهی می‌شوند) محتوا و/یا قابلیت‌های کلیدی را فهرست می‌کنند. این منوها را می‌توان طوری پیاده‌سازی کرد که کاربر بتواند از یک سلسله مراتب عناوین فرعی، که هنگام انتخاب گزینه منوی اصلی نمایش داده می‌شود، آن چه را که می‌خواهد، برگزیند.
- ***آیکون‌های گرافیکی***- دکمه‌ها، سویچ‌ها و تصاویر گرافیکی مشابهی که کاربر را قادر به انتخاب یک خاصیت یا مشخص کردن یک تصمیم می‌سازند.
- ***تصاویر گرافیکی***- نمایشی گرافیکی که کاربر می‌تواند انتخاب کند و پیوندی منتهی به یک شیء محتوایی یا قابلیت عملیاتی برنامه‌ی تحت وب را پیاده‌سازی می‌کند.

ذکر این نکته حائز اهمیت است که یک یا چند مورد از این سازوکارهای کنترلی را در هر سطح از سلسله مراتب محتوا باید فراهم ساخت.

چه سازوکارهای تعاملی در دسترس طراحان برنامه‌های تحت وب قرار دارد؟

۱۳-۵ طراحی زیبایی‌شناسانه

طراحی زیبایی شناسانه، که گاه *طراحی گرافیکی* نیز خوانده می‌شود، تلاشی هنرمندانه است که جنبه‌های فنی طراحی برنامه‌ی تحت وب را تکمیل می‌کند. برنامه‌ی تحت وب، بدون طراحی گرافیکی ممکن است کار کند، ولی جاذبه ندارد. اما طراحی گرافیکی، کاربران را به دنیایی عمیق و هوشمند جذب می‌کند.

ولی زیبایی‌شناسی چیست؟ مثلی قدیمی است که می‌گوید: «زیبایی در چشم بیننده است.» این گفته، به‌ویژه هنگامی صحت دارد که طراحی برای برنامه‌های تحت وب مد نظر باشد. برای اجرای

هر طراح وبی، استعداد هنرمندانه زیبایی‌شناسی ندارد. اگر شما از آن زمره‌اید، برای کار طراحی زیبایی‌شناختی یک طراح گرافیک کارآزموده را به کار بگیرید.

[1] در این حیطه، *استعاره*، نمایشی (برگرفته از تجربیات واقعی کاربر) است که در حیطه‌ی واسط، قابل پیاده‌سازی است. یک مثال ساده، سویچ لغزنده‌ای است که در کنترل حجم صدای فایل‌های mpg به‌کار می‌رود.

طراحی زیبایی‌شناسانه، به سلسله مراتب کاربران رجوع کنید که به‌عنوان بخشی از مدل خواسته‌ها توسعه یافت (فصل ۵) و بپرسید، *کاربران برنامه‌ی تحت وب چه کسانی هستند و چه شکل و ظاهری را می‌پسندند؟*

۱-۵-۱۳ مسائل مربوط به چیدمان

هر صفحه‌ی وب دارای مقدار محدودی «منابع» است که می‌توان از آن برای گرافیک‌های غیر عملیاتی، ویژگی‌های گشت‌وگذار، محتوای اطلاعاتی و قابلیت‌های عملیاتی اداره شده توسط کاربران استفاده کرد. توسعه‌ی این «منابع» طی طراحی زیبایی‌شناختی برنامه‌ریزی می‌شود.

همانند همه‌ی مسائل زیبایی‌شناسی، هیچ قاعده‌ی مطلقی هنگام طراحی چیدمان وجود ندارد. ولی در نظر گرفتن چند دستورالعمل کلی می‌تواند ارزش‌مند باشد:

از فضای خالی نترسید. توصیه نمی‌شود که هر سانتی‌متر مربع از صفحه‌ی وب را از اطلاعات آکنده سازید. در ازدحام حاصل، تشخیص اطلاعات و یا ویژگی‌های مورد نیاز برای کاربر دشوار می‌شود و آشفتگی بصری‌ای که به‌وجود می‌آید، چشم‌نواز نیست.

بر محتوا تأکید کنید. گذشته از همه‌ی این‌ها، دلیل حضور کاربر، همین محتواست. نیلسن [Nie00] پیشنهاد می‌کند که صفحه‌ی وب معمولاً باید ۸۰٪ مطلب داشته باشد و باقیمانده به گشت‌وگذار و سایر ویژگی اختصاص داده شود.

عناصر را از چپ به راست و از بالا به پایین سازمان‌دهی کنید. اکثریت کاربران، محتوای صفحات وب را همانند صفحات کتاب- از بالا- سمت چپ به پایین- سمت راست پویش می‌کنند.[1] اگر عناصر چیدمان، تقدم خاصی دارند، عناصر با تقدم بالا باید در بخش بالا- سمت چپ «منابع» قرار داده شود.

گشت‌وگذار، محتوا و قابلیت‌های عملیاتی را به لحاظ جغرافیایی گروه‌بندی کنید. انسان‌ها اصولاً در هر چیزی به دنبال الگوها هستند. اگر الگوی قابل تشخیص در صفحات وب پیدا نکنند، احتمالاً آزرده‌خاطر می‌شوند (به دلیل جستجوی بیهوده برای اطلاعات مورد نیاز).

«منابع» خود را با نوارهای جابه‌جایی (Scroll bar) توسعه ندهید. گرچه نیاز به استفاده از نوارهای جابه‌جایی غالباً ضروری است، اکثر مطالعات نشان می‌دهد که کاربران ترجیح می‌دهند از این امکان استفاده نکنند. بهتر است محتوای صفحه را کاهش دهید یا محتوای ضروری را در چند صفحه عرضه کنید.

هنگام طراحی چیدمان، تفکیک و اندازه‌ی صفحه‌ی پنجره را مد نظر داشته باشید. به جای تعریف اندازه‌های ثابت در یک چیدمان، در طراحی باید همه‌ی آیتم‌های چیدمان را به‌صورت درصدی از فضای متغیر مشخص سازید [Nie00].

«آدم‌هایی را می‌بینم که یک سایت را به‌سرعت و تنها از روی طراحی بصری آن قضاوت می‌کنند.»

دستورالعمل‌های استانفورد برای باورپذیری وب

آندرز

کاربران جابجایی عمودی صفحه را بیش از جابجایی افقی تحمل می‌کنند. از فرمت‌بندی‌های گسترده بپرهیزید.

۲-۵-۱۳ مسائل طراحی گرافیک

در طراحی گرافیک، هر جنبه از ظاهر و شمایل برنامه‌ی تحت وب مد نظر قرار می‌گیرد. فرایند طراحی گرافیک با چیدمان (بخش ۱-۵-۱۳) آغاز می‌شود و با پرداختن به الگوهای سرتاسری رنگ؛ انواع، اندازه‌ها و سبک‌های متون؛ استفاده از رسانه‌های مکمل (مثل صدا، تصویر، پویانمایی) و همه‌ی عناصر زیبایی‌شناسی یک برنامه کاربردی ادامه می‌یابد.

[1] از نظر فرهنگی و زبانی استثنائاتی وجود دارد ولی این قاعده برای اکثر کاربران برقرار است.

بحث جامعی درباره مسائل طراحی گرافیکی برای برنامه‌های تحت وب از حوصله این کتاب خارج است. ترفندها و دستورالعمل‌های طراحی را از بسیاری از وب‌سایت‌ها که به همین منظور اختصاص یافته‌اند (مثل **www.grantasticdesign.com**، **www.grophic-design.com** و **www.wpdft.com**) یا از چند منبع چاپی (مثلاً [Roc06] و [Gor02]) به‌دست آورد.

اطلاعات

وب‌سایت‌هایی با طراحی خوب

گاهی بهترین راه برای درک طراحی خوب برای برنامه‌های تحت وب، نگاه کردن به چند مثال است. مارسل تور در مقاله‌ای با عنوان «بیست ترفند خوب برای طراحی وب» (**www.graphic-design.com/Web/feature/tips.html**) وب‌سایت‌های زیر را به‌عنوان مثال‌هایی از طراحی خوب پیشنهاد می‌کند.

www.creativepro.com/designresource/hom/787.html

یک شرکت طراحی به مدیریت پریمو آنیلی.

www.workbook.com- این سایت، ویترینی است برای نمایش کارهای طراحان و تصویرگران.

www.pbs.org/riverofsong- یک سریال تلویزیونی برای رادیو و تلویزیون عمومی درباره‌ی موسیقی امریکایی.

www.creativehostlist.com/index.html- منبع خوبی برای سایت‌های خوش ساخت که توسط شرکت‌های تبلیغاتی، طراحان گرافیک و سایر متخصصان ارتباطات طراحی شده‌اند.

www.btdnyc.com- یک شرکت طراحی به مدیریت بِث تودرو.

۶-۱۳ طراحی محتوا

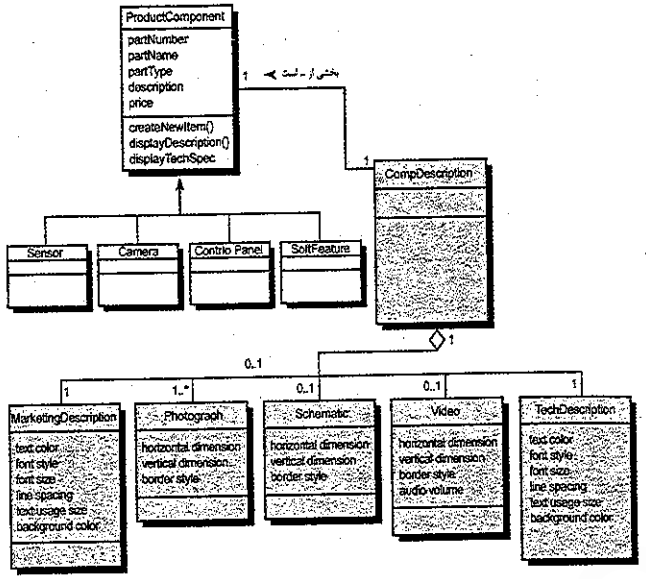
در طراحی محتوا، دو وظیفه طراحی متفاوت کانون توجه قرار می‌گیرد که هر کدام را افرادی با مجموعه مهارت‌های خاص اداره می‌کنند. نخست، یک نمایش طراحی اشیای محتوایی و سازوکارهای لازم برای ایجاد رابطه میان آن‌ها توسعه می‌یابد. به علاوه، اطلاعات درون هر شیء محتوایی خاص ایجاد می‌شود. وظیفه دوم ممکن است توسط نویسندگان مطالب، طراحان گرافیک، سایرین اجرا شود.

۱-۶-۱۳ اشیای محتوایی (Content objects)

رابطه‌ی میان اشیای محتوایی، که به‌عنوان بخشی از مدل خواسته‌ها برای برنامه‌ی تحت وب تعریف می‌شوند، و اشیای طراحی که محتوا را نشان می‌دهند، مشابه با رابطه‌ی میان کلاس‌های تحلیل و مؤلفه‌های طراحی است که در فصل قبل شرح داده شد. درحیطه‌ی طراحی برنامه‌های تحت وب، شیء محتوایی، ارتباط نزدیک‌تری با شیء داده برای نرم‌افزارهای سنتی دارد. شیء محتوایی صفاتی دارد که شامل اطلاعات خاص محتوا (که معمولاً طی مدل‌سازی خواسته‌های برنامه‌ی تحت وب تعریف می‌شوند) و صفات خاص پیاده‌سازی (که به‌عنوان بخشی از طراحی مشخص می‌شوند) می‌شود.

به‌عنوان مثال، کلاس تحلیل **ProductComponent** را در نظر بگیرید که برای سیستم تجارت الکترونیکی *SafeHome* توسعه یافته است. صفت کلاس تحلیل *description* به‌عنوان کلاس تحلیل **CompDescription** نمایش داده می‌شود که از پنج شیء محتوایی تشکیل شده است: **Schematic**

MarketingDescription، **Photograph**، **TechDescription** و **Video** که به‌صورت اشیای هاشور-خورده در شکل ۳-۱۳ نمایش داده شده‌اند. اطلاعات موجود در شیء محتوایی به‌صورت صفات نشان داده می‌شود. برای مثال **Photograph** (یک تصویر jpg) دارای صفاتی مثل *dimension horizontal* و *vertical dimension* و *border style* است.



شکل ۳-۱۳ نمایش طراحی اشیای محتوایی.

از ترکیب و ارتباط اشیا در UML[1] می‌توان برای به نمایش در آوردن روابط میان اشیای محتوایی استفاده کرد. برای مثال، ارتباط UML در شکل ۳-۱۳ نشان می‌دهد که یک **CompDescription** برای هر نمونه از کلاس **ProductComponent** به‌کار می‌رود. **CompDescription** از پنج شیء تشکیل می‌شود. ولی، نمادگذاری چندگانگی (multiplicity) نشان می‌دهد که **Shcematic** و **Video** اختیاری‌اند (صفر رخ داد امکان‌پذیر است)، یک **MarketingDescription** و یک **TechDescription** مورد نیاز است و یک یا چند نمونه از **Photograph** به‌کار برده می‌شود.

۲-۶-۱۳ مسائل طراحی محتوا

هنگامی که همه‌ی اشیای محتوایی مدل‌سازی شدند، اطلاعاتی که قرار است هر شیء تحویل دهد، باید مدیریت شود و سپس طوری فرمت‌بندی شود که با نیازهای مشتری همخوانی داشته باشد. مدیریت محتوا، وظیفه‌ی متخصصان در زمینه‌های مرتبطی است که شیء محتوایی را با فراهم ساختن طرح کلی اطلاعات تحویلی و مشخص کردن انواع کلی اشیای محتوایی (مثلاً متون توصیفی، تصاویر گرافیکی، عکس‌ها) که در تحویل‌دادن اطلاعات به‌کار می‌روند، طراحی می‌کنند. طراحی زیبایی‌شناختی (بخش ۵-۱۳) را نیز می‌توان برای نمایش محتوا به‌کار برد.

> «طراحان خوب می‌توانند از آشفتگی، نظم یافرینده آن‌ها می‌توانند به‌وضوح ایده‌ها را از طریق سازمان‌دهی و دستکاری واژه‌ها و تصاویر منتقل کنند.»
>
> **جفری وین**

---

[1] هر دو این نمایش‌ها در پیوست ۱ بحث شده‌اند.

به‌موازاتی که اشیا طراحی می‌شوند، «دسته‌بندی» می‌شوند [Pow02] تا صفحات برنامه‌ی تحت وب را تشکیل دهند. تعداد اشیای محتوایی گنجانده شده در یک صفحه، تابعی از نیازهای کاربر، قید و بندهای ناشی از سرعت دانلود اتصال اینترنتی و محدودیت‌های ناشی از تحمل کاربر در مقابل جابه‌جایی پنجره مطالب است.

۱۳-۷ طراحی معماری

طراحی معماری، ارتباطی تنگاتنگ با اهداف تعیین شده برای برنامه‌ی تحت وب، محتوایی که قرار است ارائه شود، کاربران بازدید کننده از برنامه‌ی تحت وب و فلسفه تعیین شده برای گشت‌وگذار دارد. شما به‌عنوان طراح معماری باید معماری محتوا و معماری برنامه‌ی تحت وب را تعیین کنید. در *معماری محتوا*[1]، شیوه‌ی ساختاردهی اشیای محتوایی (یا اشیای مرکب نظیر صفحات وب) برای عرضه و گشت‌وگذار، کانون توجه قرار می‌گیرد. *معماری برنامه‌های تحت وب*، به شیوه‌ی ساختاردهی به برنامه کاربردی برای مدیریت تعامل با کاربر، اداره‌ی وظایف پردازش درونی، گشت‌وگذار مؤثر و ارائه‌ی محتوا می‌پردازد.

> «ساختار معماری یک سایت با طراحی خوب همواره در چشم کاربر پنهان و نباید هم باشد.»
>
> تامس پاول

در اکثر موارد، طراحی معماری به موازات طراحی واسط، طراحی زیبایی‌شناختی و طراحی محتوا انجام می‌شود. از آن‌جا که معماری برنامه‌های تحت وب ممکن است تأثیری قوی بر گشت‌وگذار بگذارد، تصمیم‌های گرفته شده طی این کنش طراحی، بر کارهای انجام شده طی طراحی گشت‌وگذار، تأثیر خواهد گذاشت.

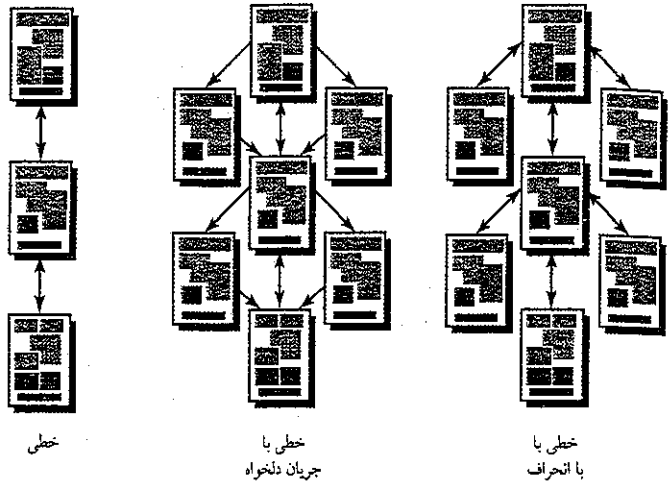
۱۳-۷-۱ معماری محتوا (Content Architecture)

در طراحی معماری محتوا، آن چه کانون توجه قرار می‌گیرد، تعریف ساختار ابر رسانه‌ای کلی برنامه‌ی تحت وب است. گرچه گاهی معماری‌های سفارشی نیز ایجاد می‌شوند، همواره این گزینه را در اختیار دارید که از چهار ساختار محتوایی زیر یکی را برگزینید [Pow00]:

> چه انواعی از معماری‌های محتوا معمولاً مشاهده می‌شوند؟

ساختارهای خطی (Linear Structures، شکل ۴-۱۳) هنگامی مشاهده می‌شوند که دنباله‌ای قابل پیش‌بینی از تعامل‌ها (با قدری تغییرات) متداول باشند. یک مثال کلاسیک می‌تواند نمایشی خودآموز باشد که در آن، صفحات اطلاعات، همراه با تصاویری گرافیکی، تصاویر ویدیویی یا صداهای مرتبط، فقط پس از ارائه‌ی اطلاعات پیش‌نیاز، ارائه می‌شوند. یک مثال دیگر می‌تواند، ترتیب وارد کردن سفارش محصولات باشد که در آن، اطلاعات ویژه باید به ترتیبی ویژه، مشخص شوند. در چنین مواردی، ساختارهای نشان داده شده در شکل ۴-۱۳، مناسب هستند. به موازاتی که محتوا و پردازش پیچیده‌تر می‌شوند، جریان خطی خالص نشان داده شده در سمت چپ شکل، راه را برای ساختارهای خطی پیچیده تری باز می‌کند که در آن‌ها، محتوای متفاوتی را می‌توان فراخوانی کرد یا برای کسب محتویات مکمل باید از یک راه انحرافی استفاده کرد (ساختار نشان داده شده در طرف راست شکل ۴-۱۳).

[1] عبارت *معماری اطلاعات* برای ساختارهایی به‌کار می‌رود که به سازمان‌دهی، نشان‌گذاری، گشت‌وگذار و جستجوی بهتر اشیای محتوایی منجر می‌شود.

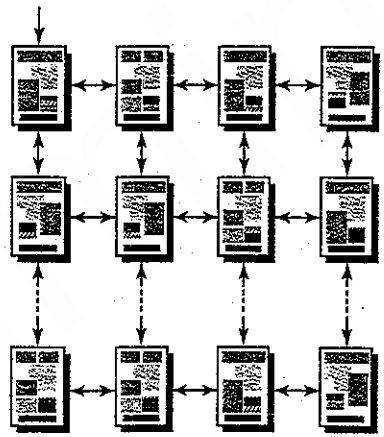


شکل ۴-۱۳ ساختارهای خطی.

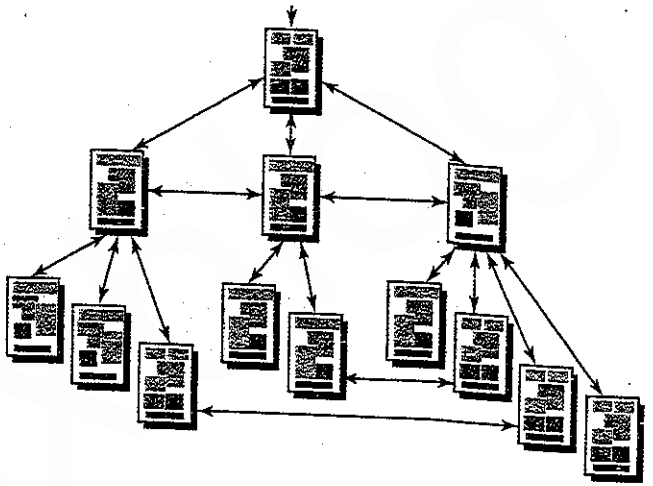
نکته‌ی کلیدی

معماری MVC واسط کاربر را از قابلیت‌های عملیاتی برنامه‌ی تحت وب و محتوای اطلاعاتی آن منفک می‌کند.

ساختارهای مشبک (Grid Structures، شکل ۵-۱۳) گزینه‌های معماری‌اند که می‌توان هنگام سازمان‌دهی محتوای برنامه‌ی تحت وب در دو (یا چند) بُعد به‌کار برد. برای مثال، وضعیتی را در نظر بگیرید که در آن یک سایت تجارت الکترونیکی، چوب گلف می‌فروشد. بُعد افقی شبکه، نوع چوب گلفی را که قرار است فروخته شود (مثلاً چوبی، آهنی، چوگانی) تعیین می‌کند. بعد عمودی نشان‌گر پیشنهادهای مطرح‌شده توسط سازندگان گوناگون چوب گلف است. از این رو، کاربر ممکن است شبکه را به‌طور افقی جستجو کند تا ستون مربوط نوع چوگانی را بیابد و سپس به‌طور عمودی شبکه را بررسی کند تا پیشنهادهای فروشندگان چوگان را بیابد . این برنامه‌ی تحت وب فقط هنگامی مفید واقع می‌شود که محتوای کاملاً منظم در آن ارائه شود [Pow00].

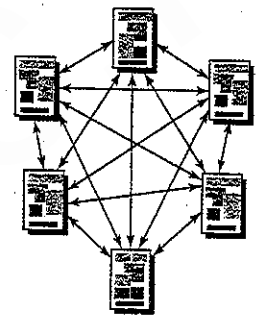


شکل ۵-۱۳ ساختار مشبک.



شکل ۶-۱۳ ساختار سلسله مراتبی.

ساختارهای سلسله مراتبی (Hierarchical Structures، شکل ۶-۱۳) بدون تردید متداول‌ترین معماری برنامه‌های تحت وب به شمار می‌روند. بر خلاف سلسله مراتب افرازشده‌ی نرم‌افزار که در فصل ۹ بحث شد و در آن جریان کنترل تنها در راستای شاخه‌های عمودی تشویق می‌شود، ساختار سلسله مراتبی برنامه‌های تحت وب را می‌توان طوری طراحی کرد که (از طریق انشعاب‌های فوق متنی) جریان افقی از میان شاخه‌های عمودی ساختار را امکان‌پذیر سازند. از این رو، محتوای ارائه‌شده در منتها الیه سمت چپ سلسله مراتب می‌تواند پیوندهای آبَرمتنی داشته باشد که مستقیماً به محتوای موجود در میانه یا شاخه سمت راستی ساختار منتهی می‌شوند. به هر حال، لازم به ذکر است که گرچه با این گونه انشعاب‌ها، گشت‌وگذار در میان محتوای برنامه‌ی تحت وب، سرعت می‌گیرد، می‌تواند برای برخی کاربران به سردرگمی بینجامد.



شکل ۷-۱۳ ساختار شبکه‌ای.

ساختارهای شبکه‌ای یا «وب-خالص» (Networked Structures، شکل ۷-۱۳) مشابه با بسیاری از شیوه‌های معماری است که برای سیستم‌های شیءگرا تکامل پیدا می‌کند. مؤلفه‌های معماری (که در این مورد، صفحات وب هستند) طوری طراحی می‌شوند که ممکن است کنترل را (از طریق پیوندهای فوق متن) به هر مؤلفه‌ی دیگر سیستم تحویل دهد. با این روش، انعطاف‌پذیری در گشت‌وگذار به‌طور چشمگیری امکان‌پذیر می‌شود، ولی در عین حال می‌تواند باعث سردرگمی کاربر هم می‌شود.

ساختارهای معماری بحث شده در پاراگراف‌های قبلی را می‌توان تلفیق کرد و *ساختارهای مرکب* را تشکیل داد. معماری کلی برنامه‌ی تحت وب ممکن است سلسله مراتبی باشد، ولی به‌عنوان بخشی از یک ساختار باشد که ممکن است خصوصیات خطی از خود نشان دهد، در حالی که بخش دیگری از معماری ممکن است شبکه‌ای باشد. هدف شما به‌عنوان طراح معماری، این است که ساختار برنامه‌ی تحت وب را بر محتوایی که قرار است ارائه شود و بر پردازشی که باید انجام شود، مطابقت دهد.

۲-۷-۱۳ معماری برنامه‌های تحت وب

معماری برنامه‌های تحت وب، زیرساختاری را توصیف می‌کند که سیستم یا برنامه کاربردی مبتنی بر وب را قادر می‌سازد تا به اهداف تجاری خود دست پیدا کند. جاسینتو و همکاران [Jac02b] خصوصیات پایه‌ی این زیرساخت را به شیوه زیر توصیف می‌کند:

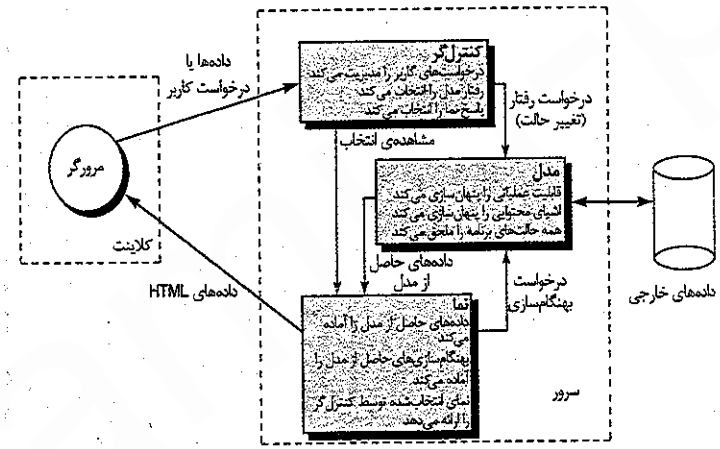
> برنامه‌های کاربردی را باید با استفاده از لایه‌هایی ساخت که در آن‌ها دغدغه‌های متفاوت به حساب آورده می‌شوند؛ به ویژه، داده‌های برنامه کاربردی را باید از محتوای صفحه (گره‌های گشت‌وگذار) جدا کرد و این محتوا، به نوبه خود، باید به‌طور واضح از «حس و ظاهر» واسط جدا شوند.

این نویسندگان یک معماری طراحی سه لایه‌ای پیشنهاد می‌کنند که واسط را از گشت‌وگذار و از رفتار برنامه کاربردی منفک می‌سازد. آن‌ها چنین استدلال می‌کنند که جدا نگه داشتن واسط، برنامه‌ی کاربردی و گشت‌وگذار، پیاده‌سازی را تسهیل می‌کند و استفاده‌ی مجدد را بهبود می‌بخشد.

معماری ***مدل-نما-کنترل‌گر*** (MVC) [Kra88][1] یکی از چند مدل زیرساختی برای برنامه‌های تحت وب است که واسط کاربر را از قابلیت عملیاتی و محتوای اطلاعاتی آن منفک می‌سازد. *مدل* (که گاهی از آن به‌عنوان «شیء مدل» یاد می‌شود) حاوی همه‌ی محتوای خاص برنامه‌ی کاربردی و منطق پردازش، از جمله کلیه اشیای محتوایی، دستیابی به منابع داده‌ای/اطلاعاتی خارجی و کلیه قابلیت‌های عملیاتی پردازشی می‌شود که کاربر نهایی به آن‌ها نیاز دارد. ***کنترل‌گر***، دستیابی به مدل و نما را مدیریت می‌کند و جریان داده‌ها را میان آن‌ها هماهنگ می‌سازد. در یک برنامه‌ی تحت وب، «نما» توسط کنترل‌گر و با داده‌های به‌دست آمده از مدل، بر اساس ورودی کاربر، به‌هنگام می‌شود» [WMT02]. طرحی از معماری MVC در شکل ۸-۱۳ نشان داده شده است.

نکته‌ی کلیدی

معماری MVC واسط کاربر را از قابلیت‌های عملیاتی برنامه‌ی تحت وب و محتوای اطلاعاتی آن منفک می‌کند.



شکل ۸-۱۳ معماری MVC.

[1] شایان ذکر است که MVC در حقیقت یک الگوی طراحی معماری است که برای محیط Smalltalk تهیه شده است (www.cetus-links.org/oo_smalltalk.html) و در هر برنامه کاربردی تعاملی قابل استفاده است.

با رجوع به شکل مشاهده می‌کنید که داده‌ها یا درخواست‌های کاربر توسط کنترل‌گر اداره می‌شوند. کنترل‌گر همچنین اشیای نمایی را که بر اساس درخواست کاربر قابل استفاده‌اند، انتخاب می‌کند. هنگامی که نوع درخواست تعیین شد، یک درخواست رفتار به مدل انتقال داده می‌شود که قابلیت عملیاتی را پیاده‌سازی می‌کند یا محتوای لازم برای پاسخ‌گویی به درخواست را بازیابی می‌کند. شیء مدل می‌تواند به داده‌های ذخیره شده در بانک اطلاعاتی شرکتی، به‌عنوان بخشی از انباره‌ی داده‌های محلی، یا به‌عنوان مجموعه‌ای از فایل‌های مستقل، دستیابی داشته باشد. داده‌های توسعه یافته توسط این مدل باید توسط شیء نمایی مناسب، فرمت‌بندی و سازمان‌دهی شوند و سپس از سرور برنامه کاربردی به مرورگر کلاینت باز پس فرستاده شود تا روی ماشین کلاینت به نمایش در آید.

در بسیاری از موارد، معماری برنامه‌ی تحت وب در حیطه‌ی محیط توسعه‌ای تعیین می‌شود که برنامه قرار است در آن پیاده‌سازی گردد. در صورت علاقه بیشتر به این مطلب، برای بحث درباره محیط‌های توسعه و نقش آن‌ها در طراحی معماری‌های برنامه‌های تحت وب، [Fow03] را ببینید.

۸-۱۳. طراحی گشت‌وگذار

هنگامی که معماری برنامه‌ی تحت وب مشخص گردید و مؤلفه‌ها (صفحات، اسکریپت‌ها، اپلت‌ها، و سایر قابلیت‌های پردازشی) معماری شناسایی شدند، باید مسیرهای گشت‌وگذاری را مشخص کنید که کاربران را قادر می‌سازند تا به محتوا و قابلیت‌های عملیاتی برنامه‌ی تحت وب دستیابی داشته باشند. برای دستیابی به این منظور، باید (۱) معناشناسی گشت‌وگذار را برای کاربران متفاوت سایت مشخص کنید و (۲) مکانیک (نحو) دستیابی به گشت‌وگذار را تعریف کنید.

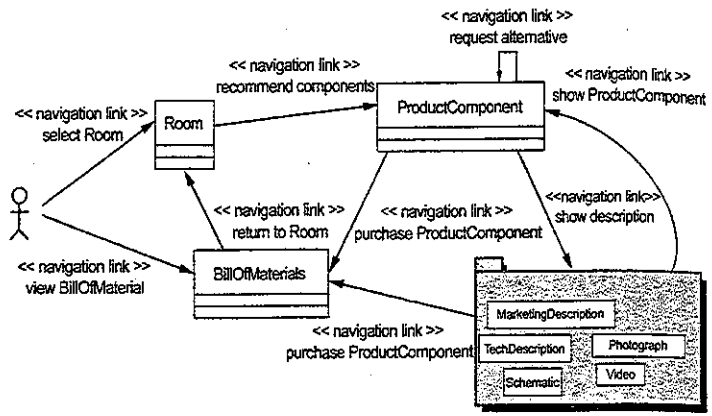
۱-۸-۱۳ معناشناسی گشت‌وگذار (Navigation Semantics)

همانند بسیاری از کنش‌های طراحی برنامه‌های تحت وب، طراحی گشت‌وگذار با در نظر گرفتن سلسله مراتب کاربران و پرونده‌های کاربرد مرتبط (فصل ۵)، که برای هر گروه از کاربران (کنش گران) تهیه شده‌اند، آغاز می‌شود. هر کنش‌گر ممکن است از برنامه‌ی تحت وب با قدری تفاوت نسبت به دیگران استفاده کند و بنابراین، خواسته‌های گشت‌وگذاری او متفاوت باشد. به علاوه، پرونده‌های کاربرد تهیه شده برای هر کنش‌گر، مجموعه‌ای از کلاس‌ها را تعریف خواهد کرد که شامل یک یا چند شیء محتوایی یا قابلیت‌های عملیاتی برنامه‌ی تحت وب است. به موازاتی که کاربر با برنامه‌ی تحت وب تعامل می‌کند، به یک سری *واحدهای معناشناختی گشت‌وگذار* (NSU) بر می‌خورد- «مجموعه‌ای از اطلاعات و ساختارهای گشت‌وگذار مرتبط که با همکاری یکدیگر، زیرمجموعه‌ای از خواسته‌های مرتبط با کاربر را برآورده می‌سازند» [Cac02].

هر NSU از مجموعه‌ای عناصر گشت‌وگذار موسوم به *راه‌های گشت‌وگذار* (WoN) [Gna99] تشکیل می‌شود. یک WoN نشان‌گر بهترین مسیر گشت‌وگذار برای دستیابی به هدف گشت‌وگذار برای نوع خاصی از کاربر است. هر WoN به‌صورت مجموعه‌ای از گره‌های گشت‌وگذار (NN) سازمان‌دهی می‌شود که از طریق پیوندهای گشت‌وگذار با هم در ارتباط هستند. در برخی از موارد، یک پیوند گشت‌وگذار ممکن است خود یک NSU دیگر باشد. بنابراین، ساختار کلی گشت‌وگذار برای یک برنامه‌ی تحت وب را می‌توان به‌صورت سلسله مراتبی از NSUها سازمان‌دهی کرد. برای به نمایش در آوردن نحوه‌ی تهیه‌ی یک NSU، use case **انتخاب مؤلفه‌های *SafeHome*** را در نظر بگیرید:

> «کنترل، کافی است صبر کنم تا ماه در بیاید و خرده‌مان‌هایی که پشت سرمان ریخته، پدیدار شوند؛ آن‌ها راه بازگشت به خانه را نشان می‌دهند.»
>
> **هانسل و گرتل**

> **نکته‌ی کلیدی**
>
> NSU خواسته‌های گشت‌وگذاری را برای هر use case توصیف می‌کند. در اصل، NSU نشان می‌دهد که هر کنش‌گر چگونه بین اشیای محتوایی یا قابلیت‌های عملیاتی برنامه‌ی تحت وب حرکت می‌کند.



شکل ۹-۱۳ ایجاد یک NSU.

use case: انتخاب مؤلفه‌های *SafeHome*

این برنامه‌ی تحت وب، مؤلفه‌های محصول (مثلاً پانل کنترل، حس‌گرها، دوربین‌ها) و سایر ویژگی‌های مربوط به هر اتاق و ورودی بیرونی (مثلاً، قابلیت‌های عملیاتی مبتنی بر PC که در یک نرم‌افزار پیاده‌سازی می‌شوند) را توصیه می‌کند. اگر گزینه‌های دیگری در خواست شود، برنامه‌ی تحت وب آن‌ها را، در صورت وجود، فراهم می‌سازد. من قادر خواهم بود که اطلاعات توصیفی و قیمت را برای هر مؤلفه محصول دریافت کنم. برنامه‌ی تحت وب با انتخاب مؤلفه‌های گوناگون، یک قبض مواد (Bill-of-Materials) ایجاد خواهد کرد و به نمایش در خواهد آورد. من قادر خواهم بود به این «قبض مواد» یک نام بدهم و آن را برای مراجعات بعدی ذخیره کنم (use case ذخیره‌ی پیکربندی را ببینید).

آیتم‌هایی که زیر آن‌ها در use case خط کشیده شده است، نشان‌گر کلاس‌ها و اشیای محتوایی هستند که در یک یا چند NSU گنجانده می‌شوند و مشتری جدید را قادر می‌سازند تا سناریوی توصیف‌شده در پرونده‌ی کاربرد **انتخاب مولفه‌های *SafeHome*** را اجرا کند.

شکل ۹-۱۳، یک تحلیل معناشناختی جزئی، گشت‌وگذاری را به تصویر می‌کشد که از use case **انتخاب مؤلفه‌های *SafeHome*** قابل استنباط است. با به‌کارگیری اصطلاح‌شناسی‌ای که قبلاً معرفی شد، این شکل همچنین یک راه گشت‌وگذار (WoN) برای برنامه‌ی تحت وب **SafeHomeAssured.com** نیز نمایش می‌دهد. کلاس‌های مهم دامنه مسأله‌ی همراه با اشیای محتوایی انتخاب شده نشان داده شده‌اند (که در این مورد، پکیج اشیای محتوایی به نام **CompDescription** است که از صفحه‌های کلاس **ProductComponent** به شمار می‌رود). این آیتم‌ها گره‌های گشت‌وگذاری هستند. هر کدام از پیکان‌ها نشان‌گر یک پیوند گشت‌وگذار[1] بوده با کنشی نشان‌گذاری می‌شود که رخ‌دادن آن پیوند را باعث می‌شود.

برای هر use case مرتبط با نقش کاربر، می‌توانید یک NSU ایجاد کنید. برای مثال، یک مشتری جدید برای **SafeHomeAssured.com** ممکن است سه use case متفاوت داشته باشد که همه‌ی آن‌ها به‌دستیابی به اطلاعات و قابلیت‌های عملیاتی متفاوت برنامه‌ی تحت وب می‌انجامند. برای هر هدف یک NSU ایجاد می‌شود.

[1] این پیوندها را گاهی پیوندهای معناشناختی گشت‌وگذار (NSL) نیز می‌نامند [Cac02].

> «مسأله‌ی گشت‌وگذار در وب‌سایت‌ها، مسأله‌ای فنی، فضایی، فلسفی و منطقی است. در نتیجه، برای حل آن باید به ترکیبی بداهه‌پردازانه از هنر، علم و روان‌شناسی سازمانی متوسل شد.»
>
> **تیم هورگان**

طی مراحل اولیه طراحی گشت‌وگذار، معماری محتوای برنامه‌ی تحت وب، ارزیابی می‌شود تا یک یا چند WoN برای هر use case تعیین شود. چنان‌که پیش‌تر نیز گفته شد، یک WoN، گره‌های گشت‌وگذار (مثلاً محتوا) و سپس پیوندهایی را مشخص می‌کند که گشت‌وگذار میان این گره‌ها را میسر می‌سازند. WoNها در قالب چند NSU سازمان‌دهی می‌شوند.

۱۳-۸-۲ قالب نحوی گشت‌وگذار (Navigation Syntax)

به موازاتی، که طراحی ادامه پیدا می‌کند، وظیفه بعدی شما تعریف مکانیک گشت‌وگذارهاست. در توسعه بخشیدن به روش پیاده‌سازی هر NSU، چند گزینه پیش روی شماست:

- *پیوند گشت‌وگذار انفرادی*- شامل پیوندهای متنی، آیکون‌ها، دکمه‌ها و کلیدها و استعاره‌های گرافیکی می‌شود. باید پیوندهایی برای گشت‌وگذار برگزینید که با محتوای مناسبت داشته باشند و به طراحی واسطی با کیفیت بالا بینجامد.
- *نوار افقی گشت‌وگذار*- محتوا یا گروه‌های عملیاتی اصلی را در یک نوار افقی حاوی پیوندهای مناسب فهرست می‌کند. به‌طور کلی، بین چهار تا هفت گروه فهرست می‌شود.
- *ستون عمودی گشت‌وگذار*- (۱) محتوا یا گروه‌های عملیاتی اصلی را فهرست می‌کند یا (۲) همه‌ی اشیای محتوایی اصلی موجود در برنامه‌ی تحت وب را به‌طور عمودی فهرست می‌کند. اگر گزینه دوم را انتخاب کنید، چنین ستون‌های گشت‌وگذاری می‌توانند «بسط» پیدا کنند و اشیای محتوایی را به‌عنوان بخشی از یک سلسله مراتب عرضه کنند (یعنی انتخاب یک مدخل از ستون اولیه باعث می‌شود یک فهرست دوم باز شود که سطح دومی از اشیای محتوایی مرتبط را به نمایش در آورد).
- *برگه ها* (Tabs)- استعاره‌ای که چیزی نیست جز شکل تغییریافته‌ی ستون عمودی یا افقی گشت‌وگذار، که محتوا یا گروه‌های عملیاتی را به‌صورت برگه‌هایی نشان می‌دهد، و درصورت نیاز به یک پیوند، انتخاب می‌شود.
- *نقشه‌ی سایت*- جدولی همه‌جانبه از محتوا را برای گشت‌وگذار در تمامی اشیای محتوایی و قابلیت‌های عملیاتی موجود در برنامه‌ی تحت وب فراهم می‌آورد. علاوه بر انتخاب مکانیک گشت‌وگذار، باید قراردادها و کمک‌های مناسب برای شیوه‌ی گشت‌وگذار را نیز ارائه دهید. برای مثال، با مورب کردن لبه‌های آیکون‌ها و پیوندهای گرافیکی، ظاهری سه بعدی به آن‌ها می‌دهید که نشان می‌دهد این‌ها «قابل کلیک کردن» هستند. باید بازخوردهای سمعی یا بصری طراحی شوند تا به‌کاربر خاطرنشان کنند که گزینه‌ای برای گشت‌وگذار انتخاب شده است. برای گشت‌وگذارهای متنی باید از رنگ استفاده شود تا پیوندهای گشت‌وگذار مشخص شوند و پیوندهایی که تا کنون طی شده‌اند، مشخص گردند. این‌ها تنها چند مورد از ده‌ها قرارداد طراحی هستند که گشت‌وگذار را «کاربرپسند» می‌سازند.

اندرز

در اکثر شرایط، تنها یکی از دو سازوکار گشت‌وگذار افقی یا عمودی را انتخاب کنید نه هر دو آن‌ها را.

اندرز

نقشه‌ی سایت باید از هر صفحه قابل دستیابی باشد. خود نقشه باید طوری سازمان‌دهی شود که ساختار اطلاعات برنامه‌ی تحت وب به آسانی پدیدار باشد.

۱۳-۹ طراحی در سطح مؤلفه‌ها

برنامه‌های تحت وب مدرن، قابلیت‌هایی ارائه می‌دهند که پیوسته بر پیچیدگی آن‌ها افزوده می‌شود و (۱) پردازش‌های محلی را برای ایجاد محتوا و قابلیت‌های گشت‌وگذاری را به شیوه‌ای پویا انجام می‌دهند، (۲) توانایی پردازش و انجام محاسبات مناسب را برای دامنه تجاری برنامه‌ی تحت وب

فراهم می‌سازند، (۳) درخواست از بانک‌های اطلاعاتی پیچیده و دستیابی به آن‌ها را فراهم می‌آورند و (۴) واسط‌های داده‌ای را میان سیستم‌های شرکتی خارجی برقرار می‌کنند. برای دستیابی به این قابلیت‌ها (و قابلیت‌های دیگر) باید مؤلفه‌هایی برای برنامه طراحی کنید و بسازید که از نظر شکلی، همسان با مؤلفه‌های نرم‌افزاری در نرم‌افزارهای تجاری است.

روش‌های طراحی بحث شده در فصل ۱۰ با قدری اصلاح (در صورت نیاز) برای مؤلفه‌های برنامه‌ی تحت وب نیز کاربرد دارند. محیط پیاده سازی، زبان‌های برنامه‌نویسی و الگوهای طراحی، چارچوب‌های طراحی و نرم‌افزار، ممکن است قدری تغییر کنند، ولی رویکرد کلی طراحی تغییر چندانی نخواهد کرد.

۱۳-۱۰ طراحی ابر رسانه‌ها به روش شیءگرا (OOHDM)

در دهه‌ی گذشته چند روش طراحی برای برنامه‌های تحت وب پیشنهاد شده است. تاکنون، هیچ روشی به تنهایی پیروز میدان نبوده است.[1] در این بخش به ارائه‌ی مروری مختصر بر یکی از روش‌های طراحی برنامه‌ی تحت وب خواهیم پرداخت که بحث فراوان درباره آن شده است- OOHDM.

دانیل شوابه و همکاران وی [Sch95,Sch98b] برای اولین بار، طراحی ابر رسانه‌ها به روش شیءگرا (OOHDM) را پیشنهاد کردند که از چهار فعالیت طراحی متفاوت تشکیل می‌شود: طراحی مفهومی، طراحی گشت‌وگذارها، طراحی واسط انتزاعی و پیاده‌سازی. خلاصه‌ای از این فعالیت‌های طراحی در شکل ۱۳-۱۰ نشان داده شده است که به اختصار در بخش‌های بعدی بحث خواهد شد.

شکل ۱۳-۱۰ خلاصه روش OOHDM.

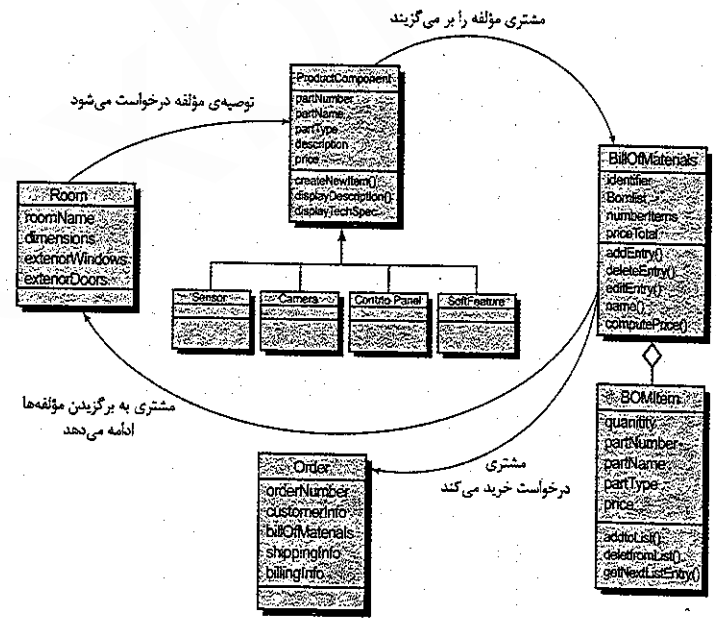
۱۳-۱۰-۱ طراحی مفهومی برای OOHDM

طراحی مفهومی در OOHDM، نمایشی از زیر سیستم‌ها، کلاس‌ها و روابط را ایجاد می‌کند که دامنه‌ی کاربرد را برای برنامه‌ی تحت وب تعریف می‌کند. برای ایجاد نمودارهای کلاس مناسب، نمایش‌های

[1] در واقع، تعداد نسبتاً کمی از طراحان وب، هنگام طراحی یک برنامه تحت وب از روشی مشخص استفاده می‌کنند. امید می‌رود که این روش طراحی خاص با گذر زمان تغییر کند.

کلاس‌های مرکب، نمودارهای همکاری و سایر اطلاعات توصیف‌گر دامنه کاربرد، می‌توان از UML استفاده کرد.[1]

به‌عنوان مثال ساده‌ای از طراحی مفهومی OOHDM، برنامه کاربردی تجارت الکترونیک **SafeHomeAssured.com** را در نظر بگیرید. در شکل ۱۱-۱۳، بخشی از یک «طرح مفهومی» نشان داده شده است. نمودارهای کلاس، کلاس‌های مرکب و اطلاعات وابسته که به‌عنوان بخشی از تحلیل برنامه‌ی تحت وب تهیه می‌شوند، طی طراحی مفهومی، مورد استفاده‌ی مجدد قرار می‌گیرند تا روابط میان کلاس‌ها را به نمایش در آورند.



شکل ۱۱-۱۳ بخشی از طرح مفهومی مربوط به SafeHomeAssured.com.

۲-۱۰-۱۳ طراحی امکانات گشت‌وگذار برای OOHDM

در *طراحی امکانات گشت‌وگذار*، مجموعه‌ای از «اشیا» تعریف می‌شوند که از کلاس‌های تعریف شده در طراحی مفهومی به‌دست می‌آیند. یک سری «کلاس‌های گشت‌وگذاری» یا «گره ها» تعریف می‌شوند تا این اشیا را پنهان‌سازی کنند. برای ایجاد پرونده‌های کاربرد مناسب، نمودارهای حالت و نمودارهای ترتیب- همه‌ی نمایش‌هایی که شما را در فهم بهتر خواسته‌های گشت‌وگذار کمک می‌کنند- از UML می‌توان استفاده کرد. به علاوه، در اثنای پیشرفت طراحی نیز می‌توان از الگوهای طراحی برای طراحی گشت‌وگذارها استفاده کرد. OOHDM از یک مجموعه کلاس‌های گشت‌وگذار

[1] OOHDM نمادگذاری خاصی را تجویز نمی‌کند؛ به هرحال، استفاده از UML هنگام به‌کارگیری این روش رایج است.

ازپیش‌تعریف‌شده- گره‌ها، پیوندها، لنگرها (anchors) و ساختارهای دستیابی [Sch98]- استفاده می‌کند. در ساختارهای دستیابی، جزئیات بیشتری تعیین شده است و شامل سازوکارهایی نظیر نمایه‌ی برنامه‌ی تحت وب، نقشه سایت یا راهنمای تور می‌شوند.

هنگامی که کلاس‌های گشت‌وگذار تعریف شدند، OOHDM «با گروه‌بندی اشیای گشت‌وگذار در مجموعه‌هایی به نام «حیطه» به فضای گشت‌وگذار، ساختار می‌دهد.» [Sch98b]. هر حیطه شامل توصیفی از ساختار گشت‌وگذار محلی، محدودیت‌های ناشی از دستیابی اشیای محتوایی و متدها (عملیات‌های) مورد نیاز برای دستیابی به اشیای محتوایی می‌شود. قالب حیطه‌ای (مشابه با کارت‌های CRC که در فصل ۶ بحث شد) تهیه می‌شود و می‌توان از آن‌ها برای دنبال کردن خواسته‌های گشت‌وگذاری هر گروه از کاربران طی حیطه‌های گوناگون تعریف شده در OOHDM می‌توان از آن‌ها بهره برد. به این ترتیب، مسیرهای گشت‌وگذار مشخص (آن چه که در بخش ۱-۸-۱۳، WoN نامیدیم) ظهور می‌یابند.

۳-۱۰-۱۳ طراحی و پیاده‌سازی واسط انتزاعی

در کنش *طراحی واسط انتزاعی*، اشیای واسطی مشخص می‌شود که کاربر در رخ‌دادن تعامل با برنامه‌ی تحت وب می‌بیند. مدل رسمی از اشیای واسط، که *نمای داده‌های انتزاعی* (ADV) خوانده می‌شود، برای به نمایش در آوردن ارتباط میان اشیای واسط و اشیای گشت‌وگذار و خصوصیات رفتاری اشیای واسط به‌کار گرفته می‌شود.

مدل ADV یک «چیدمان ایستا» [Sch98b] تعریف می‌کند که استعاره واسط را به نمایش در می‌آورد و شامل نمایشی از اشیای گشت‌وگذار در داخل واسط و مشخص کردن اشیای واسط (مثلاً منوها، دکمه‌ها، آیکون‌ها) می‌شود که به گشت‌وگذار و تعامل کمک می‌کنند. به علاوه، مدل ADV حاوی یک مؤلفه رفتاری (مشابه با نمودار حالت UML) است که نشان می‌دهد رویدادهای خارجی چگونه «شروع گشت‌وگذار را رقم می‌زنند و هنگامی که کاربر با برنامه تعامل دارد، کدام تبدیلات واسط رخ می‌دهد.» [Sch01a].

فعالیت پیاده‌سازی OOHDM نشان‌گر یک تعامل طراحی است که خاص محیطی است که برنامه‌ی تحت وب در آن پیاده‌سازی می‌شود. کلاس‌ها، گشت‌وگذار و واسط، هر کدام به شیوه‌ای تعیین می‌شوند که بتوان آن‌ها را برای محیط کلاینت-سرور، سیستم‌های عامل، نرم‌افزار پشتیبان، زبان‌های برنامه‌نویسی و سایر خصوصیات محیطی مرتبط با مسأله ایجاد کرد.

۱۱-۱۳ خلاصه

کیفیت یک برنامه‌ی تحت وب- که بر حسب قابلیت استفاده، قابلیت عملیاتی، قابلیت اطمینان، بازدهی، قابلیت نگهداری، امنیت، گسترش‌پذیری و زمان ارائه به بازار تعریف می‌شود- در مرحله‌ی طراحی وارد می‌شود. برای دستیابی به این صفات کیفیتی، طراحی خوب برنامه‌ی تحت وب باید خصوصیات زیر را از خود نشان دهد: سادگی، سازگاری، هویت، استحکام، قابلیت گشت‌وگذار و جاذبه‌ی بصری. برای دستیابی به این خصوصیت‌ها، در فعالیت طراحی برنامه‌ی تحت وب، شش عنصر طراحی متفاوت، کانون توجه قرار می‌گیرند.

طراحی واسط، ساختار و سازمان‌دهی واسط کاربر را توصیف می‌کند و شامل نمایشی از چیدمان صفحه نمایش، تعریف شیوه‌های تعامل و توصیف سازوکارهای گشت‌وگذار می‌شود. مجموعه‌ای از اصول طراحی واسط و جریان کاری طراحی واسط، شما را هنگام طراحی سازوکارهای کنترل واسط و چیدمان یاری می‌دهد.

طراحی زیبایی‌شناختی، که طراحی گرافیکی نیز نامیده می‌شود، «شکل و شمایل» برنامه‌ی تحت وب را توصیف می‌کند و شامل الگوهای رنگ؛ چیدمان هندسی؛ اندازه، نوع فونت و محل قرار گرفتن متن؛ استفاده از تصاویر گرافیکی؛ و ابعاد زیبایی‌شناختی مربوط می‌شود. مجموعه‌ای از دستورالعمل‌های طراحی گرافیکی، اساس یک رویکرد طراحی را فراهم می‌سازند.

طراحی محتوا، چیدمان، ساختار و طرح‌بندی همه‌ی محتوای ارائه شده به‌عنوان بخشی از برنامه‌ی تحت وب را تعریف می‌کند و رابطه میان اشیای محتوایی را تعیین می‌کند. طراحی محتوا با نمایش اشیای محتوایی، روابط و همبستگی‌های میان آن‌ها آغاز می‌شود. مجموعه‌ای از اصول مرورگری، مبنای طراحی گشت‌وگذار را تعیین می‌کند.

طراحی معماری، ساختار ابر رسانه‌ای کلی را برای برنامه‌ی تحت وب مشخص می‌کند و شامل دو بخش یعنی معماری محتوا و معماری برنامه‌ی تحت وب می‌شود. سبک‌های معماری برای محتوا شامل ساختارهای خطی، مشبک، سلسله مراتبی و شبکه‌ای می‌شود. معماری برنامه‌ی تحت وب، زیرساختی را توصیف می‌کند که به برنامه‌ی تحت وب امکان دستیابی به اهداف مورد نظرش را می‌دهد.

طراحی گشت‌وگذار، جریان گشت‌وگذار میان اشیای محتوایی و برای کلیه قابلیت‌های عملیاتی برنامه‌ی تحت وب را نمایش می‌دهد. معناشناسی گشت‌وگذار با توصیف مجموعه‌ای از واحدهای معناشناختی گشت‌وگذار تعریف می‌شود. هر واحد از راه‌های گشت‌وگذار و پیوندها و گره‌های گشت‌وگذاری تشکیل می‌شود. قالب نحوی گشت‌وگذار، سازوکارهای به‌کار رفته برای انجام‌پذیر ساختن گشت‌وگذاری را به تصویر می‌کشد که به‌عنوان بخشی از معناشناسی توصیف می‌شود.

طراحی مؤلفه‌ها، جزئیات منطق پردازش مورد نیاز برای پیاده‌سازی مؤلفه‌هایی را توسعه می‌دهد که یک قابلیت عملیاتی کامل را در برنامه‌ی تحت وب پیاده‌سازی می‌کنند. تکنیک‌های توصیف شده در فصل ۱۰ برای مهندسی مولفه‌های برنامه‌ی تحت وب قابل استفاده‌اند.

طراحی ابر رسانه‌ای به روش شیءگرا (OOHDM) یکی از چند روش پیشنهاد شده برای طراحی برنامه‌ی تحت وب است. OOHDM یک فرایند طراحی پیشنهاد می‌کند که شامل طراحی مفهومی، طراحی گشت‌وگذاری، طراحی واسط انتزاعی و پیاده‌سازی می‌شود.

مسائل و نکاتی برای تعمق

۱-۱۳ چرا «ایده‌آل هنرمندانه» در ساخت برنامه‌های تحت وب مدرن، فلسفه طراحی کافی به شمار نمی‌رود؟ آیا موردی وجود دارد که در آن، ایده‌آل هنرمندانه فلسفه‌ای باشد که باید دنبال کرد؟

۲-۱۳ در این فصل، آرایه‌ی گسترده‌ای از صفات کیفیتی را برای برنامه‌های تحت وب انتخاب کردیم. سه موردی را که فکر می‌کنید بیشترین اهمیت را دارند، انتخاب کنید و استدلال کنید که چرا در کار طراحی برنامه‌های تحت وب باید بر هر کدام از آن‌ها تأکید داشت.

۳-۱۳ دست کم پنج پرسش اضافی برای طراحی برنامه‌ی تحت وب- چک‌لیست کیفیتی ارائه شده در بخش ۱۳-۲ اضافه کنید.

۴-۱۳ شما طراح وب شرکت آموزش آینده هستید که کار آن آموزش از راه دور است. می‌خواهید یک «دستگاه آموزشی» اینترنتی پیاده‌سازی کنید که شما را قادر به ارائه محتوای آموزشی یک دوره به دانشجو کند. این دستگاه آموزشی، زیرساخت اساسی را برای تحویل محتوای آموزشی در هر موضوع دلخواه فراهم می‌آورد (طراحی محتوا، محتوای مناسب را تأمین خواهند کرد). نمونه اولیه‌ای برای طراحی واسط این دستگاه تهیه کنید.

۵-۱۳ از وب‌سایت‌هایی که تاکنون دیده‌اید، کدام یک را از نظر زیبایی‌شناسی از همه دلپذیرتر یافته‌اید؟ چرا؟

۶-۱۳ شیء محتوایی Order را در نظر بگیرید که وقتی تولید می‌شود که کاربر برنامه‌ی تحت وب SafeHomeAssured.com انتخاب همه‌ی مؤلفه‌ها را کامل می‌کند و آماده‌ی نهایی کردن خرید خود است. یک توصیف UML برای Order همراه با نمایش‌های طراحی مناسب تهیه کنید.

۷-۱۳ اختلاف میان معماری محتوا و معماری برنامه‌ی تحت وب در چیست؟

۸-۱۳ با در نظر گرفتن دوباره‌ی شرکت آموزش آینده و «دستگاه آموزشی» آن که در مسأله ۴-۱۳ شرح داده شد، یک معماری محتوا انتخاب کنید که برای این برنامه‌ی تحت وب مناسب باشد. درباره مبنای انتخاب خود بحث کنید.

۹-۱۳ با استفاده از UML، سه یا چهار نمایش طراحی برای اشیای محتوایی تهیه کنید که در حین طراحی «دستگاه آموزشی» توصیف شده در مسأله ۴-۱۳ با آن‌ها مواجه می‌شوید.

۱۰-۱۳ قدری تحقیق اضافی روی معماری MVC انجام دهید و تصمیم بگیرید آیا برای «دستگاه آموزشی» بحث شده در مسأله ۴-۱۳ معماری مناسبی به شمار می‌رود یا خیر.

۱۱-۱۳ اختلاف میان قالب نحوی گشت‌وگذار و معناشناسی گشت‌وگذار در چیست؟

۱۲-۱۳ دو یا سه NSU برای برنامه‌ی تحت وب SafeHomeAssured.com تعریف کنید. هر کدام را به تفصیل شرح دهید.

۱۳-۱۳ مقاله مختصری درباره طراحی ابر رسانه‌ها به روشی غیر از روش شیءگرا بنویسید.

بخش سوم

مدیریت کیفیت

در این بخش از کتاب، مطالبی درباره اصول، مفاهیم و تکنیک‌های به‌کاررفته در مدیریت و کنترل کیفیت نرم‌افزار خواهید آموخت.

در فصل‌های آینده به این پرسش‌ها خواهیم پرداخت:

- خصوصیات کلی نرم‌افزار با کیفیت بالا کدام است؟
- کیفیت را چگونه مرور می‌کنیم و مرورهای مؤثر چگونه انجام می‌شود؟
- تضمین کیفیت نرم‌افزار چیست؟
- چه راهبردهایی برای آزمایش نرم‌افزارها قابل استفاده است؟
- در طراحی موارد آزمون مؤثر از چه روش‌هایی استفاده می‌شود؟
- آیا روش‌های واقع بینانه‌ای وجود دارند که ما را از صحت نرم‌افزار مطمئن سازند؟
- چگونه می‌توانیم تغییراتی را که همواره به هنگام ساخته شدن نرم‌افزار رخ می‌دهند، اداره و کنترل کنیم.
- چه موازین و معیارهایی را می‌توان در ارزیابی کیفیت مدل خواسته‌ها و مدل طراحی، کد منبع و موارد آزمون به‌کار برد؟

هنگامی که به این پرسش‌ها پاسخ گفته شد، بهتر می‌توانید اطمینان حاصل کنید که نرم‌افزار با کیفیت بالا تولید شده است.

فصل ۱۴

مفاهیم کیفیتی

نگاهی گذرا

مفاهیم کیفیت چیست؟ پاسخ به این پرسش به آن آسانی که تصور می‌کنید، نیست. شما کیفیت را وقتی که می‌بینید، می‌شناسید و در عین حال، یک چیزی است که به‌درستی نمی‌توان آن را تعریف کرد. ولی برای نرم‌افزارهای کامپیوتری، کیفیت چیزی است که باید آن را تعریف کنیم و این کاری است که در این فصل خواهیم کرد.

چه کسی آن را انجام می‌دهد؟ هر کسی- مهندسان نرم‌افزار، مدیران، همه‌ی طرف‌های ذی‌نفع- که در فرایند نرم‌افزار دخیل هستند، مسؤول کیفیت هستند.

چرا اهمیت دارد؟ می‌توانید کار را درست انجام دهید یا این که آن را بارها تکرار کنید. اگر تیم نرم‌افزاری در تمامی فعالیت‌های مهندسی نرم‌افزار بر کیفیت تأکید ورزد، مقدار دوباره کاری‌ها کاهش می‌یابد. این منجر به کاهش هزینه‌ها و، مهم‌تر از آن، بهبود زمان ارائه به بازار می‌شود.

مراحل کار کدام است؟ برای دستیابی به نرم‌افزارهای با کیفیت بالا، چهار فعالیت باید رخ دهد: فرایند و کار مهندسی نرم‌افزار اثبات شده، مدیریت منسجم پروژه، کنترل فراگیر کیفیت و وجود زیرساخت برای تضمین کیفیت.

محصول کار چیست؟ نرم‌افزاری که نیازهای مشتری اش را برآورده می‌سازد، به‌طور صحیح و با قابلیت اطمینان کار می‌کند و برای کسانی که از آن استفاده می‌کنند، ایجاد ارزش می‌کند.

چطور اطمینان حاصل کنم که درست از عهده کارها برآمده‌ام؟ کیفیت را با بررسی نتایج همه‌ی فعالیت‌های کنترل کیفیت دنبال کنید و کیفیت را با بررسی خطاها قبل از تحویل و نقایص وارد شده در میدان اندازه‌گیری کنید.

هشدار برای بهبود بخشیدن به کیفیت نرم‌افزار رو به فزونی نهاده است، زیرا استفاده از آن‌ها در تمامی شؤونات زندگی ما متداول شده است. تا دهه ۱۹۹۰ شرکت‌های عظیم به این نتیجه رسیده بودند که سالانه میلیاردها دلار بیهوده صرف نرم‌افزارهایی می‌شود که ویژگی‌ها و قابلیت‌های وعده داده شده را تحویل نمی‌دهند. بدتر از آن، هم دولت و هم صنایع، به‌طور فزاینده‌ای نگران این موضوع بودند که یک خطای بزرگ نرم‌افزاری ممکن است به فلج شدن زیرساختی مهم بینجامد که باعث میلیاردها دلار ضرر و زیان شود. در اوایل قرن حاضر، مجله CIO [Lev01] این عنوان را چاپ کرد که «به ۷۸ میلیارد دلار ضرر و زیان سالانه پایان دهیم!» و این واقعیت تلخ را مطرح کرد که «شرکت‌های تجاری در امریکا میلیاردها دلار صرف نرم‌افزارهایی می‌کنند که آن‌چه را که قرار است انجام دهند، انجام نمی‌دهند.» Information Week [Ric01] نیز همین نگرانی را منعکس کرد:

بنا به اظهار نظر شرکت Standish Group، به رغم نیات خوب، کدنویسی ناقص همچنان به‌عنوان هیولای صنعت نرم‌افزار مطرح است و 45% از اتلاف وقت‌ها را باعث می‌شود و سالانه هزینه‌ای حدود صد میلیارد دلار را متوجه شرکت‌های امریکایی می‌کند. این عدد شامل هزینه‌ی از دست‌دادن مشتریان ناراضی نمی‌شود. از آن‌جا که فروشگاه‌های IT برنامه‌هایی می‌نویسند که بر نرم‌افزارهای زیرساختی پکیج شده تکیه دارند، کدنویسی بد می‌تواند باعث تخریب برنامه‌های کاربردی سفارشی نیز بشود...

نرم‌افزار بد چقدر می‌تواند بد باشد؟ تعاریف، متغیر است، ولی کارشناسان می‌گویند که کافی است تنها سه یا چهار نقص در هزار خط کد وجود داشته باشد تا برنامه ضعیف عمل کند. این نکته را در نظر داشته باشید که اکثر برنامه نویسان به‌ازای هر ۱۰ خط برنامه که می‌نویسند یک خطا وارد آن می‌کنند، آن را در میلیون‌ها خط کدی که در بسیاری از محصولات تجاری نوشته می‌شود، به حساب بیاورید و خواهید دید که سازندگان نرم‌افزارها حداقل نیمی از بودجه‌های توسعه‌ی نرم‌افزار را صرف بر طرف ساختن خطاها به هنگام آزمون می‌کنند. قضیه روشن است؟

در سال ۲۰۰۵، Computer World [Hil05] چنین گزارش کرد که «نرم‌افزارهای بد تقریباً هر سازمانی را که از کامپیوتر استفاده می‌کند به آشوب می‌کشد و باعث می‌شود در مدت زمانی که کامپیوترها از کار می‌افتند، ساعت‌ها وقت کارمندان هدر رود، داده‌ها از بین برود یا مخدوش شود، فرصت‌های فروش از بین برود، هزینه‌های گزاف صرف نگهداری و پشتیبانی بخش IT شود و از رضایت مشتری کاسته شود. یک سال بعد، Infoworld [Fos06] درباره «وضعیت اسف‌بار کیفیت نرم‌افزار» نوشت و گزارش کرد که مسأله‌ی کیفیت اصلاً بهبود نیافته است.

امروزه، کیفیت نرم‌افزار همچنان به عنوان یک مشکل باقی است، ولی چه کسی را باید ملامت کرد؟ سازندگان، مشتریان (و سایر ذی‌نفع‌ها) را ملامت می‌کنند و استدلال آن‌ها هم این است که تاریخ تحویل‌های غیر موجه و جریان پیوسته تغییرات، آن‌ها را وادار می‌سازد تا نرم‌افزار را پیش از اعتبارسنجی کامل تحویل دهند. حق با کیست؟ هر دو- و مسأله همین است. در این فصل، به کیفیت نرم‌افزار به‌عنوان یک مفهوم خواهیم پرداخت و بررسی خواهیم کرد که چرا هرگاه کار مهندسی نرم‌افزار انجام می‌شود، باید آن را به‌طور جدی مد نظر قرار داد.

۱-۱۴ کیفیت چیست؟

رابرت پرسینگ در کتاب رمزآلود خود با عنوان «زن و هنر نگهداری موتور سیکلت» [Per74] درباره چیزی که آن را کیفیت می‌نامیم، چنین توضیح می‌دهد:

کیفیت... می‌دانید چیست، ولی در عین حال نمی‌دانید چیست. ولی بعضی چیزها بهتر از بقیه‌اند؛ یعنی، کیفیت آن‌ها بیشتر است. ولی هنگامی که سعی می‌کنید بگویید کیفیت چیست، جدا از چیزهایی که آن را دارند، همه چیز خراب می‌شود! چیزی وجود ندارد که بتوان درباره اش حرف زد. ولی اگر نتوانید بگویید کیفیت چیست، پس چطور می‌فهمید که چیست یا حتی چطور می‌دانید که وجود دارد؟ ولی برای همه‌ی اهداف عملی، واقعاً وجود دارد. نمرات بر اساس چه چیز دیگری داده می‌شوند؟ به چه دلیل دیگری مردم به یک چیزهایی اقبال نشان می‌دهند و چیزهای دیگری را روانه خاکروبه می‌کنند؟ بدیهی است یک چیزهایی از بقیه بهترند... ولی این بهتر بودن، چیست؟... خلاصه این چرخه همچنان ادامه دارد و راه به جایی نمی‌برد. این کیفیت، چیست؟

حقیقتاً، کیفیت چیست؟

دیوید گاروین [Gar84] از دانشکده تجاری هاروارد در سطحی عمل گرایانه‌تر معتقد است که «کیفیت، مفهومی پیچیده و چند وجهی است» که از پنج دیدگاه متفاوت قابل توصیف است. *دیدگاه متعالی* (همانند پرسینگ) استدلال می‌کند که کیفیت چیزی است که بلافاصله آن را تشخیص می‌دهید، ولی نمی‌توانید به صراحت آن را تعریف کنید. در *دیدگاه کاربری*، کیفیت، بر حسب اهداف خاص کاربر نهایی دیده می‌شود. اگر محصولی این اهداف را برآورده سازد، از خود کیفیت نشان می‌دهد. در *دیدگاه سازندگان*، کیفیت بر حسب مشخصات اولیه محصول تعریف می‌شود. اگر محصول با مشخصات تعیین شده مطابقت داشته باشد، از خود کیفیت نشان می‌دهد. در *دیدگاه محصولی*، اعتقاد بر این است که کیفیت را می‌توان به خصوصیات ذاتی (از قبیل قابلیت‌ها و ویژگی‌های) محصول ربط داد. سرانجام، در *دیدگاه ارزش محور*، کیفیت بر اساس میزان پولی سنجیده می‌شود که مشتری حاضر به پرداخت برای محصول است. در حقیقت، کیفیت شامل همه‌ی این دیدگاه‌ها و حتی بیشتر از آن می‌شود.

اندرز
شیوه‌های متفاوت نگرش به کیفیت را برشمرید.

کیفیت طراحی به خصوصیاتی اشاره دارد که طراحان برای محصول مشخص می‌کنند. درجه‌ی مصالح، تولرانس و مشخصات کارایی، همگی در کیفیت طراحی سهم دارند. به موازاتی که مصالح درجه بالاتر استفاده می‌شود، تولرانس‌های دقیق‌تر و سطوح کارایی بالاتری مشخص می‌شود، کیفیت طراحی محصول افزایش می‌یابد مشروط بر آن که محصول مطابق با مشخصات ساخته شود.

در توسعه نرم‌افزارها، کیفیت طراحی میزانی از دیده شدن قابلیت‌ها و ویژگی‌های مشخص شده در مدل خواسته توسط طراحی است. در *کیفیت متابعتی* (Quality of Conformance)، میزان متابعت پیاده‌سازی از طراحی و برآورده شدن خواسته‌ها و اهداف کارایی توسط سیستم حاصل، کانون توجه قرار می‌گیرد. ولی آیا کیفیت طراحی و کیفیت متابعتی، تنها مسائلی هستند که مهندس نرم‌افزار باید در نظر بگیرد؟ رابرت گلاس [Gla98] استدلال می‌کند که یک رابطه‌ی «مستقیم‌تر» وجود دارد:

تحویل در زمان‌بندی و بودجه تعیین شده + کیفیت خوب + محصول مطابق با استاندارد = رضایت کاربر

گلاس در کل مدعی است که کیفیت اهمیت دارد، ولی اگر کاربر رضایت نداشته باشد، هیچ چیز دیگری اهمیت ندارد. دومارکو [DeM98] این دیدگاه را تقویت می‌کند وقتی که می‌گوید: «کیفیت یک محصول تابعی است از این که چه مقدار دنیا را تغییر می‌دهد تا بهتر شود.» این دیدگاه کیفیتی، مدعی است که اگر یک محصول نرم‌افزاری، مزیتی اساسی برای کاربران نهایی خود به ارمغان بیاورد، ممکن است مشتاق به تحمل مشکلات گاه به گاه در کارایی یا قابلیت اطمینان باشند.

«مردم فراموش می‌کنند که چقدر سریع کاری را انجام داده‌اید- ولی همیشه به‌خاطر خواهند سپرد که چقدر خوب آن را انجام داده‌اید.»
هاوارد نیوتن

۲-۱۴ کیفیت نرم‌افزار

حتی بی‌حال‌ترین شرکت‌های نرم‌افزار نیز قبول دارند که تولید نرم‌افزارهای با کیفیت بالا، هدفی مهم است. ولی کیفیت نرم‌افزار را چگونه تعریف کنیم؟ در عمومی‌ترین حالت، کیفیت نرم‌افزار را می‌توان به این صورت تعریف کرد:[1]

یک فرایند نرم‌افزاری مؤثر، که به شیوه‌ای به‌کار برده می‌شود که محصولی مفید ایجاد می‌کند تا ارزشی قابل سنجش برای سازندگان این محصول و استفاده کنندگان از آن ایجاد کند.

بدون تردید تعریف فوق را می‌توان اصلاح کرد یا بسط داد ومدت‌ها درباره آن بحث و مشاجره کرد. برای اهداف این کتاب، این تعریف به تأکید بر سه نکته مهم کمک می‌کند.

۱. *فرایند نرم‌افزار اثربخش*، زیرساختی را بنا می‌کند که هرگونه تلاش برای ساخت یک محصول نرم‌افزاری با کیفیت بالا را پشتیبانی می‌کند. جنبه‌های مدیریتی فرایند، موازنه‌ها و نقاطی برای بررسی ایجاد می‌کنند که به پروژه کمک می‌کند تا از آشوب- یک عامل کلیدی در ضعف کیفیت- در امان بمانند. کار مهندسی نرم‌افزار به سازنده امکان می‌دهد تا مسأله را تحلیل کرده راهکاری منسجم طراحی کنند که هر دوی این‌ها در ساخت نرم‌افزار با کیفیت بالا، اهمیت حیاتی دارند. سرانجام، فعالیت‌های چتری نظیر مدیریت تغییر و بازبینی‌های فنی به اندازه هر بخش دیگری در کار مهندسی نرم‌افزار با کیفیت در ارتباط هستند.

۲. *محصول مفید*، محتویات، قابلیت‌ها و ویژگی‌هایی را تحویل می‌دهد که مطلوب کاربر نهایی هستند، ولی تحویل این موارد به شیوه‌ای مطمئن و عاری از خطا نیز به همان اندازه دارای اهمیت است. یک محصول مفید همواره خواسته‌هایی را که به صراحت توسط طرف‌های ذی‌نفع بیان شده است، پاسخ می‌دهد. به علاوه، مجموعه خواسته‌های ناگفته‌ای را که از همه‌ی نرم‌افزارهای با کیفیت بالا انتظار می‌رود (نظیر سهولت استفاده) پاسخ می‌دهد.

۳. یک نرم‌افزار باکیفیت *بالا با افزودن ارزش برای تولید کننده و کاربر* این محصول نرم‌افزاری، هم برای سازمان نرم‌افزاری و هم برای جامعه کاربران نهایی مزیت فراهم می‌کند. سازمان نرم‌افزاری از آن رو ارزش افزوده کسب می‌کند که نرم‌افزار با کیفیت بالا به تلاش کمتر برای نگهداری، اشکال‌زدایی کمتر و پشتیبانی کمتر برای مشتری نیاز دارد. این به مهندسان نرم‌افزار امکان می‌دهد تا وقت بیشتری را صرف ایجاد برنامه‌های کاربردی جدید کنند و کمتر به دوباره‌کاری بپردازند. جامعه کاربران از آن رو ارزش افزوده کسب می‌کنند که برنامه یک قابلیت مفید فراهم می‌سازد به‌طوری که یک فرایند تجاری خاص با سرعت بیشتری انجام شود. نتیجه‌ی نهایی (۱) درآمد بیشتر برای محصول نرم‌افزاری، (۲) منفعت بهتر هنگام پشتیبانی یک برنامه کاربردی از یک فرایند تجاری و/یا (۳) بهبود دسترسی به اطلاعات حیاتی برای شرکت تجاری خواهد بود.

۱-۲-۱۴ ابعاد کیفیتی گاروین

دیوید گاروین [Gar87] معتقد است که به کیفیت باید با در نظر گرفتن یک دیدگاه چند بعدی پرداخت که با ارزیابی پیروی از طراحی آغاز می‌شود و با دیدگاه متعالی (زیبایی شناسی) به پایان

[1] این تعریف از [Bes04] برگرفته شده است و جایگزین دیدگاه صنعتی‌تر ویرایش‌های قبلی این کتاب شده است.

می‌رسد. گرچه هشت بُعد کیفیتی گاروین مشخصاً برای نرم‌افزار توسعه نیافته‌اند، آن‌ها را هنگام پرداختن به کیفیت نرم‌افزار می‌توان به‌کار برد.

کیفیت کارآیی. آیا نرم‌افزار همه‌ی محتویات، قابلیت‌ها و ویژگی‌های مشخص شده در مدل خواسته‌ها را طوری تحویل می‌دهد که برای کاربر نهایی ایجاد ارزش کند؟

کیفیت ویژگی‌ها. آیا نرم‌افزار ویژگی‌هایی فراهم می‌آورد که کاربران نهایی را در نخستین استفاده از نرم‌افزار شگفت زده و خشنود سازد؟

قابلیت اطمینان. آیا نرم‌افزار، همه‌ی ویژگی‌ها و قابلیت‌ها را بدون شکست تحویل می‌دهد؟ آیا هنگام نیاز در دسترس هست؟ آیا قابلیت‌های عملیاتی را عاری از خطا تحویل می‌دهد؟

متابعت. آیا نرم‌افزار از استانداردهای محلی و خارجی که به کاربرد مورد نظر مربوط می‌شوند، پیروی می‌کند؟ آیا با قراردادهای طراحی و کدنویسی غیر رسمی همخوانی دارد؟ برای مثال، واسط کاربر با قواعد طراحی پذیرفته شده برای انتخاب منوها یا وارد کردن داده‌ها همخوانی دارد؟

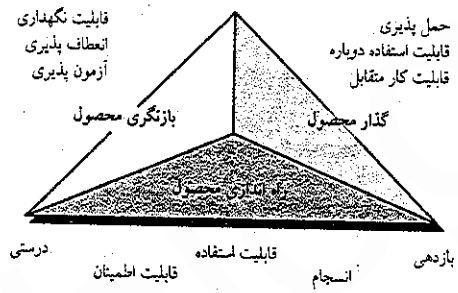
دوام. آیا نرم‌افزار را می‌توان نگهداری کرد (تغییر داد) یا تصحیح (اشکال‌زدایی) کرد بدون این‌که اثرات جانبی ایجاد شود؟ آیا تغییرات باعث می‌شود که میزان خطا یا قابلیت اطمینان با زمان کاهش یابد؟

قابلیت سرویس‌دهی. آیا نرم‌افزار را می‌توان در زمانی کوتاه نگهداری کرد (تغییر داد) یا تصحیح (اشکال‌زدایی) کرد؟ آیا کارمندان پشتیبانی همه‌ی اطلاعاتی را که برای اعمال تغییرات یا تصحیح نقایص لازم دارند، می‌توانند به‌دست آورند؟ داگلاس آدامز [Ada93] گفته‌ای طعنه آمیز دارد که در این جا مناسب به نظر می‌رسد: «اختلاف میان چیزی که امکان اشتباه در آن هست و چیزی که امکان اشتباه در آن نیست، در آن است که وقتی چیزی که احتمال اشتباه در آن نیست، به خطا برود، معمولاً ترمیم و تصحیح آن غیر ممکن به نظر می‌رسد.»

زیبایی‌شناسی. بدون تردید، هر کدام از ما چشم‌اندازی منحصر به فرد و متفاوت در خصوص زیبایی داریم. و در عین حال، اکثریت با این موضوع موافق هستند که یک موجودیت زیبایی‌شناسی دارای ظرافت معین، جریان منحصر به فرد و «حضوری» آشکار است که تعیین کمیتی برای آن دشوار است، ولی با همه‌ی این‌ها مشهود است. نرم‌افزار زیبا این خصوصیات را دارد.

ادراک. در برخی شرایط، یک مجموعه پیش داوری دارید که بر ادراک شما از کیفیت تأثیر دارند. برای مثال، اگر یک محصول نرم‌افزاری به شما معرفی شده باشد که توسط سازنده‌ای ساخته شده است که در گذشته کیفیتی ضعیف ارائه کرده است، در مقابل آن حالت دفاعی می‌گیرید و ادراک شما از کیفیت نرم‌افزار ممکن است تأثیر منفی بپذیرد. به‌طور مشابه، اگر سازنده‌ای از آوازه‌ای عالی برخوردار باشد، ممکن است کیفیت در ذهن شما متبادر شود، حتی اگر واقعاً وجود نداشته باشد.

ابعاد کیفیتی گاروین، دیدی «ملایم» از کیفیت نرم‌افزار در اختیار شما قرار می‌دهند. بسیاری از این ابعاد (نه همه‌ی آن‌ها) را تنها می‌توان به‌طور موضوعی در نظر گرفت. به همین دلیل، به یک مجموعه عوامل کیفیتی «سخت» نیز احتیاج دارید که می‌توان آن‌ها را در دو گروه گسترده طبقه‌بندی کرد: (۱) عواملی که به‌طور مستقیم قابل اندازه‌گیری‌اند (مثلاً نقایص بر ملا شده طی آزمون) و (۲) عواملی که تنها به‌طور غیر مستقیم قابل اندازه‌گیری‌اند (مثلاً قابلیت استفاده یا قابلیت نگهداری). در هر مورد، اندازه‌گیری باید رخ دهد. باید نرم‌افزار را با داده‌ای مقایسه کنید و به شاخصی از کیفیت برسید.



شکل ۱-۱۴ عوامل کیفیتی مک‌کال در نرم‌افزار.

۱۴-۲-۲ عوامل کیفیتی مک‌کال

مک کال، ریچاردز و والترز [McC77] عوامل تأثیرگذار بر کیفیت نرم‌افزار را دسته‌بندی کردند. این عوامل کیفیتی نرم‌افزار (شکل ۱-۱۴)، بر سه جنبه مهم از یک محصول نرم‌افزاری تأکید دارند: ویژگی‌های عملیاتی، توانایی تحمل تغییرات و تطبیق‌پذیری با محیط‌های جدید.

مک‌کال با توجه به شکل ۱-۱۴ توصیفات زیر را ارائه می‌دهد:

درستی. حد برآورده شدن مشخصه‌های یک برنامه توسط آن برنامه و رسیدن به اهداف مشتری.

قابلیت اطمینان. حدی که می‌توان از برنامه انتظار داشت تا عملکردهای موردنظر را با دقت لازم ارائه دهد. (لازم به ذکر است که تعاریف کاملتری از قابلیت اطمینان را در فصل ۸ ارائه دادیم.)

بازدهی. مقدار منابع کامپیوتری و کد لازم برای آنکه برنامه قادر به اجرای عملکردهای خود باشد.

انسجام. حد کنترل دستیابی افراد غیرمجاز به نرم‌افزار یا داده‌ها.

قابلیت استفاده. کار لازم برای فراگیری، راه‌اندازی، آماده کردن ورودی و تفسیر خروجی برنامه.

قابلیت نگهداری. کار لازم برای یافتن و تصحیح خطاهای برنامه (این تعریف بسیار محدود است).

انعطاف‌پذیری. کار لازم برای اصلاح برنامه کامل شده.

آزمون‌پذیری. کار لازم برای آزمودن برنامه برای اطمینان یافتن از اینکه عملکرد موردنظر را به خوبی اجرا می‌کند.

حمل‌پذیری (Portability). کار لازم برای انتقال دادن نرم‌افزار از یک سخت‌افزار و / یا محیط سیستم نرم‌افزاری به دیگری.

قابلیت استفاده‌ی مجدد. حدی که می‌توان یک برنامه (یا بخش‌هایی از برنامه) را دوباره در کاربردهای دیگر - مرتبط با پکیج‌سازی و دامنه عملیاتی که برنامه اجرا می‌کند - استفاده کرد.

قابلیت کار متقابل. کار لازم برای جفت کردن یک سیستم به سیستم دیگر.

توسعه موازین مستقیمی[1] از عوامل کیفیتی بالا، کاری دشوار و در برخی موارد، غیرممکن است. درواقع، بسیاری از معیارهایی که مک‌کال مشخص می‌کند، فقط به‌طور غیرمستقیم قابل اندازه‌گیری‌اند. به هر حال، ارزیابی کیفیت یک برنامه کاربردی با استفاده از این عوامل، شاخصی نسبی از کیفیت نرم‌افزاری در اختیارتان قرار خواهد داد.

> «تلخ‌کامی کیفیت ضعیف تا مدت‌ها پس از فراموشی شیرینی سروقت بودن در خاطر می‌ماند.»
>
> کارل ویگرز

[1] *میزان مستقیم* به آن معناست که تک مقدار قابل شمارشی وجود دارد که شاخص مستقیمی از صفت مورد نظر به‌دست می‌دهد. برای مثال، «اندازه» برنامه را مستقیماً با شمارش تعداد خطوط کد می‌توان سنجید.

۱۴-۲-۳ عوامل کیفیتی ISO 9126

استاندارد ISO 9126 به منظور تعیین صفات کیفیتی مهم برای نرم‌افزارهای کامپیوتری تدوین شده است. این استاندارد شش صفت کلیدی را برای کیفیت در نظر می‌گیرد:

قابلیت عملیاتی. حدی که نرم‌افزار، نیازهای ذکر شده براساس این صفات را برآورده می‌کند: مناسب بودن، صحیح بودن، قابلیت کار متقابل، تطابق و امنیت.

قابلیت اطمینان. مدت زمانی که نرم‌افزار براساس این صفات در دسترس است: بلوغ، تحمل در برابر خطاها، قابلیت رهایی یافتن از خطا.

قابلیت استفاده. حد سهولت استفاده از نرم‌افزار براساس این صفات: قابلیت درک، قابلیت فراگیری، قابلیت‌کار با آن.

بازدهی. حدی که نرم‌افزار براساس این صفات، از منابع سیستم استفاده بهینه به عمل می‌آورد: رفتار زمانی، رفتار منابعی.

قابلیت نگهداری. سهولت ترمیم نرم‌افزار براساس این صفات: تحلیل‌پذیری، تغییرپذیری، پایداری و آزمون‌پذیری.

حمل‌پذیری. سهولت انتقال نرم‌افزار از محیطی به محیط دیگر براساس این صفات: تطبیق‌پذیری، ناپایداری، مطابقت، قابلیت جایگزینی.

همانند عوامل دیگر کیفیتی که در بخش‌های قبل بحث شد، عوامل ISO 9126 نیز الزاماً ربطی به اندازه‌گیری مستقیم ندارند. ولی بدون تردید، مبنای ارزشمندی برای موازین غیرمستقیم و یک لیست کنترلی عالی برای ارزیابی کیفیت سیستم به‌دست می‌دهند.

> **اندرز**
>
> گرچه توسعه‌ی موازین کمّی برای عوامل کیفیتی ذکر شده در این جا وسوسه انگیز است، می‌توانید چک‌لیست ساده‌ای از صفات نیز تهیه کنید که حضور هر عامل را به‌طور قطع نشان دهد.

۱۴-۲-۴ عوامل کیفیتی هدف‌مند

در ابعاد و عوامل کیفیتی ارائه شده در بخش‌های ۱-۲-۱۴ و ۲-۲-۱۴، نرم‌افزار به‌عنوان یک کل، کانون توجه قرار می‌گیرد و از آن‌ها می‌توان به‌عنوان شاخصی کلی از کیفیت برنامه کاربردی استفاده کرد. تیم نرم‌افزاری می‌تواند مجموعه‌ای از خصوصیات کیفیتی و پرسش‌های مربوط به آن‌ها را تهیه کند که میزان دستیابی به هر عامل را از طریق آن‌ها مورد تفحص قرار دهد.[1] برای مثال، مک‌کال *قابلیت استفاده* را یک عامل کیفیتی مهم می‌شناسد. اگر از شما خواسته می‌شد که یک واسط کاربر را بازبینی کنید و قابلیت استفاده آن را مورد ارزیابی قرار دهید، چه می‌کردید؟ ممکن است با صفات فرعی پیشنهاد شده توسط مک کال- درک‌پذیری، قابلیت فراگیری، و قابلیت کار- شروع کنید، ولی این‌ها از نظر عمل‌گرایی چه معنایی دارد؟

> «هر فعالیت هنگامی خلاقانه می‌شود که کنده‌ی آن به انجام درست یا انجام بهتر آن اهمیت دهد.»
>
> جان آپدایک

برای اجرای ارزیابی، به صفاتی مشخص و قابل اندازه‌گیری (یا حداقل قابل تشخیص) از واسط بپردازید. برای مثال [Bro03]:

بصیرت‌گرایی (Intuitiveness). میزان پیروی واسط از الگوهای کاربرد مورد انتظار به‌طوری که حتی یک کاربر تازه‌کار بتواند بدون نیاز به آموزش زیاد از آن استفاده کند.

- آیا چیدمان واسط، درک آسان را فراهم می‌سازد؟

[1] به این خصوصیات و پرسش‌ها به‌عنوان بخشی از بازبینی نرم‌افزار خواهیم پرداخت.

- آیا عملیات‌های واسط را به راحتی می‌توان یافت و اجرا کرد؟
- آیا واسط از یک استعاره قابل تشخیص استفاده می‌کند؟
- آیا ورودی طوری مشخص شده است که تعداد کلیک‌های ماوس یا ضربات صفحه کلید را به حداقل برساند؟
- آیا واسط از سه قاعده طلایی (فصل ۱۱) پیروی می‌کند؟
- آیا زیبایی‌شناسی به درک و استفاده از واسط کمک می‌کند؟

بازدهی (Efficiency). میزانی از امکان یافتن عملیات‌ها و اطلاعات یا استفاده از آن‌ها.

- آیا سبک و چیدمان واسط به کاربر این امکان را می‌دهد که عملیات‌ها و اطلاعات را به‌طرزی اثربخش پیدا کند؟
- آیا یک سری عملیات‌ها (یا وارد کردن داده‌ها) را می‌توان با حداقل حرکت انجام داد؟
- آیا داده‌های خروجی یا محتویات به گونه‌ای ارائه می‌شوند که بلافاصله قابل درک باشند؟
- آیا عملیات‌های سلسله مراتبی به گونه‌ای سازمان‌دهی شده‌اند که عمق پیشروی برای انجام کاری دیگر را توسط کاربر به حداقل برسانند؟

استحکام (Robustness). میزان اداره‌کردن داده‌های ورودی بد یا تعامل نامناسب کاربر توسط نرم‌افزار.

- اگر داده‌ها در مرزهای تعیین شده یا خارج از آن وارد شوند، آیا نرم‌افزار قادر به تشخیص خطا خواهد بود؟ مهم‌تر از آن آیا نرم‌افزار بدون شکست یا تنزل به عملکرد خود ادامه خواهد داد؟
- آیا واسط قادر به تشخیص اشتباهات شناختی یا دستکاری رایج خواهد بود و به صراحت کاربر را به مسیر درست باز خواهد گرداند؟
- آیا هنگامی که شرایط خطا (مربوط به قابلیت‌های عملیاتی نرم‌افزار) آشکار می‌شود، واسط راهنمایی و عیب یابی مفید را فراهم می‌آورد؟

غنا (Richness). میزان ارائه مجموعه‌ای غنی از ویژگی‌ها به‌وسیله‌ی واسط.

- آیا کاربر می‌تواند واسط را مطابق سلیقه خود تغییر دهد تا نیازهای خاص خود را برآورده کند؟
- آیا واسط، *قابلیت ماکرویی* فراهم می‌آورد که کاربر به کمک آن بتواند انجام یک سری عملیات متداول را با یک فرمان مشخص کند؟

به موازاتی که طراحی واسط توسعه می‌یابد، تیم نرم‌افزاری به بازبینی نمونه اولیه طراحی می‌پردازد و سؤالات ذکر شده را می‌پرسد. اگر پاسخ اکثر این سؤالات «مثبت» باشد، این احتمال وجود دارد که واسط کیفیت بالایی از خود نشان دهد. مجموعه‌ای از پرسش‌ها مشابه با پرسش‌های فوق برای هر عامل کیفیتی باید تهیه شود.

۱۴-۲-۵ کمّی‌کردن کیفیت

در بخش‌های قبلی، مجموعه‌ای از عوامل کیفی برای «اندازه‌گیری» کیفیت نرم‌افزار مورد بحث قرار گرفت. می‌کوشیم تا موازین دقیقی برای کیفیت نرم‌افزار توسعه دهیم، ولی ماهیت موضوعی فعالیت، ما را آزرده می‌سازد. کاوانو و مک‌کال این وضعیت را چنین توصیف می‌کنند:

> تعیین کیفیت، یک عامل کلیدی در رویدادهای روزمره است - مسابقات تعیین طعم و نوشیدنی‌ها، رویدادهای ورزشی (مثل ژیمناستیک)، مسابقات استعدادها و غیره. در این وضعیت، کیفیت به بنیادی‌ترین و مستقیم‌ترین شیوه مورد قضاوت قرار می‌گیرد: مقایسه پهلو به پهلوی اشیاء تحت شرایط یکسان و با مفاهیم از پیش تعیین شده. نوشیدنی ممکن است براساس زلالیت، رنگ، طعم و غیره مورد قضاوت قرار گیرد. ولی چنین قضاوتی بسیار موضوعی است و برای آن که ارزش داشته باشد باید توسط فردی کارشناس انجام شود.

> موضوعی بودن و تخصصی بودن در تعیین کیفیت نرم‌افزار نیز صادق است، برای کمک به حل این مشکل، تعریف دقیق‌تر کیفیت نرم‌افزار و نیز راهی برای به دست آوردن اندازه‌گیری‌های کمّی جهت تحلیل عینی موردنیاز است ... چون چنین چیزی در حالت مطلق وجود ندارد، نباید انتظار اندازه‌گیری دقیق کیفیت نرم‌افزار را داشت. زیرا هر اندازه‌گیری تا حدی ناقص است. جیکاب برانکوسکی این پارادوکس را چنین شرح می‌دهد: «سال به سال دستگاه‌هایی با دقت بیشتر ابداع می‌کنیم که به کمک آنها طبیعت را با دقت بیشتر می‌توان مشاهده کرد. هنگامی که به مشاهدات خود نگاه می‌کنیم، از این ناراضی هستیم که هنوز از دقتی که خواهان آن هستیم برخوردار نیستند و احساس می‌کنیم هنوز دارای همان عدم قطعیت گذشته‌اند».

در فصل ۲۳ ، مجموعه‌ای از معیارها را مورد بررسی قرار خواهیم داد که می‌توان آنها را برای ارزیابی کمّی کیفیت نرم‌افزار به‌کار برد. در تمامی موارد، معیارها نشان‌گر موازینی غیر مستقیم هستند، یعنی هرگز کیفیت را اندازه‌گیری نمی‌کنیم، بلکه نمودی از آن را می‌سنجیم. عاملی که بر پیچیدگی‌ها می‌افزاید، رابطه دقیق میان متغیر اندازه‌گیری شده و کیفیت است.

۱۴-۳ معضل کیفیت نرم‌افزار

برتران مایر طی مصاحبه‌ای [Ven03] که در وب منتشر شد، آن چه را که در این‌جا معضل کیفیت خوانده ایم، چنین مورد بحث قرار می‌دهد:

> اگر یک سیستم نرم‌افزاری با کیفیت افتضاح تولید کنید، هیچ کس مایل به خریدن آن نخواهد بود. از طرف دیگر، اگر بی‌نهایت زمان صرف این امر کنید، بی‌اندازه تلاش کنید و مقادیر هنگفتی پول صرف ساخت یک قطعه نرم‌افزار مطلقاً کامل کنید، تولید نرم‌افزار آن قدر به طول خواهد انجامید و چنان پر هزینه خواهد شد که در هر حال از کار عقب خواهید ماند. یا زمان مناسب بازار را از دست خواهید داد یا به سادگی همه‌ی منابع خود را هدر می‌دهید. پس آن‌ها که در این صنعت مشغول هستند، سعی می‌کنند به آن حد میانی دست پیدا کنند که محصول آن قدر خوب باشد که بلافاصله طی مرحله‌ی ارزیابی، برگشت داده نشود و در عین حال آن قدر هم در پی کمال‌گرایی نباشد که برای کامل شدن، به کار و هزینه‌ی بیش از حد نیاز داشته باشد.

اندرز
هنگامی که با معضل کیفیت مواجه شدید (و همگان با آن مواجه خواهند شد) بکوشید تا موازنه برقرار کنید: تلاش کافی برای ایجاد کیفیت قابل قبول بدون مدفون کردن پروژه.

خوب است متذکر شویم که مهندسان باید بکوشند تا سیستم‌های با کیفیت بالا تولید کنند. حتی بهتر است برای این منظور از روش‌های عملی خوب استفاده کنید. ولی وضعیتی که مایر درباره آن بحث می‌کند، چیزی است که در عمل و واقعیت رخ می‌دهد و نشان‌گر معضلی است که حتی بهترین سازمان‌های مهندسی نرم‌افزار با آن دست به گریبان هستند.

۱-۳-۱۴ نرم‌افزارِ «به‌قدر کافی خوب»

به بیان صریح، اگر قرار باشد استدلال مایر را بپذیریم، آیا قابل قبول است که نرم‌افزار «به‌قدر کافی خوب» تولید کنیم؟ پاسخ به این سؤال باید «مثبت» باشد زیرا اکثر شرکت‌های بزرگ نرم‌افزار همین کار را می‌کنند. آن‌ها نرم‌افزارهایی با اشکال‌های معلوم و آشکار ایجاد می‌کنند و آن‌ها را به جمعیت گسترده‌ای از کاربران نهایی عرضه می‌کنند. آن‌ها می‌دانند که برخی ویژگی‌ها و قابلیت‌های ارائه شده در نسخه 1.0 ممکن است دارای بالاترین کیفیت ممکن نباشد و برنامه‌ریزی می‌کنند که در نسخه 2.0 آن را بهبود بخشند. آن‌ها می‌دانند که مشتری شکایت خواهد کرد، ولی این را هم می‌دانند که زمان عرضه به بازار (مادامی که محصول «به‌قدر کافی خوب باشد») ممکن است کیفیت بهتر را تحت‌شعاع قرار دهد.

دقیقاً منظور از «به‌قدر کافی خوب» چیست؟ نرم‌افزاری که به‌قدر کافی خوب باشد، قابلیت‌ها و ویژگی‌های مطلوب کاربر را با کیفیت بالا به او تحویل می‌دهد، ولی در عین حال، قابلیت‌ها و ویژگی‌های تخصص‌یافته‌تر یا گمنام‌تری را تحویل می‌دهد که حاوی اشکال‌های شناخته شده‌اند. فروشنده نرم‌افزار امیدوار است که اکثریت وسیع کاربران نهایی به این اشکال‌ها به دیده اغماض بنگرند، چون از سایر قابلیت‌های عملیاتی برنامه بسیار راضی‌اند.

این ایده ممکن است به مذاق بسیاری از خوانندگان خوش بیاید. اگر شما یکی از آن‌ها هستید، فقط می‌توانیم از شما بخواهیم برخی استدلال‌های ارائه شده در مخالفت با «به‌قدر کافی خوب» را در نظر بگیرید.

این درست است که «به‌قدر کافی خوب» ممکن است در برخی دامنه‌های کاربردی و برای چند شرکت بزرگ، جواب بدهد. ولی گذشته از همه‌ی این‌ها، اگر شرکتی یک بودجه بازاریابی بزرگ داشته باشد و بتواند تعداد کافی از مشتریان را قانع سازد که نسخه 1.0 را بخرند، موفق شده است که آن‌ها را درگیر کند. چنان که پیش از این گفته شد، شرکت می‌تواند استدلال کند که کیفیت را در نسخه‌های بعدی بهبود می‌بخشد. این شرکت با تحویل نسخه‌ی 1.0 که به‌قدر کافی خوب است، بازار را به دام انداخته است.

اگر برای یک شرکت کوچک کار می‌کنید، مراقب این فلسفه باشید. هنگامی که محصولی «به‌قدر کافی خوب» (دارای اشکال) تحویل می‌دهید، این ریسک را می‌کنید که اعتبار و آبروی شرکت را برای همیشه به خطر اندازید. ممکن است هرگز فرصت تحویل نسخه‌ی 2.0 را پیدا نکنید چون این آوازه‌ی بد ممکن است باعث شود فروش شما افت کند و شرکت ورشکست شود.

اگر در دامنه‌ی کاربری معین کار می‌کنید (مثلاً نرم‌افزارهای زمان حقیقی تعبیه‌شده) یا برنامه‌های کاربردی می‌سازید که با سخت افزار ارائه می‌شوند (مثلاً نرم‌افزارهای خودرو، نرم‌افزارهای مخابراتی)، تحویل نرم‌افزارهایی با اشکال‌های شناخته شده و معلوم، می‌تواند سهل‌انگاری باشد و شرکت را در مظان اتهام قرار دهد. ممکن است در برخی موارد، قضیه حتی جنایی شود. هیچ کس نرم‌افزار «به‌قدر کافی خوب» برای ناوبری هواپیما نمی‌خواهد!

بنابراین، اگر بر این باورید که «به‌قدر کافی خوب» میان‌بری است که می‌تواند مسائل کیفیتی شما را حل کند، با احتیاط گام بردارید. این فلسفه می‌تواند جواب بدهد، ولی تنها برای چند دامنه کاربرد معدود و در مجموعه‌ی محدودی از کاربردها[1].

[1] بحثی ارزشمند درباره مزایا و معایب نرم‌افزارهای «به‌قدر کافی خوب» را می‌توانید در [Bre02] بیابید.

۲-۳-۱۴ هزینه‌ی کیفیت

عده‌ای چنین استدلال می‌کنند: می‌دانیم *کیفیت مهم است، ولی زمان و پول می‌خواهد- مقادیر بیش از حد زمان و پول برای رسیدن به آن سطح از نرم‌افزار که می‌خواهیم*. این استدلال، ظاهراً منطقی به نظر می‌رسد (توضیحات مایر را در بخش قبل ببینید). تردیدی نیست که کیفیت، هزینه بردار است، ولی فقدان کیفیت نیز هزینه بردار است- نه تنها برای کاربران نهایی که باید با یک نرم‌افزار اشکال‌دار زندگی کند، بلکه برای سازمان نرم‌افزار که آن را ساخته است و باید نگهداری آن را بر عهده بگیرد. پرسش واقعی این است: *درباره کدام هزینه باید نگران بود؟* برای پاسخ گفتن به این پرسش باید هم هزینه دستیابی به کیفیت و هم هزینه نرم‌افزار با کیفیت پایین را بشناسید.

هزینه کیفیت شامل همه‌ی هزینه‌هایی می‌شود که در جستجوی کیفیت یا اجرای فعالیت‌های مرتبط با کیفیت و هزینه‌های ناشی از فقدان کیفیت تحمیل می‌شوند. برای شناخت این هزینه‌ها، سازمان باید معیارهایی جمع‌آوری کند که بستری برای هزینه جاری کیفیت، شناسایی فرصت‌ها برای کاهش‌دادن این هزینه‌ها و فراهم ساختن مبنایی بهنجار جهت مقایسه به‌دست دهد. هزینه‌ی کیفیت را می‌توان به هزینه‌های مرتبط با پیش‌گیری، ارزیابی و شکست تقسیم کرد.

هزینه‌های پیش‌گیری عبارتند از (۱) فعالیت‌های مدیریتی مورد نیاز برای برنامه‌ریزی و هماهنگ کردن کلیه فعالیت‌های تضمین کیفیت، (۲) هزینه فعالیت‌های فنی برای توسعه و تکمیل مدل خواسته‌ها و مدل طراحی، (۳) هزینه‌های برنامه‌ریزی آزمون و (۴) هزینه همه‌ی آموزش‌های مرتبط با این فعالیت‌ها.

هزینه‌های ارزیابی شامل فعالیت‌های انجام شده برای به‌دست آوردن دیدی از وضعیت محصول در «نخستین گذر» از هر فرایند می‌شود. مثال‌هایی از هزینه‌های ارزیابی عبارتند از:

- هزینه‌ی اجرای بازبینی‌های فنی (فصل ۱۵) برای محصولات کاری مهندسی نرم‌افزار.
- هزینه جمع‌آوری داده‌ها و ارزیابی معیارها (فصل ۲۳).
- هزینه آزمون و اشکال‌زدایی (فصل‌های ۱۸ تا ۲۱)

هزینه‌های شکست به آن دسته از هزینه‌هایی گفته می‌شود که در صورت عدم بروز خطا قبل یا بعد از رسیدن محصول به‌دست مشتری، ناپدید می‌شوند. هزینه‌های شکست را می‌توان به هزینه‌های شکست داخلی و هزینه‌های شکست خارجی تقسیم کرد. *هزینه‌های شکست داخلی* هنگامی تحمیل می‌شوند که در محصول و قبل از رسیدن آن به مشتری کشف می‌شوند. این هزینه‌ها عبارتند از:

- هزینه لازم برای اجرای دوباره کاری (ترمیم) برای تصحیح خطا
- هزینه ناشی از اثرات جانبی که در اثر دوباره کاری‌ها ایجاد می‌شود.
- هزینه‌های مربوط به جمع‌آوری معیارهای کیفیتی که به سازمان این امکان را می‌دهند تا به حالت‌های شکست دست پیدا کند.

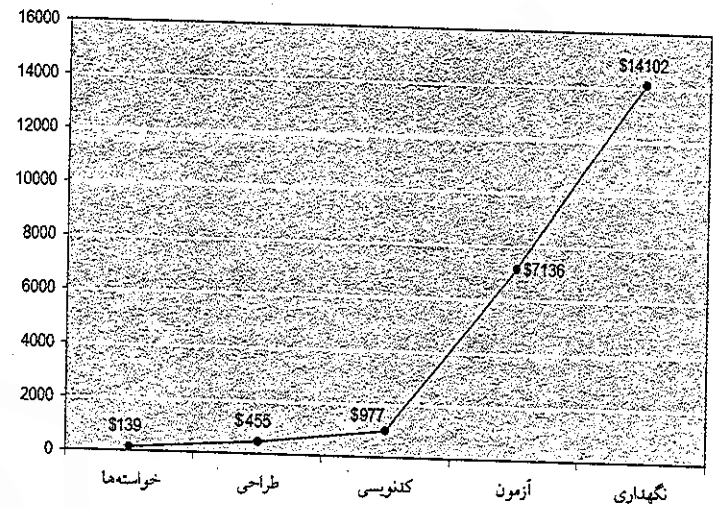
هزینه‌های شکست خارجی به نقایصی مربوط می‌شود که پس از رسیدن محصول به‌دست مشتری کشف می‌شوند. مثال‌هایی از هزینه‌های شکست خارجی عبارتند از جلب رضایت شاکی، مرجوع و جایگزین کردن محصول، کمک به پشتیبانی خطی و هزینه‌های کار مرتبط با ضمانت. بدنامی و ضرر و زیان نیز یک هزینه شکت خارجی دیگر است که تعیین کمّی آن دشوار است، ولی بسیار واقعی است. در صورت تولید نرم‌افزار با کیفیت پایین، اتفاقات بدی رخ می‌دهد.

اندرز

از تحمیل هزینه‌های پیش‌گیری چشمگیر نهراسید. مطمئن باشید که این سرمایه‌گذاری، عایدی بسیار عالی خواهد داشت.

درست انجام‌دادن کاری کمتر زمان می‌برد از این که توضیح دهید چرا آن را غلط انجام داده‌اید.

اچ. دبلیو. لانگفلو



شکل ۱۴-۲ هزینه نسبی تصحیح خطاها و نقایص.

سیم کانر [Kan98] در نکوهش سازندگانی که از درنظرگرفتن هزینه‌های شکست خارجی سرباز می‌زنند، چنین می‌گوید:

> بسیاری از هزینه‌های شکست خارجی، نظیر خوش‌نامی را به سختی می‌توان سنجید و از این رو بسیاری از شرکت‌ها هنگام محاسبه تراز سود و زیان خود از این هزینه‌ها غفلت می‌کنند. سایر هزینه‌های شکست خارجی را می‌توان (با فراهم آوردن پشتیبانی ارزان‌تر، با کیفیت کمتر و پس از فروش، یا با گرفتن هزینه پشتیبانی از مشتری) کاهش داد، بدون این که بر رضایت مشتری افزوده شود. مهندسان کیفیت، با غفلت از هزینه‌هایی که محصولات بد به مشتری‌ها وارد می‌سازند، تصمیم‌گیری‌هایی مرتبط با کیفیت را ترغیب می‌کنند که مشتریان را قربانی می‌کنند به جای این که آن‌ها را راضی نگه دارند.

همان‌طور که انتظار می‌رود، هزینه‌های نسبی برای یافتن و ترمیم خطاها یا نقایص با رفتن از پیش‌گیری به سوی کشف خطا و سپس شکست داخلی و سرانجام شکست خارجی به شدت افزایش می‌یابد. شکل ۲-۱۴ بر اساس داده‌های جمع‌آوری شده توسط بوهم [Boe01b]، و نشان داده شده توسط شرکت سیجیتال [Cig07]، این پدیده را مطرح می‌کند.

هزینه صنعتی میانگین برای تصحیح یک نقص طی کدنویسی تقریباً ۹۷۷ دلار به‌ازای هر خطاست. هزینه میانگین صنعتی برای تصحیح همان خطا اگر طی آزمون سیستم کشف گردد، ۷۱۳۶ دلار است. شرکت سیجیتال [Cig07] یک برنامه کاربردی بزرگ را در نظر می‌گیرد که طی کدنویسی آن ۲۰۰ خطا وارد شده است.

طبق داده‌های میانگین صنعتی، هزینه‌ی کشف و تصحیح نقایص طی مرحله‌ی کدنویسی، ۹۷۷ دلار به‌ازای هر نقص است. از این رو، هزینه کل برای تصحیح این ۲۰۰ نقص «حیاتی» طی این مرحله، (۹۷۷×۲۰۰) حدوداً ۱۹۵۴۰۰ دلار می‌شود.

داده‌های میانگین صنعتی نشان می‌دهد که هزینه‌ی کشف و تصحیح نقایص طی آزمون سیستم، ۷۱۳۶ دلار به‌ازای هر نقص است. در این مورد، با این فرض که در مرحله آزمون، تقریباً ۵۰ نقص حیاتی (یا فقط ۲۵٪ از آن چه که سیجیتال در فاز کدنویسی کشف کرده است) آشکار شود، هزینه این کشف و تصحیح (۷۱۳۶×۵۰) تقریباً برابر با ۳۵۶۸۰۰ دلار می‌شود. علاوه بر این، ۱۵۰ خطا نیز باقی می‌ماند که در مرحله‌ی نگهداری ۲۱۱۵۳۰۰ دلار (۱۴۱۰۲×۱۵۰) باید صرف کشف و تصحیح آن‌ها کرد. پس، هزینه کل کشف و تصحیح ۲۰۰ نقص پس از مرحله‌ی کدنویسی به ۲۴۷۲۱۰۰ دلار می‌رسد (۲۱۱۵۳۰۰+۳۵۶۸۰۰).

حتی اگر سازمان نرم‌افزاری شما نصف این مقادیر میانگین هزینه کند (خیلی‌ها اصلاً از این هزینه‌ها خبر ندارند!) صرفه جویی در هزینه‌ها در اثر فعالیت‌های زود هنگام در زمینه کنترل و تضمین کیفیت (که طی طراحی و تحلیل خواسته‌ها اجرا می‌شود) کاملاً توجیه دارد.

۱۴-۳-۳ ریسک

در فصل ۱ این کتاب نوشتیم که «انسان‌ها، راحتی، امنیت، سرگرمی، تصمیم‌گیری‌ها و زندگی خود را روی نرم‌افزارهای کامپیوتری می‌گذارند. پس بهتر است درست این کار را انجام دهند.» منظور این است که نرم‌افزارهای با کیفیت پایین، هم برای سازنده و هم برای کاربر نهایی ایجاد ریسک می‌کنند. در بخش قبل درباره یکی از این ریسک‌ها (افزایش هزینه‌ها) بحث شد. ولی اثرات سوء طراحی و پیاده‌سازی ضعیف برنامه‌های کاربردی همواره به پول و وقت خلاصه نمی‌شود. یک مثال حدی [Gay04] ممکن است به روشن شدن مطلب کمک کند.

در سرتاسر ماه نوامبر ۲۰۰۰، در بیمارستانی در پاناما، ۲۸ بیمار طی درمان انواع سرطان‌ها، مقادیر بیش از حد تجویز شده، پرتو گاما دریافت کردند. در ماه‌های بعد، پنج تن از آن‌ها بر اثر تابش در گذشتند و ۱۵ تن گرفتار مشکلات جدی شدند. چه چیز باعث این اتفاق ناگوار شد؟ بسته نرم‌افزاری‌ای که توسط یک شرکت امریکایی تهیه شده بود، توسط تکنسین‌های بیمارستان اصلاح شده بود تا دوز تابش هر بیمار را محاسبه کند.

سه فیزیکدان پزشکی پانامایی که نرم‌افزار را «دستکاری» کرده بودند تا قابلیت بیشتری برای آن فراهم آورند، به قتل عمد محکوم شدند. شرکت امریکایی در هر دو کشور با شکایات جدی مواجه شده است. کیج و مک کورمیک در این خصوص چنین می‌گویند:

> «این یک داستان عبرت انگیز برای تکنسین‌ها نیست، هر چند که خواهند دانست اگر از فن‌آوری درک درستی نداشته باشند یا از آن به درستی استفاده نکنند، سر از زندان در خواهند آورد. این، داستان صدمه دیدن انسان از نرم‌افزارهای با کیفیت یا طراحی ضعیف هم نیست، هر چند که مثال‌های زیادی برای این وضعیت نیز وجود دارد. این هشداری است برای سازندگان برنامه‌های کامپیوتری: که کیفیت نرم‌افزار اهمیتی اساسی دارد، که برنامه‌های کاربردی باید ضد خطا باشند و اینکه- خواه در موتور خودرو، بازوی روباتیک یک کارخانه یا یک دستگاه درمان بخش در بیمارستان تعبیه‌شده باشد- کدهای ضعیف می‌توانند باعث مرگ شوند.

کیفیت ضعیف به ریسک‌هایی منجر می‌شود که برخی از آن‌ها بسیار جدی‌اند.

۱۴-۳-۴ اهمال و بی‌کفایتی

این داستان بسیار رایج است. یک مؤسسه‌ی دولتی یا شرکتی، یک سازنده بزرگ نرم‌افزار یا یک شرکت مشاوره بزرگ را استخدام می‌کند که خواسته‌ها را تحلیل کند و سپس «سیستمی» کامپیوتری را طراحی

و ایجاد می‌کند که یک فعالیت مهم را پشتیبانی کند. این سیستم ممکن است یک وظیفه شرکتی مهم (مثلاً مدیریت مستمری) یا یک وظیفه دولتی مهم (مثلاً مدیریت بهداشت عمومی یا امنیت ملی) را پشتیبانی می‌کند.

کار با بهترین نیات از هر دو طرف آغاز می‌شود، ولی هنگامی که سیستم تحویل می‌شود، اوضاع خراب می‌شود. سیستم دیر جواب می‌دهد، از تحویل‌دادن قابلیت‌ها و ویژگی‌های مطلوب باز می‌ماند، مستعد خطاست و رضایت و تصویب مشتری را کسب نمی‌کند و شکایت‌ها شروع می‌شود.

در اکثر موارد، مشتری ادعا می‌کند که سازنده اهمال کرده است (از نظر شیوه اعمال کارهای نرم‌افزاری) و بنابراین نباید پولی دریافت کند. سازنده غالباً ادعا می‌کند که مشتری مکرراً خواسته هایش را تغییر داده است و مشارکت در توسعه را به مسیری دیگر کشانده است. در هر حال، کیفیت محصول تحویل شده جای تردید دارد.

۱۴-۳-۵ کیفیت و امنیت

به موازات رشد اهمیت حیاتی برنامه‌های کاربردی و سیستم‌های مبتنی بر وب، امنیت برنامه‌های کاربردی نیز اهمیتی فزاینده یافته است. به بیان ساده، نفوذ در نرم‌افزاری که کیفیت بالایی از خود نشان ندهد، راحت‌تر است و در نتیجه، نرم‌افزارهای با کیفیت پایین می‌توانند به‌طور غیر مستقیم ریسک امنیتی را با تمامی هزینه‌ها و مشکلات مربوط به آن افزایش دهند.

نویسنده و کارشناس امور امنیتی، گری مک‌گرا در مصاحبه‌ای با **ComputerWorld** [Wil05] می‌گوید:

> امنیت نرم‌افزار به‌طور کامل با کیفیت در ارتباط است. درباره امنیت، قابلیت اطمینان، قابلیت دسترسی و سازگاری- در تمامی مراحل شروع، در طراحی، معماری، آزمون و کدنویسی و در سرتاسر چرخه حیات [فرایند] نرم‌افزار باید بیندیشید. حتی آن‌ها که از مسأله امنیت آگاهی دارند، موارد مربوط به اواخر چرخه حیات را کانون توجه قرار داده‌اند. هرچه مشکل نرم‌افزار را زودتر بیابید، بهتر است و دو نوع مشکل در نرم‌افزار می‌تواند وجود داشته باشد. یکی اشکال‌ها (bugs) هستند که مربوط به پیاده‌سازی می‌شوند. دیگری نقص‌های نرم‌افزار (flaws) است- مشکلات معماری در طراحی. به اشکال‌ها توجه زیادی می‌شود، ولی به نقص‌ها توجه چندانی نمی‌شود.

برای ساخت سیستمی امن، باید کیفیت را کانون توجه قرار دهید و این توجه ویژه باید طی طراحی شروع شود. مفاهیم و روش‌های بحث شده در بخش دوم این کتاب به معماری نرم‌افزاری منجر می‌شود که «نقص ها» را کاهش می‌دهد. با حذف نقص‌های معماری (و در نتیجه، بهبود بخشیدن به کیفیت نرم‌افزار)، نفوذ بیگانگان به نرم‌افزار به مراتب دشوارتر خواهد شد.

۱۴-۳-۶ تأثیر کنش‌های مدیریتی

کیفیت نرم‌افزار غالباً به همان اندازه که از تصمیم‌گیری‌های فن‌آوری تأثیر می‌پذیرد، از تصمیم‌گیری‌های مدیریتی نیز تأثیرپذیر است. حتی بهترین کارهای مهندسی نرم‌افزار ممکن است با تصمیم‌گیری‌های تجاری ضعیف و کنش‌های مدیریت پروژه ضعیف به بیراهه کشیده شود.

در بخش ۴ از این کتاب به مدیریت پروژه در حیطه‌ی فرایند نرم‌افزار خواهیم پرداخت. با شروع هر وظیفه پروژه، یک مدیر پروژه، تصمیم‌هایی می‌گیرد که می‌توانند تأثیرات چشمگیری بر کیفیت نرم‌افزار بگذارند.

تصمیم‌گیری‌های برآوردی. چنان که در فصل ۲۶ گفته شد، تیم نرم‌افزار به ندرت این فرصت تجملی را به‌دست می‌آورد که پروژه‌ای را پیش از تعیین تاریخ‌های تحویل و مشخص شدن بودجه کلی برآورد کند. در عوض، تیم پروژه تاریخ‌های تحویل و نقاط عطف را از نظر منطقی چک می‌کند تا اطمینان یابد که موجه هستند. در بسیاری موارد، فشاری از نظر زمان عرضه به بازار وجود دارد که تیم را به پذیرش تاریخ‌های تحویل غیر واقع بینانه وادار می‌سازد، در نتیجه‌ی میان‌برهایی زده می‌شود، ممکن است فعالیت‌هایی جا انداخته شوند که منجر به کیفیت بالاتر نرم‌افزار می‌شوند و به کیفیت محصول خدشه وارد آید. اگر تاریخ تحویل ناموجه بود، از مواضع خود دفاع کنید. توضیح دهید که چرا به زمان بیشتری نیاز دارید، یا به طریق دیگر، زیرمجموعه‌ای از قابلیت‌های عملیاتی را پیشنهاد کنید که در زمان مشخص شده (با کیفیت بالا) قابل تحویل باشند.

تصمیم‌گیری‌های زمان‌بندی. هنگامی که زمان‌بندی یک پروژه نرم‌افزاری تعیین شد (فصل ۲۷)، ترتیب وظایف بر اساس وابستگی‌ها تعیین می‌شوند. برای مثال، از آن‌جا که مؤلفه A به پردازشی وابسته است که در مؤلفه‌های B، C و D رخ می‌دهد، زمان‌بندی برای آزمودن مؤلفه A امکان‌پذیر نخواهد بود تا این که مؤلفه‌های B، C و D به‌طور کامل آزموده شوند. زمان‌بندی پروژه این را نشان خواهد داد. ولی اگر زمان بسیار کوتاه باشد و A باید برای آزمون‌های حیاتی بیشتر در دسترس باشد، ممکن است تصمیم بگیرید که A را بدون مؤلفه‌های زیردست آن (که قدری از برنامه عقب هستند) بیازمایید، به‌طوری که آن را برای آزمون‌های بعدی که باید قبل از تحویل انجام شوند، در دسترس قرار دهید. به هرحال موعد مقرر فرا می‌رسد و در نتیجه، A ممکن است نقایصی داشته باشد که پنهان بمانند و فقط مدت‌ها بعد کشف شوند. کیفیت، خدشه‌دار می‌شود.

تصمیم‌گیری‌های مربوط به ریسک. مدیریت ریسک (فصل ۲۸) یکی از صفات کلیدی پروژه‌های نرم‌افزاری موفق است. شما واقعاً نمی‌دانید که چه چیزهایی ممکن است خراب شود و از پیش، خود را برای آن آماده کنید. بسیاری از تیم‌های نرم‌افزاری، خوش‌بینی کورکورانه را ترجیح می‌دهند و برای پیشرفت کار، زمان‌بندی‌ای را تعیین می‌کنند، با این فرض که همه چیز به خوبی پیش خواهد رفت. بدتر این که برای شرایط ناگوار هیچ چیزی در دست ندارند. در نتیجه، هنگامی که ریسک واقعیت پیدا می‌کند، آشوب حکمفرما می‌شود و با افزایش درجه‌ی بی‌نظمی، سطح کیفیت ناگزیر افت می‌کند.

معضل کیفیت نرم‌افزار را به بهترین نحو می‌توان با بیان قانون مسکیمن خلاصه کرد- هرگز وقت برای انجام درست کار وجود ندارد، ولی همواره برای دوباره کاری وقت هست. نصیحت من: شتاب نکردن برای انجام درست کارها، تقریباً هرگز تصمیم اشتباهی نیست.

۱۴-۴ دستیابی به کیفیت نرم‌افزار

کیفیت نرم‌افزار چیزی نیست که یک باره ظاهر شود. نتیجه‌ی مدیریت خوب، پروژه و کار مهندسی نرم‌افزار مستحکم است. مدیریت و کار در حیطه‌ی چهار فعالیت گسترده است که تیم نرم‌افزاری را در دستیابی به کیفیت بالای نرم‌افزار یاری می‌دهد: روش‌های مهندسی نرم‌افزار، تکنیک‌های مدیریت پروژه، کنش‌های کنترل کیفیت و تضمین کیفیت نرم‌افزار.

۱۴-۴-۱ روش‌های مهندسی نرم‌افزار

اگر انتظار دارید نرم‌افزاری با کیفیت بالا بسازید، باید مسأله‌ای را که قرار است حل شود، خوب درک کنید. همچنین طراحی شما باید طوری باشد که با مسأله همخوانی داشته باشد، در حالی که همزمان،

ویژگی‌هایی را از خود به نمایش بگذارد که نتیجه‌ی آن‌ها نرم‌افزاری با ابعاد و عوامل کیفیتی بحث شده در بخش ۲-۱۴ باشد.

۲-۴-۱۴ تکنیک‌های مدیریت پروژه

تأثیر ابعاد مدیریتی ضعیف بر کیفیت نرم‌افزار در بخش ۶-۳-۱۴ بحث شد. همه چیز واضح است: اگر (۱) مدیر پروژه از برآوردها استفاده کند تا ببیند آیا تاریخ‌های تحویل قابل تحقق هستند، (۲) وابستگی‌های زمان‌بندی درک شده باشد و تیم در برابر وسوسه‌ی استفاده از میان‌برها مقاومت کند، (۳) برنامه‌ریزی برای ریسک انجام شده باشد، به‌طوری که مسائل، تولید آشوب نکنند، کیفیت نرم‌افزار را می‌توان به نحوی مثبت تحت تأثیر قرار داد.

به علاوه، برنامه‌ریزی پروژه باید شامل تکنیک‌های صریح برای مدیریت کیفیت و تغییر باشد. تکنیک‌هایی که به کارهای خوب در زمینه مدیریت پروژه منجر می‌شوند، در بخش ۴ این کتاب بحث خواهند شد.

چه باید بکنم تا بر کیفیت تأثیری مثبت بگذارم؟

۳-۴-۱۴ کنترل کیفیت

کنترل کیفیت شامل مجموعه‌ای از کنش‌های مدیریت نرم‌افزار می‌شود که به کمک آن‌ها می‌توان اطمینان حاصل کرد که هر محصول کاری، اهداف کیفیتی‌اش را برآورده ساخته است. مدل‌ها بازبینی می‌شوند تا اطمینان حاصل شود که کامل و سازگارند. کدها را می‌توان وارسی کرد تا خطاها قبل از شروع آزمون، کشف و تصحیح شوند. یک سری مراحل آزمون به‌کار برده می‌شوند تا خطاهای موجود در منطق پردازش، دستکاری داده‌ها و ارتباط میان واسط‌ها برملا شود. ترکیبی از اندازه‌گیری و بازخورد به تیم این امکان را می‌دهد که وقتی هر کدام از محصولات کاری نتوانست به اهداف، کیفیتی دست پیدا کند، فرایند را تنظیم کند. فعالیت‌های کنترل کیفیت در سرتاسر بخش سوم این کتاب مورد بحث قرار خواهند گرفت.

کنترل کیفیت نرم‌افزار چیست؟

۴-۴-۱۴ تضمین کیفیت

تضمین کیفیت، زیرساختی را تعیین می‌کند که روش‌های مهندسی نرم‌افزار، مدیریت پروژه موجه و کنش‌های کنترل کیفیت را- که همگی در ساخت نرم‌افزارهای با کیفیت بالا اهمیت محوری دارند- پشتیبانی می‌کند. به علاوه، تضمین کیفیت شامل یک مجموعه وظایف ممیزی و گزارش‌دهی می‌شود که اثربخش بودن و کامل بودن کنش‌های کنترل کیفیت را ارزیابی می‌کنند. هدف تضمین کیفیت، فراهم ساختن داده‌های لازم در خصوص کیفیت محصول برای مدیران و کارکنان فنی است تا به این ترتیب، اطمینان پیدا کنند که کنش‌های مربوط به‌دستیابی به کیفیت محصول، اثربخش هستند. البته، اگر داده‌های فراهم آمده از طریق تضمین کیفیت، مشکلات را برملا سازند، مسؤولیت مدیریت است که به این مشکلات بپردازد و منابع لازم برای حل مسائل کیفیتی را به‌کار گیرد. تضمین کیفیت نرم‌افزار را به تفصیل در فصل ۱۶ مورد بحث قرار خواهیم داد.

مرجع وب

پیوندهای مفیدی به SOA را می‌توانید در آدرس زیر بیابید:
www.niwotridge.com/Resources/PM-SWEResources/SoftwarQualityAssurance.htm

۵-۱۴ خلاصه

دغدغه برای کیفیت سیستم‌های مبتنی بر نرم‌افزار با رسوخ نرم‌افزار در تمامی شؤونات زندگی روزمره ما رشد کرده است. ولی ارائه و بسط توصیف جامع و فراگیری از کیفیت نرم‌افزار دشوار است. در این

فصل، کیفیت به‌عنوان یک فرایند نرم‌افزار اثربخش تعریف شده است که طوری به‌کار برده می‌شود که محصول آن ارزشی قابل سنجش برای تولید کنندگان و نیز برای استفاده کنندگان از آن فراهم می‌آورد.

گستره وسیعی از عوامل و ابعاد کیفیت نرم‌افزار طی سال‌ها پیشنهاد شده است. همه تلاش دارند مجموعه‌ای از خصوصیت‌ها را تعریف کنند که در صورت دستیابی به آن‌ها، نتیجه، کیفیت بالای نرم‌افزار خواهد بود. عوامل کیفیتی مک کال و ISO9126 خصوصیاتی نظیر قابلیت اطمینان، قابلیت استفاده، قابلیت نگهداری، قابلیت عملیاتی و حمل‌پذیری را به‌عنوان شاخص‌هایی معرفی می‌کنند که وجود کیفیت را نشان می‌دهند.

هر سازمان نرم‌افزاری با معضل کیفیت نرم‌افزار مواجه است. در اصل، همه می‌خواهند سیستم‌هایی با کیفیت بالا بسازند، ولی زمان و تلاش لازم برای تولید نرم‌افزار «کامل» در دنیایی که بازار آن را به پیش می‌راند، غیر قابل دستیابی است. سؤال این خواهد بود که آیا باید نرم‌افزاری بسازیم که «به‌قدر کافی خوب» باشد؟ گرچه، بسیاری شرکت‌ها دقیقاً چنین می‌کنند، تبعات چشمگیری وجود دارد که باید آن‌ها را مد نظر داشت.

فارغ از رویکردی که استفاده می‌شود، کیفیت بی‌تردید هزینه بردار است و می‌توان آن را بر حسب پیش‌گیری، ارزیابی و شکست مورد بحث قرار داد. هزینه‌های پیش‌گیری شامل همه‌ی کنش‌های مهندسی نرم‌افزار می‌شوند که به منظور جلوگیری از ایجاد نقایص در وهله‌ی نخست طراحی می‌شوند. هزینه‌های ارزیابی به کنش‌هایی مربوط می‌شوند که محصولات کاری نرم‌افزار را ارزیابی می‌کنند تا کیفیت آن‌ها را تعیین نمایند. هزینه‌های شکست شامل قیمت داخلی شکست و اثرات خارجی می‌شوند که کیفیت ضعیف از خود به جای می‌گذارد.

کیفیت نرم‌افزار از طریق به‌کار بردن روش‌های مهندسی نرم‌افزار، کارهای مدیریتی مستحکم و کنترل کیفیتی فراگیر به‌دست می‌آید- که همگی توسط یک زیرساخت تضمین کیفیت نرم‌افزار پشتیبانی می‌شوند. در فصل‌های بعد، کنترل کیفیت و تضمین کیفیت با قدری تفصیل بحث خواهد شد.

مسائل و نکاتی برای تعمق

۱-۱۴ شرح دهید که کیفیت یک دانشگاه را قبل از درخواست ورود به آن چگونه ارزیابی می‌کنید. چه عواملی برای شما اهمیت خواهد داشت؟ کدام عوامل، حیاتی‌اند؟

۲-۱۴ گاروین [Gar84] پنج دیدگاه متفاوت درباره کیفیت شرح می‌دهد. با به‌کارگیری یک یا چند محصول الکترونیکی که با آن‌ها آشنا هستید، برای هر دیدگاه مثال بیاورید.

۳-۱۴ با استفاده از تعریف کیفیت نرم‌افزار که در بخش ۲-۱۴ پیشنهاد شد، آیا تصور می‌کنید بتوان محصول مفیدی تولید کرد که ارزش قابل سنجشی را بدون استفاده از یک فرایند اثربخش فراهم نماید؟

۴-۱۴ به هر کدام از ابعاد کیفیتی گاروین که در بخش ۱-۲-۱۴ ارائه شد، دو پرسش دیگر اضافه کنید.

۵-۱۴ عوامل کیفیتی مک کال طی دهه ۱۹۷۰ توسعه یافتند. تقریباً همه‌ی جنبه‌های کار با کامپیوتر از زمانی که توسعه یافتند، تغییر پیدا کرده‌اند و با این حال، عوامل مک کال همچنان برای نرم‌افزارهای مدرن کاربرد دارند. آیا می‌توانید بر اساس این واقعیت، نتیجه‌ای بگیرید.

۶-۱۴ با به‌کارگیری صفات ذکر شده برای عامل کیفیتی ISO9126 «قابلیت نگهداری» در بخش ۳-۲-۱۴، یک مجموعه پرسش تهیه کنید که وجود یا نبود این صفات را به کمک آن‌ها بتوان بررسی کرد. از مثال نشان داده شده در بخش ۴-۲-۱۴ پیروی کنید.

۷-۱۴ معضل کیفیت نرم‌افزار را به زبان ساده شرح دهید.

۱۴-۸ نرم‌افزار «به‌قدر کافی خوب» چیست؟ یک شرکت مشخص و محصولات مشخصی را نام ببرید که معتقدید با استفاده از فلسفه «به‌قدر کافی خوب» تهیه شده‌اند.

۱۴-۹ با در نظر گرفتن هر کدام از چهار جنبه‌ی هزینه کیفیت، تصور می‌کنید کدام یک بیشترین هزینه را در بر دارد و چرا؟

۱۴-۱۰ در وب جستجو کنید و سه مثال دیگر از ریسک‌هایی را بیابید که از سوی کیفیت ضعیف نرم‌افزار ممکن است متوجه عموم مردم شود. جستجوی خود را از http://catless.ncl.ac.uk/risks می‌توانید آغاز کنید.

۱۴-۱۱ آیا *کیفیت* و *امنیت* یک مقوله‌اند؟ توضیح دهید.

۱۴-۱۲ توضیح دهید چرا بسیاری از ما همچنان طبق قانون مسکیمن زندگی می‌کنیم. تجارت نرم‌افزار چه خاصیتی دارد که باعث این امر می‌شود؟ شیوه‌های متفاوت نگرش به کیفیت کدام‌اند؟

فصل ۱۵

تکنیک‌های مرور نرم‌افزار

نگاهی گذرا

مرور چیست؟ شما هنگام توسعه‌ی محصولات کاری مهندسی نرم‌افزار، مرتکب اشتباه می‌شوید و جای شرمندگی هم ندارد، البته مشروط بر آن‌که حداکثر تلاش خود را کرده باشید که خطاها را پیش از تحویل محصول به کاربران، تصحیح کنید. مرورهای فنی، اثربخش‌ترین سازوکار برای یافتن زودهنگام خطاها در فرایند نرم‌افزار به شمار می‌روند.

چه کسی این کار را انجام می‌دهد؟ مهندسان نرم‌افزار همراه با همکاران خود، مرورهای فنی را انجام می‌دهند که از آن‌ها به‌عنوان مرورهای فنی یا مرورهای نظیر یاد می‌شود.

چرا اهمیت دارد؟ اگر خطای موجود در فرایند را زودهنگام بیابید، تصحیح آن، هزینه‌ی کمتری در بر خواهد داشت. به علاوه، طبیعت خطاها به گونه‌ای است که با پیشرفت فرایند، قوت می‌گیرند. بنابراین، یک خطای نسبتاً جزئی که در اوایل فرایند بر طرف نشده باشد، بعداً در پروژه می‌تواند به مجموعه‌ای از خطاها منجر گردد. سرانجام، این مرورها با کاستن از مقدار دوباره‌کاری‌هایی که ممکن است در آینده ضرورت پیدا کنند، باعث صرفه‌جویی در زمان خواهند شد.

مراحل کار کدام است؟ رویکرد شما در قبال مرورها بسته به درجه‌ی رسمیتی که بر می‌گزینید، متغیر است. به‌طور کلی، شش مرحله به‌کار می‌رود، هرچند که همه‌ی آن‌ها برای همه‌ی انواع مرور به‌کار نمی‌رود: برنامه ریزی، آماده سازی، سازمان‌دهی به جلسات، ذکر خطاها، انجام تصحیحات (که خارج از مرور انجام می‌شود) و وارسی درستی انجام تصحیحات.

محصول کاری چیست؟ خروجی مرور، فهرستی از مسائل و/یا خطاهاست که کشف نشده‌اند. به علاوه، وضعیت فنی محصول کاری نیز ذکر می‌شود.

چگونه اطمینان حاصل کنم که درست از انجام کارها بر آمده‌ام؟ نخست نوع مرور مناسب برای فرهنگ خود را برگزینید و دستور العمل‌های منتهی به مرورهای موفق را دنبال کنید. اگر مرورهایی که اجرا می‌کنید به نرم‌افزار با کیفیت بالاتری منجر شود، کار را درست انجام داده‌اید.

مرورهای نرم‌افزار به‌مثابه فیلترهایی برای فرایند مهندسی نرم‌افزار عمل می‌کنند. یعنی در نقاط گوناگونی از توسعه نرم‌افزار اِعمال می‌شوند و به کشف خطاها و نقایصی که قابل رفع باشند، کمک می‌کنند. مرورهای نرم‌افزار به «خالص‌سازی» فعالیت‌های مهندسی نرم‌افزار که آنها را تحلیل، طراحی و کدنویسی نامیدیم، کمک می‌کنند. فریدمن و واینبرگ [Fre90] نیاز به مرور را چنین مورد بحث قرار می‌دهند:

> کار فنی به همان دلیل نیاز به مرور دارد که مداد نیاز به پاک‌کن دارد. انسان جایزالخطاست. دلیل دوم بر نیاز به مرور آن است که گرچه افراد در یافتن برخی خطاهای خود وارد هستند، گروه بزرگی از خطاها هستند که از دست کسی که مرتکب آنها می‌شود، به‌مراتب، آسانتر از دست بقیه افراد در می‌روند. پس فرایند مرور پاسخی به دعای رابرت برنز است:
>
> خداوندا قدرتی به من عطا فرما که خود را چنان ببینم که دیگرانم می‌بینند.

مرور - از هر نوع که باشد - راهی برای استفاده‌ی عده‌ای از افراد، برای مقاصد زیر است:

۱. اشاره به بهسازی‌های لازم در محصول یک فرد یا تیم.
۲. تأیید بخش‌هایی از محصول که در آن بهسازی یا لازم نیست یا موردنظر نیست.
۳. انجام کارهای فنی یکنواخت، یا حداقل قابل پیش‌بینی‌تر و کیفیتی که بدون مرور قابل حصول است.

انواع بسیاری از مرور وجود دارد که به‌عنوان بخشی از مهندسی نرم‌افزار قابل اجراست. هر یک از آنها جایگاه خاص خود را دارد، اگر در یک ملاقات غیررسمی در آبدارخانه، مشکلات فنی مورد بحث قرار گیرند، این خود شکلی از مرور است. ارائه رسمی طراحی نرم‌افزار به مخاطبی از مشتریان، مدیریت و کارمندان فنی نیز شکلی از مرور است. در این کتاب، مرورهای فنی را کانون توجه قرار خواهیم داد که نمونه‌های مختلف آن عبارتند از *مرورهای اتفاقی* (Casual)، *مرورهای فنی رسمی* (Walkthroughs) و *بازرسی‌ها* (Inspections). مرور فنی رسمی (FTR)، از دیدگاه تضمین کیفیت مؤثرترین فیلتر است. FTR که توسط مهندسان نرم‌افزار (و دیگران) انجام می‌شود، ابزاری مؤثر برای بهبود بخشیدن به کیفیت نرم‌افزار است.

اندرز

مرورها در جریان کاری فرایند نرم‌افزار همانند فیلتر عمل می‌کنند. اگر تعداد آن‌ها بیش از حد کم باشد، جریان «کثیف» می‌شود و اگر تعداد آن‌ها بیش از حد زیاد باشد، جریان تا حد یک «آب‌بازیکه» کاهش می‌یابد. از معیارها استفاده کنید تا تعیین کنید که کدام مرورها جواب می‌دهند و بر آن‌ها تأکید کنید. مرورهایی را که فاقد اثربخشی هستند، از جریان حذف کنید تا فرایند شتاب بگیرد.

۱۵-۱ تأثیر نقایص نرم‌افزار بر هزینه‌ها

در حیطه‌ی فرایند نرم‌افزار، واژه‌های *نقص* و *عیب* مترادف هستند. هر دو تداعی‌گر مشکلی هستند که پس از ارائه نرم‌افزار به کاربر نهایی (یا فعالیت دیگری در فرایند نرم‌افزار) کشف می‌شوند. در فصول اولیه، از واژه خطا برای مشکلات کیفیتی استفاده کردیم که توسط مهندسان نرم‌افزار (یا دیگران) پیش از ارائه نرم‌افزار به کاربر نهایی (یا فعالیت دیگری در فرایند نرم‌افزار) کشف می‌شوند.

هدف اصلی مرورهای فنی رسمی، یافتن خطاها در اثنای فرایند است به‌طوری که پس از ارائه نرم‌افزار به نقص تبدیل نشوند. مزیت آشکار مرورهای فنی رسمی، کشف زودهنگام خطاهاست، به‌طوری که به مرحله بعدی فرایند نرم‌افزار انتشار پیدا نکنند.

چند مطالعه‌ی صنعتی نشان می‌دهد که فعالیت‌های طراحی بین ۵۰ تا ۶۵٪ خطاها (و نهایتاً همه‌ی نقایص) را در اثنای فرایند نرم‌افزار باعث می‌شوند. ولی ثابت شده است که تکنیک‌های مرور رسمی

اندرز

هدف اصلی یک FTR یافتن خطاها قبل از منتقل شدن به یک فعالیت مهندسی نرم‌افزار دیگر یا رسیدن به‌دست کاربر نهایی است.

اطلاعات

اشکال‌ها، خطاها و نقایص

هدف از کنترل کیفیت نرم‌افزار و از دیدگاهی گسترده‌تر، مدیریت کیفیت به‌طور کل، حل مشکلات کیفیتی نرم‌افزار است. از این مشکلات با نام‌های گوناگون(*اشکال*، *خطا*، *نقص*) یاد می‌شود. آیا این اصطلاحات، مترادف هستند یا تفاوت ظریفی میان آن هاست؟

در این کتاب میان *خطا* (*error*، یک مشکل کیفیتی که قبل از ارائه نرم‌افزار به کاربران نهایی یافته می‌شود) و *نقص* (*defeat*، یک مشکل کیفیتی که تنها پس از ارائه نرم‌افزار به کاربران نهایی کشف می‌شود[1]) تفاوت قائل می‌شویم. این تصمیم از آن رو گرفته شده است که خطاها و نقص‌ها تأثیرات اقتصادی، تجاری، روان شناختی و انسانی بسیار متفاوتی دارند. به‌عنوان مهندس نرم‌افزار مایل هستیم حداکثر تعداد ممکن خطاها را پیش از برخورد مشتری و/یا کاربر نهایی کشف کنیم. می‌خواهیم از نقایص پرهیز کنیم- چون نقایص باعث بدنامی نرم‌افزار نویسان می‌شوند. به هر حال، شایان ذکر است که این تمایز قائل شدن میان خطا و نقص در کتاب حاضر، تفکری همه جایی نیست. اجماع عمومی در جامعه‌ی مهندسی نرم‌افزار از این قرار است که خطا، نقص و اشکال همگی مترادف یکدیگرند. یعنی نقطه‌ای از زمان که مشکل در آن مشاهده می‌شود، تأثیری بر اصطلاح به‌کاررفته برای توصیف مسأله ندارد.

در این خصوص استدلال می‌شود که تمایز میان قبل و بعد از ارائه نرم‌افزار چندان آسان نیست. این اصطلاحات را هرگونه که تفسیر می‌کنید، باید بدانید که زمان کشف مشکل بسیار مهم است و مهندسان نرم‌افزار باید سخت- بسیار سخت- بکوشند تا مشکلات را پیش از آن که مشتریان و کاربران نهایی به آن برخورد کنند، کشف کنند. در صورت علاقه‌ی بیشتر به این موضوع، بحث کاملی از اصطلاح‌شناسی در خصوص «اشکال‌ها» را در آدرس زیر می‌توانید بیابید:

www.softwaredevelopment.ca/bugs.shtml

تا ۷۵٪ در کشف معایب طراحی مؤثر واقع می‌شوند [Jon86]. فرایند مرور با یافتن و حذف درصد بزرگی از این خطاها، هزینه‌ی مراحل بعدی را در فازهای توسعه و پشتیبانی، به‌طور چشمگیر کاهش می‌دهد.

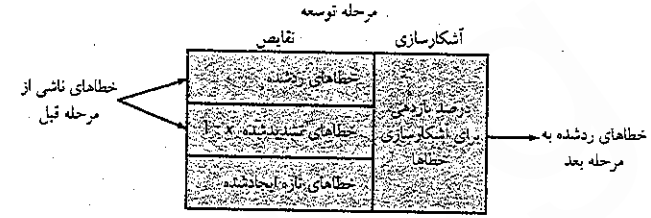
۱۵-۲ تشدید نقایص و حذف آنها

از *مدل تشدید نقص* [IBM81] می‌توان برای نمایش تولید و یافتن خطاها طی طراحی مقدماتی، طراحی مشروح و مراحل کدنویسی در فرایند مهندسی نرم‌افزار استفاده کرد. این مدل به‌طور شماتیک در شکل ۱-۱۵ نشان داده شده است. چهارگوش خاکستری نشان‌گر یک مرحله توسعه نرم‌افزار است. طی این مرحله، خطاها ممکن است به‌طور ناخواسته تولید شوند. مرور ممکن است از کشف خطاهای تازه تولید شده، و خطاهای مراحل پیشین باز بماند و درنتیجه چند خطا به مرحله بعدی راه پیدا کنند. در برخی موارد، خطاهایی که از مراحل قبلی عبور می‌کنند توسط کار فعلی تشدید می‌شوند (با ضریب تشدید x). در تقسیمات فرعی این چهارگوش، هر یک از ویژگی‌ها و درصد بازدهی برای یافتن خطا که تابعی از کامل بودن مرور است، نشان داده شده است.

[1] اگر بهبود فرایند نرم افزار مد نظر باشد، مشکلی کیفیتی را که از یک فعالیت چارچوبی فرایند (مثلا مدل‌سازی) به فعالیت چارچوبی دیگر مثلاً ساخت) منتشر می شود نیز می‌توان «نقص» نامید (چون این مشکل می‌بایست قبل از «تحویل» یک محصول کاری به فعالیت دیگر یافته می‌شده است.)

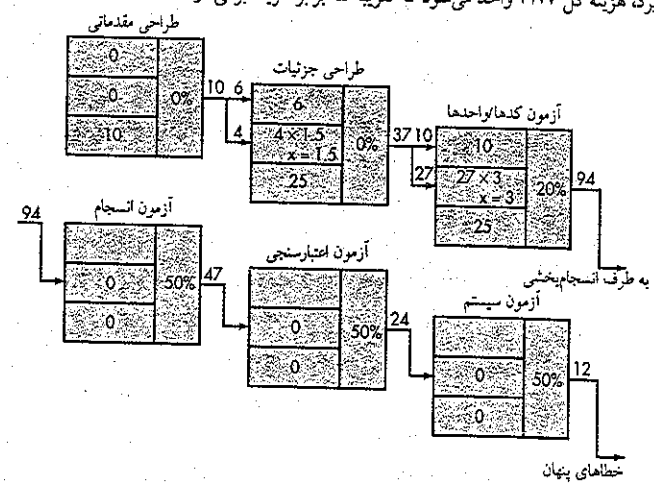
«برخی امراض، آن سان که طبیبان می‌گویند، در بدو امر به سهولت قابل معالجه‌اند، ولی دشوار می‌توان آن‌ها را تشخیص داد... اما اگر از همان آغاز، تشخیص داده و درمان نشده باشند، با گذر زمان، تشخیص آن‌ها آسان، ولی معالجه‌ی آن‌ها دشوار خواهد شد.»

نیکولو ماکیاولی



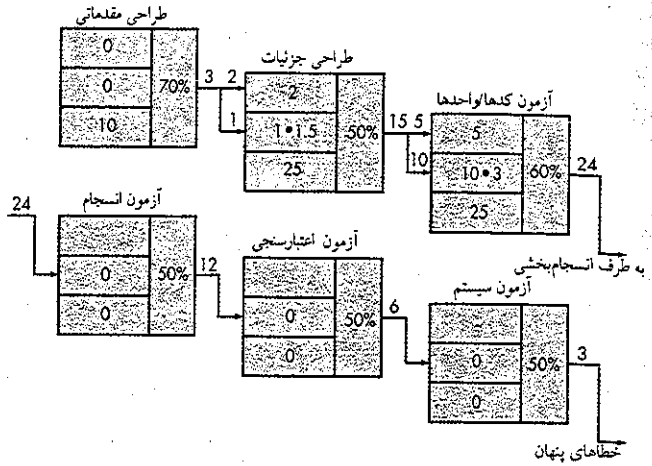
شکل ۱-۱۵ مدل تشدید نقایص.

در شکل ۲-۱۵ یک مثال فرضی از تشدید نقص برای فرایند توسعه نرم‌افزار نشان داده شده است که در آن هیچ مروری صورت نمی‌پذیرد. در این شکل فرض شده است که در هر مرحله ۵۰٪ از کلیه خطاهای وارد شده کشف می‌شوند، بدون اینکه خطای جدیدی وارد شود (یک فرض بهینه). ده نقص طراحی مقدماتی پیش از شروع آزمون تا ۹۴ خطا تشدید می‌شوند. دوازده خطای نهفته وارد میدان می‌شود. در شکل ۳-۱۵ همین شرایط فرض می‌شود، با این تفاوت که مرورهای طراحی و کدها به‌عنوان بخشی از هر مرحله‌ی توسعه انجام می‌شود. در این مورد، ده خطای طراحی اولیه پیش از شروع آزمون، تا ۲۴ خطا تشدید می‌شوند. فقط سه خطای نهفته وجود دارد. با به خاطر آوردن هزینه‌های نسبی کشف و تصحیح خطاها، هزینه کل (با مرور مثال فرضی ما و بدون مرور آن) را می‌توان تعیین کرد. تعداد خطاهای کشف شده طی هر یک از مراحل ذکر شده در شکل‌های ۲-۱۵ و ۳-۱۵، در هزینه لازم برای حذف یک خطا ضرب می‌شود (۱/۵ واحد برای طراحی، ۶/۵ واحد پیش از آزمون و ۱۵ واحد حین آزمون و ۶۷ واحد پس از آزمون)[1]. با استفاده از این داده‌ها، هزینه‌ی کل توسعه و نگهداری در هنگام اجرای مرورها، ۷۸۳ واحد می‌شود. هنگامی که هیچ مروری صورت نپذیرد، هزینه کل ۲۱۷۷ واحد می‌شود که تقریباً سه برابر هزینه برمی‌دارد.



شکل ۲-۱۵ تشدید نقایص- بدون مرور.

[1] این ضرایب با داده‌های ارائه شده در شکل ۲-۱۴ که نسبتاً جدیدتر است، قدری تفاوت دارند. ولی به خوبی به نشان‌دادن هزینه تشدید نقایص کمک می‌کنند.



شکل ۳-۱۵ تشدید نقایص- مرورهای انجام شده.

مهندس نرم‌افزار برای اجرای مرورها باید انرژی و زمان صرف کند و سازمان توسعه باید پول خرج کند. ولی، نتایج مثال قبلی، دیگر کوچکترین تردیدی را برجای نمی‌گذارد، می‌توان حالا پرداخت کرد یا بعداً خیلی بیشتر پرداخت.

۳-۱۵ معیارهای مرور و کاربرد آن‌ها

مرورهای فنی از جمله چندین کنشی هستند که به‌عنوان بخشی از کار مهندسی نرم‌افزار خود به شمار می‌روند. هر کنش نیاز به تلاش انسانی دارد. چون منابع پروژه متناهی است، مهم است که سازمان مهندسی نرم‌افزار با تعریف یک مجموعه معیار (فصل ۲۳) که در ارزیابی به‌کار می‌روند، اثربخشی و مؤثر بودن هر کنش را بداند.

گرچه برای مرورهای فنی، معیارهای بسیاری را می‌توان به‌کار برد، یک زیرمجموعه نسبتاً کوچک می‌تواند دید مفیدی فراهم سازد. معیارهای مرور زیر را می‌توان برای هر کدام از مرورهای اجراشده جمع‌آوری کرد:

- *تلاش آماده سازی*، E_p - تلاش و کار لازم (بر حسب نفر-ساعت) برای مرور یک محصول کاری قبل از جلسه مرور واقعی.
- *تلاش ارزیابی*، E_a - تلاش صرف شده (بر حسب نفر-ساعت) طی مرور واقعی.
- *تلاش دوباره کاری*، E_r - تلاش اختصاص داده شده (بر حسب نفر-ساعت) به تصحیح آن دسته از خطاهایی که طی مرور کشف می‌شوند.
- *اندازه محصول کاری*، *WPS* - میزانی از اندازه محصول کاری که مرور شده است (مثلاً *تعداد* مدل‌های UML، یا تعداد صفحات مستندات یا تعداد خطوط کد).
- *خطاهای جزئی یافته شده*، Err_{minor} - تعداد خطاهای یافت شده که می‌توان آن‌ها را در زمره‌ی خطاهای جزئی دسته‌بندی کرد (تلاش لازم *برای* تصحیح آن‌ها از یک مقدار تعیین شده کوچک‌تر است).

- *خطاهای عمده‌ی یافته شده*، Err_{major}-تعداد خطاهای یافت شده که می‌توان آن‌ها را در زمره خطاهای عمده *دسته‌بندی* کرد (تلاش لازم برای تصحیح آن‌ها از یک مقدار تعیین‌شده، بزرگ‌تر است).

این معیارها را با مشخص کردن نوع محصول کاری که برای معیارهای جمع‌آوری شده مرور شده است، باز هم می‌توان پالایش کرد.

۱-۳-۱۵ تحلیل معیارها (Analyzing Metrics)

پیش از شروع تحلیل، به چند محاسبه‌ی ساده نیاز است. تلاش مرور کل و تعداد کل خطاهای کشف‌شده به‌صورت زیر تعریف می‌شوند:

$$E_{review} = E_p + E_a + E_r$$

$$Err_{tot} = Err_{minor} + Err_{major}$$

چگالی خطا نشان‌گر خطاهای یافته شده به‌ازای واحد محصول کاری مرور شده است:

$$\text{چگالی خطا} = \frac{Err_{tot}}{WPS}$$

برای مثال، اگر مدل خواسته‌ها برای برملا ساختن خطاها، ناسازگاری‌ها و جا افتادگی‌ها مرور شود، محاسبه‌ی چگالی خطا به چند روش متفاوت امکان‌پذیر است. مدل خواسته‌ها در یک مجموعه مستندات ۳۲ صفحه‌ای حاوی ۱۸ نمودار UML است. پس از مرور، ۱۸ خطای جزئی و ۴ خطای عمده کشف شد. بنابراین، $Err_{tot} = 22$. چگالی خطا، ۱٫۲ خطا به‌ازای هر نمودار UML یا ۰٫۶۸ خطا به‌ازای هر صفحه از مدل خواسته‌ها می‌شود.

اگر مرورها برای چند نوع محصول کاری متفاوت اجرا شوند (مثلاً مدل خواسته‌ها، مدل طراحی، کد، موارد آزمون)، درصد خطاهای کشف شده برای هر مرور را می‌توان برحسب تعداد خطاهای یافته‌شده برای تمامی مرورها محاسبه کرد. به‌علاوه، چگالی خطا برای هر محصول کاری قابل محاسبه است.

هنگامی که داده‌ها برای چندین مرور اجرا شده در پروژه‌های مختلف جمع‌آوری شوند، به کمک مقادیر میانگین چگالی خطا می‌توانید تعداد خطاهایی را که ممکن است در یک سند جدید کشف شود، پیش از مرور آن، برآورد کنید. برای مثال، اگر چگالی خطای میانگین برای یک مدل خواسته‌ها، ۰٫۶ خطا به‌ازای هر صفحه باشد و مدل خواسته‌های جدید ۳۲ صفحه باشد، به‌عنوان برآوردی حدودی، پیش‌بینی می‌شود که تیم مرور، طی مرور این سند، ۱۹ یا ۲۰ خطا کشف کند. اگر تنها ۶ خطا پیدا کنید، یا کار خود را در تهیه‌ی مدل خواسته‌ها بسیار خوب انجام داده‌اید یا این که رویکرد مرور شما به‌قدر کافی کامل نبوده است. پس از این‌که آزمون انجام شد (فصل‌های ۱۷ تا ۲۰) جمع‌آوری داده‌های اضافی درخصوص خطاها، از جمله تلاش لازم برای یافتن و تصحیح خطاهای کشف شده طی آزمون و چگالی خطای نرم‌افزار نیز امکان‌پذیر است. هزینه‌های مرتبط با یافتن و تصحیح خطا طی آزمون را می‌توان با مرورها مقایسه کرد. این موضوع در بخش ۲-۳-۱۵ بحث شده است.

۲-۳-۱۵ اثربخشی هزینه‌ی مرورها

اندازه‌گیری اثربخشی هزینه‌ی هر مرور فنی به‌صورت همزمان، دشوار است. یک سازمان مهندسی نرم‌افزار می‌تواند اثربخشی مرورها و عایدی حاصل از صرف هزینه در این بخش را تنها پس از پایان مرورها، جمع‌آوری معیارها، محاسبه داده‌های میانگین و سپس اندازه‌گیری کیفیت پایین دستی (از طریق انجام آزمون) ارزیابی کند.

با توجه به مثال ارائه شده در بخش ۱-۳-۱۵، چگالی خطای میانگین برای مدل خواسته‌ها برابر با ۰٫۶ تعیین شد. تلاش لازم برای تصحیح یک خطای جزئی در مدل خواسته‌ها (بلافاصله پس از مرور) برابر با ۴ نفر-ساعت به‌دست آمد. تلاش لازم برای تصحیح یک خطای عمده در مدل خواسته‌ها نیز برابر با ۱۸ نفر-ساعت به‌دست آمد. با بررسی داده‌های جمع‌آوری شده می‌توانید دریابید که فراوانی رخ‌دادن خطاهای جزئی تقریباً شش برابر بیشتر از رخ‌دادن خطاهای عمده است. بنابراین، می‌توانید برآورد کنید که تلاش میانگین برای یافتن و تصحیح خطایی در مدل خواسته‌ها طی مرور، حدوداً ۶ نفر-ساعت می‌شود.

خطاهای مرتبط با خواسته‌ها که طی آزمون برملا می‌شوند، برای یافتن و تصحیح به‌طور میانگین به ۴۵ نفر-ساعت تلاش نیاز دارند (درباره شدت نسبی خطاها هیچ گونه داده‌ای در اختیار نداریم). با به‌کارگیری میانگین‌های ذکر شده داریم:

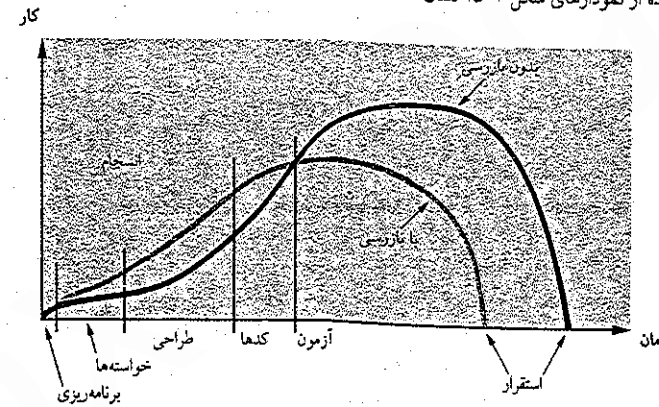
$$\text{تلاش صرفه‌جویی‌شده به‌ازای هر خطا} = E_{testing} - E_{reviews}$$

$$= 45 - 6 = 30 \text{ (نفر ساعت به‌ازای هر خطا)}$$

از آن‌جا که طی مرور مدل خواسته‌ها، ۲۲ خطا کشف شده است، در کل حدود ۶۶۰ نفر ساعت در تلاش برای آزمون، صرفه‌جویی می‌شود و این تنها برای خطاهای مربوط به خواسته‌هاست. خطاهای مرتبط با طراحی و کدنویسی نیز به این مزیت کل اضافه می‌شود. خلاصه‌ی کلام این که صرفه‌جویی در تلاش‌ها به چرخه‌های تحویل کوتاه‌تر و بهبود بخشیدن به زمان تحویل به بازار منجر خواهد شد.

کارل ویگرز در کتاب خود در باب مرورهای نظیر [Wie02] درباره داده‌های پرحرف‌وحدیث مربوط به شرکت‌های بزرگی بحث می‌کند که از وارسی (یک نوع نسبتاً رسمی از مرور) به‌عنوان بخشی از فعالیت‌های کنترل کیفیت خود استفاده کرده‌اند. هیولت پاکارد، عایدی ده به یک را برای وارسی‌ها گزارش کرده است و گفته است که تحویل محصول واقعی به‌طور میانگین حدود ۱٫۸ ماه تسریع یافته است. AT&T خاطر نشان ساخته است که وارسی‌ها در کل، هزینه‌های ناشی از خطاها را ده برابر کاهش داده‌اند، کیفیت ده برابر بهتر شده است و بهره وری به میزان ۱۴٪ افزایش یافته است. دیگران نیز مزیت‌های مشابهی را گزارش کرده‌اند. مرورهای فنی (برای طراحی و سایر فعالیت‌های فنی) منافعی فراهم می‌آورند که قابل مشاهده است و واقعاً باعث صرفه‌جویی در وقت می‌شوند. ولی برای بسیاری از آن‌ها که در کار نرم‌افزار هستند، این جمله خلاف منطق به نظر می‌رسد. آن‌ها استدلال می‌کنند که «مرورها وقت گیرند و ما وقت اضافی برای این کارها نداریم!» آن‌ها می‌گویند زمان در هر پروژه نرم‌افزاری، دارایی گران بهایی است و توانایی مرور هر محصول کاری آن هم به‌طور مشروح و مفصل، زمان بسیاری زیادی طلب می‌کند.

مثال‌های ارائه شده در این بخش اما از چیز دیگری حکایت دارند. مهم‌تر این که داده‌های صنعتی برای مرورهای نرم‌افزار به مدت بیش از دو دهه جمع‌آوری شده‌اند که خلاصه‌ای کیفی از آن‌ها با استفاده از نمودارهای شکل ۴-۱۵ نشان داده شده است.

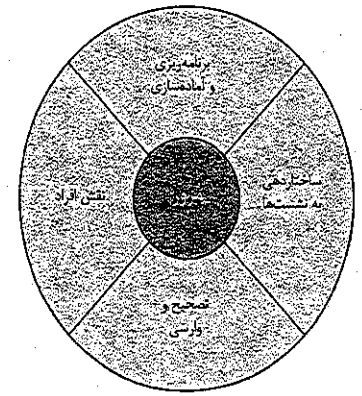


شکل ۴-۱۵ تلاش‌های صرف شده با مرور و بدون مرور.

همان‌طور که در شکل می‌بینید، تلاش صرف شده هنگام به‌کارگیری مرورها از همان ابتدای توسعه‌ی یک نسخه از نرم‌افزار افزایش می‌یابد، ولی این سرمایه‌گذاری زودهنگام برای مرورها دهها برابر سود خواهد داشت، زیرا تلاش‌های مورد نیاز برای آزمون و تصحیح را کاهش می‌دهد. علاوه بر آن، تاریخ استقرار فرایندهایی که شامل مرور می‌شوند، از تاریخ استقرار بدون مرور زودتر خواهد بود. مرورها زمان بر نیستند. بلکه باعث صرفه‌جویی در زمان می‌شوند.

۴-۱۵ مرورها: یک طیف رسمیت

مرورهای فنی را باید با سطحی از رسمیت به‌کار برد که با محصولی که قرار است ساخته شود، خط زمانی پروژه و کسانی که کار را انجام می‌دهند، تناسب داشته باشد. در شکل ۵-۱۵ یک مدل مرجع برای مرورهای فنی ارائه شده است [Lai02] که چهار خصوصیت سهیم در تعیین سطح رسمیت اجرای مرور را مشخص می‌کند.



شکل ۵-۱۵ مدل مرجع برای مرورهای فنی.

هر کدام از این خصوصیت‌های مدل مرجع به تعریف سطح رسمیت مرور کمک می‌کنند. رسمیت مرور هنگامی افزایش می‌یابد که (۱) نقش‌های متمایزی به صراحت برای افراد تیم مرور تعریف می‌شود، (۲) مقدار کافی برنامه ریزی و آماده‌سازی برای مرور وجود داشته باشد، (۳) یک ساختار متمایز برای مرور (از جمله وظایف و محصولات کاری درونی) تعریف شود و (۴) هر گونه تصحیحاتی که قرار است انجام شود، توسط افراد تیم مرور پیگیری شود.

برای درک مدل مرجع، فرض کنید که تصمیم گرفته‌اید طراحی واسط برای SafeHomeAssured.com را مرور کنید. می‌توانید این را به چند شیوه‌ی متفاوت انجام دهید که از میزان نسبتاً اتفاقی تا کاملاً شدید در تغییرند. اگر به این نتیجه رسیدید که رویکرد اتفاقی بیش از بقیه مناسب است، از چند همکار (همتا) خواهش می‌کنید که نمونه‌ی اولیه‌ی واسط را بررسی کنند تا مشکلات بالقوه‌ی آن را کشف کنند. همه‌ی شما به این نتیجه می‌رسید که هیچ آماده‌سازی از قبل لازم نیست، ولی نمونه‌ی اولیه را به شیوه‌ای ساخت‌یافته- نخست با نگاه کردن به چیدمان، سپس زیبایی‌شناسی، بعد از آن گزینه‌های گشت‌وگذار و غیره- ارزیابی کنید. به‌عنوان طراح تصمیم می‌گیرید که چند یادداشت بردارید، ولی هیچ کار رسمی‌ای نکنید.

ولی اگر واسط در موفقیت کل پروژه نقش محوری داشته باشد، چطور، اگر جان انسان‌ها به واسطی وابسته باشد که از نظر ارگونومی مناسب است، چطور؟ ممکن است به این نتیجه برسید که به یک رویکرد شدیدتر نیاز دارید. یک تیم مرور تشکیل می‌شود. هر یک از اعضای تیم باید نقشی به عهده بگیرند- رهبری تیم، ثبتِ یافته‌ها، ارائه مطالب و غیره. به هر کدام از افراد تیم مرور اجازه داده می‌شود به یک محصول کاری (که در این مورد، نمونه اولیه‌ی واسط است) دستیابی داشته باشد و او زمانی را صرف یافتن خطاها، ناسازگاری‌ها و جا افتادگی‌ها می‌کند. بر اساس دستور کاری که قبل از شروع مرور تهیه شده است، مجموعه‌ای از وظایف ویژه انجام خواهد شد. نتایج مرور به‌طور رسمی ثبت می‌شود و تیم بر اساس نتایج مرور، درباره وضعیت محصول کاری تصمیم خواهد گرفت. همچنین ممکن است اعضای تیم، درستی انجام تصحیحات را وارسی کنند.

در این کتاب به دو گروه عمده از مرورهای فنی خواهیم پرداخت: مرورهای **غیررسمی** و **مرورهای فنی رسمی**. در هر یک از این گروه‌های عمده، چند رویکرد متفاوت می‌تواند انتخاب کرد. این دو گروه در بخش‌های آینده ارائه خواهد شد.

۵-۱۵ مرورهای غیر رسمی (Informal Reviews)

مرورهای غیر رسمی شامل بررسی ساده رومیزی (desk check) یک محصول کاری مهندسی نرم‌افزار با یکی از همکاران، یک جلسه اتفاقی (با بیش از دو نفر) به هدف مرور یک محصول کاری یا جنبه‌های مرورگرایی در برنامه‌نویسی جفتی (فصل ۳) می‌شود.

بررسی رومیزی ساده یا *نشست اتفاقی* با یک همکار، یک مرور است. ولی از آن‌جا که هیچ برنامه ریزی یا آماده‌سازی قبلی وجود ندارد، دستور کار یا ساختاری برای نشست تنظیم نمی‌شود و برای خطاهای کشف شده، هیچ پیگیری در کار نیست، اثربخشی این گونه مرورها به‌طور چشمگیری کوچک‌تر از رویکردهای رسمی‌تر است. ولی یک بررسی رومیزی ساده می‌تواند به کشف خطاهایی منجر شود که در غیر این صورت ممکن است در فرایند نرم‌افزار منتشر گردد.

یک راه برای بهبود بخشیدن به بازدهی مرورهای بررسی رومیزی، تهیه‌ی مجموعه‌ای از چک‌لیست‌های مرور ساده برای هر محصول کاری است که توسط تیم نرم‌افزاری ایجاد می‌شود. پرسش‌های مطرح شده در هر چک‌لیست، کلی هستند، ولی افراد تیم مرور را در بررسی محصول کاری راهنمایی می‌کنند. برای مثال، بررسی رومیزی نمونه‌ی اولیه واسط **SafeHomeAssured.com** را دوباره مورد توجه قرار می‌دهیم. طراح و یکی از همکاران او به‌جای این‌که صرفاً با نمونه اولیه در ایستگاه کاری طراح بازی کنند، نمونه‌ی اولیه را با استفاده از یک چک‌لیست برای واسط‌ها بررسی می‌کنند:

- آیا چیدمان با به‌کارگیری قراردادهای استاندارد طراحی شده است؟ چپ به راست؟ بالا به پایین؟
- آیا نیازی به حرکت در پنجره برای ارائه‌ی مطالب هست؟
- آیا استفاده از رنگ و محل قرار گرفتن، نوع فونت و اندازه اثربخش است؟
- آیا قابلیت‌ها و گزینه‌های گشت‌وگذار در یک سطح انتزاع ارائه شده‌اند؟
- آیا همه‌ی انتخاب‌های گشت‌وگذار به وضوح نشانه‌گذاری شده‌اند؟

و غیره. هرگونه خطا یا مشکل ذکر شده توسط افراد تیم مرور، توسط طراح ثبت می‌شود تا بعداً بر طرف گردد. بررسی‌های رومیزی را می‌توان به شیوه‌ای منظم زمان‌بندی کرد یا به‌عنوان بخشی از کار مهندسی نرم‌افزار خوب، اجباری کرد. به‌طور کلی، مقدار مطالبی که باید مرور شود، نسبتاً کوچک بوده زمان کل صرف شده در بررسی رومیزی از دو ساعت فراتر نمی‌رود.

در فصل ۳، برنامه‌نویسی جفتی به این صورت توصیف شد: «XP توصیه می‌کند که دو برنامه‌نویس روی یک ایستگاه کاری با هم کار کنند تا کد مربوط به یک داستان را ایجاد کنند. به این ترتیب، سازوکاری برای حل مسأله به‌صورت زمان حقیقی و تضمین کیفیت زمان حقیقی فراهم می‌آید (دو فکر بهتر از یک فکر هستند).»

برنامه‌نویسی جفتی را می‌توان یک بررسی رومیزی پیوسته دانست. برنامه‌نویسی جفتی به‌جای زمان‌بندی یک مرور در نقاط زمانی مشخص، مرور پیوسته به موازات ایجاد محصول کاری (طراحی یا کدها) را ترغیب می‌کند. مزیت آن، کشف بلافاصله‌ی خطاها و در نتیجه، کیفیت بهتر محصول کاری است.

ویلیام و کسلر [Wil00] درباره بازدهی برنامه‌نویسی جفتی چنین می‌نویسند:

شواهد آماری اولیه و حرف و حدیث‌ها حکایت از آن دارند که برنامه‌نویسی جفتی، تکنیکی پرقدرت برای تولید محصولات نرم‌افزاری با کیفیت و با بهره وری بالا به شمار می‌رود. زوج برنامه‌نویس، با هم کار می‌کنند و در ایده‌ها شریک می‌شوند تا از عهده پیچیدگی‌های توسعه نرم‌افزار برآیند. آن‌ها پیوسته ساخته‌ی دست یکدیگر را در زود هنگام‌ترین و اثربخش‌ترین شکل ممکن، وارسی می‌کنند تا نقایص را بر طرف سازند. به علاوه، هر یک باعث می‌شود که دیگری به کاری که در دست دارد، توجه داشته باشد.

برخی مهندسان نرم‌افزار چنین استدلال می‌کنند که زوائد ذاتی ناشی از برنامه‌نویسی جفتی باعث هدر رفتن منابع می‌شود. اصلاً چرا باید به دو نفر کاری را محول کرد که یک نفر هم می‌تواند آن را انجام دهد؟ پاسخ این پرسش را می‌توان در بخش ۲-۳-۱۵ یافت. اگر کیفیت محصول کاری تولیدشده به‌عنوان نتیجه‌ای از برنامه‌نویسی جفتی به‌طور چشمگیری بهتر از کار یک فرد تنها باشد، عایدی‌های مرتبط با کیفیت می‌توانند چیزی بیش از «زوائد» ناشی از برنامه‌نویسی جفتی را توجیه کنند.

اطلاعات

چک‌لیست‌های مرور

حتی هنگامی که مرورها به خوبی سازمان‌دهی و به درستی اجرا شده باشند، بد نیست یک برگه‌ی یادداشت در اختیار افراد تیم مرور قرار داده شود. یعنی، چک‌لیستی به هر نفر داده شود که حاوی پرسش‌هایی است که باید درباره محصول کاریِ مورد مرور، پرسیده شود.

یکی از جامع‌ترین مجموعه‌های چک‌لیست مرور توسط NASA در مرکز پروازهای فضایی گدارد تهیه شده است که از آدرس زیر قابل دستیابی است:

http://sw-assurance.gsfs.nasa.gov/disciplines/quality/

چک‌لیست‌های مرور فنی مفید دیگری نیز در وب‌سایت‌های زیر پیشنهاد شده‌اند:

Process Impact (**www.processimpact.com/pr_goodies.html**)

SoftwareDioxide (**www.softwaredioxide.com/Channels/ConView.asp?id=6309**)

Macadamian (**www.macadamian.com**)

The Open Group Architecture Review Checklist

(**www.theopengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm**)

DFAS (**www.dfas.mil/technolgy/pal/ssps/docstds/spm036.doc**)

۱۵-۶ مرورهای فنی رسمی (Formal Technical Reviews)

مرور فنی رسمی (FTR) یکی از فعالیت‌های SQA است که مهندسان نرم‌افزار (و دیگران) انجام می‌دهند. اهداف FTR عبارتند از: (۱) کشف خطاها در عملکرد، منطق یا پیاده‌سازی هر نمایشی از نرم‌افزار؛ (۲) تصدیق اینکه نرم‌افزارِ مورد مرور، خواسته‌های خود را برآورده می‌سازد؛ (۳) حصول اطمینان از اینکه نرم‌افزار طبق استانداردهای از پیش تعیین شده ارائه شده است؛ (۴) رسیدن به نرم‌افزاری که به شیوه‌ای یکنواخت توسعه یافته است و (۵) قابل اداره کردن پروژه‌ها. به‌علاوه، FTR به‌عنوان یک پایه آموزشی عمل کرده مهندسان رده پایین را قادر به مشاهده روش‌های متفاوت برای تحلیل، طراحی و پیاده‌سازی نرم‌افزار می‌سازد. FTR همچنین به ارتقای پیوستگی و پشتیبانی کمک می‌کند، زیرا چند نفر با بخش‌هایی از نرم‌افزار آشنا می‌شوند که ممکن است در غیر این صورت امکان دیدن آنها برایشان فراهم نشود.

> برای انسان هیچ چیز ضروری‌تر از آن نیست که کار دیگران را ویرایش کند.
>
> مارک تواین

FTR درواقع طبقه‌ای از مرورهاست که شامل بررسی مقدماتی، بازرسی‌ها، مرورهای نوبت چرخشی و ارزیابی فنی دیگر نرم‌افزاری است. هر FTR به صورت یک ملاقات به اجرا درمی‌آید و فقط در صورتی موفق خواهد بود که به‌طور مناسب برنامه‌ریزی، کنترل و توجه شود. در بخش‌هایی که به دنبال خواهد آمد، دستورالعمل‌هایی مشابه دستورالعمل‌های مربوط به یک بررسی مقدماتی به‌عنوان یک مرور فنی رسمیِ نمونه ارائه خواهد شد. درصورت علاقه به بازرسی‌های نرم‌افزار، و نیز برای کسب اطلاعات بیشتر درخصوص مرورهای فنی رسمی، [Fre90]، [Wie02] یا [Rad02] را ببینید.

۱-۶-۱۵ نشست مرور (Review Meeting)

هر قالب FTR که انتخاب شود، در کلیه نشست‌های مرور باید شرایط حدی زیر را رعایت کرد:

- بین سه تا پنج نفر (معمولاً) باید در نشست حضور یابند؛
- آمادگی قبلی لازم است، ولی نباید بیش از دو ساعت از وقت هر نفر را بگیرد؛
- مدت زمان جلسه باید کمتر از دو ساعت باشد.

با توجه به شرایط حدی فوق، پیداست که FTR بر بخش خاص (و کوچکی) از کل نرم‌افزار تأکید دارد. برای مثال، به‌جای کوشش در مرور کل طراحی، برای هر مؤلفه یا گروه کوچکی از مؤلفه‌ها یک سری بررسی‌های مقدماتی اجرا می‌شود. FTR با تمرکز بیشتر، احتمال کشف خطاها را افزایش می‌دهد.

FTR محصول کاری را کانون توجه قرار می‌دهد. (مثلاً بخشی از مشخصه خواسته‌ها، طراحی مشروحی از مؤلفه‌ها، فهرست کد منبع مربوط به مؤلفه‌ها). فردی که محصول کاری را ایجاد کرده است - *تولیدکننده* - به اطلاع رهبر پروژه می‌رساند که محصول کاری کامل است و نیاز به مرور وجود دارد. رهبر پروژه با یک رهبر *مرور* تماس برقرار می‌کند تا او محصول را از لحاظ آمادگی ارزیابی کند، چند کپی از مواد محصول تهیه کند و آنها را برای آمادگی قبلی در اختیار دو یا سه مسؤول مرور قرار دهد. انتظار می‌رود هر یک از این افراد برای مرور محصول بین یک تا دو ساعت وقت صرف کنند، یادداشت بردارند یا به هر طریق دیگری که ممکن باشد، با کار آشنا شوند. در همان زمان، رهبر مرور نیز محصول را مرور کرده دستورکاری برای نشست تنظیم می‌کند که معمولاً برای روز بعد در نظر گرفته شود.

کسانی که در نشست شرکت می‌کنند عبارتند از رهبر مرور، همه‌ی مسؤولان مرور و تولیدکننده. یکی از مسؤولان مرور، وظیفه ثبت موارد مهم را بر عهده می‌گیرد. FTR با ذکر دستور کار و معرفی مختصر تولید کننده آغاز می‌شود. سپس تولید کننده به تفصیل محصول کاری را توضیح می‌دهد و مسؤولان مرور نیز مسائلی را عنوان می‌کنند که از قبل آماده کرده‌اند. هنگامی که مشکلات و خطاهای معتبر کشف شد، مسؤول ثبت آنها را ثبت می‌کند.

در پایان، همه‌ی حاضران FTR باید تصمیم بگیرند که آیا (۱) محصول کاری را بدون هرگونه اصلاح اضافی بپذیرند، (۲) محصول را به خاطر خطاهای جدی رد کنند (پس از تصحیح به یک مرور دیگر نیاز است)، یا (۳) محصول را بپذیرند مشروط بر آنکه خطاهای جزئی آن تصحیح شود (ولی دیگر نیازی به مرور نخواهد بود). پس از اتخاذ تصمیم، حضار FTR، برگه‌ای را امضا می‌کنند تا حضور خود را در نشست مرور خاطرنشان سازند.

مرجع وب

کتاب راهنمای وارسی‌های فنی NASA - SATC را می‌توان از وب‌سایت زیر دانلود کرد:

Safc.gsfc.nasa.gov/Documents/fi/gdb/fi.pdf

نکته‌ی کلیدی

FTR بخش نسبتاً کوچکی از یک محصول کاری را کانون توجه قرار می‌دهد.

اندرز

در برخی شرایط، فکر خوبی است که کسی غیر از سازنده‌ی محصول، آن را مرور کند. این به تغییر لفظی محصول کاری و شناخت بهتر خطاها خواهد انجامید.

۲-۶-۱۵ گزارش مرور و ثبت امور

در اثنای FTR، مسؤول ثبت، فعالانه همه‌ی مسائلی را که مطرح شده است، ثبت می‌کند. این مسائل در انتهای نشست مرور خلاصه می‌شوند و فهرستی از مسائل تهیه می‌شود. به‌علاوه، یک گزارش خلاصه از مرور فنی رسمی کامل می‌شود. گزارش خلاصه مرور، سه پرسش زیر را پاسخ می‌دهد:

۱. چه چیزی مرور شده است؟

۲. چه کسانی آن را مرور کرده‌اند؟

۳. یافته‌ها و نتایج کدام‌اند؟

گزارش خلاصه مرور، یک فرم تک‌صفحه‌ای (با پیوست‌های احتمالی) است. این گزارش در سوابق پروژه ثبت می‌شود و ممکن است بین رهبر پروژه و علاقمندان دیگر توزیع شود.

فهرست مسائل مرور به دو منظور استفاده می‌شود: (۱) شناسایی زمینه‌های مشکلات در محصول و (۲) به‌عنوان چک‌لیستی عمل می‌کند که تولید کننده را در راه انجام تصحیحات کمک می‌کند. فهرست مسائل، معمولاً به گزارش خلاصه‌ی مرور الصاق می‌شود.

باید رویه‌ای برای پیگیری ایجاد شود تا اطمینان حاصل گردد که موارد موجود در فهرست مسائل، به‌طور مناسب تصحیح شده‌اند. اگر این کار انجام نشود، احتمال می‌رود که مسائل مطرح شده نادیده گرفته شوند. یک روش، اعطای مسؤولیت پیگیری به رهبر مرور است.

۳-۶-۱۵ دستورالعمل‌های مرور

دستورالعمل‌های مربوط به اجرای مرورهای فنی رسمی را باید از پیش وضع نمود و در میان تمامی مسؤولان بازرسی توزیع نمود، آنها را به تصویب رساند و سپس رعایت کرد. مرور کنترل‌نشده غالباً بدتر از آن است که اصلاً مروری صورت نپذیرد. آنچه به دنبال خواهد آمد، مجموعه‌ای از حداقل تعداد دستورالعمل‌ها برای مرورهای فنی رسمی است.

۱. *مرور محصول، نه تولید کننده محصول.* هر FTR شامل مجموعه‌ای از افراد و برداشت‌های آنها است. اگر FTR درست اجرا شود، باید پس از خاتمه، احساسی گرم از موفقیت را بر جای بگذارد. اگر FTR درست اجرا نشود، می‌تواند حال و هوای جلسه‌ی تفتیش عقاید را به خود بگیرد. خطاها را به ملایمت باید متذکر شد؛ لحن مذاکرات باید دوستانه و سازنده باشد؛ نباید هدف شرمنده کردن طرف مقابل باشد. رهبر مرور باید نشست مرور را به نحوی اداره کند که لحن مناسب حفظ شود و اگر کنترل اوضاع از دست وی خارج شد، فوراً ختم جلسه کند.

۲. *تهیه یک دستور کار و رعایت آن.* یکی از مشکلات همه‌ی نشست‌ها، انحراف است. FTR باید طبق زمان‌بندی انجام شود. رهبر مرور، مسؤول تعیین زمان نشست بوده نباید به هنگام انحراف از موضوع، از برحذر داشتن افراد واهمه داشته باشد.

۳. *محدود کردن بحث و مشاجره.* هنگامی که یکی از مسؤولان مرور، مسأله‌ای را مطرح می‌کند، ممکن است همه با آن موافق نباشند. به جای صرف وقت برای مشاجره درباره آن، باید مسئله را برای بحث بیشتر ثبت نمود.

۴. *بیان واضح بخش‌های مشکل‌دار و نه کوشش برای حل همه‌ی مشکلات ذکرشده.* مرور، یک جلسه حل مشکل نیست. حل مشکل غالباً به دست خود تولید کننده یا به کمک یک نفر دیگر صورت می‌پذیرد. حل مشکل را باید به بعد از نشست مرور موکول کرد.

۵. *یادداشت برداری.* گاهی بد نیست که مسؤول ثبت، نکاتی را روی یک تابلوی دیواری یادداشت کند تا مسؤولان دیگر مرور بتوانند توضیحات و اولویت‌بندی‌ها را ارزیابی کنند.

۶. *محدود کردن تعداد شرکت‌کنندگان و اصرار بر آمادگی قبلی.* دو نفر از یک نفر بهتر است، ولی ۱۴ نفر الزاماً از ۴ نفر بهتر نیست. تعداد افراد دخیل در نشست را در حداقل نگهدارید. ولی همه‌ی اعضای تیم مرور باید آمادگی قبلی داشته باشند. رهبر مرور باید توضیحات کتبی را درخواست کند (این نشان می‌دهد که مسؤولان مرور مطالب را قبلاً مرور کرده‌اند).

اندرز

با خشونت به خطاها اشاره نکنید. یک راه برای ملاطفت، پرسیدن سؤالی است که تولیدکننده‌ی خطا را قادر به کشف آن کند.

«نشست، غالباً رویدادی است که در آن، دقایق گرفته می‌شوند و ساعت‌ها تلف می‌شوند.»

ناشناس

۷. *تهیه چک‌لیستی برای هر محصولی که احتمال مرور آن می‌رود.* این چک‌لیست به رهبر مرور کمک می‌کند تا نشست FTR را سازماندهی کند و به هر یک از مسؤولان مرور کمک می‌کند تا بر مسائل مهم تأکید کنند. برای تحلیل، طراحی، کدنویسی و حتی مستندات آزمون نیز باید فهرست‌های کنترلی تهیه کرد.

۸. *تخصیص منابع و زمان‌بندی برای FTRها.* برای آنکه مرورها مؤثر واقع شوند، باید آنها را به‌عنوان یکی از وظایف در اثنای فرایند مهندسی نرم‌افزار زمان‌بندی کرد. به‌علاوه، زمان را باید برای اطلاعات اجتناب‌ناپذیر برنامه‌ریزی کرد که به‌عنوان نتیجه‌ای از FTR رخ می‌دهد.

۹. *اجرای آموزش معنی‌دار برای همه‌ی مسؤولان مرور.* همه‌ی مسؤولان مرور، برای بالا رفتن بازدهی باید آموزش رسمی ببینند. در این آموزش باید هم بر مسائل مرتبط با فرایند و هم بر بُعد روان‌شناسی انسانی مرورها تأکید شود. فریدمن و واینبرگ [Fre90] برای هر ۲۰ نفری که قرار است به‌طور مؤثر در مرورها شرکت کنند، یک دوره آموزش یک ماهه برآورد می‌کنند.

۱۰. *مرور مرورهای قبلی.* مرور خود فرایند مرور در کشف مشکلات مفید واقع می‌شود. نخستین محصولی که باید مرور شود، ممکن است خود دستورالعمل‌ها باشد.

از آنجا که متغیرهای فراوانی (مثل تعداد شرکت کنندگان، نوع محصولات کاری، زمان‌بندی و طول مدت، روش مرور مشخص) بر مرور تأثیر می‌گذارند، سازمان نرم‌افزاری باید براساس تجربه تعیین کند که در یک شرایط خاص، چه روشی مؤثر واقع می‌شود.

> «این یکی از زیباترین تعادل‌های حیات است که هیچ کس نمی‌تواند خالصانه در کمک به دیگری بکوشد بدون این که به خود کمک کند.»
>
> رالف والدو امرسون

۴-۶-۱۵ مرورهای نمونه‌محور

در شرایط ایده‌آل، هر محصول کاری مهندسی نرم‌افزار دستخوش یک مرور فنی رسمی می‌شود. در پروژه‌های نرم‌افزار به معنای واقعی کلمه، منابع، محدود و زمان، کوتاه است. در نتیجه، از مرورها غالباً چشم‌پوشی می‌شود هرچند که ارزش آن‌ها به‌عنوان یک سازوکار کنترل کیفی واضح است.

تلین و همکاران [The01] یک فرایند مرور نمونه‌محور را پیشنهاد می‌کنند که در آن، نمونه‌هایی از محصولات کاری مهندسی نرم‌افزار، وارسی می‌شوند تا تعیین شود که کدام محصولات کاری بیش از همه مستعد خطا هستند. سپس منابع FTR کامل فقط روی آن دسته از محصولات کاری متمرکز می‌شوند که احتمال بروز خطا در آن‌ها بیشتر است (بر اساس داده‌های جمع‌آوری شده طی نمونه‌برداری).

در فرایند مرور نمونه محور برای اثربخشی باید تلاش به عمل آید که محصولات کاری هدف اولیه برای FTRهای کامل باشند. برای نیل به این مقصود، مراحل زیر پیشنهاد می‌شود [The01]:

۱. کسر a_i را برای هر کدام از محصولات کاری I وارسی کنید. تعداد خطاهای f_i یافته شده در a_i را ثبت کنید.

۲. تعداد خطاهای موجود در محصول کاری I را با ضرب کردن f_i در $1/a_i$ برآورد کنید.

۳. محصولات کاری را به ترتیب نزولی بر حسب برآورد حدودی تعداد خطاهای موجود در هر کدام از آن‌ها مرتب کنید.

۴. منابع در دسترس برای مرور را روی آن دسته از محصولات کاری متمرکز کنید که دارای بالاترین تعداد خطاهای برآورد شده‌اند.

اندرز

مرورها زمان می‌برند، ولی این زمانی است که خوب صرف می‌شود. ولی اگر زمان کوتاه است و گزینه‌ی دیگری پیش روی شما نیست، مرورها را حذف نکنید بلکه از مرورهای نمونه‌محور استفاده کنید.

SafeHome

مسائل کیفیتی

صحنه: دفتر داگ میلر در شروع پروژه نرم‌افزار *SafeHome*

نقش‌آفرینان: داگ میلر (مدیر تیم مهندسی نرم‌افزار *SafeHome*) و سایر اعضای تیم نرم‌افزاری.

گفتگو:

داگ: می‌دانم زمان زیادی صرف تهیه یک برنامه کیفیتی برای این پروژه کرده‌ایم، ولی مدت‌هاست در فکر آن بوده‌ایم و باید کیفیت را هم در نظر داشته باشیم... درست است؟

جیمی: حتماً ما قبلاً تصمیم گرفته‌ایم که به موازات توسعه‌ی مدل خواسته‌ها (فصل‌های ۶ و ۷)، یک رویه آزمون برای هر خواسته تهیه کند.

داگ: این خیلی خوب است، ولی باید تا زمان آزمون کیفیت صبر کنیم؟ نه؟

وینود: نه. البته که نه. ما یک سری مرور برای پروژه مربوط به این گام نرم‌افزار ترتیب داده‌ایم و کنترل کیفیت را با این مرورها شروع می‌کنیم.

جیمی: من یک‌کم نگران هستم که وقت کافی برای انجام همه مرورها نداشته باشیم. در واقع حتم دارم.

داگ: پس چه پیشنهادی داری؟

جیمی: من می‌گویم عناصری از مدل خواسته‌ها و طراحی را انتخاب کنیم که بیشترین اهمیت را در *SafeHome* و مرور آن‌ها دارند.

وینود: ولی اگر در بخشی از مدل که مرور نمی‌شود، چیزی را از دست دهیم چه خواهد شد؟

شکیرا: من یک چیزهایی درباره تکنیک نمونه‌برداری [بخش ۴-۶-۱۵] خوانده‌ام که ممکن است ما را در هدف گرفتن کاندیداهایی برای مرور کمک کنند [او این روش را توضیح می‌دهد.]

جیمی: شاید... ولی من حتی مطمئن نیستم که وقت داشته باشیم از هر کدام از عناصر نمونه‌برداری کنیم.

وینود: از ما می‌خواهی چه کار کنیم داگ؟

داگ: بیایید یک چیزی از برنامه‌نویسی حذی [فصل ۳] به عاریه بگیریم. ما عناصر هر مدل را به‌صورت جفتی-دو نفری- تهیه می‌کنیم و همین طور که جلو می‌رویم، هر کدام را مورد مرور غیر رسمی قرار می‌دهیم. بعداً عناصر «حیاتی» را برای مرور تیمی رسمی‌تر هدف می‌گیریم، ولی آن مرورها را در حداقل سطح ممکن نگه می‌داریم. به این صورت، هر چیزی از نظر بیشتر از یک نفر خواهد گذشت، ولی تاریخ‌های تحویل را هم حفظ می‌کنیم.

جیمی: یعنی باید برنامه زمان‌بندی را تغییر دهیم.

داگ: عیبی ندارد. در این پروژه، کیفیت از زمان‌بندی مهم‌تر است.

کسری از محصولات کاری که نمونه‌برداری شده‌اند باید نماینده‌ی کل محصولات کاری باشد و به‌قدر کافی بزرگ باشد تا برای افراد تیم مرور که از آن نمونه‌برداری می‌کنند، معنا داشته باشند. با افزایش a_i، احتمال این که نمونه، نماینده‌ی معتبری از محصول کاری باشد نیز افزایش می‌یابد. تیم مهندسی نرم‌افزار باید بهترین مقدار a_i را برای انواع محصولات کاری تولید شده تعیین کند.[1]

[1] تلین و همکارن شبیه‌سازی مشروحی اجرا کرده‌اند که می‌تواند به این تعیین کمک کند. برای جزئیات بیشتر [The01] را ببینید.

۱۵-۷ خلاصه

هدف و نیت از هر مرور فنی، یافتن خطاها و کشف مسائلی است که بر نرم‌افزار استقراریافته تأثیر منفی می‌گذارند. خطا هر چه زودتر کشف و تصحیح شود، احتمال انتشار آن به سایر محصولات کاری مهندسی نرم‌افزار و تشدید شدن آن کمتر می‌شود که نتیجه‌اش، تلاش بسیار کم‌تر برای تصحیح آن خواهد بود.

برای تعیین این که آیا فعالیت‌های کنترل کیفیت، نتیجه بخش هستند، مجموعه‌ای از معیارها را باید جمع‌آوری کرد. معیارهای مرور، تلاش‌های لازم جهت اجرای مرور و انواع و شدت خطاهای کشف‌شده طی مرور را کانون توجه قرار می‌دهند. هنگامی که داده‌های مربوط به معیارها جمع‌آوری شدند، از آن‌ها می‌توان برای ارزیابی بازدهی مرورهای انجام شده استفاده کرد. داده‌های صنعتی نشان می‌دهند که مرورها سود چشمگیری در سرمایه‌گذاری بر می‌گردانند.

یک مدل مرجع برای سطح رسمیت مرور، نقش افراد به‌صورت برنامه‌ریزی و آماده‌سازی، ساختاردهی به جلسات، رویکرد تصحیح و وارسی تعریف می‌شود که این‌ها خصوصیاتی هستند که بر اساس آن‌ها درجه رسمیت مرور تعیین می‌شود. مرورهای غیر رسمی، ماهیتی اتفاقی دارند، ولی هنوز هم می‌توان به‌طور مؤثر در کشف خطاها از آن‌ها بهره برد. مرورهای رسمی، ساخت‌یافته‌تر بوده احتمال منجر شدن به کیفیت بالای نرم‌افزار را فزونی می‌بخشند.

مشخصه‌ی مرورهای غیر رسمی، حداقل برنامه‌ریزی و آماده‌سازی و حفظ سوابق اندک است. بررسی‌های رومیزی و برنامه‌نویسی جفتی در زمره‌ی مرورهای غیر فنی قرار می‌گیرند.

مرور فنی رسمی، یک جلسه‌ی سازمان یافته است که اثربخشی آن در کشف خطاها ثابت شده است. با بررسی‌های گام به گام و وارسی‌ها، نقش‌هایی معین برای هر کدام از افراد تیم مرور تعیین می‌شود، برنامه ریزی و آماده‌سازی قبلی تشویق می‌شود، به‌کارگیری دستورالعمل‌های مرور تعریف‌شده، الزامی می‌شود و حفظ سوابق و گزارش وضعیت اجباری می‌شود. از مرورهای نمونه محور می‌توان هنگامی بهره برد که اجرای مرورهای فنی رسمی برای تمامی محصولات کاری، امکان‌پذیر نباشد.

مسائل و نکاتی برای تعمق

۱-۱۵ اختلاف میان خطا و نقص را شرح دهید.

۲-۱۵ چرا نمی‌توانیم تا زمان آزمون صبر کنیم تا در همان زمان همه‌ی خطاهای نرم‌افزار را کشف و تصحیح کنیم؟

۳-۱۵ فرض کنید ۱۰ خطا وارد مدل خواسته‌ها شده است و هر خطا با ضریب دو به یک در طراحی تشدید خواهد شد و علاوه بر آن ۲۰ خطای طراحی هم اضافه خواهد شد که سپس با ضریب ۱/۵ به یک وارد کدها می‌شوند و در آن‌جا ۳۰ خطای دیگر نیز اضافه می‌شود. همچنین فرض کنید که در کل آزمون واحدها ۳۰٪ از همه‌ی خطاها کشف شود. در قسمت انسجام بخشی، ۳۰٪ از خطاهای باقیمانده یافته شود و آزمون‌های اعتبارسنجی نیز ۵۰٪ از خطاهای باقیمانده را کشف کنند. هیچ مرور انجام نمی‌شود. چند خطا وارد میدان خواهد شد؟

۴-۱۵ وضعیت توصیف شده در مسأله ۳-۱۵ را دوباره در نظر بگیرید. ولی اکنون فرض کنید مرورهای خواسته‌ها، طراحی و کدها اجرا می‌شوند و در کشف همه‌ی خطاها در آن مرحله ۶۰٪ مؤثر واقع می‌شوند

چند خطا وارد میدان می‌شوند.

۵-۱۵ وضعیت توصیف شده در مسأله‌های ۳-۱۵ و ۴-۱۵ را دوباره در نظر بگیرید. اگر یافتن و تصحیح هر کدام از خطاهای وارد میدان شده ۴۸۰۰ دلار و برای مرحله مرور، ۲۴۰ دلار هزینه بردارد، با اجرای مرور چه مقدار پول صرفه‌جویی خواهد شد؟

۶-۱۵ معنی شکل ۴-۱۵ را به زبان ساده بیان کنید.

۷-۱۵ فکر می‌کنید کدام خصوصیات مدل مرجع بیشترین تأثیر را بر رسمیت مرور دارد؟ توضیح چرا؟

۸-۱۵ آیا می‌توانید چند نمونه مثال بزنید که در آن‌ها یک بررسی رومیزی ممکن است به جای نفع رساند باعث ایجاد مشکل شود؟

۹-۱۵ مرور فنی رسمی تنها در صورتی اثربخش خواهد بود که همگان از قبل آماده شده باشند. چگونه تشخیص می‌دهید که یکی از مشارکت کنندگان در مرور، آماده نشده است؟ اگر رهبر تیم مرور باشید، چه می‌کنید؟

۱۰-۱۵ با در نظر گرفتن همه‌ی دستورالعمل‌های ارائه شده در بخش ۳-۶-۱۵، فکر می‌کنید کدام مهم‌تر است و چرا؟

فصل ۱۶

تضمین کیفیت نرم‌افزار

نگاهی گذرا

تضمین کیفیت نرم‌افزار چیست؟ فقط گفتن این جمله که کیفیت نرم‌افزار اهمیت دارد، کافی نیست. بلکه، باید (۱) به‌طور واضح تعیین کنید منظور از «کیفیت نرم‌افزار» چیست، (۲) مجموعه‌ای از فعالیت‌ها را ایجاد کنید که به کمک آنها بتوان اطمینان حاصل کرد هر محصول کاری مهندسی نرم‌افزار کیفیت بالایی از خود نشان می‌دهد، (۳) فعالیت‌های تضمین کیفیت را روی همه‌ی پروژه‌های نرم‌افزاری اجرا کنید، (۴) برای توسعه راهبردهایی جهت بهبود بخشیدن به فرایند نرم‌افزار، از معیارها استفاده کنید و درنتیجه، کیفیت محصول نهایی را بهبود بخشید.

چه کسی آن را انجام می‌دهد؟ هر کسی که در فرایند مهندسی نرم‌افزار شرکت داشته باشد.

چرا اهمیت دارد؟ می‌توانید آن را درست انجام دهید یا آن را دوباره انجام دهید. اگر یک تیم نرم‌افزاری در همه‌ی فعالیت‌های مهندسی نرم‌افزار بر کیفیت تأکید کند، از مقدار دوباره‌کاری‌ها خواهد کاست. این منجر به کاهش هزینه‌ها و مهمتر از آن تسریع در زمان تحویل به بازار می‌شود.

مراحل کار کدام است؟ پیش از آنکه بتوان فعالیت‌های تضمین کیفیت را آغاز کرد، تعریف «کیفیت نرم‌افزار» در چند سطح متفاوت از انتزاع اهمیت دارد. هنگامی که دانستید کیفیت چیست، تیم نرم‌افزار باید یک مجموعه فعالیت SQA را شناسایی کند که خطاها را پیش از تحویل محصولات کاری، از آنها جدا کند.

محصول کار چیست؟ یک برنامه‌ریزی تضمین کیفیت برای تعیین راهبرد SQA مربوط به تیم نرم‌افزاری ایجاد می‌شود. طی تحلیل، طراحی، و تولید کد، محصول کاری اولیه SQA، یک گزارش خلاصه از بازبینی فنی رسمی است. در اثنای آزمون، روال‌ها و طرح‌های آزمون تولید می‌شوند. محصولات کاری دیگر مرتبط با بهسازی فرایند نیز ممکن است تولید شوند.

چگونه می‌توانم اطمینان حاصل کنم که درست از انجام کار برآمده‌ام؟ خطاها را پیش از آنکه به نقص تبدیل شوند، بیابید! یعنی، بکوشید تا بازدهی رفع نقص را بهبود بخشید (فصل‌های ۴ تا ۷) تا مقدار دوباره کاری کاهش یابد.

روش مهندسی نرم‌افزار توصیف شده در این کتاب، یک هدف واحد را دنبال می‌کند: تولید نرم‌افزاری با کیفیت بالا. ولی بسیاری از خوانندگان درگیر این سؤال هستند: «کیفیت نرم‌افزار چیست؟» فیلیپ کرازبی [Cro79] در کتاب خود درباره کیفیت، پاسخ این پرسش را می‌دهد:

> مشکل مدیریت کیفیت چیزی نیست که مردم درباره آن نمی‌دانند. مشکل چیزی است که فکر می‌کنند می‌دانند ...
>
> در این خصوص، کیفیت مثل مسائل جنسی است. همه با آن موافق هستند. (البته تحت شرایط معین). همه احساس می‌کنند که آن را درک می‌کنند (هرچند نمی‌خواهند درباره آن توضیح دهند). همه فکر می‌کنند جزئی از تمایلات طبیعی است (و به خوبی با آن کنار می‌آیند). و البته، اکثر مردم احساس می‌کنند، مشکلاتی که در این زمینه پیش می‌آید، به علت وجود افراد دیگر است. (اگر فقط وقت کافی داشته باشند که کارها را به خوبی انجام دهند).

کیفیت در واقع مفهومی چالش‌برانگیز است- مفهومی که به تفصیل در فصل ۱۴ به آن پرداخته شد.[1] برخی نرم‌افزارنویسان همچنان بر این باورند که کیفیت نرم‌افزار چیزی است که پس از تولید کُد باید نگران آن بود. هیچ چیز نمی‌تواند از حقیقت فراتر رود! تضمین کیفیت نرم‌افزار یک فعالیت چتری است (فصل ۲) که در سرتاسر فرایند نرم‌افزار اجرا می‌شود.

تضمین کیفیت نرم‌افزار (SQA) شامل موارد زیر می‌شود: (۱) یک فرایند SQA، (۲) وظایف خاص تضمین کیفیت و کنترل کیفیت (شامل مرورهای فنی و راهبرد آزمون چندلایه)، (۳) کار مهندسی نرم‌افزار اثربخش (روش‌ها و ابزارها)، (۴) کنترل همه‌ی محصولات کاری نرم‌افزاری و تغییرات اعمال شده در آن‌ها (فصل ۲۲) (۵) رویه‌ای برای حصول اطمینان از مطابقت با استانداردهای توسعه نرم‌افزار (در صورت امکان) و (۶) سازوکارهای اندازه‌گیری و گزارش‌دهی.

در این فصل به نکات مدیریتی و فعالیت های خاص فرایند خواهیم پرداخت که سازمان نرم‌افزاری را قادر می‌سازند تا اطمینان حاصل کند که «در زمان درست و به شیوه‌ای درست، کار درست را انجام داده است.».

۱-۱۶ مسائل پس‌زمینه

کنترل کیفیت و تضمین کیفیت، فعالیت‌هایی ضروری برای هر شرکتی هستند که کار آن تولید محصولاتی برای استفاده توسط دیگران است. پیش از قرن بیستم، کنترل کیفیت فقط مسؤولیت صنعتگری بود که محصول را می‌ساخت. با گذشت زمان، و رواج فنون تولید انبوه، کنترل کیفیت به فعالیتی تبدیل شد که افرادی غیر از سازندگان محصول آن را انجام می‌دادند.

نخستین وظیفه‌ی کنترل کیفیت و تضمین کیفیت در بل لَبز (Bell Labs) در سال ۱۹۱۶ معرفی شد و به‌سرعت در سرتاسر جهان تولید شایع شد. طی دهه ۱۹۴۰، رویکردهای رسمی‌تری در قبال کنترل کیفیت پیشنهاد شد. این رویکردها بر اندازه‌گیری و بهبود مستمر فرایند تکیه دارند [Dem86] که عناصر کلیدی مدیریت کیفیت هستند.

[1] اگر فصل ۱۴ را نخوانده‌اید، اکنون آن را بخوانید.

«تعداد اشتباهات شما بیش از حد زیاد بوده است.»

یوگی برا

امروزه، هر شرکتی دارای سازوکارهایی برای حصول اطمینان از کیفیت محصولات خود است. درواقع، طی چند دهه‌ی اخیر، بیانیه‌های واضح یک شرکت در خصوص دغدغه‌های کیفیتی آن به یک مزیت بازاریابی تبدیل شده است.

سابقه تضمین کیفیت در توسعه نرم‌افزار، موازی سابقه کیفیت در تولید سخت‌افزار بود. طی سال‌های اولیه علم کامپیوتر (دهه‌های ۱۹۵۰ و ۱۹۶۰)، کیفیت، مسؤولیت اصلی برنامه‌نویس بود. استانداردهای تضمین کیفیت برای نرم‌افزارها در اثنای توسعه نرم‌افزارهای نظامی، طی دهه ۱۹۷۰ وارد کار شدند و به سرعت در توسعه نرم‌افزارهای مربوط به جهان تجارت راه پیدا کردند [IEE93]. با گسترش دادن تعریفی که در بالا ارائه شد، می‌توان گفت تضمین کیفیت یک الگوی برنامه‌ریزی شده و سیستماتیک از عملیات است [Sch98c] که برای حصول اطمینان از کیفیت نرم‌افزار موردنیاز هستند. دامنه مسؤولیت تضمین کیفیت را شاید به بهترین وجه بتوان در این شعار خودروسازان خلاصه کرد: «کیفیت، حرف اول را می‌زند». در مورد نرم‌افزارها، مسؤولیت تضمین کیفیت بر عهده افراد متفاوتی است - مهندسان نرم‌افزار، مدیران، مشتریان، فروشندگان و افرادی که در گروه SQA خدمت می‌کنند.

گروه SQA به عنوان نماینده خانگی مشتری عمل می‌کند. یعنی، کسانی که SQA را انجام می‌دهند باید از دیدگاه مشتری به نرم‌افزار بنگرند. آیا نرم‌افزار به قدر کافی عوامل کیفیتی ذکر شده در فصل ۱۴ را برآورده می‌سازد؟ آیا توسعه نرم‌افزار مطابق با استانداردهای از پیش تعیین شده صورت پذیرفته است؟ آیا قواعد فنی به طور مناسب نقش خود را به عنوان بخشی از فعالیت SQA ایفا کرده‌اند؟ گروه SQA می‌کوشد تا به این سؤال و سؤالاتی از این دست پاسخ دهد تا از حفظ کیفیت نرم‌افزار اطمینان حاصل شود.

۲-۱۶ عناصر تضمین کیفیت نرم‌افزار

تضمین کیفیت نرم‌افزار شامل گستره وسیعی از دغدغه‌ها و فعالیت‌ها می‌شود که مدیریت کیفیت نرم‌افزار را کانون توجه قرار می‌دهند. آن‌ها را می‌توان به شیوه زیر خلاصه کرد [Hor03]:

استانداردها. سازمان‌های IEEE ،ISO و سایر سازمان‌ها، آرایه وسیعی از استانداردهای مهندسی نرم‌افزار و مستندات وابسته به آن را تدوین کرده‌اند. استانداردها را ممکن است سازمان مهندسی نرم‌افزار، داوطلبانه بپذیرد یا مشتری یا یک طرف ذینفع دیگر الزامی کند. وظیفه‌ی SQA (تضمین کیفیت نرم‌افزار) حصول اطمینان از رعایت استانداردهای پذیرفته شده و مطابقت تمامی محصولات کاری با این استانداردهاست.

مرورها و ممیزی‌ها. مرورهای فنی یک نوع فعالیت کنترل کیفیت هستند که توسط مهندسان نرم‌افزار برای مهندسی نرم‌افزار اجرا می‌شوند (فصل ۱۵). هدف از انجام آن‌ها کشف خطاهاست. ممیزی‌ها نوعی از مروری هستند که توسط پرسنل SQA و به قصد حصول اطمینان از رعایت دستورالعمل‌های کیفیتی برای کار مهندسی نرم‌افزار دنبال می‌شوند. برای مثال، ممکن است ممیزی روی فرایند مرور انجام شود تا اطمینان حاصل شود که مرورها به شیوه‌ای اجرا می‌شوند که به بالاترین احتمال کشف خطا بینجامند.

آزمون. آزمون نرم‌افزار (فصل‌های ۱۷ تا ۲۰) یکی از وظایف کنترل کیفیت است که یک هدف اصلی را دنبال می‌کند- یافتن خطاها. وظیفه‌ی SQA، حصول اطمینان از طرح ریزی درست و

مرجع وب

بخشی عمقی از SQA که شامل انواع تعریف‌ها نیز می‌شود از وب‌سایت زیر قابل حصول است:

www.swqual.com/newsletter/vol2/no1/vol2no1.html

مناسب آزمون‌ها و اجرای اثربخش آن‌هاست به‌طوری که احتمال دستیابی به هدف اصلی آن (یعنی یافتن خطاها) به حداکثر برسد.

جمع‌آوری و تحلیل خطاها/نقایص. تنها راه بهبود بخشیدن، سنجیدن عملکرد است. SQA داده‌های مربوط به خطاها و نقایص را جمع‌آوری و تحلیل می‌کند تا بهتر معلوم شود که خطاها چگونه وارد می‌شوند و کدام فعالیت‌های مهندسی نرم‌افزار برای حذف آن‌ها از همه مناسب‌ترند.

مدیریت تغییرات. تغییر، یکی از مخرب‌ترین جنبه‌های هر پروژه نرم‌افزار است و اگر خوب مدیریت نشود می‌تواند به سردرگمی منجر شود و سردرگمی همیشه به کیفیت ضعیف می‌انجامد. با SQA می‌توان اطمینان حاصل کرد که اقدامات مناسبی برای مدیریت تغییرات (فصل ۲۲) نهادینه شده است.

آموزش. هر سازمان نرم‌افزاری مایل است کارهای مهندسی نرم‌افزار خود را بهبود بخشد. یک عامل کلیدی سهیم در این بهبود بخشی، آموزش مهندسان نرم‌افزار، مدیران آن‌ها و سایر طرف‌های ذی‌نفع است. سازمان SQA در بهبود فرایند نرم‌افزار جلودار است (فصل ۳۰) و در حمایت از برنامه‌های آموزشی نقش کلیدی دارد.

مدیریت منابع خرید. سه گروه از نرم‌افزارها از منابع خارجی تأمین نرم‌افزار خریداری می‌شوند- *پکیج‌های بسته‌بندی شده* (مثل Microsoft Office)، قطعات *نیمه‌آماده* [Hor03] که ساختار اسکلتی پایه را فراهم می‌آورند و می‌توان آن را مطابق با میل و سلیقه‌ی خریدار تکمیل کرد و نرم‌افزارهای *قراردادی* که از روی مشخصات ارائه شده توسط سازمان مشتری، طراحی و ساخته می‌شوند. وظیفه‌ی سازمان SQA این است که اطمینان حاصل کند با پیشنهاد اقدامات کیفیتی خاص که منبع خرید باید رعایت کند، نرم‌افزار با کیفیت بالا نتیجه می‌شود و الزامات کیفیتی خاصی را در هر قرارداد منعقد شده با منبع خرید بگنجاند.

مدیریت امنیت. با افزایش جرم‌های سایبری و مقررات دولتی در حیطه‌ی حفظ حریم خصوصی، هر سازمان نرم‌افزار باید خط مشی‌ها و سیاست‌هایی را نهادینه سازد که داده‌ها را در تمامی سطوح محافظت کنند، محافظت فایروالی برای برنامه‌های تحت وب فراهم کند و اطمینان حاصل کند که نرم‌افزار از درون دستکاری نشده باشد. با SQA می‌توان مطمئن شد که فرایند و فن‌آوری مناسب در دستیابی به کیفیت نرم‌افزار به‌کاررفته است.

ایمنی. از آن‌جا که نرم‌افزارها تقریباً همیشه یک جزء محوری در سیستم‌های در خدمت انسان (مثل خودروها و هواپیماها) هستند، تأثیر نقایص پنهان می‌تواند مصیبت بار باشد. SQA ممکن است مسؤول ارزیابی تأثیر شکست نرم‌افزار و شروع مراحل لازم برای کاهش ریسک باشد.

مدیریت ریسک. گرچه تحلیل ریسک و کاستن از میزان آن (فصل ۲۸) دغدغه مهندسان نرم‌افزار است، سازمان SQA است که باید اطمینان حاصل کند فعالیت‌های مدیریت ریسک به‌طور مناسب اجرا می‌شوند و طرح‌های مربوط به احتمال بروز ریسک تدوین شده‌اند.

علاوه بر هر کدام از این دغدغه‌ها و فعالیت‌ها، SQA تلاش می‌کند تا اطمینان حاصل کند که فعالیت‌های پشتیبانی نرم‌افزار (نظیر نگهداری، دستورالعمل‌های راهنما، مستندسازی و جزوات) انجام یا ایجاد شده‌اند، در حالی که دغدغه اصلی در انجام یا ایجاد آن‌ها، کیفیت بوده است.

> «عالی بودن، توانایی نامحدود برای بهبود بخشیدن به کیفیت آن چیزی است که می‌خواهید ارائه دهید.»
>
> ریک پتین

اطلاعات

منابع مدیریت کیفیت

ده‌ها منبع مدیریت کیفیت در وب موجود است که شامل جوامع حرفه‌ای، سازمان‌های استاندارد و منابع اطلاعات عمومی می‌شود. سایت‌هایی که در زیر خواهند آمد، می‌توانند نقطه شروع خوبی باشند.

جامعه امریکایی کیفیت (SQA) شاخه‌ی نرم‌افزار
www.asq.org/software

اتحادیه ماشین آلات کامپیوتری
www.acm.org

مرکز داده‌ها و تحلیل نرم‌افزارها
www.dacs.dtic.mil

سازمان جهانی استانداردسازی (ISO)
www.iso.ch

ISO SPICE
www.isospice.com

جایزه ملی کیفیت مالکوم بالدریج
www.quality.nist.gov

مؤسسه مهندسی نرم‌افزار
www.sei.cmu.edu/

مهندسی کیفیت و آزمون نرم‌افزار
www.stickminds.com

منابع شش سیگما
www.isixsigma.com
www.asq.org/sixsigma

سازمان بین المللی TickIT: مباحث گواهی کیفیت
www.tickit.org/international.htm

مدیریت کیفیت فراگیر (TQM)
اطلاعات کلی:
www.gslis.utexas.edu/~rpollock/tqm.html

مقالات:
www.work911.com/tqmarticles.htm

واژگان:
www.quality.org/TQM-MSI/TQM-glossary.html

۱۶-۳ وظایف، اهداف و معیارهای SQA

تضمین کیفیت نرم‌افزار از چند وظیفه‌ی مرتبط با دو گروه متفاوت تشکیل می‌شود (این دو گروه عبارتند از مهندسان نرم‌افزار که کارهای فنی را انجام می‌دهند و یک گروه SQA که مسؤولیت برنامه‌ریزی برای تضمین کیفیت، نظارت، ثبت وقایع، تحلیل و گزارش‌دهی بر عهده آنان است).

مهندسان نرم‌افزار با اِعمال روشهای فنی و موازین منسجم، اجرای بازبینی‌های فنی رسمی و اجرای آزمونهای نرم‌افزاری برنامه‌ریزی شده، کیفیت را کنترل می‌کنند.

۱-۳-۱۶ وظایف SQA

وظیفه گروه SQA، کمک به تیم نرم‌افزاری، جهت دستیابی به یک محصول نهایی با کیفیت بالاست. مؤسسه مهندسی نرم‌افزار، مجموعه‌ای از کنش‌های SQA را توصیه می‌کند که برنامه‌ریزی تضمین کیفیت، نظارت، ثبت وقایع، و گزارش‌دهی را مشخص می‌کند. همین کنش‌ها هستند که توسط یک گروه SQA مستقل اجرا (یا تسهیل) می‌شوند:

تهیه یک طرح SQA برای پروژه. این طرح طی برنامه‌ریزی پروژه توسعه می‌یابد و توسط همه‌ی طرف‌های علاقه‌مند مورد بازبینی قرار می‌گیرد. فعالیت‌های تضمین کیفیت که توسط تیم مهندسی نرم‌افزار و گروه SQA اجرا می‌شوند، از این طرح دستور می‌گیرند. در این طرح موارد زیر مشخص می‌شود: ارزیابی‌هایی که باید انجام شوند، بازرسی‌ها و بازبینی‌هایی که باید اجرا شوند، استانداردهایی که در پروژه لازم‌الاجرا هستند، روال‌هایی برای گزارش و پیگیری خطا، مستنداتی‌که باید توسط گروه SQA تولید شود، مقدار بازخوردی که برای تیم پروژه نرم‌افزاری فراهم می‌آید.

شرکت در توسعه توصیف فرایند نرم‌افزاری پروژه. تیم نرم‌افزاری، فرایندی برای انجام کار انتخاب می‌کند. گروه SQA توصیف فرایند را برای مطابقت با سیاست سازمانی، استانداردهای داخلی، استانداردهای تحمیل شده از خارج سازمان (مثل ISO 9001) و بخش‌های دیگر برنامه پروژه نرم‌افزار، مورد بازبینی قرار می‌دهد.

بازبینی فعالیت‌های مهندسی نرم‌افزار برای وارسی مطابقت با فرایند نرم‌افزاری مشخص. گروه SQA انحرافات از فرایند را شناسایی، مستندسازی و پیگیری کرده انجام تصحیحات را مورد وارسی قرار می‌دهد.

بازرسی محصولات کاری برای وارسی مطابقت با محصولات تعیین‌شده به عنوان بخشی از فرایند نرم‌افزار. گروه SQA محصولات کاری انتخاب شده را بازبینی می‌کند؛ انحرافات را شناسایی، مستندسازی و پیگیری می‌کند؛ انجام تصحیحات را مورد وارسی قرار می‌دهد و به طور ادواری نتایج کار خود را به مدیر پروژه گزارش می‌کند.

حصول اطمینان از مستندسازی انحرافات در کار نرم‌افزار و محصولات کاری، و مقابله با آنها براساس یک رویه مستندسازی شده. ممکن است انحرافاتی در برنامه پروژه، توصیف فرایند، استانداردهای لازم الاجرا یا محصولات کاری به چشم بخورد.

ثبت هرگونه عدم مطابقت و گزارش به مدیریت ارشد. موارد عدم مطابقت آنقدر پیگیری می‌شوند تا برطرف شوند.

علاوه بر این فعالیت‌ها، گروه SQA کنترل و مدیریت تغییر را هماهنگ کرده (فصل ۲۲) به جمع‌آوری و تحلیل معیارهای نرم‌افزاری کمک می‌کند.

۲-۳-۱۶ اهداف، صفات و معیارها

کنش‌های SQA که در بخش قبل شرح داده شدند، برای دستیابی به اهداف عملی زیر اجرا می‌شوند:

کیفیت خواسته‌ها. صحت، کامل بودن و سازگاری مدل خواسته‌ها تأثیری قوی بر کیفیت همه‌ی محصولات کاری بعدی خواهد گذاشت. SQA باید مطمئن شود که تیم نرم‌افزاری به‌طور مناسب مدل خواسته‌ها را مرور کرده است تا به سطوح بالایی از کیفیت دست پیدا کند.

نقش گروه SQA چیست؟

کیفیت طراحی. هر عنصر از مدل طراحی باید توسط تیم نرم‌افزاری ارزیابی شود تا اطمینان حاصل گردد که کیفیت بالایی از خود نشان می‌دهد و خود طراحی با خواسته‌ها مطابقت دارد. SQA به‌دنبال صفاتی از طراحی است که نشان‌گر کیفیت هستند.

کیفیت کدها. کد منبع و محصولات کاری مرتبط با آن (نظیر سایر اطلاعات توصیفی) باید با استانداردهای محلی کدنویسی مطابقت داشته باشند و خصوصیاتی از خود به نمایش بگذارند که قابلیت نگهداری را بهبود بخشند. SQA باید این صفات را که تحلیل منطقی کیفیت کدها را میسر می‌سازند، جدا کند.

اثربخشی کنترل کیفیت. تیم نرم‌افزاری باید منابع محدود را به گونه‌ای به‌کار گیرد که احتمال دستیابی به نتیجه‌ای با کیفیت بالاتری، بسیار زیاد باشد. SQA، تخصیص منابع به مرورها و آزمون‌ها را تحلیل می‌کند تا ارزیابی کند که آیا به بهترین نحو ممکن تخصیص داده شده‌اند یا خیر.

در شکل ۱-۱۶ (بر گرفته از [Hya96]) صفاتی مشخص شده است که شاخص‌های وجود کیفیت برای هر کدام از اهداف بحث شده‌اند. معیارهایی که برای نشان‌دادن قدرت نسبی یک صفت می‌توان به‌کار برد نیز نشان داده شده‌اند.

«کیفیت هرگز تصادفی نیست؛ همواره نتیجه‌ی قصد و نیت قوی، تلاش بی‌غش، جهت گیری هوشمندانه و اجرای ماهرانه است؛ کیفیت نشان‌گر انتخاب عاقلانه از میان راه‌های بی‌شمار پیش رو است.»

ویلیام ا. فاستر

۴-۱۶ رویکردهای رسمی در SQA

در بخش قبل استدلال کردیم که کیفیت نرم‌افزار وظیفه همه‌ی افراد است و از طریق کار مهندسی نرم‌افزار رقابتی و نیز از طریق به کارگیری بازبینی‌های فنی، راهبرد آزمون چندلایه، کنترل بهتر محصولات کاری نرم‌افزاری و تغییرات به‌عمل‌آمده در آنها و سرانجام به‌کارگیری استانداردهای پذیرفته‌شده در مهندسی نرم‌افزار قابل حصول است. به‌علاوه، کیفیت را می‌توان برحسب انواع صفات کیفیتی تعریف کرد و با استفاده از انواع شاخص‌ها و معیارها (به‌طور غیرمستقیم) سنجید.

طی سه دهه‌ی گذشته، بخش کوچک ولی تاثیرگذاری از جامعه مهندسی نرم‌افزار استدلال کرده است که رویکرد رسمی‌تری برای تضمین کیفیت نرم‌افزار مورد نیاز است. می‌توان استدلال کرد که یک برنامه کامپیوتری، شیء‌ای ریاضی است. برای هر زبان برنامه نویسی می‌توان یک معناشناسی و نحو دقیق تعریف کرد و رویکردی دقیق برای خواسته‌های نرم‌افزار (فصل ۲۱)، در دسترس است. اگر مدل (مشخصه‌ی) خواسته‌ها و زبان برنامه‌نویسی را بتوان به شیوه‌ای اکید نمایش داد، اثبات ریاضی درستی و نشان دادن پیروی دقیق برنامه از مشخصات آن نیز باید امکان پذیر باشد.

تلاش‌های به عمل آمده برای تصحیح برنامه‌ها، جدید نیستند. دیکسترا [Dij76a] و لینگر، میلز و ویت [Lin79] و بسیاری دیگر، از اثبات درستی برنامه‌ها پشتیبانی کرده‌اند و آن را به مفاهیم برنامه‌نویسی ساخت‌یافته ربط داده‌اند.

مرجع وب

اطلاعات مفیدی درباره SQA و روش‌های کیفیت رسمی را می‌توانید در وب‌سایت زیر بیابید:

www.gslis.utexas.edu/~rpollock/tqm.html

۵-۱۶ تضمین کیفیت آماری نرم‌افزار

تضمین کیفیت آماری نرم‌افزار رفته‌رفته در سرتاسر صنعت گسترش می‌یابد، تا کیفیت، ویژگی کمّی بیشتری بگیرد. برای نرم‌افزار، تضمین کیفیت آماری شامل مراحل زیر می‌شود:

۱. اطلاعات مربوط به نقایص نرم‌افزار جمع‌آوری و گروه‌بندی می‌شود.

چه مراحلی برای اجرای SQA آماری مورد نیاز است؟

شکل ۱-۱۶ اهداف، صفات و معیارهای کیفیت نرم‌افزار.

	صفت	معیار
کیفیت خواسته‌ها	ابهام	تعداد اصلاح‌گرهای مبهم (مثلاً زیاد، بزرگ و دوستدار انسان)
	کامل‌بودن	تعداد TBA، TBD
	قابلیت درک	تعداد بخش‌ها/زیربخش‌ها
	فرّاریت	تعداد چالش‌ها به‌ازای هر خواسته زمان درخواست تغییر (توسط فعالیت)
	قابلیت ردگیری	تعداد خواسته‌هایی که تا طراحی/کدها قابل ردگیری نیستند
	وضوح مدل	تعداد مدل‌های UML تعداد صفحات توصیفی به ازای هر مستند تعداد خطاهای UML
کیفیت طراحی	انسجام در معماری	وجود مدل معماری
	کامل‌بودن مؤلفه‌ها	تعداد مؤلفه‌هایی که تا مدل طراحی قابل ردگیری هستند پیچیدگی طراحی رویه‌ها
	پیچیدگی واسط	تعداد میانگین انتخاب‌ها برای رسیدن به محتویات یا قابلیت مورد نظر مناسب‌بودن چیدمان
	الگوها	تعداد الگوهای به‌کاررفته
کیفیت کدها	پیچیدگی	پیچیدگی سیکلوماتیک
	قابلیت نگهداری	عوامل طراحی (فصل ۸)
	قابلیت درک	درصد توضیحات درونی قرارداد نام‌گذاری متغیرها
	قابلیت استفاده‌ی مجدد	درصد مؤلفه‌هایی که دوباره استفاده شده‌اند
	مستندسازی	شاخص خوانایی
	تخصیص منابع	درصد نفر-ساعت برای هر فعالیت
	سرعت تکمیل	زمان واقعی پایان پروژه در مقایسه با زمان بودجه‌ای
اثربخشی	اثربخشی بازبینی‌ها	معیارهای بازبینی (فصل ۱۴) را ببینید
	اثربخشی آزمون‌ها	تعداد خطاهای یافته شده و اهمیت آنها تلاش لازم برای تصحیح یک خطا منشاء خطا

۲. کوشش می‌شود ردّ هر نقص تا علت اصلی آن پیگیری شود (مثلاً ناهمخوانی با مشخصه، خطای طراحی، عدول از استانداردها، ارتباط ضعیف با مشتری).

۳. با استفاده از اصل پارتو (ریشه‌ی ۸۰٪ نقایص را می‌توان در ۲۰٪ از همه‌ی علل ممکن یافت)، آن ۲۰٪ علل جدا می‌شود.

۴. هنگامی که چند علت حیاتی شناسایی شدند، حرکت برای تصحیح مشکلاتی که باعث این نقایص شده‌اند، آغاز می‌شود.

این مفهوم نسبتاً ساده، گام مهمی در راستای ایجاد یک فرایند نرم‌افزار تطبیقی است که در آن تغییراتی صورت می‌پذیرد تا عناصری از فرایند را بهبود بخشد که تولید خطا می‌کنند.

۱-۵-۱۶ یک مثال کلی

برای روشن شدن این فرایند، فرض کنید یک سازمان مهندسی نرم‌افزار، اطلاعاتی درباره نقایص را برای یک دوره‌ی یک‌ساله جمع‌آوری می‌کند. برخی از نقایص در اثنای توسعه نرم‌افزار کشف می‌شوند. بقیه نقایص پس از ارائه نرم‌افزار به کاربر نهایی به چشم می‌خورند. گرچه صدها خطای مختلف کشف می‌شوند، ریشه‌ی همه‌ی آنها را می‌توان در یک (یا چند) مورد از علل زیر یافت:

- مشخصات نادرست یا ناقص (IES)
- تفسیر نادرست ارتباط با مشتری (MCC)
- انحراف عمدی از مشخصات (IDS)
- عدول از استانداردهای برنامه‌نویسی (VPS)
- خطا در نمایش داده‌ها (EDR)
- واسط ناسازگار مؤلفه‌ها (ICI)
- خطا در منطق طراحی (EDL)
- آزمون نادرست یا ناقص (IET)
- مستندسازی نادرست یا ناقص (IID)
- خطا در ترجمه طراحی به زبان برنامه‌نویسی (PLT)
- رابطه ناسازگار یا مبهم انسان – ماشین (HCI)
- متفرقه (MIS)

برای اعمال SQA آماری، جدول ارائه شده در شکل ۲-۱۶ ساخته می‌شود. این جدول نشان می‌دهد که IES، MCC و EDR چند علت حیاتی هستند که باعث ۵۳٪ از کل خطاها می‌شوند. ولی لازم به ذکر است که اگر فقط خطاهای جدی در نظر گرفته شوند، IES، EDR، PLT و EDL هستند که به عنوان علل حیاتی انتخاب می‌شوند. هنگامی که علل حیاتی تعیین شدند، سازمان مهندسی نرم‌افزار می‌تواند عملیات تصحیحی را آغاز کند. برای مثال، جهت تصحیح MCC، نرم‌افزار نویس ممکن است تکنیک‌های مشخصات کاربرد تسهیل شده را پیاده‌سازی کند (فصل ۵) تا کیفیت ارتباط با مشتری و مشخصه بالا رود. برای بهبود بخشیدن به EDR، سازنده ممکن است ابزارهای کِیس (CASE) را برای مدل‌سازی داده‌ها و اجرای بازبینی اجباری طراحی داده‌ها به‌کار گیرد.

«تحلیل آماری‌ای که به‌طرز مناسب اجرا شده باشد، کالبدشکافی ظریف عدم قطعیت‌هاست، جراحی فرضیات.»

ام. جی. موارتی

«۸۰ درصد خطاها در ۲۰ درصد از کدها قرار دارند. آن‌ها را بیابید و تصحیح کنید!»

لاول آرتور

شکل ۲-۱۶ جمع‌آوری داده‌ها برای SQA آماری.

	جزئی		میانه		جدی		کل	
خطا	تعداد	%	تعداد	%	تعداد	%	تعداد	%
IES	103	24%	68	18%	34	27%	205	22%
MCC	76	17%	68	18%	12	9%	156	17%
IDS	23	5%	24	6%	1	1%	48	5%
VPS	10	2%	15	4%	0	0%	25	3%
EDR	36	8%	68	18%	26	20%	130	14%
ICI	31	7%	18	5%	9	7%	58	6%
EDL	19	4%	12	3%	14	11%	45	5%
IET	48	11%	35	9%	12	9%	95	10%
IID	14	3%	20	5%	2	2%	36	4%
PLT	26	6%	19	5%	15	12%	60	6%
HCI	8	2%	17	4%	3	2%	28	3%
MIS	41	9%	15	4%	0	0%	56	6%
Totals	435	100%	379	100%	128	100%	942	100%

شایان ذکر است که در عملیات تصحیح، اساساً همان علل حیاتی مورد تأکید قرار می‌گیرند. به موازاتی که این علل حیاتی تصحیح می‌شوند، نامزدهای جدیدی به صدر جدول منتقل می‌شوند.

نشان داده شده است که تکنیک‌های تضمین کیفیت آماری برای نرم‌افزار، تأثیر چشمگیری بر بهبودبخشیدن به کیفیت داشته‌اند [Art97]. در برخی موارد، سازمان‌های نرم‌افزاری پس از به‌کارگیری این تکنیک‌ها سالانه ۵۰٪ از میزان نقایص کاسته‌اند.

استفاده از SQA آماری و اصل پارتو را می‌توان در یک جمله خلاصه کرد: *وقت خود را صرف اموری کنید که واقعاً اهمیت دارند، اما نخست اطمینان حاصل کنید که می‌دانید واقعاً چه چیزهایی اهمیت دارند!*

۲-۵-۱۶ شش سیگما برای مهندسی نرم‌افزار

شش سیگما پر کاربردترین راهبرد برای تضمین کیفیت آماری در صنایع کنونی است. راهبرد شش‌سیگما، که در ابتدا توسط موتورولا در دهه ۱۹۸۰ به عموم شناسانده شد، «یک روش‌شناسی شدید و منضبط است که از داده‌ها و تحلیل آماری برای اندازه‌گیری عملکرد شرکت و بهبود بخشیدن به آن از طریق شناسایی و حذف نقایص در فرایندهای تولیدی و خدماتی بهره می‌برد.» [ISI08]. اصطلاح شش‌سیگما از شش برابر مقدار «انحراف معیار» به‌دست آمده است که معادل با ۳/۴ نمونه (معیوب) به‌ازای هر یک میلیون نمونه است- و این حد اعلای استانداردهای کیفیتی است. در روش‌شناسی شش سیگما، سه مرحله اصلی وجود دارد:

- *تعریف* خواسته‌های مشتری، محصولات قابل تحویل و اهداف پروژه از طریق روش‌های کاملاً مشخص برای برقراری ارتباط با مشتری.
- *اندازه‌گیری* فرایند موجود و خروجی آن برای تعیین کیفیت فعلی (جمع‌آوری معیارهای نقص)
- *تحلیل* معیارهای نقص و تعیین چند علت حیاتی.

اگر یک فرایند نرم‌افزار موجود، در جای خود باشد، ولی به بهبود نیاز داشته باشد، شش سیگما دو مرحله اضافی برای آن پیشنهاد می‌کند:

مراحل اصلی روش شناسی شش‌سیگما کدام است؟

- *بهبود بخشیدن* به فرایند با حذف علل ریشه‌ای نقایص.
- *کنترل* فرایند برای حصول اطمینان از این که کارهای آینده باعث ورود دوباره‌ی علل این نقایص نخواهد شد.

این مراحل اصلی و اضافی را گاهی روش DMAIC (تعریف، اندازه‌گیری، تحلیل، بهبودبخشی و کنترل) می‌نامند.

اگر سازمانی در حال توسعه‌ی یک فرایند نرم‌افزار (به جای بهبود بخشیدن به یک فرایند موجود) است، مراحل اصلی به‌صورت زیر بسط پیدا می‌کنند:

- *طراحی* فرایند برای (۱) پرهیز از علل ریشه‌ای نقایص و (۲) برآورده ساختن خواسته‌های مشتریان.
- *وارسی* این که مدل فرایندی واقعاً از نقایص پرهیز می‌کند و خواسته‌های مشتری را برآورده می‌سازد.

این شکل اصلاح‌شده را گاهی روش DMADV (تعریف، اندازه‌گیری، تحلیل، طراحی و وارسی) می‌نامند.

بحث جامعی درباره شش سیگما را می‌توانید در منابع مربوط به همین مبحث دنبال کنید. در صورت علاقه بیشتر می‌توانید [ISI08]، [Pyz03] و [Snc03] را ببینید.

۶-۱۶ قابلیت اطمینان نرم‌افزار

بدون تردید، قابلیت اطمینان یک برنامه کامپیوتری، عنصر مهمی از کیفیت کلی آن به‌شمار می‌رود. اگر برنامه‌ای به کرّات از اجرا باز بماند، عوامل کیفیتی دیگر نرم‌افزار، اهمیت خود را از دست خواهند داد.

قابلیت اطمینان برخلاف عوامل کیفیتی دیگر، با استفاده از داده‌های تاریخی و توسعه‌ای، مستقیماً قابل برآورد و اندازه‌گیری است. قابلیت اطمینان به زبان آماری، به عنوان «احتمال عملکرد بدون شکست یک برنامه کامپیوتری در محیطی مشخص برای یک زمان معین» تعریف می‌شود [Mus87]. برای مثال، برآورد می‌شود که برنامه X در عرض هشت ساعت پردازش دارای قابلیت اطمینان ۰/۹۹۹ است. به عبارت دیگر، اگر قرار باشد در عرض هشت ساعت پردازش (زمان اجرا) این برنامه ۱۰۰۰ بار اجرا شود، احتمالاً ۹۹۹ بار از ۱۰۰۰ بار را درست کار می‌کند.

هرگاه قابلیت اطمینان موردبحث قرار می‌گیرد، یک سؤال محوری مطرح می‌شود: منظور از واژه‌ی شکست چیست؟ در حیطه بحث کیفیت نرم‌افزار و قابلیت اطمینان، شکست عبارت از عدم همخوانی با خواسته‌های نرم‌افزار است. با این وجود، حتی در همین حیطه نیز درجات مختلفی از شکست وجود دارد. شکست‌ها ممکن است ناراحت کننده یا فاجعه‌بار باشند. یک شکست را شاید در عرض چند ثانیه بتوان تصحیح کرد، حال آنکه تصحیح شکست دیگر شاید به هفته‌ها یا حتی ماه‌ها زمان نیاز داشته باشد. اگر بخواهیم مسأله را پیچیده‌تر کنیم، تصحیح یک شکست ممکن است منجر به وارد شدن خطاهای دیگر شود که آنها نیز منجر به شکست‌های دیگر می‌شوند.

«بهای گریزناپذیر قابلیت اطمینان، سادگی است.»

سی. ای. آر. هوار

۱-۶-۱۶ موازین مرتبط با قابلیت اطمینان و دسترس‌پذیری

در کارهای اولیه‌ای که در زمینه قابلیت نگهداری نرم‌افزار صورت پذیرفت، کوشش شد تا یک نظریه ریاضی برای قابلیت اطمینان سخت‌افزار (مثل [ALV64]) و نیز پیش‌بینی قابلیت اطمینان نرم‌افزار ارائه

شود. اکثر مدل‌های قابلیت اطمینان مرتبط با سخت‌افزار، مبتنی بر شکست‌های ناشی از فرسودگی هستند نه نقایص طراحی. در سخت‌افزار، احتمال شکست‌های ناشی از فرسودگی فیزیکی (مثل اثرات دما، خوردگی، شوک) نسبت به شکست‌های طراحی بیشتر است. متأسفانه، در مورد نرم‌افزار، عکس این مورد صحت دارد. درواقع ریشه همه‌ی شکست‌های نرم‌افزاری را می‌توان در مشکلات طراحی یا پیاده‌سازی جستجو نمود؛ فرسایش (فصل ۱) در اینجا نقشی ندارد.

هنوز بر سر روابط میان مفاهیم کلیدی موجود در قابلیت اطمینان سخت‌افزار و کاربرد آنها در نرم‌افزار اختلاف نظر وجود دارد. گرچه هنوز این روابط باید به اثبات برسند، بد نیست چند مفهوم ساده را در نظر بگیریم که در هر دو عنصر سیستم کاربرد داشته باشند.

اگر یک سیستم کامپیوتری را در نظر بگیریم، میزان ساده‌ای از قابلیت اطمینان، *زمان میانگین بین شکست* (MTBF) است که در آن:

$$MTBF = MTTF + MTTR$$

که MTTF و MTTR به‌ترتیب زمان میانگین شکست و زمان میانگین ترمیم هستند[1].

بسیاری از پژوهش‌گران معتقدند MTBF به مراتب مفیدتر از سایر معیارهای مرتبط با کیفیت نرم‌افزارند که در فصل ۲۳ بحث خواهند شد. به بیان ساده‌تر، کاربر نهایی با شکست‌ها سروکار دارد نه با تعداد کل خطاها. از آنجا که همه‌ی خطاهای موجود در یک برنامه، نسبت شکست یکسانی ندارند، تعداد کل خطاها، شاخص ضعیفی از قابلیت اطمینان سیستم را به دست می‌دهد. برای مثال، برنامه‌ای را در نظر بگیرید که مدت ۳۰۰۰۰ ساعتِ CPU در حال کار بوده است. ممکن است نقایص بسیاری مدت‌ها قبل از اینکه کشف شوند، خود را آشکار نکنند. MTBF چنین خطاهای پنهانی ممکن است ۳۰۰۰۰ ساعتِ CPU یا حتی ۶۰۰۰۰ ساعتِ CPU باشد. نسبت شکست خطاهای دیگری که هنوز کشف نشده‌اند، ممکن است ۴۰۰۰ تا ۵۰۰۰ ساعتِ CPU باشد. حتی اگر همه‌ی خطاهای گروه اول (آنهایی که MTBF درازمدتی دارند) حذف شوند، اثری که بر قابلیت اطمینان دارند، قابل چشم‌پوشی است.

ولی MTBF به دو دلیل می‌تواند مشکل‌آفرین باشد: (۱) یک بازه زمانی را میان دو شکست تصویر می‌کند، ولی میزانی از شکست به‌دست نمی‌دهد و (۲) MTBF را ممکن است به غلط به‌عنوان بازه زمانی میانگین تعبیر کنند، هر چند که معنای آن این نیست.

یک میزان دیگر برای قابلیت اطمینان، شکست در زمان (FIT) است- میزانی آماری از تعداد شکست‌هایی است که یک مؤلفه طی یک میلیارد ساعت کار از خود ممکن است نشان دهد. بنابراین، یک FIT معادل با یک شکست در هر یک میلیارد ساعت عملکرد است.

علاوه بر میزان قابلیت اطمینان، باید میزانی از دسترس‌پذیری نیز تدارک ببینیم. *دسترس‌پذیری نرم‌افزار* عبارت است از احتمال کارکردن برنامه طبق خواسته‌ها در یک نقطه مفروض از زمان، که به‌صورت زیر تعریف می‌شود:

$$\text{دسترس‌پذیری} = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

[1] گرچه ممکن است به‌عنوان پیامدی از شکست، نیاز به اشکال‌زدایی (و تصحیحات مربوط به آن) وجود داشته باشد، نرم‌افزار در بسیاری موارد، پس از شروع دوباره بدون هیچ گونه تغییر دوباره به خوبی کار خواهد کرد.

نکته‌ی کلیدی

ریشه مسائل قابلیت اطمینان نرم‌افزار را تقریباً همیشه می‌توان در نقایص طراحی یا پیاده‌سازی یافت.

نکته‌ی کلیدی

ذکر این نکته حائز اهمیت است که MTBF و موازین مرتبط با آن مبتنی بر زمان CPU هستند و نه زمان ساعت دیواری.

اندرز

برخی جنبه‌های دسترس‌پذیری (که در این‌جا بحث نشده‌اند) ربطی به شکست ندارند. برای مثال، زمان‌بندی برای اوقات خرابی سیستم (برای پشتیبانی از قابلیت‌های عملیاتی) باعث می‌شود تا نرم‌افزار در دسترس قرار نگیرد.

میزان قابلیت اطمینان MTBF به یک اندازه نسبت به MTTF و MTTR حساس است. میزان دسترس‌پذیری تا حدی نسبت به MTTR که میزان غیرمستقیمی از قابلیت نگهداری نرم‌افزار است، حساس است.

۲-۶-۱۶ ایمنی نرم‌افزار (Software Safety)

ایمنی نرم‌افزار یکی از فعالیت‌های تضمین کیفیت است که بر شناسایی و سنجش ریسک‌های بالقوه‌ای تأکید دارد که ممکن است تأثیری منفی بر نرم‌افزار داشته منجر به شکست کل سیستم شود. اگر بتوان ریسک‌ها را در همان ابتدای فرایند نرم‌افزار شناسایی کرد، می‌توان ویژگی‌هایی را در طراحی نرم‌افزار مشخص کرد که ریسک‌های بالقوه را حذف یا کنترل کند.

به عنوان بخشی از ایمنی نرم‌افزار، یک فرایند مدل‌سازی و تحلیل به اجرا درمی‌آید. در آغاز، ریسک‌های شناسایی و براساس اهمیت و احتمال وقوعی که دارند، گروه‌بندی می‌شوند. برای مثال، برخی ریسک‌های مربوط به کنترل گشت کامپیوتری با یک خودرو عبارتنداز: (۱) باعث شتاب کنترل‌نشده‌ای می‌شود که غیرقابل توقف است؛ (۲) به فشاردادن پدال ترمز پاسخ نمی‌دهد (پس از خاموش‌شدن)؛ (۳) وقتی سویچ فعال می‌شود، سیستم کار نمی‌کند؛ (۴) سرعت به آهستگی زیاد و کم می‌شود؛ هنگامی که این ریسک‌های سیستمی شناسایی شدند، برای تعیین شدت و احتمال وقوع هر یک، از تکنیک‌های تحلیل استفاده می‌شود. برای آنکه نرم‌افزار مؤثر واقع شود، باید در کل سیستم تحلیل شود[1]. برای مثال، یک خطای ظریف در ورودی کاربر (افراد از مؤلفه‌های سیستم به شمار می‌روند) ممکن است ترسط یک نقص نرم‌افزاری در تولید داده‌های کنترلی تشدید شود و یک دستگاه مکانیکی نامناسب را به کار اندازد. اگر مجموعه‌ای از شرایط محیطی خارجی رعایت شوند (و فقط اگر رعایت شوند) موقعیت نامناسب دستگاه مکانیکی باعث یک شکست مصیبت‌بار می‌شود. تکنیک‌های تحلیل نظیر تحلیل درخت خطاها [Eri05]، منطق زمان حقیقی یا مدل‌های پتری‌نت را می‌توان برای پیش‌بینی زنجیره وقایعی به کار برد که باعث بروز ریسک‌ها شده‌اند و احتمال رخ دادن هر یک از این رویدادها را برای ایجاد این زنجیره پیش‌بینی کرد.

هنگامی که ریسک‌ها شناسایی و تحلیل شد، خواسته‌های مرتبط با ایمنی را می‌توان برای نرم‌افزار مشخص کرد. یعنی، مشخصه می‌تواند حاوی لیستی از رویدادهای نامطلوب و پاسخ‌های سیستمی مطلوب به این رویدادها باشد. سپس نقش نرم‌افزار در مدیریت رویدادهای نامطلوب مشخص می‌شود.

گرچه قابلیت اطمینان نرم‌افزار و ایمنی نرم‌افزار ارتباطی تنگاتنگ با هم دارند، درک اختلاف ظریفی که بین آنها وجود دارد، حائز اهمیت است. در قابلیت اطمینان نرم‌افزار، از تحلیل آماری برای تعیین احتمال وقوع شکست نرم‌افزار استفاده می‌شود. به‌هرحال، وقوع یک شکست الزاماً به ریسک یا اتفاق سوء نمی‌انجامد. یعنی شکست‌ها در خلاء در نظر گرفته نمی‌شوند، بلکه در کلیت سیستم کامپیوتری ارزیابی می‌شوند.

بحث جامع ایمنی نرم‌افزار خارج از حوصله‌ی این کتاب است. خوانندگان علاقه‌مند می‌توانند برای اطلاعات بیشتر درخصوص ایمنی نرم‌افزار و مباحث مرتبط با آن به [Smi05]،[Dun02] و [Lev95] رجوع کنند.

[1] این رویکرد مشابه با روش‌های تحلیل ریسکی است که در فصل ۲۸ بحث خواهد شد. اختلاف اصلی در مسائل فن‌آوری است نه مباحث مرتبط با پروژه.

«ایمنی انسان‌ها باید بالاترین قانون باشد.»

سیسرون

«نمی‌توانم شرایطی را تصور کنم که باعث غرق این کشتی شود. کشتی‌سازی مدرن این مشکل را پشت سر گذاشته است.»

ای. جی. اسمیت (ناخدای تایتانیک)

مرجع وب

مجموعه ارزش‌مندی از مقالات درباره ایمنی نرم‌افزارها را می‌توان در وب‌سایت زیر یافت:
www.software-eng.com

اطلاعات

استاندارد ISO9001/2000

در طرحی که به‌دنبال خواهد آمد، عناصر اصلی استاندارد ISO9001/2000 تعریف می‌شود. اطلاعات جامع درباره این استاندارد را می‌توانید از ISO (**www.iso.ch**) و سایر منابع داخلی (**www.parxiom.com**) به‌دست آورید.

تعیین عناصر یک سیستم مدیریت کیفیت.
توسعه، پیاده‌سازی و بهبود بخشیدن به سیستم.
تعریف یک خط مشی که بر اهمیت سیستم تأکید داشته باشد.
مستندسازی سیستم کیفیت.
توصیف فرایند.
تولید یک جزوه عملیاتی.
توسعه روش‌های کنترل (بهنگام سازی) مستندات.
وضع روش‌هایی برای حفظ سوابق.
پشتیبانی از تضمین و کنترل کیفیت.
ارتقا بخشیدن به اهمیت کیفیت در میان طرف‌های ذی‌نفع.
توجه ویژه به رضایت مشتری.
تعریف یک طرح کیفیتی که به اهداف، مسؤولیت‌ها و مدیریت می‌پردازد.
تعریف سازوکارهای مستندسازی در میان طرف‌های ذی‌نفع.
وضع سازوکارهای بازبینی برای سیستم مدیریت کیفیت.
شناسایی روش‌های بازبینی و سازوکارهای بازخوردی.
تعریف روال‌های پیگیری.
شناسایی منابع کیفیتی شامل عناصر پرسنلی، آموزشی و زیرساختی.
وضع سازوکارهای کنترلی.
برای برنامه‌ریزی.
برای خواسته‌های مشتری.
برای فعالیت‌های فنی (مانند تحلیل، طراحی و آزمون).
برای پایش و مدیریت پروژه.
تعریف روش‌هایی برای تصحیح.
ارزیابی داده‌ها و معیارهای کیفیتی.
تعریف رویکردی برای بهبود پیوسته‌ی فرایند و کیفیت.

۷-۱۶ استاندارد کیفیتی ISO 9001

سیستم تضمین کیفیت را می‌توان به عنوان ساختار سازمانی، مسؤولیت ها، روال ها، فرایندها و منابع پیاده‌سازی مدیریت کیفیت تعریف کرد [ANS87]. سیستم‌های تضمین کیفیت به این منظور ایجاد می‌شوند که به سازمان کمک کنند تا اطمینان حاصل کنند رضایت مشتریان حاصل شده است و آنچه مشخص کرده‌اند، برآورده شده است. این سیستم‌ها گستره‌ی وسیعی از فعالیت‌ها را پوشش می‌دهند که

ابزارهای نرم‌افزاری

مدیریت کیفیت نرم‌افزار

هدف: هدف ابزارهای SQA، کمک به تیم پروژه در ارزیابی و بهبود بخشیدن به کیفیت محصول کاری نرم‌افزار است.

مکانیک: مکانیک این ابزارها متغیر است. به‌طور کلی، هدف دستیابی به کیفیت یک محصول کاری ویژه است. توجه: آرایه گسترده‌ای از ابزارهای آزمون (فصل‌های ۱۷ تا ۲۰) غالباً در ابزارهای SQA گنجانده می‌شوند.

ابزارهای نمونه

ARM: که توسط NASA(satc.gsfc.nasa.gov/tools/index.html) توسعه یافته است و موازینی فراهم می‌آورد که می‌توان آن‌ها را در ارزیابی مستندات مربوط به خواسته‌های نرم‌افزار به‌کار برد.

راهنمای فرایند QPR که توسط شرکت نرم‌افزاری QPR (**www.qpronline.com**) توسعه یافته است و پشتیبانی برای شش سیگما و سایر رویکردهای مدیریت کیفیت را فراهم می‌آورد.

قالب‌ها و ابزارهای کیفیتی که توسط iSixSigma (**www.isixsigma.com/tt/**) توسعه یافته است و آرایه گسترده‌ای از روش‌ها و ابزارهای مفید برای مدیریت کیفیت را توصیف می‌کند.

NASA Quality Resources که توسط مرکز پروازهای فضایی گودارد (**sw-assurance.gsfc.nasa.gov/index.php**) توسعه یافته است و فرم‌ها، قالب‌ها، چک‌لیست‌ها و ابزارهایی برای SQA فراهم می‌سازد.

کل چرخهٔ حیات محصول را از طرح ریزی، کنترل، اندازه‌گیری، آزمون و گزارش‌دهی و بهبود سطوح کیفیت در سرتاسر فرایند توسعه و تولید در بر می‌گیرد. ISO 9000 عناصر کیفیتی را به زبان کلی توصیف می‌کند که برای هر تجارتی با هر محصول یا خدماتی که ارائه می‌دهد، قابل اعمال هستند.

برای اینکه نام شرکتی در یکی از مدل‌های تضمین کیفیت ISO 9000 ثبت گردد، سیستم و عملیات‌های کیفیت شرکت باید توسط یک شرکت مجاز ممیزی شود تا پیروی آن از این استاندارد به تأیید برسد. پس از ثبت موفق، یک گواهی از سوی شرکت ممیزی برای این شرکت صادر می‌شود. هر شش ماه یک بار شرکت دوباره مورد ممیزی قرار می‌گیرد تا از پیروی مستمر از استاندارد اطمینان حاصل شود.

الزامات ترسیم شده توسط ISO 9000:2000 به مباحثی همچون مسؤولیت پذیری مدیران، سیستم کیفیت، بازبینی قراردادها، کنترل طراحی، کنترل داده‌ها و مستندات، اقدامات تصحیحی و پیش گیرانه، کنترل سوابق کیفیتی، ممیزی‌های کیفیتی درونی، آموزش، خدمات رسانی و تکنیک‌های آماری می‌پردازد. برای آن که یک شرکت نرم‌افزاری موفق به اخذ گواهینامه ISO 9000:2000 شود، باید برای پرداختن به هر کدام از الزامات ذکر شده در بالا، خط مشی‌ها و روال‌هایی را تعیین کند و سپس بتواند نشان دهد که این خط مشی‌ها و روال‌ها دنبال می‌شوند. در صورت تمایل به کسب اطلاعات بیشتر در خصوص ISO 9000:2000 می‌توانید [Ant06]، [Mut03] یا [Dob04] را ببینید.

مرجع وب

چندین پیوند متهی به منابع ISO9000/9001 را می‌توان در آدرس زیر یافت:
www.tantara.ab.ca/info.htm

۱۶-۸ طرح SQA

طرح SQA راهنمایی برای نهادینه کردن تضمین کیفیت نرم‌افزار فراهم می‌آورد. این طرح که توسط گروه SQA تهیه می‌شود، به عنوان الگویی برای فعالیت‌های SQA عمل می‌کند که برای هر پروژه نرم‌افزاری نهادینه می‌شوند.

IEEE استانداردی برای طرح‌های SQA پیشنهاد کرده است [IEEE93]. این استاندارد ساختاری را پیشنهاد می‌کند که در آن موارد زیر باید مشخص گردد: (۱) هدف و دامنه کاربرد طرح؛ (۲) توصیفی از همه‌ی محصولات کاری مهندسی نرم‌افزار؛ (۳) همه‌ی استانداردهای قابل استفاده در طول فرایند نرم‌افزار؛ (۴) وظایف و کنش‌های SQA (ازجمله مرورها و ممیزی‌ها) و محل قرارگرفتن آنها در سرتاسر فرایند نرم‌افزار؛ (۵) ابزارها و روش‌هایی که از وظایف و کنش‌های SQA پشتیبانی می‌کنند؛ (۶) رویه‌های مدیریت پیکربندی نرم‌افزار (فصل ۲۲)؛ (۷) روش‌های مونتاژ، ایمن‌سازی و نگهداری از کلیه سوابق مرتبط با SQA و (۸) نقش‌ها و مسؤولیت‌های سازمانی در قبال کیفیت محصول. شرح داده شده است.

۱۶-۹ خلاصه

تضمین کیفیت یکی از فعالیت‌های چتری است که در هر مراحل فرایند نرم‌افزار اجرا می‌شود. SQA شامل روال‌هایی برای به‌کارگیری مؤثر روش‌ها و ابزارها؛ بازبینی‌های فنی رسمی؛ راهبردها و تکنیک‌های آزمون، دستگاه‌های پوکایوک؛ روال‌هایی برای کنترل تغییرات؛ روال‌هایی برای حصول اطمینان از مطابقت با استانداردها، و سازوکارهای اندازه‌گیری و گزارش‌دهی می‌شود.

برای اجرای مناسب تضمین کیفیت، داده‌های مربوط به فرایند مهندسی نرم‌افزار را باید جمع‌آوری، ارزیابی و تکثیر کرد. SQA آماری به بهسازی کیفیت محصول و خود فرایند نرم‌افزار کمک می‌کند. مدل‌های قابلیت اطمینان نرم‌افزار با گسترش دادن اندازه‌گیری‌ها، امکان برون‌یابی داده‌های مربوط به نقایص جمع‌آوری شده و به دست آوردن نسبت‌های شکست پیش‌بینی شده و پیش‌بینی قابلیت اطمینان را فراهم می‌آورند.

به‌طور خلاصه، باید به سخنان دان و اولمان [Dun82] توجه داشته باشید که می‌گویند: «تضمین کیفیت نرم‌افزار، نقشی است که ادراک‌های مدیریتی و شاخه‌های طراحی تضمین کیفیت بر فضای مدیریتی و فن‌آوری مهندسی نرم‌افزار می‌زنند.» توانایی حصول اطمینان از کیفیت، میزانی از رشد و بلوغ یک رشته مهندسی است. هنگامی که این نقش زدن با موفقیت انجام شود، مهندسی نرم‌افزار بلوغ یافته است.

مسائل و نکاتی برای تعمق

۱-۱۶ عده‌ای بر این باورند که «گوناگونی تغییرات، قلب کنترل کیفیت است.» چون هر برنامه‌ای که ایجاد می‌شود با هر برنامه دیگر متفاوت است، در جستجوی چه تغییراتی هستیم و آنها را چگونه کنترل می‌کنیم؟

۲-۱۶ آیا اگر مشتری پیوسته نظر خود را درباره آنچه که باید انجام شود، تغییر دهد، امکان دستیابی به کیفیت نرم‌افزار وجود دارد؟

۳-۱۶ کیفیت و قابلیت اطمینان مفاهیمی مرتبط هستند، ولی از جهاتی اساساً با هم تفاوت دارند. آنها را مورد بحث قرار دهید.

۴-۱۶ آیا ممکن است برنامه‌ای بدون نقص و در عین حال غیرقابل اطمینان باشد؟ توضیح دهید.

۵-۱۶ آیا ممکن است برنامه‌ای بدون نقص بوده در عین حال از خود کیفیتی نشان ندهد؟

۶-۱۶ چرا غالباً بین گروه مهندسی نرم‌افزار و یک گروه تضمین کیفیت نرم‌افزار مستقل تنش وجود دارد؟ آیا این اشکال ندارد؟

۷-۱۶ به شما مسؤولیت داده شده است تا کیفیت نرم‌افزارهای سازمان خود را بهبود ببخشید. اولین کاری که باید بکنید چیست؟ گام بعدی کدام است؟

۸-۱۶ آیا علاوه بر شمارش خطاها، ویژگی‌های قابل شمارش دیگری از نرم‌افزار نیز وجود دارند که مبین کیفیت باشند؟ آنها کدامند و آیا به طور مستقیم قابل اندازه‌گیری هستند؟

۹-۱۶ مفهوم MTBF برای نرم‌افزار جای نقد دارد. آیا می‌توانید چند دلیل بیاورید.

۱۰-۱۶ دو سیستم ایمنی بحرانی در نظر بگیرید که توسط کامپیوتر کنترل می‌شوند. حداقل سه ریسک برای هر کدام ذکر کنید که مستقیماً به شکست‌های نرم‌افزاری مربوط باشند.

۱۱-۱۶ یک نسخه از استانداردهای ISO 9001:2000 و ISO 9000-3 به‌دست آورید. سه مورد از خواسته‌های ISO 9001 و چگونگی اجرای آنها در حیطه نرم‌افزار را مورد بحث قرار دهید.

فصل ۱۷

راهبردهای آزمون نرم‌افزار

نگاهی گذرا

راهبرد آزمون چیست؟ نرم‌افزار مورد آزمون قرار می‌گیرد تا خطاهایی که سهواً در طراحی و پیاده‌سازی وارد شده‌اند، برملا شوند، ولی برای اجرای آنها چگونه باید عمل کرد؟ آیا باید یک طرح رسمی برای آزمون‌های خود تهیه کنید؟ آیا باید کل برنامه را آزمون کنید یا فقط بخش‌های کوچکی از آن را آزمون کنید؟ آیا باید آزمون‌هایی را که قبلاً اجرا کرده‌اید با افزودن مؤلفه‌های جدید به یک سیستم بزرگ، دوباره اجرا کنید؟ مشتری را چه هنگام باید در آزمون شرکت داد؟ هنگام توسعه یک راهبرد آزمون نرم‌افزار به این پرسش‌ها و پرسش‌های دیگر پاسخ داده می‌شود.

چه کسی آن را انجام می‌دهد؟ راهبرد آزمون نرم‌افزار، توسط مدیر پروژه توسعه، مهندسان نرم‌افزار و متخصصان آزمون توسعه می‌یابد.

چرا اهمیت دارد؟ آزمون غالباً در میان کنش‌های مهندسی نرم‌افزار، بیشترین تلاش را به خود اختصاص می‌دهد. اگر بدون برنامه‌ریزی اجرا شود، زمان هدر می‌رود، کار بیهوده صرف می‌شود، خطاها کشف نشده باقی می‌مانند. بنابراین منطقی به نظر می‌رسد که یک راهبرد سیستماتیک برای آزمون نرم‌افزار وضع شود.

مراحل کار کدام است؟ آزمون در «ابعاد کوچک» آغاز می‌شود و به «ابعاد بزرگ» پیشرفت می‌کند. منظور آن است که در آزمون‌های اولیه بر یک مؤلفه منفرد تکیه می‌شود و آزمون‌های جعبه سفید و سیاه برای کشف خطاهای موجود در منطق و عملکرد برنامه به اجرا در می‌آیند. پس از آنکه مؤلفه‌ها به‌طور انفرادی آزمون شدند، باید با هم مجتمع شوند. آزمون به موازات ساخته شدن نرم‌افزار ادامه می‌یابد. سرانجام، وقتی برنامه‌ی کامل برای کار آماده شد، آزمون‌های سطح بالا اجرا می‌شود. این آزمون‌ها برای کشف خطاهای موجود در خواسته‌ها طراحی می‌شوند.

محصول کاری چیست؟ *مشخصات آزمون* که روش تیم نرم‌افزاری برای انجام آزمون را با تعریف طرحی مستندسازی می‌کند. رویه‌ای که مراحل مشخص را تعریف می‌کند و آزمون‌هایی که اجرا خواهند شد.

چگونه اطمینان حاصل کنم که درست از عهده امور برآمده‌ام؟ با بازبینی مشخصات آزمون، قبل از انجام آزمون می‌توانید کامل بودن موارد آزمون و وظایف آزمون را بسنجید. یک طرح و رویه آزمون اثربخش، منجر به ساخت منظم نرم‌افزار و کشف خطاها در هر مرحله از فرایند می‌شود.

راهبرد آزمون نرم‌افزار، نقشه راهنمایی فراهم می‌آورد که مراحل اجرای آزمون، زمان برنامه‌ریزی و میزان کار، زمان و منابع لازم را توصیف می‌کند. بنابراین هر راهبرد آزمونی باید برنامه‌ریزی آزمون، طراحی موارد آزمون، اجرای آزمون و جمع‌آوری و ارزیابی داده‌های حاصل را با هم مرتبط کند.

راهبرد آزمون نرم‌افزار باید آنقدر انعطاف‌پذیر باشد که روش آزمون سفارشی را ارتقاء بخشد. در عین حال، باید به قدر کافی محکم باشد که برنامه‌ریزی منطقی و پیگیری مدیریتی را به موازات پیشرفت پروژه ارتقاء بخشد. شومان [Sho83] این مسائل را مورد بحث قرار می‌دهد:

> آزمون، از بسیاری جهات یک فرایند فردگرایانه است و تعداد انواع متفاوت آزمون‌ها بسته به تعداد روش‌های متفاوت توسعه، متغیر است. سال‌ها بود که تنها دفاع در برابر خطاهای برنامه‌نویسی، دقت در طراحی و هوش و فراست خود برنامه‌نویس بود. اکنون در دوره‌ای به‌سر می‌بریم که در آن تکنیک‌های مدرن طراحی (و بازبینی‌های رسمی فنی) به کاهش‌دادن تعداد خطاهای اولیه‌ای که ذاتاً در کد وجود دارند، کمک می‌کنند. به‌طور مشابه، روش‌های متفاوت آزمون رفته‌رفته به‌صورت چند روش و فلسفه متمایز در آمده است.

این «رویکردها و فلسفه‌ها» همان چیزهایی هستند که آنها را راهبرد خواهیم نامید و توجه خود را به آنها معطوف خواهیم کرد. در فصل‌های ۱۸ تا ۲۰، روش‌های آزمون نرم‌افزار ارائه خواهد شد.

۱-۱۷ روشی راهبردی برای آزمون نرم‌افزار

آزمون، مجموعه‌ای از فعالیت‌هاست که می‌توان آنها را از قبل برنامه‌ریزی کرد و به‌طور سیستماتیک اجرا نمود. به همین دلیل، باید الگویی برای آزمون نرم‌افزارها – مجموعه‌ای از مراحل برای اجرای تکنیک‌های طراحی موارد آزمون و روش‌های آزمون – در فرایند نرم‌افزار تعریف شود.

چند راهبردآزمون پیشنهاد شده است. همه‌ی این راهبردها الگویی برای آزمون در اختیار سازنده‌ی نرم‌افزار قرار می‌دهند و همگی دارای ویژگی‌های کلی زیرند:

- برای اجرای آزمون اثربخش، باید بازبینی‌های فنی اثربخش (فصل ۱۵) اجرا شود. با این اقدام، بسیاری از خطاها پیش از آغازشدن آزمون، حذف خواهند شد.
- آزمون در سطح مؤلفه‌ها شروع می‌شود و رفته رفته کل سیستم کامپیوتری را در برمی‌گیرد.
- تکنیک‌های آزمون متفاوت برای رویکردهای متفاوت مهندسی نرم‌افزار و در نقاط زمانی مختلف، مناسب هستند.
- آزمون توسط سازنده نرم‌افزار و (برای پروژه‌های بزرگتر) توسط یک گروه آزمون‌گر مستقل انجام می‌شود.
- آزمون و اشکال‌زدایی دو فعالیت جداگانه‌اند، ولی اشکال‌زدایی باید در هر راهبرد آزمون، جایی داشته باشد.

راهبرد آزمون باید آزمون‌های سطح پایینی را انجام دهد که به کمک آنها بتوان وارسی کرد که بخش کوچکی از کد منبع، به‌طور صحیح پیاده‌سازی شده است و همچنین حاوی آزمون‌های سطح بالایی باشد که عملکرد اصلی سیستم‌ها را در مقابل خواسته‌های مشتری اعتبار بخشد. این راهبرد باید راهنمایی برای سازنده و یک مجموعه نشانه‌ها برای مدیر فراهم آورد. از آنجا که مراحل راهبرد آزمون در زمانی رخ می‌دهند که مهلت تعیین شده در حال پایان است، پیشرفت باید قابل اندازه‌گیری باشد و مشکلات باید حدالامکان بروز کنند.

مرجع وب

منابع مفیدی برای آزمون نرم‌افزار را می‌توانید در وب‌سایت زیر بیابید:
www.mtsu.edu/~storm

۱-۱-۱۷ وارسی و اعتبارسنجی (Verification and Validation)

آزمون نرم‌افزار، یکی از عناصر یک موضوع وسیع‌تر است که غالباً از آن به‌عنوان وارسی و اعتبارسنجی (V&V) یاد می‌شود. *وارسی* عبارت از یک مجموعه فعالیت‌هاست که پیاده‌سازی صحیح یک عملکرد خاص توسط نرم‌افزار را تضمین می‌کند. اعتبارسنجی عبارت از مجموعه متفاوتی از فعالیت‌هاست که تضمین می‌کنند نرم‌افزار ساخته شده با خواسته‌های مشتری مطابقت دارد. بوهم [Boe81] این نکته را به صورت دیگری بیان می‌کند:

> «آزمون، بخش آجاب‌ناپذیری از هر تلاش مسؤولانه برای توسعه‌ی یک سیستم نرم‌افزاری است.»
>
> ویلیام هوتن

وارسی: «آیا محصول را درست ساخته‌ایم؟»

اعتبارسنجی: «آیا محصول درست را ساخته‌ایم؟»

تعریف V&V بسیاری از فعالیت‌ها را دربرمی‌گیرد که از آنها به‌عنوان تضمین کیفیت نرم‌افزار (SQA) یاد کردیم (فصل ۱۶).[1]

وارسی و اعتبارسنجی شامل گستره‌ی وسیعی از فعالیت‌های SQA می‌شود که از آن جمله‌اند: بازبینی‌های فنی رسمی، ممیزی‌های کیفیتی و پیکربندی، نظارت بر کارایی، شبیه‌سازی، مطالعه امکان‌سنجی، بازبینی مستندات، بازبینی بانک‌های اطلاعاتی، تحلیل الگوریتم‌ها، آزمون توسعه، آزمون قابلیت‌ها و آزمون نصب. گرچه آزمون نقش مهمی در V&V دارد، بسیاری از فعالیت‌های دیگر نیز ضروری هستند.

> **اندرز**
> منظم باشید و به آزمون به دیده‌ی یک «تور ایمنی» بنگرید که همه‌ی خطاهای رخ داده به‌خاطر کارهای ضعیف مهندسی نرم‌افزار را برایتان می‌گیرد. چنین نیست.

آزمون، آخرین فرصت را برای سنجش کیفیت و کشف خطاها فراهم می‌آورد. ولی آزمون را نباید یک تور ایمنی پنداشت. معروف است که می‌گویند «کیفیت را نمی‌توان آزمود. اگر هنگامی که آزمون را شروع می‌کنید، کیفیتی در کار نباشد، وقتی هم که آزمون را به پایان می‌برید، باز کیفیتی وجود نخواهد داشت». کیفیت در سرتاسر فرایند مهندسی با آن همراه است. به‌کارگیری مناسب روش‌ها و ابزارها، بازبینی‌های فنی رسمی مؤثر و مدیریت و اندازه‌گیری‌های منسجم، همگی به کیفیتی منجر می‌شوند که در اثنای آزمون، مهر تأییدی بر آن زده می‌شود.

میلر [Mil77] ارتباط میان آزمون و تضمین کیفیت نرم‌افزار را چنین بیان می‌کند: «انگیزه‌ی زیربنایی آزمون برنامه آن است که کیفیت برنامه با روش‌هایی مورد تأیید قرار گیرد که در سیستم‌های مقیاس کوچک و بزرگ به‌طور اقتصادی و مؤثر قابل اجرا باشد».

۲-۱-۱۷ سازمان‌دهی برای آزمون نرم‌افزار

در هر پروژه نرم‌افزار، جدالی بر سر چگونگی شروع آزمون آغاز می‌شود. اکنون از کسانی که نرم‌افزار را ساخته‌اند درخواست می‌شود که آن را آزمون کنند. این به خودی خود امری زیان‌بار به نظر می‌رسد؛ با همه‌ی اینها، چه کسی برنامه را بهتر از سازندگان آن می‌شناسد؟ متأسفانه همین سازندگان علاقه زیادی دارند که نشان دهند برنامه‌شان عاری از خطا است، طبق خواسته‌های مشتری کار می‌کند و طبق زمان‌بندی و با بودجه تعیین شده کامل خواهد شد.

> «خوش‌بینی خطری شغلی در برنامه‌نویسی است که درمان آن، آزمون است.»
>
> کنت بک

[1] شایان ذکر است که درخصوص انواع آزمون‌های تشکیل‌دهنده «اعتبارسنجی» آرا بسیار متشتت است. برخی بر این باورند که هر آزمونی یک وارسی است و اعتبارسنجی هنگامی که خواسته‌ها بازبینی و تصویب می‌شوند و بعداً توسط کاربر، هنگام عملیاتی شدن نرم‌افزار اجرا می‌شود. عده ای دیگر، آزمون واحدها و انسجام (بخش‌های ۱-۳-۱۷ و ۲-۳-۱۷) و آزمون‌های مرتبه بالاتر (بخش‌های ۶-۱۷ و ۷-۱۷) را اعتبارسنجی می‌دانند.

از دیدگاه روان‌شناختی، تحلیل و طراحی نرم‌افزار (همراه با کدنویسی) وظایفی سازنده هستند. مهندس نرم‌افزار، برنامه کامپیوتری، مستندات و ساختمان داده‌های مربوط به آن را خلق می‌کند. همانند هر سازنده دیگر، مهندس نرم‌افزار نیز به آنچه ساخته است می‌بالد و به هر کسی که بخواهد آن را فرو بریزد، نگاهی خصمانه دارد. هنگامی که آزمون شروع می‌شود، تلاشی ظریف ولی محرز برای خردکردن چیزی که مهندس نرم‌افزار ساخته است، صورت می‌پذیرد. از دیدگاه سازنده، آزمون را می‌توان (از نظر روان‌شناختی) فعالیتی ویران‌گر دانست. بنابراین وی با ملایمت عمل کرده آزمون‌هایی را طراحی و اجرا می‌کند که حاکی از کارکردن برنامه باشد، نه اینکه خطاها را کشف کند. متأسفانه خطاها وجود دارند و اگر مهندس نرم‌افزار آنها را نیابد، مشتری خواهد یافت!

غالباً بحث بالا به چند مورد سوءتفاهم منجر می‌شود: (۱) این که سازنده نرم‌افزار به هیچ وجه نباید آزمونی را انجام دهد؛ (۲) اینکه نرم‌افزار به شکست خواهد انجامید؛ (۳) اینکه آزمون‌گر فقط وقتی درگیر پروژه می‌شود که مراحل آزمون در حال آغاز است. هیچ یک از این جملات درست نیست.

سازنده نرم‌افزار همیشه مسؤول آزمون تک‌تکِ واحدها (مؤلفه‌های) برنامه بوده اطمینان حاصل می‌کند که هر کدام قادر به اجرای وظیفه‌ای است که جهت آن طراحی شده است. سازنده در بسیاری موارد، آزمون‌های انسجام را نیز اجرا می‌کند - یعنی آزمونی که منجر به ساخت (و آزمون) کامل ساختار برنامه می‌شود. فقط پس از آن‌که معماری نرم‌افزار کامل شد، یک گروه آزمون مستقل وارد کار می‌شود.

نقش *گروه آزمون مستقل* (ITG) برطرف کردن مشکلات ذاتی است که در واگذاری آزمون به شخص سازنده وجود دارد. آزمون مستقل، اختلاف سلیقه‌ها را برطرف می‌سازد. به هر حال، برای یافتن خطاها، به اعضای این گروه مستقل پول پرداخت می‌شود.

ولی چنین نیست که مهندس نرم‌افزار برنامه را به ITG بدهد و دنبال کار خود برود. سازنده و ITG در سرتاسر پروژه نرم‌افزاری رابطه کاری تنگاتنگی دارند تا اطمینان یابند که آزمون‌های کاملی اجرا خواهد شد. در حالی که آزمون انجام می‌شود، سازنده باید در دسترس باشد تا خطاهای پیدا شده را برطرف سازد.

از این جهت که ITG در فعالیت تعیین مشخصات وارد عمل می‌شود و در سرتاسر یک پروژه بزرگ خواهد ماند، بخشی از تیم پروژه توسعه نرم‌افزار به شمار می‌رود. ولی، در بسیاری موارد، ITG به سازمان تضمین کیفیت نرم‌افزار گزارش می‌دهد و در نتیجه قدری استقلال به دست می‌آورد که اگر بخشی از سازمان مهندسی نرم‌افزار می‌بود، این میزان استقلال امکان‌پذیر نبود.

نکته‌ی کلیدی

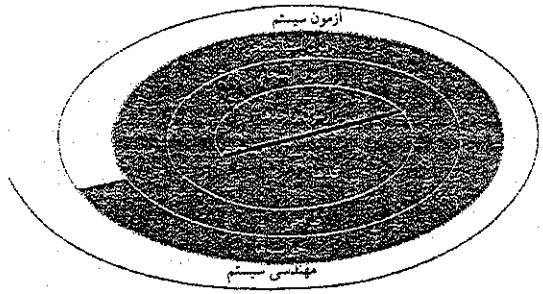
گروه مستقلی از آزمون‌ها «تضاد علائقی» را که سازندگان نرم‌افزار ممکن است تجربه کنند، ندارد.

«نخستین اشتباهی که آدم‌ها مرتکب می‌شوند، این است که فکر کنند فقط تیم آزمون‌گر مسؤول تضمین کیفیت است.»

برایان ماریک

۳-۱-۱۷ راهبردی برای آزمون نرم‌افزار

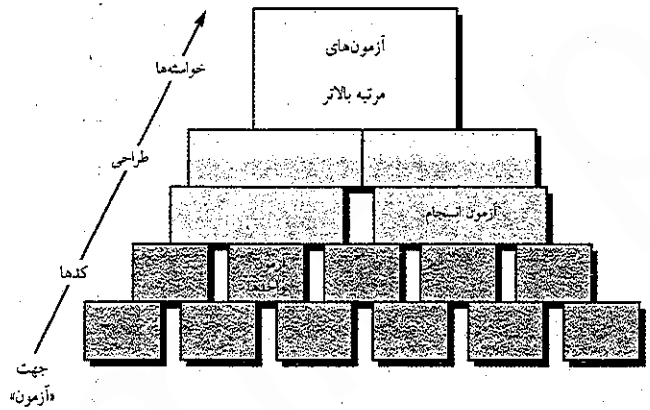
فرایند مهندسی نرم‌افزار را می‌توان به صورت مارپیچ شکل ۱-۱۷ در نظر گرفت. در ابتدا، مهندسی سیستم یا مهندسی اطلاعات، نقش نرم‌افزار را تعیین می‌کند و به تحلیل خواسته‌های نرم‌افزار می‌انجامد، که در آن دامنه اطلاعاتی، عملکرد، رفتار، کارایی، شرایط حدی و ملاک‌های اعتبارسنجی مربوط به نرم‌افزار برقرار می‌شوند. با حرکت به طرف داخل مارپیچ، به طراحی و سرانجام کدنویسی می‌رسیم. برای توسعه نرم‌افزار کامپیوتری، به طرف داخل مارپیچ حرکت می‌کنیم تا در هر دور، از سطح انتزاع کاسته شود.



شکل ۱-۱۷ راهبرد آزمون.

راهبرد کلی برای آزمون نرم‌افزار چیست؟

یک راهبرد برای آزمون نرم‌افزار را می‌توان در حیطه‌ی این مارپیچ در نظر گرفت (شکل ۱-۱۷). *آزمون واحدها* در مرکز مارپیچ آغاز می‌شود و بر هر واحد (یعنی مؤلفه) از نرم‌افزار که در کد منبع پیاده‌سازی می‌شود، تمرکز دارد. آزمون با حرکت به طرف بیرون مارپیچ ادامه یافته به *آزمون انسجام* می‌رسد، که در آن به طراحی و ایجاد معماری نرم‌افزار تأکید می‌شود. با یک دور دیگرِ پیش‌رفتن، به آزمون *اعتبارسنجی* می‌رسیم که در آن خواسته‌های تثبیت شده به‌عنوان بخشی از تحلیل خواسته‌های نرم‌افزار، در مقابل نرم‌افزار ساخته شده اعتبارسنجی می‌شوند. سرانجام، به *آزمون سیستم* می‌رسیم که در آن نرم‌افزار و دیگر عناصر سیستمی به صورت یک کل آزمون می‌شوند. برای آزمون نرم‌افزار کامپیوتری، به بیرون مارپیچ حرکت می‌کنیم تا دامنه آزمون در هر دور وسعت پیدا کند.



شکل ۲-۱۷ مراحل آزمون نرم‌افزار.

مرجع وب

منابع مفیدی برای آزمون‌گران نرم‌افزار را می‌توانید در www.SQAtester.com بیابید.

با در نظر گرفتن فرایند از دیدگاه رویه‌ای، آزمون در حیطه مهندسی نرم‌افزار، چهار مرحله است که به‌طور ترتیبی پیاده‌سازی می‌شوند. این مراحل در شکل ۲-۱۷ نشان داده شده‌اند. در آغاز، آزمون‌ها بر تک‌تک مؤلفه‌ها تأکید دارند تا اطمینان حاصل شودکه هریک به‌طور منفرد درست عمل‌می‌کند. ازاین‌رو آن را *آزمون واحدها* نام نهاده‌اند. آزمون واحدها از تکنیک‌های جعبه سفید استفاده زیادی به‌عمل

می‌آورد و مسیرهای مشخصی از ساختار کنترلی پیمانه‌ها را امتحان می‌کند تا از پوشش‌دهی کامل و حداکثر تشخیص خطاها اطمینان حاصل شود. سپس باید مؤلفه‌ها را مونتاژ یا مجتمع کرد تا بسته‌ی نرم‌افزاری کامل تشکیل شود. *آزمون انسجام* با مسائل مربوط به مشکلات دوگانه‌ی وارسی و ساخت برنامه سروکار دارد. در هنگام انسجام، تکنیک‌های طراحی آزمون‌های جعبه سیاه بیشترین کاربرد را دارند. به تعداد محدودی از آزمون‌های جعبه سفید نیاز است تا اطمینان حاصل شود که مسیرهای کنترلی اصلی تحت پوشش قرار گرفتند. پس از آنکه نرم‌افزار انسجام یافت (ساخته شد) مجموعه‌ای از *آزمون‌های سطح بالا* اجرا می‌شود. *آزمون اعتبارسنجی* برای حصول اطمینان نهایی از رعایت همه‌ی خواسته‌های رفتاری، عملیاتی و کارایی، صورت می‌پذیرد. در این آزمون‌ها انحصاراً از آزمون‌های جعبه سیاه استفاده می‌شود.

آخرین مرحله آزمون سطح بالا، خارج از مرز مهندسی نرم‌افزار و در حیطه‌ای وسیع‌تر، یعنی مهندسی سیستم کامپیوتری قرار می‌گیرد. هنگامی که نرم‌افزار اعتبارسنجی شد، باید با عناصر سیستمی دیگر (مثل سخت‌افزار، افراد و بانک‌های اطلاعاتی) تلفیق شود. *آزمون سیستم* است که تصدیق می‌کند همه‌ی عناصر به‌درستی عمل می‌کنند و کارایی/عملکرد کلی سیستم قابل حصول است.

۱۷-۱-۴ ملاک‌هایی برای کامل کردن آزمون

هر بار که بحثی از آزمون به میان می‌آید، یک پرسش کلاسیک مطرح می‌شود: «آزمون چه هنگام به پایان می‌رسد - یعنی چگونه بدانیم که به قدر کافی آزمون انجام داده‌ایم؟» متأسفانه، هیچ پاسخ مشخصی برای این پرسش وجود ندارد، ولی چند پاسخ واقع‌گرایانه در این خصوص وجود دارد.

یک پاسخ به این پرسش این است: «آزمون هیچ‌گاه تمامی ندارد، فقط بار مسؤولیت از شما (مهندس نرم‌افزار) به مشتری محول می‌شود.» هر بار که مشتری/کاربر یک برنامه کامپیوتری را اجرا می‌کند، برنامه روی مجموعه جدیدی از داده‌ها آزمون می‌شود. این واقعیت تأمل برانگیز، اهمیت فعالیت‌های دیگر تضمین کیفیت را نمایان می‌سازد. یک پاسخ دیگر (قدری بدبینانه ولی صحیح) عبارت است از «آزمون وقتی پایان می‌یابد که زمان یا پول شما تمام شود».

گرچه معدودی از دست اندرکاران با همین پاسخ‌ها استدلال می‌کنند، برای تعیین این که چه هنگام آزمون کافی انجام شده است، به ملاک‌های محکم‌تری نیاز دارید. در رویکرد مهندسی نرم‌افزار اتاق تمیز (فصل ۲۱)، تکنیک‌های کاربرد آماری [Kel00] پیشنهاد می‌شوند که یک سری آزمون‌های به‌دست‌آمده از یک نمونه آماری کلیه‌ی اجراهای ممکن برنامه توسط تمامی کاربران یک جمعیت هدف را به‌انجام می‌رسانند. سایرین [Sin99] استفاده از مدل‌سازی آماری و نظریه قابلیت اطمینان را برای پیش‌بینی کامل بودن آزمون مطرح می‌کنند.

با جمع‌آوری معیارها در زمان آزمون نرم‌افزار و استفاده از مدل‌های موجود جهت قابلیت اطمینان نرم‌افزار، می‌توان دستورالعمل‌های معنی‌داری جهت پاسخ به این پرسش یافت که آزمون کی به پایان می‌رسد. در این مورد تردیدی وجود ندارد که پیش از وضع قواعدی کمّی برای آزمون، باز هم کارهایی باقی می‌ماند که باید انجام داد، ولی روش‌های تجربی که در حال حاضر وجود دارند به مراتب بهتر از روش‌های شهودی هستند.

آزمون چه هنگام تمام می‌شود؟

مرجع وب

واژگان کاملی از اصطلاحات آزمون را در آدرس زیر می‌یابید:
www.testingstandards.co.uk/living_glossary.htm

SafeHome

آماده‌سازی برای آزمون

صحنه: دفتر داگ میلر، همچنان که طراحی و ساخت مؤلفه‌های معین آغاز می‌شود.

نقش‌آفرینان: داگ میلر، مدیر مهندسی نرم‌افزار، وینود، جیمی، اد و شکیرا- اعضای تیم مهندسی نرم‌افزار *SafeHome*.

گفتگو:

داگ: به نظرم می‌رسد که وقت زیادی صرف حرف زدن درباره آزمون نکردیم.

وینود: درست، ولی همه‌ی ما قدری مشغول بودیم. و به علاوه، در فکرش بوده‌ایم- در واقع بیشتر از در فکر بودن.

داگ (با لبخند): می‌دانم... همه‌ی ما زیادی مشغول بودیم، ولی هنوز باید تا ته خط برویم.

شکیرا: من این ایده را دوست دارم که آزمون‌های واقعی را قبل از شروع کدنویسی برای هر کدام از مؤلفه‌ها طراحی کنیم. با این کار، وقتی کدهای مؤلفه‌ها نوشته شدند، یک فایل کاملاً بزرگ از آزمون‌ها را دارم.

داگ: این یکی از مفاهیم برنامه‌نویسی حدی [یک فرایند توسعه نرم‌افزار چابک- فصل ۳] است، نه؟

اد: چرا، ما از خود برنامه‌نویسی حدی استفاده نکردیم، ولی به این نتیجه رسیدیم که طراحی آزمون‌های واحد قبل از ساخت مؤلفه ایده خوبی است- طراحی، همه‌ی اطلاعات لازم را در اختیارمان می‌گذارد.

جیمی: من هم همین کار را کردم.

وینود: و من هم نقش انسجام دهنده را بر عهده گرفتم تا هر زمان که یکی از بچه‌ها مؤلفه‌ای را به من تحویل داد، آن را منسجم کنم و یک سری آزمون‌های رگرسیون روی برنامه‌ای انجام بدهم که بخشی از آن کامل شده است. من کار کرده‌ام تا یک مجموعه آزمون مناسب برای هر کدام از قابلیت‌های عملیاتی موجود در سیستم طراحی کنم.

داگ (به وینود): آزمون‌ها را هر چند وقت یک بار اجرا می‌کنی؟

وینود: هر روز... تا این که سیستم به انسجام برسد... خب، منظورم این است تا وقتی که گام نرم‌افزاری که خیال داریم تحویل دهیم، منسجم شود.

داگ: شما بچه‌ها از من جلوترید!

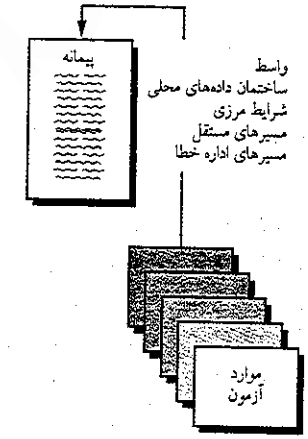
وینود (با خنده): در تجارت نرم‌افزار، پیش‌دستی همه چیز است رییس.

۱۷-۲ مسائل راهبردی

بعداً در همین فصل یک راهبرد سیستماتیک برای آزمون نرم‌افزار را مورد بررسی قرار خواهیم داد. ولی اگر یک سری مسائل جانبی رعایت نشود، حتی بهترین راهبردها نیز محکوم به شکست خواهد بود. تام گیلب [Gil95] استدلال می‌کند که اگر قرار است راهبرد آزمون نرم‌افزار موفق شود، نکات زیر باید رعایت شوند:

خواسته‌ها را به شیوه‌ای کمیت‌پذیر، مدت‌ها قبل از شروع آزمون مشخص کنید. گرچه هدف اصلی آزمون، یافتن خطاهاست، یک راهبرد آزمون خوب، ویژگی‌های کیفیتی دیگر نظیر قابلیت حمل، قابلیت نگهداری، و قابلیت استفاده (فصل ۱۴) را نیز مورد سنجش قرار می‌دهد. این موارد را باید به شیوه‌ای مشخص کرد که قابل اندازه‌گیری و نتایج آزمون عاری از ابهام باشند.

اهداف و آزمون را به وضوح بیان کنید. اهداف مشخص آزمون باید به‌صورتی قابل اندازه‌گیری بیان شوند. مثلاً، اثربخشی آزمون، پوشش‌دهی آزمون، میانگین زمان شکست، هزینه یافتن و برطرف‌ساختن نقایص، چگالی نقایص باقیمانده یا فراوانی وقوع و تعداد ساعت‌های کار به ازای هر آزمون رگرسیون، همگی باید در طرح آزمون بیان شوند [Gil95].



شکل ۳-۱۷ شدت شکست به‌عنوان تابعی از زمان اجرا.

کاربران نرم‌افزار را بشناسید و برای هر گروه از کاربران یک سابقه تهیه کنید. موارد آزمونی که سناریوی تعامل را برای هر طبقه از کاربران توصیف می‌کنند، می‌توانند با تأکید بر کاربرد واقعی محصول، از کل کار لازم برای آزمون بکاهند.

طرح آزمونی تهیه کنید که بر «آزمون چرخه سریع» تأکید داشته باشد. گیلب [Gil95] توصیه می‌کند که تیم مهندسی نرم‌افزار چرخه‌های سریع آزمون را انجام دهند (۲ درصد از کار پروژه). برای کنترل سطوح کیفیت و راهبردهای آزمون متناظر می‌توانید بازخوردهای ناشی از این چرخه‌های آزمون سریع را به‌کار بگیرید.

نرم‌افزاری پرقدرت بسازید که برای آزمودن خودش طراحی شده باشد. نرم‌افزار باید به شیوه‌ای طراحی شود که از تکنیک‌های ضداشکال (بخش ۱-۳-۱۷) استفاده کند. یعنی، نرم‌افزار باید قادر به یافتن طبقات معینی از خطاها باشد. علاوه بر این، طراحی باید آزمون خودکار و رگرسیون را انجام دهد.

از بازبینی‌های فنی رسمی اثربخش به‌عنوان فیلتر قبل از آزمون استفاده کنید. بازبینی‌های فنی رسمی (فصل ۸) می‌توانند به اندازه خود آزمون در کشف خطاها مؤثر واقع شوند. به همین دلیل، بازبینی‌ها می‌توانند مقدار کار لازم برای آزمون را کاهش داده نرم‌افزاری با کیفیت بالا تولید کنند.

کدام دستورالعمل‌ها به یک راهبرد آزمون نرم‌افزار موفق می‌انجامند؟

مرجع وب

فهرستی عالی از منابع آزمون را می‌توان در وب‌سایت زیر یافت:
www.io.com/~wazmo/qa

«آزمون نرم‌افزار تنها بر اساس خواسته‌های کاربر نهایی همانند وارسی یک ساختمان بر اساس کارهای انجام‌شده توسط طراحی داخلی و نادیده گرفتن پی‌ریزی، اسکلت‌بندی و لوله‌کشی‌هاست»
بوریس بیزر

یک روش بهسازی پیوسته برای فرایند آزمون توسعه دهید. راهبرد آزمون باید اندازه‌گیری شود. معیارهای جمع‌آوری شده طی آزمون را باید به‌عنوان بخشی از روش کنترل فرایند برای آزمون نرم‌افزار به‌کار برد.

۱۷-۳ راهبردهای آزمون برای نرم‌افزارهای سنتی[1]

راهبردهای فراوانی وجود دارد که در آزمون نرم‌افزار می‌توان به کاربرد. در یک سوی طیف، می‌توانید آن‌قدر صبر کنید که سیستم به‌طور کامل ایجاد شود و سپس آزمون‌ها را روی سیستم کامل شده اجرا کنید، به این امید که خطاها پیدا شوند. این رویکرد گرچه جالب به نظر می‌رسد، خیلی ساده جواب نمی‌دهد و به نرم‌افزار اشکال‌داری منجر می‌شود که همه‌ی طرف‌های ذی‌نفع را ناامید می‌سازد. در سوی دیگر طیف، می‌توانید آزمون‌ها را به‌صورت روزمره و هرگاه که بخشی از سیستم ساخته می‌شود، اجرا کنید. این رویکرد، گرچه برای خیلی‌ها جالب به‌نظر نمی‌رسد، می‌تواند بسیار مفید واقع شود. متأسفانه، برخی سازندگان در به‌کارگیری آن تردید می‌کنند. چه باید کرد؟

یک راهبرد آزمون که اکثر تیم‌های نرم‌افزار از آن استفاده می‌کنند، جایی در میانه‌ی این طیف قرار می‌گیرد. یعنی به آزمون، به‌صورت گام به گام نگاه می‌شود، به‌طوری که با آزمون تک تک واحدهای برنامه شروع می‌شود، به سمت آزمون‌های طراحی شده برای تسهیل انسجام بخشیدن به واحدها حرکت می‌کند و با آزمون‌هایی که سیستم ایجادشده را تمرین می‌دهند، به اوج می‌رسد. هر کدام از این آزمون‌ها در بخش‌های بعدی شرح داده خواهد شد.

۱۷-۳-۱ آزمون واحدها (Unit Testing)

آزمون واحدها تلاش‌های وارسی مربوط به کوچکترین واحد طراحی نرم‌افزار – یعنی مؤلفه‌ها و پیمانه‌ها – را کانون توجه قرار می‌دهد. با به‌کارگیری توصیف طراحی در سطح مؤلفه‌ها به‌عنوان راهنما، مسیرهای کنترلی مهم، آزمون می‌شوند تا خطاهای موجود در داخل مرزهای پیمانه برملا شوند. پیچیدگی نسبی آزمون‌ها و خطاهایی که این آزمون‌ها برملا می‌سازند توسط قیدوبندی محدود می‌شود که در دامنه‌ی تعیین شده برای آزمون واحدها مشخص می‌شود. این نوع آزمون را می‌توان به‌طور موازی برای چند مؤلفه انجام داد.

ملاحظات مربوط به آزمون واحدها. آزمون‌هایی که به‌عنوان بخشی از آزمون واحدها انجام می‌شوند، در شکل ۳-۱۷ نشان داده شده است. واسط پیمانه‌ها آزمون می‌شود تا اطمینان حاصل شود که اطلاعات به‌طور مناسب به درون و بیرون واحد مورد آزمون، جریان می‌یابند. ساختمان داده‌های محلی بررسی می‌شوند تا اطمینان حاصل شود که پیمانه به‌طور مناسب در مرزهای تعیین شده برای محدود کردن پردازش عمل می‌کند. همه‌ی مسیرهای مستقل (مسیرهای پایه) که از ساختار کنترلی عبور می‌کنند، امتحان می‌شوند تا اطمینان حاصل شود که همه‌ی دستورهای موجود در یک پیمانه حداقل یک بار اجرا شده‌اند و سرانجام همه‌ی مسیرهای کنترل خطا آزمون می‌شوند.

اندرز

فکر بدی نیست که موارد آزمون واحد را قبل از نوشتن کد برای یک مؤلفه طراحی کنید. به این ترتیب، اطمینان حاصل می‌کنید که کدهای نوشته شده آزمون را با موفقیت پشت سر می‌گذارند.

[1] در سرتاسر این کتاب از عبارت نرم‌افزار سنتی برای اشاره به معماری‌های سلسله مراتبی یا فراخوانی و بازگشت استفاده خواهیم کرد که به وفور در انواع دامنه‌های کاربردی مشاهده می‌شوند. معماری‌های نرم‌افزار سنتی، شیءگرا نیستند و شامل برنامه‌های تحت وب هم نمی‌شوند.

جریان داده‌هایی که از واسط یک پیمانه عبور می‌کند، باید پیش از شروع هر آزمون دیگری آزموده شود. اگر داده‌ها به‌طور مناسب وارد یا خارج نشوند، همه‌ی آزمون‌های دیگر بیهوده خواهند بود. به‌علاوه، ساختمان داده‌های محلی باید امتحان شوند و تأثیر محلی بر داده‌های سرتاسری باید (در صورت امکان) طی آزمون واحد مورد ارزیابی قرار گیرد.

در حین آزمون واحدها، آزمون انتخابی مسیرهای اجرا اهمیت ویژه‌ای دارد. موارد آزمون باید طوری طراحی شوند که خطاهای ناشی از محاسبات غلط، مقایسه‌های نادرست، یا جریان کنترل نامناسب را کشف کنند. آزمون‌های مسیرهای پایه و حلقه‌ها، تکنیک‌هایی اثربخش برای کشف آرایه وسیعی از خطاهای مسیر است.

آزمون مرزی، آخرین (و احتمالاً مهمترین) وظیفه در مرحله آزمون واحدها است. نرم‌افزارها معمولاً در مرزها به شکست می‌انجامند. یعنی خطاها غالباً زمانی رخ می‌دهند که n-اُمین عنصر از یک آرایه n-بعدی پردازش می‌شود؛ تکرار i-اُم از حلقه‌ای با i مرتبه گذر فرا می‌رسد، یا حداقل و حداکثر یک مقدار مجاز پردازش می‌شود. موارد آزمونی که ساختمان داده‌ها، جریان کنترلی و مقادیر داده‌ها را درست در مقادیر کمتر، مساوی و بیشتر از بیشینه و کمینه امتحان می‌کنند، احتمال زیادی برای کشف خطاها دارند.

طراحی خوب، حکم می‌کند که شرایط خطا پیش‌بینی شود و مسیرهایی برای اداره خطا مشخص شود که در صورت بروز خطا پردازش را درباره جهت‌دهی کند یا به آن پایان دهد. یوردون [You75] این روش را *خداشکال‌سازی* می‌نامد. متأسفانه، معمولاً کنترل خطا را در نرم‌افزار می‌گنجانند، ولی به آزمون آن نمی‌پردازند. شاید این داستان واقعی به روشن شدن مطلب کمک کند:

> یک سیستم طراحی تعاملی طبق قرارداد توسعه یافت. در یک پیمانه پردازش تراکنش، یکی از سازندگان شوخ‌طبع، پیام اداره خطای زیر را پس از تعدادی آزمون‌های شرطی که حاوی شاخه‌های متعددی از جریان کنترل داشت، قرار داده بود: *خطا! برای ورود به این قسمت راهی وجود ندارد!* این پیام خطا را یک مشتری هنگام آموزش کاربران کشف کرده بود!

از میان خطاهای متداول‌تر در محاسبات، می‌توان به موارد زیر اشاره نمود: (۱) تقدم محاسباتی نادرست، یا آنهایی که به‌درستی درک نشده‌اند؛ (۲) گوناگونی عملیات؛ (۳) مقداردهی اولیه نادرست؛ (۴) دقت نادرست؛ (۵) ارائه نمادهای نادرست برای یک رابطه یا عبارت.

رویه‌های آزمون واحدها. آزمون واحدها معمولاً به‌عنوان الحاقی بر مرحله کدنویسی در نظر گرفته می‌شود. طراحی آزمون‌های واحدها ممکن است قبل از شروع کدنویسی یا پس از تولید کدهای منبع رخ دهد. با مروری بر اطلاعات طراحی، راهنمایی برای تعیین موارد آزمونی را فراهم می‌آورد که احتمالاً خطاهای موجود در هر کدام از گروه‌های بحث شده‌ی قبلی را برملا نکرده‌اند. هر مورد آزمون باید با مجموعه‌ای از نتایج قابل انتظار همراه شود.

چون مؤلفه یک برنامه‌ی مستقل و قائم‌به‌ذات نیست، برای آزمون هر واحد باید یک نرم‌افزار درایور و/یا stub توسعه یابد. محیط آزمون واحدها در شکل ۴-۱۷ نشان داده شده است. در اکثر کاربردها، درایور همان برنامه اصلی است که داده‌های مورد آزمون را پذیرفته این داده‌ها را به مؤلفه (مؤلفه‌ای که باید آزمون شود) تحویل داده، نتایج مربوطه را چاپ می‌کنند. stubها جایگزین پیمانه‌هایی می‌شوند که توسط مؤلفه مورد آزمون فراخوانی می‌شوند. stub از واسط پیمانه فراخوانده شده استفاده می‌کند، حداقل دستکاری داده‌ها را انجام می‌دهد و کنترل را به پیمانه‌ای که در حال آزمون است، بازمی‌گرداند.

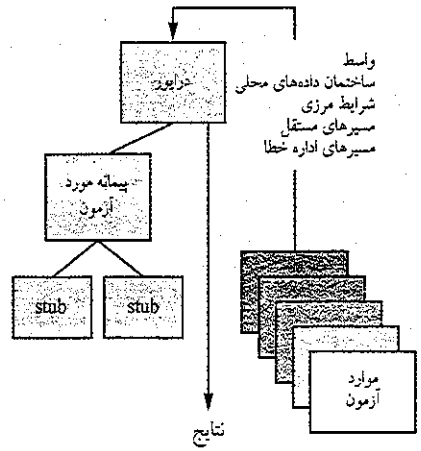
کدام خطاها معمولاً طی آزمون واحد کشف می‌شوند؟

مرجع وب

درباره انواع مقالات و منابع مربوط به «آزمون چابک» در نشانی testing.com/agile اطلاعات مفیدی خواهید یافت.

اندرز

حتماً آزمون‌هایی طراحی کنید که هر مسیر اداره‌کننده‌ی خطایی را اجرا کند. در غیر این صورت، این مسیر ممکن است هنگام فراخوانی به شکست بینجامد و وضعیتی خطرناک به بار آورد.



شکل ۴-۱۷ محیط آزمون واحد.

درایورها و stubها ایجاد سربار (overhead) می‌کنند. یعنی هر دو نرم‌افزارهایی هستند که باید نوشته شوند (طراحی رسمی معمولاً اجرا نمی‌شود) ولی با محصول نهایی تحویل نمی‌شوند. اگر درایورها و stubها ساده نگه داشته شوند، سربار واقعی نسبتاً کم است. متأسفانه، بسیاری از مؤلفه‌ها را نمی‌توان به‌طور مناسب با نرم‌افزارهای سربار ساده آزمون کرد. در بسیاری موارد، آزمون کامل را می‌توان تا مرحله آزمون انسجام به تعویق انداخت (که در آن نیز از درایورها یا stubها استفاده می‌شود).

آزمون واحدها هنگامی ساده می‌شود که مؤلفه‌ای با انسجام بالا طراحی شود. هنگامی که فقط یک عملکرد بر عهده مؤلفه‌ای قرار داده شود، تعداد موارد آزمون کاهش می‌یابد و خطاها را می‌توان به سهولت پیش‌بینی و کشف کرد.

اندرز

شرایطی وجود دارد که در آن‌ها منابع لازم برای انجام آزمون‌های واحد جامع و فراگیر را در اختیار ندارید. پیمانه‌های پیچیده و بحرانی را برگزینید و تنها آن‌ها را آزمون کنید.

۱۷-۳-۲ آزمون انسجام (Integration Testing)

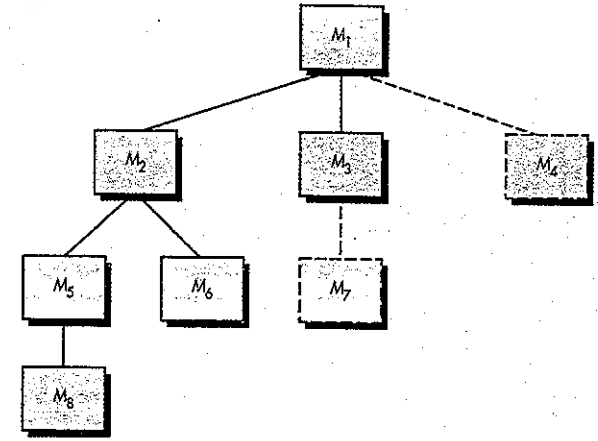
یک فرد مبتدی در جهان نرم‌افزار ممکن است پس از انجام آزمون واحدها روی همه‌ی پیمانه‌ها این پرسش را مطرح کند: «اگر همه‌ی پیمانه‌ها به‌تنهایی کار می‌کنند، چه دلیلی دارد که در کنار هم کار نکنند؟» البته در کنار هم کار کردن، مسأله‌ی ایجاد رابطه میان آنها را مطرح می‌سازد. ممکن است داده‌ها در گذر از یک واسط از بین بروند؛ یک پیمانه می‌تواند اثری وارونه بر دیگری داشته باشد، عملکردهای فرعی پس از ترکیب، ممکن است عملکرد اصلی مطلوب را نتیجه ندهند؛ نشانه‌های قابل قبول در موارد انفرادی، ممکن است تا سطوح غیر قابل قبول تشدید شود؛ ساختمان داده‌های سرتاسری ممکن است باعث ایجاد مشکلات شود و مواردی دیگر.

آزمون انسجام، تکنیکی سیستماتیک برای ایجاد ساختار برنامه و در عین حال اجرای آزمون‌هایی جهت کشف خطاها در ایجاد واسط‌هاست. هدف، آزمون مؤلفه‌ها و ساخت برنامه‌ای مطابق با طراحی است.

غالباً انسجام مؤلفه‌ها به شیوه غیر تدریجی انجام می‌شود؛ یعنی ایجاد برنامه با استفاده از روش «انفجار بزرگ». همه‌ی مؤلفه‌ها از قبل ترکیب می‌شوند و کل برنامه یکجا آزمون می‌شود و معمولاً بی‌نظمی ایجاد می‌شود! مجموعه‌ای از خطاها مشاهده می‌شود. تصحیح دشوار است، زیرا جداسازی علت‌ها در کل برنامه امری پیچیده است. هنگامی که این خطاها تصحیح شدند، خطاهای جدید ظاهر می‌شوند و فرایند در یک دور ظاهراً بی‌پایان ادامه می‌یابد.

انسجام تدریجی، آنتی‌تز روش انفجار بزرگ است. برنامه تدریجاً ایجاد می‌شود و مورد آزمون قرار می‌گیرد و در نتیجه جداکردن خطاها و تصحیح آنها آسان‌تر است؛ احتمال این که واسطها به‌طور کامل مورد آزمون قرار گیرند، افزایش می‌یابد و یک روش آزمون سیستماتیک را می‌توان به‌کار برد. در بخش‌های بعدی چند راهبرد انسجام تدریجی مورد بحث قرار می‌گیرند.

انسجام بالا به پایین. *آزمون انسجام بالا* به پایین یک روش تدریجی برای ایجاد ساختار برنامه است. پیمانه‌ها با حرکت به طرف پایین در سلسله مراتب کنترلی و با شروع از پیمانه کنترل اصلی (برنامه اصلی) مجتمع می‌شوند. پیمانه‌های زیردست (subordinate) پیمانه‌ی کنترل اصلی به شیوه عمقی یا عرضی در ساختار قرار داده می‌شود.



شکل ۵-۱۷ انسجام پایین به بالا.

در شکل ۵-۱۷، *انسجام عمقی*، همه‌ی مؤلفه‌ها را در یک مسیر کنترلی اصلی از ساختار، مجتمع می‌کند. گزینش یک مسیر اصلی تا حدی اختیاری است و به ویژگی‌های کاربرد بستگی دارد. برای مثال، با گزینش مسیر سمت چپ، ابتدا مؤلفه‌های M_1، M_2 و M_5 مجتمع می‌شوند. سپس M_8 یا (در صورت درست کارکردن M_5) M_6 مجتمع می‌شوند. سپس، مسیرهای کنترل سمت راست و مرکزی ساخته می‌شوند. در انسجام عرضی همه‌ی مؤلفه‌هایی که مستقیماً زیردست هر سطح هستند، با حرکت افقی در ساختار، به یکدیگر ملحق می‌شوند. در شکل، ابتدا مؤلفه‌های M_2، M_3 و M_4 به هم ملحق می‌شوند. سطح کنترل بعدی، M_5، M_6 و غیره است. فرایند انسجام در مراحل پنج گانه اجرا می‌شود:

۱. پیمانه کنترل اصلی به‌عنوان درایور آزمون به‌کار می‌رود و stubها جایگزین کلیه مؤلفه‌هایی می‌شود که مستقیماً زیردست پیمانه کنترلی اصلی است.

اندرز

انتخاب رویکرد «انفجار بزرگ» برای انسجام بخشی، راهبردی برای افراد تنبل است که محکوم به شکست است. کار انسجام بخشی را به‌صورت گام به گام انجام دهید و همچنان که پیش می‌روید، آزمون‌ها را انجام دهید.

اندرز

هنگامی که پروژه را زمان‌بندی می‌کنید، باید شیوه‌ی انسجام را در نظر بگیرید به‌طوری که مؤلفه‌ها در صورت نیاز در دسترس باشند.

مراحل انسجام از بالا به پایین کدام‌اند؟

۲. بسته به روش انسجام انتخاب شده (یعنی عمقی یا عرضی)، stubهای زیردست، به نوبت، جای خود را به مؤلفه‌های واقعی می‌دهند.

۳. همزمان با انسجام هر مؤلفه، آزمون صورت می‌گیرد.

۴. با کامل شدن هر مجموعه از آزمون‌ها، یک stub دیگر جای خود را به مؤلفه واقعی می‌دهد.

۵. آزمون رگرسیون (که بعداً در همین بخش بحث خواهدشد) را می‌توان اجرا کرد تا مطمئن شد که خطاهای جدیدی وارد نشده‌اند.

فرایند از مرحله ۲ آنقدر ادامه می‌یابد که کل ساختار برنامه ساخته شود.

راهبرد انسجام بالا به پایین، نقاط کنترلی و تصمیم‌گیری اصلی را در ابتدای فرایند آزمون، وارسی می‌کند. در یک ساختار برنامه با «فاکتوربندی خوب»، تصمیم‌گیری در سطوح بالاتر سلسله مراتب رخ می‌دهد و بنابراین، ابتدا به آنها بر می‌خوریم. اگر مشکلات کنترلی عمده‌ای وجود داشته باشند، تشخیص زود هنگام، ضروری است. اگر انسجام عمقی انتخاب شود، ممکن است قابلیت عملیاتی کامل نرم‌افزار پیاده‌سازی شود و به نمایش درآید. نمایش اولیه قابلیت عملیاتی، به سازنده و مشتری قوت قلب می‌دهد.

هنگام انتخاب انسجام از بالا به پایین، به چه مسائلی ممکن است برخورد کنید؟

راهبرد بالا به پایین چندان پیچیده به‌نظر نمی‌رسد، ولی در عمل، ممکن است مشکلات لوجستیکی پیش آید. متداول‌ترین این مشکلات، هنگامی رخ می‌دهد که برای آزمون سطوح بالاتر نیاز به پردازش در سطوح پایین‌تر باشد. stubها جایگزین پیمانه‌های سطح پایین می‌شوند؛ از این رو، داده‌ها در ساختار برنامه چندان به طرف بالا جریان پیدا نمی‌کنند. سه انتخاب پیش روی آزمون‌گر خواهد بود: (۱) به‌تأخیرانداختن بسیاری از آزمون‌ها تا اینکه stubها جای خود را به پیمانه‌های واقعی بدهند؛ (۲) توسعه stubهایی که با اجرای عملیاتی محدود، پیمانه واقعی را شبیه‌سازی می‌کنند یا (۳) انسجام نرم‌افزار از پایین سلسله مراتب به طرف بالا.

اولین رویکرد (به تأخیر انداختن آزمون‌ها تا زمانی که stubها جای خود را به پیمانه‌های واقعی بدهند) باعث می‌شود تا ارتباط بین بعضی از آزمون‌ها و ملحق کردن بعضی از پیمانه‌ها تحت کنترل نباشد. این امر می‌تواند منجر به دشوار شدن تعیین علت خطاها شود و از ماهیت مقید روش بالا به پایین عدول شود. روش دوم عملی است، ولی ممکن است سربار چشمگیری را طلب کند، زیرا stubها پیچیده و پیچیده‌تر می‌شوند. روش سوم، که آزمون پایین به بالا نیز خوانده می‌شود، در بندهای بعدی بحث خواهد شد.

انسجام پایین به بالا. *آزمون انسجام پایین به بالا*، چنان‌که از نامش پیدا است، ساخت و آزمون پیمانه‌ها را با *پیمانه‌های ساده* آغاز می‌کند (یعنی مؤلفه‌هایی که در پایین‌ترین سطح از سلسله مراتب ساختار برنامه قرار دارند). از آنجا که مؤلفه‌ها از پایین به بالا به هم ملحق می‌شوند، پردازش لازم برای مؤلفه‌های زیردست یک سطح مفروض، همواره در دسترس بوده نیازی به حذف stubها نیست. راهبرد انسجام پایین به بالا را می‌توان طی مراحل زیر پیاده‌سازی کرد:

۱. مؤلفه‌های سطح پایین به صورت خوشه‌هایی با هم ترکیب می‌شوند که یک عملکرد فرعی از نرم‌افزار را انجام می‌دهند.

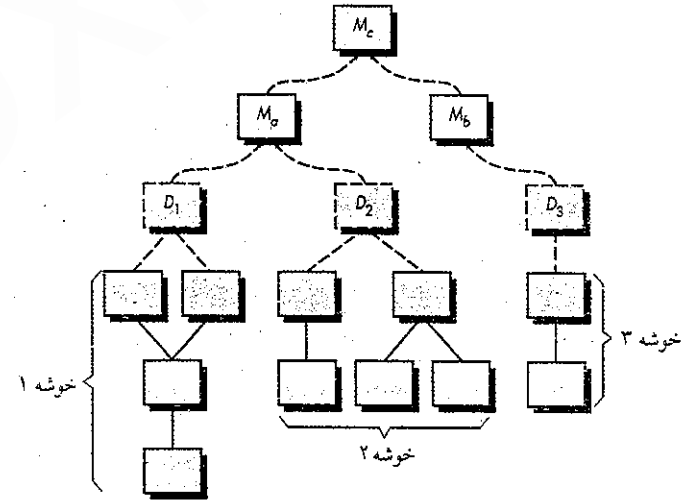
۲. یک درایور (برنامه کنترلی برای آزمون) نوشته می‌شود تا ورودی و خروجی موارد آزمون را هماهنگ کند.

مراحل انسجام از پایین به بالا کدام‌اند؟

۳. خوشه مورد آزمون قرار می‌گیرد.

۴. درایورها حذف می‌شوند و خوشه‌ها در حرکت به طرف بالای ساختار برنامه با یکدیگر ترکیب می‌شوند.

انسجام از الگوی نشان داده شده در شکل ۶-۱۷ پیروی می‌کند. مؤلفه‌ها با یکدیگر ترکیب می‌شوند تا خوشه‌های ۱، ۲ و ۳ را تشکیل دهند. هر یک از خوشه‌ها با استفاده از یک درایور (که به صورت بلوک نقطه‌چین نشان داده شده است)، مورد آزمون قرار می‌گیرد. مؤلفه‌های موجود در خوشه‌های ۱ و ۲ زیردست M_a هستند. درایورهای D_1 و D_2 حذف می‌شوند و بین خوشه‌ها و M_a واسطی ایجاد می‌گردد. به‌طور مشابه، درایور D_3 برای خوشه‌ی ۳ پیش از انسجام با پیمانه M_b حذف می‌شود. M_a و M_b با مؤلفه M_c مجتمع می‌شوند و غیره.



شکل ۶-۱۷ انسجام بالا به پایین.

با حرکت به طرف بالا، نیاز به درایورهای آزمون جداگانه کمتر می‌شود. در حقیقت، اگر دو سطح بالایی ساختار برنامه از بالا به پایین به هم ملحق شوند، تعداد درایورها را می‌توان تا حد چشمگیری کاهش داد و الحاق خوشه‌ها را به مقدار زیاد ساده‌تر کرد.

آزمون رگرسیون. هر بار که یک پیمانه جدید به‌عنوان بخشی از آزمون انسجام افزوده می‌شود، نرم‌افزار تغییر می‌کند. مسیرهای جریان داده‌ای جدیدی برقرار می‌شوند، I/Oهای جدید ممکن است رخ دهند، و باید به منطق کنترلی جدیدی روی آورده شود. این تغییرات ممکن است باعث مشکلات در عملکردهایی بشود که قبلاً بدون نقص کار می‌کرده‌اند. در زمینه‌ی راهبرد آزمون انسجام، آزمون رگرسیون عبارت از اجرای دوباره‌ی زیرمجموعه‌ای از آزمون‌ها است که قبلاً اجرا شده‌اند تا اطمینان حاصل شود که تغییرات باعث انتشار اثرات جانبی ناخواسته نشده‌اند.

نکته‌ی کلیدی

انسجام از پایین به بالا نیاز به stubهای پیچیده را مرتفع می‌سازد.

اندرز

آزمون رگرسیون، راهبردی مهم برای کاهش دادن «اثرات جانبی» است. آزمون‌های رگرسیون را هر بار که تغییری عمده (از جمله انسجام مؤلفه‌های جدید) در نرم‌افزار رخ می‌دهد، اجرا کنید.

در زمینه‌های گسترده‌تر، آزمون‌های موفق (از هر نوع) به کشف خطاها می‌انجامند و این خطاها باید تصحیح شوند. هرگاه نرم‌افزار تصحیح شد، جنبه‌ای از پیکربندی نرم‌افزار (برنامه، مستندات، یا داده‌هایی که آن را پشتیبانی می‌کنند) تغییر می‌یابد. آزمون رگرسیون، فعالیتی است که به کمک آن می‌توان اطمینان یافت که تغییرات (ناشی از آزمون یا به هر دلیل دیگر)، خطاهای رفتاری ناخواسته‌ای را ایجاد نمی‌کنند. شکل ۷-۱۸ انسجام پایین به بالا .

آزمون رگرسیون را می‌توان به‌طور دستی، با اجرای دوباره‌ی زیرمجموعه‌ای از کلیه موارد آزمون یا با استفاده از ابزارهای عقبگرد انجام داد. ابزارهای عقبگرد، مهندس نرم‌افزار را قادر می‌سازد تا موارد آزمون و نتایج مربوط به عقبگرد و مقایسه بعدی را تهیه کند.

آزمون‌های رگرسیون (زیرمجموعه‌ای از آزمون‌ها که دوباره اجرا می‌شوند)، حاوی سه طبقه متفاوت از موارد آزمون است:

- یک نمونه‌ی نمایشی از آزمون‌ها که همه‌ی عملکردهای نرم‌افزار را امتحان می‌کنند؛
- آزمون‌های اضافی که بر عملکردهای نرم‌افزار تأکید دارند، احتمالاً از تغییرات تأثیر می‌پذیرند؛
- آزمون‌هایی که بر مؤلفه‌های تغییریافته نرم‌افزار تأکید دارند.

به موازات پیشرفت آزمون، تعداد آزمون‌های رگرسیون می‌تواند بسیار زیاد شود. از این رو، آزمون‌های رگرسیون را باید طوری طراحی کرد که فقط آزمون‌های مرتبط با یک یا چند طبقه از خطاها را در هر یک از عملکردهای اصلی برنامه در برگیرند. اجرای دوباره‌ی همه‌ی آزمون‌ها برای هر یک از عملکردهای برنامه، پس از اعمال تغییر، کاری غیر عملی است و بازدهی چندانی هم ندارد.

آزمون دود. آزمون دود یک روش آزمون انسجام است که به‌طور متداول هنگام توسعه محصولات نرم‌افزاری «بسته‌بندی شده» به‌کار می‌رود. این آزمون به‌عنوان یک راهکار گام‌به‌گام برای پروژه‌هایی طراحی می‌شود که زمان در آنها اهمیت فراوان دارد و تیم نرم‌افزاری را قادر می‌سازد تا پروژه را چندین بار ارزیابی کند. روش آزمون دود، در اصل شامل فعالیت‌های زیر می‌شود:

۱. مؤلفه‌های نرم‌افزاری که به کد ترجمه شده‌اند، گرد هم آورده می‌شوند تا یک سازه را تشکیل دهند. این سازه کلیه فایل‌های داده، کتابخانه‌ها، پیمانه‌های قابل استفاده‌ی مجدد و مؤلفه‌های مهندسی شده موردنیاز برای پیاده‌سازی یک یا چند عملکرد از محصول را دربرمی‌گیرند.

۲. آزمون‌هایی طراحی می‌شود تا خطاهایی را پیدا کنند که مانع ازاجرای درست عملکرد می‌شوند. هدف باید کشف خطاهای «خطرناکی» باشد که پروژه را با احتمال بیشتری به تعویق می‌اندازند.

۳. این سازه به سازه‌های دیگر ملحق می‌شود و کل محصول (در شکل فعلی خود) روزانه مورد آزمون دود قرار می‌گیرد. روش انسجام ممکن است از بالا به پایین یا پایین به بالا باشد.

تعداد دفعات روزانه‌ی آزمون کل محصول ممکن است برخی خوانندگان را شگفت زده کند. ولی آزمون‌های فراوان، به مدیران و سازندگان نسبت به پیشرفت آزمون انسجام دید واقع‌بینانه می‌دهد. مک کانل [MCO96] آزمون دود را چنین وصف می‌کند:

> آزمون دود باید سیستم کامل را از ابتدا تا انتها امتحان کند. لازم نیست این آزمون جامع باشد بلکه باید قادر به آشکارکردن مشکلات اصلی باشد. این آزمون باید به اندازه‌ای کامل باشد که اگر سازه در آن قبول شد، بتوان فرض کرد که پایداری آن به حدی است که بتوان آزمون‌های کامل‌تری روی آن انجام داد.

نکته‌ی کلیدی

آزمون دود را می‌توان به‌عنوان یک راهبرد انسجام دوار در نظر گرفت. نرم‌افزار، دوباره ساخته می‌شود (با افزودن مؤلفه‌های جدید) و آزمون دود هر روز انجام می‌شود.

ساخت روزانه را به‌عنوان نبض پروژه در نظر بگیرید. اگر نبضی نبود، پروژه مرده است.

جیم مک‌کارتی

اجرای آزمون دود بر روی پروژه‌های مهندسی نرم‌افزار پیچیده و بحرانی چند مزیت دارد:

- *خطر انسجام به حداقل می‌رسد.* چون آزمون‌های دود به‌طور روزانه اجرا می‌شوند، ناسازگاری‌ها و خطاهای خطرناک دیگر خیلی زود کشف می‌شوند و لذا احتمال وارد آمدن آسیب‌های جدی به زمان‌بندی، کاهش می‌یابد.
- *کیفیت محصول نهایی بهبود می‌یابد.* چون این روش، دارای ساختار است، احتمال یافتن خطاهای عملیاتی و نیز نقایص طراحی در سطح مؤلفه‌ها و معماری بیشتر می‌شود. اگر این نقایص، زودهنگام برطرف شوند، محصول با کیفیتی بهتر تولید خواهد شد.
- *تشخیص و تصحیح خطاها آسان می‌شود.* همانند کلیه روش‌های آزمون انسجام، خطاهای پیدا شده در اثنای آزمون دود، احتمالاً با «گام‌های جدید نرم‌افزار» همراهند - یعنی نرم‌افزاری که به سازه(ها) افزوده می‌شود، خطاهای جدیدی را ایجاد می‌کند.
- *ارزیابی پیشرفت آسان است.* با گذشت هر روز، مقدار بیشتری از نرم‌افزار انسجام می‌یابد و کارهای بیشتری باید انجام شوند. این موضوع، روحیه تیم را بهبود می‌بخشد و شاخص خوبی از پیشرفت کار را به مدیران ارائه می‌دهد.

چه مزایایی بر آزمون دود مترتب است؟

گزینه‌های راهبردی. درباره آزمون انسجام بحث‌های زیادی درباره مزایا و معایب نسبی آزمون انسجام بالا به پایین، در مقابل آزمون انسجام پایین به بالا وجود داشته است (مثلاً [Bei84]). به‌طور کلی، مزایای یک روش، در روش دیگر منجر به معایب می‌شود. عیب اصلی روش بالا به پایین، نیاز به stub‌ها و مشکلات مرتبط با آن است. مشکلات مرتبط با stub‌ها را می‌توان با مزیت آزمون زودهنگام عملکردهای کنترل اصلی جبران کرد. عیب اصلی انسجام پایین به بالا در آن‌است‌که «تا زمان افزوده‌شدن آخرین پیمانه، برنامه به‌عنوان یک موجودیت وجود ندارد» [Mye79] . این پیامد منفی، با طراحی آسان‌تر موارد آزمون و نبود stub‌ها تعدیل می‌شود.

مرجع وب

اشاره‌گرهایی به توضیحات مربوط به راهبردهای آزمون را در www.qualink.com می‌توانید بیابید.

گزینش راهبرد انسجام به ویژگی‌های نرم‌افزار و زمان‌بندی پروژه بستگی دارد. یک روند ترکیبی که گاهی *آزمون ساندویچی* نامیده می‌شود، به این صورت قابل استفاده است که برای سطوح بالاتر ساختار برنامه، از روش بالا به پایین و برای سطح زیرین از روش پایین به بالا استفاده می‌گردد.

«پیمانه‌ی بحرانی» چیست و چرا باید آن را شناسایی کرد؟

به موازاتی که آزمون انسجام اجرا می‌شود، آزمون‌گر باید پیمانه‌های بحرانی را شناسایی کند. پیمانه بحرانی یک یا چند ویژگی زیر را دارد: (۱) با چندین خواسته نرم‌افزار سروکار دارد؛ (۲) سطح بالایی از کنترل را داراست (در ساختار برنامه جایگاه نسبتاً بالایی دارد)؛ (۳) پیچیده یا مستعد خطا است (به‌عنوان یک شاخص از پیچیدگی سیکلوماتیک می‌توان استفاده کرد)؛ یا (۴) خواسته‌های کارایی مشخصی دارد. پیمانه‌های بحرانی را هر چه زودتر باید آزمون کرد. به‌علاوه، آزمون‌های رگرسیون نیز باید بر عملکرد پیمانه‌های بحرانی تأکید کنند.

محصولات کاری آزمون انسجام. طرح کلی برای انسجام بخشیدن به نرم‌افزار و توصیف آزمون‌های خاص در مشخصه‌ی آزمون مستندسازی می‌شود. این محصول کاری شامل یک طرح آزمون و یک روال آزمون می‌شود و بخشی از پیکربندی نرم‌افزار به شمار می‌رود. آزمون به چند مرحله و ساخت تقسیم می‌شود که به خصوصیات رفتاری و عملیاتی مشخصی از نرم‌افزار می‌پردازند. برای مثال، آزمون انسجام برای سیستم امنیتی *SafeHome* را می‌توان به مراحل زیر تقسیم کرد:

- *تعامل با کاربر* (ورودی و خروجی فرمان‌ها، ارائه صفحه نمایش، پردازش و نمایش خطاها)
- *پردازش حس گرها* (به‌دست آوردن خروجی حس گرها، تعیین شرایط حس گرها، کنش‌های مورد نیاز به‌عنوان پیامد شرایط)
- *قابلیت‌های عملیاتی ارتباطاتی* (توانایی برقراری ارتباط با ایستگاه پایش مرکزی)
- *پردازش هشدارها* (آزمون کنش‌هایی از نرم‌افزار که در صورت برخوردن به هشدار رخ می‌دهد.)

در طراحی آزمون‌های انسجام از چه ملاک‌هایی باید استفاده کرد؟

هر کدام از این مراحل آزمون انسجام، یک گروه عملیاتی گسترده را در نرم‌افزار ترسیم می‌کند و عموماً می‌توان آن را به یک دامنه مشخص در معماری نرم‌افزار ربط داد. بنابراین، گروه‌هایی از پیمانه‌ها در قالب برنامه جدید ایجاد می‌شوند تا به هر مرحله پاسخ دهند:

انسجام واسط‌ها. با ملحق شدن هر پیمانه (یا خوشه) به ساختار، واسط‌های داخلی و خارجی آزمایش می‌شوند.

اعتبار عملیاتی. آزمون‌هایی اجرا می‌شوند که برای کشف خطاهای عملیاتی طراحی شده‌اند.

محتویات اطلاعاتی. آزمون‌هایی اجرا می‌شوند که برای کشف خطاهای مرتبط با ساختمان داده‌های محلی یا سرتاسری طراحی شده‌اند.

کارایی. آزمون‌هایی اجرا می‌شوند که برای وارسی بر مرزهای کارایی تعیین شده طی مرحله‌ی طراحی نرم‌افزار طراحی شده‌اند.

زمان‌بندی برای انسجام، توسعه نرم‌افزار سربار و مباحث مرتبط نیز به‌عنوان بخشی از طرح آزمون نیز بحث می‌شوند. تاریخ آغاز و پایان برای هر مرحله تعیین می‌شود و «پنجره‌های دسترسی» برای پیمانه‌هایی که مورد آزمون واحد قرار گرفته‌اند، تعریف شده‌اند. شرح مختصری از نرم‌افزارهای سربار (stub‌ها و راه‌اندازها) بر خصوصیاتی تاکید دارد که ممکن است نیاز به تلاش ویژه داشته باشند. سرانجام، محیط و منابع آزمون نیز شرح داده می‌شوند. پیکربندی‌های غیرعادی ساخت‌افزار، شبیه‌سازی‌های عجیب و تکنیک‌ها و ابزارهای آزمون ویژه تنها چند مورد از مباحث فراوانی هستند که ممکن است بحث شوند.

سپس روال مشروح آزمونی توصیف می‌شود که برای دستیابی به یک طرح آزمون مورد نیاز است. فهرستی از کلیه‌ی موارد آزمون (که برای ارجاع‌های بعدی حاشیه‌گذاری می‌شود) و همچنین نتایج مورد انتظار لحاظ می‌شود. تاریخچه‌ای از نتایج، مشکلات یا وقایع عجیب در یک گزارش آزمون ثبت می‌شود که در صورت تمایل می‌توان آن را ضمیمه‌ی مشخصه‌ی آزمون کرد. اطلاعات موجود در این بخش به هنگام نگهداری نرم‌افزار می‌توانند حیاتی باشند. مراجع و پیوست‌های مناسب نیز ارائه می‌شوند.

قالب‌بندی مشخصه‌ی آزمون را همانند سایر عناصر پیکربندی نرم‌افزار، می‌توان مطابق با نیازهای سازمان مهندسی نرم‌افزار تغییر داد. به هر حال، ذکر این نکته حائز اهمیت است که یک راهبرد انسجام (موجود در طرح آزمون) و جزئیات آزمون (که در روال آزمون شرح داده می‌شود) از عناصر ضروری بوده باید لحاظ شوند.

۱۷-۴ راهبردهای آزمون برای نرم‌افزارهای شیءگرا[1]

هدف آزمون به بیانی ساده، یافتن حداکثر تعداد خطاها با مقدار مشخصی تلاش در یک دوره زمانی

[1] مفاهیم پایه‌ی شیءگرایی در پیوست ۲ ارائه شده‌اند.

واقع‌بینانه است. گرچه این هدف بنیادی برای نرم‌افزارهای شیءگرا همچنان به قوت خود باقی می‌ماند، ماهیت برنامه‌های شیءگرا ، راهبرد و تاکتیک آزمون (فصل ۱۹) را تحت تأثیر قرار می‌دهد.

۱۷-۴-۱ آزمون واحدها در حیطه‌ی OO (شیءگرا)

در نرم‌افزارهای شیءگرا مفهوم واحد فرق می‌کند. بسته‌بندی، تعریف کلاس‌ها و اشیاء را راهبری می‌کند. این بدان معناست که هر کلاس و هر نمونه از یک کلاس (شیء) صفات (داده‌ها) و عملیاتی را که این داده‌ها را دستکاری می‌کنند، بسته‌بندی می‌کند. به‌جای آزمون یک پیمانه‌ی منفرد، کوچکترین واحد قابل آزمون، کلاس یا شیء بسته‌بندی شده است. چون کلاس می‌تواند حاوی چند عمل متفاوت باشد و یک عمل خاص ممکن است به‌عنوان بخشی از چند کلاس متفاوت وجود داشته باشد، معنای آزمون واحد تغییر می‌کند.

دیگر نمی‌توانیم یک عمل را به‌طور جداگانه (دیدگاه سنتی آزمون واحدها) آزمون کنیم، بلکه آن را باید به‌عنوان جزئی از یک کلاس بیازماییم. برای روشن شدن مطلب، سلسله مراتبی از کلاس‌ها را در نظر بگیرید که در آن عملیات X برای کلاس پایه تعریف می‌شود و چند زیرکلاس آن را به ارث می‌برند. هر زیرکلاسی از عملیات X استفاده می‌کند، ولی در حیطه‌ی صفات خصوصی و عملیاتی به‌کار برده می‌شود که برای هر زیرکلاس تعریف شده‌اند. چون حیطه‌ای که عملیات X در آن به‌کار می‌رود، تغییر می‌کند، لازم است عملیات X در حیطه هر یک از زیرکلاس‌ها آزمون شود. به عبارت دیگر، آزمون عملیات X در محیط خلاء (روش سنتی آزمون واحدها) در زمینه شیءگرا بازدهی ندارد.

آزمون کلاس‌ها برای نرم‌افزارهای OO، هم‌ارز آزمون واحدها برای نرم‌افزارهای سنتی است. برخلاف آزمون واحدها در نرم‌افزارهای سنتی که بیشتر بر جزئیات الگوریتمی یک پیمانه و داده‌هایی تأکید دارد که در میان واسط پیمانه جریان پیدا می‌کند، آزمون کلاس‌ها در نرم‌افزارهای شیءگرا ، به وسیله عملیات بسته‌بندی شده توسط کلاس و رفتار حالت‌های کلاس انجام می‌شود.

نکته‌ی کلیدی

آزمون کلاس‌ها برای نرم‌افزارهای OO مشابه با آزمون پیمانه‌ها برای نرم‌افزارهای سنتی است و برای آزمون عملیات‌های منفرد توصیه نمی‌شود.

۱۷-۴-۲ آزمون انسجام در حیطه‌ی OO

از آنجا که نرم‌افزارهای شیءگرا فاقد ساختار کنترلی سلسله مراتبی‌اند، راهبردهای سنتی بالا به پایین و پایین به بالا چندان معنایی ندارند. به‌علاوه، عملیات انسجام به صورت هر بار یک کلاس (روش منسجم‌سازی تدریجی سنتی) نیز به‌خاطر تعامل‌های مستقیم و غیرمستقیم میان مؤلفه‌های تشکیل دهنده کلاس، غیر ممکن است [Ber93].

دو راهبرد متفاوت برای آزمون انسجام سیستم‌های شیءگرا وجود دارد [Bin94b] . اولی، یعنی آزمون نخ‌ها (threads)، مجموعه‌ای از کلاس‌های لازم برای پاسخ‌دهی به یک ورودی یا رویداد سیستم را مجتمع می‌کند. هر نخ به‌طور انفرادی مجتمع و آزمون می‌شود. از آزمون رگرسیون استفاده می‌شود تا تعیین گردد که هیچ اثر جانبی‌ای به‌وجود نمی‌آید. روش انسجام دوم، یعنی آزمون مبتنی بر کاربرد، ساخت سیستم را با آزمون آن دسته از کلاس‌هایی آغاز می‌کند که از تعداد کلاس‌های سرور اندکی استفاده می‌کنند (این کلاس‌ها را *مستقل* نیز می‌گویند). پس از آنکه کلاس‌های مستقلی آزمون شدند، لایه بعدی کلاس‌ها (موسوم به کلاس‌های وابسته) که از کلاس‌های مستقل استفاده می‌کنند، مورد آزمون قرار می‌گیرند. این دنباله از آزمون کلاس‌های وابسته آنقدر ادامه می‌یابد تا کل سیستم ایجاد شود.

نکته‌ی کلیدی

یک راهبرد مهم برای آزمون انسجام‌بخشی نرم‌افزارهای OO، آزمون نخ‌هاست. نخ‌ها مجموعه‌هایی از کلاس‌ها هستند که به یک ورودی یا رویداد پاسخ می‌دهند. آزمون‌های مبتنی بر کاربرد بر کلاس‌هایی تأکید دارند که همکاری سنگینی با سایر کلاس‌ها ندارند.

استفاده از درایورها و stubها زمان اجرای آزمون‌ها را نیز تغییر می‌دهد. از درایورها می‌توان برای آزمودن عملیات‌ها در پایین‌ترین سطوح و برای آزمودن گروه کاملی از کلاس‌ها بهره برد. از درایورها برای جایگزین کردن واسط کاربر نیز می‌توان استفاده کرد، به‌طوری که آزمون‌های قابلیت عملیاتی سیستم را می‌توان پیش از پیاده‌سازی واسط اجرا کرد. از stubها می‌توان در شرایطی استفاده کرد که به همکاری میان کلاس‌ها نیاز است، ولی یک یا چند کلاس همکار هنوز به‌طور کامل پیاده‌سازی نشده‌اند.

آزمون خوشه‌ای، یک مرحله از آزمون انسجام در نرم‌افزارهای شیءگرا است. در این‌جا، خوشه‌ای از کلاس‌های همکار (که با بررسی مدل روابط میان اشیا و CRC تعیین می‌شود) با طراحی موارد آزمونی که سعی در کشف خطاهای موجود در همکاری‌ها دارند، تمرین داده می‌شود.

۱۷-۵ راهبردهای آزمون برای برنامه‌های تحت وب

راهبرد مربوط به آزمون برنامه‌های تحت وب، شامل همان اصول پایه‌ای مربوط به همه‌ی آزمون‌های نرم‌افزار می‌شود و تاکتیک‌هایی در آن به‌کار می‌رود که برای سیستم‌های شیءگرا به‌کار می‌رود. این رویکرد در مراحل زیر خلاصه می‌شود:

۱. مدل محتویات برای برنامه‌ی تحت وب بازبینی می‌شود تا خطاها آشکار شوند.
۲. مدل واسط بازبینی می‌شود تا اطمینان حاصل شود که همه‌ی موارد آزمون را می‌توان انجام داد.
۳. مدل طراحی برای برنامه‌ی تحت وب بازبینی می‌شود تا خطاهای گشت‌وگذار کشف شود.
۴. واسط کاربر آزمون می‌شود تا خطاهای موجود در شیوه‌ی عرضه و/یا مکانیک گشت‌وگذار کشف گردند.
۵. هر کدام از مؤلفه‌های عملیاتی آزمون می‌شود.
۶. گشت‌وگذار در سرتاسر معماری آزمون می‌شود.
۷. برنامه‌ی تحت وب در انواع پیکر بندی‌های محیطی متفاوت پیاده‌سازی و از نظر سازگاری با هر پیکربندی آزمون می‌شود.
۸. آزمون‌های امنیتی انجام می‌شوند تا نقاط آسیب‌پذیر در داخل برنامه‌ی تحت وب یا محیط اطراف آن آشکار گردد.
۹. آزمون‌های کارایی اجرا می‌شوند.
۱۰. برنامه‌ی تحت وب توسط تعدادی از کاربران نهایی آزمون می‌شود که تحت کنترل و پایش هستند. نتایج تعامل آن‌ها با سیستم از نظر خطاهای موجود در گشت‌وگذار و محتویات، قابلیت استفاده، قابلیت سازگاری و قابلیت اطمینان و کارایی برنامه‌ی تحت وب ارزیابی می‌شوند.

نکته‌ی کلیدی

راهبرد کلی برای آزمون برنامه‌های تحت وب را می‌توان در ده مرحله مقابل خلاصه کرد.

مرجع وب

مقالاتی عالی درباره آزمون برنامه‌های تحت وب را می‌توان در آدرس زیر یافت: www.stickyminds.com/testing.asp

از آنجا که پیوسته تعداد بی‌شماری از برنامه‌های تحت وب در حال تکامل هستند، فرایند آزمون فعالیتی مداوم است و توسط کارمندان پشتیبانی اجرا می‌شود؛ آن‌ها از آزمون‌های رگرسیونی استفاده می‌کنند که خود از آزمون‌های تهیه شده به هنگام اولین دور مهندسی برنامه‌ی تحت وب به‌دست آمده‌اند. در فصل ۲۰ به روش‌های آزمون برنامه‌های تحت وب خواهیم پرداخت.

۱۷-۶ آزمون اعتبارسنجی

آزمون اعتبارسنجی در پایان آزمون انسجام آغاز می شود، یعنی هنگامی که تک‌تک مؤلفه‌ها تمرین داده شدند، نرم‌افزار به‌طور کامل مونتاژ شد و خطاهای واسطها کشف و برطرف شدند. در سطح اعتبارسنجی یا سیستمی، تمایز میان نرم‌افزارهای سنتی، شیءگرا و تحت وب رنگ می‌بازد. آنچه در آزمون کانون توجه قرار می‌گیرد، کنش‌هایی است که به چشم کاربر می‌آیند و خروجی‌هایی از سیستم است که برای کاربر قابل تشخیص هستند.

اعتبارسنجی را به روش‌های مختلفی می‌توان تعریف کرد. ولی یک تعریف ساده آن‌است‌که اعتبارسنجی هنگامی موفق است که نرم‌افزار براساس انتظار مشتری عمل کند. برنامه‌نویس ممکن است این پرسش را مطرح کند که «منطقی بودن این انتظارات را چه کسی داوری می‌کند؟» اگر برای *خواسته‌های نرم‌افزار* یک مشخصه تهیه شده باشد، همه‌ی صفات قابل مشاهده نرم‌افزار برای کاربر را توصیف می‌کند و حاوی بخشی مختص ملاک‌های اعتبارسنجی است که مبنایی برای رویکرد آزمون اعتبارسنجی تشکیل می‌دهد.

۱-۶-۱۷ ملاک‌های آزمون اعتبارسنجی

اعتبارسنجی نرم‌افزار از طریق آزمون‌های جعبه سیاه انجام می‌شود که نشان‌دهنده‌ی مطابقت آن با خواسته‌ها هستند. در برنامه‌ریزی آزمون، انواع آزمون‌هایی که باید انجام شود، خلاصه می‌شود و رویه‌ی آزمون، مواردی از آزمون را تعریف می‌کند که مطابقت با خواسته‌ها را نشان می‌دهند. برنامه‌ریزی و رویه به این منظور طراحی می‌شوند که اطمینان حاصل شود کلیه خواسته‌های عملیاتی برآورده شده‌اند؛ همه‌ی ویژگی‌های رفتاری رعایت شده‌اند؛ همه‌ی خواسته‌های کارایی محفوظ است؛ مستندات صحیح و به دست انسان تنظیم شده است؛ و خواسته‌های دیگر رعایت شده‌اند (مثل قابلیت حمل، سازگاری، تحمل خطا و قابلیت نگهداری).

پس از اجرای هر یک از موارد آزمون، یکی از دو حالت ممکن وجود خواهد داشت: (۱) ویژگی‌های عملیاتی یا کارایی با مشخصات مطابقت دارند و پذیرفته می‌شوند، یا (۲) یک انحراف از مشخصات کشف می‌شود و فهرست کاستی‌ها تهیه می‌شود. غالباً برای رفع کاستی‌ها نیاز به مشاوره با مشتری است.

۲-۶-۱۷ بازبینی پیکربندی

یک عنصر مهم از فرایند اعتبارسنجی، بازبینی پیکربندی است. هدف این بازبینی آن‌است‌که اطمینان حاصل آید کلیه عناصر پیکربندی نرم‌افزار به‌طور مناسب توسعه یافته‌اند؛ از آنها شناسنامه تهیه شده است و دارای جزئیات لازم برای تقویت کردن فاز پشتیبانی چرخه زندگی نرم‌افزار هستند. بازبینی پیکربندی، که گاه ممیزی (audit) خوانده می‌شود، به تفصیل بیشتر در فصل ۲۲ مورد بحث قرار گرفته است.

۳-۶-۱۷ آزمون آلفا و بتا

درحقیقت، سازنده نرم‌افزار نمی‌تواند پیش‌بینی کند که مشتری چگونه از نرم‌افزار استفاده خواهد کرد. راهنمای استفاده ممکن است به‌درستی تفسیر نشود؛ ممکن است ترکیبات نامشخصی از داده‌ها به‌طور

نکته‌ی کلیدی

همانند همه‌ی مراحل دیگر آزمون، اعتبارسنجی سعی در کشف خطاها دارد، ولی آن‌چه کنون توجه قرار می‌گیرد، در سطح خواسته‌هاست- چیزهایی که بلافاصله در نظر کاربر نهایی پدیدار می‌شود.

«در صورت تعداد کافی از چشم‌های نظاره‌گر، همه‌ی اشکال‌ها کم‌عمق هستند (مثلاً اگر تعداد آزمون‌گرهای بتا و کمک سازنده‌ها کافی باشد، تقریباً هر مشکلی به سرعت مشخص می‌شود و راهکار آن برای کسی آشکار می‌شود.»

ا. ریموند

منظم استفاده شود؛ خروجی‌ای که در نظر آزمون‌گر واضح جلوه می‌کند، ممکن است برای کاربر نهایی نامفهوم باشد.

هنگامی که یک نرم‌افزار سفارشی برای یک مشتری ساخته شود، آزمون پذیرش اجرا می‌شود تا مشتری را قادر به اعتبارسنجی کلیه‌ی خواسته‌ها کند. آزمون پذیرش، که به‌جای مهندس نرم‌افزار، توسط کاربر نهایی انجام می‌شود، می‌تواند از یک «آزمون غیررسمی» تا یک سری آزمون‌های برنامه‌ریزی‌شده‌ی سیستماتیک را شامل شود. درواقع، آزمون پذیرش را می‌توان در عرض یک هفته یا یک ماه انجام داد و لذا کشف خطاهایی که ممکن‌است سیستم را با گذشت زمان تنزل دهند، امکان‌پذیر می‌شود.

اگر نرم‌افزار به‌عنوان محصولی ساخته می‌شود که قرار است مشتریان زیادی از آن استفاده کنند، انجام آزمون پذیرش توسط یکایک آنها امکان‌پذیر نیست. اکثر سازندگان محصولات نرم‌افزاری از فرایندی موسوم به آزمون آلفا و بتا، برای کشف خطاهایی استفاده می‌کنند که به نظر می‌رسد فقط کاربر نهایی قادر به یافتن آنها باشد.

آزمون آلفا، در مکان سازنده‌ی نرم‌افزار، توسط مشتری انجام می‌شود. نرم‌افزار در شرایط طبیعی اجرا می‌شود، به‌طوری که سازنده ناظر بر اجرای آزمون بوده خطاها و مشکلات را ثبت می‌کند. آزمون‌های آلفا در محیطی کنترل شده اجرا می‌شوند.

آزمون بتا در یک یا چند مکان متعلق به مشتری یا کاربر نهایی نرم‌افزار انجام می‌شود. برخلاف آزمون آلفا، سازنده معمولاً حضور ندارد. بنابراین، آزمون بتا یک کاربرد «زنده» از نرم‌افزار در محیطی است که سازنده قادر به کنترل آن نیست. مشتری کلیه مشکلات (واقعی یا تصوری) را که طی آزمون بتا یافته است، ثبت می‌کند و آنها را در فاصله‌های زمانی منظم به سازنده گزارش می‌کند. مهندسان نرم‌افزار، با توجه به مشکلات گزارش شده طی آزمون بتا، اصلاحات لازم را انجام می‌دهند و سپس محصول نرم‌افزاری را برای ارائه به مشتریان آماده می‌کنند.

شکل دیگری از آزمون بتا که به آزمون پذیرش مشتری موسوم است، گاهی اجرا می‌شود، یعنی زمانی که یک نرم‌افزار سفارشی تحت قرارداد به مشتری تحویل داده می‌شود. مشتری یک سری آزمون‌های مشخص انجام می‌دهد تا خطاها را قبل از پذیرفتن نرم‌افزار از سازنده آن کشف کند. در برخی موارد (مثلاً یک شرکت بزرگ یا سیستم دولتی) آزمون پذیرش می‌تواند بسیار رسمی باشد و روزها یا حتی هفته‌ها به طول انجامد.

اختلاف میان آزمون آلفا و بتا در چیست؟

۱۷-۷ آزمون سیستم

در آغاز این فصل، بر این واقعیت تأکید کردیم که نرم‌افزار فقط عنصری از یک سیستم کامپیوتری بزرگتر به شمار می‌رود. درنهایت، نرم‌افزار به عناصر دیگر سیستم (مثلاً سخت‌افزار، افراد و اطلاعات) ملحق می‌شود و آزمون‌های انسجام و اعتبارسنجی روی آنها انجام می‌گیرد. این آزمون‌ها در خارج از دامنه فرایند نرم‌افزار قرار دارند و فقط توسط مهندسان نرم‌افزار انجام نمی‌شوند. ولی مراحل طی شده در اثنای طراحی و آزمون نرم‌افزار می‌توانند تا حد زیادی احتمال موفقیت انسجام نرم‌افزار را در سیستم بزرگتر بهبود بخشند.

«آزمون‌ها نیز همانند مرگ و مالیات، هم ناخوشایند و هم اجتناب‌ناپذیرند.»

اد یوردون

SafeHome

آمادگی برای اعتبارسنجی

صحنه: دفتر داگ میلر، همچنان که در سطح مؤلفه‌ها و ساخت برخی مؤلفه‌ها ادامه پیدا می‌کند.

نقش‌آفرینان: داگ میلر، مدیر مهندسی نرم‌افزار، وینود، جیمی، اد و شکیرا- اعضای تیم مهندسی نرم‌افزار *SafeHome*

گفتگو:

داگ: نسخه‌ی اول برای اعتبارسنجی در حدود سه هفته حاضر می‌شود؟

وینود: تقریباً بله. انسجام به خوبی دارد پیش می‌رود. آزمون دود را هر روز انجام می‌دهیم و یک تعداد اشکال پیدا می‌کنیم، ولی چیزی نیست که نتوانیم از عهده‌اش برآییم. تا حالا خیلی خوب بوده.

داگ: از اعتبارسنجی برایم بگویید.

شکیرا: خب، ما از همه‌ی use case‌ها به‌عنوان مبنایی برای طراحی آزمون‌ها استفاده می‌کنیم. من هنوز شروع نکرده‌ام، ولی برای همه‌ی use case‌هایی که مسؤول آن‌ها بودم، یک سری آزمون طراحی کردم.

اد: من هم.

جیمی: من هم، ولی ما باید کارهایمان را برای آزمون پذیرش و همچنین برای آزمون آلفا و بتا هماهنگ کنیم. نه؟

داگ: بله. در واقع داشتم فکر می‌کردم؛ می‌توانیم با یک شرکت دیگر قرارداد ببندیم تا در کار اعتبارسنجی به ما کمک کند. بودجه‌ی کافی برای این کار داریم... و این به ما یک دیدگاه جدید می‌دهد.

وینود: فکر می‌کنم اوضاع تحت کنترل باشد.

داگ: مطمئنم که همین طور است، ولی یک ITG دید مستقلی از نرم‌افزار به ما می‌دهد.

جیمی: داگ، ما این‌جا وقت کم داریم. من یکی که وقت ندارم تا از آدم‌هایی که می‌فرستی، مراقبت بکنم.

داگ: می‌دانم، می‌دانم، ولی اگر یک ITG از خواسته‌ها و use case‌ها جواب بدهد، مراقبت زیادی لازم نیست.

وینود: من هنوز هم فکر می‌کنم همه چیز تحت کنترل است.

داگ: این را شنیدم وینود، ولی می‌خواهم در این مورد اعمال نفوذ کنم. یک قرار ملاقات با نماینده ITG در همین هفته می‌گذاریم. بگذارید کارشان را شروع کنند و ببینیم چه می‌کنند.

وینود: بسیار خب، شاید بار کاری را قدری روشن کند.

یک مشکل آزمون سیستم، «متهم کردن دیگران» است و هنگامی رخ می‌دهد که خطایی کشف شود و سازنده‌ی هر عنصر از سیستم، تقصیر را به گردن دیگری می‌اندازد. مهندس نرم‌افزار به‌جای درگیرشدن با این موضوعات، باید مشکلات بالقوه‌ی برقراری ارتباط با عناصر دیگر سیستم را

پیش‌بینی کند و (۱) مسیرهایی برای کنترل خطا طراحی کند که همه‌ی اطلاعات وارده از عناصر دیگر سیستم را آزمون کند؛ (۲) آزمون‌هایی را اجرا کند که داده‌های بد یا خطاهای بالقوه‌ی دیگر موجود در نرم‌افزار را شبیه‌سازی کنند؛ (۳) نتایج آزمون را ثبت کند تا در صورت متهم شدن، آنها را به‌عنوان مدرک ارائه کند و (۴) در برنامه‌ریزی و طراحی آزمون‌های سیستم شرکت کند تا اطمینان حاصل شود نرم‌افزار به اندازه کافی آزمون شده است.

آزمون سیستم، مجموعه‌ای از آزمون‌های متفاوت است که هدف اصلی آنها امتحان کل سیستم کامپیوتری است. گرچه هر آزمون دارای هدفی متفاوت است، وظیفه‌ی همه‌ی آنها وارسی این نکته است که عناصر سیستم به‌طور مناسب مجتمع شده‌اند و عملکردهای مربوطه را انجام می‌دهند. در بخش‌های بعدی، انواع آزمون‌های سیستم را که برای سیستم‌های کامپیوتری مفید واقع می‌شوند، مورد بحث قرار می‌دهیم.

۱-۷-۱۷ آزمون ترمیم (Recovery Testing)

بسیاری از سیستم‌های کامپیوتری باید خود را در شرایط نقص ترمیم کنند و در یک زمان از پیش تعیین شده پردازش را ادامه دهند. در برخی موارد، سیستم باید در برابر نقایص تحمل داشته باشد، یعنی نقایص در پردازش نباید باعث توقف عملکرد کلی سیستم شود. در موارد دیگر، شکست باید در یک دوره زمانی مشخص تصحیح شود وگرنه زیانهای اقتصادی شدیدی وارد خواهد آمد.

آزمون ترمیم یکی از آزمون‌های سیستم است که نرم‌افزار را به طرق گوناگون وادار به شکست می‌کند و سپس در مورد اجرای مناسب ترمیم تحقیق می‌کند. اگر ترمیم به صورت خودکار باشد (خود سیستم آن را اجرا کند)، مقداردهی دوباره، راهکارهای ایجاد نقاط کنترلی، ترمیم داده‌ها و شروع دوباره، هر کدام، مورد ارزیابی قرار می‌گیرند. اگر ترمیم نیاز به دخالت انسان داشته باشد، زمان میانگین برای ترمیم (MTTR) تعیین می‌شود تا معلوم گردد آیا در حدود قابل قبول هست یا خیر.

۲-۷-۱۷ آزمون امنیت (Security Testing)

هر سیستم کامپیوتری که اطلاعات حساس را مدیریت می‌کند یا اعمالی انجام می‌دهد که می‌توانند باعث رساندن زیانها (یا منافع) نامتعارف به افراد شود، هدف خوبی برای نفوذ غیرقانونی یا نامتعارف به شمار می‌رود. نفوذ شامل گستره وسیعی از فعالیت‌ها می‌شود: نفوذگرانی که فقط برای سرگرمی سعی در نفوذ دارند؛ کارمندان سرخورده‌ای که می‌کوشند به خاطر انتقام گرفتن نفوذ کنند؛ افراد نادرستی که برای منافع شخصی سعی در نفوذ دارند.

آزمون امنیت کوشش می‌کند تا وارسی کند که راهکارهای محافظ تعبیه شده در داخل سیستم واقعاً آن را از نفوذ نامناسب حفظ می‌کنند. بیزر [Bei84] می‌گوید: «امنیت سیستم باید از نظر آسیب‌پذیری در برابر حملات از جلو، و حملات پشت سر مورد آزمون قرار گیرد».

آزمون‌گر در اثنای آزمون امنیت، نقش فردی را بازی می‌کند که مایل به نفوذ به سیستم است. همه چیز پیش می‌رود! آزمون‌گر ممکن است کوشش کند تا کلمه‌های عبور را به طرق گوناگون به دست آورد، ممکن‌است با استفاده از نرم‌افزارهایی به ساختار دفاعی سیستم حمله کند. ممکن است سیستم را مغلوب کند و درنتیجه از ارائه خدمات به دیگران جلوگیری به عمل آید. ممکن‌است باعث ایجاد خطا

در سیستم شود تا در هنگام بازیابی سیستم به آن نفوذ کند؛ ممکن است در میان داده‌های غیر ایمن سیر کند تا کلیدی برای ورود به سیستم پیدا کند.

آزمون امنیت در صورت در اختیار داشتن زمان و منابع کافی، به سیستم نفوذ خواهد کرد. طراح سیستم باید کاری کند که هزینه نفوذ به سیستم بیشتر از هزینه اطلاعاتی باشد که در اثر نفوذ به‌دست می‌آید.

۱۷-۷-۳ آزمون فشار (Stress Testing)

در مراحل اولیه آزمون نرم‌افزار، تکنیک‌های جعبه سیاه یا جعبه سفید منجر به ارزیابی کامل کارایی و عملکردهای عادی برنامه می‌شد. آزمون‌های فشار برای مقابله با شرایط غیرعادی طراحی می‌شود. در اصل، آزمون‌گری که آزمون فشار را اجرا می‌کند، می‌پرسد: «نرم‌افزار را قبل از شکست تا کجا می‌توان تحت فشار قرار داد؟»

آزمون فشار، سیستم را به شیوه‌ای اجرا می‌کند که منابع را به میزان غیرعادی، فراوانی غیرعادی یا حجم غیرعادی طلب کند. برای مثال، (۱) ممکن است آزمون‌های خاصی طراحی شود که در هر ثانیه ده وقفه ایجاد شود، در حالی که میانگین آن یک یا دو وقفه در ثانیه است؛ (۲) ممکن‌است آهنگ ورود داده‌ها چند برابر شود تا معلوم شود کدام عملکردهای مربوط به ورودی پاسخ خواهند داد؛ (۳) موارد آزمونی که مستلزم حداکثر حافظه یا منابع دیگر هستند، اجرا می‌شوند؛ (۴) موارد آزمونی طراحی می‌شوند که ممکن است باعث از کار افتادن سیستم‌عامل شوند؛ (۵) موارد آزمونی طراحی می‌شوند که ممکن است باعث آسیب رساندن به داده‌های روی دیسک شود.

شکل دیگری از آزمون فشار، تکنیکی موسوم به *آزمون حساسیت* است. در برخی شرایط (که بیشتر در الگوریتم‌های ریاضی رخ می‌دهد) ممکن است گستره‌ی بسیار کوچکی از داده‌های موجود در مرز داده‌های معتبر برای یک برنامه، باعث پردازش زیاد یا حتی نادرست یا تنزل عمیق در کارایی شود. آزمون حساسیت می‌کوشد تا ترکیباتی از داده‌ها در انواع معتبر ورودی را کشف کند، که ممکن‌است باعث ناپایداری یا پردازش نامناسب شوند.

۱۷-۷-۴ آزمون کارایی (Performance Testing)

برای سیستم‌های بی‌درنگ (real-time) یا تعبیه شده، نرم‌افزاری که عملکرد موردنیاز را فراهم می‌آورد ولی با خواسته‌های کارایی مطابقت ندارد، قابل قبول نیست. آزمون کارایی به منظور آزمون کارایی نرم‌افزار در زمان اجرا در حیطه یک سیستم انسجام یافته طراحی می‌شود. آزمون کارایی در سرتاسر مراحل فرایند آزمون رخ می‌دهد. حتی در سطح واحدها، کارایی یک پیمانه را می‌توان در اثنای اجرای آزمون‌های جعبه سفید، مورد ارزیابی قرار داد. ولی این کار تا زمانی که کلیه عناصر سیستم به‌طور کامل به هم ملحق شوند و کارایی واقعی سیستم قابل ارزیابی شود، امکان‌پذیر نیست.

آزمون‌های‌کارایی غالباً به همراه آزمون‌های فشار انجام می‌شوند و معمولاً به تجهیزات نرم‌افزاری و نیز سخت‌افزاری نیاز دارند. یعنی غالباً لازم می‌شود تا میزان استفاده از منابع (مثلاً چرخه‌های پردازنده) به شیوه‌ای دقیق اندازه‌گیری شود. با استفاده از دستگاه‌های خارجی می‌توان بر فواصل اجرا نظارت کرد، (مثلاً وقفه‌ها) را ثبت نمود، و از حالت‌های ماشین به‌طور منظم نمونه‌برداری کرد. با تجهیزات سیستم، آزمون‌گر می‌تواند شرایطی راکشف کند که منجر به تنزل و شکست احتمالی سیستم می‌شود.

«اگر تلاش می‌کنید اشکال‌های واقعی موجود در سیستم را بیابید و نرم‌افزار خود را در معرض آزمون‌های فشار واقعی قرار نداده‌اید، اکنون زمان این کار فرا رسیده است.»

بوریس بیزر

ابزارهای نرم‌افزاری

مدیریت و برنامه‌ریزی آزمون‌ها

هدف: این ابزارها، تیم نرم‌افزاری را در برنامه‌ریزی برای راهبرد آزمون انتخاب شده و مدیریت پردازش آزمون به موازات اجرای آن یاری می‌دهند.

مکانیک: ابزارهای این گروه به برنامه‌ریزی برای آزمون‌ها، ذخیره‌سازی آزمون‌ها، مدیریت و کنترل، ردگیری خواسته‌ها، انسجام بخشی، ردگیری خطاها و تولید گزارش مربوط می‌شوند. مدیران پروژه از آن‌ها برای تکمیل ابزارهای زمان‌بندی پروژه استفاده می‌کنند. آزمون‌گران از این ابزارها برای برنامه‌ریزی فعالیت‌های آزمون و کنترل جریان اطلاعات به موازات پیشرفت فرایند آزمون بهره می‌برند.

ابزارهای نمونه

QaTraq Test Case Management Tool، که توسط شرکت Traq Sofware (**www.testmanagement.com**) توسعه یافته است و «مشوق رویکردی ساخت‌یافته برای مدیریت آزمون‌هاست.»

QADirector، که توسط Compuware Corp (**www.compuware.com/qacenter**) توسعه یافته است، یک نقطه‌ی کنترل منفرد برای مدیریت همه‌ی مراحل فرایند آزمون فراهم می‌سازد.

Test Works، که توسط Software Research, Inc. (**www.soft.com/Products/index.html**) توسعه یافته است حاوی یک مجموعه کاملاً منسجم از ابزارهای آزمون از جمله ابزارهایی برای مدیریت آزمون و گزارش‌دهی می‌شود.

Opensource Test.org (**www.opensourcetesting.org/testmgt.php**) فهرستی از انواع ابزارهای برنامه‌ریزی و مدیریت آزمون ارائه می‌دهد.

NI Test Stand که توسط National Instruments Corp. (**www.ni.com**) توسعه یافته است و به شما این امکان را می‌دهد تا سری آزمون‌های نوشته شده به هر زبان برنامه‌نویسی را توسعه دهید، مدیریت کنید و اجرا نمایید.

۱۷-۷-۵ آزمون استقرار (Deployment Testing)

نرم‌افزارها در بسیاری موارد باید روی انواع سکوها و در بیش از یک نوع سیستم عامل اجرا شوند. *آزمون استقرار*، که گاهی *آزمون پیکربندی* نیز نامیده می‌شود، نرم‌افزار را در هر کدام از محیط‌هایی که قرار است در آن عمل کند تمرین می‌دهد. به‌علاوه، در آزمون استقرار، همه‌ی روال‌های نصب و نرم‌افزارهای تخصیص‌یافته‌ی نصب را که توسط مشتریان استفاده می‌شوند و همه‌ی مستندات مورد استفاده در معرفی نرم‌افزار به کاربران نهایی بررسی می‌شوند.

به‌عنوان مثال، نسخه‌ی اینترنتی نرم‌افزار *SafeHome* را در نظر بگیرید که به مشتری امکان می‌دهد تا سیستم امنیتی را از مکان‌های دوردست پایش کند. برنامه‌ی تحت وب *SafeHome* باید با به‌کارگیری همه‌ی مرورگرهای وبی که احتمال استفاده از آن‌ها می‌رود، آزمون شود. یک آزمون استقرار کامل‌تر ممکن است شامل ترکیب‌هایی از مرورگرهای مختلف و سیستم‌های عامل متفاوت باشد. (مثلاً Linux با FireFox و Windows با Opera). از آن‌جا که امنیت، مسأله‌ای اساسی است، مجموعه‌ی کاملی از آزمون‌های امنیتی با آزمون استقرار همراه خواهد شد.

۱۷-۸ هنر اشکال‌زدایی (Debugging)

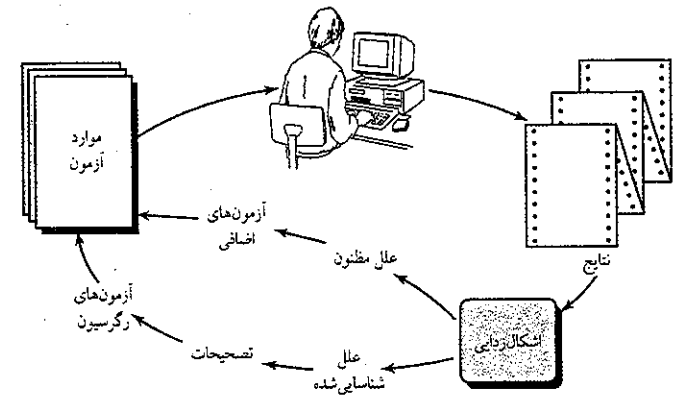
آزمون نرم‌افزار، فرایندی است که می‌توان آن را به‌طور سیستماتیک برنامه‌ریزی و مشخص نمود. مواردآزمون را می‌توان طراحی کرد، راهبردی را تعیین نمود و نتایج را در مقابل انتظارات تجویز شده مورد سنجش قرار داد.

اشکال‌زدایی در نتیجه‌ی آزمون موفق رخ می‌دهد. یعنی هنگامی که یک مورد آزمون خطایی را کشف کرد، فرایند اشکال‌زدایی برای رفع خطا به اجرا درمی‌آید. گرچه اشکال‌زدایی می‌تواند و باید فرایندی منظم باشد، مهندس نرم‌افزاری که نتایج آزمون را ارزیابی می‌کند، غالباً با «نشانگانی» (symptomatic) از شکل نرم‌افزار مواجه است. یعنی نمود خارجی خطا و علت داخلی آن ممکن‌است رابطه‌ی روشن و آشکاری با یکدیگر نداشته باشند. فرایند ذهنی که این نشانگان را به علت آن ربط می‌دهد، اشکال‌زدایی نام دارد.

۱۷-۸-۱ فرایند اشکال‌زدایی

اشکال‌زدایی، آزمون نیست بلکه پس از آزمون رخ می‌دهد[1]. همان‌طور که در شکل ۱۷-۷ نشان داده شده است، فرایند اشکال‌زدایی با اجرای یک مورد آزمون شروع می‌شود. نتایج ارزیابی می‌شوند و عدم تناظر میان نتایج واقعی و قابل انتظار مشاهده می‌شود. در بسیاری موارد، داده‌های نامربوط، نشانگان یک علت بنیادی پنهان هستند. فرایند اشکال‌زدایی تلاش می‌کند تا نشانگان را به علت ربط دهد و درنتیجه به تصحیح خطا منجر شود.

فرایند اشکال‌زدایی همواره یکی از این دو پیامد را خواهد داشت: (۱) علت پیدا شده تصحیح و برطرف می‌شود، یا (۲) علت پیدا نمی‌شود. در مورد دوم، شخصی که اشکال‌زدایی را انجام می‌دهد، ممکن‌است به علتی شک کند، مورد آزمونی طراحی کند که به اعتبارسنجی شک وی کمک کند و به‌شیوه‌ای تکراری به‌کار تصحیح خطا ادامه دهد.



شکل ۱۷-۷ فرایند اشکال‌زدایی.

«در نهایت شگفتی دریافتیم که رسیدن به برنامه‌های درست به آن آسانی‌ها هم که تصور می‌کردیم نیست. می‌توانم آن لحظه‌ای را به‌خاطر بیاورم که دریافتم بخش بزرگی از زندگی‌ام از آن پس صرف یافتن خطاهای موجود در برنامه‌هایم شده.»

موریس ویلکس،
کشف اشکال‌زدایی، ۱۹۴۹

اندرز

حتماً از پیامد سوم بپرهیزید: دلیل پیدا می‌شود، ولی تصحیحی که به عمل می‌آید، مسأله را حل نمی‌کند یا باز خطایی دیگر را باعث می‌شود.

[1]. در این بیان، وسیع‌ترین دیدگاه درخصوص آزمون را مد نظر داشتیم. نه تنها توسعه‌دهنده، نرم‌افزار را قبل از انتشار آن می‌آزماید بلکه مشتری/کاربر نیز نرم‌افزار را در هر بار استفاده از آن آزمایش می‌کند!

چرا اشکال‌زدایی تا این حد دشوار است؟ به احتمال زیاد، روان‌شناسی انسان (بخش ۲-۸-۱۷) بیشتر به این سؤال ربط دارد تا فن‌آوری نرم‌افزار. ولی، اشکال‌ها چند ویژگی دارند که سرنخ‌هایی به‌دست می‌دهند:

۱. ممکن است نشانگان و علت از نظر جغرافیایی دور از هم باشند. یعنی، نشانگان ممکن است در بخشی از برنامه ظاهر شود، در حالی که علت ممکن است واقعاً در مکانی بسیار دورتر واقع شود، مؤلفه‌هایی که چسبندگی زیادی به هم دارند (فصل ۸)، این وضعیت را بدتر می‌کنند.

۲. نشانگان ممکن است با تصحیح یک خطای دیگر (به‌طور موقت) ناپدید شود.

۳. نشانگان ممکن است واقعاً ناشی از غیرخطاها (مثلاً عدم صحت ناشی از گردکردن اعداد) باشد.

۴. نشانگان ممکن است ناشی از خطای انسانی باشد که به سادگی قابل ردگیری نیست.

۵. نشانگان ممکن است نتیجه مشکلات زمان‌بندی باشد نه مشکلات پردازشی.

۶. ممکن است بازسازی صحیح شرایط ورودی دشوار باشد (مثل یک کاربرد بی‌درنگ که در آن ترتیب ورودی نامعین است).

۷. نشانگان ممکن است متقطعی باشد. این موضوع به ویژه در سیستم‌های تعبیه شده‌ای متداول است که سخت‌افزار و نرم‌افزار را به‌طور تفکیک ناپذیر با هم تلفیق می‌کنند.

۸. نشانگان ممکن است ناشی از عللی باشد که در میان چند وظیفه‌ای توزیع شده باشند که در پردازنده‌های متفاوت اجرا می‌شوند.

در اثنای اشکال‌زدایی، بعضی از خطاها جدی‌اند (مثل خروجی با قائب نادرست) و بعضی دیگر فاجعه‌بارند (مثل شکست سیستم، که باعث آسیب جسمی یا اقتصادی می‌شود). با افزایش پیامدهای یک خطا، مقدار فشار برای یافتن علت آن نیز افزایش می‌یابد. غالباً فشار، سازنده نرم‌افزار را وادار می‌سازد تا یک خطا را برطرف کند و در همان حال دو خطا را وارد کند.

چرا اشکال‌زدایی این قدر دشوار است؟

«همه می‌دانند که اشکال‌زدایی دو برابر دشوارتر از نوشتن یک برنامه جدید است. پس اگر به همان زرنگی هستید که هنگام نوشتن برنامه بودید، آن را چگونه اشکال‌زدایی می‌کنید؟»

برایان کرنیگان

۱۷-۸-۲ ملاحظات روان‌شناختی

متأسفانه، به‌نظر می‌رسد مدارکی وجود داشته باشد مبنی بر اینکه هنر اشکال‌زدایی یکی از ویژگی‌های ذاتی انسان است. برخی انسان‌ها در آن مهارت دارند و برخی ندارند. گرچه شواهد تجربی درباره اشکال‌زدایی را به گونه‌های فراوان می‌توان تفسیر کرد، توانایی اشکال‌زدایی در برنامه‌نویسانی با دانش تحصیلی و تجربی یکسان، به میزان گسترده‌ای متفاوت بوده است.

اشنایدرمن [Sch80] در توضیح جنبه‌های بشری اشکال‌زدایی می‌گوید:

> اشکال‌زدایی یکی از بخش‌های ناراحت کننده برنامه‌نویسی است. دارای عناصر حل مسأله است و در ضمن این احساس ناراحت کننده هم با آن همراه است که اشتباهی صورت گرفته است. افزایش ناراحتی و بی‌میلی نسبت به پذیرش امکان وجود خطاها، باعث افزایش دشواری وظایف می‌شود. خوشبختانه، وقتی اشکال برطرف می‌شود، احساس راحتی زیادی حاصل می‌شود و از تنش کاسته می‌شود.

گرچه «یادگیری» اشکال‌زدایی می‌تواند دشوار باشد، می‌توان برای حل این مشکل چند روش پیشنهاد کرد که آنها را در بخش بعد بررسی خواهیم کرد.

SafeHome

اشکال‌زدایی

صحنه: کابین اد همچنان که کدنویسی و آزمون ادامه می‌یابد.

نقش‌آفرینان: اد و شکیرا- اعضای تیم نرم‌افزاری *SafeHome*

گفتگو:

شکیرا (در حالی که سرش را داخل کابینت کرده است): هی... وقت ناهار کجا بودی؟

اد: همین جا... داشتم کار می‌کردم.

شکیرا: در مانده به نظر می‌رسی... مشکل چیست؟

اد (آهی بلند می‌کشد): از وقتی که این اشکال را کشف کرده‌ام یعنی از ۹:۳۰ صبح تا الآن دارم روی آن کار می‌کنم که ساعت ۲:۴۵ است و هنوز هیچ سرنخی دستم نیامده.

شکیرا: فکر کردم همه موافقت کردیم که بیشتر از یک ساعت را صرف اشکال‌زدایی به تنهایی نکنیم، و بعد از بقیه کمک بگیریم نه؟

اد: بله، ولی...

شکیرا (قدم به درون کابین می‌گذارد): پس مشکل چیست؟

اد: پیچیده است و به علاوه، من حدود پنج ساعت روی آن کار کرده‌ام و تو نمی‌توانی در پنج دقیقه آن را بفهمی.

شکیرا: حالا بگو مسأله چی هست.

[اد مسأله را برای شکیرا توضیح می‌دهد و او سی ثانیه به آن نگاه می‌کند بدون این که حرفی بزند و بعد...]

شکیرا (در حالی که لبخندی بر لبانش می‌نشیند): آهان، این‌جاست: متغیری که نامش SetAlarmCondition است نباید قبل از شروع حلقه برابر «false» قرار داده شود؟

[اد با ناباوری به صفحه خیره می‌شود، به جلو خم می‌شود و با ملایمت بر سر خود می‌زند و شکیرا در حالی که اکنون کاملاً لبخند بر صورتش نشسته بلند می‌شود و از کابین بیرون می‌رود.]

۱۷-۸-۳ روش‌های اشکال‌زدایی

هر روشی که پیش گرفته شود، اشکال‌زدایی یک هدف اصلی را دنبال می‌کند: یافتن و تصحیح علت یک خطای نرم‌افزاری. این هدف توسط ترکیبی از ارزیابی سیستماتیک، نبوغ و شانس قابل دستیابی است. برادلی [Bra85] روش اشکال‌زدایی را به شیوه زیر توصیف می‌کند:

> اشکال‌زدایی، کاربردی صریح از روش علمی است که طی بیش از ۲۵۰۰ سال تکامل یافته است. مبنای اشکال‌زدایی، یافتن منبع (علت) مشکل از طریق افراز دودویی و از طریق فرضیات کاری است که مقادیر جدیدی را که باید بررسی شوند، پیش‌بینی می‌کند.

یک مثال ساده‌ی غیر نرم‌افزاری را در نظر بگیرید: یکی از چراغ‌های منزل کار نمی‌کند. اگر هیچ‌یک از وسایل برقی منزل کار نکند، علت ممکن است در مدار اصلی یا قطع برق باشد. نگاهی به منازل مجاور می‌اندازیم تا ببینیم آیا آنجا هم تاریک است. لامپ را در داخل یک سوکت سالم قرار می‌دهیم و وسیله سالم را درون پریز قرار می‌دهیم و به همین ترتیب فرضیات و آزمون‌ها را تغییر می‌دهیم.

اندرز

برای مدتی که می‌خواهید صرف اشکال‌زدایی از یک مسأله کنید یک محدودیت زمانی، مثلاً دو ساعت، تعیین کنید. پس از آن کمک بگیرید.

ابزارهای نرم‌افزاری

اشکال‌زدایی

هدف: این ابزارها برای آن‌ها که باید مشکلات نرم‌افزار را بر طرف کنند، کمک‌های خودکار فراهم می‌سازند. هدف این ابزارها فراهم آوردن دیدی است که ممکن است رسیدن به آن با اجرای فرایند اشکال‌زدایی به‌صورت دستی، دشوار باشد.

مکانیک: اکثر ابزارهای اشکال‌زدایی، خاص محیط و زبان برنامه نویسی‌اند.

ابزارهای نمونه

Borland Gauntlet که توسط Borland (**www.borland.com**) توزیع می‌شود، به آزمون و نیز به اشکال‌زدایی کمک می‌کند.

Coverty Prevent SQS، که توسط Coverty (**www.coverty.com**) توسعه یافته است، برای C++ و نیز Java کمک فراهم می‌کند.

C++ Test، که توسط Parasoft (**www.parasoft.com**) توسعه یافته است، یک ابزار آزمون واحدهاست که گستره‌ی کاملی از آزمون‌ها را روی کدهای C و C++ پشتیبانی می‌کند. ویژگی‌های اشکال‌زدایی به عیب یابی از خطاهای یافته شده کمک می‌کند.

Code Media، که توسط New Planet Software (**www.newplanetsoftware.com/media/**) توسعه یافته است، یک واسط گرافیکی برای اشکال‌زدایی استاندارد UNIX، یعنی gdb، فراهم می‌آورد و مهم‌ترین ویژگی‌های آن‌را پیاده‌سازی می‌کند Gdb در حال حاضر C/C++، Java، PalmOS، انواع سیستم‌های تعبیه‌شده، زبان‌اسمبلی، FORTRAN و Modula-2 را پشتیبانی می‌کند.

GNATS، یک برنامه کاربردی رایگان (www.gnu.org/software/gnat) که مجموعه‌ای از ابزارها برای ردگیری گزارش‌های اشکال است.

به‌طور کلی، سه روش اشکال‌زدایی قابل پیشنهاد است [Mye79]: (۱) جستجوی جامع (brute force)، (۲) عقبگرد (back tracking)، و (۳) حذف علت (cause elimination). هر یک از این راهبردها را می‌توان به‌صورت دستی اجرا کرد، ولی ابزارهای اشکال‌زدایی مدرن می‌توانند به مراتب بر اثربخشی فرایند بیفزایند.

تاکتیک‌های اشکال‌زدایی. گروه جستجوی جامع برای اشکال‌زدایی، احتمالاً متداول‌ترین و کم‌اثربخش‌ترین روش برای جدا کردن دلیل خطای نرم‌افزار است. روش‌های اشکال‌زدایی جستجوی جامع را هنگامی به‌کار ببرید که همه‌ی روش‌های دیگر به شکست می‌انجامد. با به‌کارگیری این فلسفه که «بگذار کامپیوتر خطاها را بیابد»، حافظه تخلیه می‌شود و برنامه با دستورات خروجی بسار می‌شود. شما امیدوارید که جایی در باتلاق اطلاعات تولید شده بتوانید سرنخی بیابید که به علت خطا منجر شود. گرچه توده‌ی اطلاعات تولید شده ممکن است سرانجام به موفقیت منجر گردد، با فراوانی بیشتری به اتلاف تلاش و زمان می‌انجامد. ابتدا باید اندیشه را به‌کار برد!

عقبگرد یک روش اشکال‌زدایی نسبتاً متداول است که در برنامه‌های کوچک اغلب موفق است. با شروع از محلی که نشانگان کشف شده است، کد منبع، رو به عقب و به‌طور دستی مورد ردگیری قرار می‌گیرد تا محل علت خطا پیدا شود. متأسفانه با افزایش تعداد خطوط کد، تعداد مسیرهای رو به عقب ممکن است افزایش یابد.

«نخستین گام در ترمیم یک برنامه خراب، این است که آن را وادارید تا یارها به شکست بینجامد (با ساده‌ترین مثال ممکن).»

ت. داف

روش سوم – *حذف علت* – توسط استقرا یا استنتاج بیان می‌شود و مفهوم افراز دودویی را وارد عمل می‌کند. داده‌های مرتبط، با رخدادن خطا سازماندهی می‌شوند تا علت‌های بالقوه مشخص شوند. یک «فرضیه علت» عنوان می‌شود و از داده‌های فوق برای اثبات یا انکار فرضیه استفاده می‌شود. به‌طرق دیگر، فهرستی از کلیه علل احتمالی تهیه شده آزمون‌هایی برای حذف هر کدام از آنها طراحی و اجرا می‌شود. اگر آزمون‌های اولیه نشان دادند که فرضیه‌ای در مورد یک علت درست است، داده‌ها مورد پالایش قرار می‌گیرند تا اشکال برطرف شود.

اشکال‌زدایی خودکار. هر کدام از این رویکردهای اشکال‌زدایی را می‌توان با ابزارهای اشکال‌زدایی تکمیل کرد که می‌توانند شما را در اجرای راهبردهای اشکال‌زدایی یاری دهند و آن را نیمه خودکار سازند. هالپیرن و سانتا نام [Hai02]، وضعیت این ابزارها را چنین خلاصه می‌کنند: «... رویکردهای جدید فراوانی پیشنهاد شده‌اند و محیط‌های بسیاری برای اشکال‌زدایی به‌صورت تجاری در دسترس قرار دارند. محیط‌های توسعه‌ی منسجم (IDEها) برای تعیین برخی خطاهای از پیش تعیین شده که خاص زبان برنامه‌نویسی هستند، (مثلاً کاراکترهای انتهای دستور که جا افتاده‌اند، متغیرهای تعریف‌نشده، و غیره) راهی فراهم می‌آورند، بدون این که نیاز به کامپایل کردن برنامه باشد.» تنوع گسترده‌ای از کامپایلرهای اشکال زدا، کمک‌های اشکال‌زدایی پویا، مولدهای مورد آزمون و ابزارهای نگاشت ارجاع متقاطع در دسترس هستند. ولی، ابزارها جایگزینی برای ارزیابی دقیق بر اساس یک مدل طراحی کامل و کد منبع واضح به شمار نمی‌روند.

عامل انسانی. هرگونه بحث درباره ابزارها و رویکردهای اشکال‌زدایی بدون ذکر یک متحد پرقدرت– یعنی آدم‌های دیگر– ناقص است! دیدگاهی تازه، می‌تواند شگفتی بیافریند.[1] در نهایت می‌توان این قاعده‌ی کلی را به‌کار برد که «هر گاه همه‌ی راهکارهای دیگر به شکست انجامید، کمک بگیرید!»

۱۷-۸-۴ تصحیح خطاها

هنگامی که اشکالی پیدا شد، باید تصحیح شود. ولی چنان‌که پیش از این متذکر شدیم، تصحیح یک اشکال می‌تواند خطاهای دیگری را وارد کند و درنتیجه زیان‌های بیشتری وارد کند. فان فلک [Van89] سه پرسش ساده مطرح می‌کند که هر مهندس نرم‌افزار پیش از تصحیح خطا باید از خود بپرسد:

۱. آیا علت اشکال ایجاد شده در بخش دیگری از برنامه قرار دارد؟ در بسیاری از شرایط ، نقص برنامه ناشی از یک الگوی منطقی اشتباه است که ممکن است در جایی دیگر بازسازی شود. بررسی واضح الگوی منطقی ممکن است به کشف خطاهای دیگر منجر شود.

۲. با رفع اشکالی که در حال انجام آن هستم، چه اشکال دیگری ممکن است بروز کند؟ پیش از اینکه تصحیح انجام شود، کد منبع (یا بهتر، طراحی) باید مورد ارزیابی قرار گیرد تا ارتباط میان ساختمان داده‌ها و منطق روشن شود. اگر قرار است تصحیح در بخشی از برنامه صورت پذیرد که ارتباط زیادی میان این ساختارها وجود دارد، دقت فراوان لازم است.

> «بهترین آزمون‌گر، کسی نیست که بیشترین اشکال‌ها را یابد، بهترین آزمون‌گر کسی است که بیشترین تعداد اشکال‌ها را برطرف سازد.»
>
> بسم کانر و سایرین

[1] مفهوم برنامه‌نویسی جفتی (که به‌عنوان بخشی از مدل برنامه‌نویسی حدی توصیه شده است و در فصل ۳ بحث شد) سازوکاری برای «اشکال‌زدایی» به موازات طراحی و کدنویسی نرم‌افزار فراهم می‌آورد.

۳. برای جلوگیری از این اشکال چه می‌توانسته‌ایم انجام دهیم؟ این پرسش، نخستین گام در جهت تثبیت یک روش تضمین کیفیت آماری است (فصل ۸). اگر فرایند و نیز محصول را تصحیح کنیم، اشکال از برنامه فعلی حذف می‌شود و ممکن است در برنامه‌های بعدی نیز وجود نداشته باشد.

۱۷-۹ خلاصه

آزمون نرم‌افزار بیشترین کار فنی را در فرایند نرم‌افزاری طلب می‌کند. هنوز در ابتدای راه درک ظرافت‌های برنامه‌ریزی ، اجرا و کنترل آزمون‌های سیستماتیک هستیم.

هدف آزمون نرم‌افزار، کشف خطاهاست. برای نیل به این مقصود درخصوص نرم‌افزارهای سنتی، چند مرحله آزمون مورد نیاز است. در آزمون‌های انسجام و واحدها، وارسی بر عملکرد انفرادی پیمانه‌ها و گروههایی آنها در قالب ساختار برنامه کانون توجه قرار می‌گیرد. آزمون اعتبارسنجی، قابلیت ردگیری را تا حدّ خواسته‌های نرم‌افزار نشان می‌دهد و آزمون سیستم، نرم‌افزار را پس از قرار گرفتن در یک سیستم بزرگتر اعتبارسنجی می‌کند. هر مرحله از آزمون، از طریق تکنیک‌های آزمون سیستماتیک قابل انجام است که به طراحی موارد آزمون کمک می‌کنند. با هر مرحله از آزمون، سطح انتزاع نگرش به نرم‌افزار وسعت می‌یابد.

راهبرد مربوط به آزمون نرم‌افزارهای شیءگرا با آزمون‌هایی آغاز می‌گردد که عملیات‌های درون یک کلاس را تمرین می‌دهند و سپس به سمت آزمون نخ‌ها برای انسجام حرکت می‌کند. نخ‌ها مجموعه‌ای از کلاس‌ها هستند که با سایر کلاس‌ها همکاری سنگینی ندارند.

برنامه‌های تحت وب تا حد زیادی مشابه با سیستم‌های شیءگرا آزمون می‌شوند، ولی آزمون‌ها طوری طراحی می‌شوند که محتویات، قابلیت‌های عملیاتی، واسط، گشت‌وگذار و جنبه‌های کارایی و امنیتی برنامه‌ی تحت وب تمرین داده شوند.

راهبرد مربوط به آزمون نرم‌افزارهای شیءگرا با آزمون‌هایی آغاز می‌گردد که عملیات‌های درون یک کلاس را تمرین می‌دهند و سپس به سمت آزمون نخ‌ها برای انسجام حرکت می‌کند. نخ‌ها مجموعه‌ای از کلاس‌ها هستند که با سایر کلاس‌ها همکاری سنگینی ندارند.

برنامه‌های تحت وب تا حد زیادی مشابه با سیستم‌های شیءگرا آزمون می‌شوند، ولی آزمون‌ها طوری طراحی می‌شوند که محتویات، قابلیت‌های عملیاتی، واسط، گشت‌وگذار و جنبه‌های کارایی و امنیتی برنامه‌ی تحت وب تمرین داده شوند.

برخلاف آزمون (سیستماتیک و با طراحی فعالیتی)، اشکال‌زدایی را باید یک هنر پنداشت. فعالیت اشکال‌زدایی باید با در نظر گرفتن نشانگان مشکل، علت خطا را بیابد. مشاوره با اعضای دیگر تیم نرم‌افزاری، مهمترین منبع برای اشکال‌زدایی است.

مسائل و نکاتی برای تعمق

۱-۱۷ تفاوت میان وارسی و اعتبارسنجی را به زبان ساده بیان کنید. آیا هر دو از راهبردهای آزمون و روش‌های طراحی موارد آزمون استفاده می‌کنند؟

۲-۱۷ مشکلاتی را بیان کنید که ممکن است در ایجاد یک گروه آزمون مستقل وجود داشته باشد. آیا گروه ITG و گروه SQA را افراد یکسان تشکیل می‌دهند؟

۱۷-۳ آیا همواره می‌توان راهبردی برای آزمون نرم‌افزار توسعه داد که از مراحل شرح داده شده در بخش ۱۷-۱-۳ استفاده کنند؟ برای سیستم‌های تعبیه شده چه نتایجی ممکن است به بار آید؟

۱۷-۴ چرا اجرای آزمون واحد برای پیمانه‌های مرتبط دشوار است؟

۱۷-۵ مفهوم «ضداشکال» (بخش ۱۷-۳-۱) شیوه‌ای است بسیار اثربخش برای فراهم‌سازی کمک‌های اشکال‌زدایی درونی، برای هنگامی که خطا کشف نشده باشد:

(الف) یک مجموعه دستورالعمل برای ضداشکال‌سازی تهیه کنید.

(ب) مزایای استفاده از این تکنیک را شرح دهید.

(پ) معایب آن را شرح دهید.

۱۷-۶ زمان‌بندی پروژه چگونه می‌تواند بر آزمون انسجام تأثیر بگذارد؟

۱۷-۷ آیا آزمون واحدها در همه‌ی شرایط امکان‌پذیر یا حتی مطلوب است؟ برای توجیه پاسخ خود مثالی بیاورید.

۱۷-۸ چه کسی باید آزمون اعتبارسنجی را انجام دهد - سازنده نرم‌افزار یا کاربر آن؟ برای پاسخ خود دلیل بیاورید.

۱۷-۹ برای سیستم *SafeHome* که در فصول قبلی کتاب معرفی شد، یک راهبرد آزمون کامل توسعه دهید. آن را در یک مشخصه آزمون مستندسازی کنید.

۱۷-۱۰ به‌عنوان یک پروژه کلاسی، یک راهنمای اشکال‌زدایی برای خود تهیه کنید. این راهنما باید زبان و تذکرات سیستم‌گرایی را که در مدرسه آموخته‌اید در بر داشته باشد، کار خود را با خلاصه‌ای از مباحث که باید توسط شما در کلاس و استادتان مرور شوند، آغاز کنید. این راهنما را در محیط محلی خود برای دیگران منتشر کنید.

فصل ۱۸

آزمون برنامه‌های کاربردی سنتی

نگاهی گذرا

آزمون نرم‌افزار چیست؟ هنگامی که کد منبع تولید شد، نرم‌افزار باید آزموده شود تا بتوان هر تعدادی از خطاها را که امکان داشته باشد، قبل از تحویل به مشتری کشف کرد. هدف، طراحی موارد آزمون است که احتمال یافتن خطا را بالا ببرند - ولی چگونه؟ اینجاست که تکنیک‌های آزمون نرم‌افزار وارد صحنه می‌شوند. این تکنیک‌ها راهنمایی سیستماتیک برای طراحی آزمون‌هایی به‌دست می‌دهند که: (۱) با منطق داخلی مؤلفه‌های نرم‌افزار تمرین می‌کنند و (۲) دامنه‌های داخلی و خارجی برنامه را تمرین می‌کنند تا خطاهای موجود در عملکرد، رفتار و کارآیی آن کشف شود.

چه کسی آن را انجام می‌دهد؟ طی مراحل اولیه آزمون، مهندس نرم‌افزار کلیه آزمون‌ها را انجام می‌دهد. ولی، به‌موازات پیشرفت فرایند آزمون، ممکن است کارشناسان آزمون نیز در این امر شرکت کنند.

چرا اهمیت دارد؟ بازبینی‌ها و فعالیت‌های SQA دیگر می‌توانند خطاها را کشف کنند و می‌کنند، ولی کافی نیستند. هر بار که برنامه اجرا شود، مشتری آن را می‌آزماید! بنابراین، باید برنامه را پیش از آنکه به‌دست مشتری برسد با هدف یافتن و حذف خطاها اجرا نمود. برای یافتن حداکثر تعداد ممکن خطاها، باید آزمون‌هایی را به‌طور سیستماتیک اجرا کرد و موارد آزمودنی را با استفاده از تکنیک‌های منظم، طراحی نمود.

مراحل کار کدام است؟ نرم‌افزار از دو دیدگاه متفاوت آزمایش می‌شود: (۱) با استفاده از تکنیک‌های طراحی موارد آزمون، با منطق داخلی برنامه (جعبه سفید) تمرین می‌شود. (۲) با خواسته‌های نرم‌افزار با استفاده از تکنیک‌های طراحی مورد آزمون «جعبه سیاه» تمرین می‌شود. در هر دو حال، هدف، یافتن حداکثر تعداد خطاها با حداقل مقدار کار و زمان است.

محصول کاری چیست؟ مجموعه‌ای از موارد آزمون که برای تمرین با منطق داخلی و خواسته‌های خارجی، طراحی و مستندسازی شده‌اند؛ نتایج مورد انتظار تعیین و نتایج واقعی ثبت می‌شوند.

چگونه مطمئن شوم که درست از عهده کارها برآمده‌ام؟ هنگامی که آزمون را شروع می‌کنید، دیدگاه خود را تغییر دهید. موارد آزمون را به شیوه‌ای منظم طراحی کنید و آنها را مرور کنید تا مطمئن شوید که کامل هستند.

آزمون برای مهندسان نرم‌افزار، که فطرتاً افرادی توسعه‌دهنده هستند، امری غیرعادی تلقی می‌شود. آزمون، مستلزم آن‌است‌که توسعه‌دهنده دید پیش‌داوری خود، مبنی بر درستی نرم‌افزار را دور بریزد و بر تناقض جالبی که از کشف خطاها عارض می‌شود، غلبه کند. بیزر [Bei90] این وضعیت را به‌خوبی شرح می‌دهد:

> پندار باطلی وجود دارد مبنی بر اینکه اگر واقعاً در برنامه‌نویسی استاد باشیم، اشکالی وجود نخواهد داشت. اگر فقط می‌توانستیم واقعاً حواس خود را جمع کنیم، اگر فقط همه از برنامه‌نویسی ساخت‌یافته، طراحی بالا به پایین و جداول تصمیم‌گیری استفاده می‌کردند، ... هیچ اشکالی پیش نمی‌آمد و این پندار باطل همچنان ادامه دارد. این پندار باطل می‌گوید اشکالات از آن رو وجود دارند که ما کارها را خوب انجام نمی‌دهیم و اگر خوب انجام نمی‌دهیم باید درباره آن احساس گناه کنیم. بنابراین، آزمودن و طراحی موارد آزمون، پذیرش شکست است که رفته‌رفته به پذیرش گناه می‌انجامد. کسالتِ امر آزمون، تنبیهی برای خطاهای ماست. تنبیه برای چه؟ برای انسان؟ گناه برای چه؟ برای شکست در دستیابی به کمالات فرانسانی؟ برای اینکه نتوانیم بین آنچه که یک برنامه‌نویس دیگر می‌گوید و می‌اندیشد تمایز قائل شویم؟ برای حل نکردن مشکلات ارتباطی انسانی که قرن‌هاست لاینحل مانده است؟

آیا آزمون باید به پذیرش گناه بینجامد؟ آیا آزمون واقعاً ویران‌گر است؟ پاسخ این پرسش‌ها منفی است.

در این فصل، به بحث درباره‌ی تکنیک‌های مربوط به طراحی موارد آزمون نرم‌افزار برای برنامه‌های کاربردی سنتی خواهیم پرداخت. در طراحی موارد آزمون، مجموعه‌ای از تکنیک‌های مربوط به ایجاد موارد آزمون کانون توجه قرار می‌گیرد که اهداف کلی آزمون و راهبردهای آزمون بحث شده در فصل ۱۷ را برآورده می‌سازد.

۱-۱۸ مبانی آزمون نرم‌افزار

هدف آزمون، یافتن خطاهاست و آزمون خوب، آزمونی است که احتمال یافتن خطا را بالا می‌برد. بنابراین، در طراحی و پیاده‌سازی یک محصول یا سیستم کامپیوتری باید «آزمون‌پذیری» را مد نظر داشت. در همان حال، آزمون‌ها خودشان باید مجموعه‌ای از خصوصیات را از خود به نمایش بگذارند که هدفِ یافتن اکثر خطاها با حداقل تلاش را برآورده سازند.

آزمون‌پذیری. جیمز بَک[1]، این تعریف را برای آزمون‌پذیری ارائه می‌دهد: «*آزمون‌پذیری* نرم‌افزار صرفاً عبارت است از این که [یک برنامه کامپیوتری] را چقدر آسان می‌توان آزمایش کرد.» خصوصیات زیر به نرم‌افزار آزمون‌پذیر منجر می‌گردد.

قابلیت کارکردن (Operability). «هر چه بهتر کار کند، آزمون آن اثربخش‌تر است.» اگر سیستم با مدنظر قراردادن کیفیت، طراحی و پیاده‌سازی شود، اشکال‌های نسبتاً معدودی سد راه اجرای آزمون‌ها می‌شود و پیشرفت آزمون‌ها بدون تلاش زیاد میسر خواهد شد.

قابلیت مشاهده (Visibility). «آنچه می‌بینید، همان است که آزمایش می‌کنید.» برای هر ورودی، یک خروجی متمایز تولید می‌شود. متغیرها و حالت‌های سیستم در اثنای اجرا قابل مشاهده و استفسارند. خروجی نادرست به آسانی قابل شناسایی است. خطاهای درونی به‌طور خودکار آشکار و گزارش می‌شوند. کد منبع قابل دستیابی است.

«هر برنامه‌ای به هر حال یک کار درست انجام می‌دهد ولی ممکن است آن چیزی نباشد که ما می‌خواهیم.»
ناشناس

خصوصیات آزمون‌پذیری چیست؟

[1] پاراگراف‌هایی که بعدنبال خواهند آمد، با کسب اجازه از جیمز بَک آورده شده‌اند.

کنترل‌پذیری (Controllability). «هر چه بهتر بتوان نرم‌افزار را کنترل کرد، آزمون را بیشتر می‌توان خودکار و بهینه کرد.»

همه‌ی خروجی‌های ممکن را می‌توان از طریق ترکیبی از ورودی‌ها تولید نمود و فرمت I/Oها سازگار و ساخت‌یافته است. همه‌ی کدها از طریق ترکیبی از ورودی‌ها قابل اجرا هستند. متغیرها و حالت‌های سخت‌افزاری و نرم‌افزاری مستقیماً توسط مهندس آزمون قابل کنترل هستند. آزمون‌ها را به‌راحتی می‌توان مشخص، خودکار و بازسازی کرد.

تجزیه‌پذیری (Decomposability). «با کنترل دامنه کاربرد آزمون، می‌توان مسائل را سریع‌تر جداسازی کرد و آزمون‌های مجدد را با هوشمندی بیشتر انجام داد.» سیستم نرم‌افزاری از پیمانه‌های مستقلی ساخته می‌شود که آن‌ها را می‌توان مستقل از هم آزمود.

سادگی (Simplicity). «هر چه مورد آزمون کوچکتر باشد، سریع‌تر می‌توان آن را آزمود.» برنامه باید دارای *سادگی عملیاتی* (مثلاً مجموعه ویژگی‌ها، حداقل مجموعه لازم برای برآوردن خواسته‌هاست)، *سادگی ساختاری* (مثلاً معماری به صورت پیمانه‌ای در می‌آید تا انتشار خطاها را به حداقل برساند) و *سادگی کد* (مثلاً یک استاندارد کدنویسی رعایت می‌شود که وارسی و نگهداری آن آسان باشد) باشد.

پایداری (Stability). «هر چه تعداد تغییرات کمتر باشد، آزمون کمتر با مانع مواجه می‌شود.» تغییرات نرم‌افزار چندان زیاد نیست، درصورت رخ دادن، کنترل شده هستند و آزمون‌های موجود را بی‌اعتبار نمی‌کنند. نرم‌افزار به‌خوبی از پس شکست‌ها برمی‌آید.

درک‌پذیری (Understandability). «هرچه اطلاعات بیشتری داشته باشیم، آزمون هوشمندانه‌تر انجام می‌شود.» طراحی و وابستگی میان مؤلفه‌های داخلی، خارجی و مشترک به خوبی درک شده است. مستندات فنی بلافاصله قابل دستیابی‌اند، به خوبی سازمان یافته‌اند، مشخص و مفصل هستند و از صحت کافی برخوردارند. تغییرات به عمل آمده در طراحی به اطلاع آزمون گران رسانده می‌شود.

می‌توانید از صفات پیشنهادی بَک استفاده کنید و یک پیکربندی نرم‌افزار (شامل برنامه، داده‌ها و مستندات) توسعه دهید که مستعد آزمون باشد.

«خطاها در نرم‌افزارها، متداول‌تر، شایع‌تر و مشکل‌آفرین‌تر از خطاهای موجود در سایر فن‌آوری‌ها هستند.»
دیوید پارناس

خصوصیات آزمون. درباره خودِ آزمون‌ها چه می‌توان گفت؟ کینر، فالک و نگوین [Kan93] برای آزمون «خوب» صفات زیر را برمی‌شمرند:

آزمون «خوب» چیست؟

آزمون خوب با احتمال زیادی خطاها را می‌یابد. برای حصول این منظور، آزمون‌گر باید نرم‌افزار را بشناسد و کوشش کند تا یک تصویر ذهنی از چگونگی شکست احتمالی نرم‌افزار بسازد. به‌طور ایده‌آل، دسته‌هایی از شکست بررسی می‌شوند. برای مثال یک دسته از شکست‌های بالقوه در واسط گرافیکی کاربر، شکست در تشخیص موقعیت ماوس است. برای تمرین دادن ماوس به‌منظور نشان‌دادن خطایی در تشخیص موقعیت ماوس، یک مجموعه آزمون طراحی می‌شود.

آزمون خوب دارای زواید نیست. زمان و منابع آزمون، محدود است. در اجرای آزمونی که هدف آن با هدف یک آزمون دیگر یکی است، هیچ نکته‌ای وجود ندارد. هر یک از آزمون‌ها باید دارای هدفی متفاوت باشد (حتی اگر این تفاوت ظریف باشد).

آزمون خوب باید «بهترین» باشد [Kan93]. در گروهی از آزمون‌ها که هدف و قصدی مشابه دارند، محدودیت‌های منابع و زمانی ممکن است فقط با اجرای زیرمجموعه‌ای از این آزمون‌ها تعدیل شوند. در چنین مواردی، آزمونی به‌کار گرفته می‌شود که با احتمال بیشتری خطا را می‌یابد.

SafeHome

طراحی آزمون‌های منحصر به فرد

صحنه: کابین ویندو.

نقش‌آفرینان: ویندو و اد- اعضای تیم نرم‌افزاری *SafeHome*

گفتگو:

ویندو: پس این‌ها موارد آزمونی هستند که خیال داری برای عملیات *passwordValidation* اجرا کنی.

اد: بله، گمان کنم همه‌ی حالت‌های ممکن برای انواع کلمات عبوری که کاربرد وارد می‌کند، پوشش بدهد.

ویندو: خب، بگذار ببینم... کلمه‌ی عبور درست، ۸۰۸۰ است نه؟

اد: آهان.

ویندو: تو هم برای آزمایش خطا در تشخیص کلمات عبور نامعتبر، دو کلمه‌ی عبور ۱۲۳۴ و ۶۷۸۹ را مشخص کردی؟

اد: درست است و البته کلمات عبور نزدیک به کلمه‌ی عبور درست مثل ۸۰۸۱ و ۸۱۸۰ را هم آزمایش می‌کنم.

ویندو: این‌ها خوب هستند ولی من نکته خاصی در واردکردن هر دو ورودی ۱۲۳۴ و ۶۷۸۹ نمی‌بینم. این‌ها اضافی هستند... چون هر دو یک چیز را آزمایش می‌کنند؛ نه؟

اد: خب، این‌ها دو تا مقدار متفاوتند.

ویندو: درست. ولی ۱۲۳۴ هیچ خطایی را آشکار نمی‌کند... به عبارت دیگر... عملیات *passwordValidation* متوجه می‌شود که کلمه‌ی عبور درست نیست، این احتمال وجود ندارد که ۶۷۸۹ چیز جدیدی را نشان دهد.

اد: متوجهم، چه منظوری داری.

ویندو: نمی‌خواهم خیلی فضولی کنم، ولی وقت آزمون خیلی کم است. پس بهتر است آزمون‌هایی را اجرا کنیم که احتمال یافتن خطاهای جدید در آن‌ها بالا باشد.

اد: مشکلی نیست... بیشتر روی آن فکر می‌کنم.

آزمون خوب نباید بیش از حد ساده و نه بیش از حد پیچیده باشد. گرچه ممکن است تعدادی آزمون را در یک آزمون خلاصه کرد، اثرات جانبی این روش ممکن است خطاها را نادیده بگیرد. به‌طور کلی، هر آزمون باید جداگانه اجرا شود.

۲-۱۸ دیدگاه‌های درونی و بیرونی نسبت به آزمون

هر محصول مهندسی شده (و اکثر چیزهای دیگر) را می‌توان به یکی از دو شیوه آزمود: (۱) با دانستن قابلیت عملیاتی مشخصی که یک محصول برای ارائه آن طراحی شده است، می‌توان آزمون‌هایی اجرا

کرد که هر قابلیت به‌طور کامل عملیاتی شود، در حالی که به‌طور هم زمان، جستجو به دنبال خطاهای موجود در هر قابلیت، انجام می‌شود. (۲) با دانستن کارکرد درونی یک محصول، می‌توان آزمون‌هایی اجرا کرد که اطمینان حاصل شود همه چیز به خوبی پیش می‌رود، یعنی عملیات‌های درونی مطابق با مشخصات اجرا می‌شوند و همه‌ی مؤلفه‌های درونی به‌طور مناسب و به قدر کافی تمرین داده شده‌اند. در رویکرد نخست در آزمون، دیدگاهی بیرونی مد نظر است که آزمون جعبه سیاه نامیده می‌شود. رویکرد دوم نیاز به دیدگاهی درونی دارد و در اصطلاح از آن به‌عنوان آزمون جعبه سفید یاد می‌شود.[1]

آزمون جعبه‌ی سیاه به آزمون‌هایی اشاره دارد که روی واسط نرم‌افزار اجرا می‌شوند. هر آزمون جعبه سیاه، جنبه‌ای عملیاتی از سیستم را بررسی می‌کند، در حالی که ساختار منطقی درونی نرم‌افزار کمتر مورد توجه قرار می‌گیرد. در *آزمون جعبه‌ی سفید*، نرم‌افزار از نظر جزئیات روالی مورد بررسی دقیق قرار می‌گیرد. مسیرهای منطقی از میان نرم‌افزار و همکاری‌های میان مؤلفه‌ها با تمرین‌دادن مجموعه‌های مشخصی از شرایط و/یا حلقه‌ها آزمایش می‌شوند.

در نگاه نخست، ممکن است به نظر برسد که آزمون‌های بسیار کامل به «برنامه‌های صددرصد درست» منجر می‌شوند. همه‌ی آن‌چه که باید کرد، تعریف کلیه‌ی مسیرهای منطقی، تهیه‌ی موارد آزمون برای تمرین‌دادن آن‌ها و ارزیابی نتایج است، یعنی تولید موارد آزمون برای تمرین‌دادن سنگین و فشرده‌ی منطق برنامه. متأسفانه، آزمون سنگین و فشرده، باعث بروز مسائل منطقی معین می‌شود. ولی آزمون جعبه سفید را نباید بیهوده و منتفی دانست. می‌توان تعداد محدودی از مسیرهای منطقی مهم را انتخاب کرد و تمرین داد. ساختمان داده‌های مهم را می‌توان اعتبارسنجی کرد.

> «در طراحی موارد آزمون تنها یک قاعده وجود دارد: همه‌ی ویژگی‌ها را پوشش دهید. پی‌آن‌که موارد آزمون را بیش از حد زیاد کنید.»
>
> تسونو یاماتورا

نکته‌ی کلیدی

آزمون‌های جعبه‌ی سفید را تنها پس از ایجاد طراحی در سطح مؤلفه‌ها (یا کد منبع) می‌توان طراحی نمود زیرا جزئیات منطقی برنامه باید در دسترس باشد.

اطلاعات

آزمون سنگین و فشرده

برنامه‌ای را در نظر بگیرید که در ۱۰۰ خط به زبان C نوشته شده است. بعد از اعلان داده‌ها، برنامه حاوی دو حلقه‌ی تودرتو است که هر کدام، بسته به شرایط مشخص شده در ورودی، از ۱ تا ۲۰ بار اجرا می‌شود. در داخل حلقه‌ی درونی، چهار ساختار if-then-else مورد نیاز است. حدوداً 10^{14} مسیر ممکن است که می‌توان در این برنامه اجرا کرد!

برای آن‌که دیدی از این عدد به‌دست آید، فرض می‌کنیم که یک پردازنده‌ی سحرآمیز (سحرآمیز از آن روی که چنین پردازنده‌ای وجود ندارد) برای آزمون سنگین و فشرده ساخته شده باشد. این پردازنده می‌تواند هر مورد آزمون را در عرض یک میلی‌ثانیه، توسعه دهد، آن را اجرا کند و نتایج را ارزیابی کند. اگر این پردازنده، روزانه ۲۴ ساعت و همه‌ی روزهای سال را کار کند، به ۳۱۷۰ سال زمان برای آزمایش برنامه نیاز خواهد داشت.

بنابراین، منطقی است که بپذیریم آزمون سنگین و فشرده برای سیستم‌های نرم‌افزار بزرگ، عملی نیست.

[1] گاهی به‌جای آزمون جعبه سیاه و جعبه سفید به ترتیب از آزمون عملیاتی و آزمون ساختاری استفاده می‌شود.

۱۸-۳ آزمون جعبه سفید (White Box Texting)

آزمون جعبه سفید که گاه آزمون جعبه شیشه‌ای نیز خوانده می‌شود، یک روش طراحی برای موارد آزمون است که برای به‌دست آوردن موارد آزمون، از ساختار کنترلی در برنامه استفاده می‌کند. مهندس نرم‌افزار با استفاده از متدهای آزمون جعبه سفید می‌تواند موارد آزمونی به‌دست آورد که (۱) تضمین می‌کنند که همه‌ی مسیرهای مستقل در یک پیمانه، حداقل یک بار امتحان شده‌اند؛ (۲) همه‌ی تصمیم‌گیری‌های منطقی را در دو بخش درست و غلط امتحان کنند؛ (۳) همه‌ی حلقه‌ها را در مرزها و در داخل مرزهای عملیاتی آنها اجرا کنند؛ و (۴) ساختمان داده‌های داخلی را امتحان کنند تا اعتبار آنها ثابت شود.

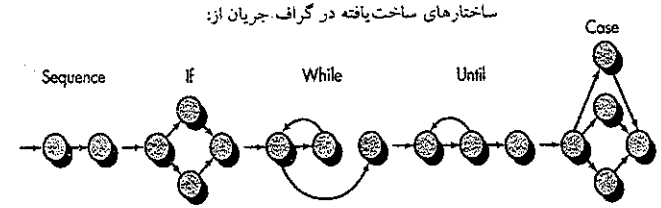
> «اشکال‌ها در گوشه و کنارها در کمین هستند و در نقاط مرزی گرد هم مجتمع می‌شوند»
>
> بوریس بیزر

۱۸-۴ آزمون مسیرهای پایه (Basic Path Tesing)

آزمون مسیرهای پایه، یک تکنیک آزمون جعبه سفید است که نخستین بار توسط تام مک‌کیب [McC76] پیشنهاد شد. روش مسیرهای پایه، طراح موارد آزمون را قادر می‌سازد تا میزانی منطقی از پیچیدگی رویه‌ای به‌دست آورد و از این میزان به‌عنوان راهنمایی جهت تعریف یک مجموعه‌ی پایه از مسیرهای اجرا استفاده کند. موارد آزمون به‌دست آمده برای امتحان‌کردن این مجموعه‌ی پایه، هر دستور از برنامه را حداقل یک بار در اثنای آزمون اجرا خواهند کرد.

۱۸-۴-۱ نمادگذاری گراف جریان (Flow Graph Notation)

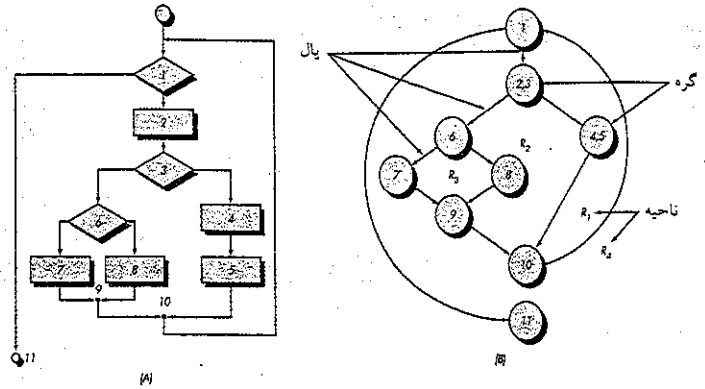
پیش از آن‌که بتوان روش مسیرهای پایه را معرفی نمود، یک نمادگذاری ساده، موسوم به *گراف جریان* یا (*گراف برنامه*) را باید برای نمایش دادن جریان کنترل معرفی کرد[1]. گراف جریان، جریان کنترل منطقی را با استفاده از نمادگذاری در شکل ۱-۱۸ تصویر می‌کند. متناظر با هر ساختار ساخت‌یافته (فصل ۱۰) یک نماد گراف جریان وجود دارد.



شکل ۱-۱۸ نمادگذاری گراف جریان.

برای نشان دادن کاربرد گراف جریان، طراحی رویه‌ای شکل ۲-۱۸الف را در نظر می‌گیریم. در اینجا، از یک نمودار گردشی برای تصویرکردن ساختار کنترلی برنامه استفاده می‌شود. شکل ۲-۱۸ب، نمودار گردشی را به صورت گراف جریان آن در آورده است (با این فرض که هیچ شرط مرکبی در

[1] در حقیقت، روش مسیرهای پایه بدون استفاده از گراف‌های جریان نیز قابل اجراست. ولی این گراف‌ها به‌عنوان یک نمادگذاری مفید به درک جریان کنترل و به‌نمایش درآوردن این روش کمک می‌کنند.



شکل ۲-۱۸ (الف) نمودار گردشی (ب) گراف جریان.

> اندرز
>
> پیچیدگی سیکلوماتیک، معیاری مفید برای پیش‌بینی پیمانه‌هایی است که احتمال مستعد خطا بودن آن‌ها بیشتر است. از آن برای برنامه‌ریزی آزمون‌ها و نیز طراحی موارد آزمون استفاده می‌شود.

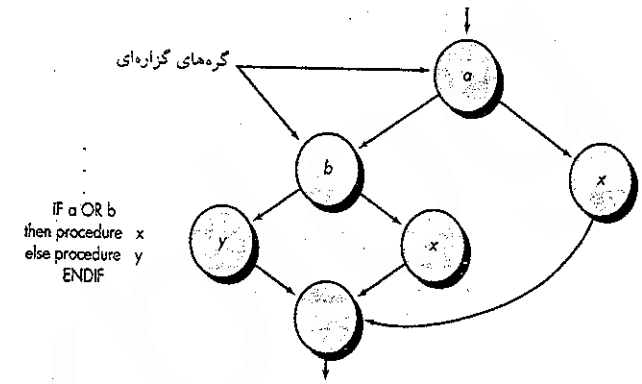
لوزی‌های تصمیم‌گیری نمودار گردشی وجود نداشته باشد). با توجه به شکل ۲-۱۸ب، هر دایره که گره *گراف جریان* خوانده می‌شود، یک یا چند دستور رویه‌ای را نشان می‌دهد. ترتیبی از مستطیل‌های پردازشی و یک لوزی تصمیم‌گیری را می‌توان در یک گره منفرد خلاصه نمود. پیکان‌های روی گراف جریان، که *یال* یا *پیوند* خوانده می‌شوند، نشان‌گر جریان کنترل بوده مشابه پیکان‌های نمودار گردشی هستند. هر رویه باید در یک گره پایان یابد، حتی اگر گره هیچ دستور رویه‌ای را نشان ندهد (مثلاً نماد مربوط به ساختمان if-then-else را ببینید). مساحت‌های محصور شده توسط یال‌ها و گره‌ها را *ناحیه* (region) می‌نامند. هنگام شمارش نواحی، مساحت خارج از گراف را نیز به‌عنوان یک ناحیه در نظر گرفته آن را لحاظ می‌کنیم[1].

هنگام مواجهه با شرط‌های مرکب در یک طراحی رویه‌ای، تولید گراف جریان قدری پیچیده‌تر می‌شود. شرط مرکب زمانی رخ می‌دهد که یک یا چند عمل‌گر بولی (OR، AND، NOR و NAND منطقی) در یک دستور شرطی وجود داشته باشند. در شکل ۳-۱۸، دستور PDL به گراف جریان ترجمه می‌شود. توجه داشته باشید که برای هر یک از شرایط a و b در دستور IF a OR b یک گره جداگانه ایجاد می‌شود. هر گره که حاوی یک شرط باشد، گره *گزاره‌ای* خوانده می‌شود و با دو یا چند پیکان که از آن بیرون می‌آیند، مشخص می‌شود.

۱۸-۴-۲ مسیرهای مستقل برنامه

مسیر مستقل، هر مسیری از برنامه است که حداقل یک مجموعه‌ی جدید از دستورهای پردازش یا یک دستور شرطی را معرفی کند. اگر مسیر مستقل برحسب گراف جریان بیان شود، حداقل باید در راستای یک یال حرکت کند که پیش از تعریف مسیر از آن عبور نشده باشد. برای مثال، مجموعه‌ای از مسیرهای مستقل در شکل ۲-۱۸ب در گراف جریان نشان داده شده‌اند:

[1] بحث مفصل‌تری درباره گراف‌ها و کاربرد آن‌ها در بخش ۱-۶-۱۸ ارائه خواهد شد.



شکل ۳-۱۸ منطق مرکب.

مسیر ۱: 1-11

مسیر ۲: 1-2-3-4-5-10-1-11

مسیر ۳: 1-2-3-6-8-9-10-1-11

مسیر ۴: 1-2-3-6-7-9-10-1-11

توجه دارید که هر مسیر جدید یک یال جدید معرفی می‌کند. مسیر زیر:

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

به‌عنوان مسیری مستقل در نظر گرفته نمی‌شود، زیرا صرفاً تلفیقی از مسیرهای مشخص شده از قبل است و از هیچ یال جدیدی عبور نمی‌کند.

مسیرهای ۱ تا ۴ که در بالا تعریف شده‌اند، یک مجموعه‌ی پایه برای گراف جریان شکل ۲-۱۸ تشکیل می‌دهند. یعنی، اگر آزمون‌ها را بتوان طوری طراحی کرد که اجرای این مسیرها (یک مجموعه‌ی پایه) را حتمی کنند، هر دستور از برنامه حداقل یک بار اجرا خواهد شد و هر دستور شرطی در هر دو حالت درست و نادرست خود اجرا می‌شود. لازم به ذکر است که مجموعه‌ی پایه منحصر به‌فرد نیست. درحقیقت، چند مجموعه‌ی پایه متفاوت را می‌توان برای یک طراحی رویه‌ای مفروض به‌دست آورد.

چگونه باید بدانیم که به دنبال چند مسیر باید بگردیم؟ پاسخ را در محاسبه پیچیدگی سیکلوماتیک می‌توان جستجو کرد.

پیچیدگی سیکلوماتیک، ریشه در نظریه گراف‌ها دارد و یک معیار نرم‌افزاری سودمند را فراهم می‌آورد. پیچیدگی به یکی از سه شیوه‌ی زیر محاسبه می‌شود:

۱. تعداد نواحی گراف جریان متناظر با پیچیدگی سیکلوماتیک.

۲. پیچیدگی سیکلوماتیک، $V(G)$، برای یک گراف جریان G به صورت $V(G) = E - N + 2$ تعریف می‌شود که در آن E تعداد یال‌های گراف جریان و N تعداد گره‌های آن است.

۳. پیچیدگی سیکلوماتیک، $V(G)$ برای یک گراف جریان G به صورت $V(G) = P + 1$ نیز تعریف می‌شود که P تعداد گره‌های گزاره‌ای موجود در گراف جریان G است.

پیچیدگی سیکلوماتیک را چگونه محاسبه می‌کنید؟

SafeHome

استفاده از پیچیدگی سیکلوماتیک

صحنه: کابین شکیرا.

نقش‌آفرینان: وینود و شکیرا- اعضای تیم نرم‌افزاری *SafeHome* که روی برنامه‌ریزی آزمون برای قابلیت امنیتی سیستم کار می‌کنند.

گفتگو:

شکیرا: ببین... من می‌دانم که باید برای همه‌ی مؤلفه‌های قابلیت امنیتی، یک مورد آزمون تهیه کنیم ولی خیلی از این‌ها هست و اگر تعداد عملیات‌هایی را که باید تمرین داده شوند، در نظر بگیری، نمی‌دانم... شاید بهتر باشد آزمون جعبه‌ی سفید را فراموش کنیم، همه چیز را منسجم کنیم و اجرای آزمون‌های جعبه‌ی سیاه را شروع کنیم.

وینود: این را می‌دانی که وقت کافی برای آزمایش‌کردن مؤلفه‌ها و تمرین‌دادن عملیات‌ها نداریم و بعد آن‌ها را منسجم می‌کنی؟

شکیرا: مهلت عرضه‌ی گام اول، نزدیک‌تر از آن چیزی است که... بله، من نگرانم.

وینود: آزمون‌های جعبه سفید را حداقل روی علمیات‌هایی اجرا کن که بیشتر مستعد خطا هستند.

شکیرا (آشفته): و دقیقاً از کجا باید بدانم کدام‌ها بیشتر مستعد خطا هستند؟

وینود: وی جی، $V(G)$.

شکیرا: هان؟

وینود: پیچیدگی سیکلوماتیک- $V(G)$ کافی است $V(G)$ را برای هر عملیات در هر کدام از مؤلفه‌ها محاسبه کنی و ببینی کدام یک بیشترین مقدار $V(G)$ را دارد. این مؤلفه‌ها همان مؤلفه‌هایی هستند که احتمال وجود خطا در آن‌ها بیشتر از همه است.

شکیرا: حالا این $V(G)$ را چطور باید محاسبه کرد؟

وینود: واقعاً آسان است. این کتاب را بخوان.

شکیرا (کتاب را ورق می‌زند): بسیار خب. زیاد سخت به‌نظر نمی‌رسد. امتحان می‌کنم. عملیات‌هایی که $V(G)$ آن‌ها بیشتر از همه باشد، نامزدهایی برای آزمون‌های جعبه سفید می‌شوند.

وینود: فقط یادت باشد که هیچ تضمینی وجود ندارد. مؤلفه‌ای با $V(G)$ هم ممکن است هنوز مستعد خطا باشد.

شکیرا: حتماً ولی حداقل این به من کمک می‌کند تا تعداد مؤلفه‌هایی را که باید دستخوش آزمون جعبه سفید بشوند، کوچک کنم.

اگر یک بار دیگر به گراف جریان شکل ۲-۱۸ب رجوع کنید، می‌بینید که پیچیدگی سیکلوماتیک را می‌توانید با به‌کارگیری هر یک از الگوریتم‌های بالا محاسبه کنید:

۱. گراف جریان، چهار ناحیه دارد.

۲. $V(G) = 11 - 9 + 2 = 4$ (۱۱ یال و ۹ گره)

۳. $V(G) = 3 + 1 = 4$ (۳ گره گزاره‌ای)

بدین‌ترتیب، پیچیدگی سیکلوماتیک گراف جریان در شکل ۲-۱۸ب، برابر با ۴ است.

نکته‌ی کلیدی

پیچیدگی سیکلوماتیک، مرز بالایی تعداد موارد آزمونی را مشخص می‌کند که لازم هستند تا تضمین شود که هر دستور از برنامه دست کم یک بار اجرا شده است.

```
PROCEDURE average;

  * This procedure computes the average of 100 or fewer
    numbers that lie between bounding values; it also computes the
    sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
(1) i = 1;
(2) total.input = total.valid = 0;
(3) sum = 0;
(4) DO WHILE value[i] <> -999 AND total.input < 100
(5)    increment total.input by 1;
(6)    IF value[i] > = minimum AND value[i] < = maximum
(7)       THEN increment total.valid by 1;
             sum = s sum + value[i]
       ELSE skip
(8)    ENDIF
(9)    increment i by 1;
(10) ENDDO
(11) IF total.valid > 0
(12)    THEN average = sum / total.valid;
(13)    ELSE average = -999;
(14) ENDIF
END average
```

شکل ۴-۱۸ PDL با گره‌های تعیین شده.

مهم‌تر اینکه، مقدار $V(G)$ یک حد فوقانی برای تعداد مسیرهای تشکیل دهنده‌ی مجموعه‌ی پایه ارائه می‌دهد و درنتیجه برای تعداد آزمون‌هایی که باید طراحی و اجرا شوند یک حد فوقانی ارائه می‌کند تا تضمین شود که کلیه دستورات برنامه تحت پوشش قرار گرفتند.

۱۸-۴-۳ به‌دست آوردن موارد آزمون

روش آزمون مسیرهای پایه را می‌توان در یک طراحی رویه‌ای یا کد منبع به‌کار برد. در این بخش، آزمون مسیرهای پایه را به‌صورت چند مرحله ارائه خواهیم داد. از رویه *average* که در PDL شکل ۴-۱۸ تصویر شده است، به‌عنوان مثالی برای نشان دادن هر یک از مراحل این روش استفاده خواهیم کرد. توجه داشته باشید که الگوریتم *average* با وجود سادگی بسیار ، حاوی حلقه‌ها و شرط‌های مرکب است. برای به‌دست آوردن مجموعه‌ی پایه، باید مراحل زیر را اجرا نمود:

۱. **استفاده از طراحی یا کد به‌عنوان یک بستر و رسم گراف جریان مربوط** . گراف جریان با استفاده از نمادهای قواعد ذکر شده در بخش ۱-۴-۱۸ ایجاد می‌شود. با توجه به PDL مربوط به رویه *average* در شکل ۵-۱۸، گراف جریان با شماره‌گذاری آن دسته از دستورهای PDL ایجاد می‌شود که در گره‌های گراف جریان مربوط، تصویر شوند.

> «راکت آریان ۵ هنگام صعود صرفاً به دلیل یک نقص نرم‌افزاری منفجر شد که شامل تبدیل یک عدد ۶۴ بیتی با ممیز شناور به یک عدد صحیح ۱۶ رقمی می‌شد. این راکت و چهار ماهواره آن ۵۰۰ میلیون دلار ارزش داشتند. [با آزمون‌های مسیری که این مسیر تبدیل را تمرین می‌دادند] می‌شد این خطا را یافت ولی به دلایل مرتبط با بودجه، رأی به عدم استفاده از این آزمون‌ها داده شده بود.»
>
> **یک گزارش خبری**

۲. **پیچیدگی سیکلوماتیک گراف جریان حاصل را تعیین کنید.** پیچیدگی سیکلوماتیک، $V(G)$ با اعمال الگوریتم‌های تشریح شده در بخش ۲-۴-۱۸ تعیین می‌شود. باید توجه داشته باشید که $V(G)$ را می‌توان بدون توسعه یک گراف جریان با شمارش کلیه دستورهای شرطی در PDL (برای رویه *average* شرط‌های ترکیبی برابر با ۲ است) و افزودن یک واحد محاسبه کرد. با توجه شکل ۵-۱۸ داریم:

$V(G)$ = 6 ناحیه

$V(G)$ = 17 - 13+2=6 گره ناحیه

$V(G)$ = 5+1=6 گره گزاره‌ای

۳. **تعیین مجموعه‌ی پایه برای مسیرهای مستقل خطی.** مقدار $V(G)$ با استفاده از تعداد مسیرهای مستقل خطی موجود در ساختار کنترلی برنامه مشخص می‌شود. در مورد رویه *average* انتظار داریم شش مسیر مشخص شود:

مسیر ۱: ۱-۲-۱۰-۱۱-۱۳

مسیر ۲: ۱-۲-۱۰-۱۲-۱۳

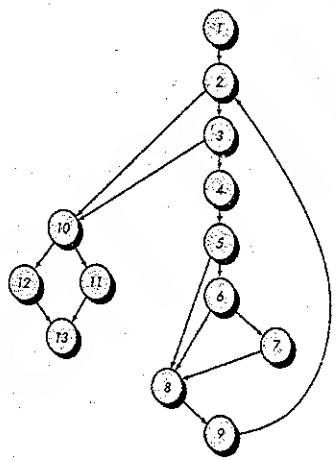
مسیر ۳: ۱-۲-۳-۱۰-۱۱-۱۳

مسیر ۴: ۱-۲-۳-۴-۵-۸-۹-۲-...

مسیر ۵: ۱-۲-۳-۴-۵-۶-۸-۹-۲-...

مسیر ۶: ۱-۲-۳-۴-۵-۶-۷-۸-۹-۲-...

سه‌نقطه‌ای (...) که بعد از مسیرهای ۴، ۵ و ۶ می‌آید، نشان می‌دهد که هر مسیر در باقیمانده ساختار کنترلی قابل قبول است. غالباً خوب است در به‌دست آوردن موارد آزمون، از گره‌های گزاره‌ای (predicate) کمک بگیریم. در این مورد، گره‌های ۱۰، ۶، ۵، ۳ و ۲ گره‌های گزاره‌ای هستند.



شکل ۵-۱۸ گراف جریان برای رویه *average*.

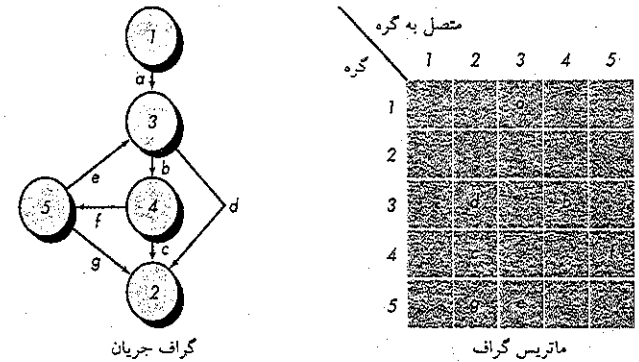
۴. **موارد آزمونی را تهیه کنید که اجرای همه‌ی مسیرها در مجموعه‌ی پایه را الزامی کنند.** داده‌ها را باید طوری انتخاب کرد که به موازات آزموده شدن هر مسیر، شرایط در گره‌های گزاره‌ای به‌طور مناسب تنظیم شود. هر مورد آزمون اجرا می‌شود و با نتایج مورد انتظار مقایسه می‌شود. هنگامی که همه‌ی موارد آزمون کامل شدند، آزمون‌گر می‌تواند مطمئن شود که همه‌ی دستورهای برنامه حداقل یک بار اجرا شده‌اند.

لازم به ذکر است که برخی مسیرهای مستقل (مثل مسیر ۱ در مثال ما) را نمی‌توان به شیوه‌ای مستقل آزمود. یعنی تلفیق داده‌های لازم برای عبور از این مسیر را نمی‌توان در جریان عادی برنامه به‌دست آورد. در چنین مواردی، این مسیرها را به‌عنوان بخشی از یک آزمون مسیر دیگر می‌توان مورد آزمایش قرار داد.

۱۸-۴-۴ ماتریس گراف (Graph Matrix)

روش به‌دست آوردن گراف جریان و حتی تعیین مجموعه‌ای از مسیرهای پایه را می‌توان مکانیزه کرد. توسعه یک ابزار نرم‌افزاری که به آزمون مسیرهای پایه کمک کند و ساختمان داده‌ای موسوم به ماتریس گراف، می‌تواند بسیار مفید واقع شود.

ماتریس گراف یک ماتریس مربعی است که اندازه آن (یعنی تعداد سطرها و ستونهای آن) برابر تعداد گره‌های موجود در گراف جریان است. هر سطر و ستون متناظر با یکی از گره‌ها است و مدخل‌های ماتریس با اتصالات (یعنی یال‌ها) میان گره‌ها متناظرند. یک مثال ساده از گراف جریان و ماتریس گراف متناظر با آن [Bei90] در شکل ۶-۱۸ نشان داده شده است.



شکل ۶-۱۸ ماتریس گراف.

چنان‌که از شکل پیدا است، هر گره از گراف جریان با یک شماره مشخص شده است، حال آنکه هر یال با یک حرف الفبا. مدخل حرفی در ماتریس، برای نشان دادن ارتباط میان دو گره به‌کار رفته است. برای مثال، گره ۳ توسط یال b به گره ۴ متصل شده است.

تا اینجای کار، ماتریس گراف چیزی بیش از یک نمایش جدول‌بندی شده از گراف جریان نیست. ولی با افزودن وزن پیوند به هر یک از مدخل‌های ماتریس، می‌توان آن را به ابزاری پرقدرت برای ارزیابی ساختار کنترلی برنامه در اثنای آزمون تبدیل کرد. وزن پیوند، اطلاعاتی درباره جریان کنترل

ماتریس گراف چیست و چگونه آن را برای استفاده در آزمون بسط دهیم؟

فراهم می‌آورد. وزن پیوند در ساده‌ترین شکل خود، برابر ۱ (وجود ارتباط) یا ۰ (نبود ارتباط) است ولی خواص جالب دیگری را نیز می‌توان به اوزان پیوند نسبت داد:

- احتمال آنکه یک پیوند (یال) اجرا شود؛
- زمان پردازش صرف‌شده برای طی‌کردن یک پیوند؛
- حافظه لازم برای طی‌کردن یک پیوند؛
- منابع لازم برای طی‌کردن یک پیوند.

بیزر [Bei90] الگوریتم‌های قابل‌اجرا روی ماتریس‌های گراف را به‌طور کامل مورد بحث قرار داده است. با استفاده از این تکنیک‌ها، تحلیل لازم برای طراحی موارد آزمون را می‌توان به‌طور جزئی یا کامل خودکار کرد.

۱۸-۵ آزمون ساختار کنترلی (Control Structure Testing)

تکنیک آزمون مسیرهای پایه، که در بخش ۱۸-۴ بحث شد، یکی از چند تکنیک مربوط به آزمون ساختار کنترلی است. گرچه آزمون مسیرهای پایه، ساده و بسیار اثربخش است، به تنهایی کافی نیست. در این بخش، شکل‌های دیگری از آزمون ساختار کنترلی را مورد بحث قرار می‌دهیم. این شکل‌ها، کیفیت آزمون جعبه سفید را بهبود بخشیده پوشش‌دهی آن را وسعت می‌بخشند.

۱۸-۵-۱ آزمون شرط‌ها (Condition Testing)

آزمون شرط‌ها [Tai89] یک روش طراحی موارد آزمون است که شرط‌های منطقی موجود در یک پیمانه برنامه را امتحان می‌کند. هر شرط ساده، یک متغیر بولی یا یک عبارت رابطه‌ای است که ممکن است قبل از آن یک NOT (¬) وجود داشته باشد. عبارت رابطه‌ای به شکل زیر است:

$$E_1 < \text{عمل‌گر رابطه‌ای} > E_2$$

که E_1 و E_2 عبارت‌های محاسباتی و < عمل‌گر رابطه‌ای > یکی از موارد >، ≤، =، ≠، <، یا ≥ است. *شرط* مرکب از دو یا چند شرط ساده، عمل‌گرهای بولی و پرانتز تشکیل می‌شود. فرض می‌کنیم که عمل‌گرهای بولی مجاز در یک شرط مرکب شامل OR (|)، AND (&) و NOT (¬) باشد. شرط بدون عبارت‌های ربطی را عبارت بولی می‌گویند.

اگر شرطی درست نباشد، در آن صورت، حداقل یک مؤلفه از شرط نادرست است. بنابراین، انواع خطاهای موجود در یک شرط شامل خطاهای عملگرهای بولی (عملگرهای بولی نادرست/ جاافتاده/ اضافی)، خطاهای متغیرهای بولی، خطاهای پرانتزهای بولی، خطاهای عملگرهای رابطه‌ای، خطاهای عبارت‌های محاسباتی می‌شوند. در روش آزمون شرط‌ها، آزمایش هر شرط در برنامه، برای حصول اطمینان از نبود خطا در آن است که کانون توجه قرار می‌گیرد.

۱۸-۵-۲ آزمون جریان داده‌ها (Data Flow Testing)

در روش آزمون جریان داده‌ها [Fra93]، مسیرهای آزمون یک برنامه، طبق موقعیت تعاریف و کاربردهای متغیرها در برنامه انتخاب می‌شود. برای نشان دادن روش آزمون جریان داده‌ها، فرض کنید به هر دستور از برنامه یک شماره دستور منحصر بفرد اختصاص داده شود و هیچ تابعی پارامترها یا

«بذل توجه بیشتر به اجرای آزمون‌ها نسبت به طراحی آن‌ها، اشتباهی کلاسیک است.»
برایان ماریک

نکته‌ی کلیدی
خطاها در مجاورت شرایط منطقی نسبت به کانون دستورات پردازشی ترتیبی رایج‌ترند.

«آزمون‌گران خوب، استاد پیداکردن «چیزهای مسخره» و اقدام‌کردن روی آن‌ها هستند.»
برایان ماریک

متغیرهای سراسری خود را اصلاح نمی‌کند. برای دستوری با شماره دستور S داریم:

$$DEF(S) = \{X \mid \text{دستور } S \text{ حاوی تعریف } X \text{ است}\}$$

$$USE(S) = \{X \mid \text{دستور } S \text{ حاوی کاربردی از } X \text{ است}\}$$

اگر دستور S یک دستور if یا حلقه باشد، مجموعه DEF آن خالی است و مجموعه USE آن مبتنی‌بر شرط دستور S است. گفته می‌شود تعریف متغیر X در دستور S در دستور S' زنده است اگر مسیری از دستور S در دستور S' وجود داشته باشد که حاوی هیچ تعریف دیگری از X نباشد.

یک *زنجیره تعریف-کاربرد* (DU) از متغیر X به شکل $[X,S,S']$ است که در آن S و S' شماره دستورها، X در $DEF(S)$ و $USE(S')$ است و تعریف X در دستور S در دستور S' زنده است.

یک راهبرد ساده برای آزمون جریان داده‌ها مستلزم آن‌است‌که هر زنجیره DU حداقل یک بار پوشش داده شود. این راهبرد را راهبرد آزمون DU می‌نامند. ثابت شده است که آزمون DU پوشش‌دهی کلیه شاخه‌های برنامه را تضمین نمی‌کند. ولی، تضمینی نیست که یک شاخه توسط آزمون DU پوشش داده شود، مگر در شرایط نادری مثل ساختارهای if-then-else که در آنها بخش *then* هیچ متغیری را تعریف نمی‌کند و بخش *else* وجود ندارد. در این وضعیت، شاخه *else* از این دستور *if* الزاماً توسط آزمون DU پوشش داده نمی‌شود.

اندرز

واقع‌بینانه نیست که تصور کنیم آزمون جریان داده‌ها انحصاراً هنگام آزمون سیستم‌های بزرگ استفاده می‌شود. ولی می‌توان به شیوه‌ای هدفمند برای نواحی نرم‌افزار که مظنون هستند، از آن استفاده کرد.

۱۸-۵-۳ آزمون حلقه‌ها (Loop Testing)

حلقه‌ها عناصر مهمی در الگوریتم‌های نرم‌افزاری‌اند. با این حال، هنگام اجرای آزمون‌های نرم‌افزاری توجه چندانی به آنها نمی‌شود.

آزمون حلقه‌ها یکی از تکنیک‌های آزمون جعبه سفید است که انحصاراً بر اعتبار ساختمان حلقه تکیه دارد. چهار دسته‌ی متفاوت از حلقه‌ها را می‌توان تعریف کرد [Bei90] : حلقه‌های ساده، حلقه‌های تسلسلی (concatenated)، حلقه‌های تودرتو، و حلقه‌های غیرساخت‌یافته (unstructured) (شکل ۱۸-۷).

حلقه‌های ساده. آزمون‌های زیر را می‌توان در مورد یک حلقه ساده اجرا کرد، که در آن n حداکثر تعداد گذرهای مجاز از میان حلقه است:

۱. عدم اجرای حلقه.

۲. فقط یک بار گذر از حلقه.

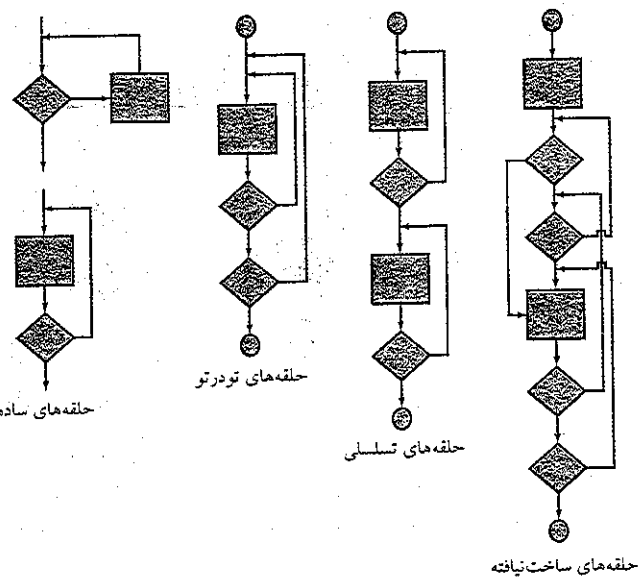
۳. دوبار گذر از حلقه.

۴. m بار گذر از حلقه که $n > m$ است.

۵. $n-1$، n و $n+1$ گذر از حلقه.

حلقه‌های تودرتو. اگر قرار بود روش آزمون مربوط به حلقه‌های ساده را به حلقه‌های تودرتو بسط دهیم، تعداد آزمون‌های ممکن با افزایش سطح تودرتویی، به‌طور هندسی رشد می‌کرد. بیزر [BEI90] روشی پیشنهاد می‌کند که به کاهش‌دادن تعداد آزمون‌ها کمک می‌کند:

۱. از داخلی‌ترین حلقه شروع کنید. همه‌ی حلقه‌های دیگر را در حداقل مقدار قرار دهید.



شکل ۱۸-۷ انواع حلقه‌ها.

۲. آزمون‌های حلقه ساده را برای داخلی‌ترین حلقه اجرا کنید و در عین حال حلقه‌های خارجی دیگر را در حداقل مقدار پارامتر تکرارشان نگه دارید. آزمون‌های دیگری برای مقادیر خارج از محدوده یا مقادیر مستثنی شده اجرا نمایید.

۳. به سمت بیرون حرکت کنید و همین روند را برای حلقه بعدی تکرار کنید، ولی همه‌ی حلقه‌های بیرونی را در حداقل مقدارشان و حلقه‌های داخلی را در مقادیر «معمولی» آنها قرار دهید.

۴. کار را تا آزمون همه‌ی حلقه‌ها ادامه دهید.

اندرز

حلقه‌های غیرساخت‌یافته را نمی‌توانید به‌طور اثربخش آزمایش کنید. آن‌ها را بازآرایی کنید.

حلقه‌های تسلسلی. حلقه‌های تسلسلی را می‌توان با استفاده از روش آزمون برای حلقه‌های ساده آزمود، با این شرط که هر یک از حلقه‌ها مستقل از دیگری باشد. ولی اگر دو حلقه تسلسلی نباشند و شمارنده حلقه ۱ به‌عنوان مقدار اولیه‌ای برای حلقه ۲ استفاده شود، در آن صورت حلقه‌ها مستقل از یکدیگر نیستند. در این حالت، روش به‌کار رفته در حلقه‌های تودرتو را توصیه می‌کنیم.

حلقه‌های غیر ساخت‌یافته. در صورت امکان، این نوع از حلقه‌ها را باید طوری دوباره طراحی کرد که منعکس کننده کاربرد ساختارهای برنامه‌نویسی ساخت‌یافته باشند (فصل ۱۰).

۱۸-۶ آزمون جعبه سیاه (Black Box Testing)

آزمون جعبه سیاه که *آزمون رفتاری* نیز خوانده می‌شود، بر خواسته‌های عملیاتی نرم‌افزار تکیه دارد. یعنی، آزمون جعبه سیاه مهندس نرم‌افزار را قادر می‌سازد تا مجموعه‌هایی از شرط‌های ورودی را به‌دست آورد که همه‌ی خواسته‌های عملیاتی برنامه را به‌طور کامل امتحان کند. آزمون جعبه سیاه

جایگزینی برای تکنیک‌های جعبه سفید به شمار نمی‌رود، بلکه یک روش مکمل است که احتمال پیداکردن دسته دیگری از خطاها را فراهم می‌آورد.

آزمون جعبه سیاه سعی می‌کند خطاهای موجود در این گروه‌ها را بیابد: (۱) عملکرد نادرست یا جاافتاده؛ (۲) خطاهای واسط؛ (۳) خطاهای موجود در ساختمان داده‌ها یا دسترسی به بانک اطلاعاتی خارجی؛ (۴) خطاهای رفتاری یا کارایی و (۵) خطاهای مقداردهی اولیه یا خاتمه برنامه.

برخلاف آزمون جعبه سفید که از اوایل فرایند آزمون اجرا می‌شود، آزمون جعبه سیاه را باید در مراحل آخر آزمون به‌کار برد (فصل ۱۷). چون آزمون جعبه سیاه، ساختار کنترلی را در نظر نمی‌گیرد، توجه به دامنه اطلاعاتی معطوف می‌شود. برای پاسخ دادن به پرسش‌های زیر، آزمون‌هایی طراحی شده است:

- اعتبار عملیاتی چگونه آزموده می‌شود؟
- رفتار و کارآیی سیستم چگونه آزمایش می‌شود؟
- چه دسته‌ای از ورودی‌ها، موارد آزمون خوبی می‌سازند؟
- آیا سیستم نسبت به بعضی از ورودی‌ها حساسیت دارد؟
- مرزهای یک دسته از داده‌ها چگونه معین می‌شود؟
- سیستم چه حجم و میزانی از جریان داده‌ها را می‌تواند تحمل کند؟
- ترکیبات مشخصی از داده‌ها چه تأثیری بر عملکرد سیستم دارند؟

با اجرای تکنیک‌های جعبه سیاه، مجموعه‌ای از موارد آزمون به‌دست می‌آید که ملاک‌های زیر را برآورده می‌کنند [Mye97]: (۱) موارد آزمونی که، تعداد موارد آزمونی را که باید برای دستیابی به آزمونی منطقی طراحی شوند، بیش از یک واحد کاهش می‌دهند، و (۲) موارد آزمونی که درباره وجود یا نبود انواع خطاهای اطلاعاتی به ما می‌دهند، نه خطای مربوط به یک آزمون خاص.

«ارتکاب به خطا، امری انسانی و باقی آن امری الهی است»
رابرت دان

آزمون‌های جعبه سیاه به چه سؤالاتی پاسخ می‌دهند؟

۱-۶-۱۸ روش‌های آزمون مبتنی بر گراف

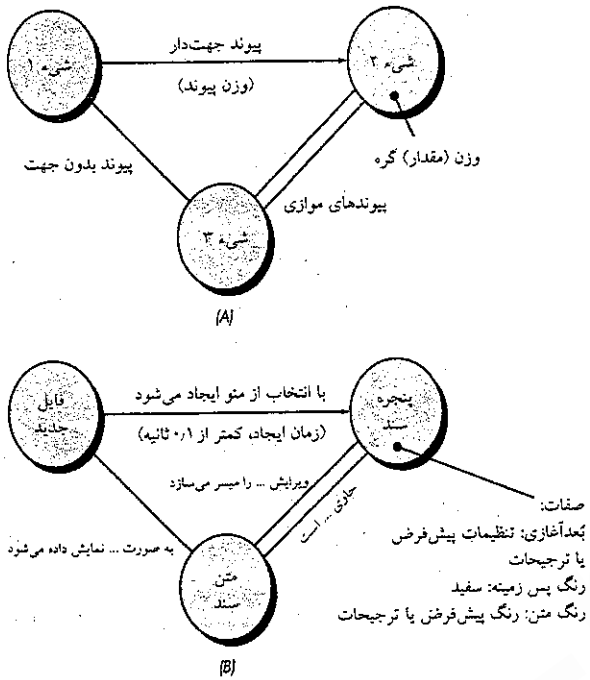
نخستین گام در آزمون جعبه سیاه، شناخت اشیایی[1] است که در نرم‌افزار مدل‌سازی می‌شوند و نیز روابط میان این اشیاست. هنگامی که این منظور برآورده شد، مرحله بعدی تعیین تعدادی آزمون است که ثابت کنند همه‌ی اشیاء، رابطه‌ی موردانتظار را با یکدیگر دارند [Bei95]. به بیان دیگر، آزمون نرم‌افزار با ایجاد گرافی از اشیای مهم و روابط میان آنها و سپس پی‌ریزی تعدادی آزمون‌ها شروع می‌شود که گراف را پوشش می‌دهند، به‌طوری که هر یک از اشیاء و روابط آن با اشیاء دیگری مورد آزمایش قرار می‌گیرد و خطاها کشف می‌شوند.

برای انجام این مراحل، مهندس نرم‌افزار با ایجاد یک گراف شروع می‌کند: مجموعه‌ای از گره‌ها که اشیاء را نشان می‌دهند؛ پیوندها که روابط میان اشیاء را نشان می‌دهند؛ وزن گره که خواص گره را توصیف می‌کنند (مثلاً مقدار داده‌های خاص یا رفتار حالت) و وزن پیوند که خواص پیوند را توصیف می‌کنند.

نکته‌ی کلیدی
گراف، نشان‌گر روابط میان اشیای داده و اشیای برنامه‌ای است و به کمک آن می‌توانید موارد آزمونی به‌دست آورید که خطاهای مرتبط با این روابط را جستجو می‌کنند.

[1] در این حیطه، اصطلاح شیء را باید در وسیع‌ترین شکل ممکن در نظر گرفت به‌گونه‌ای که اشیای داده، مؤلفه‌ها (پیمانه‌های) سنتی و عناصر شیءگرای نرم‌افزارهای کامپیوتری را در بر گیرد.

تصویری از یک گراف در شکل ۸-۱۸الف نشان داده شده است. گره‌ها به صورت دوایر نشان داده شده‌اند که با پیوندهایی به هم متصل شده‌اند و این پیوندها اشکال متفاوتی به خود می‌گیرند. *پیوند جهت‌دار* (که با یک پیکان نشان داده می‌شود) نشان می‌دهد که رابطه تنها در یک جهت برقرار است. پیوند دو جهته که پیوند متقارن نیز خوانده می‌شود، بدان معنا است که رابطه در دو جهت برقرار است. از پیوندهای موازی زمانی استفاده می‌شود که چند رابطه متفاوت بین گره‌های گراف برقرار باشد.



شکل ۸-۱۸ (الف) نمادگذاری گرافی؛ (ب) یک مثال ساده.

به‌عنوان یک مثال ساده، بخشی از گراف مربوط به یک برنامه‌ی واژه‌پرداز را در نظر بگیرید (شکل ۸-۱۸ب) که در آن داریم:

شیء ۱ : انتخاب فایل جدید از منو

شیء ۲ : پنجره سند

شیء ۳ : متن سند

همان‌طور که از شکل پیدا است، با انتخاب گزینه **فایل جدید** یک پنجره سند تولید می‌شود. وزن گره **پنجره سند**، فهرستی از صفات پنجره را فراهم می‌آورد که انتظار می‌رود هنگام تولید پنجره موجود باشند. وزن پیوند نشان می‌دهد که پنجره باید در کمتر از ۱/۰ ثانیه تولید شود. یک پیوند بدون جهت، رابطه‌ای متقارن بین انتخاب منوی **فایل جدید** و **متن سند** برقرار می‌کند و پیوندهای موازی، نشان‌دهنده‌ی روابط میان **پنجره سند** و **متن سند** هستند. در حالت واقعی، گراف تفصیلی‌تری باید برای طراحی مورد آزمون، تهیه شود. سپس مهندس نرم‌افزار موارد آزمون را با طی‌کردن گراف و

پوشش‌دادن کلیه روابط نشان داده شده، به‌دست می‌آورد. این موارد آزمون برای یافتن خطاهای موجود در هر یک از روابط طراحی می‌شوند. بیزر [Bei95] چند روش آزمون رفتاری را توصیف می‌کند که در آنها از گراف‌ها استفاده می‌شود:

مدل‌سازی جریان تراکنش. گره‌ها نشان‌گر مراحلی از یک تراکنش (مثلاً مراحل لازم برای انجام رزرو بلیط هواپیما با استفاده از یک سرویس برخط) هستند و پیوندها ارتباط منطقی میان مراحل را نشان می‌دهند (مثلاً بعد از **flight.information.input** ،*validationAvailabilityProcessing* می‌آید). برای ایجاد این گراف‌ها می‌توان از نمودار جریان داده‌ها (فصل ۱۲) استفاده کرد.

مدل‌سازی حالت متناهی. گره‌ها نشان‌گر حالت‌های مختلفی از نرم‌افزار هستند که توسط کاربر قابل مشاهده‌اند (مثلاً هر یک از «صفحات» به محض گرفتن سفارش تلفنی توسط کارمند، ظاهر می‌شوند) و پیوند گذارهایی را نشان می‌دهند که برای حرکت از حالتی به حالت دیگر رخ می‌دهند (مثلاً **orderInformation** طی *inventoryAvailabilityLook-up* مورد تصدیق قرار می‌گیرد و سپس نوبت به **customerBillingInformationInput** می‌رسد) برای ایجاد این نوع گراف‌ها می‌توان از نمودار گذار حالت (فصل ۷) استفاده کرد.

مدل‌سازی جریان داده‌ها. گره‌ها، اشیای داده هستند و پیوندها، تبدیلاتی هستند که برای تغییر دادن یک شیء به شیء دیگر رخ می‌دهند. برای مثال، گره مالیات بر درآمد FICA (FTW) با استفاده از رابطه FTW = ۰٫۶۲×GW از روی GW (دستمزد پایه) محاسبه می‌شود.

مدل‌سازی زمان‌بندی. گره‌ها اشیای برنامه هستند و پیوندها اتصالات ترتیبی میان آن اشیاء هستند. وزن پیوند برای مشخص‌کردن زمان‌های اجرای برنامه به‌کار می‌روند.

بحث جامع درباره هر کدام از این روش‌های آزمون مبتنی بر گراف، از حوصله این کتاب خارج است. درصورت علاقه بیشتر، برای بحث عمیق‌تر، [Bei95] را ببینید.

۲-۶-۱۸ افراز هم‌ارزی (Equivalence Partitioning)

افراز هم‌ارزی یکی از روش‌های آزمون جعبه سیاه است که دامنه ورودی یک برنامه را به دسته‌هایی از داده‌ها تقسیم می‌کند و موارد آزمون را می‌توان از روی آن به‌دست آورد. یک مورد آزمون ایده‌آل، دسته‌ای از خطاها (مثلاً پردازش نادرست همه‌ی داده‌های کاراکتری) را کشف می‌کند که در غیر این صورت، قبل از مشاهده‌ی یک خطای عمومی، موارد متعددی باید اجرا شوند. افراز هم‌ارزی، مورد آزمونی را تعریف می‌کند که دسته‌هایی از خطاها را کشف می‌کند و در نتیجه، از تعداد کل موارد آزمون می‌کاهد.

مورد آزمون برای افراز هم‌ارزی، مبتنی‌بر تعیین دسته‌های هم‌ارزی برای یک شرط ورودی است. با استفاده از مفاهیمی که در بخش قبل معرفی شدند، اگر بتوان مجموعه‌ای از اشیاء را توسط روابط متقارن، تعدی و انعکاسی به هم پیوند داد، یک دسته‌ی هم‌ارزی وجود دارد [Bei95] . دسته‌ی هم‌ارزی نشان‌گر مجموعه‌ای از حالت‌های معتبر و نامعتبر برای شرایط ورودی است. هر شرط ورودی معمولاً با یک مقدار عددی، بازه‌ای از مقادیر، مجموعه‌ای از مقادیر مرتبط یا یک شرط بولی است. دسته‌های هم‌ارزی را می‌توان طبق دستورالعمل‌های زیر تعریف نمود:

اندرز

کلاس‌های ورودی در فرایند نرم‌افزار نسبتاً زودهنگام شناسایی می‌شوند. به این دلیل، تفکر درباره افراز هم‌ارزی را به موازات ایجاد طراحی شروع کنید.

۱. اگر شرط ورودی، بازه‌ای را مشخص کند، یک دسته‌ی هم‌ارزی معتبر و دو دسته‌ی هم‌ارزی نامعتبر تعریف می‌شوند.
۲. اگر شرط ورودی، نیازمند مقداری مشخص باشد، یک دسته‌ی هم‌ارزی معتبر و دو دسته‌ی هم‌ارزی نامعتبر تعریف می‌شوند.
۳. اگر شرط ورودی نیازمند عضوی از یک مجموعه باشد، یک دسته‌ی هم‌ارزی معتبر و یک دسته‌ی هم‌ارزی نامعتبر تعریف می‌شوند.
۴. اگر شرط ورودی بولی باشد، یک دسته‌ی معتبر و یک دسته‌ی نامعتبر تعریف می‌شود.

با اجرای دستورالعمل‌های مربوط به نحوه به‌دست آوردن دسته‌های هم‌ارز، می‌توان موارد آزمونی برای هر آیتم از داده‌های دامنه ورودی توسعه داد و اجرا نمود. موارد آزمون طوری انتخاب می‌شوند که بیشترین تعداد از صفات یک دسته‌ی هم‌ارزی، یکباره آزمایش شوند.

کلاس‌های هم‌ارزی را برای آزمون‌ها چگونه تعریف کنم؟

۳-۶-۱۸ تحلیل مقادیر مرزی

تعداد خطاهای موجود در مرزهای دامنه ورودی، نسبت به مقادیر مرکزی دامنه بیشتر است. به همین دلیل تکنیکی موسوم به *تحلیل مقادیر مرزی* (BVA) توسعه یافته است. تحلیل مقادیر مرزی، به موارد آزمونی منجر می‌شود که مقادیر مرزی را امتحان می‌کنند.

تحلیل مقادیر مرزی یکی از تکنیک‌های طراحی موارد آزمون است که مکمل افراز هم‌ارزی است. BVA به جای انتخاب هر یک از عناصر دسته‌ی هم‌ارزی، به گزینش موارد آزمونی در «لبه» آن دسته منجر می‌شود. BVA به جای آنکه فقط بر شرطهای ورودی تکیه کند، موارد آزمونی از دامنه خروجی را نیز به‌دست می‌آورد [Mye79] .

یک راه مؤثر برای آزمایش کدها، اجرای آن‌ها در مرزهای طبیعی است.
برایان کرنیگان

دستورالعمل‌های BVA از بسیاری جهات مشابه با دستورالعمل‌های ذکرشده برای افراز هم‌ارزی است:

۱. اگر یک شرط ورودی بازه‌ای محصور به مقادیر a و b را مشخص کند، موارد آزمونی با مقادیر a و b باید طراحی کرد که به ترتیب، درست در پایین و بالای مقدار a و b باشند.
۲. اگر یک شرط ورودی چند مقدار را مشخص می‌کند، باید موارد آزمونی توسعه یابند که اعداد بیشینه و کمینه را امتحان کنند. مقادیری که درست در بالای بیشینه و درست در پایین کمینه قرار دارند نیز باید آزمایش شوند.
۳. دستورالعمل‌های ۱ و ۲ باید برای شرط‌های خروجی نیز اعمال شوند. برای مثال، فرض کنید که یک جدول دما در مقابل فشار، به‌عنوان خروجی یک برنامه تحلیل مهندسی موردنیاز است. موارد آزمون باید طوری طراحی شوند که یک گزارش خروجی ایجاد کنند که حداکثر (و حداقل) تعداد مدخل‌های ممکن برای جدول را مشخص کند.
۴. اگر ساختمان داده‌های داخلی برنامه دارای مرزهای معین باشند (مثلاً آرایه‌ای دارای ۱۰۰ مدخل باشد) حتماً باید یک مورد آزمون برای امتحان‌کردن ساختمان داده طراحی شود.

نکته‌ی کلیدی

BVA افراز هم‌ارزی را با تأکید بر «لبه‌های» یک دسته‌ی هم‌ارزی بسط می‌دهد.

اکثر مهندسان نرم‌افزار، به‌طور ضمنی BVA را تا حدی اجرا می‌کنند. با اعمال دستورالعمل‌های ذکرشده در بالا، آزمون مرزی کامل می‌شود و در نتیجه احتمال پیدا شدن خطاها باز هم فزونی می‌یابد.

۴-۶-۱۸ آزمون آرایه‌های متعامد دامنه

ورودی بسیاری از برنامه‌های کاربردی محدود است. یعنی تعداد پارامترهای ورودی، کوچک و مقادیری که هر یک از پارامترها به خود می‌گیرند دارای مرز مشخصی است. هنگامی که این اعداد

بسیار کوچک باشند (مثلاً ۳ پارامتر ورودی که هر یک ۳ مقدار مجزا می‌گیرند)، می‌توان تمام حالت‌های ورودی را در نظر گرفت و پردازش دامنه ورودی را به‌طور جامع، مورد آزمایش قرار داد. ولی، با رشد تعداد مقادیر ورودی و تعداد مقادیر مجزا جهت هر عنصر داده‌ای، آزمون جامع غیر عملی و امکان‌ناپذیر می‌شود.

آزمون آرایه‌های متعامد را می‌توان در مورد مسائلی به‌کار برد که در آنها دامنه ورودی نسبتاً کوچک است، ولی برای اجرای آزمون جامعیت بیش از حد بزرگ است. روش آرایه متعامد در یافتن خطاهای مرتبط با خطاهای ناحیه‌ای مفید واقع می‌شوند - خطای ناحیه‌ای به گروهی از خطاها اطلاق می‌شود که به منطق نادرست در نقطه‌ای از یک برنامه مربوط می‌شوند.

برای نشان دادن اختلاف میان آزمون آرایه‌های متعامد و روش سنتی «یک عنصر ورودی در هر نوبت»، سیستمی را در نظر بگیرید که دارای سه ورودی X و Y و Z است. هر یک از این عناصر ورودی دارای سه مقدار هستند. پس $۲۷ = ۳^۳$ مورد آزمون متفاوت، امکان‌پذیر است. فادکه [Pha97] یک نمای هندسی از موارد آزمون متفاوت ارائه می‌دهد که در شکل ۹-۱۸ می‌بینید. در هر زمان یکی از ورودی‌ها به ترتیب همراه با هر یک از ورودی‌های دیگر تغییر داده می‌شود. این امر، منجر به پوشش‌دهی نسبی دامنه ورودی می‌شود (که در طرف چپ نشان داده شده است).

هنگامی که آزمون آرایه‌های متعامد رخ می‌دهد یک آرایه متعامد L9 از موارد آزمون تشکیل می‌شود. آرایه متعامد L9 دارای یک «خاصیت متوازن کنندگی» است [Pha97]. یعنی موارد آزمون (که توسط نقاط سیاه در شکل نشان داده شده‌اند) به‌طور یکنواخت در سرتاسر دامنه آزمون پخش شده‌اند (مکعب سمت راست). پوشش‌دهی آزمون در دامنه ورودی کاملتر است.

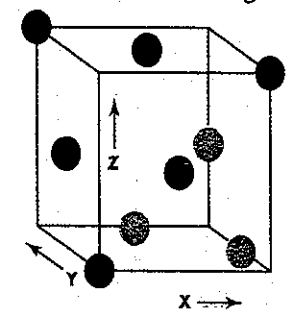
برای نشان دادن کاربرد آرایه متعامد L9 عمل ارسال را برای نمابر در نظر بگیرید. چهار پارامتر P۱، P۲، P۳ و P۴ به «عمل ارسال» تحویل داده می‌شود. هر یک از آنها سه مقدار می‌گیرد. مثلاً P۱ مقادیر زیر را می‌گیرد:

P۱=۱ همین الان ارسال کن

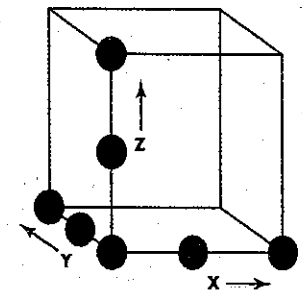
P۱=۲ یک ساعت بعد ارسال کن

P۱=۳ بعد از نیمه‌شب ارسال کن

P۲، P۳ و P۴ نیز مقادیر ۱، ۲ و ۳ را به خود می‌گیرند که هر یک از حالت‌های دیگری ارسال را نشان می‌دهند.



آرایه متعامد L9



هر بار یک ورودی

شکل ۹-۱۸یک نمای هندسی از موارد آزمون [Pha97].

نکته‌ی کلیدی

به کمک آزمون آرایه‌های متعامد می‌توانید موارد آزمونی طراحی کنید که حداکثر پوشش آزمون را با تعدادی منطقی از موارد آزمون فراهم سازند.

اگر از راهبرد آزمون «یک ورودی در هر نوبت» استفاده می‌شد، سری آزمون‌های زیر (P۱,P۲,P۳,P۴) مشخص می‌شد: (۱,۱,۱,۱)، (۲,۱,۱,۱)، (۳,۱,۱,۱)، (۱,۲,۱,۱)، (۱,۳,۱,۱)، (۱,۱,۲,۱)، (۱,۱,۳,۱)، (۱,۱,۱,۲) و (۱,۱,۱,۳). فادکه [Pha97] این موارد آزمون را به شرح زیر ارزیابی می‌کند:

> چنین موارد آزمونی فقط وقتی مفید واقع می‌شوند که شخص یقین دارد که این پارامترهای آزمون با هم تعامل دارند. آنها می‌توانند خطاهای منطقی را در جایی تشخیص دهند که یک مقدار پارامتر منفرد باعث بد کارکردن نرم‌افزار شود. این خطاها را خطاهای حالت یگانه می‌گویند. این روش قادر به یافتن خطاهای منطقی نیست که هنگام گرفتن مقادیر معینی از دو یا چند پارامتر به‌طور همزمان رخ می‌دهند. یعنی قادر به تشخیص تعامل‌ها نیست. از این رو، توانایی آن در یافتن خطاها محدود است.

نظر به تعداد نسبتاً کوچک پارامترهای ورودی و مقادیر مجزا، آزمون جامع امکان‌پذیر است. تعداد آزمون‌های لازم، ۸۱ ($۳^۴$) مورد است که گرچه بزرگ است، ولی عملی است. همه‌ی خطاهای مرتبط با حالت‌های ترکیبی متفاوت داده‌ها پیدا خواهند شد، ولی کار لازم برای رسیدن به این منظور نسبتاً زیاد است.

روش آزمون آرایه‌های متعامد، در مقایسه با آزمون جامع، با موارد آزمون کمتر، پوشش خوبی را ارائه می‌کند. آرایه متعامد L9 برای عمل ارسال نمابر در شکل ۱۰-۱۸ نشان داده شده است.

مورد آزمون	پارامترهای آزمون			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

شکل ۱۰-۱۸ آرایه متعامد L9.

فادکه [Pha97]، نتایج آزمون با استفاده از آرایه متعامد L9 را چنین ارزیابی می‌کند:

شناسایی و جداسازی کلیه خطاهای حالت یگانه. خطای حالت یگانه یک مشکل سازگار با هر یک از پارامترهای منفرد است. برای مثال، اگر همه‌ی موارد آزمون با فاکتور P۱=۱ باعث ایجاد خطا شوند، حالت یگانه شکست می‌خورد. در این مثال، آزمون‌های ۱، ۲ و ۳ (شکل ۱۰-۱۸) خطا را نشان می‌دهند. با تحلیل اطلاعات مربوط به آزمون‌هایی که از خود خطا نشان می‌دهند، می‌توان

ابزارهای نرم‌افزاری

طراحی موارد آزمون

هدف: کمک به تیم نرم‌افزاری در توسعه‌ی مجموعه کاملی از موارد آزمون برای هر دو نوع آزمون جعبه سیاه و جعبه سفید.

مکانیک: این ابزارها به دو گروه عمده قابل تقسیم هستند: **ابزارهای آزمون ایستا و ابزارهای آزمون پویا.** سه نوع ابزار آزمون ایستا در صنعت نرم‌افزار به‌کار می‌رود: ابزارهای آزمون مبتنی بر کد، زبان‌های آزمون تخصص یافته و ابزارهای مبتنی بر خواسته‌ها. ابزارهای آزمون مبتنی بر کد، کد منبع را به‌عنوان ورودی می‌پذیرد و چند تحلیل روی آن انجام می‌دهد که به تولید موارد آزمون منجر می‌گردد. زبان‌های آزمون تخصص یافته (نظیر ATLAS) مهندس نرم‌افزار را قادر می‌سازد تا مشخصات آزمون مفصلی بنویسد که هر مورد آزمون و منطق مربوط به اجرای آن را اجرا می‌کند. ابزارهای آزمون مبتنی بر خواسته‌ها، خواسته‌های مشخص کاربر را جدا می‌کنند و موارد آزمونی (یا دسته‌هایی از آزمون‌ها) را پیشنهاد می‌کنند که خواسته‌ها را بررسی می‌کنند. ابزارهای آزمون پویا با یک برنامه‌ی در حال اجرا تعامل می‌کنند، پوشش‌دهی مسیرها را چک می‌کنند، مقدار متغیرهای خاصی را تایید می‌کنند و در غیر این صورت، به جریان اجرای برنامه کمک می کند.

ابزارهای نمونه

McCabe Test، که توسط McCabe&Associations (**www.mccabe.com**) توسعه یافته است، انواع تکنیک‌های آزمون مسیر به‌دست آمده از ارزیابی پیچیدگی سیکلوماتیک و سایر معیارهای نرم‌افزار را پیاده‌سازی می‌کند.

Test Works، که توسط Software Research Inc. (**www.soft.com/Products**) مجموعه‌ای از ابزارهای آزمون خودکار است که به زبان‌های C/C++ و جاوا توسعه یافته‌اند و آزمون رگرسیون را پشتیبانی می‌کنند.

- *T-Vec Test Generation System* که توسط T-Vec Technologies (**www.t-vec.com**) توسعه یافته است، مجموعه ابزارهایی است که آزمون واحدها، انسجام و اعتبارسنجی را با کمک‌کردن به طراحی موارد آزمون و با به‌کارگیری اطلاعات موجود در مشخصه‌ی خواسته‌های شیء‌گرا پشتیبانی می‌کند.

e-Test Suite که توسط Empirix,Inc (**www.empirix.com**) توسعه یافته است، شامل مجموعه کاملی از ابزارها برای آزمایش برنامه‌های تحت وب می‌شود از جمله ابزارهایی که به طراحی موارد آزمون و برنامه‌ریزی برای آزمون‌ها کمک می‌کنند.

تشخیص داد کدام مقادیر پارامتر باعث ایجاد خطا می‌شوند. در این مثال، با توجه به اینکه آزمون‌های ۱، ۲ و ۳ باعث خطا می‌شوند، می‌توان [پردازش منطقی مرتبط با «هم‌اکنون ارسال کن» (P۱ = ۱)] را به‌عنوان منبع خطا جدا کرد. یک چنین جداسازی خطایی، برای رفع آن اهمیت دارد.

تشخیص کلیه خطاهای حالت دوگانه. اگر هنگام مقداردهی همزمان به دو یا چند پارامتر اشکالی ایجاد شود، خطای حالت دوگانه وجود دارد. درواقع، خطای حالت دوگانه نشانی از ناسازگاری جفت‌شدگی (pairwise) یا تعامل زبان‌بار میان دو پارامتر آزمون است.

خطاهای چندحالتی. آرایه‌های متعامد (از نوع نشان داده شده) می‌توانند فقط یافتن خطاهای حالت یگانه و دوگانه را تضمین کنند. ولی بسیاری از خطاهای چند حالتی را نیز توسط این آزمون‌ها می‌توان یافت. بحث جامعی درباره آزمون آرایه‌های متعامد را در [Pha97] می‌توانید بیابید.

۱۸-۷ آزمون مبتنی بر مدل (Model Based Testing)

آزمون مبتنی بر مدل (MBT) یک تکنیک آزمون جعبه سیاه است که از اطلاعات موجود در مدل خواسته‌ها به‌عنوان مبنایی برای تولید موارد آزمون بهره می‌برد. در بسیاری موارد، در تکنیک مبتنی بر مدل از نمودارهای حالت UML، عنصری از مدل رفتاری (فصل ۷) به‌عنوان مبنای طراحی موارد آزمون استفاده می‌شود[1]. تکنیک MBT به پنج مرحله نیاز دارد:

۱. **تحلیل یک مدل رفتاری موجود برای نرم‌افزار یا ایجاد آن.** به‌خاطر دارید که مدل رفتاری چگونگی پاسخ‌دادن نرم‌افزار به رویدادها یا محرک‌های بیرونی را نشان می‌دهد. برای ایجاد مدل، باید مراحل بحث شده در فصل ۷ را اجرا کنید: (۱) ارزیابی همه‌ی use case ها برای درک کامل دنباله‌ای از تعامل‌های درون سیستم، (۲) شناسایی رویدادهایی که این دنباله از تعامل‌ها را به پیش می‌رانند و درک چگونگی ارتباط این رویدادها با اشیای خاص، (۳) ایجاد دنباله‌ای برای هر use case، (۴) ساخت یک نمودار حالت UML برای سیستم (مثلاً شکل ۶-۷) و (۵) بازبینی مدل رفتاری برای وارسی سازگاری و درستی آن.

۲. **مرور مدل رفتاری و مشخص‌کردن ورودی‌هایی که نرم‌افزار را وادار می‌سازند تا از حالتی به حالت دیگر گذار کند.** ورودی‌ها باعث شروع رویدادهایی می‌شوند که خود موجب رخ‌دادن گذار خواهند شد.

۳. **مرور بر مدل رفتاری و توجه به خروجی قابل انتظار از نرم‌افزار، هنگامی که از حالتی به حالت دیگر گذار می‌کند.** به‌خاطر دارید که هر گذار توسط رویدادی آغاز می‌شود و در نتیجه‌ی این گذار، تابعی فراخوانده می‌شود و خروجی‌هایی ایجاد می‌شود. برای هر مجموعه از ورودی‌ها (موارد آزمون) که در مرحله‌ی ۲ مشخص کردید، خروجی‌های مورد انتظار را به موازات مشخص شدن آن‌ها در مدل رفتاری تعیین کنید. «یک فرض بنیادی در این آزمون، آن است که سازوکاری وجود دارد، یک حکیم *فرزانه‌ی آزمون* (test oracle)، که تعیین خواهد کرد آیا نتایج اجرای آزمون درست هست یا خیر» [DAC03]. در اصل، این حکیم فرزانه، مبنایی برای هرگونه تعیین صحت خروجی فراهم می‌آورد. در اکثر موارد، حکیم فرزانه، همان مدل خواسته هاست ولی می‌تواند هر مستند یا برنامه کاربردی دیگر، داده‌های ثبت شده در جای دیگر یا حتی یک کارشناس انسانی باشد.

۴. **اجرای موارد آزمون.** آزمون‌ها را می‌توان به‌صورت دستی اجرا کرد یا یک اسکریپت آزمون تهیه کرد و آن را با استفاده از ابزارهای آزمون، اجرا کرد.

۵. **نتایج واقعی و مورد انتظار را مقایسه کنید و درصورت نیاز، اقدامات تصحیحی را به‌عمل آورید.**

MBT به کشف خطاهای موجود در رفتار نرم‌افزار کمک می‌کند و در نتیجه، هنگام آزمایش برنامه‌های کاربردی که رویداد گرا هستند، بی‌نهایت مفید واقع می‌شوند.

> «یافتن خطا در کدها هنگامی که به دنبال آن هستید، به‌قدر کافی دشوار هست؛ وقتی که فکر کنید کدهای شما عاری از خطا هستند، از این هم سخت‌تر می‌شود.»
>
> استیومک کابل

[1] آزمون مبتنی بر مدل را می‌توان به هنگام نمایش خواسته‌های نرم‌افزار با جداول تصمیم‌گیری گرامرها یا زنجیره‌های مارکوف [Dac03] نیز به‌کار برد.

۱۸-۸ آزمون محیط‌ها، معماری‌ها و برنامه‌های کاربردی تخصص یافته

هنگامی که محیط‌ها، معماری‌ها و برنامه‌های کاربردی تخصص یافته مورد نظر قرار می‌گیرند، گاهی به رویکردها و دستورالعمل‌های منحصر به فرد نیاز است. گرچه تکنیک‌های آزمون بحث شده در این فصل و فصل‌های ۱۹ و ۲۰ را غالباً می‌توان بر شرایط خاص تطبیق داد، در نظر گرفتن این نیازهای منحصر به فرد نیز می‌تواند ارزش‌مند باشد.

۱-۸-۱۸ آزمون واسط گرافیکی کاربر

واسط گرافیکی کاربر (GUI) مشکلات جالبی سر راه مهندس نرم‌افزار قرار می‌دهد. به دلیل وجود مؤلفه‌های قابل استفاده مجدد در محیط‌های توسعه‌ی GUI، ایجاد واسط کاربر، با زمانی کمتر و با دقت بالاتر امکان‌پذیر شده است. ولی در عین حال، پیچیدگی GUI نیز فزونی یافته است و درنتیجه، طراحی و اجرای موارد آزمون دشوارتر شده است.

از آنجا که بسیاری از GUIهای جدید دارای شکل و شمایل خاصی هستند، می‌توان به آزمون‌های استاندارد دست یافت. گراف‌های مدل‌سازی حالت‌های متناهی را می‌توان برای به‌دست آوردن آزمون‌هایی به‌کار برد که با داده‌های مشخص و اشیای برنامه‌ای مرتبط با GUI سروکار دارند. این تکنیک آزمون مبتنی بر مدل در بخش ۷-۱۸ بحث شد.

به دلیل تعداد زیاد حالت‌های ممکن در عملیات GUI، آزمون باید با استفاده از ابزارهای خودکار انجام شود. طی چند سال اخیر، انواع ابزارهای آزمون GUI به بازار ارائه شدند.[1]

۲-۸-۱۸ آزمون معماری‌های کلاینت/سرور

معماری‌های کلاینت/سرور مشکل چشمگیری برای آزمون‌گر نرم‌افزار پیش می‌آورند. ماهیت توزیع‌شده محیط‌های کلاینت/سرور، مشکلات کارایی مرتبط با پردازش تراکنش‌ها، حضور بالقوه‌ی چند سایت سخت‌افزاری متفاوت، پیچیدگی‌های ارتباط شبکه‌ای، نیاز به کلاینت‌های چندگانه از یک بانک اطلاعاتی متمرکز (و در برخی موارد، توزیع شده)، و خواسته‌های هماهنگ‌سازی تحمیل شده به سرور، همگی دست به‌دست هم داده آزمون معماری‌های کلاینت/سرور و نرم‌افزارهای مربوط به آنها را دشوارتر از نرم‌افزارهای مستقل ساخته است. درحقیقت، مطالعات صنعتی جدید، نشان‌دهنده‌ی افزایش چشمگیر زمان آزمون و هزینه در هنگام توسعه محیط‌های کلاینت/سرور هستند.

به‌طور کلی، آزمون نرم‌افزارهای کلاینت/سرور در سه سطح متفاوت رخ می‌دهد:

(۱) برنامه‌های کاربردی کلاینت، یک به یک در حالتی «نامتصل» آزمایش می‌شوند؛ عملکرد سرور و شبکه زیرساختی در نظر گرفته نمی‌شود. (۲) نرم‌افزار کلاینت و برنامه‌های مرتبط با آن در سرور هماهنگ با هم آزمایش می‌شوند ولی عملکرد شبکه به صراحت تمرین داده نمی‌شود. (۳) معماری کامل کلاینت/سرور، شامل عملکرد و کارایی شبکه آزمایش می‌شود.

گرچه انواع بسیار متفاوتی از آزمون‌ها در هر کدام از این سطوح جزئیات اجرا می‌شود، رویکردهای زیر در خصوص آزمون برنامه‌های کاربردی بیشتر به چشم می‌خورند:

[1] صدها یا شاید هزاران منبع برای ابزارهای آزمون GUI در وب موجود است. یک نقطه شروع خوب برای ابزارهایی با منبع باز، www.opensourcetesting.org/functional.php است.

«مبحث آزمون، مبحثی است که در آن وجوه اشتراک زیادی میان سیستم سنتی و سیستم سرور/کلاینت وجود دارد.»

کلی بورن

مرجع وب

«اطلاعات مفیدی درباره‌ی آزمون سیستم‌های کلاینت/سرور و منابع مربوط به آن‌ها را می‌توانید در آدرس زیر بیابید:

www.csst-technologies.com

- **آزمون‌های عملکردهای برنامه کاربردی.** عملکردهای برنامه‌های کاربردی با به‌کارگیری روش‌های بحث شده در این فصل و فصل‌های ۱۹ و ۲۰ آزمایش می‌شود. در اصل، برنامه کاربردی به شیوه‌ای مستقل آزمایش شود تا خطاهای موجود در عملکرد آن کشف گردد.
- **آزمون سرور.** عملکردهای هماهنگ‌سازی و مدیریت داده‌های سرور آزمایش می‌شوند. کارایی سرور (زمان پاسخ کلی و توان عملیاتی داده ای) نیز در نظر گرفته می‌شود.
- **آزمون‌های بانک اطلاعاتی.** صحت و انسجام داده‌های نگهداری شده توسط سرور آزمایش می‌شود. تراکنش‌های اعلان شده توسط برنامه‌های کاربردی، بررسی می‌شوند تا اطمینان حاصل شود که داده‌ها به‌طور مناسب نگهداری، بهنگام و بازیابی می‌شوند. آرشیو نیز آزمایش می‌شود.
- **آزمون‌های تراکنش‌ها.** یک سری آزمون ایجاد می‌شود تا اطمینان حاصل شود که هر دسته‌ای از تراکنش‌ها مطابق با خواسته‌ها پردازش شود. در این آزمون‌ها، درستی پردازش و نیز مسائل کارایی (مانند زمان پردازش تراکنش و حجم تراکنش) کانون توجه قرار می‌گیرند.
- **آزمون‌های ارتباطات شبکه.** این آزمون‌ها وارسی می‌کنند که ارتباطات میان گره‌های شبکه به‌طور صحیح رخ دهند و تبادل پیام‌ها، تراکنش‌ها و ترافیک شبکه بدون خطا رخ می‌دهد. آزمون‌های امنیتی نیز ممکن است به‌عنوان بخشی از این آزمون‌ها اجرا شوند.

«برای سیستم‌های کلاینت/سرور، کدام انواع آزمون‌ها به کار می‌روند؟»

موزا [Mus93] برای اجرای این رویکردها، توسعه‌ی *پروفایل‌های عملیاتی* به‌دست آمده از سناریوهای کاربرد کلاینت-سرور را توصیه می‌کند.[1] پروفایل عملیاتی نشان می‌دهد که انواع متفاوت کاربران چگونه با سیستم کلاینت-سرور همکاری دارند. یعنی، این پروفایل‌ها «الگوی کاربردی» فراهم می‌آورند که می‌توان آن‌ها را در طراحی و اجرای آزمون‌ها به‌کاربرد. برای مثال، برای نوع خاصی از کاربر، چه درصدی از تراکنش‌ها، درخواستی هستند، چه درصدی بهنگام‌سازی و چه درصدی، سفارشی؟

برای توسعه بخشیدن به پروفایل عملیاتی، لازم است مجموعه‌ای از سناریوها به‌دست آید که مشابه با use case هستند (فصل‌های ۵ و ۶). در هر سناریو، چه کسی، کجا، چه، و چرا مورد توجه قرار می‌گیرد. یعنی، کاربر چه کسی است، تعامل سیستم در کجای معماری فیزیکی کلاینت-سرور رخ می‌دهد، تراکنش چیست و چرا رخ داده است. سناریوها را می‌توان با به‌کارگیری تکنیک‌های استخراج خواسته‌ها (فصل ۵) یا از طریق بحث‌های نه چندان رسمی با کاربران نهایی به‌دست آورد. به هر حال، نتیجه باید یکسان باشد. هر سناریو شاخصی می‌دهد از **عملکردهای** سیستم ارائه می‌دهد که باید به‌کاربری خاص سرویس دهند، ترتیب ضرورت پیداکردن این **عملکردها**، زمان‌بندی و پاسخ مورد انتظار و فراوانی به‌کارگیری هر کدام از این **عملکردها**. این داده‌ها سپس ترکیب می‌شوند (برای همه‌ی کاربران) تا پروفایل عملیاتی ایجاد گردد. به‌طور کلی، تلاش‌های به عمل آمده برای انجام آزمون و تعداد موارد آزمونی که باید اجرا شوند، بر اساس فراوانی کاربرد و اهمیت بحرانی وظایف اجرا شده به هر سناریوی کاربرد تخصیص داده می‌شوند.

[1] شایان ذکر است که نیمرخ‌های عملیاتی را می‌توان در آزمایش برای انواع معماری‌های سیستم و نه فقط معماری کلاینت/سرور به‌کار برد.

۳-۸-۱۸ آزمون مستندات و تسهیلات راهنما

اصطلاح «*آزمون نرم‌افزار*» تعداد زیادی موارد آزمون را در ذهن متبادر می‌کند که برای امتحان‌کردن برنامه کامپیوتری و داده‌هایی که این برنامه دستکاری می‌کند، تهیه شده‌اند. با توجه به تعریف نرم‌افزار در فصل اول، لازم به ذکر است که آزمون باید به سومین عنصر از پیکربندی نرم‌افزار (یعنی مستندات) نیز توسعه یابد.

خطاهای موجود در مستندات برنامه می‌تواند به اندازه خطاهای موجود در داده‌ها یا کد منبع، مخرب باشند. هیچ چیز ناراحت‌کننده‌تر از این نیست که دستورالعملهای موجود در جزوه راهنما یا راهنمای آنلاین یک برنامه را دنبال کنید و نتایج پیش‌بینی شده را مشاهده نکنید. لذا آزمون مستندات باید در طراحی آزمون به حساب آورده شود.

آزمون مستندات را در دو فاز می‌توان انجام داد. در فاز نخست، یعنی مرور و وارسی (فصل ۸) مستندات از لحاظ وضوح ویرایش، مورد بررسی قرار می‌گیرند. فاز دوم، یعنی آزمون زنده، از مستندات همراه با برنامه استفاده می‌شود.

آزمون زنده برای مستندسازی را می‌توان با استفاده از تکنیک‌هایی مشابه با روش‌های آزمون جعبه سیاه (بخش ۶-۱۷) انجام داد. از آزمون مبتنی بر گراف می‌توان برای توصیف کاربرد برنامه استفاده کرد؛ افراز هم‌ارزی و تحلیل مقدار مرزی را می‌توان برای تعیین دسته‌های گوناگونی از تعامل‌های ورودی و مرتبط به‌کار برد. از MBT می‌توان برای حصول اطمینان از همخوانی رفتار مستندسازی شده و رفتار واقعی استفاده نمود. در آن صورت، استفاده از برنامه از طریق مستندات تهیه‌شده قابل ردگیری خواهد بود.

اطلاعات

آزمایش مستندات

طی آزمون مستندات و/یا تسهیلات راهنما باید به پرسش‌های زیر پاسخ گفت:

- آیا مستندات، چگونگی دستیابی به هر حالت استفاده را به‌درستی توصیف می‌کنند؟
- آیا توصیف هر سری از تعامل‌ها صحیح است؟
- آیا مثال‌ها درست هستند؟
- آیا اصطلاح شناسی، توصیف منوها و پاسخ‌های سیستم با برنامه واقعی سازگاری دارند؟
- آیا یافتن راهنمایی در داخل مستندات، نسبتاً آسان هست؟
- آیا اشکال‌زدایی به کمک مستندات به آسانی قابل انجام است؟
- آیا جدول محتویات و نمایه (index)، کامل و صحیح است؟
- آیا طراحی مستندات (چیدمان، نوع فونت‌ها، تو رفتگی‌ها، گرافیک‌ها) به درک مطلب و پذیرش سریع اطلاعات کمک می‌کند؟
- آیا همه‌ی پیام‌های خطای نرم‌افزار ارائه‌شده به کاربر، با توضیحات بیشتر در مستندات شرح داده شده اند؟ آیا اقدامات که باید با دیده شدن یک پیام خطا به عمل آید، به وضوح مشخص شده است؟
- اگر از پیوندهای فوق متنی استفاده می‌شود، آیا طراحی گشت‌وگذار برای اطلاعات لازم، مناسب هست؟
- اگر از پیوندهای فوق متنی استفاده می‌شود، آیا این پیوندها صحیح و کامل هستند؟

تنها راه برای پاسخ‌دادن به این پرسش‌ها این است که از یک طرف سومی درخواست شود (مثلاً منتخبی از کاربران) تا مستندات را در حیطه‌ی کاربرد برنامه آزمایش کنند. همه‌ی مغایرت‌ها ذکر شود و نواحی ابهام یا نقاط ضعف برای بازنویسی‌های احتمالی تعیین شوند.

۴-۸-۱۸ آزمون‌های مربوط به سیستم‌های بی‌درنگ (Real-Time)

ماهیت وابسته به زمان در بسیاری از کاربردهای بی‌درنگ، یک عنصر دشوار «زمان» را به آزمون می‌افزاید. طراح موارد آزمون، نه تنها باید موارد آزمون جعبه سیاه و جعبه سفید را در نظر بگیرد، بلکه باید کنترل رویدادها (یعنی پردازش وقفه‌ها)، زمان‌بندی داده‌ها و موازی بودن وظایفی (فرایندهایی) را که با داده‌ها کار می‌کنند، نیز در نظر بگیرند. در بسیاری از شرایط، داده‌های آزمونی که در یکی از حالت‌های سیستم بی‌درنگ به‌دست آمده‌اند، به پردازش مناسب منجر می‌شود، در حالی که همان داده‌ها که در حالت دیگری از سیستم بی‌درنگ به‌دست آمده‌اند، ممکن است منجر به خطا شوند.

برای مثال، نرم‌افزارهای بی‌درنگ که یک دستگاه فتوکپی جدید را کنترل می‌کند، وقفه‌های اپراتور (یعنی وقتی اپراتور کلیدهای کنترلی مثل RESET یا DARKEN را می‌زند) را در حین گرفتن کپی (حالت «کپی‌گرفتن») می‌پذیرد. اگر دستگاه کپی در حالت گیرکردن کاغذ باشد و اپراتور کلیدهای کنترلی را فشار دهد، ماشین دچار وقفه می‌شود و درنتیجه پیامی صادر می‌شود که مشخص می‌کند مکان گیرکردن کاغذ از دست رفته است (خطا).

به‌علاوه، رابطه نزدیکی که بین نرم‌افزارهای بی‌درنگ و محیط سخت‌افزاری آن وجود دارد نیز می‌تواند باعث ایجاد مشکلات در آزمون شود. در آزمون‌های نرم‌افزاری باید تأثیر اشکالات سخت‌افزاری بر پردازش نرم‌افزار نیز در نظر گرفته شود. شبیه‌سازی واقعی چنین خطاهایی می‌تواند بی‌اندازه دشوار باشد.

روش‌های مفهومی طراحی موارد آزمون برای سیستم‌های بی‌درنگ، هنوز باید تکامل پیدا کنند. ولی یک راهبرد چهار مرحله‌ای کلی می‌توان پیشنهاد کرد:

راهبرد اثربخش برای آزمایش یک سیستم بی‌درنگ چیست؟

- **آزمون وظایف.** نخستین گام در آزمودن نرم‌افزار بی‌درنگ این است که هر یک از وظایف به‌طور مستقل آزمایش شوند. یعنی برای هر وظیفه، آزمون‌های جعبه سیاه و جعبه سفیدی طراحی و اجرا شوند. طی این آزمون‌ها هر وظیفه به‌طور مستقل اجرا می‌شود. آزمون وظایف باعث کشف خطاهای موجود در منطق و عملکرد می‌شوند، ولی خطاهای رفتاری و زمان‌بندی را برملا نمی‌کنند.
- **آزمون رفتاری.** با استفاده از مدل‌های سیستمی که توسط ابزارهای کیس ایجاد شدند می‌توان رفتار سیستم بی‌درنگ را شبیه‌سازی کرد و رفتار آن را به‌عنوان پیامدی از رویدادهای خارجی بررسی کرد. این فعالیت‌های تحلیلی می‌توانند به‌عنوان مبنایی برای طراحی موارد آزمونی عمل کنند که هنگام ساخته شدن نرم‌افزارهای بی‌درنگ اجرا می‌شوند. با استفاده از تکنیکی مشابه با افزار هم‌ارزی (بخش ۱۲-۶-۱۸)، رویدادها (مثلاً وقفه‌ها و سیگنال‌های کنترلی)، برای آزمون گروه‌بندی شده‌اند. برای مثال، رویدادهای مربوط به‌دستگاه فتوکپی ممکن است عبارت باشند از: وقفه‌های کاربر (مثل صفرکردن شمارنده)، وقفه‌های مکانیکی (مثل گیرکردن کاغذ)، وقفه‌های سیستمی (مثل کم شدن قدرت تونر) و حالت‌های خطا (داغ شدن غلطک). هر یک از این رویدادها به‌طور انفرادی مورد آزمون قرار می‌گیرد و رفتار سیستم اجرایی مورد بررسی قرار می‌گیرد تا خطاهایی که در نتیجه پردازش این رویدادها رخ می‌دهند، برملا شوند. رفتار مدل سیستمی (که طی فعالیت تحلیل توسعه می‌یابد) و نرم‌افزار قابل اجرا را می‌توان برای همخوانی مورد مقایسه قرار داد. هنگامی که انواع رویدادها مورد آزمون قرار گرفتند، رویدادها

به ترتیب تصادفی و با فراوانی تصادفی به سیستم ارائه می‌شوند. رفتار سیستم برای یافتن خطاهای رفتاری بررسی می‌شود.

- **آزمون بین وظایف.** هنگامی که خطاهای موجود در وظایف انفرادی و رفتار سیستم جداسازی شدند، آزمون به خطاهای مرتبط با زمان متمایل می‌شود. وظایف ناهمگامی که با هم ارتباط دارند، با سرعت‌های متفاوتی از داده‌ها مورد آزمون قرار می‌گیرند، تا معین شود آیا خطاهای همگام‌سازی بین وظایف رخ می‌دهد یا خیر. به‌علاوه، وظایفی که از طریق صفی از پیام‌ها یا انبار داده‌ها با هم ارتباط برقرار می‌کنند، آزمایش می‌شوند تا خطاهای موجود در تعیین اندازه نواحی نگهداری داده‌ها برملا شوند.
- **آزمون سیستم.** نرم‌افزار و سخت‌افزار به هم گره خورده‌اند و گستره‌ی کاملی از آزمون‌های سیستم اجرا می‌شوند تا خطاهای موجود در واسط میان سخت‌افزار و نرم‌افزار پیدا شوند. اکثر سیستم‌های بی‌درنگ وقفه‌ها را پردازش می‌کنند. بنابراین، آزمون مربوط به کنترل این رویدادهای بولی ضروری است. آزمون‌گر با استفاده از نمودار گذار حالت و مشخصه کنترلی (فصل ۷) فهرستی از همه‌ی وقفه‌های ممکن و پردازشی تهیه می‌کند که به‌عنوان پیامدی از وقفه رخ می‌دهد. سپس آزمون‌هایی طراحی می‌شوند که ویژگی سیستمی زیر را مورد ارزیابی قرار دهند.
  - آیا اولویت‌های وقفه به‌طور مناسب تخصیص می‌یابند و کنترل می‌شوند؟
  - آیا پردازش مربوط به هر وقفه به درستی کنترل شده‌اند؟
  - آیا کارایی (مثلاً زمان پردازش) برای هر یک از رویه‌های کنترل وقفه از خواسته‌ها تبعیت می‌کنند؟
  - آیا حجم بالایی از وقفه‌ها که در زمان‌های بحرانی رخ می‌دهند، مشکلاتی را در عملکرد یا کارآیی به‌وجود می‌آورند؟

علاوه بر این، نواحی داده‌های سرتاسری که برای انتقال دادن داده‌ها به‌عنوان بخشی از پردازش وقفه به‌کار می‌روند، باید مورد آزمون قرار گیرند تا توان بالقوه برای ایجاد اثرات جانبی سنجیده شود.

۱۸-۹ الگوهای مربوط به آزمون نرم‌افزار

استفاده از الگوها به‌عنوان سازوکاری برای توصیف راهکارهای مسائل طراحی در فصل ۱۲ بحث شد. ولی الگوها را می‌توان در پیشنهاد راهکارهایی برای سایر وضعیت‌ها در مهندسی نرم‌افزار نیز به‌کار برد- که در این جا، منظور آزمون نرم‌افزار است. الگوهای آزمون، مسائل رایج در آزمون و راهکارهایی را توصیف می‌کنند که ممکن است شما را در کار با آن‌ها یاری دهند.

الگوهای آزمون نه تنها راهنمایی مفیدی در شروع فعالیت‌های آزمون در اختیار شما قرار می‌دهند، بلکه سه مزیت اضافی فراهم می‌آورند که توسط ماریک شرح داده شده است [Mar02]:

۱. آن‌ها [الگوها] برای کسانی که مسأله حل می‌کنند، یک دایره‌ی لغات فراهم می‌آورند. «هی، می‌دانی، باید از یک شیء نول استفاده کنیم.»

۲. آن‌ها توجه را به نیروهای پشت مسأله معطوف می‌سازند و این امر به طراحان [موارد آزمون] امکان می‌دهد تا بهتر درک کنند که چه هنگام و چرا راهکاری نتیجه بخش خواهد بود.

مرجع وب

کاتالوگی از الگوهای آزمون را در آدرس زیر می‌توان یافت:
www.rbsc.com/pages/TestPatternList.htm

نکته‌ی کلیدی

الگوهای آزمون می‌توانند تیم نرم‌افزار را در برقراری ارتباط اثربخش‌تر درباره‌ی آزمون‌ها و درک بهتر نیروهای منجر به یک روش آزمون خاص یاری دهند.

۳. آن‌ها تفکر مبتنی بر تکرار را ترغیب می‌کنند. هر راهکاری یک حیطه‌ی جدید ایجاد می‌کند که در آن، مسائل جدیدی قابل حل است.

گرچه این محاسن «ملایم» هستند، نباید به آن‌ها به دیده تحقیر نگریست. بسیاری از آزمون‌های نرم‌افزار، حتی طی دهه گذشته، فعالیتی برنامه‌ریزی نشده بوده است. اگر الگوهای آزمون بتوانند تیم نرم‌افزاری را در برقراری ارتباط درباره‌ی اثربخش‌ترکردن آزمون؛ در فهم نیروهای محرکی که منجر به رویکرد آزمون می‌شوند؛ و در نزدیک شدن به طراحی آزمون‌ها به‌عنوان فعالیتی تکاملی یاری دهند که در آن، هر دور تکرار به مجموعه‌ی کاملی از موارد آزمون منجر می‌شود، در آن صورت، این الگوها کار بسیاری انجام داده‌اند.

الگوهای آزمون بسیار شبیه به الگوهای طراحی توصیف می‌شوند (فصل ۱۲) ده‌ها الگوی آزمون در نوشتار پیشنهاد شده‌اند (مانند [Mar02]). سه الگویی که برای آزمون (و فقط چکیده‌ای از آن‌ها) در زیر ارائه شده‌اند، مثال‌هایی نمونه به شمار می‌روند:

نام الگو: **آزمون جفتی**

چکیده: **آزمون جفتی**، که الگویی فرایندگراست، تکنیکی را توصیف می‌کند که مشابه با برنامه‌نویسی جفتی است (فصل ۳) از این لحاظ که در آن دو آزمون‌گر با هم کار می‌کنند تا یک سری آزمون را طراحی کنند که برای فعالیت‌های مربوط به آزمون واحدها، آزمون انسجام و آزمون اعتبارسنجی قابل استفاده باشند.

نام الگو: **واسط آزمون جداگانه**

چکیده: همه‌ی کلاس‌های یک سیستم شیء‌گرا «از جمله، کلاس‌های درونی» (یعنی کلاس‌هایی که هیچ واسطی به خارج از مؤلفه‌ی استفاده کننده از آن‌ها ندارند) باید آزمایش شود. **الگوی واسط آزمون جداگانه**، چگونگی ایجاد واسط آزمونی را شرح می‌دهد که می‌توان آن را در توصیف آزمون‌های ویژه‌ای به‌کار برد که تنها به‌صورت داخلی برای مؤلفه قابل دیدن هستند» [Lan01]

نام الگو: **آزمون سناریوها**

چکیده: هنگامی که آزمون‌های واحد و انسجام انجام شدند، لازم است تعیین شود که آیا نرم‌افزار به شیوه‌ای اجرا می‌شود که کاربران را راضی کند. الگوی **آزمون سناریوها** تکنیکی برای تمرین‌دادن نرم‌افزار از دیدگاه کاربر توصیف می‌کند. شکستی در این سطح نشان می‌دهد که نرم‌افزار نتوانسته است خواسته‌های مشهود کاربر را برآورده سازد [Kan01].

بحث جامعی درباره الگوهای آزمون، از حوصله‌ی این کتاب خارج است. در صورت علاقه‌ی بیشتر، [Bin99] و [Mar02] را برای اطلاعات بیشتر درباره این مبحث ببینید.

مرجع وب

در وب‌سایت زیر می‌توانید الگوهایی را بیابید که بازدهی سازمانی آزمون، راهبرد و حل مسأله را توصیف می‌کنند:
www.testing.com/test-patterns/patterns/

۱۸-۱۰ خلاصه

هدف اصلی برای طراحی مورد آزمون، به‌دست آوردن مجموعه‌ای از آزمون است که با احتمال بیشتری خطاهای نرم‌افزار را کشف می‌کند. برای دستیابی به این هدف، دو تکنیک متفاوت به‌کار می‌رود: آزمون جعبه سفید و آزمون جعبه سیاه.

آزمون جعبه سفید بر ساختار کنترلی برنامه تکیه دارد. موارد آزمونی به‌دست می‌آیند تا اطمینان حاصل شود همه‌ی دستورات برنامه حداقل یک بار طی آزمون اجرا شده‌اند و همه‌ی شرط‌های منطقی امتحان شده‌اند. آزمون مسیرهای پایه که یک تکنیک جعبه سفید است، از گراف‌های برنامه (با معیارهای گرافی) برای به‌دست آوردن مجموعه‌ای از آزمون‌های مستقل خطی استفاده می‌کنند، که پوشش کلیه حالت‌ها را تضمین می‌کند. آزمون شرط‌ها و جریان داده‌ها و منطق برنامه را مورد امتحان قرار می‌دهد و آزمون حلقه‌ها با فراهم آوردن رویه‌ای جهت اجرای حلقه‌هایی با پیچیدگی متنوع، مکمل تکنیک‌های دیگر جعبه سفید است.

هتزل [Het84] آزمون جعبه سفید را به‌عنوان «آزمون در ابعاد کوچک» توصیف می‌کند. منظور وی این‌است‌که آزمون‌های جعبه سفیدی که در این فصل به آنها پرداخته شد، معمولاً در مورد قطعات کوچکی از برنامه‌ها (مثلاً پیمانه‌ها یا گروه‌های کوچکی از پیمانه‌ها) قابل اجرا هستند. از طرف دیگر، آزمون جعبه سیاه حوزه عمل را وسعت بخشیده می‌توان آنها را «آزمون در ابعاد بزرگ» نام نهاد.

آزمون‌های جعبه سیاه برای اعتبارسنجی خواسته‌های عملیاتی بدون در نظر گرفتن کارهای داخلی برنامه طراحی می‌شوند. تکنیک‌های آزمون جعبه سیاه بر دامنه اطلاعاتی نرم‌افزار تأکید دارد. برای این منظور، با استفاده از افراز دامنه ورودی و خروجی برنامه، موارد آزمونی را ایجاد می‌کند. در افراز هم‌ارزی، دامنه ورودی به دسته‌هایی از داده‌ها تقسیم می‌شوند که احتمال تمرین با عملکرد مشخصی از نرم‌افزار را بیشتر می‌کنند. تحلیل مقادیر مرزی، توانایی برنامه در کنترل داده‌ها را در مرزهای داده‌های قابل قبول مورد سنجش قرار می‌دهد. آزمون آرایه متعامد یک روش اثربخش و سیستماتیک برای آزمودن سیستم‌هایی فراهم می‌آورند که تعداد پارامترهای ورودی آن اندک است.

روش‌های آزمون ویژه شامل گستره وسیعی از قابلیت‌های نرم‌افزار و زمینه‌های کاربرد می‌شود. آزمون‌های مربوط به واسط‌های گرافیکی کاربر، معماری‌های کلاینت / سرور، مستندات و امکانات راهنما و سیستم‌های بی‌درنگ، هر یک نیاز به‌دستورالعمل‌های خاص برای آزمون نرم‌افزار دارند.

نرم‌افزارنویسان کارآزموده غالباً می‌گویند «آزمون هیچ‌گاه پایان ندارد، بلکه فقط از شما (مهندس نرم‌افزار) به مشتری منتقل می‌شود. هر بار که مشتری از برنامه استفاده کند، یک آزمون انجام شده است». مهندس نرم‌افزار با به‌کارگیری طراحی موارد آزمون می‌تواند به آزمون کامل‌تری دست پیدا کند و درنتیجه بیشترین تعداد خطاها را قبل از شروع «آزمون‌های مشتری» کشف و تصحیح کند.

مسائل و نکاتی برای تعمق

۱۸-۱ مایرز [Mye79] از برنامه زیر به‌عنوان یک خودآزمایی برای توانایی شما در مشخص‌کردن آزمون‌های مناسب استفاده می‌کند: برنامه‌ای سه عدد صحیح را می‌خواند. این سه مقدار به‌عنوان طول اضلاع یک مثلث در نظر گرفته می‌شوند. برنامه پیامی چاپ می‌کند مبنی بر اینکه این مثلث متساوی الاضلاع است، متساوی الساقین است یا مختلف الاضلاع. مجموعه‌ای از موارد آزمون تهیه کنید که احساس می‌کنید این برنامه را به قدر کافی مورد آزمایش قرار می‌دهند.

۱۸-۲ برنامه شرح داده شده در مسأله ۱-۱۸ را طراحی و پیاده‌سازی کنید (با در نظر گرفتن خطاها در جایی که امکان داشته باشد). یک گراف جریان برای برنامه رسم کنید و آزمون مسیرهای پایه را برای توسعه موارد آزمونی به کار برید که آزمایش کلیه دستورات برنامه را تضمین کنند. این موارد را اجرا کرده نتایج را نشان دهید.

۱۸-۳ آیا می‌توانید اهداف دیگری برای آزمون برشمرید که در بخش ۱-۱-۱۸ ذکر نشده باشد؟

۱۸-۴ مؤلفه‌ی نرم افزاری را که خودتان به تازگی طراحی و پیاده‌سازی کرده‌اید انتخاب کنید. یک مجموعه موارد آزمون برای آن طراحی کنید که به کمک آن‌ها و با آزمون مسیرهای پایه بتوان اطمینان یافت همه‌ی دستورات اجرا شده‌اند.

۱۸-۵ یک ابزار نرم‌افزاری را مشخص، طراحی و پیاده‌سازی کنید که پیچیدگی سیکلوماتیک را برای یک زبان برنامه‌نویسی موردنظر خودتان محاسبه کند. در طراحی خود، از ماتریس گراف به‌عنوان ساختمان داده استفاده کنید.

۱۸-۶ کتاب بیزر [Bei95] یا هر منبع دیگری را که در این زمینه در وب می‌یابید (مثل www.laynetworks.com/Discrete%20Mathematics_1g.htm) بخوانید و تعیین کنید چگونه برنامه‌ای که در مسأله ۵-۱۸ ساخته‌اید، قابل بسط و گسترش برای در نظر گرفتن وزن پیوندهای گوناگون است. ابزار خود را برای پردازش احتمالات اجرا یا زمان پردازش پیوند، بسط دهید.

۱۸-۷ یک ابزار خودکار طراحی کنید که حلقه‌ها را شناسایی کرده آنها را مانند بخش ۳-۵-۱۸ گروه‌بندی کند.

۱۸-۸ ابزار شرح داده شده در مسأله ۷-۱۸ را طوری گسترش دهید که پس از برخورد به هر حلقه، موارد آزمونی برای آن تولید کند. لازم است این عمل به‌طور تعاملی با آزمون‌گر اجرا شود.

۱۸-۹ حداقل سه مثال بزنید که در آنها آزمون جعبه سیاه این احساس را بدهد که همه چیز خوب است، ولی آزمون‌های جعبه سفید مشخص کنند که ممکن است خطایی در کار باشد. حداقل سه مثال بیاورید که در آنها آزمون جعبه سفید این احساس را ایجاد کند که همه چیز خوب است، ولی آزمون‌های جعبه سیاه نشان دهند که ممکن است خطایی وجود داشته باشد.

۱۸-۱۰ آیا آزمون جامع (حتی اگر برای برنامه‌های بسیار کوچک امکان‌پذیر باشد) متضمن آن است که برنامه ۱۰۰٪ درست است؟

۱۸-۱۱ جزوه راهنمای کاربر (یا راهنمای سیستم) را برای برنامه‌ای که با آن آشنایی دارید، مورد آزمایش قرار دهید و حداقل یک خطا در مستندات آن بیابید.

فصل ۱۹

آزمون برنامه‌های شیءگرا

نگاهی گذرا

آزمون برنامه‌های شیءگرا چیست؟ معماری نرم‌افزارهای شیءگرا منجر به زیرسیستم‌های لایه‌ای می‌شود که کلاس‌های همکار را بسته‌بندی می‌کنند. هر یک از این عناصر سیستمی (زیرسیستم‌ها و کلاس‌ها) وظایفی را اجرا می‌کنند که به‌دستیابی خواسته‌های سیستم کمک می‌کنند. لازم است سیستم شیءگرا را در چند سطح متفاوت مورد آزمون قرار دهیم تا خطاهایی را که ممکن است در نتیجه‌ی همکاری کلاس‌ها با یکدیگر یا ارتباط زیرسیستم‌ها از طریق لایه‌های معماری رخ دهند، کشف کنیم.

چه کسی آن را انجام می‌دهد؟ آزمون شیءگرا توسط متخصصان آزمون و مهندسان نرم‌افزار انجام می‌شود.

چرا اهمیت دارد؟ باید برنامه را پیش از آنکه به‌دست مشتری برسد، اجرا کنید. با این نیت که همه‌ی خطاها را حذف کنید تا مشتری ناراضی نباشد. برای یافتن بالاترین تعداد ممکن خطاها، آزمون‌ها باید به‌طور سیستماتیک اجرا شوند و موارد آزمون باید با استفاده از تکنیک‌های منظم طراحی شوند.

مراحل کار کدام است؟ آزمون شیءگرا از نظر راهبردی مشابه آزمون سیستم‌های معمولی است، ولی از لحاظ تاکتیکی تفاوت دارد. چون مدل‌های تحلیل و طراحی شیءگرا از لحاظ ساختار و محتویات با برنامه شیءگرا حاصل شباهت دارند، «آزمون» با مرور این مدل‌ها آغاز می‌شود. هنگامی که کدها نوشته شدند، آزمون شیءگرا «به صورت کوچک» با آزمون کلاس‌ها شروع می‌شود. آزمون‌هایی طراحی می‌شوند که با عملیات کلاس‌ها تمرین می‌کنند و بررسی می‌کنند که آیا در همکاری کلاسی با کلاس دیگر خطا وجود دارد یا خیر. به موازاتی که کلاس‌ها مجتمع می‌شوند تا یک زیرسیستم را تشکیل دهند، آزمون مبتنی بر نخ، مبتنی بر کاربرد و خوشه‌ای همراه با روش‌های مبتنی بر خطا، اعمال می‌شوند تا با کلاس‌های همکار به‌طور کامل تمرین شود. سرانجام، use caseها (که به‌عنوان بخشی از مدل تحلیل شیءگرا توسعه یافته‌اند) برای کشف خطاها در سطح اعتبارسنجی به‌کار می‌روند.

محصول کاری چیست؟ مجموعه‌ای از موارد آزمون، که برای تمرین دادن کلاس‌ها، همکاری آنها و رفتار آنها طراحی و مستندسازی شده‌اند؛ نتایج موردانتظار تعیین و نتایج واقعی ثبت می‌شوند.

چگونه اطمینان حاصل کنم که از عهده امور برآمده‌ام؟ هنگامی که آزمون را آغاز می‌کنید، دیدگاه خود را عوض کنید. سخت کوشش کنید تا نرم‌افزار را بشکنید! موارد آزمون را به شیوه‌ای بسیار منظم طراحی کنید و آنها را مرور کنید تا کامل باشند.

در فصل ۱۸ گفتیم که هدف آزمون به بیانی ساده، یافتن حداکثر تعداد خطاها با مقدار مشخصی تلاش در یک دوره زمانی واقع‌بینانه است. گرچه این هدف بنیادی برای نرم‌افزارهای شیءگرا همچنان به قوت خود باقی می‌ماند، ماهیت برنامه‌های OO، راهبرد و تاکتیک آزمون را تحت تأثیر قرار می‌دهد.

ممکن است چنین استدلال شود که به موازات رشد تحلیل و طراحی شیءگرا، استفاده‌ی مجدد از الگوهای طراحی، باعث تسهیل نیاز به آزمون سنگین سیستم‌های شیءگرا می‌شود. دقیقاً عکس این مطلب درست است. بایندر [Bin94b] در این مورد چنین نظر می‌دهد:

> هر بار استفاده‌ی مجدد، حال و هوای جدیدی دارد و آزمون دوباره را ایجاب می‌کند. به نظر می‌رسد که برای رسیدن به قابلیت بالا در سیستم‌های شیءگرا، به آزمون بیشتری نیاز است.

برای آزمون مناسب سیستم‌های شیءگرا، سه کار باید انجام شود: (۱) تعریف آزمون باید گسترش داده شود تا تکنیک‌های کشف خطای به‌کاررفته در مدل‌های تحلیل و طراحی شیءگرا را در بر گیرد، (۲) راهبرد مربوط به آزمون واحدها و انسجام باید به‌طور معنی‌دار تغییر کند و (۳) در طراحی موارد آزمون باید خصوصیات منحصر به فرد نرم‌افزار شیءگرا مد نظر قرار گیرد.

۱-۱۹ وسعت بخشیدن به دیدگاه آزمون

ساخت نرم‌افزارهای شیءگرا با ایجاد مدل‌های خواسته‌ها (تحلیل) و طراحی آغاز می‌شود[1]. به‌دلیل ماهیت تکاملی الگوی مهندسی نرم‌افزار شیءگرا، این مدل‌ها به‌عنوان نمایش‌های نسبتاً غیررسمی از خواسته‌های سیستم شروع می‌شوند و به مدل‌های مشروحی از کلاس‌ها، ارتباطات میان کلاس‌ها، طراحی و تخصیص سیستم و طراحی اشیاء تکامل می‌یابند. در هر مرحله، می‌توان این مدل‌ها را مورد آزمون قرار داد تا خطاها پیش از انتشار یافتن در تکرار بعدی، کشف شوند.

می‌توان استدلال کرد که مرور مدل‌های تحلیل و طراحی شیءگرا از آن رو مفید است که همان ساختارهای معنایی (مثل کلاس‌ها، صفات، عملیات، پیام‌ها) در سطح تحلیل، طراحی و کدنویسی ظاهر می‌شوند. بنابراین یک مشکل در تعریف صفات کلاسی که طی تحلیل کشف می‌شود، اثرات جانبی را برطرف می‌کند که ممکن است در صورت کشف نشدن مشکل تا زمان طراحی یا کدنویسی رخ دهند.

برای مثال، کلاسی را در نظر بگیرید که در آن چند صفت طی اولین تکرار تحلیل شیءگرا تعریف می‌شوند. یک صفت اضافی به کلاس الحاق می‌شود (به‌دلیل سوءتفاهم ایجاد شده در دامنه مسأله). سپس دو عملیات برای دستکاری آن صفت تعریف می‌شود. مرور صورت می‌پذیرد و کارشناس دامنه، متوجه خطا می‌شود. با حذف صفت اضافی در این مرحله، از مشکلات و تلاش‌های بیهوده زیر در اثنای تحلیل جلوگیری خواهد شد:

۱. ممکن است زیرکلاس‌های خاصی برای جای‌دادن آن صفت اضافه یا استثناهای مربوط به آن تولید شده باشد. از کار بیهوده برای ایجاد این زیرکلاس‌های غیرضروری پرهیز شده است.
۲. سوءتعبیر از تعریف کلاس ممکن است به روابط نادرست یا بیهوده میان کلاس‌ها بینجامد.
۳. رفتار سیستم یا کلاس‌های آن ممکن است به‌طور نامناسب مشخص شود تا بتواند آن صفت اضافی را در خود جای دهد.

[1] تکنیک‌های مدل‌سازی خواسته‌ها و طراحی در بخش دوم این کتاب ارائه شدند. مفاهیم پایه‌ی شیءگرایی در پیوست ۲ ارائه خواهد شد.

اندرز

گرچه مرور تحلیل شیءگرا و مدل‌های تحلیل، بخش جدایی‌ناپذیری از «آزمود» یک برنامه‌ی کاربردی شیءگراست، باید بدانید که خود به‌تنهایی کافی نیست. باید از آزمون‌های قابل اجرا نیز استفاده کنید.

اگر خطا در مرحله تحلیل کشف نشود و باز هم انتشار یابد، ممکن است مشکلات زیر طی مرحله طراحی رخ دهد (و به‌دلیل مرور قبلی می‌شد از آن پرهیز کرد):

۱. تخصیص نامناسب کلاس به زیرسیستم و/یا وظایف ممکن است طی طراحی سیستم رخ دهد.
۲. ممکن است کار طراحی غیرضروری، صرف ایجاد طراحی رویه‌ای برای عملیاتی شود که به آن صفت اضافی مربوط می‌شوند.
۳. مدل پیام‌رسانی نادرست خواهد بود (زیرا باید پیام‌هایی برای اعمال اضافی طراحی شوند).

اگر خطا در هنگام طراحی تشخیص داده نشود و به فعالیت کدنویسی راه پیدا کند، تلاش زیادی صرف تولید کدی می‌شود که یک صفت اضافی و دو عملیات غیرضروری را پیاده‌سازی می‌کند و بسیاری از مشکلات دیگر پیش می‌آید. به‌علاوه، آزمون کلاس زمان زیادی می‌برد. وقتی مشکلی کشف نشد، سیستم باید اجرا شود تا اثرات جانبی ناشی از تغییر مشخص شوند.

طی آخرین مراحل توسعه، مدل‌های تحلیل شیءگرا و طراحی شیءگرا اطلاعاتی اساسی درباره ساختار و رفتار سیستم فراهم می‌آورند. از این رو، این مدل‌ها را باید پیش از تولید کد، مرور کرد.

همه‌ی مدل‌های شیءگرا باید از لحاظ درستی، کامل بودن و سازگاربودن در حیطه نحوی، معنایی و واقع‌گرایانه بودن مدل [Lin94a] مورد آزمون قرار گیرند (در این جا، واژه *آزمون* برای مجموعه مرورهای فنی به‌کار می‌رود).

> «ابزارهایی که استفاده می‌کنیم تأثیری عمیق (و منحرف کننده!) بر عادت‌های فکری ما و بنابراین، بر توانایی‌های فکری ما دارند.»
>
> آدزگار دیکسترا

۲-۱۹ آزمون مدل‌های تحلیل شیءگرا و طراحی شیءگرا

مدل‌های تحلیل و طراحی را نمی‌توان به صورت سنتی آزمود. زیرا آنها را نمی‌توان اجرا کرد. ولی، مرورهای فنی رسمی (فصل ۸) را می‌توان برای بررسی درستی و سازگاری مدل‌های تحلیل و نیز مدل‌های طراحی به‌کار برد.

۱-۲-۱۹ درستی مدل‌های تحلیل شیءگرا و طراحی شیءگرا

نمادگذاری و نحوی که در نمایش مدل‌های تحلیل و طراحی به‌کار می‌رود، با نوع روش تحلیل و طراحی انتخاب شده برای پروژه، رابطه‌ای تنگاتنگ دارد. از این رو، درستی نحوی، براساس استفاده‌ی درست از نمادها قضاوت می‌شود و هر یک از مدل‌ها مرور می‌شود تا اطمینان حاصل شود که قراردادهای متعارف مدل‌سازی رعایت شده‌اند.

در اثنای تحلیل و طراحی، درستی معنایی را باید براساس تطابق مدل با دامنه‌ی مسأله در جهان واقعی، قضاوت کرد. اگر مدل، جهان واقعی را به درستی منعکس کند (تا سطحی از جزئیات که مناسب مرحله‌ای از توسعه باشد که در آن مدل مرور شده است)، در آن صورت از نظر معنایی درست است. برای تعیین اینکه آیا مدل، جهان واقعی را منعکس می‌سازد یا خیر، باید آن را به‌کارشناسان دامنه مسأله عرضه کرد که تعاریف و سلسله مراتب کلاس‌ها را از لحاظ کمبودها و ابهامات بررسی کنند. روابط میان کلاس‌ها (اتصالات میان نمونه‌ها) مورد ارزیابی قرار می‌گیرد تا تعیین شود که آیا آنها اتصالات با جهان واقعی را به درستی منعکس می‌کنند یا خیر[1].

[1] use caseها می‌توانند در ردگیری مدل‌های تحلیل و طراحی در مقایسه با سناریوهای کاربرد در جهان واقعی برای سیستم‌های شیءگرا بی‌اندازه ارزشمند باشند.

۲-۲-۱۹ سازگاری مدل‌های شیءگرا

سازگاری مدل‌های تحلیل شیءگرا و طراحی شیءگرا را می‌توان با درنظرگرفتن روابط میان موجودیت‌های مدل، مورد قضاوت قرار داد. مدل ناسازگار، دارای نمایش‌هایی در یک بخش است که در بخش‌های دیگر مدل به درستی منعکس نمی‌شوند [McG94].

برای ارزیابی سازگاری، هر کلاس و اتصالات آن با کلاس‌های دیگر را باید بررسی کرد. مدل کلاس - مسؤولیت - همکاری (CRC) و نمودار روابط میان اشیاء را می‌توان برای تسهیل این فعالیت به‌کار برد. چنان که در فصل ۲۱ متذکر شدیم، مدل CRC روی کارت‌های شاخص CRC تشکیل می‌شود. روی هر کارت CRC، نام کلاس، مسؤولیت‌ها (عملیات) آن و همکارهای آن (کلاس‌های دیگری که به آنها پیام ارسال می‌کند و برای انجام مسؤولیت‌های خود به آنها وابسته است) نوشته شده است. همکاری حاکی از روابط (یعنی اتصالات)میان کلاس‌های سیستم شیءگرا است. مدل شیء-رابطه، یک نمایش گرافیکی از اتصالات میان کلاس‌ها را ارائه می‌کند. همه‌ی این اطلاعات را می‌توان از مدل تحلیل (فصل‌های ۶ و ۷) به‌دست آورد.

برای ارزیابی مدل کلاس‌ها، مراحل زیر توصیه شده است [McG94]:

۱. **مرور مدل CRC و مدل رابطه‌ای میان اشیاء.** حصول اطمینان از اینکه کلیه مشارکت‌های بیان‌شده در مدل تحلیل شیءگرا در هر دو منعکس شده‌اند.

۲. **بازرسی توصیف هر یک از کارت‌های شاخص CRC برای تعیین اینکه آیا مسؤولیتِ تفویض شده بخشی از تعریف مشارکت کننده هست یا خیر.** برای مثال، کلاسی را در نظر بگیرید که برای یک سیستم کنترل نقطه فروش تعریف شده است و فروش اعتباری نام دارد. کارت شاخص این کلاس در شکل ۱-۱۹ آمده است.

Class name: Credit sale	
Class type: Transaction event	
Class characteristics: Nontangible, atomic, sequential, permanent, guarded	
Responsibilities:	Collaborators:
Read credit card	Credit card
Get authorization	Credit authority
Post purchase amount	Product ticket
	Sales ledger
	Audit file
Generate bill	Bill

شکل ۱-۱۹ مثالی از کارت‌های شاخص use case برای مرور.

برای این مجموعه از کلاس‌ها و مشارکت‌ها می‌پرسیم که آیا مسؤولیتی (مثلاً خواندن کارت اعتباری) که به یک مشارکت کننده تفویض شده است (**CreditCard**)، انجام می‌گیرد یا خیر. به عبارت دیگر، آیا کلاس **CreditCard** دارای عملی هست که خواندن آن را امکان‌پذیر کند؟

در این مورد، پاسخ مثبت است. مدل روابط میان اشیاء به‌طور عرضی کنترل می‌شود تا اطمینان حاصل شود که کلیه این گونه اتصالات معتبرند.

۳. **معکوس کردن اتصال برای حصول اطمینان از این که هر همکاری که از آن سرویسی درخواست شود، درخواست‌ها را از یک منبع مناسب دریافت می‌کنند.** برای مثال، اگر کلاس **CreditCard** درخواست *Purchase amount* را از کلاس **CreditSale** دریافت کند، مشکل ایجاد می‌شود. **CreditCard** از محتویات *Purchase amount* آگاه نیست.

۴. **استفاده از اتصالات معکوس در مرحله ۳، تعیین اینکه آیا کلاس‌های دیگری ممکن است موردنیاز باشد یا خیر، آیا مسؤولیت‌ها به‌طور مناسب در میان کلاس‌ها تقسیم شده‌اند یا خیر.**

۵. **تعیین اینکه آیا می‌توان مسؤولیت‌هایی را که به‌طور گسترده درخواست می‌شوند، در یک مسؤولیت تلفیق کرد یا خیر.** برای مثال، در هر وضعیتی *read credit card* و *get authorization* ظاهر می‌شود. آنها را می‌توان در یک مسؤولیت تحت عنوان *validate credit request* تلفیق کرد که گرفتن شماره کارت اعتباری را با اخذ مجوز ترکیب می‌کند.

مراحل ۱ تا ۵ به‌طور تکراری برای هر کلاس و در هر بار تکامل مدل تحلیل شیءگرا تکرار می‌شوند.

هنگامی که مدل طراحی شیءگرا (فصل ۲۲) ایجاد شد، مرور طراحی سیستم و طراحی اشیاء نیز باید اجرا گردد. طراحی سیستم، تصویرگر معماری کلی محصول، زیرسیستم‌های تشکیل‌دهنده‌ی آن، شیوه تخصیص زیرسیستم‌ها به پردازنده‌ها، تخصیص کلاس‌ها به زیرسیستم‌ها و طراحی واسط کاربر است. مدل اشیاء، جزئیات هر کلاس و فعالیت‌های پیام‌رسانی موردنیاز برای پیاده‌سازی مشارکت میان کلاس‌ها را عرضه می‌کند.

طراحی سیستم به این صورت مرور می‌شود که: مدل، رفتار اشیایی که در طی تحلیل شیءگرا توسعه یافته است و رفتار سیستمی موردنظر را روی زیرسیستم‌هایی نگاشت می‌کند که برای دستیابی به این رفتار طراحی شده‌اند. همزمانی و تخصیص وظایف نیز در حیطه رفتار سیستم مرور می‌شود. حالت‌های رفتاری سیستم ارزیابی می‌شود تا تعیین شود کدام‌ها به‌طور همزمان وجود دارند. از سناریوهای موارد آزمون برای امتحان کردن طراحی واسط کاربر استفاده می‌شود.

مدل اشیاء باید در برابر شبکه‌ی روابط میان اشیاء آزموده شود تا اطمینان حاصل شود که همه‌ی اشیای طراحی، حاوی صفات و عملیات لازم برای پیاده‌سازی مشارکت‌های تعریف شده برای هر کارت شاخص CRC هستند. به‌علاوه، مشخصات مشروح جزئیات عملیاتی (یعنی الگوریتم‌هایی که عملیات را پیاده‌سازی می‌کنند) با استفاده از تکنیک‌های بازرسی سنتی مرور می‌شوند.

۳-۱۹ راهبردهای آزمون شیءگرا

چنان که در فصل ۱۸ گفته شد، راهبرد کلاسیک برای آزمون نرم‌افزارهای کامپیوتری با «آزمون در مقیاس کوچک» آغاز می‌شود و به سمت «آزمون در مقیاس بزرگ» حرکت می‌کند. در قاموس آزمون‌های نرم‌افزار (فصل ۱۸)، با آزمون واحدها شروع می‌کنیم، به سمت آزمون انسجام پیش می‌رویم و با آزمون سیستم و اعتبارسنجی، کار را خاتمه می‌دهیم. در کاربردهای سنتی، آزمون واحد بر کوچکترین مؤلفه‌ی برنامه که قابل کامپایل کردن باشد - زیربرنامه (یعنی پیمانه، زیررویه، رویه، مؤلفه)

تأکید دارد. هنگامی که همه‌ی این واحدها تک‌تک مورد آزمون قرار گرفتند، ساختار کلی برنامه مجتمع می‌شود و آزمون رگرسیون اجرا می‌شود تا خطاهای ناشی از ایجاد واسط میان پیمانه‌ها و اثرات جانبی ناشی از افزایش واحدهای جدید کشف شود. سرانجام، کل سیستم آزموده می‌شود تا اطمینان حاصل شود که خطاهای موجود در خواسته‌ها کشف شده‌اند.

۱-۳-۱۹ آزمون واحدها در حیطه شیءگرا

در نرم‌افزارهای شیءگرا مفهوم واحد فرق می‌کند. بسته‌بندی، تعریف کلاس‌ها و اشیاء را راهبری می‌کند. این بدان معنا است که هر کلاس و هر نمونه از یک کلاس (شیء) صفات (داده‌ها) و عملیاتی را که این داده‌ها را دستکاری می‌کنند، بسته‌بندی می‌کند. به جای آزمون یک پیمانه منفرد، کوچکترین واحد قابل آزمون، کلاس یا شیء بسته‌بندی شده است. چون کلاس می‌تواند حاوی چند عملیات متفاوت باشد و یک عملیات خاص ممکن است به‌عنوان بخشی از چند کلاس متفاوت وجود داشته باشد، معنای آزمون واحد تغییر می‌کند.

دیگر نمی‌توانیم یک عملیات را به‌طور جداگانه (دیدگاه سنتی آزمون واحدها) آزمایش کنیم، بلکه آن را باید به‌عنوان جزئی از یک کلاس بیازماییم. برای روشن شدن مطلب، سلسله مراتبی از کلاس‌ها را در نظر بگیرید که در آن عملیات *X( )* برای کلاس پایه تعریف می‌شود و چند زیرکلاس، آن را به ارث می‌برند. هر زیرکلاسی از عملیات *X( )* استفاده می‌کند، ولی در حیطه‌ی صفات خصوصی و عملیاتی به‌کار برده می‌شود که برای هر زیرکلاس تعریف شده‌اند. چون حیطه‌ای که عملیات *X( )* در آن به‌کار می‌رود، تغییر می‌کند، لازم است عملیات *X( )* در حیطه هر یک از زیرکلاس‌ها آزموده شود. به عبارت دیگر، آزمون عملیات *X( )* در محیط خلاء (روش سنتی آزمون واحدها) در زمینه شیءگرا بازدهی ندارد.

نکته‌ی کلیدی

کوچکترین واحد قابل آزمون در نرم‌افزارهای شیءگرا، کلاس است. آزمون کلاس‌ها توسط عملیات‌های پنهان‌سازی شده توسط کلاس و رفتار حالت‌های کلاس به پیش رانده می‌شود.

آزمون کلاس‌ها برای نرم‌افزارهای شیءگرا، هم‌ارز آزمون واحدها برای نرم‌افزارهای سنتی است[1]. برخلاف آزمون واحدها در نرم‌افزارهای سنتی که بیشتر بر جزئیات الگوریتمی یک پیمانه و داده‌هایی تأکید دارد که در میان واسط پیمانه جریان پیدا می‌کند، آزمون کلاس‌ها در نرم‌افزارهای شیءگرا، به وسیله عملیات بسته‌بندی شده توسط کلاس و رفتار حالت‌های کلاس انجام می‌شود.

۲-۳-۱۹ آزمون انسجام در حیطه‌ی شیءگرا

از آنجا که نرم‌افزارهای شیءگرا فاقد ساختار کنترلی سلسله مراتبی‌اند، راهبردهای سنتی بالا به پایین و پایین به بالا چندان معنایی ندارند. به‌علاوه، عملیات انسجام به صورت هر بار یک کلاس (روش منسجم‌سازی تدریجی سنتی) نیز به دلیل تعامل‌های مستقیم و غیر مستقیم میان مؤلفه‌های تشکیل‌دهنده‌ی کلاس، غیر ممکن است [Ber93].

دو راهبرد متفاوت برای آزمون انسجام سیستم‌های شیءگرا وجود دارد [Bin94]. اولی، یعنی *آزمون نخ‌ها*، مجموعه‌ای از کلاس‌های لازم برای پاسخ‌دهی به یک ورودی یا رویداد سیستم را مجتمع می‌کند. هر نخ به‌طور انفرادی مجتمع و آزموده می‌شود. روش انسجام دوم، یعنی آزمون مبتنی بر کاربرد، ساخت سیستم را با آزمون آن دسته از کلاس‌هایی آغاز می‌کند که از تعداد کلاس‌های سرور

[1] روش‌های طراحی موارد آزمون برای کلاس‌های شیءگرا در بخش‌های ۴-۱۹ تا ۶-۱۹ بحث خواهد شد.

اندکی استفاده می‌کنند (این کلاس‌ها را *مستقل* نیز می‌گویند). پس از آنکه کلاس‌های مستقلی آزمون شدند، لایه بعدی کلاس‌ها (موسوم به *کلاس‌های وابسته*) که از کلاس‌های مستقل استفاده می‌کنند، مورد آزمون قرار می‌گیرند. این دنباله از آزمون کلاس‌های وابسته آنقدر ادامه می‌یابد تا کل سیستم ایجاد شود. برخلاف انسجام سنتی، در صورت امکان باید از به‌کارگیری درایورها و stubها (فصل ۱۸) به‌عنوان عملیات جایگزین، پرهیز شود.

آزمون خوشه‌ای [McG94] یک مرحله از آزمون انسجام نرم‌افزار شیءگرا است. در اینجا، خوشه‌ای از کلاس‌های همکار (که توسط بررسی مدل CRC و روابط میان اشیاء تعیین می‌شوند) با طراحی موارد آزمونی که سعی در کشف خطاهای موجود در مشارکت‌ها دارند، امتحان می‌شود.

نکته‌ی کلیدی

در آزمون انسجام برای نرم‌افزارهای شیءگرا، آنچه که آزمایش می‌شود، مجموعه کلاس‌هایی است که باید به رویدادی معین پاسخ بدهند.

۳-۳-۱۹ آزمون اعتبارسنجی در حیطه شیءگرا

در سطح اعتبارسنجی یا سیستم، جزئیات اتصالات میان کلاس‌ها مطرح نمی‌شود. همانند اعتبارسنجی سنتی، اعتبارسنجی شیءگرا نیز بر عملیات قابل مشاهده کاربر و خروجی‌های قابل تشخیص او تأکید دارد. آزمون‌گر برای کمک به طرح آزمون‌های اعتبارسنجی، باید case useهایی (فصل‌های ۵ و ۶) را در نظر بگیرد که بخشی از مدل تحلیل هستند. case use سناریویی را فراهم می‌آورد که به احتمال زیاد خطاهای موجود در خواسته‌های تعامل با کاربر را می‌یابد.

روش‌های آزمون جعبه سیاه (فصل ۱۸) را می‌توان برای به‌دست آوردن آزمون‌های اعتبارسنجی به‌کار برد. به‌علاوه، موارد آزمون را می‌توان از مدل رفتار اشیاء و از نمودار جریان رویدادها که در تحلیل شیءگرا ایجاد شده است، به‌دست آورد.

۴-۱۹ روش‌های آزمون شیءگرا

معماری نرم‌افزارهای شیءگرا به یک سری زیرسیستم‌های لایه‌ای منجر می‌شود که کلاس‌های همکار را بسته‌بندی می‌کنند. هر کدام از این عناصر سیستمی (زیرسیستم‌ها یا کلاس‌ها) وظایفی را اجرا می‌کنند که به محقق شدن خواسته‌ها کمک می‌کنند. ضروری است که یک سیستم شیءگرا در انواع سطوح متفاوت آزموده شوند تا خطاهایی که ممکن است در صورت همکاری کلاس‌ها و برقراری ارتباط میان زیرسیستم‌ها در لایه‌های معماری رخ دهند، کشف گردند.

روش‌های طراحی موارد آزمون برای نرم‌افزارهای شیءگرا هنوز در حال تکامل هستند. ولی یک روش کلی برای طراحی موارد آزمون شیءگرا توسط برارد [Ber93] تعریف شده است:

۱. هر مورد آزمون را باید به‌طور انحصاری شناسایی کرد و باید ارتباط آن را با کلاسی که آزموده می‌شود، به‌طور واضح بیان کرد.
۲. هدف آزمون باید بیان گردد.
۳. برای هر آزمون، باید فهرستی از مراحل آزمون توسعه داده شود که باید حاوی موارد زیر باشد [Ber94]:

الف. فهرستی از حالت‌های مشخص شده برای شیءای که باید آزموده شود.
ب. فهرستی از پیام‌ها و عملیات که به‌عنوان پیامدی از آزمون مورد آزمون قرار می‌گیرند.
پ. فهرستی از استثنائات که ممکن است در اثنای آزمون شیء رخ دهند.

«من آزمون‌گران را همچون نگهبانان محافظ پروژه می‌بینم. ما از نرم‌افزارنویسان در برابر شکست‌ها دفاع می‌کنیم در حالی که آن‌ها، ایجاد موفقیت را کانون توجه قرار داده‌اند.»
جیمز باک

ت. فهرستی از شرایط خارجی (یعنی تغییرات در محیط خارجی نرم‌افزار که باید وجود داشته باشند تا آزمون به‌طور مناسب اجرا شود).

ث. اطلاعات مکملی که به درک یا پیاده‌سازی آزمون کمک می‌کند.

برخلاف طراحی سنتیِ موارد آزمون که توسط دیدگاه «ورودی – پردازش – خروجی» یا جزئیات الگوریتمی پیمانه‌های منفرد راهبری می‌شد، آزمون شیءگرا، بر طراحی مجموعه مناسبی از عملیات تأکید دارد که حالت‌های یک کلاس را امتحان می‌کنند.

۱-۴-۱۹ طراحی موارد آزمون در مفاهیم شیءگرا

همان‌طور که تاکنون دیده‌ایم، هدف نهایی طراحی موارد آزمون، کلاس‌های شیءگرا هستند. از آنجا که صفات و عملیات، بسته‌بندی شده‌اند، آزمودن عملیات در خارج از کلاس، معمولاً بی‌فایده است. گرچه بسته‌بندی یک مفهوم طراحی اساسی در شیءگراست، هنگام آزمون، ممکن است یک مانع جزئی ایجاد کند. همان‌طور که بایندر [Bin94a] متذکر می‌شود، «آزمون به گزارش کردن حالت انتزاعی و معین یک شیء نیاز دارد». با این حال، بسته‌بندی می‌تواند به‌دست آوردن این اطلاعات را قدری دشوار سازد. اگر عملیاتی در داخل کلاس برای گزارش کردن مقادیر صفات کلاس فراهم نیامده باشند، به‌دست آوردن تصویری از حالت شیء، ممکن است دشوار باشد.

وراثت نیز منجر به مشکلات دیگری برای طراح آزمون می‌شود. قبلاً متذکر شدیم که هر حیطه‌ی جدیدی از کاربرد، نیاز به آزمون دوباره دارد، هرچند که استفاده‌ی مجدد صورت پذیرفته است. به‌علاوه، وراثت چندگانه[1] با افزایش دادن تعداد زمینه‌هایی که نیاز به آزمون دارند، باعث پیچیده‌ترشدن آزمون می‌شود [Bin94a] . اگر زیرکلاس‌های نمونه‌سازی شده از یک کلاس پایه، در یک دامنه مسأله یکسان به‌کار گرفته شوند، این احتمال وجود دارد که موارد آزمون مربوط به کلاس پایه را بتوان هنگام آزمودن زیرکلاس به‌کار برد. ولی، اگر کلاس پایه در زمینه‌ای کاملاً متفاوت به‌کار گرفته شود، موارد آزمون کلاس پایه، کاربرد اندکی خواهند داشت و باید مجموعه جدیدی از آزمون‌ها را طراحی کرد.

۲-۴-۱۹ قابلیت به‌کارگیری روش‌های سنتی در طراحی موارد آزمون

روش‌های آزمون جعبه سفید را که در فصل ۱۸ شرح داده شد می‌توان برای عملیات تعریف شده در یک کلاس به‌کار برد. مسیر پایه، آزمون حلقه یا تکنیک‌های جریان داده‌ها می‌توانند اطمینان حاصل کنند که همه‌ی دستورهای یک عملیات، مورد آزمون قرار گرفته‌اند. ولی، ساختار فشرده‌ی بسیاری از عملیات کلاس‌ها باعث می‌شود که برخی استدلال کنند که کار صرف شده برای آزمون جعبه سفید را می‌توان صرف آزمون‌ها در سطح کلاس کرد.

روش‌های آزمون جعبه سیاه برای سیستم‌های شیءگرا همانند سیستم‌هایی که با استفاده از روش‌های سنتی مهندسی نرم‌افزار توسعه یافته‌اند، مناسب هستند. چنان که قبلاً متذکر شدیم، use caseها می‌توانند ورودی مفیدی در طراحی آزمون‌های جعبه سیاه و آزمون‌های مبتنی بر حالت فراهم آورند.

[1] مفهومی در شیءگرایی که باید با دقت فراوان به‌کاربرده شود.

مرجع وب

مجموعه‌ای عالی از مقالات و منابع مربوط به آزمون شیءگرا را می‌توانید در www.rbsc.com بیابید.

۳-۴-۱۹ آزمون مبتنی بر خطا[1]

هدف از آزمون مبتنی بر خطا در یک سیستم شیءگرا، طراحی موارد آزمونی است که احتمال بالایی در کشف خطاهای ممکن دارند. از آنجا که محصول یا سیستم باید با خواسته‌های مشتری مطابقت کند، برنامه‌ریزی مقدماتی برای اجرای آزمون مبتنی بر خطا با مدل تحلیل آغاز می‌شود. آزمون‌گر به دنبال خطاهای ممکن (یعنی جنبه‌هایی از پیاده‌سازی سیستم که ممکن است منجر به نقص شوند) می‌گردد. برای تعیین اینکه آیا این خطاها وجود دارند، موارد آزمون طراحی می‌شوند تا طراحی یا کدنویسی امتحان شود.

البته، مؤثر بودن این تکنیک‌ها به طرز تلقی آزمون‌گران از «خطای ممکن» بستگی دارد. اگر خطاهای حقیقی در یک سیستم شیءگرا، «غیرممکن» تلقی شوند، در آن صورت این روش بر تکنیک‌های آزمون تصادفی دیگر مزیتی ندارد. ولی، اگر مدل‌های تحلیل و طراحی بتوانند این دید را ایجاد کنند که احتمالاً چه اشتباهاتی ممکن است رخ دهد، در آن صورت، آزمون مبتنی بر خطا می‌تواند تعداد زیادی از خطاها را با صرف مقدار کار نسبتاً اندک بیابند.

آزمون انسجام، در جستجوی خطاهای ممکن در فراخوانی عملیات یا اتصالات پیام‌رسانی است. در این زمینه ممکن است به سه نوع خطا برخورد کنیم: نتیجه غیرمنتظره، استفاده از پیام / عملیات اشتباه، فراخوانی نادرست. برای تعیین خطاهای ممکن به هنگام اجرای توابع (عملیات)، رفتار عملیات را باید بررسی کرد.

آزمون انسجام در مورد صفات و عملیات کاربرد دارد. «رفتارهای» یک شیء توسط مقادیری تعریف می‌شوند که به صفات آن نسبت داده می‌شود. آزمون باید این صفات را امتحان کند تا معین شود که آیا برای انواع متمایزی از رفتار شیء، مقادیر مناسبی ظاهر می‌شوند یا خیر.

لازم به ذکر است که آزمون انسجام سعی می‌کند خطاهای موجود در شیء کلاینت را بیابند نه شیء سرور. به بیان دیگر، آزمون انسجام بر تعیین این نکته تأکید دارد که آیا خطاها در کد فراخوانده وجود دارند یا در کد فراخوانده شده.

نکته‌ی کلیدی

راهبرد مربوط به آزمون‌های مبتنی بر خطا این است که مجموعه‌ای از خطاهای محتمل فرض شود و سپس آزمون‌هایی برای اثبات هر فرض به‌دست آید.

کدام انواع خطاها در فراخوان عملیات‌ها و ارتباط‌های پیامی به چشم می‌خورند؟

۴-۴-۱۹ موارد آزمون و سلسله مراتب کلاس‌ها

وراثت، نیاز به آزمون کامل کلاس‌های مشتق را برطرف نمی‌کند. درواقع، وراثت می‌تواند فرایند آزمون را پیچیده کند. این وضعیت را در نظر بگیرید. کلاس **Base** حاوی عملیات *( )inherited* و *( )redefined* است. کلاس **Derived**، عملیات *( )redefined* را دوباره تعریف می‌کند تا به‌طور محلی سرویس‌دهی کند. بدون شک *( )Derived::redefined* را باید آزمود زیرا یک طراحی جدید و یک کد جدید است. ولی آیا باید *( )Derived::inherited* را دوباره آزمود؟

اگر *( )Derive::inherted* عملیات *( )redefined* را فراخوانی کند، و رفتار *( )redefined* تغییر کرده باشد، *( )Derived::inherited* ممکن است با رفتار جدید به درستی کنار نیاید. بنابراین، نیاز به آزمون‌های جدید دارد، هر چند که طراحی و کد تغییر نکرده باشد. به هر حال، لازم به ذکر است که فقط زیرمجموعه‌ای از کلیه آزمون‌های مربوط به *derived::inherited()* ممکن است لازم‌الاجرا باشند.

[1] بخش‌های ۳-۴-۱۹ تا ۶-۴-۱۹ از مقاله‌ی برایان ماریک [Mar94] و با کسب اجازه از ایشان برگرفته شده‌اند. لازم به ذکر است که روش‌های بحث‌شده در بخش‌های ۳-۴-۱۹ تا ۶-۴-۱۹ برای نرم‌افزارهای سنتی نیز قابل استفاده‌اند.

اگر بخشی از طراحی و کد مربوط به *inherited()* به *redefined()* بستگی نداشته باشد (یعنی نه آن را فراخوانی کند و نه هیچ کدی را فراخوانی کند که به‌طور مستقیم آن را فرا می‌خوانند) لازم نیست کد کلاس مشتق دوباره آزموده شود.

Base::redefined() و *Derived::redefined()* دو عملیات متفاوت با مشخصه‌ها و پیاده‌سازی‌های متفاوت هستند. هر یک از آنها دارای مجموعه‌ای از خواسته‌های آزمون هستند که از مشخصات و پیاده‌سازی به‌دست آمده‌اند. خواسته‌های آزمون به دنبال خطاهای ممکن هستند: یعنی خطاهای انسجام، خطاهای شرطی، خطاهای مرزی و غیره. ولی احتمال مشابه بودن عملیات نیز وجود دارد. خواسته‌های آزمونِ آنها همپوشانی دارد. هر چه طراحی شیءگرا بهتر باشد، همپوشانی بیشتر است. به‌دست آوردن آزمون‌های جدید برای آن دسته از خواسته‌های *Derived::redefined()* که توسط آزمون‌های *Base::redefined()* برآورده نمی‌شوند، ضرورتی ندارد.

به‌طور خلاصه، آزمون‌های *Base::redefined()* در مورد اشیای کلاس **Derived** به‌کار می‌روند. ورودی‌های آزمون ممکن است هم برای کلاس‌های پایه و هم مشتق مناسب باشند، ولی نتایج مورد انتظار ممکن است در کلاس مشتق متفاوت باشند.

نکته‌ی کلیدی

حتی اگر یک کلاس پایه به‌طور کامل آزموده شده باشد، هنوز هم باید همه‌ی کلاس‌های مشتق شده از آن را آزمون کنید.

۵-۴-۱۹ طراحی آزمون مبتنی بر سناریو در آزمون مبتنی بر خطا

دو نوع اصلی از خطاها از دست می‌رود: (۱) مشخصه‌های نادرست و (۲) تعامل‌های میان زیرسیستم‌ها. هنگامی که خطاهای مرتبط با مشخصه‌های نادرست رخ می‌دهد، محصول، خواسته مشتری را انجام نمی‌دهد. ممکن است کار اشتباهی انجام دهد یا ممکن است قابلیت عملیاتی مهمی در آن جا افتاده باشد. ولی در هر حالت، کیفیت (که مطابق با خواسته‌هاست) دچار کاستی می‌شود. خطاهای مرتبط با تعامل زیرسیستم‌ها هنگامی رخ می‌دهند که رفتار یک زیرسیستم، شرایطی را ایجاد می‌کند که باعث می‌شود زیرسیستم دیگر به شکست بینجامد.

آزمون مبتنی بر سناریو، بر آنچه که کاربر انجام می‌دهد تأکید دارد، نه بر آنچه که محصولی انجام می‌دهد. این موضوع به معنای این است که وظایف کاربر (ازطریق use case ها) به عهده گرفته شود و این وظایف و شکل‌های تغییریافته آنها به‌عنوان موارد آزمون انتخاب شوند.

نکته‌ی کلیدی

آزمون مبتنی بر سناریو، خطاهایی را آشکار خواهد کرد که وقتی رخ می‌دهند که هر کسی کار با نرم‌افزار تعامل کند.

سناریوها، خطاهای تعاملی را برملا می‌کنند. ولی برای نیل به این مقصود، موارد آزمون باید پیچیده‌تر و واقع‌بینانه‌تر از آزمون‌های مبتنی بر خطا باشد. آزمون مبتنی بر سناریو، تمایل به امتحان چندین زیرسیستم را در مورد یک آزمون انجام می‌دهد (کاربران خود را محدود نمی‌کنند که در هر زمان از یک زیرسیستم استفاده کنند).

به‌عنوان مثال، طراحی آزمون مبتنی بر سناریو را برای یک ویراستار متنی در نظر بگیرید. use case ها عبارتنداز:

use case: تثبیت پیش‌نویس نهایی

پیش‌زمینه: چاپ پیش‌نویس «نهایی»، خواندن آن و کشف خطاهای ناراحت کننده‌ای که روی صفحه‌نمایش آشکار نبوده‌اند، چیز غیرعادی نیست. این use case، رویدادهایی را توصیف می‌کند که در چنین شرایطی رخ می‌دهد.

۱. چاپ کل سند

۲. حرکت در متن سند و تغییردادن صفحات معین

۳. چاپ هر صفحه به موازاتی که تغییر می‌یابد

۴. گاهی تعدادی از صفحات چاپ می‌شود

این سناریو دو چیز را شرح می‌دهد: نیازهای آزمون و کاربر. نیازهای کاربر آشکارند: (۱) متدی برای چاپ یک صفحه و (۲) متدی برای چاپ چند صفحه. مادامی که آزمون ادامه دارد، نیاز به آزمون ویراستاری پس از چاپ (و بالعکس) وجود دارد. آزمون‌گر امیدوار است به این نتیجه برسد که عملیات printing (چاپ) باعث بروز خطاهایی در عملیات editing شود؛ به عبارت دیگر، این دو عملیات نرم‌افزاری به خوبی از هم مستقل نیستند.

use case: چاپ یک کپی جدید

پیش‌زمینه: کسی از کاربر می‌خواهد که کپی جدیدی از سند بگیرد. سند باید چاپ شود.

۱. بازکردن سند

۲. چاپ آن

۳. بستن سند

باز هم روش آزمون، نسبتاً آشکار است. مگر اینکه این سند وجود نداشته باشد. این سند در کار قبلی ایجاد شده است. آیا آن کار، این وظیفه را تحت تأثیر قرار می‌دهد؟

در بسیاری از ویراستارهای مدرن، مستندات به یاد دارند که آخرین بار چگونه چاپ شده‌اند. طبق پیش‌فرض، بار بعدی هم به همان صورت چاپ می‌شوند. پس از سناریوی **تثبیت پیش‌نویس نهایی**، فقط انتخاب Print در منو و کلیک کردن دکمه Print در کادر محاوره، باعث می‌شود تا آخرین صفحه تصحیح شده دوباره چاپ شود. بنابراین، مطابق با این ویراستار، سناریوی صحیح باید چنین باشد:

use case: چاپ یک کپی جدید

۱. بازکردن سند

۲. انتخاب Print در منو

۳. کنترل اینکه آیا چند صفحه را چاپ می‌کنید، و اگر چنین است، برای چاپ کل سند، ماوس را کلیک کنید.

۴. کلیک کردن روی دکمه Print

۵. بستن سند

ولی این سناریو یک خطای بالقوه در مشخصه را نشان می‌دهد. ویراستار، کار مورد انتظار کاربر را انجام نمی‌دهد. مشتریان غالباً مرحله سوم را نادیده می‌گیرند. سپس وقتی می‌بینید که به‌جای ۱۰۰ صفحه درخواستی، تنها یک صفحه در چاپگر چاپ شده باشد، ناراحت می‌شوند. مشتریان این اشکال‌های موجود در مشخصات را اعلام می‌کنند.

شما، به‌عنوان طراح موارد آزمون، ممکن است این وابستگی را در طراحی آزمون از دست بدهید، ولی این احتمال وجود دارد که مشکل طراحی آزمون آشکار شود. سپس آزمون‌گر باید با این پاسخ احتمالی سروکار داشته باشد که «کارها همین طوری پیش می‌رود!».

> «اگر می‌خواهید و انتظار دارید که برنامه‌ای کار کند، احتمال این که برنامه را در حال کار ببینید، بیشتر است و شکست‌ها را نخواهید دید.»
>
> سیم کانر و دیگران

اندرز

گرچه آزمون مبتنی بر سناریو مزیت‌هایی در بر دارد، با صرف زمان روی مرور use case ها به‌هنگام تهیه‌ی آن‌ها به‌عنوان بخشی از مدل تحلیل، نتیجه‌ی بیشتری عایدتان خواهد شد.

۶-۴-۱۹ آزمون ساختار سطحی و ساختار عمیق

ساختار سطحی (surface structure) به ساختار بیرونی و قابل مشاهده‌ی برنامه‌ی شیءگرا اشاره دارد. یعنی، ساختاری که بلافاصله در معرض دید کاربر نهایی است. ممکن است به‌کاربران بسیاری از

سیستم‌های شیءگرا، به جای اجرای عملکردها، اشیایی داده شود که به شیوه‌ای آنها را دستکاری کنند. ولی واسط هرچه که باشد، آزمون‌ها هنوز مبتنی بر وظایف کاربر هستند. بر عهده گرفتن این وظایف مستلزم درک، مشاهده و صحبت با کاربران است (و ارزش آن را دارد که هر تعداد از کاربران در نظر گرفته شوند).

مطمئناً در جزئیات تفاوت وجود دارد. برای مثال، در یک سیستم سنتی با واسطی که از طریق صدور فرمان کار می‌کند، کاربر ممکن است فهرستی از فرمان‌ها را به‌عنوان یک فهرست کنترلی آزمون استفاده کند. اگر هیچ سناریوی آزمونی برای امتحان فرمان وجود نداشته باشد، احتمالاً آزمون برخی از وظایف کاربر را نادیده گرفته است (یا واسط دارای فرمان‌های زاید است). در یک واسط مبتنی بر اشیاء، آزمون‌گر می‌تواند فهرستی از کلیه اشیاء را به‌عنوان فهرست کنترلی آزمون در نظر گیرد.

بهترین آزمون‌ها هنگامی به‌دست می‌آیند که طراح به شیوه‌ای جدید یا غیرسنتی به سیستم نگاه کند. برای مثال، اگر سیستم یا محصول دارای واسطی باشد که در آن از فرمان‌ها استفاده می‌شود، در صورتی که طراح آزمون وانمود کند عملیات مستقل از اشیاء هستند، آزمون‌های کامل‌تری به‌دست خواهد آمد. این پرسش‌ها را مطرح کنید: «آیا کاربر در حالی که با چاپگر کار می‌کند، می‌خواهد با این عمل، که به شیء **Scanner** مربوط می‌شود، کار کند؟» واسط هر چه باشد، طراحی مورد آزمونی که با ساختار سطحی سیستم تمرین می‌کند، باید هر دو شیء و عملیات را به‌کار گیرد.

ساختار عمیق (deep structure) به جزئیات داخلی برنامه شیءگرا می‌پردازد. یعنی ساختاری که با بررسی طراحی و / یا کد درک می‌شود. آزمون ساختار عمیق برای امتحان کردن وابستگی‌ها، رفتارها و سازوکارهای ارتباطی که به‌عنوان بخشی از طراحی سیستم و اشیاء در نرم‌افزار شیءگرا وضع می‌شوند، طراحی می‌گردد.

مدل‌های تحلیل و طراحی به‌عنوان مبنایی برای آزمون ساختار عمیق به‌کار می‌روند. برای مثال، نمودار روابط میان اشیاء یا نمودار مشارکت زیرسیستم‌ها، بیانگر مشارکت‌های میان اشیاء و زیرسیستم‌هایی است که از بیرون قابل مشاهده نیست. در آن صورت، طراح موارد آزمون ممکن است بپرسد: «آیا وظیفه‌ای را (به‌عنوان آزمون) بر عهده گرفته‌ایم که مشارکت ذکر شده در نمودار روابط میان اشیاء یا نمودار مشارکت زیرسیستم‌ها را امتحان کند؟ اگر خیر چرا؟»

نکته‌ی کلیدی

آزمون ساختار سطحی، مشابه با آزمون جعبه سیاه است. آزمون ساختار عمیق مشابه با آزمون جعبه سفید است.

۱۹-۵ روش‌های آزمون قابل اجرا در سطح کلاس‌ها

آزمون نرم‌افزار در ابعاد کوچک آغاز می‌شود و رفته رفته به سمت ابعاد بزرگ پیشرفت می‌کند. آزمون در ابعاد کوچک بر یک کلاس منفرد و متدهای موجود در آن کلاس تأکید دارد. آزمون تصادفی و افراز، روش‌هایی هستند که می‌توان آنها را برای امتحان کردن یک کلاس در اثنای آزمون شیءگرا به‌کاربرد[Kir94].

۱۹-۵-۱ آزمون تصادفی برای کلاس‌های شیءگرا

برای ارائه مثال‌های مختصری از این روش‌ها، یک برنامه کاربردی بانکداری را در نظر بگیرید که کلاس **Account** در آن حاوی عملیات‌هایی است که عبارتنداز: *open( )*، *setup( )*، *deposit( )*، *withdraw( )*، *balance( )*، *summarize( )*، *creditLimit( )* و *close( )* [Kir94]. هر یک از این

آندرز

تعداد تبدیلات جایگشتی برای آزمون تصادفی می‌تواند بسیار بزرگ باشد. برای بهبودبخشیدن به بازدهی آزمون می‌توان از راهبردی مشابه با آزمون آرایه‌های متعامد بهره برد.

داز اشتباهات خود شرمسار نشوید و از آن‌ها جرم نسازید.

کنفسیوس

عملیات را می‌توان برای **Account** اجرا کرد، ولی محدودیت‌های معینی از ماهیت مسأله برمی‌آید (مثلاً، پیش از آنکه کلیه عملیات دیگر قابل اجرا باشند، باید حساب باز شود و پس از انجام کلیه عملیات، حساب باید بسته شود). کمترین تاریخچه حیات یک نمونه از **Account** شامل عملیات زیر می‌شود:

open.setup.deposit.withdraw.close

این کمترین دنباله آزمون برای **Account** است. ولی، گستره وسیعی از رفتارهای دیگر نیز ممکن است در این دنباله رخ دهد:

open.setup.deposit.[deposit|withdraw|balance|summarize|creditLimit]n.withdraw.close

چند دنباله عملیات متفاوت را می‌توان به‌طور تصادفی تولید کرد. برای مثال:

مورد آزمون r_1:

open.setup.deposit.deposit.balance.summarize.withdraw.close

مورد آزمون r_2:

open.setup.deposit.deposit.balance.summarize.withdraw.close

این آزمون‌ها و سایر آزمون‌های تصادفی طوری بنا نهاده می‌شوند که سابقه‌های متفاوتی از حیات وراثتی کلاس‌ها را تمرین دهند.

۱۹-۵-۲ آزمون اِفراز در سطح کلاس‌ها

آزمون اِفراز، تعداد موارد آزمون لازم برای امتحان کلاس را مشابه با افراز هم‌ارزی برای نرم‌افزارهای سنتی (فصل ۱۸) کاهش می‌دهد. ورودی و خروجی گروه‌بندی می‌شوند و موارد آزمون طراحی می‌شوند تا هر گروه امتحان شود. ولی گروه‌های افراز چگونه به‌دست می‌آیند؟

افراز مبتنی بر حالت، عملیات‌های کلاس را براساس توانایی آنها در تغییردادن حالت کلاس گروه‌بندی می‌کند. با در نظر گرفتن کلاس **Account**، عملیات‌های تغییردهنده‌ی حالت شامل *deposit( )* و *withdraw( )* می‌شوند. حال آن‌که عملیاتی که حالت را تغییر نمی‌دهند شامل *balance( )*، *summarize( )* و *creditLimit( )* می‌شوند. آزمون‌ها به‌شیوه‌ای طراحی می‌شوند که عملیات تغییردهنده‌ی حالت و عملیاتی را که حالت را تغییر نمی‌دهند، به‌طور جداگانه امتحان کنند. بنابراین:

مورد آزمون p_1:

open.setup.deposit.deposit.withdraw.withdraw.close

مورد آزمون p_2:

open.setup.deposit.summarize.creditLimit.withdraw.close

مورد آزمون p_1 حالت را تغییر می‌دهد، حال آنکه مورد آزمون p_2 عملیات‌هایی را امتحان می‌کند که حالت را تغییر نمی‌دهند (غیر از آنهایی که در دنباله آزمون کمینه قرار دارند).

افراز مبتنی بر صفات، عملیات کلاس را براساس صفاتی که مورد استفاده قرار می‌دهند، گروه‌بندی می‌کند. برای کلاس **Account**، صفات *balance* و *creditLimit* را می‌توان برای تعریف افرازها به‌کار برد. عملیات به سه گروه تقسیم می‌شوند: (۱) عملیات‌هایی که از *creditLimit* استفاده می‌کنند،

کدام گزینه‌های آزمون در سطح کلاس‌ها در دسترس هستند؟

SafeHome

آزمون کلاس‌ها

صحنه: کابین شکیرا.

نقش‌آفرینان: جیمی و شکیرا- اعضای تیم مهندسی نرم‌افزار که در حال کار روی طراحی مورد آزمونی برای قابلیت امنیتی سیستم هستند.

گفتگو:

شکیرا: من چند تا آزمون برای کلاس **Detector** تهیه کردم [شکل ۴-۱۰]- می‌دانی، همان کلاسی که دستیابی به همه‌ی اشیای **Sensor** برای قابلیت امنیت را ممکن می‌کند. با آن آشنا هستی؟

جیمی (با خنده): البته، همان کلاسی که اضافه‌کردن حس‌گر «اضطراب سگی» را ممکن می‌کرد.

شکیرا: همان کلاس و فقط همان. به هر حال، با چهار تا عملیات واسط دارد: *read( )*، *enable( )* و *disable( )* و *test( )* قبل از این که بشود حس‌گری را خواند، باید فعال شود. وقتی که فعال شد، می‌شود آن را خواند و آزمود. در هر زمانی هم می‌شود آن را غیر فعال کرد مگر این که شرایط هشدار در حال پردازش باشد. بنابراین، یک سری آزمون ساده تعریف کردم که سابقه حیات رفتاری آن را تمرین بدهد. [به جیمی سری زیر را نشان می‌دهد.]

#1: enable • test • read • disable

جیمی: این جواب می‌دهد، ولی باید بیشتر از این‌ها کار کنی.

شکیرا: می‌دانم، می‌دانم. این هم چند تا سری دیگری که تهیه کردم. [به جیمی سری‌های زیر را نشان می‌دهد.]

#2: enable • test* • [read]^n • test • disable

#3: [read]^n

#4: enable*disable • [test|read]

جیمی: خب. بگذار ببینم می‌توانم بفهمم این‌ها چی هستند. شماره ۱، یک سابقه‌ی حیات عادی را طی می‌کند. یک جور کاربرد قراردادی. شماره ۲، عملیات خواندن را n بار تکرار می‌کند و این هم یک سناریوی محتمل است. شماره ۳، تلاش می‌کند حس‌گر را قبل از فعال شدن آن بخواند. این باید یک جور پیام خطا ایجاد کند، نه؟ شماره ۴ هم که حس‌گر را فعال و غیر فعال می‌کند و بعد سعی می‌کند آن را بخواند. این همان کار آزمون شماره ۲ را نمی‌کند؟

شکیرا: در واقع نه. در شماره ۴، حس‌گر فعال است. چیزی که شماره ۴ واقعاً آزمایش می‌کند، این است که آیا عملیات غیر فعال‌کردن آن طور که باید، کار می‌کند یا خیر. یک عملیات *read()* یا *test( )* بعد از *disable( )* باید پیام خطا را ایجاد کند. اگر نکرد، در عملیات *disable( )* خطا داریم.

جیمی: خیلی جالب است. فقط یادت باشد که این چهار آزمون باید بر هر نوع حس‌گر اجرا شوند، چون این عملیات‌ها ممکن است بسته به نوع حس‌گر، تفاوت‌های ظریفی داشته باشند.

شکیرا: نگران نباش. برنامه همین است.

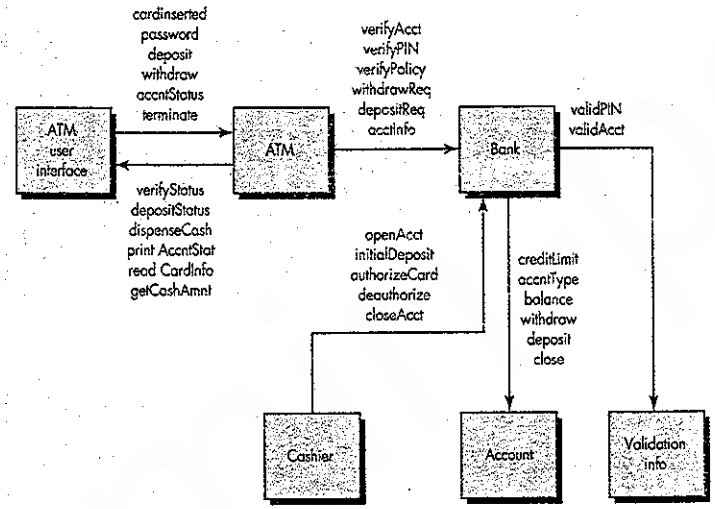
(۲) عملیات‌هایی که *creditLimit* را اصلاح می‌کنند، و (۳) عملیات‌هایی که *creditLimit* را اصلاح نمی‌کنند. سپس دنباله‌های آزمون برای هر افراز طراحی می‌شود.

افراز مبتنی بر گروه‌ها، عملیات کلاس را براساس عملکرد کلی که هر یک انجام می‌دهند، تقسیم‌بندی می‌کند. برای مثال، عملیات موجود در کلاس **Account** را می‌توان در قالب عملیات‌های آماده‌سازی (*setup ،open*)، عملیات‌های محاسباتی (*withdraw ،deposit*)، درخواست وضعیت‌ها (*balance ،creditLimit summarize*) و عملیات پایان دهنده (*close*) دسته‌بندی کرد.

۱۹-۶ طراحی موارد آزمون بین کلاس‌ها

طراحی موارد آزمون با شروع انسجام سیستم شیءگرا پیچیده‌تر می‌شود. در این مرحله، آزمون مشارکت میان کلاس‌ها باید آغاز شود. برای نشان دادن نحوه ایجاد موارد آزمون بین کلاس‌ها [Kir94]، مثال بانکداری بخش ۵-۱۹ را بسط می‌دهیم تا کلاس‌ها و مشارکت‌های شکل ۲-۱۹ را نیز دربرگیرد. جهت گروه‌ها در شکل، نشان‌گر جهت پیام‌هاست و برچسب‌ها نشان‌گر فراخوانی عملیات در نتیجه‌ی مشارکتی است که از پیام فهمیده می‌شود.

همانند آزمون کلاس‌های منفرد، آزمون مشارکت میان کلاس‌ها را می‌توان با اجرای روش‌های تصادفی و افراز و نیز آزمون مبتنی بر سناریو و آزمون رفتاری انجام داد.



شکل ۲-۱۹ نمودار مشارکت میان کلاس‌ها برای برنامه کاربردی بانکداری.

۱۹-۶-۱ آزمون کلاس‌های چندگانه

کیرانی و تسای [Kir94] برای تولید موارد آزمون تصادفیِ کلاس‌های چندگانه مراحل زیر را پیشنهاد می‌کنند:

۱. برای هر کلاس کلاینت، از عملگرهای کلاس برای تولید دنباله‌های آزمون تصادفی استفاده کنید. این عملگرها پیام‌هایی به کلاس‌های سرور دیگر ارسال خواهند کرد.

«مرزی که دامنه‌ی آزمون واحدها و آزمون انسجام را تعریف می‌کند، برای توسعه‌ی شیءگرا متفاوت است. آزمون‌ها را می‌توان در نقاط بسیاری از فرایند طراحی و اجرا کرد. از این رو، قدری طراحی، قدری کدنویسی جای خود را به قدری طراحی، قدری کدنویسی، قدری آزمون می‌دهد.»

۲. برای هر پیامی که تولید می‌شود، کلاس مشارکت کننده و عملگر مربوط را در شیء سرور تعیین کنید.

۳. برای هر عملگر در شیء سرور (که توسط پیام‌های شیء کلاینت فراخوانی شده است) پیام‌هایی را که ارسال می‌کند، تعیین کنید.

۴. برای هر یک از پیام‌ها، سطح بعدی عملگرهایی را که فراخوانی می‌شوند، تعیین کنید. آنها را در دنباله آزمون در نظر بگیرید.

به‌عنوان مثال [Kir94]، سری عملیات‌های مربوط به کلاس **Bank** را نسبت به کلاس **ATM** در نظر بگیرید (شکل ۲-۱۹):

verifyAcct.verifyPIN.[[verifyPolicy.withdrawReq]|depositReq|acciInfoREQ]n

یک مورد آزمون تصادفی برای کلاس **Bank** ممکن است چنین باشد:

مورد آزمون r_3:

verifyAcct.verifyPIN.depositReq

برای در نظرگرفتن مشارکت کننده‌های موجود در این آزمون، پیام‌های مرتبط با هر یک از عملیات ذکرشده در مورد آزمون r_3 در نظر گرفته می‌شوند. **Bank** باید با **ValidataIufo** همکاری کند تا *verifyAct( )* و *verifyPIN( )* را اجرا کند. **Bank** باید با **Account** مشارکت کند تا *depositReq( )* اجرا شود. از این رو، یک مورد آزمون جدید که مشارکت‌های فوق‌الذکر را امتحان کند، به صورت زیر خواهد بود:

مورد آزمون r_4:

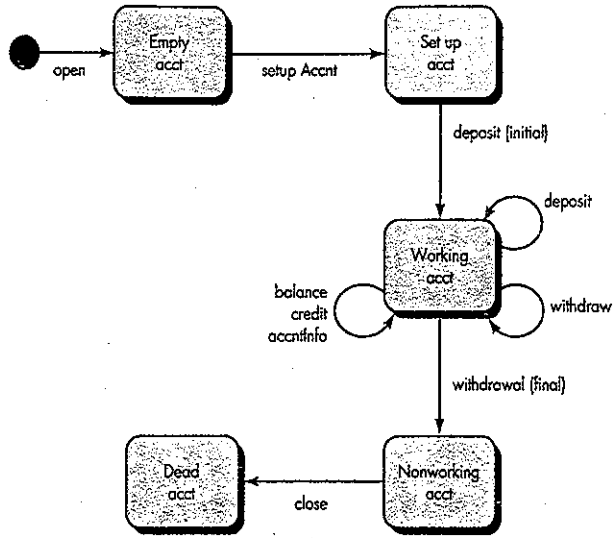
verifyAcctBank[validAcctValidationInfo].verifyPINBank

validPindataionInfo].depositReq.[depositaccount]

این رویکرد روش آزمون افراز برای کلاس‌های چندگانه، مشابه رویکرد آزمون افراز یک کلاس است. هر کلاس به صورتی که در بخش ۲-۵-۱۶ بحث شد، افراز می‌شود. ولی، دنباله آزمون بسط داده می‌شود تا آن دسته از عملیات‌هایی که از طریق پیام به کلاس‌های همکار فراخوانده می‌شوند، منظور شوند. در یک روش دیگر، آزمون‌ها براساس رابطه‌هایی که برای یک کلاس خاص وجود دارد، افراز می‌شوند. در شکل ۲-۱۹، کلاس **Bank** پیام‌ها را از کلاس‌های **ATM** و **Cashier** دریافت می‌کند. سپس متدهای موجود در **Bank** را می‌توان با افراز آن به متدهایی که به **ATM** یا به **Cashier** سرویس‌دهی می‌کنند، آزمود. افراز مبتنی بر حالت (بخش ۲-۵-۱۹) را می‌توان برای پالایش بیشتر افرازها به‌کار برد.

۲-۶-۱۹ آزمون‌های به‌دست آمده از مدل‌های رفتاری

در فصل ۷، استفاده از نمودار گذار حالت را به‌عنوان مدلی که رفتار پویای یک کلاس را نشان می‌دهد، مورد بحث قرار دادیم. به کمک نمودار گذار حالت می‌توان برای یک کلاس، دنباله‌ای از آزمون‌ها را به‌دست آورد که رفتار پویای کلاس (و کلاس‌هایی که با آن مشارکت دارند) را امتحان کند. شکل ۳-۱۹ [Kir94] یک نمودار گذار حالت برای کلاس **Account** را نشان می‌دهد که قبلاً بحث شد.



شکل ۳-۱۹ نمودار گذار حالت برای کلاس Account.

همان‌طور که از شکل پیدا است، گذارهای اولیه از حالت‌های *empty acct* (حساب خالی) و *setup acct* (تنظیم حساب) عبور می‌کنند. اغلب رفتارهای مربوط به نمونه‌های کلاس، زمانی رخ می‌نمایند که سیستم در حالت *working acct* (حساب فعال) قرار دارد. بیرون کشیدن نهایی پول از حساب و بستن آن، به ترتیب باعث گذار به حالت‌های *nonworking acct* (حساب بی‌کار) و *dead acct* (حساب مرده) می‌شود.

آزمون‌هایی که قرار است طراحی شوند، باید کلیه حالت‌ها را پوشش دهند. یعنی دنباله عملیات‌ها باید باعث شوند تا کلاس **Account** از کلیه حالت‌های مجاز گذار کند:

مورد آزمون s_1:

final).close) open.setupAccnt.deposit (initial).withdraw

لازم به ذکر است که این دنباله همسان با دنباله آزمون کمینه‌ای است که در بخش ۲-۵-۱۹ بحث شد. با افزودن دنباله آزمون‌های اضافی به دنباله کمینه خواهیم داشت:

مورد آزمون s_2:

withdraw(final).close.open.setupAccnt.deposit(initial).deposit.balance.credit

مورد آزمون s_3:

.open.setupAccnt.deposit(initial).deposit.withdraw.accntInfo.withdraw(final).close

هنوز باید موارد آزمون بیشتری به‌دست آید تا اطمینان حاصل شود که کلیه رفتارهای کلاس به‌طور مناسب مورد امتحان قرار گرفته‌اند. در شرایطی که رفتار کلاس منجر به مشارکت با یک یا چند کلاس می‌شود، از STDها برای پیگیری جریان رفتاری سیستم استفاده می‌شود.

مدل حالت را می‌توان به صورت عرضی (سطحی) پیمایش کرد [McG94] . در این مورد، روش عرضی بدان معناست که یک مورد آزمون، گذاری را تمرین می‌دهد و هنگامی که گذار جدیدی باید آزموده شود، فقط گذارهایی مورد کاربرد قرار می‌گیرند که قبلاً آزموده شده‌اند.

شیء **CreditCard** را در نظر بگیرید که بخشی از سیستم بانکداری بود. حالت اولیه **CreditCard**، نامعین است (یعنی هیچ شماره کارت اعتباری فراهم نیامده است). این شیء با خواندن کارت اعتباری در اثنای یک فروش، به حالت معین می‌رسد، یعنی صفات *card number* و *expiration date* همراه با شناسه‌های مشخصی از بانک تعریف می‌شوند. کارت اعتباری هنگامی تسلیم می‌شود که برای تأیید ارسال شود و هنگامی به تصویب می‌رسد که تأییدیه دریافت کند. گذار **CreditCard** از حالتی به حالت دیگر را می‌توان با به‌دست آوردن موارد آزمودنی که باعث رخ‌دادن گذار می‌شوند، آزمود. روش عرضی برای این نوع آزمون، پیش از امتحان حالت‌های معین و نامعین، حالت تسلیم شده را امتحان نمی‌کند. اگر چنین کند، از گذارهایی استفاده می‌کند که قبلاً آزموده نشده باشند و بنابراین، از ملاک عرضی عدول می‌کند.

۱۹-۷ خلاصه

هدف کلی آزمون شیءگرا - یافتن حداکثر تعداد خطاها با حداقل کار - همانند هدف آزمون نرم‌افزارهای سنتی است. ولی راهبرد و تاکتیک آزمون شیءگرا ، تفاوتی چشمگیر دارد. دیدگاه آزمون وسعت بیشتری می‌یابد تا مرور مدل تحلیل و نیز مدل طراحی را دربرگیرد. به‌علاوه، تأکید آزمون از مؤلفه رویه‌ای (پیمانه) به کلاس متمایل می‌شود.

از آنجا که مدل‌های خواسته‌ها و طراحی شیءگرا و کد منبع حاصل، از نظر معنایی با هم پیوسته شده‌اند، در اثنای این فعالیت‌های مهندسی آزمون (به شکل مرورهای فنی رسمی) آغاز می‌شود. به همین دلیل، مرور CRC، رابطه میان اشیاء و مدل‌های رفتار اشیاء را می‌توان به‌عنوان آزمون مرحله اول در نظر گرفت.

هنگامی که کدها در دسترس باشند، آزمون کلاس‌ها برای هر کلاس به‌کار برده می‌شود. در طراحی آزمون‌ها برای یک کلاس، انواع روش‌ها به‌کار برده می‌شود: آزمون مبتنی بر خطا، آزمون تصادفی، و آزمون افراز. هر کدام از این روش‌ها، عملیات‌های بسته‌بندی شده توسط کلاس را تمرین می‌دهد. مجموعه آزمون‌هایی طراحی می‌شوند که اطمینان حاصل شود عملیات‌های مرتبط، تمرین داده می‌شوند. وضعیت کلاس، که توسط مقادیر صفات آن نشان داده می‌شود، بررسی می‌شود تا معلوم شود آیا خطا وجود دارد یا خیر.

آزمون انسجام را می‌توان با استفاده از یک راهبرد مبتنی بر بندها یا مبتنی بر کاربرد انجام داد. آزمون مبتنی بر بند، مجموعه‌ای از کلاس‌هایی را که مشارکت می‌کنند تا به یک رویداد یا ورودی پاسخ دهند، منسجم می‌کند. آزمون مبتنی بر کاربرد، سیستم را به صورت لایه‌ای و با شروع از کلاس‌هایی ایجاد می‌کند که از کلاس‌های سرور استفاده نمی‌کنند. روش‌های طراحی موارد آزمون می‌توانند از آزمون‌های تصادفی یا افراز نیز بهره بگیرند. به‌علاوه، آزمون مبتنی بر سناریو و آزمون‌های به‌دست آمده از مدل‌های رفتاری را می‌توان برای آزمودن یک کلاس و مشارکت کننده‌های آن به‌کار برد. هر دنباله آزمون، جریان عملیات را در میان مشارکت‌های کلاس پیگیری می‌کند.

آزمون اعتبارسنجی سیستم شیءگرا، جهت‌گیری جعبه سیاه دارد و می‌توان آن را با اجرای همان روش جعبه سیاه برای نرم‌افزارهای سنتی انجام داد. ولی، آزمون مبتنی بر سناریو، اعتبارسنجی سیستم‌های شیءگرا را تعیین می‌کند و use case را به یک محرک اصلی برای آزمون اعتبارسنجی تبدیل می‌کند.

مسائل و نکاتی برای تعمق

۱۹-۱ به زبان ساده شرح دهید که چرا کلاس کوچکترین واحد مناسب برای آزمون در یک سیستم شیءگرا است.

۱۹-۲ چرا باید زیرکلاس‌هایی را که از یک کلاس موجود نمونه‌برداری می‌شوند، با وجودی که کلاس قبلاً به‌طور کامل آزموده شده است، دوباره باید آزمود؟ آیا می‌توان موارد آزمون طراحی شده برای کلاس موجود را به‌کار برد؟

۱۹-۳ چرا «آزمون» باید با فعالیت‌های تحلیل شیءگرا و طراحی شیءگرا آغاز شود؟

۱۹-۴ مجموعه‌ای از کارت‌های شاخص CRC برای *SafeHome* به‌دست آورید و مراحل ذکر شده در بخش ۱۹-۲-۲ را اجرا کنید تا تعیین شود که آیا ناسازگاری‌هایی وجود دارد؟

۱۹-۵ اختلاف میان راهبردهای مبتنی بر نخ و مبتنی بر کاربرد برای آزمون انسجام در چیست؟ آزمون خوشه‌ای چگونه در آن می‌گنجد؟

۱۹-۶ آزمون تصادفی و افراز را در مورد سه کلاس تعریف شده در طراحی سیستم *SafeHome* اجرا کنید. موارد آزمونی ایجاد کنید که نشان‌گر دنباله عملیات باشند.

۱۹-۷ آزمون کلاس‌های چندگانه و آزمون‌های به‌دست آمده از مدل رفتاری را روی طراحی *SafeHome* اجرا کنید.

۱۹-۸ با استفاده از روش‌های افراز و آزمون تصادفی و نیز آزمون کلاس‌های چندگانه‌ی به‌دست‌آمده از مدل رفتاری، چهار آزمون اضافی برای برنامه بانکداری ذکرشده در بخش‌های ۱۹-۶۵ و ۱۹-۶ به‌دست آورید.

فصل ۲۰

آزمون برنامه‌های کاربردی تحت وب

نگاهی گذرا

آزمون برنامه‌های تحت وب چیست؟ آزمون برنامه‌های تحت وب مجموعه‌ای از فعالیت‌های مرتبط است که هدفی واحد را دنبال می‌کنند: کشف خطاهای موجود در محتوا، قابلیت عملیاتی، قابلیت استفاده، قابلیت گشت‌وگذار، کارایی، ظرفیت و امنیت برنامه‌ی تحت وب. برای دستیابی به این منظور، راهبرد آزمونی به کار می‌رود که هم شامل مرور و هم آزمون‌های اجرایی می‌شود.

چه کسی آن را انجام می‌دهد؟ مهندسان وب و سایر ذی‌نفع‌های پروژه (مدیران، مشتریان و کاربران نهایی) همگی در آزمون برنامه‌ی تحت وب مشارکت دارند.

چرا اهمیت دارد؟ اگر کاربران به خطاهایی برخورد کنند که در وفاداری آنها به برنامه‌ی تحت وب خلل وارد کند، برای محتوا و قابلیت‌های مورد نیاز، به‌جای دیگر خواهند رفت و برنامه‌ی تحت وب شکست خواهد خورد. به همین دلیل، باید کار کنید تا حداکثر تعداد خطای ممکن را قبل از آنلاین شدن برنامه‌ی تحت وب حذف کنید.

مراحل کار کدام است؟ فرایند آزمون برنامه‌ی تحت وب با مورد توجه قراردادن جنبه‌هایی از برنامه‌ی تحت وب آغاز می‌شود که پیش چشم کاربر قرار دارند و در ادامه، نوبت به آزمون‌هایی می‌رسد که فن‌آوری و زیرساخت را تمرین می‌دهند. هفت مرحله در آزمون اجرا می‌شود: آزمون محتوا، آزمون واسط، آزمون گشت‌وگذار، آزمون مولفه‌ها، آزمون کارایی و آزمون امنیت.

محصول کاری چیست؟ در برخی موارد، یک برنامه‌ریزی آزمون برنامه‌ی تحت وب ایجاد می‌شود. در هر حال، مجموعه‌ای از موارد آزمون برای هر مرحله از آزمون تهیه می‌شود و آرشیوی از نتایج آزمون برای استفادی آتی حفظ می‌شود.

چگونه اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ گرچه هرگز نمی‌توان اطمینان حاصل کرد که همه‌ی آزمون‌های لازم انجام شده‌اند، می‌توان مطمئن بود که آزمون، خطاها را آشکار کرده است (و این خطاها تصحیح شده‌اند). به‌علاوه، اگر یک برنامه‌ریزی آزمون تهیه کرده اید، می‌توانید تحقیق کنید و مطمئن شوید که آیا همه‌ی آزمون‌های برنامه‌ریزی شده اجرا شده‌اند یا خیر.

اضطراری وجود دارد که همواره پروژه‌های مربوط به برنامه‌های تحت وب را در برمی‌گیرد. ذی‌نفع‌ها، -نگران از رقابت با برنامه‌های تحت وب دیگر، تحت فشار تقاضاهای مشتریان و نگران از اینکه زمان مناسب را در بازار از دست بدهند- فشار وارد می‌آورند تا برنامه‌ی تحت وب را هر چه زودتر آنلاین کنند. در نتیجه، به فعالیت‌های فنی که غالباً در اواخر فرایند رخ می‌دهند، نظیر آزمون برنامه‌ی تحت وب، گاهی فرصت کوتاهی داده می‌شود. این عمل می‌تواند اشتباهی فاجعه‌بار باشد و برای پرهیز از آن، شما و سایر اعضای تیم باید اطمینان حاصل کنید که هر محصول کاری، کیفیت بالایی از خود نشان می‌دهد. والاس و همکاران [Wal03] در این خصوص چنین می‌نویسند:

آزمون نباید منتظر بماند تا پروژه به پایان برسد. آزمون را قبل از نوشتن حتی یک خط از کد شروع کنید. اگر به‌طور پیوسته و اثربخش، آزمایش کنید، وب‌سایتی بسیار پردوام‌تر خواهید ساخت.

از آنجا که مدل‌های خواسته‌ها و طراحی را از نظر کلاسیک نمی‌توان آزمایش کرد، شما و تیم شما باید مرورهای فنی (فصل ۱۵) و نیز آزمون‌های اجرایی را انجام دهید. هدف، کشف و تصحیح خطاهاست، پیش از آن‌که برنامه‌ی تحت وب در دسترس کاربران نهایی‌اش قرار گیرد.

۲۰-۱ مفاهیم آزمون برای برنامه‌های تحت وب

آزمون، فرایند تمرین دادن نرم‌افزار با هدفِ یافتن (و سرانجام تصحیح) خطاهاست. این فلسفه‌ی بنیادی، که نخستین بار در فصل ۱۷ ارائه شد، برای برنامه‌های تحت وب نیز برقرار است. در واقع، از آنجا که برنامه‌ها و سیستم‌های تحت وب روی شبکه قرار دارند و با انواع متفاوت سیستم‌های عامل، مرورگرها (روی دستگاه‌های متنوع)، سکوهای سخت افزاری، پروتکل‌های ارتباطاتی در حال همکاری و تعامل هستند، جستجو به‌دنبال خطاها، چالشی چشمگیر خواهد بود.

برای درک اهداف آزمون در حیطه‌ی مهندسی وب، باید ابعاد بسیاری از کیفیت برنامه‌ی تحت وب را در نظر بگیرند[1]. در حیطه بحث حاضر، به آن دسته از ابعاد کیفیتی خواهیم پرداخت که به ویژه به بحث آزمون برنامه‌های تحت وب مربوط می‌شوند. همچنین به ماهیت خطاهایی که به‌عنوان نتیجه‌ی آزمون مشاهده می‌شوند و نیز به راهبرد آزمون به کاررفته در کشف این خطاها خواهیم پرداخت.

۲۰-۱-۱ ابعاد کیفیتی

کیفیت در یک برنامه‌ی تحت وب، نتیجه‌ی طراحی خوب است. تعیین آن با به‌کارگیری یک سری مرورهای فنی انجام می‌شود که عناصر گوناگون مدل طراحی را با به‌کارگیری یک فرایند آزمون مورد ارزیابی قرار می‌دهد. موضوع این فصل، همین فرایند آزمون است. هم مرورها و هم آزمون‌ها، یک یا چند بُعد از ابعاد کیفیتی زیر را بررسی می‌کنند [Mil00a]:

- محتوا در هر دو سطح نحوی و معنایی ارزیابی می‌شود. در سطح نحوی، املای واژه‌ها، علامت گذاری‌ها و گرامر برای مستندات متنی مورد ارزیابی قرار می‌گیرد. در سطح معنایی، درستی (اطلاعات ارائه شده)، سازگاری (در میان شیء محتوایی و اشیای مرتبط) و فقدان ابهام، همگی بررسی می‌شوند.

در حیطه‌ی یک برنامه‌ی تحت وب و محیط آن، کیفیت را چگونه ارزیابی می‌کنیم؟

[1] ابعاد کلی کیفیت نرم‌افزار، که برای برنامه‌های تحت وب نیز مصداق دارند، در فصل ۱۴ بحث شدند.

- قابلیت عملیاتی مورد آزمایش قرار می‌گیرد تا خطاهایی که عدم مطابقت با خواسته‌های مشتری را نشان می‌دهند، برملا شوند. هر قابلیت عملیاتی برنامه‌ی تحت وب، از نظر درستی، ناپایداری و مطابقت کلی با استانداردهای پیاده‌سازی مناسب (مثلاً استانداردهای زبان‌های جاوا یا AJAX) ارزیابی می‌شود.
- ساختار ارزیابی می‌شود تا اطمینان حاصل شود که محتوا و قابلیت‌های عملیاتی برنامه‌ی تحت وب را به‌طور مناسب تحویل می‌دهد؛ یعنی قابل بسط باشد و در صورت افزوده شدن محتوا یا قابلیت‌های جدید، بتوان آن را پشتیبانی کرد.
- قابلیت استفاده آزمایش می‌شود تا اطمینان حاصل شود که هر گروه از کاربران به وسیله‌ی واسط، پشتیبانی می‌شود و می‌توانند همه‌ی جنبه‌های نحوی و معناشناختی لازم برای گشت‌وگذار را بیاموزند و به کارگیرند.
- قابلیت گشت‌وگذار مورد آزمایش قرار می‌گیرد تا اطمینان حاصل شود که همه‌ی جنبه‌های نحوی و معناشناختی تمرین داده شده‌اند و همه‌ی خطاهای گشت‌وگذاری (مثلاً پیوندهای منتهی به بن بست، پیوندهای نامناسب و پیوندهای خطادار) برملا شوند.
- کارایی تحت انواع شرایط عملیاتی، پیکربندی‌ها و بارهای مختلف آزمایش می‌شود تا اطمینان حاصل شود که سیستم، پاسخ‌گوی تعامل‌های کاربر هست و حداکثر ازدحام بار را بدون تنزل غیرقابل قبول در عملکرد فراهم می‌سازد.
- سازگاری با اجرای برنامه‌ی تحت وب در انواع پیکربندی‌های میزبان متفاوت در هر دو طرف یعنی کلاینت و سرور اجرا می‌شود. هدف از این آزمون، یافتن خطاهایی است که خاص یک پیکربندی میزبان منحصر به فرد هستند.
- عملکرد متقابل آزمایش می‌شود تا اطمینان حاصل شود که برنامه‌ی تحت وب به‌طور مناسب با سایر برنامه‌های کاربردی و/یا بانک‌های اطلاعاتی ارتباط دارد.
- امنیت با ارزیابی آسیب‌پذیری‌ها و تلاش برای کشف این آسیب‌پذیری‌ها مورد آزمایش قرار می‌گیرد. هرگونه تلاش موفق برای نفوذ به سیستم، به‌عنوان یک شکست امنیتی در نظر گرفته می‌شود.

راهبرد و تاکتیک‌هایی برای آزمون برنامه‌های تحت وب تدارک دیده شده‌اند که هر کدام از این ابعاد کیفیتی را تمرین دهند؛ آنها را در همین فصل مورد بحث قرار خواهیم داد.

۲۰-۱-۲ خطاهای موجود در محیط یک برنامه‌ی تحت وب

خطاهایی که در نتیجه‌ی آزمایش موفق برنامه‌ی تحت وب آشکار می‌شوند، چند خصوصیت منحصر به فرد دارند [Ngu00]:

۱. از آنجا که انواع بسیاری از آزمون‌های برنامه‌ی تحت وب، مشکلاتی را آشکار می‌کنند که نخستین بار به چشم کلاینت می‌آیند (یعنی از طریق یک واسط پیاده‌سازی شده روی مرورگر خاص یا یک دستگاه ارتباطی شخصی)، غالباً نشانه‌ای از خطا را می‌بینید، نه خود خطا را.

۲. چون برنامه‌ی تحت وب در چند پیکربندی متفاوت و در محیط‌های متفاوت پیاده‌سازی می‌شود، ممکن است بازسازی یک خطا در خارج از محیطی که خطا در ابتدا در آن مشاهده شده است، دشوار یا غیرممکن باشد.

> فن‌آوری برای آزمون‌گران نرم‌افزار یک شیرینی تلخ به‌شمار می‌رود. درست هنگامی که به‌نظر می‌رسد می‌دانیم چگونه یک فن‌آوری خاص را آزمایش کنیم، یک [برنامه‌ی تحت وب] جدید از راه می‌رسد و همه‌ی نقشه‌ها نقش بر آب می‌شود.
>
> جیمز بک

چه چیز باعث تفاوت میان خطاهای مشاهده‌شده در اجرای برنامه‌ی تحت وب و خطاهای مشاهده‌شده برای نرم‌افزارهای سنتی می‌شود؟

۳. گرچه برخی خطاها نتیجه‌ی طراحی نادرست یا کدنویسی HTM (یا هر زبان دیگر) به شیوه‌ای مناسب هستند، بسیاری از خطاها را می‌توان تا پیکربندی برنامه‌ی تحت وب دنبال کرد.
۴. از آنجا که برنامه‌های تحت وب در زمره معماری‌های کلاینت/سرور دسته‌بندی می‌شوند، دنبال‌کردن خطا در سه لایه‌ی معماری یعنی کلاینت، سرور یا خود شبکه می‌تواند دشوار باشد.
۵. برخی خطاها از محیط عملیاتی ایستا (یعنی پیکربندی خاصی که در آن آزمون اجرا می‌شود) ناشی می‌شوند در حالی که خطاهای دیگر را می‌توان به محیط عملیاتی پویا (یعنی دانلود منابع لحظه‌ای یا خطاهای مرتبط با زمان) نسبت داد.

این پنج صفت که برای خطا برشمرده شدند، حکایت از آن دارند که محیط، نقش مهمی در تشخیص همه‌ی خطاهای کشف نشده طی آزمون برنامه‌ی تحت وب دارد. در برخی شرایط، (مثلاً آزمایش محتوا)، جایگاه خطا پیداست، ولی در بسیاری از انواع آزمون‌های برنامه‌ی تحت وب (از قبیل آزمون گشت‌وگذار، آزمون کارایی، آزمون امنیت) تعیین دلیل بنیادی خطا ممکن است به‌طور چشمگیری دشوارتر باشد.

۳-۱-۲۰ راهبرد آزمون

در راهبرد مربوط به آزمون برنامه‌های تحت وب، همان اصول پایه‌ای مربوط به آزمون همه‌ی نرم‌افزارها (فصل ۱۷) مد نظر قرار می‌گیرد و راهبرد و تاکتیک‌های توصیه شده برای سیستم‌های شیءگرا (فصل ۱۹) نیز به کار گرفته می‌شود. در مراحلی که به دنبال خواهد آمد، این رویکرد به‌طور خلاصه بیان می‌شود:

۱. مدل محتوای برنامه‌ی تحت وب مرور می‌شود تا خطاها برملا گردد.
۲. مدل واسط مرور می‌شود تا اطمینان حاصل آید که همه‌ی پرونده‌های کاربرد قابل پاسخ‌گویی باشند.
۳. مدل طراحی برنامه‌ی تحت وب مرور می‌شود تا خطاهای گشت‌وگذار آشکار شوند.
۴. واسط کاربر آزمایش می‌شود تا خطاهای موجود در ارائه و/یا گشت گذار برملا شود.
۵. مولفه‌های عملیاتی مورد آزمون واحد قرار می‌گیرند.
۶. گشت‌وگذار در سرتاسر معماری، آزمایش می‌شود.
۷. برنامه‌ی تحت وب در انواع پیکربندی‌های محیطی متفاوت پیاده‌سازی می‌شود و از نظر سازگاری با هر پیکربندی، آزمایش می‌شود.
۸. آزمون‌های امنیتی اجرا می‌شوند تا آسیب پذیری‌های موجود در برنامه‌ی تحت وب یا موجود در محیط آن برملا گردد.
۹. آزمون‌های کارایی اجرا می‌شوند.
۱۰. برنامه‌ی تحت وب توسط تعدادی از کاربران نهایی آزمایش می‌شود که تحت کنترل و پایش هستند؛ نتایج تعامل آنها با سیستم از نظر خطاهای محتوا و گشت‌وگذار، دغدغه‌های مربوط به قابلیت استفاده، دغدغه‌های مربوط به سازگاری و همچنین امنیت، قابلیت اطمینان و کارایی برنامه‌ی تحت وب مورد ارزیابی قرار می‌گیرد.

از آنجا که بسیاری از برنامه‌های تحت وب به‌طور پیوسته تکامل می‌یابند، فرایند آزمون یک فعالیت در حال پیشرفت است که توسط کارمندان پشتیبانی برنامه‌ی تحت وب انجام می‌شوند؛ این کارمندان،

نکته‌ی کلیدی
راهبرد کلی برای آزمودن برنامه‌های تحت وب را می‌توان در ده مرحله‌ی مقابل خلاصه کرد.

مرجع وب
مقالاتی عالی درباره آزمون برنامه‌های تحت وب را در وب‌سایت زیر می‌توانید بیابید:
www.stickyminds.com/testing.asp

از آزمون‌های رگرسیونی استفاده می‌کنند که از آزمون‌های تهیه شده در هنگام اولین دور مهندسی شدن برنامه‌ی تحت وب به‌دست آمده‌اند.

۴-۱-۲۰ برنامه‌ریزی آزمون‌ها

استفاده از واژه‌ی *برنامه‌ریزی* (در هر حیطه ای) منفور برخی سازندگان برنامه‌های تحت وب است. این سازندگان، برنامه‌ریزی نمی‌کنند؛ آنها فقط شروع می‌کنند به این امید که یک شاهکار ایجاد کنند. در یک رویکرد منضبط‌تر، پذیرفته می‌شود که برنامه‌ریزی، نقشه‌ی راهنمایی برای همه‌ی کارهای بعدی فراهم می‌سازد، این تلاش، ارزش‌مند است. اسپلین و جاسکیل [Spl01] در کتاب خود درباب آزمایش برنامه‌های تحت وب چنین می‌گویند:

جز برای ساده‌ترین وب‌سایت‌ها، بلافاصله می‌توان دریافت که نوعی برنامه‌ریزی برای آزمون ضروری است. بسیار پیش می‌آید که تعداد خطاهای اولیه‌ی یافته شده در آزمونی که از قبل برنامه‌ریزی نشده باشد، به قدر کافی بزرگ هست که همه‌ی آنها در اولین مرتبه‌ی کشف، برطرف نشوند. این خود باری اضافی بر گردن افرادی می‌گذارد که وب‌سایت‌ها و برنامه‌های کاربردی را آزمایش می‌کنند. آنها نه تنها باید به آزمون‌های جدید متمسک شوند، بلکه باید این را هم به خاطر داشته باشند که چگونه آزمون‌های قبلی اجرا شدند تا وب‌سایت/برنامه با اطمینان مورد آزمایش دوباره قرار گیرند و اطمینان حاصل شود که خطاهای شناخته شده حذف شده‌اند و هیچ خطای جدیدی وارد نشده است.

پرسش‌هایی که باید بپرسید، عبارتند از: «چگونه به آزمون‌های جدید دست پیدا کنیم.» و در این آزمون‌ها چه چیز باید کانون توجه قرار گیرد؟ پاسخ این پرسش‌ها در برنامه‌ریزی آزمون نهفته است.

در برنامه‌ریزی آزمون برنامه‌ی تحت وب، مواردی که به دنبال خواهد آمد، مشخص می‌شود: (۱) مجموعه وظایفی[1] که باید با شروع آزمون به انجام برسند، (۲) محصولات کاری که باید با اجرای هر وظیفه تولید شوند و (۳) شیوه ارزیابی، ثبت و استفاده‌ی مجدد از نتایج، هنگامی که آزمون‌های رگرسیون اجرا می‌شود. در برخی موارد، برنامه‌ریزی آزمون با طرح پروژه منسجم می‌شود. در سایر موارد، برنامه‌ریزی آزمون خود مستندی جداگانه است.

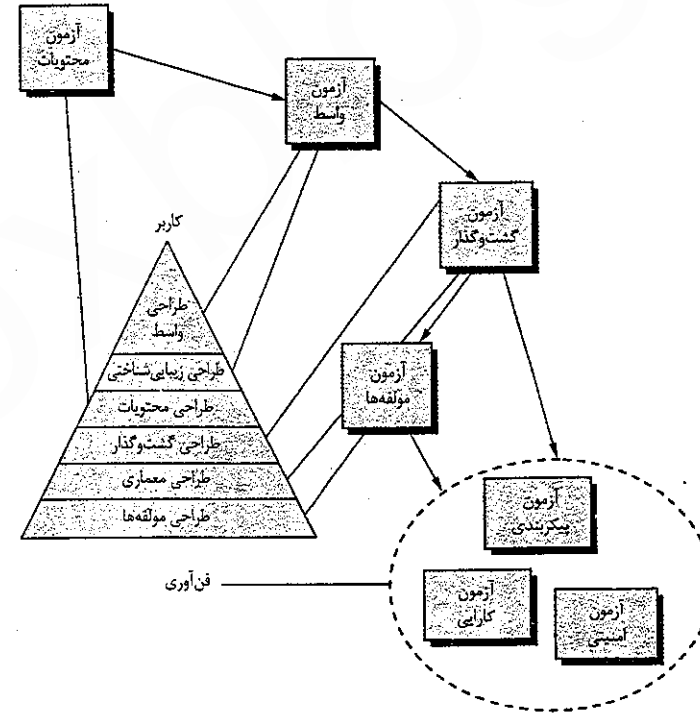
نکته‌ی کلیدی
در طرح آزمون، مجموعه وظایف آزمون، محصولات کاری‌ای که باید توسعه داده شوند و روش ارزیابی ثبت و استفاده‌ی مجدد از نتایج تعیین می‌شود.

۲-۲۰ فرایند آزمون – نگاهی اجمالی

فرایند آزمون برنامه‌ی تحت وب با آزمون‌هایی آغاز می‌شود که محتوا و قابلیت‌های عملیاتی واسط را که فوراً به چشم کاربران نهایی می‌آیند، تمرین می‌دهند. با ادامه‌ی آزمون، جنبه‌هایی از معماری طراحی و گشت‌وگذار تمرین داده می‌شوند. سرانجام، کانون توجه به سمت آزمون‌هایی جابجا می‌شود که قابلیت‌های فنی‌ای را بررسی می‌کنند که همیشه پیش چشم کاربران نهایی پدیدار نیستند (مسائل زیرساختی و نصب و پیاده‌سازی برنامه‌ی تحت وب).

[1] این مجموعه وظایف در فصل ۲ بحث شدند. یک اصطلاح مرتبط –*جریان کاری*– نیز در توصیف مجموعه وظایف لازم برای انجام یک فعالیت مهندسی نرم‌افزار به کار می‌رود.

در شکل ۱-۲۰ فرایند آزمون برنامه‌های تحت وب با هرم طراحی برنامه‌های تحت وب (فصل ۱۳) مطابقت داده شده است. توجه دارید که جریان آزمون از چپ به راست و از بالا به پایین پیش می‌رود، یعنی ابتدا عناصری از طراحی برنامه‌ی تحت وب که به چشم کاربر می‌آیند (عناصر بالای هرم) ابتدا آزمایش می‌شوند و سپس نوبت به عناصر طراحی زیرساختی می‌رسد.



شکل ۱-۲۰ فرایند آزمون.

۲۰-۳ آزمون محتوا (Content Testing)

خطاهای موجود در محتوا می‌توانند بسیار بی اهمیت و در حد خطاهای تایپی یا بسیار مهم در حد اطلاعات نادرست، سازمان‌دهی نامناسب یا عدول از قوانین مالکیت فکری باشند. در *آزمون محتوا* تلاش می‌شود که این مشکلات و بسیاری مشکلات دیگر، قبل از مواجهه‌ی کاربر با آنها کشف شوند.

در آزمون محتوا، هر دو بخش مرور و تولید موارد آزمون قابل اجرا با هم ترکیب می‌شوند. مرورهایی برای کشف خطاهای موجود در محتوا، اعمال می‌شوند (که در بخش ۳۱-۲۰ بحث خواهد شد). آزمون قابل اجرا در کشف خطاهای محتوا را تا محتوایی که به صورت پویا به‌دست می‌آیند و با داده‌های به‌دست آمده از یک یا چند بانک اطلاعاتی رانده می‌شوند، می‌توان دنبال کرد.

اندرز

مرورهای فنی بخشی از آزمون به شمار نمی‌روند، ولی مرور محتوا باید انجام گردد تا اطمینان حاصل شود که محتوا، دارای کیفیت هستند.

۲۰-۳-۱ اهداف آزمون محتوا

در آزمون محتوا سه هدف مهم دنبال می‌شود: (۱) کشف خطاهای نحوی (مثلاً اشتباهات تایپی و گرامری) در مستندات متنی، نمایش‌های گرافیکی و سایر رسانه‌ها، (۲) کشف خطاهای معنایی (یعنی خطای صحت یا کامل بودن اطلاعات) در همه‌ی اشیای محتوایی ارائه شده هنگامی که گشت‌وگذار رخ می‌دهد و (۳) یافتن خطاهای موجود در سازمان‌دهی یا ساختار محتوایی که به کاربر نهایی ارائه می‌شود.

نکته‌ی کلیدی

اهداف آزمون محتوا عبارتند از: (۱) کشف خطاهای نحوی در محتوا، (۲) کشف خطاهای معناشناختی و (۳) یافتن خطاهای ساختاری.

برای دستیابی به هدف نخست، می‌توان از برنامه‌های چک کننده‌ی املا و گرامر استفاده کرد. ولی، بسیاری خطاهای نحوی ممکن است از دید چنین ابزارهایی پنهان بمانند و باید توسط آزمون گران انسانی کشف شوند. در واقع، یک وب‌سایت بزرگ ممکن است یک ویراستار حرفه‌ای را به خدمت بگیرد تا خطاهای تایپی و گرامری، خطاهای موجود در سازگاری محتوا، خطاهای موجود در نمایش‌های گرافیکی و نیز خطاهای مربوط به ارجاع‌های متقابل را کشف کند.

در آزمون معنایی، آنچه که مورد توجه قرار می‌گیرد، اطلاعات ارائه شده در هر شیء محتوایی است. آزمون‌گر (مسؤول مرور) باید به این پرسش‌ها پاسخ گوید:

- آیا اطلاعات واقعیت دارند؟
- آیا اطلاعات، فشرده و موجز هستند؟
- آیا درک شیء محتوایی برای کاربر آسان است؟
- آیا اطلاعات ادغام شده در داخل یک شیء محتوایی را به راحتی می‌توان یافت؟
- آیا برای تمامی اطلاعات به‌دست آمده از منابع دیگر، ارجاع‌های مناسب تدارک دیده شده است؟
- آیا اطلاعات ارائه شده از نظر درونی سازگارند و آیا با اطلاعات ارائه شده توسط اشیای محتوایی دیگر سازگاری دارند؟
- آیا محتوا، اهانت‌بار یا گمراه کننده نیستند یا منع قانونی ندارند؟
- آیا محتوا از قوانین رعایت حقوق مولفان و مصنفان عدول نمی‌کند؟
- آیا در محتوا، پیوندهای درونی وجود دارند که محتوای موجود را تکمیل کنند؟ آیا این پیوندها صحیح‌اند؟
- آیا سبک زیبایی‌شناختی محتوا با سبک زیباشناختی واسط در تضاد نیست؟

برای کشف خطاهای معناشناختی در محتوا، چه پرسش‌هایی باید پرسیده و پاسخ داده شوند؟

به‌دست آوردن پاسخ هر کدام از این پرسش‌ها برای یک برنامه‌ی تحت وب بزرگ (که حاوی صدها شیء محتوایی است) می‌تواند کاری وحشتناک باشد. به هر حال، شکست در کشف خطاهای معنایی باعث ایجاد خلل در وفاداری کاربر به برنامه‌ی تحت وب شده می‌تواند کاربرد آن را با شکست مواجه سازد.

اشیای محتوایی در داخل معماری‌ای جای دارند که سبکی خاص دارد (فصل ۱۳). طی آزمون محتوا، ساختار و سازمان‌دهی معماری محتوا، آزمایش می‌شود تا اطمینان حاصل شود که محتوای مورد نیاز با ترتیب و روابط مناسب به کاربر نهایی ارائه می‌شود. برای مثال، برنامه‌ی تحت وب **SafeoHomeAssured.com** انواع اطلاعات مرتبط با حس گرهای به کاررفته به‌عنوان بخشی از محصولات پایش و امنیت را ارائه می‌دهد. اشیای محتوایی، اطلاعات فنی، مشخصات فنی، نمایش

به‌طور کلی، تکنیک‌های آزمون نرم‌افزار که در سایر برنامه‌های کاربردی اعمال می‌شوند همان تکنیک‌هایی هستند که برای برنامه‌های کاربردی مبتنی بر وب اعمال می‌شوند ... اختلاف در اینجاست که متغیرهای فن‌آوری در محیط وب چند برابر می‌شوند.

تصاویر و اطلاعات مرتبط را فراهم می‌آورند. در آزمون‌های مربوط به معماری محتوای **SafeHomeAssured.com** تلاش می‌شود که خطاهای موجود در ارائه‌ی این اطلاعات (مثلاً توصیف حس‌گر X با عکسی از حس‌گر Y ارائه می‌شود) برملا شود.

۲-۳-۲۰ آزمون بانک اطلاعاتی

کار برنامه‌های تحت وب مدرن به مراتب بیشتر از ارائه‌ی اشیای محتوایی ایستاست. برنامه‌های تحت وب در بسیاری از دامنه‌های کاربردی با سیستم‌های مدیریت بانک‌های اطلاعاتی پیچیده‌ای رابطه برقرار می‌کنند و اشیای محتوایی پویایی می‌سازند که به صورت زمان حقیقی و با استفاده از داده‌های به‌دست آمده از یک بانک اطلاعاتی ایجاد می‌شوند.

برای مثال، یک برنامه‌ی تحت وب سرویس‌های مالی می‌تواند اطلاعات متنی، جدول‌بندی شده و گرافیکی درباره‌ی یک موضوع مالی (مثل سهام بورس) ایجاد کند. شیء محتوایی مرکبی که این اطلاعات را ارائه می‌دهد به شیوه‌ای پویا پس از درخواست اطلاعات درباره‌ی یک موضوع مالی خاص ایجاد می‌شود. برای این منظور، مراحل زیر ضروری است: (۱) به یک بانک اطلاعاتی بزرگ حاوی موضوعات مالی نیاز است، (۲) داده‌های مربوط از این بانک اطلاعاتی استخراج می‌شوند، (۳) داده‌های استخراج شده باید در قالب یک شیء محتوایی سازمان‌دهی شوند و (۴) این شیء محتوایی (که ارائه‌گر اطلاعات درخواست شده توسط کاربر نهایی است) به محیط کلاینت انتقال داده می‌شود تا به نمایش درآید. در نتیجه‌ی انجام هر کدام از مراحل، ممکن است خطاهایی رخ دهد. هدف آزمون بانک اطلاعاتی، کشف این خطاهاست، ولی آزمون بانک اطلاعاتی را عوامل مختلف پیچیده می‌کند:

۱. درخواست اولیه‌ی کلاینت-سرور برای اطلاعات به ندرت به شکلی ارائه می‌شود [مثلاً زبان پرس‌وجوی ساخت‌یافته (SQL)] که بتواند ورودی یک سیستم مدیریت بانک اطلاعاتی (DBMS) باشد. بنابراین، آزمون‌هایی را باید طراحی کرد که خطاهای موجود در تبدیل درخواست کاربر به شکلی قابل پردازش برای DBMS آشکار کنند.

۲. بانک اطلاعاتی ممکن است از سروری که برنامه‌ی تحت وب را در خود جای داده است، دور باشد. بنابراین، آزمون‌هایی باید تدارک دیده شود که خطاهای موجود در برقراری ارتباط میان‌برنامه‌ی تحت وب و بانک اطلاعاتی دوردست را برملا سازند[1].

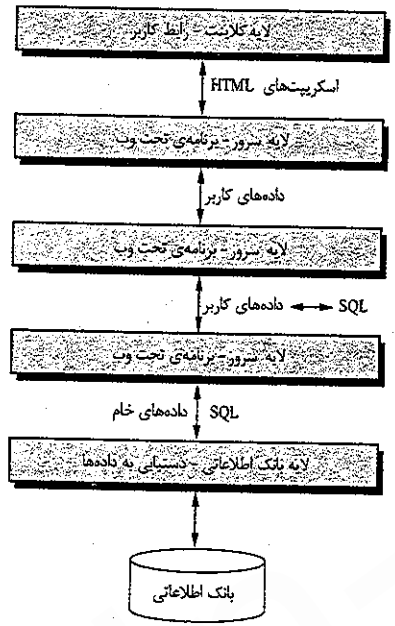
۳. داده‌های خام به‌دست آمده از بانک اطلاعاتی باید به سرور برنامه‌ی تحت وب ارسال شوند و برای انتقال بعدی به کلاینت به فرمت مناسب درآیند. بنابراین، آزمون‌هایی باید فراهم آورده شوند که اعتبار داده‌های خام دریافت شده توسط سرور برنامه‌ی تحت وب را به نمایش بگذارند و علاوه بر آن، آزمون‌هایی نیز باید ایجاد شود که اعتبار تبدیل‌های به کارگرفته شده برای داده‌های خام و ایجاد اشیای محتوایی معتبر را نشان دهند.

۴. اشیای محتوایی پویا باید به شکلی به کلاینت انتقال داده شوند که به کاربر نهایی نشان داده شوند. بنابراین، یک سری آزمون باید طراحی شود تا (۱) خطاهای موجود در قالب شیء محتوایی را برملا کنند و (۲) سازگاری با پیکربندی‌های متفاوت محیط کلاینت را آزمایش کنند.

چه مسائلی آزمون بانک اطلاعاتی برای برنامه‌های تحت وب را پیچیده می‌کند؟

[1] هنگامی که بانک‌های اطلاعاتی توزیع شده وجود داشته باشند یا هنگامی که دستیابی به یک انبار داده‌ای (فصل ۱) مورد نیاز باشد، این آزمون‌ها می‌توانند بسیار پیچیده شوند.

با در نظر گرفتن این چهار عامل، روش‌های طراحی موارد آزمون باید برای هر کدام از «لایه‌های تعامل» [Ngu01] که در شکل ۲-۲۰ ذکر شده اند، به کارگرفته شوند. آزمون باید این اطمینان را ایجاد کند که (۱) اطلاعات معتبر بین کلاینت و سرور از لایه‌ی واسط عبور می‌کند، (۲) برنامه‌ی تحت وب، اسکریپت‌ها را به درستی پردازش می‌کند و به‌طور مناسب، داده‌های کاربر را استخراج یا فرمت‌بندی می‌کند، (۳) داده‌های کاربر به‌طور صحیح به توابع تبدیل داده‌ای طرف سرور تحویل می‌شوند که درخواست‌های مناسب را فرمت‌بندی می‌کنند (مثل SQL)، (۴) درخواست‌ها به یک لایه‌ی مدیریت داده‌ها[1] عبور داده می‌شوند که با روال‌های دستیابی به بانک اطلاعاتی ارتباط برقرار می‌کنند (این روال‌ها به‌طور بالقوه روی ماشین دیگر قرار دارند).



شکل ۲-۲۰ لایه‌های تعامل

تبدیل داده‌ها، مدیریت داده‌ها و لایه‌های دستیابی به بانک اطلاعاتی که در شکل ۲-۲۰ نشان داده شده اند، غالباً با مولفه‌هایی با قابلیت استفاده‌ی مجدد اجرا می‌شوند که به‌طور جداگانه و در قالب یک پکیج اعتبارسنجی شده‌اند. اگر چنین باشد، آزمون برنامه‌ی تحت وب، طراحی موارد آزمون را کانون توجه قرار می‌دهد تا تعامل‌های میان لایه‌ی کلاینت و دو لایه‌ی نخست سرور (تبدیل داده‌ها و برنامه‌ی تحت وب) را که در شکل نشان داده شده اند، تمرین دهد.

لایه‌ی واسط کاربر آزمایش می‌شود تا اطمینان حاصل شود که اسکریپت‌ها برای هر درخواست کاربر از ساختاری مناسب برخوردارند و به شیوه‌ای مناسب به سرور انتقال داده می‌شوند. لایه‌ی

[1] لایه مدیریت داده‌ها معمولاً شامل یک رابط SQL در سطح فراخوانی (SQL-CLI) مثل Microsoft OLE/ADO یا Java Database Connectivity (JDBC) می‌شود.

«... احتمال اعتماد به وب‌سایتی که هر دم قطع است، در میانه‌ی یک تراکنش قفل می‌کند یا از نظر قابلیت استفاده ضعیف عمل می‌کند، زیاد نیست. بنابراین، آزمون، نقشی حیاتی در کل فرایند توسعه دارد.»

وینگ لم

برنامه‌ی تحت وب در طرف سرور، آزمایش می‌شود تا اطمینان حاصل شود که داده‌های کاربر به‌طور مناسب از اسکریپت‌ها استخراج شده و به‌طور مناسب به لایه تبدیل داده‌ها روی سرور انتقال داده شده‌اند. توابع تبدیل داده‌ها آزمایش می‌شوند تا این اطمینان ایجاد شود که SQL صحیح ایجاد و به مولفه‌های مدیریت داده‌ای مناسب تحویل شده است.

بحث مشروح درباره فن‌آوری زیرساختی که اطلاع از آن برای طراحی مناسب این آزمون‌های بانک اطلاعاتی ضروری است، از حوصله‌ی این کتاب خارج است. در صورت علاقه‌ی بیشتر [Sce02]، [Ngu01] و [Bro01] را ببینید.

۲۰-۴ آزمون واسط کاربر

وارسی و اعتبارسنجی واسط کاربر برنامه‌ی تحت وب، در سه نقطه‌ی متمایز رخ می‌دهد. طی مرحله‌ی تحلیل خواسته‌ها، مدل واسط مرور می‌شود تا اطمینان حاصل شود که با خواسته‌های ذی‌نفع‌ها و سایر عناصر مدل خواسته‌ها همخوانی دارد. طی طراحی، مدل طراحی واسط مرور می‌شود تا اطمینان حاصل شود که ملاک‌های کیفیتی کلی وضع شده برای همه‌ی واسط‌های کاربر (فصل ۱۱) برقرار است و به مسائل طراحی واسط خاص کاربرد به‌طور مناسب پرداخته شده است. طی آزمون، کانون توجه به اجرای جنبه‌های خاص کاربرد تعامل کاربر جابجا می‌شود زیرا توسط قالب نحوی واسط و معناشناسی واسط اعلام می‌شود. به علاوه، آزمون قابلیت استفاده را مورد ارزیابی نهایی قرار می‌دهد.

۲۰-۴-۱ راهبرد آزمون واسط

آزمون واسط، سازوکارهای تعامل را تمرین می‌دهد و جنبه‌های زیبایی‌شناختی واسط کاربر را اعتبارسنجی می‌کند. راهبرد کلی برای آزمون واسط عبارت است از (۱) کشف خطاهای مرتبط با سازوکارهای خاص واسط (مثل خطاهای موجود در اجرای متعارف یک پیوند منو یا شیوه واردکردن داده‌ها در یک فرم) و (۲) کشف خطاهای موجود در روش پیاده‌سازی معناشناسی گشت‌وگذار، قابلیت‌های عملیاتی برنامه‌ی تحت وب یا نمایش محتوا توسط واسط. برای دستیابی به این راهبرد، چند مرحله‌ی تاکتیکی آغاز می‌شود:

- *ویژگی‌های واسط، آزمایش می‌شوند تا اطمینان حاصل شود که قواعد طراحی، زیبایی‌شناسی و محتوای بصری مرتبط، بدون خطا برای کاربر در دسترس هستند.* این ویژگی‌ها عبارتند از نوع فونت‌ها، کاربرد رنگ‌ها، قاب‌ها، تصاویر، مرزها، جداول و ویژگی‌های مرتبط با واسط که با ادامه‌ی اجرای برنامه‌ی تحت وب ایجاد می‌شوند.
- *سازوکارهای واسط فردی به گونه‌ای آزمایش می‌شوند که مشابه با آزمون واحدهاست.* برای مثال، آزمون‌هایی برای تمرین دادن همه‌ی فرم‌ها، اسکریپت نویسی از سوی کلاینت، HTML پویا، اسکریپت‌ها، محتوای جریان‌دار (streaming) و سازوکارهای *واسط* خاص برنامه‌ی کاربردی (همانند کارت خرید برای یک برنامه کاربردی تجارت الکترونیک) طراحی می‌شود. در بسیاری موارد، آزمون می‌تواند انحصاراً یکی از این سازوکارهای «واحد» را کانون توجه قرار دهد و سایر ویژگی‌ها و قابلیت‌های عملیاتی واسط را طرد کند.

اندرز

راهبرد واسط ذکر شده در اینجا به استثنای موارد خاص برنامه‌های تحت وب، در انواع نرم‌افزارهای کلاینت – سرور کاربرد دارد.

- *هر سازوکار واسط در حیطه‌ی یک* use case *یا NSU (فصل ۱۳) برای گروه خاصی از کاربران آزمایش می‌شود.* این رویکرد آزمون، مشابه با آزمون انسجام است، از این لحاظ که آزمون‌ها به موازات انسجام یافتن سازوکارهای واسط انجام می‌شوند تا اجرای یک use case یا NSU میسر گردد.
- *واسط کامل در برابر پرونده‌های کاربرد و NSUهای انتخاب شده آزمایش می‌شوند تا خطاهای موجود در معناشناسی واسط کشف شوند.* این رویکرد آزمون مشابه با آزمون اعتبارسنجی است زیرا هدف آن، نشان دادن همخوانی با معناشناسی use case یا NSU است. در این مرحله است که یک سری آزمون‌های قابلیت استفاده، اجرا می‌شود.
- *واسط در انواع محیط‌ها (مثلاً مرورگرهای متفاوت) آزموده می‌شود تا اطمینان حاصل شود که از سازگاری برخوردار است.* در واقع، این سری آزمون‌ها را می‌توان به‌عنوان بخشی از آزمون پیکربندی نیز در نظر گرفت.

۲۰-۴-۲ آزمون سازوکارهای واسط

هنگامی که کاربری با یک برنامه‌ی تحت وب تعامل می‌کند، این تعامل از طریق یک یا چند سازوکار واسط رخ می‌دهد. در پاراگراف‌هایی که به دنبال خواهد آمد، هر کدام از سازوکارهای واسط به اختصار شرح داده خواهد شد [Spl01]:

پیوندها (Links). هر پیوند گشت‌وگذار، آزمایش می‌شود تا اطمینان حاصل شود که به شیء محتوایی یا قابلیت عملیاتی مناسب منتهی می‌شود[1]. از همه‌ی پیوندهای مرتبط با چیدمان واسط (مانند خطوط منو یا آیتم‌های نمایه) فهرستی تهیه می‌شود و سپس هر کدام اجرا می‌گردد. به علاوه، پیوندهای موجود در هر شیء داده‌ای باید تمرین داده شوند تا URLهای بد با پیوندهای منتهی به اشیای داده‌ای یا قابلیت‌های نامناسب کشف شوند. سرانجام، پیوندهای منتهی به برنامه‌های تحت وب باید آزمایش شوند تا صحت آنها مسجل شود و نیز ارزیابی می‌شوند تا خطر نامعتبرشدن آنها در اثر گذشت زمان تعیین گردد.

فرم‌ها (Forms). در سطح ماکروسکوپی، آزمون‌هایی اجرای می‌شوند تا این اطمینان ایجاد گردد که (۱) برچسب‌ها به‌طور صحیح فیلدهای داخل فرم را مشخص می‌کنند و فیلدهایی که پرکردن آنها اجباری است به‌طور بصری برای کاربر مشخص شده‌اند، (۲) سرور همه‌ی اطلاعات موجود در فرم را دریافت می‌کند و هیچ داده‌ای در اثنای انتقال میان کلاینت و کاربر از دست نمی‌رود، (۳) هنگامی که کاربر چیزی را از منوهای بازشونده یا مجموعه‌ای از دکمه‌ها انتخاب نمی‌کند، از مقادیر پیش فرض مناسب استفاده می‌شود، (۴) قابلیت‌های موجود در مرورگر (مثل پیکان بازگشت «Back») باعث از بین رفتن داده‌های وارد شده در یک فرم نمی‌شود و (۵) اسکریپت‌هایی که داده‌های وارد شده را برای خطا چک می‌کنند، به‌طور مناسب عمل می‌کنند و پیام‌های خطای بامعنی فراهم می‌آورند. در یک سطح هدفمندتر، آزمون‌ها باید این اطمینان را بدهند که (۱) فیلدهای موجود در فرم از طول مناسب برخوردارند و نوع داده‌ها در آنها رعایت شده است، (۲) فرم، حفاظ‌هایی برقرار می‌کند که کاربر را از واردکردن رشته‌های متنی بزرگ‌تر از یک حد معین، منع می‌کنند، (۳) همه‌ی گزینه‌های مناسب برای

اندرز

آزمون پیوندهای خارجی باید در سرتاسر حیات برنامه‌ی تحت وب رخ دهد. بخشی از راهبرد پشتیبانی باید آزمون‌های زمان‌بندی‌شده‌ی منظم روی پیوندها باشد.

[1] این آزمون‌ها را می‌توان به‌عنوان بخشی از آزمون رابط یا آزمون گشت‌وگذار اجرا نمود.

منوهای بازشونده مشخص می‌شوند و به‌گونه‌ای مرتب می‌شوند که برای کاربر نهایی بامعنی باشد، (۴) ویژگی‌های «پرکردن خودکار» به واردکردن داده‌های خطا منجر نمی‌شوند و (۵) یک کلید Tab (یا هر کلید دیگر) حرکت مناسب میان فیلدهای فرم را ممکن می‌سازد.

اسکریپت‌نویسی از سوی کلاینت. برای کشف هر گونه خطا در پردازش به هنگام اجرای اسکریپت، آزمون‌های جعبه سیاه انجام می‌شود. این آزمون‌ها غالباً با آزمون فرم‌ها همراه می‌شوند زیرا ورودی اسکریپت غالباً از داده‌های فراهم آمده به‌عنوان بخشی از پردازش فرم‌ها به‌دست می‌آیند. برای حصول اطمینان از اینکه زبان اسکریپت نویسی انتخاب شده، در پیکربندی‌های محیطی پشتیبان برنامه‌ی تحت وب، به‌طور مناسب عمل می‌کنند، باید آزمون سازگاری به عمل آید. اسپلین و جسکیل [Spl01] علاوه بر آزمایش خود اسکریپت، پیشنهاد می‌کنند که «باید اطمینان حاصل کنید که استانداردهای [برنامه‌ی تحت وب] شرکت شما، زبان مناسب و نسخه‌ی زبان اسکریپت نویسی مورد استفاده در طرف کلاینت (و در طرف سرور) را بیان می‌کنند.»

اندرز

هرگاه که نسخه‌ی جدیدی از یک مرورگر پرطرفدار روانه‌ی بازار شود، آزمون اسکریپت‌نویسی در طرف کلاینت و آزمون‌های مرتبط با HTML پویا باید تکرار شوند.

HTML پویا. هر صفحه‌ی وبی که حاوی HTML پویاست، اجرا می‌شود تا اطمینان حاصل شود که نمایش پویایی آن درست است. به علاوه، آزمون سازگاری نیز باید اجرا شود تا اطمینان حاصل شود که HTML پویا در پیکربندی‌های پشتیبان برنامه‌ی تحت وب درست عمل می‌کنند.

پنجره‌های Pop-up. با یک سری آزمون می‌توان مطمئن شد که (۱) popup از اندازه‌ی مناسب برخوردار است و در جای مناسب قرار داده شده است، (۲) popup، پنجره اصلی برنامه‌ی تحت وب را نمی‌پوشاند، (۳) طراحی popup از نظر زیباشناسی با طراحی واسط همساز است و (۴) نوارهای جابه‌جایی محتوا و سایر سازوکارهای ملحق شده به popup در جای مناسب خود قرار دارند و به همان صورتی که لازم است، عمل می‌کنند.

اسکریپت‌های CGI. آزمون‌های جعبه سیاه با تاکید بر انسجام اسکریپت (با دریافت داده‌های اعتبارسنجی شده) اجرا می‌شوند. به علاوه، آزمون کارایی را نیز می‌توان اجرا کرد تا این اطمینان ایجاد شود که پیکربندی در طرف سرور می‌تواند پاسخ گوی چند تقاضای پردازشی از سوی اسکریپت‌های CGI باشد [Spl01].

محتوای جریان‌دار (Streaming). آزمون‌هایی باید انجام شوند تا نشان دهند که داده‌های جریان دار بهنگام هستند، به طرزی مناسب نمایش داده می‌شوند و بدون خطا می‌توان آنها را معلق کرد و دوباره آغاز کرد.

کوکی‌ها (Cookies). آزمون در هر دو طرف سرور و کلاینت مورد نیاز است. در طرف سرور، باید آزمون‌هایی به عمل آید تا اطمینان حاصل شود که وقتی محتوای یا قابلیت عملیاتی خاصی درخواست می‌شود، کوکی‌ها به‌طور مناسب ایجاد شده‌اند (حاوی داده‌های صحیح هستند) و به‌طور مناسب به‌طرف کلاینت ارسال می‌شود. به علاوه، ماندگاری مناسب کوکی‌ها آزمایش می‌شود تا اطمینان حاصل شود که تاریخ انقضای آن درست است. در طرف کلاینت، آزمون‌ها تعیین می‌کنند که آیا برنامه‌ی تحت وب به‌طور مناسب، کوکی‌های موجود را به یک درخواست خاص (که به سرور ارسال می‌شود) متصل می‌کند یا خیر.

سازوکارهای واسط خاص برنامه‌ی کاربردی. آزمون‌ها، همخوانی با چک‌لیستی از ویژگی‌ها و قابلیت‌های عملیاتی را می‌سنجند که توسط سازوکار واسط تعریف می‌شوند. برای مثال، اسپلین و جسکیل [Spl01] چک لیست زیر را برای قابلیت «سبد خرید» در برنامه‌های تجارت الکترونیکی پیشنهاد می‌کند:

- آزمون مرزی (فصل ۱۸) حداقل و حداکثر تعداد آیتم‌هایی که می‌توان در یک سبد قرار داد.
- آزمون درخواست خروج برای سبد خرید خالی.
- آزمون حذف مناسب یک آیتم از سبد خرید.
- آزمون برای تعیین اینکه آیا خریدی، سبد را از محتوای آن خالی می‌کند یا خیر.
- آزمون برای تعیین ماندگاری محتوای سبد خرید (این باید به‌عنوان بخشی از خواسته‌های مشتری مشخص شود).
- آزمون برای تعیین اینکه آیا برنامه‌ی تحت وب می‌تواند محتوای سبد خرید را در آینده به‌خاطر آورد (با این فرض که هیچ خریدی انجام نشده است).

۲۰-۴-۳ آزمون معناشناختی واسط

هنگامی که هر کدام از سازوکارهای واسط، مورد آزمون واحد قرار گرفت، کانون توجه آزمون، به معناشناسی واسط تغییر می‌کند. آزمون معناشناختی واسط «این را تعیین می‌کند که طراحی تا چه حد، کاربران را در نظر دارد، راهنمایی‌های روشنی ارائه می‌دهد، بازخوردها را تحویل می‌دهد و سازگاری زبان و رویکرد را حفظ می‌کند.» [Ngu00].

با مرور کامل مدل طراحی واسط می‌توان پاسخ‌هایی جزئی به پرسش پاراگراف قبل داد. به هر حال، سناریوی use case (برای هر گروه) باید هنگام پیاده‌سازی برنامه‌ی تحت وب آزمایش شود. در اصل، یک پرونده‌ی کاربرد، ورودی طراحی یک سری آزمون می‌شود. مقصود این سری آزمون، کشف خطاهایی است که کاربر را از دستیابی به هدف مرتبط با use case محروم می‌کند.

همچنان که هر use case آزمایش می‌شود، خوب است چک لیستی تهیه کنید تا اطمینان حاصل کنید که هر کدام از عناصر منو دست کم یک بار تمرین داده شده‌اند و هر پیوند تعبیه‌شده‌ای در داخل یک شیء محتوایی مورد استفاده قرار گرفته است. به علاوه، سری آزمون‌ها باید شامل انتخاب منوهای نامناسب و استفاده‌ی نابجا از پیوندها شود. هدف، این است که تعیین شود آیا برنامه‌ی تحت وب به طرز مناسب با خطاها مواجه می‌شود یا خیر

۲۰-۴-۴ آزمون‌های قابلیت استفاده (Usablity Tests)

آزمون قابلیت استفاده مشابه با آزمایش معناشناختی واسط (بخش ۳-۴-۲۰) است، از این لحاظ که در این آزمون نیز میزان اثربخشی تعامل کاربران با برنامه‌ی تحت وب، ارزیابی می‌شود و میزان راهنمایی برنامه‌ی تحت وب درخصوص کنش‌های کاربر، بازخوردی بامعنی فراهم می‌آورد و تعاملی سازگار را باعث می‌شود. مرورها و آزمون‌های قابلیت استفاده، به‌جای آن که معناشناسی برخی اشیای تعاملی را کانون توجه قرار دهند، به این منظور طراحی می‌شوند که تعیین کنند واسط برنامه‌ی تحت وب تا چه میزان کارها را برای کاربر آسان می‌کند[1].

مرجع وب

دستورالعمل ارزشمندی درباره آزمون قابلیت استفاده را در نشانی زیر می‌توانید بیابید:
www.ahref.com/guide/design/199806/0615jef.html

[1] اصطلاح «کاربرپسندی» در همین حیطه به‌کار رفته است. البته، مسأله این است که برداشت یک کاربر از «موردپسندبودن» ممکن است با برداشت کاربر دیگر، تفاوت بنیادی داشته باشد.

شما به‌ناگزیر در طراحی آزمون‌های قابلیت استفاده سهم خواهید داشت، ولی خود آزمون‌ها توسط کاربران نهایی انجام می‌شوند. مراحلی که اجرا می‌شوند، عبارتند از [Spl01]:

۱. تعریف مجموعه‌ای از گروه‌های آزمون قابلیت استفاده و شناسایی اهداف مربوط به هر گروه.
۲. تعریف آزمون‌هایی که ارزیابی هر هدف را میسر می‌سازند.
۳. انتخاب مشارکت کنندگانی که آزمون‌ها را اجرا می‌کنند.
۴. تعامل مشارکت‌کنندگان با برنامه‌ی تحت وب در حالی که آزمون در حال اجراست.
۵. توسعه‌ی سازوکاری برای ارزیابی قابلیت استفاده‌ی برنامه‌ی تحت وب.

آزمون قابلیت استفاده می‌تواند در سطوح متفاوتی از انتزاع رخ دهد: (۱) قابلیت استفاده از یک سازوکار واسط ویژه (مثلاً یک فرم) را می‌توان ارزیابی کرد، (۲) قابلیت استفاده از یک صفحه‌ی وب کامل (شامل سازوکارهای واسط، اشیای داده‌ای و قابلیت‌های مرتبط) را می‌توان تعیین کرد یا (۳) قابلیت استفاده از برنامه‌ی تحت وب کامل را مد نظر قرار داد.

نخستین گام در آزمون قابلیت استفاده، شناسایی مجموعه گروه‌های قابلیت استفاده و وضع اهداف آزمون برای هر گروه است. مجموعه اهداف و گروه‌های آزمون زیر (که به شکل پرسش نوشته می‌شوند) این رویکرد را نشان می‌دهند[1].

قابلیت تعامل- آیا سازوکارهای تعامل (مثلاً منوهای بازشونده، دکمه‌ها، اشاره‌گرها) به آسانی قابل درک و استفاده‌اند؟

چیدمان- آیا سازوکارهای گشت‌وگذار، محتوا و قابلیت‌های عملیاتی به شیوه‌ای قرار داده شده‌اند که کاربر بتواند آنها را به سهولت بیابد؟

خوانایی- آیا منون به خوبی نوشته شده‌اند و قابل درک هستند؟ آیا نمایش‌های گرافیکی را می‌توان به آسانی درک کرد؟

زیبایی شناسی- آیا چیدمان، رنگ، نوع فونت و خصوصیات مرتبط، به‌سهولت استفاده می‌انجامد؟ آیا کاربران «با ظاهر برنامه‌ی تحت وب احساس راحتی می‌کنند؟

خصوصیات صفحه نمایش- آیا برنامه‌ی تحت وب از اندازه صفحه و تفکیک آن استفاده‌ی بهینه به عمل می‌آورد؟

حساسیت زمانی- آیا ویژگی‌ها، قابلیت‌های عملیاتی و محتوا را می‌توان به شیوه‌ای سر وقت به‌کار برد یا به‌دست آورد؟

شخصی‌سازی- آیا برنامه‌ی تحت وب را می‌توان مطابق با نیازهای خاص گروه‌های متفاوت کاربران یا تک تک کاربران سفارشی کرد؟

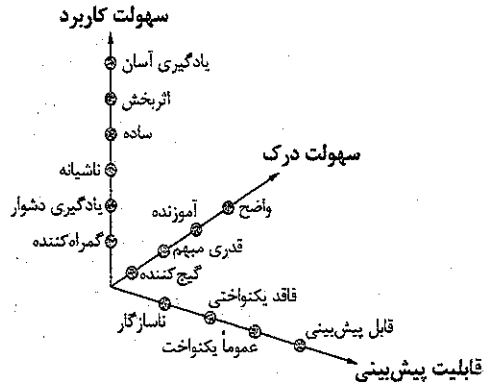
دسترس‌پذیری- آیا برنامه‌ی تحت وب برای افرادی با ناتوانی‌های خاص قابل دستیابی هست؟

در هر کدام از این گروه‌ها یک سری آزمون طراحی می‌شود. در برخی موارد، آزمون ممکن است مرور بصری یک صفحه وب باشد. در سایر موارد، آزمون‌های معناشناختی ممکن است دوباره اجرا شود. ولی در این مورد، دغدغه‌های مربوط به قابلیت استفاده بیشترین اهمیت را دارند. برای مثال، ارزیابی قابلیت (استفاده برای تعامل و سازوکارهای واسط را در نظر می‌گیریم. کنستانتین و لاکوود

کدام خصوصیات قابلیت استفاده در آزمون، کانون توجه قرار می‌گیرند و چه اهداف خاصی باید دنبال شوند؟

[1] برای اطلاعات بیشتر درباره‌ی قابلیت استفاده، فصل ۱۱ را ببینید.

[Con99] پیشنهاد می‌کنند که فهرست ویژگی‌های واسط زیر باید از نظر قابلیت استفاده، مرور گردد: پویانمایی، دکمه‌ها، رنگ، کنترل، کادرهای دیالوگ، فیلدها، فرم‌ها، قاب‌ها، گرافیک‌ها، نشانه‌ها، پیوندها، منوها، پیام‌ها، گشت وگذار، صفحات، سلکتورها، متون و نوارهای ابزار. با ارزیابی هر ویژگی کاربرانی که آزمون را انجام می‌دهند، در یک مقیاس کیفی به آن ویژگی امتیاز می‌دهند. در شکل ۳-۲۰، مجموعه‌ای ممکن از «امتیازهای» قابل انتخاب توسط کاربران تصویر شده است. این امتیازها برای هر ویژگی به‌طور مجزا به کار می‌روند تا یک صفحه وب یا کل برنامه‌ی تحت وب کامل شود.



شکل ۳-۲۰ ارزیابی کیفی قابلیت استفاده.

۵-۴-۲۰ آزمون‌های سازگاری

کامپیوترها، دستگاه‌های نمایش، سیستم‌های عامل، مرورگرها و سرعت‌های متفاوت اتصال شبکه می‌توانند تاثیری چشمگیر بر عملکرد برنامه‌ی تحت وب بگذارند. هر پیکربندی کامپیوتری می‌تواند به اختلاف‌هایی در سرعت پردازش در طرف کلاینت، تفکیک صفحه نمایش و سرعت اتصال منجر شود. تفاوت‌های میان سیستم‌های عامل ممکن است باعث ایجاد مسائل و مشکلاتی در پردازش برنامه‌ی تحت وب شود. مرورگرهای متفاوت گاهی، هر قدر هم که HTML در برنامه‌ی تحت وب استاندارد شود، نتایجی با تفاوت اندک تولید می‌کنند. برای یک پیکربندی خاص، برنامه‌های اتصالی (plug-in) لازم ممکن است به آسانی در دسترس باشند و ممکن هم هست در دسترس نباشند.

در برخی موارد، مسائل سازگاری کوچک هیچ مشکل چشمگیری به بار نمی‌آورند، ولی در سایر موارد، خطاهای جدی ممکن است مشاهده شود. برای مثال، سرعت‌های دانلود ممکن است غیرقابل قبول باشد، فقدان برنامه‌ی اتصالی لازم ممکن است محتوا را از دسترس خارج کند، اختلاف میان مرورگرها می‌تواند چیدمان صفحه را به‌طور جدی تغییر دهد، سبک فونت‌ها ممکن است تغییر داده شود و ناخوانا شود، یا سازمان‌دهی فرم‌ها ممکن است تغییر کند. *آزمون سازگاری* تلاش دارد این مشکلات را قبل از آنلاین‌شدن برنامه‌ی تحت وب کشف کند.

نخستین گام در آزمون سازگاری، تعریف مجموعه‌ای از پیکربندی‌های رایج کلاینت-سرور و شکل‌های متفاوت آنهاست. در اصل، یک ساختار درختی ایجاد می‌شود که در آن هر سکوی کامپیوتری، دستگاه‌های نمایش متداول، سیستم‌های عامل پشتیبانی شده روی سکو، مرورگرهای

نکته‌ی کلیدی
برنامه‌های تحت وب در طرف کلاینت در انواع گسترده‌ای از محیط‌ها اجرا می‌شوند. هدف آزمون سازگاری، کشف خطاهای مرتبط با یک محیط خاص (مثلاً مرورگر) است.

در دسترس، سرعت‌های اتصال اینترنتی محتمل و اطلاعات مشابه شناسایی می‌شود. سپس، یک سری آزمون‌های اعتبارسنجی سازگاری به‌دست می‌آید که غالباً از آزمون‌های واسط، آزمون‌های گشت‌وگذار، آزمون‌های کارایی و آزمون‌های امنیتی موجود گرفته شده‌اند. هدف این آزمون‌ها، کشف خطاها یا مشکلاتی در اجراست که تا اختلاف‌های پیکربندی قابل ردگیری باشند.

۲۰-۵ آزمون در سطح مولفه‌ها

آزمون در سطح مولفه‌ها، که گاهی *آزمون توابع* نیز نامیده می‌شود، مجموعه‌ای از آزمون‌ها را کانون توجه قرار می‌دهد که سعی در کشف خطاهای موجود در توابع برنامه‌ی تحت وب دارند. هر تابع برنامه‌ی تحت وب، یک مولفه از نرم‌افزار است (که در یکی از انواع زبان‌های برنامه‌نویسی یا اسکریپت‌نویسی پیاده‌سازی می‌شود) و با به‌کارگیری تکنیک‌های جعبه سیاه (و در برخی موارد، جعبه‌ی سفید) به صورت بحث شده در فصل ۱۸ قابل آزمایش است.

موارد آزمون در سطح مولفه‌ها غالباً به وسیله‌ی ورودی در سطح فرم‌ها طراحی می‌شود. هنگامی که داده‌های فرم‌ها تعیین شدند، کاربر یک دکمه یا سازوکار کنترلی دیگری را برای شروع اجرا انتخاب می‌کند. روش‌های طراحی زیر برای موارد آزمون رایج هستند (فصل ۱۸):

- *افراز هم‌ارزی*- دامنه ورودی تابع به گروه‌ها یا طبقاتی از ورودی تقسیم می‌شوند که موارد آزمون از آنها به‌دست می‌آیند. شکل ورودی ارزیابی می‌شود تا تعیین شود کدام طبقات از داده‌ها به تابع مربوط می‌شوند. موارد آزمون برای هر دسته از ورودی به‌دست می‌آیند و اجرا می‌شوند، در حالی که سایر دسته‌های ورودی ثابت نگه داشته می‌شوند. برای مثال، در یک برنامه تجارت الکترونیکی ممکن است تابعی پیاده‌سازی شود که هزینه‌های حمل و نقل را محاسبه می‌کند. از جمله انواع اطلاعات حمل و نقلی که از طریق یک فرم فراهم می‌آید، کدپستی کاربر است. در تلاش برای کشف خطاها در پردازش کدپستی با مشخص‌کردن مقادیر کد پستی که ممکن است طبقات متفاوتی از خطاها (مثلاً کدپستی ناقص،کدپستی درست، نبودکدپستی، قالب خطادار برای‌کد پستی) را کشف کنند، یک سری موارد آزمون طراحی می‌شود.
- *تحلیل مقادیر مرزی*- داده‌های به‌دست آمده از فرم‌ها در مرزها آزمایش می‌شوند. برای مثال، تابع محاسبه‌ی هزینه‌ی حمل و نقل که قبلاً ذکر شد، حداکثر تعداد روزهای لازم برای تحویل محصول را درخواست می‌کند. در فرم، حداقل ۲ روز و حداکثر ۱۴ روز ذکر شده است. ولی در آزمون‌های مقادیر مرزی ممکن است مقادیر صفر، ۱، ۲، ۱۳، ۱۴ و ۱۵ وارد شود تا واکنش تابع نسبت به این داده‌ها و خارج از مرزهای ورودی معتبر چگونه است[1].
- *آزمون مسیرها*. اگر پیچیدگی منطقی تابع، بالا باشد[2]، آزمون مسیرها (با روش طراحی جعبه سفید) را می‌توان به کاربرد و اطمینان حاصل کرد که همه‌ی مسیرهای مستقل در برنامه، اجرا شده‌اند.

[1] در این مورد، با طراحی ورودی بهتر می‌توان خطاهای بالقوه را حذف کرد. حداکثر تعداد روزها اگر از یک منوی کرکره‌ای انتخاب شوند، کاربر نمی‌تواند ورودی خارج از مرز بدهد.

[2] پیچیدگی منطقی را می‌توان با محاسبه‌ی پیچیدگی سیکلوماتیک الگوریتم تعیین کرد.برای جزئیات بیشتر، فصل ۱۸ را ببینید.

SafeHome

آزمون برنامه‌ی تحت وب

صحنه: دفتر داگ میلر

نقش‌آفرینان: داگ میلر (مدیر گروه مهندسی نرم‌افزار *SafeHome*) و وینود رامان (عضو تیم مهندسی نرم‌افزار محصول).

گفتگو:

داگ: نظرت درباره تجارت الکترونیک **SafeHomeAssured.com** نسخه‌ی V0.0 چیست؟

وینود: شرکتی که این قسمت از کار را به عهده گرفته، خوب کار کرده است. شارون [مدیر توسعه‌ی شرکت همکار] می‌گوید همین الان که داریم صحبت می‌کنیم، مشغول آزمایش هستند.

داگ: می‌خواهم تو و بقیه‌ی تیم کمی آزمایش غیررسمی روی سایت تجارت الکترونیکی انجام بدهید.

وینود (با طعنه): فکر می‌کردم قرار بود یک شرکت دیگر برای اعتبارسنجی برنامه‌ی تحت وب استخدام کنیم. ما هنوز داریم سعی می‌کنیم محصول را به موقع به بازار برسانیم.

داگ: ما یک شرکت دیگر برای آزمون کارایی و امنیت استخدام خواهیم کرد و شرکتی که بخش تجارت الکترونیک را عهده‌دار شده هم خودش آزمایش می‌کند. فقط فکر کردم یک دیدگاه دیگر می‌تواند مفید باشد و به علاوه، دوست داریم هزینه‌ها خیلی بالا نرود. لذا ...

وینود (آه می‌کشد): دنبال چی هستی؟

داگ: می‌خواهم مطمئن بشوم که واسط و همه‌ی گشت‌وگذار، مستحکم هستند.

وینود: فکر می‌کنم می‌توانیم با موارد آزمون برای هر کدام از عملکردهای واسط اصلی شروع کنیم.

Learn about *SafeHome*.

Specify the *SafeHome* system you need.

Purchase a *SafeHome* system.

Get technical support.

داگ: خوب است. ولی همه‌ی مسیرهای گشت‌وگذار را تا انتها دنبال کنید.

وینود: (در حالی که دفترچه‌های حاوی پرونده‌های کاربرد را ورق می‌زند): بله، وقتی Specify the *SafeHome* system you need را انتخاب می‌کنی، به این بخش می‌روی:

Select *SafeHome* components.

Get SafeHome component recommendations.

می‌توانیم هر مسیر را از نظر معناشناسی تمرین بدهیم.

داگ: در حالی که اینجا هستی، محتوایی را که در هر گره گشت‌وگذار ظاهر می‌شود، چک کن.

وینود: حتماً ... و البته عناصر عملیاتی را. کی قابلیت استفاده را آزمایش می‌کند؟

داگ: آه ... شرکت آزمایش‌گر، آزمایش قابلیت استفاده را هماهنگ می‌کند. ما یک شرکت پژوهش بازار را هم استخدام کرده‌ایم تا بیست کاربر معمولی برای مطالعه‌ی قابلیت استفاده به صف کند، ولی اگر شماها هر مشکلی مرتبط با قابلیت استفاده کشف کردید ...

وینود: می‌دانم، به آن‌ها بگوییم.

داگ: ممنون وینود.

علاوه بر این، در روش‌های طراحی موارد آزمون، تکنیکی موسوم به *آزمون خطاهای واداشته* (Forced Error Testing) [Ngu01] به‌کار می‌رود تا موارد آزمونی به‌دست آید که به‌طور هدف‌مند، مولفه‌های برنامه‌ی تحت وب را به شرایط خطا سوق می‌دهند. هدف، کشف خطاهایی است که طی پرداختن به خطاها رخ می‌دهند (مانند پیام‌های خطای نادرست یا نبود این پیام‌ها، شکست برنامه‌ی تحت وب در نتیجه‌ی خطا، خروجی خطادار در نتیجه‌ی ورودی خطادار، اثرات جانبی مرتبط با پردازش مولفه‌ها).

هر مورد آزمون در سطح مولفه‌ها، کلیه‌ی مقادیر ورودی و خروجی مورد انتظار از مولفه را مشخص می‌کند، خروج واقعی که به‌عنوان نتیجه‌ای از آزمون تولید می‌شود، برای ارجاع‌های بعدی در اثنای پشتیبانی و نگهداری، ثبت می‌شود.

در بسیاری مواقع، اجرای درست یک تابع با برقراری واسط مناسب میان یک بانک اطلاعاتی خارج از برنامه‌ی تحت وب، ارتباطی تنگاتنگ دارد. بنابراین، آزمون بانک اطلاعاتی به بخشی لاینفک از رویکرد آزمون مولفه‌ها تبدیل می‌شود.

۶-۲۰ آزمون گشت‌وگذار

سیاحت کاربر در یک برنامه‌ی تحت وب شباهت بسیار به گشت‌وگذار افراد در یک فروشگاه یا موزه دارد. مسیرهای فراوانی وجود دارد که می‌توان پیش گرفت، در نقاط فراوانی می‌توان توقف کرد، چیزهای بسیاری که می‌توان یاد گرفت و دید، فعالیت‌هایی که می‌توان انجام داد و تصمیم‌هایی که باید گرفته شود. این فرایند گشت‌وگذار از این لحاظ که هر بازدید کننده‌ای هنگام ورود، یک سری اهداف در ذهن دارد، قابل پیش بینی است. در عین حال، فرایند گشت‌وگذار می‌تواند غیرقابل پیش بینی هم باشد زیرا بازدیدکننده، که از چیزهایی که می‌بیند یا فرا می‌گیرد، تاثیر می‌پذیرد، ممکن است مسیری را برگزیند یا کنشی را آغاز کند که برای هدف اولیه‌ی او مناسب نباشد. وظایف آزمون گشت‌وگذار عبارتند از (۱) حصول اطمینان از این که سازوکارهایی که به کاربر امکان می‌دهند تا در برنامه‌ی تحت وب به گشت‌وگذار بپردازد، همگی درست عمل می‌کنند و (۲) اعتبارسنجی اینکه هر واحد معناشناختی گشت‌وگذار (NSU) از طریق گروه مناسبی از کاربران، قابل انجام است.

۱-۶-۲۰ آزمون نحوی گشت‌وگذار

نخستین مرحله از آزمون گشت‌وگذار، در واقع طی آزمون واسط آغاز می‌شود. سازوکارهای گشت‌وگذار، آزمایش می‌شوند تا اطمینان حاصل شود که هر کدام وظیفه‌ی خاص خود را اجرا می‌کند. اسپلین و جسکیل [Spl01] پیشنهاد می‌کنند که هر کدام از سازوکارهای گشت‌وگذار زیر باید آزمایش شود:

- *پیوندهای گشت‌وگذار*- این سازوکار شامل پیوندهای درونی در داخل برنامه‌ی تحت وب، پیوندهای بیرونی منتهی به سایر برنامه‌های تحت وب و لنگرهایی در داخل یک صفحه وب خاص می‌شود. هر پیوند باید آزمایش شود تا اطمینان حاصل شود که هنگام انتخاب پیوند، محتوا یا عملکرد مناسب دریافت می‌شود.
- *تغییر مسیرها*- این پیوندها هنگامی وارد عمل می‌شوند که کاربر یک URL انتخاب می‌کند که دیگر وجود ندارد یا پیوندی را انتخاب می‌کند که محتوای آن حذف شده است یا نام آن تغییر

«ما گم نشده‌ایم بلکه از نظر مکانی با جالش مواجه شده‌ایم»

جان ام. فورد

پیدا کرده است. پیامی برای کاربر به نمایش در می‌آید و گشت‌وگذار به صفحه‌ی دیگر تغییر مسیر داده می‌شود (مثلاً به صفحه اصلی وب‌سایت). تغییر مسیرها باید با درخواست URLهای خارجی یا پیوندهای درونی نادرست و ارزیابی چگونگی رویارویی برنامه‌ی تحت وب با این درخواست‌ها آزمایش شود.

- *چوب الف‌ها* (Bookmarks)- گرچه چوب الف‌ها از جمله قابلیت‌های مرورگر به شمار می‌روند، برنامه‌ی تحت وب باید آزمایش شود تا اطمینان حاصل شود که با ایجاد یک چوب الف، عنوان صفحه‌ی معنی داری قابل استخراج است.
- *قاب‌ها و مجموعه قاب‌ها*- هر قاب حاوی محتوای یک صفحه‌ی وب خاص است و یک مجموعه قاب، حاوی چند قاب بوده، نمایش همزمان چند صفحه وب را امکان‌پذیر می‌سازد. از آنجا که ایجاد ساختار تودرتو برای قاب‌ها و مجموعه قاب‌ها امکان‌پذیر است، این سازوکارهای گشت‌وگذار و صفحه نمایش باید از نظر درستی محتوا، مناسب بودن چیدمان و اندازه‌ها، کارایی در دانلود و سازگاری مرورگر آزمایش شوند.
- *نقشه‌های سایت*- نقشه سایت، جدول کاملی از محتوای همه‌ی صفحات وب فراهم می‌سازد. هر مدخل از نقشه سایت باید آزمایش شود تا اطمینان حاصل شود که پیوندها کاربر را به محتوا یا قابلیت عملیاتی مناسب رهنمون می‌شدند.
- *موتورهای جستجوی درونی*- برنامه‌های تحت وب پیچیده غالباً حاوی صدها یا حتی هزاران شیء محتوایی‌اند. در صورت وجود موتور جستجوی درونی، کاربر می‌تواند یک جستجوی کلید واژه‌ای در داخل برنامه‌ی تحت وب اجرا کند تا محتوای مورد نیاز را بیابد. با آزمایش موتور جستجو، صحت و کامل بودن جستجو، خواص مقابله با خطای موتور جستجو و ویژگی‌های جستجوی پیشرفته (مثلاً استفاده از عملگرهای بولی در فیلد جستجو) اعتبارسنجی می‌شود.

برخی آزمون‌های ذکر شده را می‌توان با ابزارهای خودکار (نظیر چک کننده‌های پیوندها) انجام داد، درحالی که عده‌ای دیگر به صورت دستی، طراحی و اجرا می‌شوند. هدف و مقصود کلی، حصول اطمینان از پیدا شدن خطاهای موجود در گشت‌وگذار است قبل از این که برنامه‌ی تحت وب آنلاین شود.

۲-۶-۲۰ آزمون معناشناختی گشت‌وگذار

با آزمایش هر NSU چه پرسش‌هایی باید پرسیده و پاسخ داده شوند؟

در فصل ۱۳، واحد معناشناختی گشت‌وگذار (NSU) را به‌عنوان «مجموعه‌ای از اطلاعات و ساختارهای گشت‌وگذاری مرتبط دانستیم که با همکاری یکدیگر، زیرمجموعه‌ای از خواسته‌های مرتبط کاربر را برآورده می‌سازند» [Cac02]. هر NSU توسط مجموعه‌ای از مسیرهای گشت‌وگذار (موسوم به «راه‌های گشت‌وگذار») تعریف می‌شود که گروه‌های گشت‌وگذار (مانند صفحات وب، اشیای محتوایی یا قابلیت‌های عملیاتی) را به یکدیگر متصل می‌کنند. هر NSU در کل به کاربر این امکان را می‌دهد تا به خواسته‌های مشخص تعیین شده در یک یا چند use case برای گروهی از کاربران دست پیدا کند. آزمون گشت‌وگذار، تمامی NSUها را تمرین می‌دهد تا اطمینان حاصل شود که این خواسته‌ها قابل دستیابی‌اند. با آزمایش هر NSU باید به پرسش‌های زیر پاسخ گویید:

- آیا NSU به‌طور کامل و بدون خطا قابل دستیابی است؟

- آیا هر گروه گشت‌وگذار (که برای یک NSU تعریف می‌شود) در حیطه‌ی مسیرهای گشت‌وگذار تعریف شده برای آن NSU قابل دستیابی است؟
- اگر NSU با به‌کارگیری بیش از یک مسیر گشت‌وگذار قابل دستیابی است، آیا همه‌ی مسیرهای مرتبط آزمایش می‌شوند؟
- اگر واسط کاربر برای کمک به گشت‌وگذار، راهنمایی ارائه می‌دهد، آیا دستورالعمل‌ها با پیشرفت گشت‌وگذار، درست و قابل درک هستند؟
- آیا سازوکاری (غیر از پیکان Back مرورگر) برای بازگشت به گره گشت‌وگذاری قبلی و شروع مسیر گشت‌وگذار وجود دارد؟
- آیا سازوکارهای مربوط به گشت‌وگذار، در یک گره گشت‌وگذاری بزرگ (یعنی یک صفحه وب بلند و طولانی) درست عمل می‌کنند؟
- اگر قرار باشد تابعی در یک گره اجرا شود و کاربر تصمیم بگیرد که ورودی برای آن ارائه ندهد، آیا باقیمانده‌ی NSU را می‌توان کامل کرد؟
- اگر قرار باشد تابعی در یک گره اجرا شود و خطایی در پردازش تابع رخ دهد، آیا NSU را می‌توان کامل کرد؟
- آیا راهی برای ادامه ندادن به گشت‌وگذار، قبل از رسیدن به همه‌ی گره‌ها وجود دارد، به‌طوری که بعداً دوباره بتوان از همان نقطه که گشت‌وگذار متوقف شده دوباره ادامه داد؟
- آیا هر گره از طریق نقشه‌ی سایت قابل دستیابی است؟ آیا نام‌های گره‌ها برای کاربران معنی دارند؟
- اگر گرهی در یک NSU از یک منبع خارجی قابل دستیابی است، آیا پردازش گره بعدی روی مسیر گشت‌وگذار امکان‌پذیر است؟ آیا روی مسیر گشت‌وگذار می‌توان به گره قبلی بازگشت؟
- آیا کاربر هنگام اجرای NSU موقعیت خود را در معماری محتوا می‌داند؟

آزمون گشت‌وگذار، همانند آزمون واسط و قابلیت استفاده، باید توسط هر تعداد ممکن از هیأت‌ها اجرا شود. مسؤولیت مراحل اولیه‌ی آزمون گشت‌وگذار بر عهده‌ی شماست، ولی مراحل بعدی باید توسط سایر ذی‌نفع‌ها، یک تیم آزمون‌گر مستقل و سرانجام توسط کاربران غیرفنی اجرا شود.

هدف، تمرین دادن کامل گشت‌وگذار برنامه‌ی تحت وب است.

اندرز:
اگر NSUها به‌عنوان بخشی از تحلیل یا طراحی برنامه‌ی تحت وب ایجاد نشده باشند، می‌توانید برای طراحی موارد آزمون از use caseها بهره ببرید. همان مجموعه پرسش‌ها پرسیده و پاسخ داده می‌شوند.

۲۰-۷ آزمون پیکربندی

تغییرپذیری و ناپایداری پیکربندی، عوامل مهمی هستند که آزمون برنامه‌ی تحت وب را به چالش می‌کشند. سخت افزار، سیستم(های) عامل، مرورگرها، ظرفیت ذخیره‌سازی، سرعت ارتباط شبکه‌ای و انواع عوامل دیگر در طرف کلاینت را مشکل بتوان برای تک تک کاربران پیش بینی کرد. به علاوه، پیکربندی برای یک کاربر مفروض می‌تواند مرتب تغییر کند [مثلاً ارتقای سیستم عامل، ISP جدید و تغییر سرعت اتصال]. نتیجه می‌تواند محیطی در طرف کلاینت باشد که مستعد خطاهایی ظریف و در عین حال مهم است. اگر دو کاربر با پیکربندی‌های مشابه در طرف کلاینت کار نکنند، برداشت یک کاربر از برنامه‌ی تحت وب و شیوه‌ی تعامل او با آن می‌تواند با تجربه‌ی کاربر دیگر، تفاوتی اساسی داشته باشد.

وظیفه‌ی آزمون پیکربندی، تمرین دادن هر پیکربندی ممکن در طرف کلاینت نیست. در عوض، باید مجموعه‌ای از پیکربندی‌های محتمل در طرف کلاینت و در طرف سرور را بیازمایید تا اطمینان حاصل شود که تجربه‌ی کاربر، روی همه‌ی آنها یکسان خواهد بود و خطاهایی را که ممکن است خاص یک پیکربندی ویژه باشند، جدا کند.

۲۰-۷-۱ مسائل طرف سرور

موارد آزمون پیکربندی در طرف سرور طوری طراحی می‌شوند که وارسی کنند آیا اطلاعات پیش‌بینی‌شده برای سرور [یعنی سرور برنامه‌ی تحت وب، سرور بانک اطلاعاتی، سیستم(های) عامل، نرم‌افزار دیوار آتش، برنامه‌های کاربردی همروند] می‌توانند بدون خطا، برنامه‌ی تحت وب را پشتیبانی کنند. در اصل، برنامه‌ی تحت وب در محیط طرف سرور نصب و آزمایش می‌شود تا این اطمینان حاصل شود که بدون خطا کار می‌کند.

در همان حال که آزمون‌های پیکربندی طرف سرور طراحی می‌شوند، باید هر مولفه از پیکربندی سرور را مد نظر قرار دهید. از جمله پرسش‌هایی که باید طی آزمون پیکربندی طرف سرور پرسید و به آنها پاسخ گفت، عبارتند از:

- آیا برنامه‌ی تحت وب به‌طور کامل با سیستم عامل سرور سازگار است؟
- آیا فایل‌های سیستمی، شاخه‌ها و داده‌های سیستمی مرتبط، هنگام عملیاتی شدن برنامه‌ی تحت وب به‌طور صحیحی ایجاد می‌شوند؟
- آیا راهکارهای امنیتی سیستم (مانند دیوارهای آتش یا کپسوله) به برنامه‌ی تحت وب امکان می‌دهند که اجرا شود و بدون تداخل یا کاهش در کارایی، به کاربران ارائه‌ی خدمات کند؟
- آیا برنامه‌ی تحت وب با پیکربندی سرور توزیع شده‌ی انتخاب شده (در صورت وجود) آزمایش شده است[1]؟
- آیا برنامه‌ی تحت وب از انسجام مناسب با نرم‌افزار بانک اطلاعاتی برخوردار هست؟
- آیا برنامه‌ی تحت وب به تفاوت نسخه در نرم‌افزار بانک اطلاعاتی حساس است؟
- آیا اسکریپت‌های برنامه‌ی تحت وب در طرف سرور به‌طور مناسب اجرا می‌شوند؟
- آیا خطاهای مدیریت سیستم از نظر تاثیری که بر عملکرد برنامه‌ی تحت وب می‌توانند داشته باشند، بررسی شده‌اند؟
- اگر از سرورهای پروکسی استفاده می‌شود، آیا اختلاف در پیکربندی آنها در آزمون روی سایت در نظر گرفته شده است؟

به هنگام اجرای آزمون پیکربندی در طرف سرور، چه پرسش‌هایی باید پرسیده و پاسخ داده شوند؟

۲۰-۷-۲ مسائل طرف کلاینت

در طرف کلاینت، آزمون‌های پیکربندی، بر سازگاری با پیکربندی‌های حاوی یک یا چند ترکیب جایگشتی از مولفه‌های زیر تأکید دارند [Ngu01]:

- *سخت افزار*- CPU، حافظه، دیسک‌ها و دستگاه‌های چاپ

[1] برای مثال، از یک کارگزار جداگانه برای برنامه کاربردی و بانک اطلاعاتی ممکن است استفاده شود. برقراری ارتباط میان دو ماشین از طریق یک اتصال شبکه‌ای رخ می‌دهد.

- *سیستم‌های عامل*- Linux، Macintosh، Windows، یک سیستم عامل تلفن همراه.
- *نرم‌افزار مرورگر*- Fire Fox، Safari، Chrome، Internet Explorer، Opera و غیره.
- *مولفه‌های واسط کاربر*- Active X، اپلت‌های Java و غیره.
- *برنامه‌های اتصالی*- Quick Time، Real Player و بسیاری دیگر.
- *اتصال*- کابل، DSL، مودم‌های معمولی، T1، WiFi.

علاوه بر این مولفه‌ها، متغیرهای دیگر عبارتند از نرم‌افزار شبکه سازی، تغییرات ISP و برنامه‌های کاربردی‌ای که همزمان اجرا می‌شوند.

برای طراحی آزمون‌های پیکربندی در طرف کلاینت، باید تعداد متغیرهای پیکربندی را به تعدادی تقلیل دهید که قابل مدیریت باشد[1]. برای دستیابی به این منظور، همه‌ی گروه‌های کاربران ارزیابی می‌شوند تا پیکربندی‌های محتملی که ممکن است در هر گروه مشاهده شوند، تعیین شود. به علاوه، برای پیش‌بینی محتمل‌ترین سهم‌های مولفه‌ها می‌توان از داده‌های سهم بازار استفاده کرد. سپس برنامه‌ی تحت وب در این محیط‌ها آزمایش می‌شود.

۸-۲۰ آزمون امنیت

امنیت برنامه‌ی تحت وب موضوعی پیچیده است که باید پیش از دستیابی به آزمون امنیتی اثربخش، آن را به‌طور کامل درک کرد[2]. برنامه‌های تحت وب و محیط‌هایی که طرف سرور و طرف کلاینت در آنها قرار دارند، برای نفوذگران کارمندان ناراضی، رقیبان متقلب و هر کس دیگری که می‌خواهد اطلاعات حساس را سرقت کند، محتوا را خدشه‌دار کند، کارایی را تنزل دهد، قابلیت عملیاتی را مختل سازد یا شخصی، سازمانی یا شرکتی را شرمنده سازد، هدفی جذاب به شمار می‌رود.

آزمون‌های امنیتی طوری طراحی می‌شوند که آسیب پذیری‌های محیط طرف کلاینت، ارتباطات شبکه‌ای که در تبادل اطلاعات میان سرور و کلاینت رخ می‌دهند و محیط طرف سرور را برملا سازند. به هر کدام از این دامنه‌ها می‌توان حمله کرد و کشف نقاط ضعفی که توسط افراد سودجو ممکن است مورد سوءاستفاده قرار گیرد، وظیفه‌ی آزمونگر امنیتی است.

در طرف کلاینت، آسیب‌پذیری‌ها را غالباً می‌توان تا اشکال‌های موجود در مرورگرها، برنامه‌های پست الکترونیکی، یا نرم‌افزار ارتباطی ردگیری کرد. نگوین [Ngu01] حفره امنیتی را چنین توصیف می‌کند:

> یکی از اشکال‌هایی که معمولاً ذکر می‌شود، سرریز بافر است که اجرای کدهای زیان‌بار را روی ماشین کلاینت میسر می‌سازد. برای مثال، اگر مرورگر فاقد کد تشخیص خطا برای اعتبارسنجی طول URL وارد شده باشد، واردکردن یک URL بسیار طولانی در مرورگری که طول بافر اختصاص داده شده به URL در آن بسیار کوچکتر است، باعث خطای رونویسی حافظه (سرریز حافظه) می‌شود. نفوذگران خبره می‌توانند هوشمندانه با نوشتن یک URL بلند با کد قابل اجرا باعث از هم پاشیدن مرورگر شوند یا تنظیمات امنیتی را (از بالا به پایین) تغییر دهند یا بدتر از آن، داده‌های کاربر را از بین ببرند.

[1] اجرای آزمون‌ها روی همه‌ی ترکیب‌های ممکن از مولفه‌های پیکربندی بسیار بسیار وقت گیر است.

[2] درباره این مطلب، اطلاعات مفیدی را که در کتاب‌های کراس و فیشر [Cro01]، اندروز و ویتیکر [And06] و تریودی [Tri03] می‌توانید بیابید.

> «اینترنت جایی پرخطر برای انجام تجارت یا نگهداری دارایی‌هاست. نفوذگران، فریب‌کاران، ...، ویروس‌نویسان و فروشندگان بی‌انصاف در آن می‌تازند.»
>
> دوروتی و پیتر دنینگ

اندرز

اگر برنامه‌ی تحت وب از نظر تجاری اهمیت حیاتی داشته باشد، حاوی داده‌های حساس باشد یا این احتمال که هدف نفوذگران قرار گیرد، زیاد باشد، برون‌سپاری آزمایش امنیت آن به یک شرکت متخصص در این امر، فکر خوبی است.

نکته‌ی کلیدی

آزمون‌های امنیتی باید طوری طراحی شوند که دیوارهای آتش، صدور مجوز، پنهان‌سازی و احراز هویت را تمرین دهند.

یک آسیب‌پذیری بالقوه‌ی دیگر در طرف کلاینت، دستیابی غیرمجاز به کوکی‌های قرار داده شده در مرورگر است. وب‌سایت‌های طراحی شده با اهداف و نیات سوء می‌توانند اطلاعات موجود در کوکی‌های قانونی را به‌دست آورند و از این اطلاعات به شیوه‌هایی استفاده کنند که حریم خصوصی کاربر را به خطر اندازد یا بدتر از آن، صحنه را برای سرقت هویت آماده سازد.

داده‌های تبادل شده میان سرور و کلاینت نسبت به Spoofing آسیب‌پذیرند. Spoofing هنگامی رخ می‌دهد که یک انتهای مسیر ارتباطی توسط موجودیتی با نیت‌های سوء تخریب گردد. برای مثال، وب‌سایتی که وانمود می‌کند سرور قانونی یک برنامه‌ی تحت وب است (با گرفتن ظاهر و شکلی مشابه)، می‌تواند کاربران را فریب دهد. هدف، دزدیدن کلمه‌های عبور، اطلاعات شخصی و داده‌های مربوط به کارت اعتباری است.

از سوی دیگر، آسیب‌پذیری‌ها شامل حملات انکار سرویس (Denial of Service) و اسکریپت‌های زیان‌باری می‌شوند که ممکن است به ماشین کلاینت راه پیدا کنند یا برای از کار انداختن سرور استفاده شوند. به علاوه، بانک‌های اطلاعاتی در طرف سرور، بدون کسب اجازه قابل دستیابی خواهد بود (دزدی داده‌ها).

برای محافظت در مقابل این آسیب‌پذیری‌ها (و سایر آسیب‌پذیری‌ها)، یک یا چند عنصر امنیتی پیاده‌سازی می‌شود [Ngu01]:

- *دیوار آتش*- یک سازوکار فیلترکردن که تلفیقی از نرم‌افزار و سخت افزار بوده هر بسته‌ی ورودی اطلاعات را بررسی می‌کند تا اطمینان حاصل شود که این بسته از طرف منبعی قانونی می‌آید و سد راه داده‌های مشکوک می‌شود.
- *احراز هویت*- یک سازوکار وارسی که هویت همه‌ی سرورها و کلاینت‌ها را اعتبارسنجی می‌کند و تنها هنگامی برقراری ارتباطات را امکان‌پذیر می‌سازد که هر دو طرف وارسی شده باشند.
- *پنهان‌سازی*- سازوکاری برای رمزنگاری که داده‌های حساس را محافظت می‌کند به این ترتیب که آنها را طوری اصلاح می‌کند که خواندن آنها توسط افرادی با نیات سوء غیرممکن شود. پنهان‌سازی با به‌کارگیری *گواهی نامه‌های دیجیتالی* که به کلاینت این امکان را می‌دهند تا مقصد ارسال داده‌ها را وارسی کند، تقویت می‌شود.
- *اخذ مجوز*- یک سازوکار فیلتری که دستیابی به کلاینت یا سرور را تنها توسط افرادی مجاز می‌شمارد که دارای کدهای اخذ مجوز مناسب (نام کاربر و کلمه‌ی عبور) باشند.

برای پرداختن به هر کدام از این فن‌آوری‌های امنیتی باید آزمون‌های خاصی طراحی شود تا حفره‌های امنیتی کشف شود.

طراحی واقعی آزمون‌ها به دانشی عمیق از عملکرد درونی هر کدام از عناصر امنیتی و درکی جامع از گستره کاملی از فن‌آوری‌های شبکه‌سازی نیاز دارد. در بسیاری موارد، آزمون امنیتی به شرکت‌هایی برون سپاری می‌شود که در این فن‌آوری‌ها تخصص دارند.

۹-۲۰ آزمون کارایی

هیچ چیز ناراحت کننده‌تر از برنامه‌ی تحت وبی نیست که چند دقیقه وقت صرف دانلود محتوا کند،

درحالی که سایت‌های رقیب همان محتوا را در عرض چند ثانیه دانلود می‌کنند. هیچ چیز بدتر از این نیست که تلاش کنید وارد یک برنامه‌ی تحت وب شوید و پیام «Server Busy» را دریافت کنید که پیشنهاد می‌کند بعداً به سایت مراجعه کنید. هیچ چیز بیشتر از این اعصاب انسان را به هم نمی‌ریزد که برنامه‌ی تحت وب لحظه‌ای پاسخ دهد و سپس به نظر برسد که سایت در سایر شرایط وارد یک حالت انتظار بی‌پایان شده است. همه‌ی این رخ دادها هر روز در وب رخ می‌دهد و همه‌ی آنها با کارایی در ارتباط است.

از *آزمون کارایی* برای کشف مسائل کارایی استفاده می‌شود که ممکن است در اثر فقدان منابع طرف سرور، پهنای باند نامناسب برای شبکه، قابلیت‌های ناکافی بانک اطلاعاتی، قابلیت‌های ناقص یا ضعیف سیستم عامل، قابلیت‌های عملیاتی برنامه‌ی تحت وب با طراحی ضعیف و سایر مسائل نرم‌افزار که ممکن است به کاهش کارایی کلاینت-سرور بینجامد، رخ می‌دهد. هدفی دوگانه دنبال می‌شود: (۱) درک چگونگی پاسخ گویی سیستم با افزایش بار تحمیل شده (یعنی تعداد کاربران، تعداد تراکنش‌ها یا حجم داده‌ها) و (۲) جمع‌آوری معیارهایی که به اصلاحاتی در طراحی برای بهبودبخشیدن به کارایی می‌انجامند.

۱-۹-۲۰ اهداف آزمون کارایی

آزمون‌های کارایی برای شبیه‌سازی شرایط ازدحام بار در جهان واقعی طراحی می‌شوند. با رشد تعداد کاربران همزمان برنامه‌ی تحت وب، یا افزایش تراکنش‌های آنلاین، یا افزایش مقدار داده‌ها (دانلودشده یا آپلودشده)، آزمون کارایی به پرسش‌های زیر پاسخ خواهیم داد:

- آیا زمان پاسخ سرور به نقطه‌ای قابل تشخیص و غیرقابل قبول تنزل پیدا می‌کند؟
- در چه نقطه‌ای (برحسب ازدحام بار کاربران، تراکنش‌ها، یا داده‌ها) کارایی غیرقابل قبول می‌شود؟
- کدام مولفه‌های سیستم مسؤول تنزل کارایی‌اند؟
- زمان پاسخ میانگین برای کاربران، تحت انواع شرایط ازدحام بار چقدر است؟
- آیا تنزل کارایی بر امنیت سیستم تاثیر دارد؟
- آیا قابلیت اطمینان یا صحت برنامه‌ی تحت‌وب با رشد ازدحام بار روی سیستم تاثیر می‌پذیرد؟
- هنگامی که ازدحام بار از ظرفیت بیشینه‌ی سرور فراتر می‌رود، چه اتفاقی رخ می‌دهد؟
- آیا تنزل کارایی بر درآمد شرکت تاثیر دارد؟

برای ارائه پاسخ به این پرسش‌ها دو آزمون کارایی متفاوت اجرا می‌شود: (۱) *آزمون ازدحام بار*، میزان بار تحمیل شده در انواع سطوح باری و انواع ترکیب‌ها را بررسی می‌کند و (۲) *آزمون فشار*، بار وارد شده را تا نقطه‌ی شکست افزایش می‌دهد تا تعیین شود محیط برنامه‌ی تحت وب چه مقدار ظرفیت را می‌تواند مدیریت کند. هر یک از این راهبردها را در بخش‌های بعدی بررسی خواهیم کرد.

اندرز

آزمایش برخی جنبه‌های کارایی برنامه‌ی تحت وب، حداقل از دیدگاه کاربر نهایی، دشوار است. ازدحام بار شبکه، تغییرات سخت‌افزارهای واسط و مسائل مشابه در سطح برنامه‌ی تحت وب به آسانی قابل آزمایش نیستند.

۲-۹-۲۰ آزمون ازدحام بار (Load Testing)

هدف آزمون ازدحام بار، تعیین چگونگی پاسخ‌دهی برنامه‌ی تحت وب و محیط سرور به انواع شرایط ازدحام بار است. با پیشرفت آزمون، تبدیلات جایگشتی متغیرهای زیر، یک مجموعه شرایط آزمون تعریف می‌کند:

N: تعداد کاربران همزمان

T: تعداد تراکنش‌های آنلاین در واحد زمان

D: ازدحام بار داده‌ای پردازش شده توسط سرور به ازای هر تراکنش

در هر مورد، این متغیرها در داخل مرزهای عملیاتی سیستم تعریف می‌شوند. با اجرای هر کدام از شرایط آزمون، یک یا چند مورد از موازین زیر جمع آوری می‌شود: میانگین پاسخ کاربران، میانگین زمان دانلود یک واحد استاندارد از داده‌ها، یا زمان میانگین لازم برای پردازش یک تراکنش. باید این موازین را بررسی کنید تا تعیین شود که آیا کاهشی سریع در کارایی تا ترکیب مشخصی از N، T و D قابل ردگیری هست یا خیر.

اندرز

اگر برنامه‌ی تحت وبی برای فراهم آوردن ظرفیت بالا از چند سرور استفاده می‌کند، آزمون ازدحام بار باید در محیط چندسروری اجرا شود.

از آزمون ازدحام بار در ارزیابی سرعت‌های اتصال توصیه شده برای کاربران برنامه‌ی تحت وب می‌توان استفاده کرد. توان عملیاتی کلی، P، به شیوه زیر محاسبه می‌شود:

$$P = N \times T \times D$$

به‌عنوان مثال، یک سایت خبری ورزشی پرطرفدار را در نظر بگیرید. در یک لحظه‌ی معین، ۲۰۰۰۰ کاربر، همزمان به‌طور میانگین هر دو دقیقه یک درخواست (تراکنش T) تسلیم می‌کنند. هر تراکنش مستلزم آن است که برنامه‌ی تحت وب، مقاله‌ی جدیدی به طول میانگین ۳ کیلو بایت دانلود کند. بنابراین، توان عملیاتی را می‌توان به صورت زیر محاسبه کرد:

$$P = [۲۰۰۰۰ \times ۰/۵ \times ۳\text{KB}]/۶۰ = ۵۰۰\text{KB/sec} = ۴\text{(مگابیت بر ثانیه)}$$

بنابراین، اتصال شبکه برای سرور باید این سرعت انتقال داده‌ها را پشتیبانی کند و باید آزمایش شود تا اطمینان حاصل شود که چنین قابلیتی را دارد.

۳-۹-۲۰ آزمون فشار (Stress Testing)

آزمون فشار، ادامه‌ی آزمون ازدحام بار است، ولی در این مورد، متغیرهای N، T و D به سمت مقادیر عملیاتی حدی سوق داده می‌شوند و سپس از این مقادیر فراتر می‌روند. هدف از این آزمون‌ها، پاسخ دادن به هر کدام از پرسش‌های زیر است:

نکته‌ی کلیدی

هدف آزمون فشار، درک بهتر چگونگی شکست‌خوردن یک سیستم با اعمال فشارهایی ورای حدود عملیاتی آن است.

- آیا با فراتر رفتن از ظرفیت‌ها، سیستم «به ملایمت» تنزل می‌یابد یا سرور از کار می‌افتد؟
- آیا نرم‌افزار سرور پیام‌هایی نظیر «سرور در دسترس نیست» تولید می‌کند؟ به‌طور کلی‌تر، آیا کاربران می‌دانند که نمی‌توانند به سرور دست پیدا کنند؟
- آیا سرور، درخواست‌های منابع را صف‌بندی می‌کند و با کاهش یافتن تقاضاهای ظرفیت، صف را خالی می‌کند؟
- آیا با فراتر رفتن از حد ظرفیت، تراکنش‌ها از بین می‌روند؟
- آیا با فراتر رفتن از حد ظرفیت، انسجام داده‌ها تاثیر می‌پذیرد؟
- چه مقادیری از N، T و D محیط سرور را به شکست سوق می‌دهند؟ شکست چگونه خودش را به نمایش می‌گذارد؟ آیا پیام‌هایی به‌طور خودکار به کارمندان پشتیبان فنی مستقر در جایگاه سرور ارسال می‌شود؟
- اگر سیستم با شکست مواجه شود، چه مدت به طول خواهد انجامید تا دوباره آنلاین شود؟

- آیا قابلیت‌های عملیاتی معینی از برنامه‌ی تحت وب (مانند جریان پیوسته اطلاعات یا محاسبه توان عملیاتی) با رسیدن ظرفیت به سطح ۸۰ یا ۹۰ درصد، متوقف می‌شوند؟

شکل دیگری از آزمون فشار گاه به‌عنوان *آزمون spike/bounce* شناخته می‌شود [Spl01]. در این روش آزمون، بار به ظرفیت spike می‌شود، سپس به سرعت تا شرایط عملیاتی عادی کاهش داده و سپس دوباره spike می‌شود. با ایجاد جهش در ازدحام بار می‌توانید تعیین کنید که سرور می‌تواند منابع را سامان‌دهی کند تا تقاضاهای بسیار زیاد را برآورده سازد و سپس با ظهور دوباره‌ی شرایط عادی، این منابع را آزاد سازد (به‌طوری که برای spike بعدی آماده باشند).

ابزارها

طبقه‌بندی ابزارهای مربوط به آزمون برنامه‌های تحت وب

لَم [Lam01] در مقاله خود راجع به سیستم‌های تجارت الکترونیک، از ابزارهای خودکاری که در آزمون برنامه‌های تحت وب کاربرد مستقیم دارند، طبقه‌بندی مفیدی ارئه می‌دهد. ما در هر گروه، چند ابزار نمونه اضافه کردیم.

ابزارهای مدیریت پیکربندی و محتوا، کنترل تغییرات و نسخه‌های اشیای محتوایی و مؤلفه‌های عملیاتی را مدیریت می‌کنند.

ابزار(های) نمونه: فهرست جامعی از این ابزارها را در **www.daveeaton.com/scm/CMTools.html** می‌توانید بیابید.

ابزارهای کارایی بانک اطلاعاتی، کارایی بانک اطلاعاتی را از نظر زمان لازم برای اجرای درخواست از بانک اطلاعاتی اندازه‌گیری می‌کنند. این ابزارها بهینه‌سازی بانک اطلاعاتی را تسهیل می‌کنند.

ابزار(های) نمونه: نرم‌افزار

BMC Software (www.bmc.com)

اشکال‌زداها نمونه ابزارهایی در برنامه‌نویسی هستند که نقایص نرم افزارها را در سطوح کدنویسی، کشف و برطرف می‌کنند. این ابزارها بخشی از اکثر محیط‌های توسعه‌ی برنامه‌های کاربردی مدرن هستند

ابزار(های) نمونه:

Accelerated Technology (**www.acceleratedtechnology.com**)
Apple Debugging Tools (**developer.apple.com/tools/performance/**)
IBM VisualAge Environment (**www.ibm.com**)
Microsoft Debugging Tools (**www.microsoft.com**)

سیستم‌های مدیریت نقایص، نقایص را ثبت کرده وضعیت و برطرف کردن آنها را پیگیری می‌کنند. برخی از این ابزارها شامل ابزارهای گزارش‌دهی می‌شوند که اطلاعاتی درخصوص انتشار نقایص و میزان برطرف شدن آنها در اختیار مدیریت قرار می‌دهند.

ابزار(های) نمونه:

EXCEL Quickbigs (**www.excelsoftware.com**)
ForeSoft BugTrack (**www.bugtrack.com**)
McCabe TrueTrack (**www.mccabe.com**)

ابزارهای پایش شبکه، سطح ترافیک شبکه را پایش می‌کنند. این ابزارها برای شناسایی گلوگاه‌های شبکه و آزمودن پیوند میان سیستم‌های پیشین و پسین مفیدند.

ابزار(های) نمونه: فهرست جامعی از این ابزارها را در نشانی زیر می‌توانید بیابید:

www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html

ابزارهای آزمون رگرسیون، موارد آزمون و داده‌های آزمون را ذخیره می‌کنند و می‌توانند این موارد آزمون را پس از تغییرات بعدی نرم‌افزار دوباره به‌کار ببرند.

ابزار(های) نمونه:

Compuware QARun (**www.compuware.com/products/qacenter/qarun**)
Rational VisualTest (**www.rational.com**)
Seque Softwar (**www.seque.com**)

ابزارهای پایش سایت، کارایی سایت را (غالباً از دیدگاه کاربر) پایش می‌کنند. از آنها برای جمع آوری آمارهایی نظیر زمان پاسخ انتها به انتها و توان عملیاتی و نیز چک کردن ادواری قابلیت دسترسی به سایت استفاده می‌شود.

ابزار(های) نمونه:

Keynote Systems (**www.keynote.com**)

ابزارهای فشار، به سازندگان کمک می‌کنند تا رفتار سیستم را تحت سطوح بالایی از کارکرد عملیاتی بررسی کند و نقاط شکست سیستم را بیابد.

ابزار(های) نمونه:

Mercury Interactive (**www.merc-int.com**)
Open-source testing tools (**www.open-sourcetesting.org/performance.php**)
Web Performance Load Tester (**www.webperformanceinc.com**)

پایش‌گرهای منابع سیستم، بخشی از اکثر نرم افزارهای سرور OS و سرور وب هستند؛ این ابزارها منابعی نظیر فضای دیسک، استفاده از CPU و حافظه را پایش می‌کنند.

ابزار(های) نمونه:

Successful Hosting.com (**www.successfulhosting.com**)
Quest Software Foglight (**www.quest.com**)

ابزارهای تولید داده‌های آزمون، کاربر را در تولید داده‌های آزمون یاری می‌دهند.

ابزار(های) نمونه: فهرست جامعی از این ابزارها را در نشانی زیر می‌توانید بیابید:

www.softwareqatest.com/qatweb1.html

تا با برنامه‌ی تحت وب ارتباط برقرار کند و جنبه‌های زیبایی‌شناختی واسط را نیز اعتبارسنجی می‌کنند. هدف، کشف خطاهایی است که از پیاده‌سازی ضعیف سازوکارهای تعامل یا از ناسازگاری‌ها، ابهامات یا جاافتادگی‌ها در معناشناسی واسط نتیجه می‌شوند.

در آزمون گشت‌وگذار، پرونده‌های کاربرد (که به‌عنوان بخشی از فعالیت مدل‌سازی به می‌آیند) در طراحی موارد آزمونی استفاده می‌شوند که هر کدام از سناریوهای کاربرد را در قبال طراحی گشت‌وگذار تمرین می‌دهند. سازوکارهای گشت‌وگذار مورد آزمایش قرار می‌گیرند تا اطمینان حاصل شود همه‌ی خطاهایی که مانع از کامل شدن یک use case می‌شوند، شناسایی و تصحیح گردند. آزمون مولفه‌ها، واحدهای محتوایی و عملیاتی موجود در برنامه‌ی تحت وب را تمرین می‌دهند.

آزمون پیکربندی تلاش می‌کند تا خطاها و/یا مسائل سازگاری را که مختص یک محیط کلاینت یا یک محیط سرور خاص هستند، برملا سازد. سپس آزمون‌هایی اجرا می‌شوند تا خطاهای مرتبط با هر پیکربندی ممکن آشکار شوند. آزمون امنیتی شامل یک سری آزمون‌های طراحی شده برای سواستفاده از آسیب پذیری‌های موجود در برنامه‌ی تحت وب و محیط آن می‌شود. هدف، یافتن حفره‌های امنیتی است. آزمون کارایی شامل یک سری آزمون می‌شود که برای ارزیابی زمان پاسخ‌دهی و قابلیت اطمینان با افزایش تقاضای ظرفیت منابع در طرف سرور طراحی می‌شوند.

مسائل و نکاتی برای تعمق

۲۰-۱ شرایطی وجود دارد که در آنها، آزمون برنامه‌ی تحت وب به‌طور کامل کنار گذاشته شود؟

۲۰-۲ اهداف آزمون را در حیطه‌ی برنامه‌ی تحت وب به زبان ساده شرح دهید.

۲۰-۳ سازگاری، یک بعد کیفیتی مهم است. چه چیزی باید آزمایش شود تا اطمینان حاصل گردد که سازگاری برای یک برنامه‌ی تحت وب وجود دارد؟

۲۰-۴ کدام خطاها جدی‌ترند- خطاهای طرف سرور یا خطاهای طرف کلاینت؟ چرا؟

۲۰-۵ کدام عناصر برنامه‌ی تحت وب را می‌توان «واحد به واحد آزمود»؟ کدام انواع آزمون‌ها را باید تنها پس از منسجم ساختن عناصر برنامه‌ی تحت وب اجرا کرد؟

۲۰-۶ آیا تهیه‌ی یک برنامه‌ریزی آزمون مکتوب و رسمی همواره ضروری است؟ توضیح دهید.

۲۰-۷ آیا عادلانه است که بگوییم راهبرد کلی آزمون برنامه‌ی تحت وب با عناصری آغاز می‌شود که پیش چشم کاربر قرار دارند و به سمت عناصر فن‌آوری می‌رود؟ آیا این راهبرد استثنا هم دارد؟

۲۰-۸ آیا آزمون محتوا واقعاً از دیدگاه سنتی آزمایش می‌کند؟ توضیح دهید.

۲۰-۹ مراحل مربوط به آزمون بانک اطلاعاتی را برای یک برنامه‌ی تحت وب شرح دهید. آیا آزمون بانک اطلاعاتی بیشتر یک فعالیت در طرف سرور است یا در طرف کلاینت؟

۲۰-۱۰ اختلاف میان آزمون مرتبط با سازوکارهای واسط و آزمونی که به معناشناسی واسط می‌پردازد، چیست؟

۲۰-۱۱ فرض کنید در حال توسعه یک داروخانه‌ی آنلاین (YourCornerPharmacy.com) هستید که برای شهروندان بزرگ سال دارو تامین می‌کند. این داروخانه یک سری قابلیت عملیاتی رایج فراهم می‌سازد و علاوه بر آن، حاوی بانک اطلاعاتی برای هر مشتری است به‌طوری که می‌تواند اطلاعات داروها را ارائه دهد و درباره اثر متقابل داروها هشدار بدهد. درباره هرگونه آزمون قابلیت استفاده برای این برنامه‌ی تحت وب بحث کنید.

۲۰-۱۲ فرض کنید برای YourCornerPharmacy.com (مساله ۲۰-۱۱) یک قابلیت عملیاتی برای چک‌کردن تاثیر متقابل داروها پیاده‌سازی کرده‌اید. درباره انواع آزمون‌ها در سطح مولفه‌ها که باید اجرا شوند تا اطمینان حاصل گردد که این قابلیت عملیاتی به‌طور مناسب عمل می‌کند، بحث کنید. [توجه: برای پیاده‌سازی این قابلیت عملیاتی باید یک بانک اطلاعاتی پیاده‌سازی شود.]

مقایسه‌گرهای نتایج آزمون، به مقایسه‌ی نتایج یک مجموعه از آزمون با نتایج مجموعه‌ای دیگر کمک می‌کنند. با استفاده از آنها می‌توان چک کرد که آیا تغییرات کد باعث تغییرات سوء در رفتار سیستم شده‌اند یا خیر.

ابزار(های) نمونه: فهرست مفیدی از این ابزارها را در نشانی زیر می‌توانید بیابید:

www.aptest.com/resources.html

پایش‌گرهای تراکنش، کارایی سیستم‌های پردازش تراکنشی در حجم انبوه را اندازه‌گیری می‌کنند.

ابزار(های) نمونه:

QuotiumPro (**www.quotium.com**)

Software Research eValid (**www.soft.com/eValid/index.html**)

ابزارهای امنیت وب‌سایت، به آشکارسازی مشکلات امنیتی بالقوه کمک می‌کنند. غالباً می‌توانند ابزارهای پایش و بررسی امنیت را طوری پیکربندی کنند که به صورت زمان‌بندی شده اجرا شوند.

ابزار(های) نمونه: فهرست جامعی از این ابزارها را در نشانی زیر می‌توانید بیابید:

www.timberlinetechnologies.com/products/www.html

۲۰-۱۰ خلاصه

هدف آزمون برنامه‌ی تحت وب، تمرین دادن هر کدام از چندین بعد کیفیت برنامه‌ی تحت وب به‌نیت پیداکردن خطاها یا کشف مسائلی است که ممکن است به شکست‌های کیفیتی منجر شوند. آنچه در این آزمون کانون توجه قرار می‌گیرد، محتوا، قابلیت‌های عملیاتی، ساختار، قابلیت استفاده، قابلیت گشت‌وگذار، کارایی، سازگاری، قابلیت همکاری، ظرفیت و امنیت است. این آزمون شامل مرورهایی می‌شود که به هنگام طراحی برنامه‌ی تحت وب رخ می‌دهند و آزمون‌هایی که پس از پیاده‌سازی برنامه‌ی تحت وب اجرا می‌شوند.

راهبرد آزمون برنامه‌ی تحت وب، همه‌ی ابعاد کیفیتی را تمرین می‌دهد و برای این منظور، پیش از هر چیز، «واحدهای» محتوا، قابلیت‌های عملیاتی یا گشت‌وگذار را بررسی می‌کند. هنگامی که تک تک این واحدها اعتبارسنجی شدند، کانون توجه به آزمون‌هایی جابجا خواهد شد که کل برنامه‌ی تحت وب را تمرین می‌دهد. برای دستیابی به این هدف، آزمون‌های بسیار از دیدگاه کاربر و براساس اطلاعات موجود در پرونده‌های کاربرد به‌دست خواهد آمد. برنامه‌ریزی آزمونی برای برنامه‌ی تحت وب تهیه می‌شود و مراحل آزمون، محصولات کاری (مثلاً موارد آزمون) و سازوکارهایی برای ارزیابی نتایج آزمون تعیین خواهد شد. فرایند آزمون شامل هفت نوع آزمون متفاوت می‌شود.

در آزمون (و مرور) محتوای گروه‌های گوناگون محتوا، کانون توجه قرار می‌گیرد. هدف، کشف هر دو نوع خطای معناشناختی و نحوی است که بر صحت محتوا یا شیوه‌ی ارائه آنها به کاربر نهایی تاثیر می‌گذارند. در آزمون واسط، سازوکارهای تعاملی تمرین داده می‌شوند که کاربر را قادر می‌سازند

۲۰-۱۳ اختلاف میان آزمون مربوط به نحو گشت‌وگذار و معناشناسی گشت‌وگذار در چیست؟

۲۰-۱۴ آیا این امکان وجود دارد که همه‌ی پیکربندی‌های ممکن در طرف سرور آزمایش شوند؟ در طرف کلاینت چطور؟ اگر این امکان وجود ندارد، چگونه مجموعه‌ای بامعنی از آزمون‌های پیکربندی را انتخاب می‌کنید؟

۲۰-۱۵ هدف آزمون امنیتی چیست؟ این فعالیت آزمون را چه کسی اجرا می‌کند؟

۲۰-۱۶ YourCornerPharmacy.com (مساله ۱۱-۲۰) به موفقیت گسترده‌ای دست یافته است و تعداد کاربران در دو ماه نخست راه اندازی آن به‌طور چشمگیری افزایش یافته است. نموداری رسم کنید که زمان پاسخ‌دهی محتمل را به‌عنوان تابعی از تعداد کاربران برای مجموعه‌ی ثابتی از منابع سرور، نشان دهد. گراف را نشان‌گذاری کنید تا نقاط برجسته و جالب را روی «منحنی پاسخ دهد» مشخص کند.

۲۰-۱۷ YourCornerPharmacy.com (مساله ۱۱-۲۰) در پاسخ به موفقیت اش یک سرور انحصاراً برای پیچیدن نسخه‌ها پیاده‌سازی کرده است. به‌طور میانگین، هر دو دقیقه یک بار، ۱۰۰۰ کاربر یک درخواست برای پیچیدن نسخه تسلیم می‌کنند. برنامه‌ی تحت وب در پاسخ به هر درخواست، یک قطعه داده‌ای به طول ۵۰۰ بایت دانلود می‌کند. توان عملیاتی مورد نیاز این سرور، برحسب مگابیت بر ثانیه، تقریباً چقدر است؟

۲۰-۱۸ چه اختلافی میان آزمون ازدحام بار و آزمون فشار وجود دارد؟ کیفیت را در حیطه‌ی یک برنامه‌ی تحت وب و محیط آن چگونه ارزیابی می‌کنیم؟

فصل ۲۱

مدل‌سازی و وارسی رسمی

نگاهی گذرا

مدل‌سازی و وارسی رسمی چیست؟ چند بار این جمله را شنیده‌اید که می‌گوید «کار را همان بار اول درست انجام بده»؟ در ساخت نرم‌افزار اگر این روال را پیش بگیرید، تلاش کمتری صرف دوباره‌کاری بیهوده خواهید کرد. دو روش مهندسی نرم‌افزار پیشرفته- مهندسی نرم‌افزار اتاق تمیز و روش‌های رسمی- با فراهم ساختن یک رویکرد ریاضی-محور برای برنامه‌ریزی مدل‌سازی و توانایی وارسی مدل حاصل، به تیم نرم‌افزاری کمک می‌کنند تا «کار را در همان بار نخست درست انجام دهد.» مهندسی نرم‌افزار اتاق تمیز بر وارسی ریاضی، پیش از شروع ساخت برنامه و بر تأیید قابلیت اطمینان به‌عنوان بخشی از فعالیت آزمون تأکید دارد. در روش‌های رسمی از نظریه مجموعه‌ها و نمادگذاری منطقی برای ایجاد بیان واضحی از حقایق (خواسته‌ها) استفاده می‌شود که می‌توان آن‌ها را برای بهبود بخشیدن به صحت و درستی (و حتی اثبات آن) تحلیل کرد. خط مبنای هر دو روش، ایجاد نرم‌افزاری با مقادیر بسیار پایینِ خطاست.

چه کسی آن را انجام می‌دهد؟ یک مهندس نرم‌افزار که آموزش‌های خاص را در این زمینه دیده باشد.

چرا اهمیت دارد؟ اشتباهات باعث دوباره‌کاری می‌شوند. دوباره‌کاری زمان می‌برد و هزینه‌ها را افزایش می‌دهد. بهتر نیست اگر بتوانیم تعداد اشتباهات (خطاها) را در زمان طراحی و ساخت نرم‌افزار کاهش دهیم؟ مدل‌سازی و وارسی رسمی نوید بخش همین مزیت هستند.

مراحل کار کدام است؟ مدل‌های خواسته‌ها و طراحی با به‌کارگیری یک نمادگذاری تخصصی ایجاد می‌شوند که قابلیت وارسی ریاضی را داشته باشند. مهندسی نرم‌افزار اتاق تمیز، از نمایش ساختارهای چهارگوش استفاده می‌کند که سیستم (یا جنبه‌ای از سیستم) را در سطح مشخصی از انتزاع کپسوله می‌کنند. وارسی، هنگامی به‌کار برده می‌شود که طراحی ساختارهای چهارگوش کامل باشد. هنگامی که صحت برای هر ساختار چهارگوش به تأیید رسید، آزمون کاربرد آماری آغاز می‌شود. روش‌های رسمی، با به‌کارگیری نمادها و مفاهیم نظریه مجموعه‌ها برای تعریف داده‌های ثابت، حالت‌ها و عملیات‌های مربوط به یک وظیفه سیستمی، خواسته‌های نرم‌افزار را به نمایشی رسمی‌تر ترجمه می‌کنند.

محصول کاری چیست؟ مدلیِ تخصص‌یافته و رسمی از خواسته‌ها تهیه می‌شود. نتایج وارسی‌ها و آزمون‌های کاربرد آماری ثبت می‌شود.

چگونه اطمینان حاصل کنم که درست از انجام کار برآمده‌ام؟ اثبات رسمی درستی در مدل خواسته‌ها به‌کار برده می‌شود. آزمون کاربرد آماری، سناریوهای کاربرد را تمرین می‌دهد تا اطمینان حاصل شود که خطاهای موجود در قابلیت عملیاتی کاربر کشف و تصحیح شوند.

برخلاف مرور و آزمون که پس از توسعه‌ی مدل‌ها و کدها آغاز می‌شوند، مدل‌سازی و وارسی رسمی شامل روش‌های مدل‌سازی تخصص یافته‌ای می‌شود که در کنار رویکردهای وارسی تجویزی به‌کار برده می‌شوند. بدون رویکردهای مدل‌سازی مناسب، وارسی را نمی‌توان به انجام رساند.

در این فصل به بحث درباره دو روش مدل‌سازی و وارسی خواهیم پرداخت- *مهندسی نرم‌افزار اتاق تمیز* (clean room) و روش‌های رسمی. هر دو روش، یک رویکرد تخصص یافته را طلب می‌کنند و هر دو از یک روش وارسی منحصر به فرد استفاده می‌کنند. هر دو کاملاً دشوارند و هیچ یک از آن‌ها به‌طور گسترده توسط جامعه مهندسی نرم‌افزار به‌کار گرفته نمی‌شود. ولی اگر قصد دارید یک نرم‌افزار ضد گلوله بسازید، این روش‌ها می‌توانند شما را بی‌اندازه یاری دهند و فراگیری آن‌ها می‌تواند ارزش‌مند باشد.

مهندسی نرم‌افزار اتاق تمیز، رویکری است که بر نیاز به نهادینه‌کردن صحت و درستی در همان زمان توسعه‌ی نرم‌افزار تأکید می‌ورزد. به‌جای چرخه کلاسیک تحلیل، طراحی، کدنویسی، آزمون و اشکال زدایی، در رویکرد اتاق تمیز یک دیدگاه متفاوت [Lin94b] پیشنهاد می‌شود:

> فلسفه‌ای که در پس مهندسی نرم‌افزار اتاق تمیز نهفته است، پرهیز از وابستگی به فرایندهای پرهزینه‌ی حذف نقایص؛ با درست‌نوشتن نسخه‌های افزایشی کد از همان بار نخست و وارسی آن‌ها قبل از آزمون است. مدل فرایندی آن شامل تأیید کیفیت آماری نسخه‌های کد به‌موازات انباشته‌شدن آن‌ها در سیستم می‌شود.

رویکرد اتاق تمیز به طرق بسیار، مهندسی نرم‌افزار را با تأکید ورزیدن بر نیاز به *اثبات* درستی، به سطح دیگر ارتقا می‌دهد.

مدل‌هایی که با به‌کارگیری *روش‌های رسمی* توسعه می‌یابند با استفاده از یک قالب نحوی و معناشناسی رسمی توصیف می‌شوند که عملکرد و رفتار سیستم را توصیف می‌کند. این مشخصات به شکلی ریاضی ارائه می‌شود (مثلاً از جبر گزاره‌ها می‌توان به‌عنوان مبنایی برای زبان مشخصات استفاده کرد). آنتونی هال [Hal90] در کتاب مقدماتی خود در باب روش‌های رسمی توضیحی می‌دهد که برای روش‌های اتاق تمیز نیز به همان میزان کاربرد دارد:

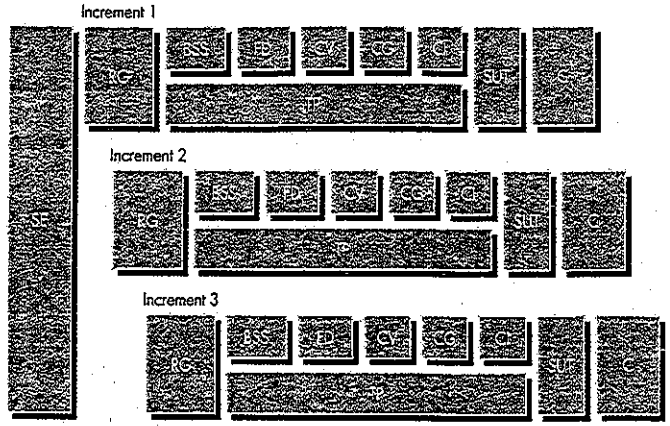
> روش‌های رسمی [و مهندسی نرم‌افزار اتاق تمیز] بحث برانگیزند. مدافعان این روش‌ها ادعا می‌کنند که می‌توانند [نرم‌افزار] را متحول سازند و مخالفان این روش‌ها معتقدند که بسیار دشوار و غیر ممکن هستند. در ضمن، برای اکثر افراد، روش‌های رسمی [و مهندسی نرم‌افزار اتاق تمیز] چنان ناآشنا هستند که قضاوت درباره این ادعاهای رقیب، دشوار است.

در این فصل به بررسی روش‌های وارسی و مدل‌سازی رسمی و نیز بررسی تأثیر بالقوه‌ی آن‌ها بر مهندسی نرم‌افزار در سال‌های آینده خواهیم پرداخت.

۲۱-۱ راهبرد اتاق تمیز (Clean Room Strategy)

مهندسی نرم‌افزار اتاق تمیز از یک نسخه‌ی تخصص یافته از مدل نرم‌افزار افزایشی است که در فصل ۲ معرفی شد. چند تیم نرم‌افزاری کوچک و مستقل، «لوله‌ای از نسخه‌های نرم‌افزار» [Lin94b] را توسعه می‌دهند. هر نسخه پس از این که به تأیید رسید به کل سیستم افزوده می‌شود. از این رو، قابلیت عملیاتی سیستم با گذر زمان رشد می‌کند.

> «تنها راه رخ دادن خطا در یک برنامه این است که نویسنده‌ی برنامه، آن را باعث شود. هیچ سازوکار دیگری شناخته نشده است... کار درست این است که از درج خطاها جلوگیری شود و اگر این هم نشد، آن‌ها را قبل از آزمون یا هر گونه اجرای دیگر برنامه حذف کنیم.»
>
> هارلان میلز



SE — مهندسی سیستم
RG — جمع‌آوری خواسته‌ها
BSS — مشخصه‌ی ساختار چهارگوش
FD — طراحی رسمی
CV — وارسی درستی
CG — تولید کد
CI — بازرسی کد
SUT — آزمون کاربرد آماری
C — صدورگواهی
TP — برنامه‌ریزی آزمون

شکل ۱-۲۱ مدل فرایندی اتاق تمیز.

وظایف اتاق تمیز برای هر نسخه در شکل ۱-۲۱ نشان داده شده است. در داخل لوله‌ی نسخه‌های اتاق تمیز، وظایف زیر باید به انجام برسد:

برنامه‌ریزی برای نسخه‌ها. یک طرح پروژه تهیه می‌شود که راهبرد افزایشی در آن انتخاب می‌شود. قابلیت عملیاتی هر نسخه، اندازه پیش بینی شده برای آن و یک زمان‌بندی برای توسعه‌ی اتاق تمیز ایجاد می‌شود. مراقبت ویژه‌ای باید به عمل آید تا اطمینان حاصل شود که نسخه‌های تأییدشده، سروقت، منسجم می‌شوند.

جمع‌آوری خواسته‌ها. با استفاده از تکنیک‌هایی شبیه به آن‌چه که در فصل ۵ معرفی شد، توصیف مشروح تری از خواسته‌ها در سطح مشتری (برای هر کدام از نسخه‌ها) تهیه می‌شود.

مشخصه‌ی ساختار چهارگوش. یک روش تعیین مشخصات که در آن از ساختارهای چهارگوش برای توصیف مشخصات عملیاتی استفاده می‌شود. ساختارهای چهارگوش «تعریف خلاقانه‌ی رفتار، داده‌ها و روال‌های موجود در هر سطح از پالایش را جداسازی و تفکیک می‌کنند» [Hev93].

طراحی رسمی. با استفاده از رویکرد ساختارهای چهارگوش، طراحی اتاق تمیز بسطی طبیعی و یکپارچه از مشخصات است. گرچه می‌توان میان این دو فعالیت، تمایز قائل شد، مشخصات (که چهارگوش سیاه نامیده می‌شوند) طی چند دور تکرار (در یک نسخه) پالایش می‌شود تا به طراحی‌های معماری یا در سطح مؤلفه‌ها (موسوم به چهارگوش‌های حالت و چهارگوش‌های شفاف) شباهت پیدا کند.

وارسی. تیم اتاق تمیز یک سری فعالیت‌های شدید وارسی را روی طراحی و سپس روی کدها اجرا می‌کند. وارسی (بخش ۲-۳-۲۱) با بالاترین سطح از ساختارهای چهارگوش (مشخصات) آغاز می‌شود و به سوی جزئیات طراحی و کد حرکت می‌کند. نخستین سطح از وارسی

مرجع وب

یک منبع عالی از اطلاعات و منابعی برای مهندسی نرم‌افزار اتاق تمیز را می‌توانید در www.cleansoft.com بیابید.

با به‌کارگیری مجموعه‌ای از «پرسش‌های صحت» آغاز می‌شود [Lin88]. اگر این‌ها نشان ندهد که مشخصات درست است، روش‌های رسمی‌تر (ریاضی‌تر) برای وارسی به‌کارگرفته خواهدشد.

تولید، بازرسی و وارسی کدها. مشخصات ساختارهای چهارگوش که به زبانی تخصص‌یافته ارائه می‌شوند، به زبان برنامه‌نویسی مناسب ترجمه می‌شوند. سپس از مرورهای فنی (فصل ۱۵) برای حصول اطمینان از مطابقت با ساختارهای چهارگوش و کدها و صحت نحوی کدها استفاده می‌شود. پس از آن، وارسی برای کد منبع انجام می‌شود.

برنامه‌ریزی برای آزمون‌های آماری. کاربرد پیش بینی شده برای نرم‌افزار تحلیل می‌شود و مجموعه‌ای از موارد آزمون، برنامه‌ریزی و طراحی می‌شود تا «توزیع احتمالی» از این کاربرد را تمرین دهد (بخش ۴-۲۱). همان طور که از شکل ۱-۲۱ پیداست، این فعالیت اتاق تمیز به موازات تعیین مشخصات، وارسی و تولید کد اجرا می‌شود.

آزمون کاربرد آماری. با یادآوری این نکته که آزمون کامل و فراگیر نرم‌افزار امری غیر ممکن است (فصل ۱۸)، همواره لازم است چند مورد آزمون متناهی طراحی شود. در تکنیک‌های کاربرد آماری [Poo88] یک سری آزمون اجرا می‌شود که از یک نمونه آماری (توزیع احتمال ذکر شده در قبل) از کلیه‌ی اجراهای ممکن برنامه توسط کلیه کاربران جمعیت هدف‌مند (بخش ۴-۲۱) به‌دست می‌آید.

صدور گواهی. هنگامی که وارسی، بازرسی و آزمون کاربرد به انجام رسید (و همه‌ی خطاها تصحیح شد)، آماده‌بودن نسخه برای انسجام یافتن به سیستم به تأیید می‌رسد.

چهار فعالیت نخست در فرایند اتاق تمیز، صحنه را برای وارسی رسمی زیر آماده می‌کند. به همین دلیل، بحث مربوط به رویکرد اتاق تمیز را با فعالیت‌های مدل‌سازی آغاز می‌کنیم که برای انجام وارسی رسمی ضروری اند.

> «مهندسی نرم‌افزار اتاق تمیز، کنترل کیفیت آماری برای توسعه نرم‌افزار را از طریق جداسازی اکید فرایند طراحی از فرایند آزمون در خط لوله‌ای از توسعه‌ی نرم‌افزار میسر می‌سازد.»
>
> هارلان میلز

> **اندرز**
>
> در اتاق تمیز، بر آزمون‌هایی تأکید می‌گردد که نرم‌افزار را به همان گونه که قرار است از آن استفاده شود، تمرین می‌دهند. ورودی فرایند برنامه‌ریزی برای آزمون را use case‌ها تشکیل می‌دهند.

> «از نکات جالب زندگی این است که اگر از پذیرفتن هر چیزی مگر بهترین آن سرباز بزنید، معمولاً بهترین‌ها نصیبتان خواهد شد.»
>
> سامرست موام

> پالایش چگونه به‌عنوان بخشی از مشخصات ساختار چهارگوش عملی می‌شود؟

۲-۲۱ مشخصات علمیاتی

در رویکرد مدل‌سازی در مهندسی نرم‌افزار اتاق تمیز، از روشی موسوم به *تعیین مشخصات ساختارهای چهارگوش* استفاده می‌شود. یک «چهارگوش»، سیستم (یا جنبه‌ای از سیستم) را در سطحی از جزئیات پنهان‌سازی می‌کند. از طریق فرایند پالایش مرحله به مرحله و پرداختن به جزئیات، چهارگوش‌ها به‌صورت یک سلسله مراتب پالایش می‌شوند که در آن هر چهارگوش دارای شفافیت ارجاعی است. یعنی، «محتوای اطلاعاتی هر مشخصه برای تعریف پالایش آن کفایت می‌کند، بدون این که به پیاده‌سازی هیچ جعبه دیگری وابسته باشد» [Lin94b]. به این ترتیب، تحلیل‌گر می‌تواند سیستم را به‌صورت سلسله مراتبی با حرکت از نمایش اساسی در بالا تا جزئیات خاص پیاده‌سازی در پایین، افراز کند. از سه نوع چهارگوش استفاده می‌شود:

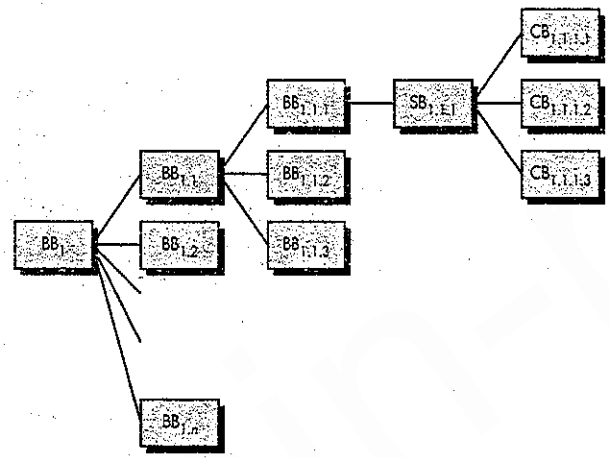
چهارگوش سیاه (Black Box). چهارگوش سیاه، رفتار یک سیستم یا بخشی از یک سیستم را مشخص می‌کند. سیستم (یا بخشی از آن) با به‌کارگیری مجموعه‌ای از قواعد انتقال که محرک را به پاسخ تبدیل می‌کنند به محرک‌ها (رویدادها) پاسخ می‌دهند.

چهارگوش حالت (State Box). چهارگوش حالت، سرویس‌ها (عملیات‌ها) و داده‌های حالت را به شیوه‌ای مشابه با اشیا کپسوله می‌کنند. در این نما از تعیین مشخصات، ورودی‌های چهارگوش

حالت (محرک‌ها) و خروجی‌ها (پاسخ‌ها) عرضه می‌شوند. چهارگوش حالت، «تاریخچه‌ی محرک» چهارگوش سیاه را نیز نشان می‌دهد، یعنی داده‌های کپسوله شده در چهارگوش حالت که باید بین گذارهای موجود حفظ شوند.

چهارگوش شفاف (Clear Box). توابع گذاری (transition functions) که توسط چهارگوش حالت مشخص می‌شوند، در چهارگوش شفاف تعریف می‌شوند. به بیان ساده، یک چهارگوش شفاف حاوی طراحی روالی برای چهارگوش حالت است.

در شکل ۲-۲۱ رویکرد پالایش با استفاده از تعیین مشخصات ساختارهای چهارگوش نشان داده شده است. یک چهارگوش سیاه (BB_i) پاسخ مربوط به مجموعه کاملی از محرک‌ها را تعریف می‌کند. BB_i را می‌توان به مجموعه‌ای از سه چهارگوش $BB_{i.1}$ تا $BB_{i.n}$ پالایش نمود که هر کدام به یک کلاس رفتار مربوط می‌شود. پالایش چندان ادامه می‌یابد که کلاس یکپارچه‌ای از رفتار (مثلاً $BB_{i.1.1}$) تعریف شود. سپس یک چهارگوش حالت ($SB_{i.1.1}$) برای چهارگوش سیاه ($BB_{i.1.1}$) تعریف می‌شود. در این مورد، $SB_{i.1.1}$ حاوی تمامی داده‌ها و سرویس‌های مورد نیاز برای پیاده‌سازی رفتار تعریف شده توسط $BB_{i.1.1}$ است. سرانجام، $SB_{i.1.1}$ به چهارگوش‌های شفاف ($CB_{i.1.1.n}$) پالایش می‌شود و جزئیات طراحی روالی مشخص می‌شوند.



شکل ۲-۲۱ پالایش ساختار چهارگوش.

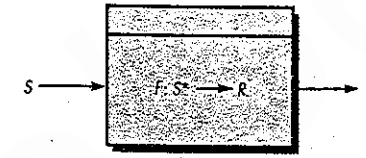
با رخ دادن هر کدام از این مراحل پالایش، وارسی نیز انجام می‌شود. مشخصات چهارگوش حالت، وارسی می‌شوند تا اطمینان حاصل شود که هر کدام با رفتار تعریف شده توسط مشخصه‌ی چهارگوش سیاه مادر مطابقت دارند. به‌طور مشابه، مشخصات چهارگوش شفاف در برابر چهارگوش حالت والد وارسی می‌شود.

> **نکته‌ی کلیدی**
>
> پالایش ساختارهای چهارگوش و وارسی، همزمان انجام می‌شود.

۱-۲-۲۱ مشخصات چهارگوش سیاه

مشخصات *چهارگوش سیاه*، توصیف‌گر انتزاع، محرک‌ها و پاسخ با استفاده از نمادگذاری نشان داده در شکل ۳-۲۱ است [Mil88]. تابع *ارروی سری* S^* از ورودی‌ها (محرک‌های) S اعمال می‌شود و آن‌ها

را به خروجی (پاسخ) R تبدیل می‌کند. برای مؤلفه‌های یک نرم‌افزار ساده، f ممکن است یک تابع ریاضی باشد، ولی در کل، f یا به‌کارگیری زبان طبیعی (یک زبان رسمی برای تعیین مشخصات) توصیف می‌شود.

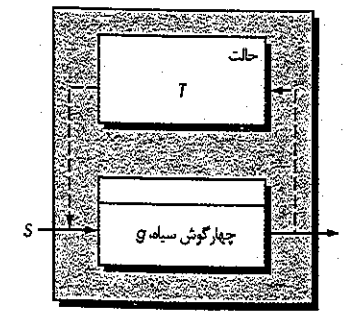


شکل ۳-۲۱ یک مشخصه‌ی چهارگوش سیاه.

بسیاری از مفاهیم معرفی شده برای سیستم‌های شیء گرا برای چهارگوش سیاه نیز مصداق دارند. انتزاع‌های داده‌ای و علمیات‌هایی که آن انتزاع‌ها را دستکاری می‌کنند، توسط چهارگوش سیاه، کپسوله می‌شوند. مشخصه‌ی چهارگوش سیاه، همانند سلسله مراتب کلاس‌ها می‌توانند سلسله مراتب‌های کاربردی را به نمایش بگذارند که در آن‌ها، چهارگوش‌های سطح پایین خواص آن دسته از چهارگوش‌هایی را به ارث می‌برند که در ساختار درختی در سطوح بالاتر قرار دارند.

۲-۲-۲۱ مشخصات چهارگوش حالت

چهارگوش حالت «تعمیم ساده‌ای از یک ماشین حالت است» [Mil88]. با به خاطر آوردن بحث مدل‌سازی رفتاری و نمودارهای حالت در فصل ۷، هر حالت یک شیوه‌ی مشاهده‌پذیر از رفتار سیستم است. با رخ دادن پردازش، سیستم به رویدادها (محرک ها) با انجام گذار از حالت فعلی به حالت جدید پاسخ می‌دهد. با انجام این گذار، ممکن است کنشی رخ دهد. چهارگوش حالت از یک انتزاع داده‌ای برای تعیین‌گذار به حالت بعدی واکنش (پاسخی) استفاده می‌کندکه در نتیجه‌ی‌گذار رخ می‌دهد.



شکل ۴-۲۱ یک مشخصه‌ی چهارگوش حالت.

با رجوع به شکل ۴-۲۱ مشاهده می‌شود که چهارگوش حالت شامل چهارگوش سیاه g می‌شود. محرک S که ورودی چهارگوش سیاه است از یک منبع بیرونی و یک مجموعه حالت‌های درونی سیستم T ناشی می‌شود. میلز [Mil88] توصیفی ریاضی از تابع f چهارگوش سیاه موجود در چهارگوش حالت فراهم ساخته است.

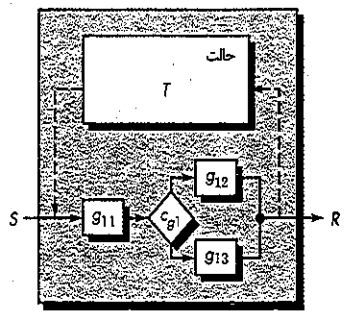
$$g : S^* \times T^* \rightarrow R \times T$$

که g زیرتابعی مرتبط با حالت خاص t است. جفت‌های زیرتابع (t,g) اگر یک‌جا در نظر گرفته شوند، تابع f چهارگوش سیاه را تعریف می‌کنند.

۳-۲-۲۱ مشخصه‌ی چهارگوش شفاف

مشخصات چهارگوش شفاف، همسویی تنگاتنگی با طراحی روالی و برنامه‌نویسی ساخت‌یافته دارد. در اصل، زیر تابع g در چهارگوش حالت، جای خود را به ساختارهای برنامه‌نویسی ساخت‌یافته‌ای می‌دهد که g را پیاده‌سازی می‌کنند.

به‌عنوان مثال، چهارگوش شفاف نشان داده شده در شکل ۵-۲۱ را در نظر بگیرید. چهارگوش سیاه g، که در شکل ۳-۲۱ نشان داده شد، جای خود را به یک ساختار ترتیبی می‌دهد که شامل یک ساختار شرطی می‌شود. این‌ها نیز به نوبه خود و در ادامه‌ی پالایش مرحله‌ای به چهارگوش‌هایی با سطح پایین‌تر قابل پالایش هستند.



شکل۵-۲۱ یک مشخصه‌ی چهارگوش شفاف.

درستی مشخصه‌ی روالی توصیف شده در این سلسله مراتب چهارگوش‌های شفاف را می‌توان اثبات کرد. به این مبحث در بخش ۳-۲۱ خواهیم پرداخت.

۳-۲۱ طراحی اتاق تمیز (Clean Room Design)

در مهندسی نرم‌افزار اتاق تمیز، از فلسفه‌ی برنامه‌نویسی ساخت‌یافته، استفاده‌ی گسترده به‌عمل می‌آید (فصل ۱۰). ولی در این مورد، برنامه‌نویسی ساخت‌یافته بسیار شدیدتر استفاده می‌شود.

توابع پردازشی پایه (که طی پالایش‌های اولیه مشخصات توصیف می‌شوند) با استفاده از یک «روش بسط مرحله‌ای توابع ریاضی به ساختارهای زیرتابع‌ها و ارتباط‌های منطقی [مانند if-then-else]، پالایش می‌شوند که در آن، این بسط، آن‌قدر ادامه می‌یابد تا همه‌ی زیرتابع‌های شناسایی شده را بتوان مستقیماً به زبان برنامه‌نویسی به‌کار رفته برای پیاده‌سازی بیان کرد.» [Dye92]

از رویکرد برنامه‌نویسی ساخت‌یافته می‌توان به‌طور اثربخش برای پالایش تابع استفاده کرد، ولی درباره طراحی داده‌ها چطور؟ در این‌جا، تعدادی از مفاهیم بنیادی طراحی (فصل ۸) وارد عمل می‌شوند. داده‌های برنامه به‌صورت مجموعه‌ای از انتزاع‌ها کپسوله می‌شوند که زیرتابع‌ها به آن‌ها

سرویس می‌دهند. در ایجاد طراحی داده‌ها، از مفاهیم کپسوله‌سازی داده‌ها، پنهان‌سازی اطلاعات و تعیین نوع داده‌ها استفاده می‌شود.

۱-۳-۲۱ پالایش طراحی

هر مشخصه‌ی چهارگوش شفاف، نشانگر طراحی یک روال (زیرتابع) لازم برای محقق ساختن یک گذار چهارگوش حالت است. در داخل یک چهارگوش شفاف، از ساختارهای برنامه‌نویسی ساخت‌یافته و پالایش مرحله‌ای برای پایش جزئیات روالی استفاده می‌شود. برای مثال، تابع f به یک سری زیرتابع‌های g و h پالایش می‌شود. این زیر تابع‌ها نیز به‌نوبه‌ی خود ساختارهای شرطی (مانند if-then-else و do-while) پالایش می‌شوند. پالایش باز هم ادامه می‌یابد تا این که جزئیات روالی کافی برای ایجاد مؤلفه‌ی مورد نظر موجود باشد.

تیم اتاق تمیز[1] در هر سطح از پالایش، یک **وارسی صحت رسمی** انجام می‌دهد. برای دستیابی به این منظور، مجموعه‌ای از شرایط عمومی صحت، به ساختارهای برنامه‌نویسی ساخت‌یافته متصل می‌شوند. اگر تابع h به یک سری g و h بسط داده شود، شرط صحت برای همه‌ی ورودی‌های h عبارت است از:

- آیا g و پس از آن h، f را انجام می‌دهند؟

هنگامی که p به یک ساختار شرطی به شکل if <c> then q, else r پالایش شود، شرط صحت برای کلیه ورودی‌های p عبارت است از:

- هر گاه <c> درست باشد، آیا q p را انجام می‌دهد؛ و هرگاه <c> نادرست باشد، آیا r p را انجام می‌دهد؟

هنگامی که تابع m به‌صورت یک حلقه پالایش شود، شرط‌های درستی برای همه‌ی ورودی‌های m عبارتنداز:

- آیا پایان یافتن حلقه تضمین شده است؟
- هرگاه <c> درست باشد، آیا n که پس از آن m می‌آید، m را انجام می‌دهد؛ و هرگاه <c> نادرست باشد، آیا با جا انداختن حلقه، هنوز m انجام می‌شود؟

هر بار که یک چهارگوش شفاف به سطح بعدی جزئیات پالایش شود، این شرط‌های درستی اعمال می‌شوند.

برای اثبات درستی ساختارهای ساخت‌یافته چه شرط‌هایی به کار برده می‌شود؟

اندرز

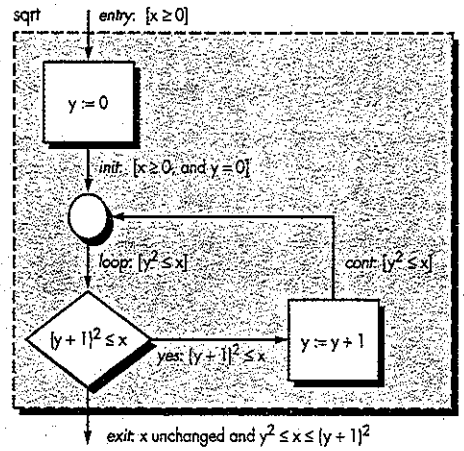
اگر در توسعه‌ی طراحی روالی، فقط خودتان را به ساختارهای ساخت‌یافته محدود کنید، اثبات درستی، کاری صریح خواهد بود. اگر از این ساختارها عدول کنید، اثبات درستی ممکن است دشوار یا حتی غیرممکن شود.

۲-۳-۲۱ وارسی طراحی

باید توجه داشته باشید که استفاده از ساختارهای برنامه‌نویسی ساخت‌یافته، تعداد آزمون‌های درستی را که باید اجرا شود، محدود می‌کند. برای ساختارهای ترتیبی یک شرط؛ دو شرط برای if-then-else و سه شرط برای حلقه‌ها چک می‌شود.

برای نشان دادن وارسی برای یک طراحی روالی از مثال ساده‌ای استفاده می‌کنیم که نخستین بار توسط لینگر، میلز و ویت [Lin79] معرفی شده است. هدف، طراحی و وارسی برنامه کوچکی است که

[1] چون کل تیم در فرایند وارسی شرکت دارد، این احتمال وجود دارد که در اجرای خود وارسی هم خطایی رخ دهد.



شکل۶-۲۱ محاسبه جزء صحیح جذر یک عدد [Lin94].

۷، جزء صحیح جذر یک عدد صحیح x را می‌دهد. طراحی روالی با استفاده از نمودار گردشی شکل ۶-۲۱ نشان داده شده است.[1]

برای وارسی صحت طراحی، شرط‌های ورودی و خروجی به‌صورت نشان داده شده در شکل ۶-۲۱ افزوده می‌شوند. شرط ورودی خاطر نشان می‌سازد که x باید بزرگ‌تر یا مساوی صفر باشد. شرط خروجی ایجاب می‌کند که x تغییر نکند و y در عبارت ذکر شده در شکل صدق کند. برای اثبات صحت طراحی، لازم است شرط‌های *init*، *loop*، *cont*، *yes* و *exit* که در شکل ۶-۲۱ نشان داده شده‌اند، در تمامی حالت‌ها اثبات شوند. این اثبات‌ها را گاهی *اثبات فرعی* می‌نامند.

۱. شرط *init* مستلزم آن است که $[x \geq 0 \text{ and } y = 0]$. بر اساس خواسته‌های مسأله، شرط ورودی، درست فرض می‌شود[2]. بنابراین، نخستین بخش از شرط *init* $x \geq 0$، بر قرار است. با رجوع به نمودار گردشی مشاهده می‌شود صحت گزاره‌ای که بلافاصله قبل از شرط *init* قرار دارد، y را برابر با 0 قرار می‌دهد. بنابراین، بخش دوم شرط *init* نیز برقرار است. پس *init* درست است.

۲. شرط *loop* ممکن است به دو صورت مشاهده شود: (۱) مستقیماً از *init* (در این مورد، شرط *loop* به‌طور مستقیم برقرار می‌شود) یا از طریق کنترل جریاتی که از میان شرط *cont* می‌گذرد. چون شرط *cont* هم‌ارز با شرط *loop* است، *loop* درست است و این ربطی به مسیر جریان منتهی به آن ندارد.

۳. شرط *cont* تنها پس از افزایش مقدار y به میزان یک واحد، مشاهده می‌شود. به‌علاوه، مسیر جریان کنترلی که به *cont* منتهی می‌شود تنها در صورتی قابل فراخوانی است که شرط *yes* نیز درست باشد. از این رو، اگر $x \geq (y+1)^2$، لازم می‌آید که $y^2 \leq x$. شرط *cont* برقرار است.

نکته‌ی کلیدی

برای اثبات درستی طراحی، ابتدا باید همه‌ی شرایط را معین کرده سپس ثابت کنید که هر کدام، ارزش بولی مناسب را به خود می‌گیرد. به این‌ها اثبات فرعی گفته می‌شود.

[1] شکل ۶-۲۱ از [Lin94]، و با کسب اجازه، برگرفته شده است.

[2] مقدار منفی برای جذر در این حیطه بی‌معنی است.

۴. شرط *yes* در منطق شرطی نشان داده شده آزموده می‌شود. در این جا، شرط *yes* باید هنگامی درست باشد که جریان کنترل در راستای مسیر نشان داده شده حرکت می‌کند.

۵. شرط *exit* مستلزم آن است که x بدون تغییر باقی بماند. بررسی این طراحی نشان می‌دهد که x در هیچ جا در طرف چپ یک عملگر انتساب ظاهر نمی‌شود. در هیچ فراخوانی تابعی از x استفاده نمی‌شود. لذا، x بدون تغییر باقی می‌ماند. چون آزمون شرطی $x \leq y^2$ $(y+1)^2$ باید نادرست باشد تا به شرط *exit* برقرار شود، لازم می‌آید که $x \leq (y+1)^2$. به علاوه، شرط *loop* هنوز باید درست باشد (یعنی $y^2 \leq x$). بنابراین، برای برقرار شدن شرط exit می‌توان $x < (y+1)^2$ و $y^2 \leq x$ را با هم تلفیق نمود. به علاوه باید اطمینان حاصل کنید که حلقه به پایان می‌رسد. با بررسی شرط *loop* در می‌یابیم که چون y زیاد می‌شود و $x \geq 0$، حلقه باید سرانجام پایان بگیرد.

پنج مرحله‌ای که در بالا ذکر شدند، صحت طراحی الگوریتم شکل ۶-۲۱ را اثبات می‌کنند. اکنون مطمئن هستید که این طراحی واقعاً جزء صحیح جذر یک عدد صحیح را محاسبه می‌کند.

یک روش ریاضی دشوارتر برای وارسی صحت طراحی‌ها، امکان‌پذیر است. ولی بحث درباره این موضوع از حوصله‌ی این کتاب خارج است و در صورت علاقه می‌توانید به [Lin79] مراجعه کنید.

۲۱-۴ آزمون اتاق تمیز

راهبرد و تاکتیک‌های آزمون اتاق تمیز با رویکردهای سنتی آزمون (فصل‌های ۱۷ تا ۲۰) تفاوت بنیادی دارند. در روش‌های سنتی یک مجموعه موارد آزمون به‌دست می‌آید که خطاهای طراحی و کدنویسی را کشف می‌کنند. هدف آزمون اتاق تمیز، اعتبارسنجی خواسته‌های نرم‌افزار است و برای این منظور نشان می‌دهد که یک نمونه آماری از موارد آزمون (فصل ۵) با موفقیت اجرا شده است.

۲۱-۴-۱ آزمون کاربرد آماری (Statistical Use Testing)

کاربر یک برنامه کامپیوتری، به‌ندرت نیاز پیدا می‌کند تا جزئیات فنی طراحی را بداند. رفتار برنامه که به چشم کاربر می‌آید، به وسیله ورودی‌ها و رویدادهایی تعیین می‌شوند که غالباً توسط کاربر تولید می‌شوند. ولی در سیستم‌های پیچیده، طیف ورودی‌ها و رویدادهای ممکن (یعنی use case) می‌تواند بی‌اندازه گسترده باشد. چه زیرمجموعه‌ای از use case ها به‌طور مناسب، رفتار برنامه را وارسی می‌کند؟ این نخستین پرسشی است که آزمون کاربرد آماری باید به آن بپردازد.

آزمون کاربرد آماری «در کل عبارت است از آزمودن نرم‌افزار به شیوه‌ای که کاربران تمایل به استفاده از آن دارند» [Lin94b]. تیم‌های آزمون اتاق تمیز (که *تیم‌های تأیید* نیز نامیده می‌شوند) برای نیل به این مقصود، باید یک توزیع احتمال کاربرد برای نرم‌افزار تعیین کنند. مشخصات (چهارگوش سیاه) برای هر نسخه از نرم‌افزار تحلیل می‌شود تا مجموعه‌ای از محرک‌ها (رویدادها یا ورودی ها) که باعث تغییر رفتار نرم‌افزار می‌شوند، تعریف شوند. بر اساس مصاحباتی که با کاربران بالقوه به عمل می‌آید، ایجاد سناریوهای کاربرد و درک کلی از دامنه‌ی کاربرد، یک احتمال کاربرد به هر محرک نسبت داده می‌شود.

> «کیفیت، نه یک عمل بلکه عادت است.»
>
> ارسطو

> **اندرز**
>
> حتی اگر تصمیم گرفتید که از رویکرد اتاق تمیز استفاده نکنید، خوب است آزمون کاربرد آماری را به‌عنوان بخشی جدایی‌ناپذیری از راهبرد آزمون خود در نظر بگیرید.

برای هر مجموعه از محرک‌ها[1] مطابق با توزیع احتمال کاربرد، موارد آزمونی ایجاد می‌شود. برای روشن شدن مطلب، سیستم *SafeHome* را، که قبلاً در همین کتاب بحث شده است، در نظر بگیرید. از مهندسی نرم‌افزار اتاق تمیز برای توسعه‌ی نسخه‌ای از نرم‌افزار استفاده می‌شود که تعامل کاربر با صفحه کلید سیستم امنیتی را مدیریت می‌کند. برای این نسخه‌ی افزایشی از نرم‌افزار پنج محرک شناسایی شده است. تحلیل، درصد توزیع احتمال هر محرک را نشان می‌دهد. برای آسان‌ترکردن امر انتخاب موارد آزمون، این احتمال‌ها روی یک بازه عددی ۱ تا ۹۹ نگاشت می‌شوند [Lin94] و در جدول زیر به نمایش در می‌آیند:

محرک برنامه	احتمال	بازه
فعال/غیر فعال (AD)	٪۵۰	۱-۴۹
مجموعه نواحی (ZS)	٪۱۵	۵۰-۶۳
درخواست (Q)	٪۱۵	۶۴-۷۸
آزمون (T)	٪۱۵	۷۹-۹۴
هشدار	٪۵	۹۵-۹۹

برای تولید یک سری موارد آزمون کاربردی که با توزیع احتمال کاربرد همخوانی داشته باشد، اعداد تصادفی میان ۱ و ۹۹ تولید می‌شوند. هر عدد تصادفی متناظر با یک بازه روی توزیع احتمال قبلی است. از این رو، سری موارد آزمون کاربردی به‌طور تصادفی تعریف می‌شود، ولی با احتمال مناسبی از رخداد محرک، متناظر است. برای مثال، فرض کنید که سری اعداد تصادفی زیر تولید می‌شود:

۱۳-۹۴-۲۲-۲۴-۴۵-۵۶

۸۱-۱۹-۳۱-۶۹-۴۵-۹

۳۸-۲۱-۵۲-۸۴-۸۶-۴

با انتخاب محرک‌های مناسب روی بازه توزیع نشان داده شده در جدول، موارد آزمون زیر به‌دست می‌آید:

AD-T-AD-AD-AD-ZS
T-AD-AD-AD-Q-AD-AD
AD-AD-ZS-T-T-AD

تیم آزمون این موارد آزمون را اجرا می‌کند و رفتار نرم‌افزار را در برابر مشخصات سیستم وارسی می‌کند. زمان‌بندی برای موارد آزمون طوری ثبت می‌شود که بازه‌های زمانی تعیین شوند. با استفاده از بازه‌های زمانی، تیم تأیید می‌تواند میانگین زمان شکست را محاسبه کند. اگر یک سری طولانی از آزمون‌ها بدون شکست اجرا شود، MTTF پایین بوده قابلیت اطمینان نرم‌افزار را می‌توان بالا انگاشت.

[1] برای این منظور می‌توان از ابزارهای خودکار بهره برد. برای اطلاعات بیشتر [Dye99] را ببینید.

۲۱-۴-۲ صدور گواهی (Certification)

تکنیک‌های وارسی و آزمون که قبلاً در این فصل بحث شدند به مؤلفه‌های نرم‌افزار (و نسخه‌های کاملی) می‌انجامند که می‌توان بر آن‌ها مهر تأیید زد. در حیطه‌ی مهندسی نرم‌افزار اتاق تمیز، تأیید به این معناست که قابلیت اطمینان را [که توسط زمان میانگین شکست (MTTF) سنجیده می‌شود] برای هر یک از مؤلفه‌ها می‌توان مشخص کرد.

تأثیر بالقوه‌ی مؤلفه‌های نرم‌افزار قابل تأیید، فراتر از یک پروژه اتاق تمیز است. مؤلفه‌های نرم‌افزار قابل استفاده‌ی مجدد را می‌توان همراه با سناریوهای کاربرد آن‌ها، محرک‌های برنامه و توزیع‌های احتمال نگهداری کرد. هر مؤلفه تحت سناریوی کاربرد و رژیم آزمون توصیف شده دارای یک قابلیت اطمینان تأیید شده است. این اطلاعات برای سایر افرادی که تمایل به استفاده از مؤلفه‌ها دارند، بی‌اندازه ارزش‌مند است.

رویکرد تأیید شامل پنج مرحله می‌شود [Woh94]: (۱) سناریوهای کاربرد باید ایجاد شوند، (۲) پروفایل کاربرد مشخص می‌شود، (۳) موارد آزمون از روی پروفایل ایجاد می‌شوند، (۴) آزمون‌ها اجرا و داده‌های مربوط به شکست‌ها، ثبت و تحلیل می‌شوند و (۵) قابلیت اطمینان، محاسبه و تأیید می‌شود. مراحل ۱ تا ۴ را در بخش قبل مورد بحث قرار دادیم. تأیید برای مهندسی نرم‌افزار اتاق تمیز به ایجاد سه مدل نیاز دارد [Poo93]:

یک مؤلفه نرم‌افزار را چگونه تأیید می‌کنید؟

مدل نمونه‌برداری. آزمون نرم‌افزار، m مورد آزمون تصادفی را اجرا می‌کند و اگر هیچ شکستی مشاهده نشود یا تعداد مشخصی از خطا رخ دهد، تأیید انجام می‌شود. مقدار m به روش ریاضی به‌دست می‌آید تا اطمینان حاصل شود که قابلیت اطمینان لازم وجود دارد.

مدل مؤلفه‌ها. سیستمی متشکل از n مؤلفه باید تأیید شود. مدل مؤلفه‌ها به تحلیل‌گر این امکان را می‌دهد تا احتمال شکست مؤلفه‌ی i قبل از کامل شدن سیستم را تعیین کند.

مدل تأیید. قابلیت اطمینان کل سیستم، پیش بینی می‌شود و به تأیید می‌رسد.

با کامل شدن آزمون کاربرد آماری، تیم تأیید دارای اطلاعات لازم برای تحویل نرم‌افزاری است که دارای MTTF تأیید شده و محاسبه شده با به‌کارگیری هر کدام از این مدل‌هاست. در صورت علاقه‌ی بیشتر، [Cur86]، [Mus87] یا [Poo93] را ببینید.

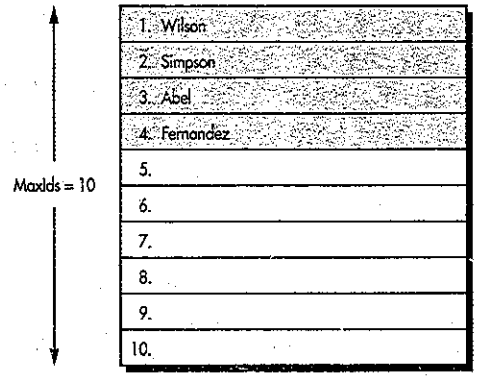
۲۱-۵ مفاهیم روش‌های رسمی

در *فرهنگ بزرگ مهندسی نرم‌افزار* [Mar01]، روش‌های رسمی به شیوه زیر تعریف می‌شوند:

> روش‌های رسمی به‌کار رفته در توسعه‌ی سیستم‌های کامپیوتری، تکنیک‌هایی با اساس و پایه ریاضی برای توصیف خواص سیستم هستند. این روش‌های رسمی، چارچوب‌هایی فراهم می‌آورند که از طریق آن‌ها می‌توان سیستم‌ها را به شیوه‌ای سیستماتیک و پیش بینی شده، مشخص کرد، توسعه داد و وارسی کرد.

خواص مطلوب یک مشخصه‌ی رسمی- سازگاری، کامل بودن و فقدان ابهام- اهداف همه‌ی روش‌های تعیین مشخصات هستند. به هر حال، زبان به‌کار رفته برای تعیین مشخصات در روش‌های رسمی که پایه‌ای ریاضی دارد، احتمال دستیابی به این خواص را به مراتب افزایش می‌دهد. قالب نحوی رسمی یک زبان تعیین مشخصات (بخش ۷-۲۱) تفسیر خواسته‌ها یا طراحی را تنها در یک

جهت میسر می‌سازد یعنی در جهت حذف ابهامی که غالباً هنگام تفسیر یک زبان طبیعی (مثلاً فارسی) یا نمادگذاری گرافیکی (مثلاً UML) توسط خواننده رخ می‌دهد. به کمک امکانات توصیفی نظریه مجموعه‌ها و نمادگذاری منطقی، می‌توان خواسته‌ها را به وضوح و روشنی بیان کرد. برای سازگاری، خواسته‌های بیان شده در یک نقطه از مشخصات نباید با خواسته‌هایی در جای دیگر در تناقض باشند. سازگاری[1] با اثبات ریاضی این نکته محقق می‌شود که حقایق اولیه را می‌توان به‌طور رسمی (با استفاده از قواعد استنباط) در گزاره‌های بعدی موجود در مشخصات، تصویر کرد. برای معرفی مفاهیم روش‌های رسمی، به بررسی چند مثال ساده برای روشن شدن کاربرد مشخصه‌ی ریاضی می‌پردازیم بی‌آن که خود را بیش از حد غرق جزئیات ریاضی کنیم.



شکل ۷-۲۱ جدول نمادها

مثال ۱: جدول نمادها (Symbol Table). برای نگهداری جدول نمادها از یک برنامه استفاده می‌شود. چنین جدولی به‌وفور در انواع متفاوت برنامه‌های کاربردی استفاده می‌شود. این جدول شامل مجموعه‌ای از آیتم‌ها بدون هرگونه تکرار می‌شود. مثالی از جدول نمادها در شکل ۷-۲۱ نشان داده شده است. این شکل، جدولی را نشان می‌دهد که توسط یک سیستم عامل به‌کار می‌رود تا نام کاربران سیستم را نگه دارد. سایر مثال‌های این جدول شامل مجموعه‌ای از نام کارمندان در سیستم پرداخت حقوق، مجموعه نام‌های کامپیوترهای موجود در یک سیستم ارتباطات شبکه‌ای و مجموعه‌ای از مقاصد در سیستمی برای تولید جدول‌های زمانی حمل و نقل می‌شود.

فرض کنید که تعداد نام‌های موجود در جدول ارائه شده در این مثال، از *MaxIds* بیشتر نباشد. این گزاره، که جدول را مقید می‌سازد، مؤلفه‌ای از یک شرط است که به‌عنوان *ثابت داده‌ای* (data invariant) شناخته می‌شود. ثابت داده‌ای، شرطی است که در سرتاسر اجرای سیستمی حاوی

روش‌های رسمی توان بالقوه‌ی عظیمی برای بهبود بخشیدن به وضوح و دقت مشخصات خواسته‌ها و یافتن خطاهای مهم و ظریف دارند.
استیو ایستربروک

نکته‌ی کلیدی
ثابت داده‌ای، مجموعه‌ای از شرایط است که در سرتاسر اجرای سیستمی، حاوی مجموعه داده‌های درست است.

[1] در واقع، حصول اطمینان از کامل بودن، دشوار است حتی هنگامی که از روش‌های رسمی استفاده شود. به موازاتی که مشخصه ایجاد شود، جنبه‌هایی از سیستم، تعریف‌نشده باقی می‌مانند؛ خصوصیات دیگری نیز ممکن است به عمد حذف شوند تا به طراحان امکان دهند تا در انتخاب رویکرد پیاده‌سازی قدری آزادانه عمل کنند؛ و سرانجام، این امکان وجود ندارد که هر سناریوی عملیاتی در یک سیستم پیچیده و بزرگ در نظر گرفته شود. چیزهایی ممکن است صرفاً به اشتباه حذف شوند.

یک مجموعه از داده‌ها، درست است. ثابت داده‌ای که برای جدول نمادهای ما بر قرار است، دو مؤلفه دارد: (۱) این که تعداد نام‌های جدول بیش از *MaxIds* نیست و (۲) این که هیچ نام تکراری در جدول وجود ندارد. در مورد برنامه‌ی جدول نمادها، این بدان معناست که جدول نمادها در هر زمان از اجرای سیستم که بررسی شود، همواره بیش از *MaxIds* نام ندارد و حاوی هیچ نام تکراری نیست.

یک مفهوم مهم دیگر، *حالت* است. زبان‌های رسمی بسیاری نظیر OCL (بخش ۱-۷-۲۱) از مفهوم حالت، آن طور که در فصل ۷ بحث شد، استفاده می‌کنند؛ یعنی، سیستم می‌تواند در یکی از چند حالت خود به سر برد که هر کدام یک شیوه‌ی رفتاری قابل مشاهده از بیرون را از خود نشان می‌دهد. به هر حال، تعریف متفاوتی از عبارت «*حالت*» در زبان Z (بخش ۲-۷-۲۱) به‌کار می‌رود. در Z (و زبان‌های مرتبط با آن)، حالت یک سیستم با داده‌های ذخیره شده در آن نمایش داده می‌شود (و از این رو، Z تعداد حالت‌های بسیار بیشتری را پیشنهاد می‌کند که هر کدام از پیکربندی‌های ممکن برای داده‌ها را نشان می‌دهد). با استفاده از تعریف اخیر در مثال برنامه جدول نمادها، حالت، جدول نمادهاست.

اندرز

یک راه دیگر برای نگاه کردن به مفهوم حالت این است که بگوییم داده‌ها حالت را تعیین می‌کند. یعنی، می‌توانید داده‌ها را بررسی کنید تا دریابید سیستم در چه حالتی به سر می‌برد.

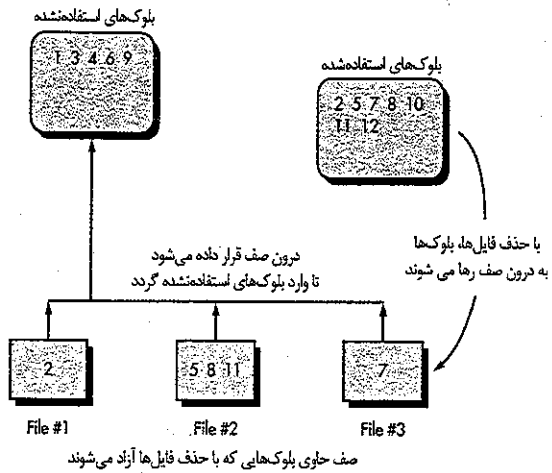
مفهوم نهایی، *عملیات* است. عملیات، فعالیتی است که در داخل سیستم انجام می‌شود و داده‌ها را می‌خواند یا می‌نویسد. اگر افزودن یا حذف‌کردن نام‌ها از جدول نمادها از جمله وظایف برنامه‌ی جدول نمادها باشد، این برنامه با دو عملیات مرتبط خواهد بود: یک عملیات *add( )* برای افزودن نام مشخصی به جدول نمادها و یک عملیات *remove( )* برای حذف یک نام از جدول[1]. اگر برنامه امکاناتی فراهم سازد که به کمک آن بتوان چک کرد که آیا نام خاصی در جدول هست و آن‌گاه علمیاتی وجود خواهد داشت که مقداری بر می‌گرداند که وجود آن نام در جدول را خاطرنشان سازد.

عملیات‌ها می‌توانند سه نوع شرط داشته باشند: ثابت‌ها، پیش‌شرط‌ها و پس‌شرط‌ها. *ثابت* چیزی را تعریف می‌کند که عدم تغییر آن تضمین می‌شود. برای مثال، جدول نمادها دارای ثابتی است که بیان می‌کند تعداد عناصر جدول همواره کوچک‌تر یا مساوی *MaxIds* است. *پیش‌شرط*، وضعیتی را توصیف می‌کند که یک علمیات خاص در آن معتبر است. برای مثال، پیش‌شرط مربوط به عملیاتی که به جدول نمادها یک شناسه اضافه می‌کند تنها در صورتی معتبر است که نامی که قرار است به جدول اضافه شود، از قبل در آن موجود نباشد و همچنین، تعداد نام‌های موجود در آن کوچک‌تر از *MaxIds* باشد. *پس‌شرط* یک عملیات، چیزی را تعریف می‌کند که درست بودن آن پس از کامل شدن آن عملیات، تضمین می‌شود و با اثری که بر داده‌ها دارد، تعریف می‌شود. برای عملیات *add( )* پس‌شرط به‌طور ریاضی مشخص می‌کند که جدول با شناسه‌ای جدید ارتقا یافته است.

مثال ۲: مدیریت بلوک‌ها. یکی از بخش‌های مهم سیستم‌های عامل ساده، زیرسیستمی است که فایل‌های ایجاد شده توسط کاربران را نگهداری می‌کند. بخشی از این زیرسیستم فایل بندی، مدیریت بلوک‌هاست. فایل‌های موجود در انبار فایل‌ها از بلوک‌های ذخیره‌سازی تشکیل می‌شوند که روی دستگاه ذخیره‌سازی فایل‌ها قرار دارند. طی مدت زمانی که کامپیوتر کار می‌کند، فایل‌هایی ایجاد یا حذف می‌شوند که این به معنی اشغال و رهاکردن بلوک‌هاست. برای این منظور، زیرسیستم فایل‌بندی، مخزنی از بلوک‌های استفاده‌نشده (آزاد) را نگه می‌دارد و مشخص می‌کند چه بلوک‌هایی در حال حاضر مورد استفاده قرار گرفته‌اند. هنگامی که بلوک‌ها با حذف فایل آزاد شدند، معمولاً به صفی از بلوک‌ها افزوده می‌شوند که منتظرند تا به مخزن بلوک‌های استفاده نشده افزوده شوند (شکل ۸-۲۱). در

[1] لازم به ذکر است که افزودن یک نام در حالت پر و حذف یک نام در حالت خالی امکان‌پذیر نیست.

این شکل، چند مؤلفه نشان داده شده است: مخزن بلوک‌های استفاده نشده، بلوک‌هایی که در حال حاضر فایل‌های سیستم عامل را تشکیل می‌دهند، و بلوک‌هایی که منتظرند تا به مخزن افزوده شوند. بلوک‌های منتظر در یک صف نگه داشته می‌شوند، به‌طوری که هر عنصر از صف حاوی مجموعه‌ای از بلوک‌های یک فایل حذف شده باشد.



شکل ۸-۲۱ سیستم مدیریت بلوک‌ها.

حالت برای این زیرسیستم، مجموعه‌ای از بلوک‌های آزاد، مجموعه‌ای از بلوک‌های استفاده شده و صف بلوک‌های بازگشتی است. ثابت داده‌ای، که به زبان طبیعی بیان می‌شود، عبارت است از:

- هیچ بلوکی هم به‌عنوان استفاده شده و هم به‌عنوان استفاده نشده علامت زده نمی‌شود.
- همه‌ی مجموعه بلوک‌های نگه داشته شده در صف، زیرمجموعه‌هایی از بلوک‌هایی هستند که در حال حاضر مورد استفاده‌اند.
- هیچ عنصری از صف حاوی تعداد بلوک‌های یکسان نیست.
- مجموعه بلوک‌های استفاده شده و بلوک‌هایی که استفاده نشده‌اند، همان مجموعه کل بلوک‌هایی است که فایل‌ها را تشکیل می‌دهند.
- مجموعه بلوک‌های استفاده‌نشده، هیچ بلوک تکراری ندارند.
- مجموعه بلوک‌های استفاده شده، هیچ بلوک تکراری ندارند.

برخی عملیات‌های مرتبط با این داده‌ها عبارتند از *add( )*، یعنی افزودن بلوک‌ها به انتهای هر صف، *remove( )* یعنی حذف مجموعه‌ای از بلوک‌ها از ابتدای یک صف و قرار دادن آن‌ها در مجموعه بلوک‌های استفاده نشده و *check( )* یعنی چک‌کردن این که آیا صف بلوک‌ها خالی است یا خیر.

پیش‌شرط *add( )* این است که بلوک‌هایی که قرار است اضافه شوند، باید در مجموعه بلوک‌های استفاده شده باشند. پس‌شرط این است که مجموعه بلوک‌ها اکنون در انتهای صف یافته می‌شوند. پیش‌شرط *remove( )* این است که صف باید حداقل یک آیتم در خود داشته باشد. پس‌شرط آن است

اندرز

هنگامی که باید یک ثابت داده‌ای برای قابلیت عملیاتی پیچیده توسعه دهید، تکنیک‌های طوفان فکری می‌توانند نتایج خوبی به‌دست دهند. از اعضای تیم بخواهید که محدودیت‌ها و قیدوبندها را بنویسند و سپس آن‌ها را ترکیب و ویرایش کنید.

که بلوک‌ها باید به مجموعه بلوک‌های استفاده نشده افزوده شوند. عملیات *check( )* فاقد پیش‌شرط است و این بدان معناست که این عملیات همواره تعریف شده است و اهمیتی ندارد که حالت چه مقداری داشته باشد. اگر صف خالی باشد، پس‌شرط، مقدار درست را بر می‌گرداند و در غیر این صورت، مقدار *نادرست* را بر می‌گرداند.

در مثال‌های ذکر شده در این بخش، مفاهیم کلیدی مشخصه‌ی رسمی را معرفی کردیم، ولی بدون این که بر ریاضیات مورد نیاز برای رسمی ساختن مشخصه تأکید ورزیم. در بخش ۶-۲۱ خواهیم دید که نمادگذاری ریاضی را چگونه می‌توان در تعیین مشخصات رسمی عنصری از سیستم به‌کار برد.

۶-۲۱ استفاده از نمادگذاری ریاضی[1] برای مشخصه‌ی رسمی

برای نمایش کاربرد نمادگذاری ریاضی در مشخصه‌ی رسمی یک مؤلفه‌ی نرم‌افزار، به همان مثال مدیریت بلوک‌ها خواهیم پرداخت که در بخش ۵-۲۱ بحث شد. گفتیم که مؤلفه‌ی مهمی از سیستم عامل، فایل‌های ایجاد شده توسط کاربران را نگهداری می‌کند. مدیریت بلوک‌ها، مخزنی از بلوک‌های استفاده نشده را نگهداری می‌کند و در عین حال حساب بلوک‌هایی را هم دارد که در حال حاضر مورد استفاده هستند. هنگامی که بلوک‌ها از یک فایل حذف شده آزاد شدند، معمولاً به صف بلوک‌هایی افزوده می‌شوند که در انتظارند تا به مخزن بلوک‌های استفاده نشده اضافه شوند. طرحی از این فرایند در شکل ۸-۲۱ ارائه شده است.

چگونه می‌توانم حالت‌ها و ثابت‌های داده‌ای را با استفاده از یک مجموعه و عملگرهای منطقی ارائه دهم؟

مجموعه‌ای با نام *BLOCKS* شامل شماره‌ی تمامی بلوک‌ها می‌شود. *AllBlocks* مجموعه‌ای از بلوک‌هاست که بین ۱ تا *MaxBlocks* قرار می‌گیرد. حالت توسط دو مجموعه و یک دنباله مدل‌سازی می‌شود. این دو مجموعه عبارتند از *used* و *free* هر دو مجموعه حاوی بلوک هستند- مجموعه‌ی *used* حاوی بلوک‌هایی است که هم اکنون در فایل‌ها استفاده شده‌اند و مجموعه‌ی *free* حاوی بلوک‌هایی است که برای فایل‌های جدید در دسترس‌اند. دنباله حاوی مجموعه‌ای از بلوک‌هایی خواهد بود که آماده‌ی آزادشدن از فایل‌های حذف‌شده‌اند. حالت را می‌توان به‌صورت زیر توصیف کرد:

$$used, free: \mathbb{P}\ BLOCKS$$

$$BlockQueue: \text{seq}\ \mathbb{P}\ BLOCKS$$

این توصیف، شباهت بسیار به اعلان متغیرهای برنامه دارد و بیان می‌کند که *used* و *free* مجموعه‌ی بلوک‌ها خواهند بود و *BlockQueue* یک دنباله است که هر عنصر از آن، مجموعه‌ای از بلوک‌هاست. ثابت داده‌ای را می‌توان به‌صورت زیر نوشت:

$$used \cap free = \varnothing \wedge$$

$$used \cup free = AllBlocks \wedge$$

$$\forall i: dom\ BlockQueue \bullet BlockQueue\ i \subseteq used \wedge$$

$$\forall i, j: dom\ BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

[1] این بخش با فرض آشنایی خواننده با نمادگذاری ریاضی مربوط به مجموعه‌ها و دنباله‌ها و نیز نمادگذاری منطقی به‌کاررفته در جبر گزاره‌ها نوشته شده است. در صورت نیاز به مرور این مباحث، مرور مختصری در وب‌سایت ویراست هفتم کتاب ارائه شده است. برای اطلاعات مشروح‌تر، [Jec06] یا [Pot04] را ببینید.

مرجع وب

اطلاعات مبسوطی درباره روش‌های رسمی را می‌توانید در آدرس زیر بیابید:
www.afm.sbu.ac.uk

مؤلفه‌های ریاضی ثابت داده‌ای با چهار مورد از مؤلفه‌های بیان شده به زبان طبیعی در بالا، همخوانی دارد. نخستین خط از ثابت داده‌ای بیان می‌کند که در مجموعه بلوک‌های استفاده شده و آزاد، هیچ اشتراکی وجود ندارد. خط دوم بیان می‌کند که مجموعه بلوک‌های استفاده شده و آزاد همواره برابر با مجموعه‌ی کل بلوک‌های سیستم است. خط سوم نشان می‌دهد که عنصر i ام در صف بلوک‌ها همواره زیرمجموعه‌ای از بلوک‌های استفاده شده است. خط پایانی هم می‌گوید که به ازای هر دو عنصر از صف بلوک‌ها که یکسان نیستند، هیچ بلوک مشترکی میان این دو عنصر وجود ندارد. دو مؤلفه‌ی آخر ثابت داده‌ای، که به زبان طبیعی بیان شد، با توجه به این واقعیت پیاده‌سازی می‌شوند که *used* و *free* مجموعه هستند و بنابراین حاوی عضو تکراری نیستند.

نخستین عملیاتی که باید تعریف شود، حذف عضوی از سه صف بلوک‌هاست. پیش‌شرط، این است که باید دست کم یک آیتم در صف وجود داشته باشد:

$$\#BlockQueue > 0$$

پس‌شرط، این است که ابتدای صف باید حذف شود و در مجموعه بلوک‌های آزاد قرار داده شود و صف طوری تنظیم شود که این حذف را نشان دهد:

$$used' = used \setminus head\ BlockQueue \wedge$$

$$free' = free \cup head\ BlockQueue \wedge$$

$$BlockQueue' = tail\ BlockQueue$$

پس‌شرط‌ها و پیش‌شرط‌ها را چگونه نمایش می‌دهیم؟

قراردادی که در بسیاری از روش‌های رسمی به‌کار می‌رود، آن است که مقدار یک متغیر پس از یک عملیات، پریم دار می‌شود. از این رو، نخستین مؤلفه از عبارت قبلی بیان می‌کند که بلوک‌های استفاده‌شده‌ی جدید (*used'*) برابر با بلوک‌های استفاده شده‌ی قدیمی منهای بلوک‌های حذف شده است. مؤلفه‌ی دوم بیان می‌کند که بلوک‌های آزاد جدید (*free'*)، همان بلوک‌های آزاد قدیمی است که سر صف بلوک‌ها به آن افزوده شده است. مؤلفه‌ی سوم بیان می‌کند که صف بلوک‌های جدید برابر با دم مقدار قدیمی صف بلوک‌ها یعنی، همه‌ی عناصر موجود در صف، جدا از عنصر اول، است. عملیات دوم، مجموعه‌ای از بلوک‌ها، *Ablocks* را به صف بلوک‌ها می‌افزاید. پیش‌شرط این عملیات آن است که *Ablocks* در حال حاضر مجموعه‌ای از بلوک‌های استفاده شده باشد.

$$Ablocks \subseteq used$$

پس‌شرط این عملیات آن است که مجموعه بلوک‌ها به انتهای صف بلوک‌ها افزوده شود و مجموعه بلوک‌های آزاد و استفاده نشده تغییر نکند:

$$BlockQueue' = BlockQueue \cap \langle Ablocks \rangle \wedge$$

$$used' = used \wedge$$

$$free' = free$$

مشخص‌کردن صف بلوک‌ها، بدون تردید دشوارتر از توصیف‌های روایی یا مدل‌های گرافیکی است. ولی آن‌چه که از سازگاری و کامل بودن این روش به‌دست می‌آید، برای برخی دامنه‌های کاربرد قابل توجیه است.

۲۱-۷ زبان‌های تعیین مشخصات رسمی

زبان تعیین مشخصات رسمی معمولاً از سه مؤلفه اصلی تشکیل می‌شود: (۱) یک قالب نحوی که تعیین می‌کند مشخصات با چه نمادگذاری خاصی باید ارائه شود، (۲) معناشناختی برای کمک به تعریف «مجموعه اشیاء مرجع» [Win90] که برای توصیف سیستم به‌کار گرفته خواهد شد و (۳) مجموعه‌ای از روابط که قواعدی را تعریف می‌کنند که نشان می‌دهند کدام اشیا به‌طور مناسب در مشخصه صدق می‌کنند.

دامنه‌ی نحوی یک زبان تعیین مشخصات رسمی غالباً مبتنی بر *نحوی* است که از نمادگذاری نظریه استاندارد مجموعه‌ها و جبر گزاره‌ها به‌دست آمده است. *دامنه‌ی معنا شناختی* یک زبان تعیین مشخصات، نشان می‌دهد که این زبان چگونه خواسته‌های سیستم را نمایش می‌دهد.

استفاده از انتزاع‌های معناشناختیِ متفاوت برای توصیف یک سیستم به شیوه‌های متفاوت، امکان‌پذیر است. در فصل‌های ۶ و ۷ این کار را به شیوه‌ای با رسمیت کمتر انجام دادیم. اطلاعات، عملکرد و رفتار، همگی ارائه شدند. برای به نمایش در آوردن یک سیستم می‌توان از مدل‌سازی‌های متفاوت استفاده کرد. معناشناسی هر نمایش، دیدگاه‌های مکملی از سیستم به‌دست می‌دهند. برای نشان دادن این رویکرد، هنگام استفاده از روش‌های رسمی، فرض کنید برای توصیف مجموعه رویدادهایی که باعث رخ دادن یک حالت خاص می‌شوند از یک زبان تعیین مشخصات رسمی استفاده می‌شود. یک رابطه رسمی دیگر، کلیه‌ی کارهایی را که در یک حالت مفروض انجام می‌شوند، به تصویر می‌کشد. اشتراک این دو رابطه، نشان‌گر رویدادهایی است که باعث می‌شوند تا کارهای خاصی انجام شوند.

مجموعه متنوعی از زبان‌های تعیین مشخصات رسمی هم اکنون در حال استفاده است. OCL [OMG03b] ، Z [ISO02]، LARCH [Gut93] و VDM [Jon91] چند نمونه از زبان‌های تعیین مشخصات رسمی‌اند که خصوصیات ذکر شده در بالا را از خود نشان می‌دهند. در این بخش، به اختصار درباره OCL و Z بحث خواهیم کرد.

۲۱-۷-۱ زبان قیدوبند اشیا (OCL)[1]

زبان قیدوبند اشیا (OCL) یک نمادگذاری رسمی است که طوری توسعه یافته است که کاربران UML بتوانند دقت بیشتری به مشخصات خود بیفزایند. همه‌ی قدرت منطق و ریاضیات گسسته، در این زبان در دسترس قرار دارد. ولی، طراحان OCL تصمیم گرفته‌اند که در گزاره‌های OCL فقط از کاراکترهای ASCII (به‌جای نمادگذاری سنتی ریاضی) استفاده شود. این باعث می‌شود که زبان مذکور نزد افرادی که میانه‌ی چندان خوبی با ریاضیات ندارند، ظاهری دوست داشتنی‌تر بگیرد و کامپیوتر راحت‌تر بتواند آن را پردازش کند. ولی این باعث می‌شود که OCL گاهی ظاهر کلامی به خود بگیرد.

برای استفاده از OCL با یک یا چند نمودار UML آغاز می‌کنید- معمولاً نمودارهای کلاس‌ها، حالت‌ها و فعالیت (پیوست ۱). عبارت‌های OCL به این نمودارها افزوده می‌شوند و حقایقی را درباره

[1] این بخش از کتاب، کار پروفسور تیموتی لتبریج از دانشگاه اتاواست و با کسب اجازه از ایشان در این کتاب آورده شده است.

جدول ۱-۲۱ خلاصه‌ای از نمادگذاری OCL

$x.y$	به‌دست آوردن خاصیت y از x. این خاصیت می‌تواند یک صفت، مجموعه‌ای از اشیا در پایان یک همبستگی، نتیجه ارزیابی یک عملیات، یا سایر چیزهایی باشد که به نوع نمودار UML بستگی دارد. اگر x یک مجموعه باشد، y روی تمامی عناصر x اعمال می‌شود؛ نتایج در یک مجموعه جدید گردآوری می‌شود.
$c->f()$	انجام عملیات OCL توکار f روی خود مجموعه c (در مقابل هر کدام از اشیای c). مثال‌هایی از عملیات‌های توکار در زیر فهرست شده‌اند.
and, or, =, <>	«و» منطقی، «یا» منطقی، مساوی، نامساوی.
$p \Rightarrow q$	درست است اگر q درست یا اگر p نادرست باشد.

مثال‌هایی از عملیات‌های قابل انجام روی مجموعه‌ها و دنباله‌ها.

$C->size()$	تعداد عناصر موجود در مجموعه یا دنباله c
$C->isEmpty()$	درست است اگر c فاقد عنصر باشد و در غیر این صورت، نادرست.
$c1->includesAll()$	درست است اگر همه‌ی عناصر $c2$ در $c1$ موجود باشند.
$c1->excludesAll()$	درست است اگر هیچ عنصری از $c2$ در $c1$ موجود نباشد.
$C->forAll(\text{elem} \mid \text{boolexpr})$	درست است اگر عبارت بولی boolexpr به هنگام اعمال روی هر عنصر از c درست باشد. هنگامی که عنصری ارزیابی می‌شود، به متغیر elem مقید است که در boolexpr قابل استفاده است. این همان کمّی‌سازی را پیاده‌سازی می‌کند که قبلا ذکر شد.
$C->forAll(\text{elem1}, \text{elem2} \mid \text{boolexpr})$	همانند بالا، با این تفاوت که boolexpr برای هر جفت عنصر بر گرفته شده از c، از جمله مواردی که در آن‌ها این جفت شامل عنصر یکسان می‌شود، ارزیابی می‌شود.
$C->isUnique(\text{elem} \mid \text{boolexpr})$	درست است اگر expr در صورت اِعمال روی هر کدام از عناصر c نتیجه‌ای متفاوت داشته باشد.

نمونه‌هایی از عملیات‌های خاص مجموعه‌ها

$s1->intersction(s2)$	مجموعه عناصر یافته‌شده در $s1$ و نیز در $s2$
$s1->union(s2)$	مجموعه عناصر یافته‌شده در $s1$ یا در $s2$
$s1->excluding(x)$	مجموعه $s1$ با شیء x حذف شده.

نمونه‌ای از علمیات‌های خاص دنباله‌ها

$Seq->first()$	شیء‌ای که نخستین عنصر از دنباله seq است.

عناصر نمودارها بیان می‌کنند. این عبارت‌ها را *قیدوبند* (constraint) می‌نامند؛ در هر پیاده‌سازی به‌دست آمده از مدل، باید اطمینان حاصل شود که قیدوبندها همواره بر قرارند.

عبارت‌های OCL همانند زبان‌های برنامه‌نویسی شیء گرا شامل عملگرهایی می‌شوند که روی اشیاء عمل می‌کنند. به هر حال، نتیجه‌ی یک عبارت کامل همواره باید یک ارزش بولی باشد که یا درست است یا نادرست. اشیا می‌توانند نمونه‌هایی از کلاس **Collection** در OCL باشند که خود شامل دو زیر کلاس **Set** و **Sequence** می‌شود.

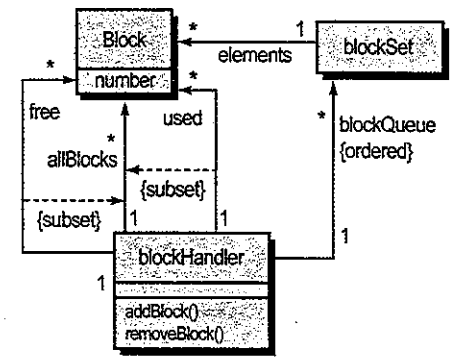
شیء **Self** عنصر نمودار UML است که عبارت OCL در حیطه‌ی آن تعیین می‌شود. سایر اشیا را می‌توان با *گشت و گذار* و با استفاده از نماد نقطه (.) از شیء **Self** به‌دست آورد. برای مثال:

- اگر **Self** کلاس **C** با صفت **a** باشد، در آن صورت، **self.a** شیء نگهداری شده در **a** را تعیین می‌کند.
- اگر **C** یک همبستگی یک به چند با نام *assoc* با کلاس دیگر **D** داشته باشد، در آن صورت self.assoc یک **Set** را تعیین می‌کند که عناصر آن از نوع **D** هستند.
- سرانجام (با قدری ظرافت بیشتر) اگر **D** دارای صفت b باشد، در آن صورت عبارت self.assoc.b مجموعه‌ای از همه‌ی bهای متعلق به همه‌ی **D**ها را تعیین می‌کند.

OCL عملیات‌های توکاری را فراهم می‌سازد که عملگرهای منطقی و مجموعه ها، مشخصات سازنده و ریاضیات وابسته به این مباحث را پیاده‌سازی می‌کنند. نمونه کوچکی از این عملگرها در جدول ۱-۲۱ ارائه شده است.

برای نشان دادن کاربرد OCL در تعیین مشخصات، به بررسی دوباره‌ی مثال مدیریت بلوک‌ها می‌پردازیم که در بخش ۵-۲۱ ارائه شد. مرحله نخست، توسعه‌ی یک مدل UML است (شکل ۹-۲۱). این نمودار کلاس‌ها روابط بسیاری را میان اشیای موجود مشخص می‌کند. به‌هرحال، عبارت‌های OCL افزوده می‌شوند تا کسانی که سیستم را پیاده‌سازی می‌کنند، دقیق‌تر بدانند چه چیز باید به هنگام اجرای سیستم، درست باقی بماند.

عبارت‌های OCL که نمودار کلاس‌ها را تکمیل می‌کنند، متناظر با شش بخش از ثابت‌هایی هستند که در بخش ۵-۲۱ بحث شدند. در مثالی که به‌دنبال خواهد آمد، ثابت به یک زبان طبیعی (مثلاً فارسی



شکل ۹-۲۱ نمودار کلاس‌ها برای سیستم مدیریت بلوک‌ها.

یا انگلیسی) تکرار می‌شود و سپس عبارت OCL متناظر با آن نوشته می‌شود. فراهم آوردن متنی به زبان طبیعی همراه با منطق رسمی، کار خوبی است؛ انجام این کار به شما کمک می‌کند تا منطق را دریابید و همچنین به کسانی که وظیفه‌ی بازبینی را بر عهده دارند کمک می‌کند تا اشتباهات را برملا سازند، مثل شرایطی که زبان طبیعی و منطق با یکدیگر مطابقت نداشته باشند.

۱. *هیچ بلوکی هم به‌عنوان استفاده شده و هم به‌عنوان استفاده نشده علامت زده نمی‌شود.*

```
context BlockHandler inv:
    (self.used -> intersection(self.free))  -> isEmpty()
```

توجه دارید که هر عبارت با واژه کلیدی context آغاز می‌شود. این واژه نشان‌گر عنصری از نمودار UML است که عبارت بر آن قیدوبند می‌گذارد. به‌طریق دیگر، می‌توانید قیدوبند را مستقیماً روی نمودار UML بگذارید و آن را با آکلاد {} محصور کنید. واژه کلیدی self در اینجا به نمونه‌ای از BlockHandler اشاره دارد؛ در مورد بعدی، آن گونه که در OCL رواست، self را حذف خواهیم کرد.

۲. *همه‌ی مجموعه بلوک‌های موجود در صف، زیرمجموعه‌هایی هستند از مجموعه بلوک‌هایی که در حال حاضر مورد استفاده‌اند.*

```
context BlockHandler inv:
    blockQueue -> forAll(aBlockSet | used -> includesA11(aBlockSet))
```

۳. *هیچ عنصری از صف حاوی تعداد بلوک‌های یکسان نیست.*

```
context BlockHandler inv:
    blockQueue -> forAll(BlockSet1, BlockSet2 |
      BlockSet1< > BlockSet2 implies
      BlockSet1.elements.number -> excludesA11(BlockSet1, BlockSet2))
```

عبارت قبل از implies لازم است تا اطمینان حاصل شود که از جفت‌های حاوی دو بلوک یکسان چشم‌پوشی می‌شود.

۴. *مجموعه بلوک‌های استفاده شده و بلوک‌هایی که استفاده نشده‌اند برابر با مجموعه کل بلوک‌های تشکیل دهنده فایل‌ها خواهد بود.*

```
context BlockHandler inv:
    allBlocks = used -> union (free)
```

۵. *مجموعه بلوک‌های استفاده‌نشده فاقد بلوک‌های تکراری خواهد بود.*

```
context BlockHandler inv:
    free -> isUnique(aBlocks | aBlocks.number)
```

۶. *مجموعه بلوک‌های استفاده‌شده فاقد بلوک‌های تکراری خواهد بود.*

```
context BlockHandler inv:
    used -> isUnique(aBlocks | aBlocks.number)
```

از OCL می‌توان در مشخص کردن پیش‌شرط‌ها و پس‌شرط‌های هر عملیات نیز استفاده کرد. برای مثال، عبارت زیر، عملیات‌هایی را توصیف می‌کند که مجموعه‌ای از بلوک‌ها را از صف حذف یا به آن

اضافه می‌کنند. توجه دارید که نماد x@pre شیء x را آن طوری نشان می‌دهد که قبل از عملیات وجود داشته است؛ این برخلاف نمادگذاری ریاضی است که قبلاً بحث شد؛ در آن حالت، x پس از عملیات، به‌صورت x' نشان داده می‌شود.

```
context BlockHandler::removeBlock()
    pre: blockQueue->size()>0
    post: used = used@pre-block@pre->first() and
      free = free@pre->union(blockQueue@pre->first()) and
      blockQueue = blockQueue@pre-> excluding(blockQueue@pre->first)

context BlockHandler::addBlock()
    pre: used->includesAll(aBlockSet.elements)
    post: blockQueue.elements = blockQueueSet.elements@pre
          - >append(aBlockSet)) and
          used = used@pre and
          free = free@pre
```

OCL یک زبان مدل‌سازی است، ولی همه‌ی صفات یک زبان رسمی را دارد. با OCL می‌توان قیدوبندهای گوناگون، پس‌شرط‌ها و پیش‌شرط‌ها، محافظ‌ها و سایر خصوصیات مرتبط با اشیای ارائه‌شده در مدل‌های گوناگون UML را بیان کرد.

۲-۷-۲۱ زبان تعیین مشخصات Z

Z (با تلفظ زِد) یک زبان تعیین مشخصات است که در جامعه‌ی روش‌های رسمی، کاربردی گسترده دارد. در زبان Z مجموعه‌های نوع‌دار، رابطه‌ها و وظایف در حیطه‌ی منطق گزاره‌ای مرتبه‌ی اول به‌کار برده می‌شوند تا شِماها (schema) را بسازند- شِما ابزاری برای ساختاردهی به مشخصه‌ی رسمی است.

مشخصات Z به‌صورت مجموعه‌ای از شِماها سازمان‌دهی می‌شوند- زبانی ساختاری که متغیرها را معرفی کرده واسط میان این متغیرها را مشخص می‌سازد. شِما اساساً همان مؤلفه‌ی زبان برنامه‌نویسی در مشخصه‌ی رسمی است. از شِماها در ساختاردهی به مشخصه رسمی استفاده می‌شود، درست همان‌گونه که مؤلفه‌ها در ساختاردهی به سیستم به‌کار می‌روند.

یک شِما داده‌های انبار شده‌ای را توصیف می‌کند که سیستم به آن‌ها دستیابی دارد و آن‌ها را تغییر می‌دهد. در حیطه‌ی Z این را «حالت» می‌نامند. این کاربرد واژه‌ی *حالت* در Z قدری با کاربرد آن در بقیه‌ی کتاب تفاوت دارد[1]. به علاوه، شِما، عملیات‌هایی را توصیف می‌کند که برای تغییر دادن حالت و روابطی به‌کار می‌روند که در داخل سیستم رخ می‌دهند. ساختار کلی شِما به شکل زیر است:

نام شِما

اعلان‌ها

ثابت‌ها

[1] به‌خاطر دارید که در فصل های دیگر، *حالت* برای شناسایی یک شیوه رفتاری سیستم به‌کار رفته است که از بیرون قابل مشاهده است.

مرجع وب

اطلاعات مشروح درباره زبان Z را می‌توانید در وب سایت زیر بیابید:

www.users.cs.york.ac.uk/~susan/abs/z.htm

که اعلان‌ها، متغیرهای تشکیل‌دهنده‌ی حالت سیستم را معین می‌کنند و ثابت‌ها، قیدوبندهای حاکم بر شیوه‌ی تکامل حالت را تعیین می‌کنند. خلاصه‌ای از نمادگذاری زبان Z در جدول ۲-۲۱ ارائه شده است.

جدول ۲-۲۱ خلاصه نمادگذاری Z

نمادگذاری Z مبتنی بر نظریه‌ی مجموعه‌های نوع‌دار و منطق گزاره‌ای مرتبه‌ی اول است. Z ساختاری به نام شِما ارائه می‌دهد که از آن برای توصیف فضای حالت و عملیات‌های مشخصه استفاده می‌شود. در زبان Z، شِمای X به شکل زیر مشخص می‌شود.

X

اعلان‌ها

گزاره‌ها

توابع و ثوابت سراسری به شکل زیر تعیین می‌شوند:

اعلان‌ها

گزاره‌ها

اعلان، نوع ثابت یا تابع را مشخص می‌کند در حالی که گزاره مقدار آن را مشخص می‌کند. تنها مجموعه خلاصه‌ای از نمادهای Z در این جدول ارائه شده است:

مجموعه‌ها

$S: \mathbb{P}\, X$	S به‌عنوان مجموعه‌ای از Xها اعلان می‌شود.
$x \in S$	x عضو S است.
$x \notin S$	x عضو S نیست.
$S \subseteq T$	S زیرمجموعه T است: هر عضو S در T نیز هست.
$S \cup T$	اتحاد S و T است: حاوی هر عضوی از S یا T یا هر دو آن‌هاست.
$S \cap T$	اشتراک S و T: حاوی هر عضوی که در S و در T باشد.
$x \setminus S$	تفاضل S و T: حاوی هر عضوی از S به استثنای اعضایی که در T هم هستند.
$\varnothing$	مجموعه تهی: فاقد عضو.
$\{x\}$	مجموعه تک‌عضوی: فقط حاوی عضو x
$\mathbb{N}$	مجموعه اعداد طبیعی 0,1,2,...
$S: \mathbb{F}\, X$	S به‌عنوان مجموعه‌ای متناهی از Xها اعلان می‌شود.
$\max(S)$	بیشینه‌ی یک مجموعه اعداد غیر تهی S

توابع

$f: X \rightarrowtail Y$	f به‌عنوان تزریق جزئی (partial injection) از x به y اعلان می‌شود.
$\mathrm{dom}\ f$	دامنه‌ی f: مجموعه مقادیری از x که به ازای آن‌ها $f(x)$ تعریف شده باشد.
$\mathrm{ran}\ f$	برد f: مجموعه مقادیری که $f(x)$ در اثر تغییر x روی دامنه‌ی f به خود می‌گیرد.
$f \oplus \{x \mapsto y\}$	تابعی که با f همخوانی دارد به جز این که x به y نگاشت می‌شود.
$\{x\} \lhd f$	تابعی مثل f به جز این که x از دامنه آن حذف می‌شود.

منطق:

$P \wedge Q$	P و Q: تنها در صورتی درست است که P و Q درست باشند.
$P \Rightarrow Q$	P و Q: درست است اگر P یا Q درست باشد.
$\theta S' = \theta S$	هیچ مؤلفه‌ای از شِمای S در یک عملیات تغییر نمی‌کند.

مثال زیر از یک شِما، حالت مدیریت بلوک‌ها (*BlockHandler*) و ثابت داده‌ای را توصیف می‌کند:

BlockHandler

$used,\ free:\ \mathbb{P}\ BLOCKS$

$BlockQueue: \text{seq}\ \mathbb{P}\ BLOCKS$

$used \cap free = \varnothing \land$

$used \cup free = AllBlocks \land$

$\forall i: dom\ BlockQueue \bullet BlockQueue\ i \subseteq used \land$

$\forall i, j: dom\ BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$

همان طور که گفته شد، شِما از دو بخش تشکیل می‌شود. بخش بالای خط مرکزی، نشان‌گر متغیرهای حالت است، در حالی که بخش پایینی خط مرکزی ثابت داده‌ای را توصیف می‌کند. هر گاه که شِما، عملیات‌های ویژه‌ای را مشخص کند که حالت را تغییر می‌دهند، قبل از آن نماد Δ آورده می‌شود. مثال زیر از یک شِما، عملیاتی را توصیف می‌کند که عنصری از صف بلوک‌ها را حذف می‌کند:

RemoveBlocks

$\Delta BlockHandler$

$\#BlockQueue > 0$

$used' = used \setminus head\ BlockQueue \land$

$free' = free \cup head\ BlockQueue \land$

$BlockQueue' = tail\ BlockQueue$

گنجاندن $\Delta BlockHandler$ باعث می‌شود که همه‌ی متغیرهای تشکیل‌دهنده‌ی حالت برای شمای *RemoveBlocks* در دسترس قرار گیرند و این اطمینان را ایجاد می‌کند که ثابت داده‌ای، قبل و بعد از اجرای عملیات، برقرارند.

عملیات دوم، که مجموعه‌ای از بلوک‌ها را به انتهای صف اضافه می‌کند، به‌صورت زیر نمایش داده می‌شود:

AddBlocks

$\Delta BlockHandler$

$Ablocks? : BLOCKS$

$Ablocks? \subseteq used$

$BlockQueue' = BlockQueue \cap \langle Ablocks \rangle \land$

$used' = used \land$

$free' = free$

طبق قرارداد در Z، یک متغیر ورودی که خوانده می‌شود، ولی بخشی از حالت را تشکیل نمی‌دهد، با علامت سؤال پایان می‌یابد. از این رو، Ablocks? که به‌عنوان پارامتر ورودی عمل می‌کند با علامت سؤال پایان می‌یابد.

ابزارهای نرم‌افزاری

روش‌های رسمی

هدف: هدف ابزارهای روش‌های رسمی، کمک به تیم نرم‌افزاری در تعیین مشخصات و وارسی است.

مکانیک: مکانیک این ابزارها متفاوت است. به‌طور کلی، این ابزارها به تعیین مشخصات و خودکارسازی اثبات صحت کمک می‌کنند که معمولاً توسط یک زبان تخصص یافته برای اثبات قضایا قابل تعریف هستند. بسیاری از این ابزارها جنبه‌ی تجاری ندارند و برای اهداف پژوهشی توسعه یافته‌اند.

ابزارهای نمونه

ACL2، که توسط دانشگاه تگزاس (**www.cs.utexas.edu/users/moore/acl2**) توسعه یافته است، یک زبان برنامه‌نویسی است که در آن می‌توانید «سیستم‌های کامپیوتری را مدل‌سازی کنید و به علاوه، ابزاری است که شما را در اثبات خواص مدل‌های ساخته شده یاری می‌دهد.»

EVES، که توسط ORA Canada (**www.ora.on.ca/eves.html**) توسعه یافته است، زبان وردی (Verdi) را برای تعیین مشخصات رسمی و یک مولد اثبات خودکار پیاده‌سازی می‌کند.

فهرست مبسوطی از بالغ بر نود ابزار روش‌های رسمی را در وب سایت **http://vl.fmnet.info/** می‌توانید بیابید.

۲۱-۸ خلاصه

مهندسی نرم‌افزار اتاق تمیز یک رویکرد رسمی در توسعه‌ی نرم‌افزار است که می‌تواند به نرم‌افزارهایی با کیفیت بسیار بالا منجر شود. در این رویکرد از مشخصه‌های ساختاری چهارگوش برای مدل‌سازی طراحی و تحلیل استفاده می‌شود و به‌عنوان سازوکار اصلی برای یافتن و حذف خطاها بر وارسی تأکید می‌شود نه بر آزمون.

در تهیه‌ی اطلاعات مربوط به آهنگ شکست که برای تایید قابلیت اطمینان نرم‌افزار تحویل شده ضرورت دارد، از آزمون کاربرد آماری استفاده می‌شود.

رویکرد اتاق تمیز با مدل‌های تحلیل و طراحی آغاز می‌شود که از نمایش ساختارهای چهارگوش در آن‌ها استفاده می‌شود. هر «چهارگوش»، سیستم (یا جنبه‌ای از سیستم) را در سطح معینی از انتزاع پنهان‌سازی می‌کند. از چهارگوش‌های سیاه برای نمایش رفتار مشاهده‌پذیر سیستم استفاده می‌شود. چهارگوش‌های حالت، داده‌ها و عملیات‌های حالت را پنهان‌سازی می‌کنند. چهارگوش شفاف در مدل‌سازی طراحی روالی به‌کار می‌رود که از داده‌ها و عملیات‌های یک چهارگوش حالت انتظار می‌رود.

وارسی، هنگامی به‌کار می‌رود که طراحی ساختارهای چهارگوش کامل باشد. طراحی روالی برای یک مؤلفه‌ی نرم‌افزار به یک سری زیر تابع افراز می‌شود. برای اثبات صحت زیر تابع‌ها، شرط‌های خروج برای هر زیر تابع و مجموعه‌ای از اثبات‌های فرعی تعریف می‌شود. اگر هر شرط خروجی برقرار باشد، طراحی باید صحیح باشد.

هنگامی که وارسی به پایان رسید، آزمون کاربرد آماری آغاز می‌شود. بر خلاف آزمون‌های سنتی، مهندسی نرم‌افزار تمیز بر آزمون واحدها یا انسجام تأکید نمی‌کند. در عوض، نرم‌افزار با تعریف مجموعه‌ای از سناریوهای کاربرد، تعیین احتمال کاربرد برای هر سناریو و سپس تعریف آزمون‌های تصادفی می‌شود که با احتمالات مطابقت دارند. سوابق خطایی که جمع‌آوری می‌شوند با مدل‌های نمونه برداری، مؤلفه‌ها، و تأیید، ترکیب می‌شوند تا محاسبه قابلیت اطمینان پیش بینی شده برای مؤلفه نرم‌افزار امکان‌پذیر گردد.

روش‌های رسمی از امکانات توصیفی نظریه مجموعه‌ها و نمادگذاری منطقی استفاده می‌کنند تا مهندس نرم‌افزار این امکان را بیابد که حقایق (خواسته ها) را به وضوح بیان کند. مفاهیم بنیادی حاکم بر روش‌های رسمی عبارتند از: (۱) ثابت داده‌ای، شرطی که در سرتاسر اجرای سیستمی که حاوی یک مجموعه از داده هاست، درست است؛ (۲) حالت، نمایشی از شیوه رفتاری سیستم که از بیرون قابل مشاهده است، یا (به زبان Z یا زبان‌های مرتبط با آن) داده‌های ذخیره شده‌ای که یک سیستم به آن‌ها دسترسی دارد و می‌تواند آن‌ها را تغییر دهد؛ و (۳) عملیات، کنشی که در سیستم رخ می‌دهد و داده‌ها را روی یک حالت می‌نویسد یا می‌خواند یک عملیات با دو شرط همراه است: پیش‌شرط و پس‌شرط.

آیا مهندسی نرم‌افزار اتاق تمیز یا روش‌های رسمی به‌طور گسترده کاربرد پیدا خواهند کرد. پاسخ این است: احتمالاً خیر. فراگیری آن‌ها از روش‌های مهندسی نرم‌افزار سنتی دشوارتر است و برای برخی نرم‌افزارنویسان احتمالاً یک شوک فرهنگی به دنبال خواهد داشت. ولی بار دیگر که شنیدید کسی می‌پرسد چرا نمی‌توان نرم‌افزاری ساخت که همان بار اول درست کار کند، این را می‌دانید که تکنیک‌هایی وجود دارد که دقیقاً این کار را میسر می‌سازد.

مسائل و نکاتی برای تعمق

۲۱-۱ اگر قرار بود یک جنبه از مهندسی نرم‌افزار اتاق تمیز را انتخاب کنید که آن را به‌طور بنیادی از رویکردهای مهندسی نرم‌افزار سنتی یا شیءگرا متمایز کند، آن جنبه چه بود؟

۲۱-۲ یک مدل فرایند افزایشی و صدورگواهی چگونه با هم کار می‌کنند تا نرم‌افزاری با کیفیت بالا تولید کنند؟

۲۱-۳ با استفاده از مشخصه‌ی ساختاری چهارگوش، مدل‌های تحلیل و طراحی را برای سیستم *SafeHome* «در گنر اول» تهیه کنید.

۲۱-۴ الگوریتم مرتب‌سازی حبابی به شیوه زیر تعریف می‌شود:

```
procedure bubblesort;
var i, t, integer;
begin
repeat until t=a[1]
      t:=a[1];
      for j:= 2 to n do
      if a[j-1] > a[j] then begin
```



```
      t:=a[j-1];[
      a[j-1]:=a[j];
      a[j]:=t;
      end
endrep
end
```

این طراحی را به چند زیرتابع، افراز کنید و مجموعه شرایطی را مشخص کنید که به کمک آن‌ها بتوان درستی این الگوریتم را اثبات کرد.

۲۱-۵ اثبات درستی مرتب‌سازی بحث شده در مسأله ۴-۲۱ را مستندسازی کنید.

۲۱-۶ برنامه‌ای را که مرتب از آن استفاده می‌کنید (مثلاً برنامه پست الکترونیکی، واژه پرداز یا صفحه گسترده) انتخاب کنید. یک مجموعه سناریوی کاربرد برای آن بنویسید. احتمال استفاده از هر سناریو را تعیین کنید و سپس یک جدول توزیع احتمال و محرک برنامه مشابه با آن‌چه در بخش ۱-۴-۲۱ ارائه شد، تهیه کنید.

۲۱-۷ برای جدول توزیع احتمال و محرک برنامه که در مسأله ۶-۲۱ تهیه کردید، از یک مولد اعداد تصادفی استفاده کنید و مجموعه‌ای از موارد آزمون را برای استفاده در آزمون کاربرد آماری توسعه دهید.

۲۱-۸ به زبان ساده هدف از صدورگواهی را در حیطه‌ی مهندسی نرم‌افزار اتاق تمیز شرح دهید.

۲۱-۹ در تیمی که برای توسعه نرم‌افزار یک فکس‌سودم کار می‌کند، وظیفه‌ای به شما محول شده است. وظیفه شما، توسعه‌ی بخش «دفترچه تلفن» برای این برنامه کاربردی است. این قابلیت، نگهداری حداکثر *MaxName* نفر همراه با نام‌های شرکت تجاری مربوط، شماره نمابرها و سایر اطلاعات را میسر می‌سازد. با استفاده از زبان طبیعی، موارد زیر را تعریف کنید:

الف. ثابت داده‌ای

ب. حالت

پ. عملیات‌هایی که محتمل هستند.

۲۱-۱۰ در تیمی که برای توسعه نرم‌افزاری با نام *MemoryDoubler* (مضاعف کننده حافظه) کار می‌کند، وظیفه‌ای بر عهده شما نهاده شده است؛ این برنامه حافظه ظاهری بیشتری نسبت به حافظه فیزیکی فراهم می‌آورد. برای این منظور، بلوک‌هایی از حافظه که به یک برنامه‌ی موجود نسبت داده شده‌اند، ولی از آن‌ها استفاده نمی‌شود، شناسایی، جمع‌آوری و دوباره تخصیص داده می‌شوند. این بلوک‌های استفاده‌نشده، دوباره به برنامه‌هایی نسبت داده می‌شوند که به حافظه اضافی نیاز دارند. با پذیرفتن فرض‌های مناسب و به‌کارگیری زبان مناسب، موارد زیر را تعریف کنید:

الف. ثابت داده‌ای

ب. حالت

پ. عملیات‌هایی که محتمل هستند.

۲۱-۱۱ با استفاده از نمادگذاری OCL یا Z که در جدول ۱-۲۱ یا جدول ۲-۲۱ ارائه شدند، بخشی از سیستم امنیتی *SafeHome* را انتخاب کنید و سعی کنید آن را با OCL یا Z مشخص کنید.

۲۱-۱۲ در خصوص معناشناسی و قالب نحوی یک زبان تعیین مشخصات رسمی غیر از OCL یا Z یک سمینار نیم‌ساعته ارائه دهید.

فصل ۲۲

مدیریت پیکربندی نرم‌افزار

نگاهی گذرا

مدیریت پیکربندی چیست؟ هنگامی که یک نرم‌افزار کامپیوتری می‌سازید، تغییراتی رخ می‌دهند. چون تغییر رخ می‌دهند، نیاز به کنترل مؤثر آن دارید. مدیریت پیکربندی نرم‌افزار (SCM) مجموعه‌ای از فعالیت‌هاست که برای کنترل تغییرات طراحی شده‌اند. به این ترتیب که محصولات کاری‌ای را که باید تغییر نمایند، شناسایی می‌کنند، روابط میان آنها را مشخص می‌سازند، سازوکارهایی برای مدیریت نسخه‌های متفاوتی از این محصولات کاری تعریف می‌کنند، تغییرات تحمیل شده را کنترل می‌کنند و تغییرات اعمال شده را ممیزی و گزارش می‌کنند.

چه کسی آن را انجام می‌دهد؟ همه‌ی کسانی که در فرایند مهندسی نرم‌افزار شرکت دارند تا حدی با SCM سروکار دارند، ولی گاه برای مدیریت فرایند SCM افراد خاصی در نظر گرفته می‌شوند.

چرا اهمیت دارد؟ اگر تغییرات را کنترل نکنید، آنها شما را کنترل می‌کنند. این اصلاً خوب نیست. یک جریان کنترل نشده از تغییرات، به آسانی می‌تواند یک پروژه خوب را به آشوب بکشاند. از این رو، مدیریت تغییرات، بخشی ضروری از مدیریت کیفیت است.

مراحل کار کدام است؟ چون هنگام ساخت یک نرم‌افزار، محصولات کاری فراوانی ساخته می‌شود، هر یک را باید منحصراً مورد شناسایی قرار داد. هنگامی که این کار انجام شد، می‌توان سازوکارهایی برای کنترل تغییر و نسخه نرم‌افزار بنا نهاد. برای حصول اطمینان از حفظ کیفیت نرم‌افزار به موازات اعمال تغییرات، فرایند مورد ممیزی قرار می‌گیرد و برای حصول اطمینان از آگاهی افراد ذی‌صلاح، گزارش‌هایی ارائه می‌شود.

محصول کار چیست؟ برنامه مدیریت پیکربندی نرم‌افزار، راهبرد پروژه را برای SCM تعیین می‌کند. به‌علاوه، وقتی به SCM رسمی روی آورده شود، فرایند کنترل تغییرات، درخواست‌های تغییر نرم‌افزار را تولید کرده درخواست تغییر مهندسی را گزارش می‌کند.

چگونه اطمینان حاصل کنم که درست از عهده کار برآمده‌ام؟ هنگامی که هر محصول کاری را بتوان توضیح داد، پیگیری کرد و کنترل نمود؛ هنگامی که هر تغییر را بتوان تحلیل کرد و هنگامی که همه‌ی افراد ذی‌صلاح از یک تغییر مطلع شدند؛ شما کار خود را درست انجام داده‌اید.

در ساخت نرم‌افزارهای کامپیوتری، تغییر امری اجتناب‌ناپذیر است. تغییرات باعث افزایش سردرگمی مهندسان نرم‌افزاری می‌شود که روی یک پروژه کار می‌کنند. اگر پیش از اِعمال تغییرات آنها را مورد تحلیل قرار ندهند، پیش از پیاده‌سازی ثبت نکنند، به افراد ذی‌صلاح گزارش نکنند یا به شیوه‌ای کنترل نشوند که باعث بهسازی کیفیت و کاهش خطا شوند، کارها بغرنج خواهد شد. بابیج [Bab86] در این مورد می‌گوید:

> هنر هماهنگ‌سازی توسعه نرم‌افزار برای به حداقل رساندن ... سردرگمی را مدیریت پیکربندی می‌گویند. مدیریت پیکربندی، هنر شناسایی، سازمان‌دهی و کنترل اصلاحاتی است که باید در یک نرم‌افزار در حال ساخت توسط تیم برنامه‌نویس، اعمال شوند. هدف، به حداکثر رساندن بهره‌وری با به حداقل رساندن اشتباهات است.

مدیریت پیکربندی نرم‌افزار (SCM) یک فعالیت چتری است که در سرتاسر فرایند نرم‌افزار قابل اجراست. چون امکان تغییر در هر زمانی وجود دارد، فعالیت‌های SCM به دلایلی که به دنبال خواهد آمد، اجرا می‌شوند: (۱) شناسایی تغییر، (۲) کنترل تغییر، (۳) حصول اطمینان از پیاده‌سازی مناسب تغییر، و (۴) گزارش تغییر به دیگرانی که ممکن است علاقمند باشند.

درک تفاوت میان پشتیبانی نرم‌افزار و مدیریت پیکربندی نرم‌افزار حائز اهمیت است. پشتیبانی عبارت از یک مجموعه فعالیت‌های مهندسی نرم‌افزار است که پس از تحویل نرم‌افزار به مشتری و به‌کار انداختن آن رخ می‌دهد. مدیریت پیکربندی نرم‌افزار، مجموعه‌ای از فعالیت‌های پیگیری و کنترل است که با شروع‌شدن پروژه مهندسی نرم‌افزار آغاز می‌شود و با به‌کار انداختن نرم‌افزار پایان می‌یابد.

یک هدف اصلی مهندسی نرم‌افزار، بهبودبخشیدن به سهولت اسکان تغییرات و کاهش دادن مقدار تلاش صرف شده به هنگام اعمال تغییرات است. در این فصل، به بررسی فعالیت‌های خاصی خواهیم پرداخت که شما را قادر به اداره‌ی تغییرات خواهند ساخت.

۲۲-۱ مدیریت پیکربندی نرم‌افزار

خروجی فرایند نرم‌افزار، اطلاعاتی است که به سه گروه عمده قابل تقسیم است: (۱) برنامه‌های کامپیوتری (چه در سطح منبع و چه اجرایی)؛ (۲) مستنداتی که این برنامه‌های کامپیوتری را توصیف می‌کنند (چه در بُعد فنی و چه در بُعد کاربر) و (۳) داده‌ها (که یا در دل برنامه‌ها جای دارند یا در خارج از آنها). چیزهایی که کلیه اطلاعات تولید شده را به‌عنوان بخشی از فرایند نرم‌افزار تشکیل می‌دهند، در مجموع *پیکربندی نرم‌افزار* نام دارند.

با پیشرفت فرایند نرم‌افزار، تعداد آیتم‌های پیکربندی نرم‌افزار (SCIها) به سرعت رشد می‌کند. مشخصه سیستم شامل یک برنامه پروژه نرم‌افزاری و مشخصه خواسته‌های نرم‌افزار و مستندات مرتبط با سخت‌افزار است. اینها نیز به نوبه خود شامل مستندات دیگری می‌شوند که سلسله مراتبی از اطلاعات را تولید می‌کند. اگر هر SCI فقط شامل SCIهای دیگر می‌شد، مسأله چندان بغرنج نمی‌شد. متأسفانه یک متغیر دیگر نیز در فرایند دخالت دارد: *تغییر*. تغییرات ممکن است در هر زمان و به هر دلیلی رخ دهد. درواقع، قانون اول مهندسی سیستم‌ها [Ber80] می‌گوید: «در هر جای چرخه حیات سیستم که باشید، سیستم تغییر می‌کند و تمایل به تغییر در سرتاسر چرخه حیات باقی است.»

> «هیچ چیز دائمی نیست، جز تغییر»
> هراکلیتوس
> ۵۰۰ قبل از میلاد

منشاء این تغییرات چیست؟ پاسخ این سؤال به اندازه خود تغییرات با تغییر مواجه است، ولی چهار منبع اصلی برای تغییرات می‌توان ذکر کرد:

- شرایط بازاری یا تجاری جدید که تغییراتی در خواسته‌های محصول یا قواعد تجاری دیکته می‌کنند.
- نیازهای جدید ذی‌نفع که اصلاحاتی را در داده‌های تولید شده توسط سیستم‌های اطلاعاتی، عملکرد تحویل‌شده توسط محصولات یا سرویس‌های تحویل شده توسط یک سیستم کامپیوتری طلب می‌کنند.
- سازمان‌دهی مجدد یا رشد/ زوال تجاری که باعث تغییر در اولویت‌های پروژه یا ساختار تیم مهندسی نرم‌افزار می‌شود.
- قیدوبندهای بودجه‌ای یا زمان‌بندی که باعث تعریف مجدد محصول یا سیستم می‌شود.

مدیریت پیکربندی نرم‌افزار (SCM) مجموعه‌ای از فعالیت‌هاست که برای مدیریت تغییرات در سرتاسر چرخه حیات نرم‌افزار کامپیوتری توسعه یافته‌اند.

مدیریت پیکربندی نرم‌افزار، مجموعه‌ای از فعالیت‌هاست که برای مدیریت تغییرات در سرتاسر چرخه حیات نرم‌افزار کامپیوتری توسعه یافته‌اند. SCM را می‌توان به عنوان یک فعالیت تضمین کیفیت نرم‌افزار در نظر گرفت که در سرتاسر فرایند نرم‌افزار به کار می‌رود. در بخش‌های بعدی، به توصیف وظایف اصلی SCM و مفاهیم مهمی خواهیم پرداخت که می‌توانند شما را در مدیریت تغییرات یاری دهند.

> منشاء تغییراتی که برای نرم‌افزار درخواست می‌شود چیست؟

۲۲-۱-۱ سناریوی SCM [1]

یک سناریوی عملیاتی CM متداول شامل این افراد می‌شود: مدیر پروژه‌ای که مسؤولیت گروه نرم‌افزار را بر عهده دارد، مدیر پیکربندی که مسؤولیت روال‌ها و خط مشی‌های CM را بر عهده دارد، مهندسان نرم‌افزار که مسؤول توسعه و نگهداری محصول نرم‌افزاری هستند و مشتری که از محصول استفاده می‌کند. در این سناریو، فرض می‌شود که محصول یک نرم‌افزار کوچک شامل حدوداً ۱۵۰۰۰ خط است که توسط تیمی شش نفره توسعه می‌یابد. (توجه دارید که سناریوهای دیگری با تیم‌های بزرگتر یا کوچک تر، امکان پذیرند، ولی در اصل، مسائل کلی وجود دارند که هر کدام از این پروژه‌ها در خصوص CM با آنها مواجه‌اند).

این سناریو در سطح عملیاتی شامل نقش‌ها و وظایف گوناگون می‌شود. برای مدیر پروژه، این هدف عبارت است از حصول اطمینان از این که محصول در یک چارچوب زمانی معین توسعه می‌یابد. از این رو، مدیر، پیشرفت توسعه را پایش می‌کند و با تشخیص مشکلات به آنها واکنش نشان می‌دهد. این کار با ایجاد و تحلیل گزارش‌هایی در خصوص وضعیت سیستم نرم‌افزار و اجرای مرور روی سیستم انجام می‌شود.

اهداف مدیر پیکربندی عبارتند از حصول اطمینان از این که روال‌ها و خط مشی‌های ایجاد، تغییر و آزمون کدها رعایت می‌شوند و نیز تهیه‌ی اطلاعاتی درباره قابلیت دسترسی به پروژه. این مدیر برای پیاده‌سازی تکنیک‌های مربوط به حفظ کنترل تغییرات کد، سازوکارهایی برای انجام درخواست‌های

> اهداف و فعالیت‌های اجرا شده توسط هر کدام از هیأت‌های دخیل در مدیریت تغییرات، کدام‌اند؟

[1] این بخش از [Dar01] استخراج شده است.

رسمی جهت تغییرات، برای ارزیابی آن‌ها (از طریق یک هیأت کنترل تغییرات که مسؤول تصویب تغییرات روی سیستم نرم‌افزار است) و سازوکارهایی برای مجاز ساختن تغییرات معرفی می‌کند. این مدیر فهرست‌های وظایف مهندسان را تهیه و در میان آن‌ها توزیع می‌کند و اساساً حیطه‌ی پروژه را مشخص می‌سازد. مدیر همچنین آمار مربوط به مؤلفه‌های موجود در سیستم نرم‌افزار شامل اطلاعات مربوط به مؤلفه‌های مشکل آفرین سیستم را جمع‌آوری می‌کند.

هدف برای مهندسان نرم‌افزار، کارکردن اثربخش است. این بدان معناست که مهندسان برای ایجاد و آزمون کدها و در تولید محصولات کاری پشتیبان، بیهوده در کار یکدیگر دخالت نمی‌کنند، ولی در همان حال، تلاش می‌کنند به‌طور اثربخش با هم در ارتباط باشند و هماهنگ عمل کنند. مهندسان به‌طور مشخص از ابزارهایی بهره می‌برند که به ساخت محصول نرم‌افزار سازگار کمک می‌کنند. آن‌ها با آگاه ساختن یکدیگر از وظایف لازم و وظایف به انجام رسیده، با یکدیگر ارتباط برقرار می‌کنند و هماهنگ می‌شوند. تغییرات با ادغام فایل‌ها در یکدیگر در کارهای دیگران انتشار می‌یابند. سازوکارهایی وجود دارد که به کمک آن‌ها می‌توان اطمینان یافت برای مؤلفه‌هایی که دستخوش تغییرات همزمان می‌شوند، راهی برای بر طرف ساختن تضادها و ادغام تغییرات در یکدیگر وجود دارد. سابقه‌ای از تکامل همه‌ی مؤلفه‌های سیستم، همراه با شرحی از دلایل و نیز ثبت آن‌چه که واقعاً تغییر داده شده است، نگه داشته می‌شود. مهندسان برای ایجاد، تغییر، آزمون و منسجم ساختن کدها، فضای کاری خاص خود را خواهند داشت. در نقطه‌ای معین، از کدها یک خط مبنا ساخته می‌شود که توسعه‌ی بیشتر بر اساس آن خط مبنا ادامه می‌یابد و شکل‌های تغییر یافته‌ی کد برای سایر ماشین‌های هدف از آن ساخته می‌شوند.

مشتری از محصول استفاده می‌کند. چون محصول تحت کنترل CM است، مشتری روال‌های رسمی مربوط به درخواست تغییرات و برای خاطر نشان ساختن اشکال‌ها در محصول را دنبال می‌کند.

به‌طور ایده‌آل، سیستم CM به‌کار رفته در این سناریو باید این نقش‌ها و وظایف را پشتیبانی کند؛ یعنی، نقش‌ها هستند که قابلیت عملیاتی لازم برای سیستم CM را تعیین می‌کنند. مدیر پروژه، CM را به مثابه یک سازوکار ممیزی می‌بیند؛ مدیر پیکربندی آن را به‌عنوان سازوکارهای کنترلی، ردگیری و تعیین خط مشی می‌بیند؛ مهندس نرم‌افزار آن را به‌عنوان سازوکار اعمال تغییرات، ساختمان و کنترل دستیابی می‌بیند؛ و مشتری آن را سازوکاری برای تضمین کیفیت می‌داند.

نکته‌ی کلیدی

برای حصول اطمینان از ردگیری، مدیریت و اجرای مناسب تغییرات همزمان باید سازوکاری وجود داشته باشد.

۲-۱-۲۲ عناصر سیستم مدیریت پیکربندی

سوزان دارت [Dar01] در مقاله جامع خود در باب مدیریت پیکربندی نرم‌افزار، چهار عنصر مهم را بر می‌شمرد که باید هنگام توسعه‌ی سیستم مدیریت پیکربندی موجود باشند:

- *عناصر مؤلفه ای*- مجموعه‌ای از ابزارهای نهاده شده در داخل یک سیستم مدیریت فایل (مثلاً یک بانک اطلاعاتی) که دستیابی به هر کدام از آیتم‌های پیکربندی نرم‌افزار و مدیریت آن‌ها را میسر می‌سازد.
- *عناصر پردازشی*- مجموعه‌ای از کنش‌ها و وظایف که رویکردی اثربخش برای تغییر دادن مدیریت (و فعالیت‌های مرتبط با آن) را برای کلیه گروه‌های موجود در مدیریت، مهندسی و کاربران نرم‌افزار کامپیوتری تعریف می‌کند.

- *عناصر ساختمانی*- مجموعه‌ای از ابزارها که ساخت خودکار نرم‌افزار را با حصول اطمینان از مونتاژ مجموعه‌ی مناسبی از مؤلفه‌های اعتبارسنجی شده (یعنی نسخه مناسب) امکان‌پذیر می‌سازند.
- *عناصر انسانی*- مجموعه‌ای از ابزارها و ویژگی‌های پردازشی (شامل سایر عناصر CM) که تیم مهندسی نرم‌افزار به‌کار می‌گیرد تا SCM را به‌طور مناسب پیاده‌سازی کند.

این عناصر (که به تفصیل بیشتر در بخش‌های آینده بحث خواهند شد) یکدیگر را طرد نمی‌کنند. به‌عنوان مثال، عناصر مؤلفه‌ای در ارتباط با عناصر ساختمانی کار می‌کنند تا فرایند نرم‌افزار تکامل پیدا کند. عناصر پردازشی بسیاری از فعالیت‌های انسانی را که با SCM در ارتباط هستند، راهنمایی می‌کنند و بنابراین شاید بتوان آن‌ها را عناصر انسانی نیز در نظر گرفت.

۳-۱-۲۲ خط مبنا (Baseline)

تغییر، حقیقت انکارناپذیری در توسعه نرم‌افزار است. مشتریان می‌خواهند خواسته‌ها را اصلاح کنند. سازندگان می‌خواهند رویکرد فنی را اصلاح کنند. مدیران می‌خواهند راهبرد پروژه را اصلاح کنند. این همه اصلاحات برای چیست؟ پاسخ در واقع بسیار ساده است. با گذشت زمان، همه‌ی گروه‌ها بیش از قبل می‌دانند (درباره آن‌چه که نیاز دارند، این که کدام رویکرد، بهترین است و چطور می‌توان کار را به پایان رساند و باز هم سود کرد). این دانش اضافی، نیروی محرکه‌ای است که در پس اکثر تغییرات قرار دارد و به بیان این واقعیت می‌انجامد که پذیرش آن برای بسیاری از دست‌اندرکاران مهندسی نرم‌افزار دشوار است: *اکثر تغییرات موجه هستند.*

اندرز

اکثر تغییرات، توجیه دارند لذا جایی برای شکایت از آن‌ها وجود ندارد. در عوض، یقین حاصل کنید که برای مواجهه با آن‌ها، سازوکارهایی وجود دارد.

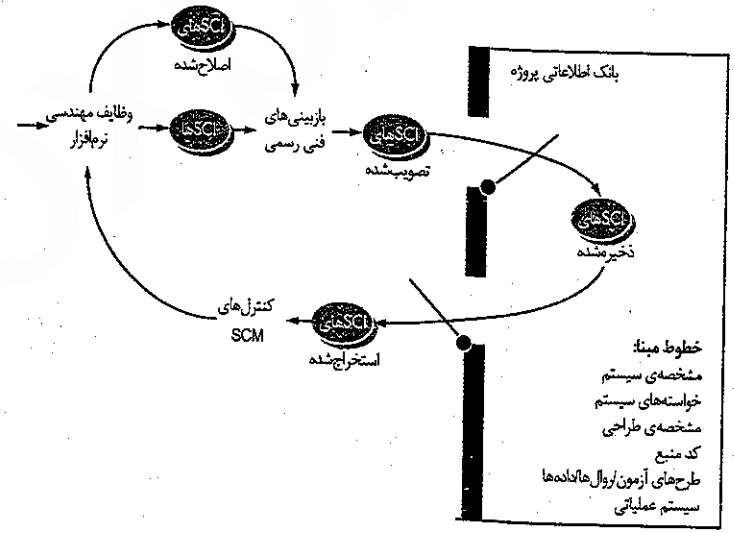
خط مبنا یک مفهوم مدیریت پیکربندی نرم‌افزار است که به کنترل تغییرات، بدون جلوگیری کردن از تغییراتِ موجه، کمک می‌کند. در استاندارد IEEE 610.12-1940 خط مبنا به صورت زیر تعریف می‌شود:

> مشخصه یا محصولی که رسماً مورد مرور و توافق قرار گرفته است و از آن پس به‌عنوان مبنایی برای توسعه بیشتر عمل می‌کند و تنها از طریق روال‌های رسمی کنترل تغییرات، قابل تغییر است.

پیش از آنکه هر یک از آیتم‌های پیکربندی به یک خط مبنا تبدیل شود، تغییر را می‌توان به‌سرعت و به‌طور غیررسمی انجام داد، ولی به مجرد تثبیت یک خط مبنا، تغییرات قابل اعمال است، ولی برای ارزیابی و اعتبارسنجی هر تغییر باید یک رویه رسمی و مشخص به اجرا گذاشته شود.

در حیطه‌ی مهندسی نرم‌افزار، خط مبنا، یک نقطه عطف در توسعه نرم‌افزار است که مشخصه آن، تحویل یک یا چند SCI و تصویب این SCIهاست که از طریق مرور فنی (فصل ۱۵) به‌دست می‌آید. برای مثال، عناصر یک مشخصه، طراحی، مستندسازی و مرور می‌شوند. خطاها کشف و تصحیح می‌شوند. هنگامی که کلیه‌ی بخش‌های مشخصه مرور شدند، تصحیح شدند و به تصویب رسیدند، مشخصه طراحی به یک خط مبنا تبدیل می‌شود. تغییرات بیشتر در معماری برنامه را (که در مشخصه‌ی طراحی، مستندسازی شده است) می‌توان فقط پس از ارزیابی و تصویب آنها اعمال نمود. گرچه خطوط مبنا را می‌توان در هر سطح از جزئیات تعریف نمود، متداول‌ترین خطوط مبنا در شکل ۱-۲۲ نشان داده شده‌اند.

پیشرفت رویدادهایی که منجر به تشکیل یک خط مبنا می‌شود نیز در شکل ۱-۲۲ نشان داده شده است. وظایف مهندسی نرم‌افزار، یک یا چند SCI تولید می‌کنند. پس از مرور و تصویب SCIها، آنها را در یک *بانک اطلاعاتی پروژه* (که *کتابخانه پروژه* یا *مخزن پروژه* نیز نامیده می‌شود و در بخش ۲-۲۲ بحث خواهد شد) قرار می‌دهند. هنگامی که عضوی از تیم پروژه یکی از SCIهای خط مبنایی را اصلاح کند، آن را از بانک اطلاعاتی پروژه به فضای کاری اختصاصی خود کپی می‌کند، ولی این SCI استخراج شده فقط در صورتی قابل اصلاح است که کنترل‌های SCM رعایت شوند (کنترل‌های SCM را بعداً در همین فصل مورد بحث قرار خواهیم داد. پیکان‌های شکل ۱-۲۲ نشان‌گر مسیر اصلاح برای یک SCI خط مبنا هستند.



شکل ۱-۲۲ SCIهای خط مبنا و بانک اطلاعاتی پروژه.

۴-۱-۲۲ آیتم‌های پیکربندی نرم‌افزار

آیتم‌های پیکربندی نرم‌افزار را پیش از این به‌عنوان اطلاعاتی تعریف کردیم که به‌عنوان بخشی از فرایند مهندسی نرم‌افزار ایجاد می‌شوند. در حالت حدی، SCI را می‌توان به‌عنوان بخش منفردی از یک مشخصه بزرگ یا یک مورد آزمون در یک مجموعه بزرگ از آزمون‌ها در نظر گرفت. در حالتی واقع‌بینانه‌تر، SCI یک سند، مجموعه کاملی از موارد آزمون یا یک قطعه برنامه‌ی با نام (مثل یک تابع C++ یا یک پکیج در Ada) است.

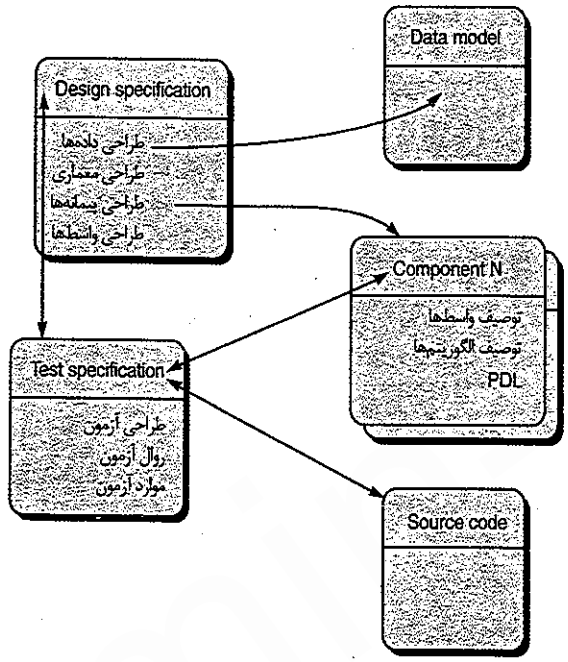
بسیاری از سازمان‌های مهندسی نرم‌افزار، علاوه بر SCIهای به‌دست آمده از محصولات کاری نرم‌افزار، ابزارهای نرم‌افزاری تحت کنترل پیکربندی را هم قرار می‌دهند. یعنی، نسخه‌های مشخصی از ویراستارها، کامپایلرها، مرورگرها و سایر ابزارهای خودکار شده به‌عنوان بخشی از پیکربندی نرم‌افزار «منجمد» می‌شوند. از آن‌جا که این ابزارها در تولید مستندات، کدهای منبع و داده‌ها به‌کار می‌روند، باید هنگامی در دسترس باشند که تغییرات در پیکربندی نرم‌افزار به عمل آمده باشد. گرچه مشکلات به ندرت پیش می‌آید، این امکان وجود دارد که نسخه‌ی جدیدی از یک ابزار (مثلاً یک کامپایلر)

اندرز

پیش از آن که کد یک بانک اطلاعاتی پروژه در مکانی متمرکز و کنترل‌شده نگهداری می‌شود

نتایجی متفاوت با نتایج نسخه‌ی اولیه به بار آورد. به همین دلیل، ابزارها، همانند نرم‌افزاری که به تولید آن کمک می‌کنند، می‌توانند به‌عنوان خط مبنا برای بخشی از یک فرایند جامع پیکربندی عمل کنند.

در واقع، SCIها برای تشکیل اشیای پیکربندی سازمان‌دهی می‌شوند که ممکن است در بانک اطلاعاتی، دارای یک نام باشند. هر شیء پیکربندی یک نام و چند صفت دارد و از طریق یک سری روابط به اشیای دیگر «متصل» است. در شکل ۲-۲۲ اشیای پیکربندی **DesignSpecification**، **DataModel**، **ComponentN**، **SourceCode** و **TestSpecification** هر یک به‌طور جداگانه تعریف شده‌اند، ولی هر یک از این اشیاء توسط چند پیکان با اشیای دیگر ارتباط دارد. پیکان خمیده نشان‌دهنده‌ی یک رابطه ترکیبی است. یعنی **DataModel** و **ComponentN** هر دو، بخشی از **DesignSpecification** هستند. پیکان دوسر نشان دهنده یک رابطه‌ی متقابل است. اگر تغییری در **SourceCode** ایجاد شود، یک رابطه متقابل، مهندس نرم‌افزار را قادر می‌سازد تا تعیین کند چه اشیا (و CMIهای) دیگری ممکن است تأثیر بپذیرند[1].



شکل ۲-۲۲ اشیای پیکربندی.

۲-۲۲ مخزن SCM

در نخستین روزهای مهندسی نرم‌افزار، آیتم‌های پیکربندی نرم‌افزار به صورت مستندات کاغذی (یا روی کارت‌های سوراخ شده!) نگهداری می‌شدند، در پوشه‌های مقوایی و در کابینت‌های فلزی قرار

[1] این روابط در داخل بانک اطلاعاتی تعریف می‌شوند ساختار این بانک اطلاعاتی (مخزن) در بخش ۲-۲۲ با جزئیات بیشتر بحث خواهد شد.

داده می‌شدند. این روش به چند دلیل، مشکل‌آفرین بود. (۱) یافتن یک آیتم پیکربندی، هنگامی که به آن نیاز بود، غالباً دشوار بود، (۲) تعیین این که کدام آیتم‌ها، چه هنگام و توسط چه کسی تغییر داده شده‌اند، غالباً ایجاد چالش می‌کرد، (۳) ایجاد نسخه‌ی جدیدی از یک برنامه موجود، وقت گیر بود و در معرض خطا قرار داشت و (۴) توصیف جزئیات یا روابط پیچیده‌ی میان آیتم‌های پیکربندی غیر ممکن بود.

امروزه، SCIها در بانک اطلاعاتی یا مخزن پروژه نگهداری می‌شوند. در فرهنگ‌های لغات، مخزن به‌عنوان «محلی برای انباشتن یا ذخیره سازی» تعریف می‌شود. طی روزهای اولیه‌ی مهندسی نرم‌افزار، مخزن در واقع یک آدم بود- برنامه‌نویسی که می‌بایست موقعیت همه‌ی اطلاعات مرتبط با پروژه نرم‌افزار را به‌خاطر بسپرد، می‌بایست اطلاعاتی را که هرگز نوشته نمی‌شد به خاطر بیاورد و اطلاعاتی را که از بین رفته بود، بازسازی کند. متأسفانه، استفاده از یک آدم به‌عنوان «مرکز انباشتن یا ذخیره سازی» چندان خوب جواب نمی‌دهد. امروزه، مخزن یک «چیز» است- یک بانک اطلاعاتی که به‌عنوان مرکزی برای انباشتن و ذخیره‌سازی اطلاعات مهندسی نرم‌افزار عمل می‌کند. نقش شخص (مهندس نرم‌افزار) تعامل با مخزن با استفاده از ابزارهای موجود در آن است.

۲۲-۲-۱ نقش مخزن

مخزن SCM، مجموعه‌ای از سازوکارها و ساختمان داده‌هاست که به تیم نرم‌افزاری این امکان را می‌دهد تا تغییرات را به شیوه‌ای اثربخش مدیریت کند. این مخزن، قابلیت‌های عملیاتی آشکار یک سیستم مدیریت بانک اطلاعاتی مدرن را با حصول اطمینان از انسجام، یکپارچگی و اشتراک داده‌ها فراهم می‌سازد. به علاوه، مخزن SCM به‌عنوان مرکزی برای انسجام بخشیدن به ابزارهای نرم‌افزاری عمل می‌کند و جریان فرایند نرم‌افزار را متمرکز کرده می‌تواند باعث یکنواختی ساختار و فرمت‌بندی محصولات کاری مهندسی نرم‌افزار شود.

برای دستیابی به این قابلیت‌ها، مخزن بر حسب یک شبه‌مدل تعریف می‌شود. این شبه مدل، چگونگی ذخیره‌سازی اطلاعات در مخزن، چگونگی دستیابی ابزارها به داده‌ها و مشاهده‌ی آن‌ها توسط مهندسان نرم‌افزار، چگونگی نگهداری و حفظ امنیت و یکپارچگی و میزان سهولت بسط مدل موجود برای پاسخ گویی به نیازهای جدید تعیین می‌شود.

۲۲-۲-۲ محتوا و ویژگی‌های عمومی مخزن

محتوا و ویژگی‌های مخزن را می‌توان به بهترین وجه با نگریستن به آن از دو دیدگاه شناخت: (۱) آن‌چه که قرار است در مخزن انباشته گردد و (۲) خدمات ویژه‌ای که توسط مخزن فراهم خواهد آمد. جزئیات انواع روابط، مستندات و سایر محصولات کاری که در مخزن انباشته می‌شوند، در شکل ۲۲-۳ نشان داده شده است.

یک مخزن پرقدرت، دو دسته خدمات ارائه می‌دهد: (۱) همان انواع خدماتی که ممکن است از هر سیستم مدیریت بانک اطلاعاتی پیچیده‌ای انتظار رود، و (۲) خدماتی که مختص محیط مهندسی نرم‌افزار است.

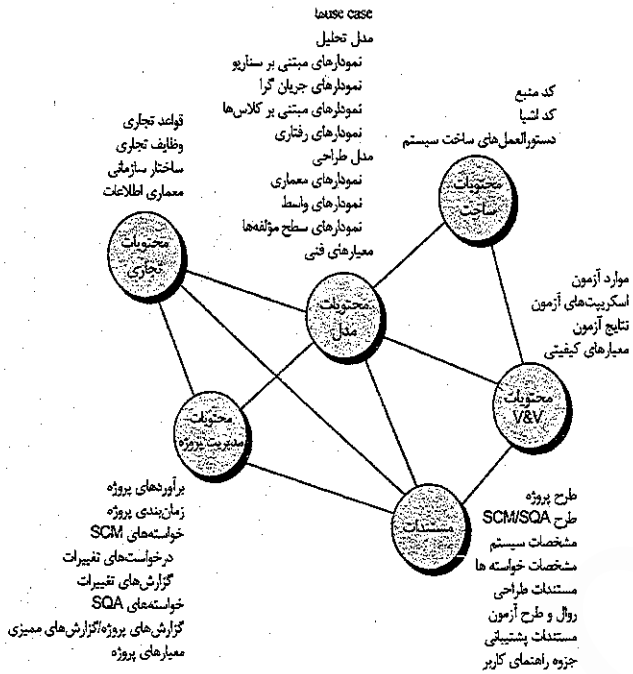
مخزنی که به تیم مهندسی نرم‌افزار خدمات می‌دهد، همچنین باید (۱) با قابلیت‌های مدیریت فرایند پشتیبانی منسجم باشند یا مستقیماً آن‌ها را پشتیبانی کند، (۲) قواعد ویژه‌ای را که بر عملکرد

مرجع وب

نمونه‌ای از یک مخزن تجاری را می‌توانید در آدرس زیر به‌دست آورید:

www.oracle.com/technology/products/repository/index.html

SCM و داده‌های موجود در مخزن حاکم هستند، پشتیبانی کند، (۳) فراهم ساختن واسطی با سایر ابزارهای مهندسی نرم‌افزار و (۴) ذخیره‌سازی اشیای داده‌ای پیچیده (مانند متون، تصاویر گرافیکی، تصاویر ویدیویی و فایل‌های صوتی).



شکل ۲۲-۳ محتوای مخزن.

۲۲-۲-۳ ویژگی‌های SCM

برای پشتیبانی SCM، مخزن باید مجموعه ابزارهایی داشته باشد که ویژگی‌های زیر را پشتیبانی کنند:

ایجاد نسخه‌ها. به موازاتی که پروژه پیش می‌رود، نسخه‌های فراوان (بخش ۲۲-۳-۲) از تک تک محصولات کاری، ایجاد خواهد شد. مخزن باید بتواند همه‌ی این نسخه‌ها را ذخیره کند تا مدیریت اثربخش ویرایش‌های محصول، امکان‌پذیر شود و سازندگان بتوانند طی انجام وظایف آزمون و اشکال‌زدایی به نسخه‌های پیشین بازگردند.

مخزن باید قادر به کنترل گستره وسیعی از انواع اشیا، از جمله متون، تصاویر گرافیکی، مستندات پیچیده و اشیای منحصر به فرد نظیر تعاریف صفحه نمایش و گزارش‌ها، فایل‌های اشیا، داده‌های آزمون و نتایج آزمون باشد. یک مخزن کامل، نسخه‌های اشیا را با سطوح دلخواهی از دانه‌بندی (granularity) ردگیری می‌کند؛ برای مثال، یک تعریف منفرد از داده‌ها یا خوشه‌ای از پیمانه‌ها را می‌توان ردگیری کرد.

نکته‌ی کلیدی

مخزن باید قادر به حفظ SCIهای مربوط به چندین نسخه‌ی متفاوت از نرم‌افزار باشد. مهم‌تر این که باید سازوکارهایی برای موثر کردن این SCIها و ایجاد یک پیکربندی خاص نسخه‌ها فراهم سازد.

مدیریت تغییرات و ردگیری وابستگی‌ها. مخزن، گستره وسیعی از روابط میان عناصر داده‌ای ذخیره‌شده در خودش را مدیریت می‌کند. این‌ها شامل روابط میان فرایندها و موجودیت‌های شرکتی، میان بخش‌های یک طراحی کاربرد، میان مؤلفه‌های طراحی و معماری اطلاعات شرکتی، میان عناصر طراحی و محصولات قابل تحویل و غیره می‌شود. برخی از این روابط، فقط از نوع وابستگی و برخی دیگر از نوع اجباری هستند.

توانایی ردگیری همه‌ی این روابط در انسجام اطلاعات ذخیره شده در مخزن و در ایجاد محصولات قابل تحویل مبتنی بر آن، اهمیت حیاتی دارد و یکی از مهم‌ترین سهم‌هایی است که مفهوم مخزن در بهبود بخشیدن به فرایند نرم‌افزار دارد. برای مثال، اگر یک نمودار کلاس‌های UML اصلاح شود، مخزن می‌تواند تشخیص دهد که آیا کلاس‌های مرتبط، توصیف‌های واسط، و مؤلفه‌های کد نیز نیاز به اصلاح دارند و می‌توانند SCI تأثیر پذیرفته را مورد توجه سازنده قرار دهند یا خیر.

ردگیری خواسته‌ها. این وظیفه‌ی خاص به مدیریت پیوندها مربوط می‌شود و توانایی ردگیری کلیه‌ی مؤلفه‌های طراحی و ساخت و محصولات قابل تحویلی را که از یک مشخصه‌ی خواسته‌ها نتیجه می‌شوند، فراهم می‌سازد (ردگیری رو به جلو). به علاوه، به کمک آن می‌توان تعیین کرد کدام خواسته‌ها، کدام محصول کاری را ایجاد کرده است (ردگیری رو به عقب).

مدیریت پیکربندی. یک تسهیلات مدیریت پیکربندی، اطلاعات پیکربندی‌هایی را نگهداری می‌کند که تولید ویرایش جدید محصول یا نقاط عطف پروژه را نشان می‌دهد.

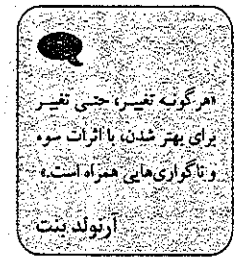
جلسات ممیزی. در جلسه ممیزی، اطلاعات اضافی درباره زمان، علت و عامل انجام دهنده‌ی تغییرات فراهم می‌آید. اطلاعات مربوط به منبع تغییرات را می‌توان به‌عنوان صفات اشیای خاص در مخزن وارد کرد. یک سازوکار آغازگر مخزن می‌تواند در پیام دادن به سازنده یا ابزار مورد استفاده کمک کند تا واردکردن اطلاعات ممیزی (از قبیل دلیل تغییر) به هنگام اصلاح یک عنصر طراحی آغاز شود.

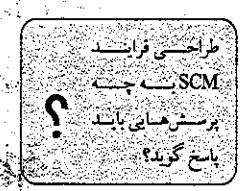
۲۲-۳ فرایند SCM

فرایند مدیریت پیکربندی نرم‌افزار، وظایفی را تعریف می‌کند که چهار هدف اصلی را دنبال می‌کنند: (۱) شناسایی همه‌ی آیتم‌هایی که در مجموع، پیکربندی نرم‌افزار را تعریف می‌کنند، (۲) مدیریت تغییرات به عمل آمده در یک یا چندتا از این آیتم‌ها، (۳) تسهیل در ایجاد نسخه‌های متفاوتی از یک برنامه کاربردی و (۴) حصول اطمینان از این که کیفیت نرم‌افزار با تکامل پیکربندی به مرور زمان حفظ می‌شود.

فرایندی که این اهداف را دنبال می‌کند، ضرورتی ندارد که چندان سنگین و رسمی و اداری باشد، بلکه باید به‌شیوه‌ای مشخص شود که تیم نرم‌افزاری را قادر سازد تا پاسخ مجموعه پرسش‌های پیچیده‌ی زیر را بدهد:

- تیم نرم‌افزاری عناصر مجزای پیکربندی یک نرم‌افزار را چگونه شناسایی می‌کند؟
- سازمان چگونه نسخه‌های متعدد یک برنامه (و مستندات آن) را شناسایی و اداره می‌کند، طوری که بتوان تغییرات را به‌طور مؤثر اِعمال کرد؟
- سازمان چگونه تغییرات را قبل و بعد از ارائه نرم‌افزار به مشتری کنترل می‌کند؟
- چه کسی مسؤول به تصویب رساندن تغییرات و اولویت‌بندی آنهاست؟

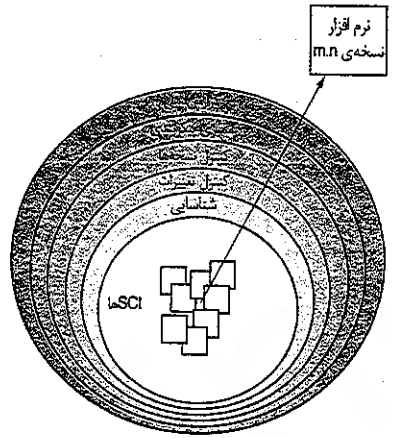




- چگونه می‌توان اطمینان حاصل کرد که تغییرات به‌طور مناسب اِعمال شده است؟
- چه سازوکاری برای آگاه ساختن دیگران از اِعمال تغییرات به‌کار می‌رود؟

این پرسش‌ها ما را به تعیین پنج وظیفه SCM رهنمون می‌شوند: شناسایی، کنترل نسخه، کنترل تغییرات، ممیزی پیکربندی و گزارش‌دهی(شکل ۴-۲۲).

با رجوع به شکل، مشاهده می‌کنید که وظایف SCM را می‌توان به صورت لایه‌های هم مرکز در نظر گرفت. SCIها از میان این لایه‌ها در سرتاسر حیات مفید خود به طرف بیرون جریان می‌یابند و سرانجام به بخشی از پیکربندی یک یا چند نسخه از یک سیستم یا برنامه کاربردی تبدیل می‌شوند. با عبور SCI از یک لایه، کنش‌هایی که هر وظیفه‌ی SCM به آن‌ها اشاره دارد، ممکن است قابل استفاده باشند و ممکن هم هست که نباشند. برای مثال، هنگامی که یک SCI جدید ایجاد می‌گردد، باید شناسایی شود. به هر حال، اگر هیچ تغییراتی برای SCI درخواست نشود، لایه کنترل تغییرات، کاربردی ندارد. SCI به نسخه‌ی خاصی از نرم‌افزار نسبت داده می‌شود (سازوکارهای کنترل نسخه وارد صحنه می‌شوند). سابقه‌ای از SCI (نام، تاریخ ایجاد، شماره نسخه و غیره) برای اهداف ممیزی پیکربندی نگهداری می‌شود و به آن‌هایی که نیاز دارند، گزارش داده می‌شود. در بخش‌هایی که به دنبال خواهد آمد، هر کدام از لایه‌های فرایندی SCM را با تفصیل بیشتر مطالعه خواهد کرد.



شکل ۴-۲۲ لایه‌های فرایند SCM.

۲۲-۳-۱ شناسایی اشیاء در پیکربندی نرم‌افزار

برای کنترل و اداره آیتم‌های پیکربندی نرم‌افزار، هر یک از آنها را باید جداگانه نامگذاری و سپس با استفاده از روشی شیءگرا سازمان‌دهی کرد. دو نوع از اشیا قابل شناسایی است [Cho89]: اشیای پایه و اشیای مرکب[1]. شیء پایه یک «واحد متنی» است که در اثنای تحلیل، طراحی، کدنویسی یا آزمون ایجاد شده است. برای مثال، یک شیء پایه ممکن است بخشی از مشخصه خواسته‌ها، کد مربوط به یک مؤلفه، یا مجموعه‌ای از موارد آزمون باشد که برای تمرین با کد به‌کار می‌رود. شیء مرکب

[1] مفهوم شیء مرکب، [Gus89] به‌عنوان سازوکاری برای نمایش نسخه کاملی از پیکربندی نرم‌افزار پیشنهاد شده است.

مجموعه‌ای از اشیای پایه و اشیای مرکب دیگر است. برای مثال، **DesignSpecification** یک شیء مرکب است. از نظر مفهومی، می‌توان آن را به‌عنوان یک فهرست با نام (شناسایی شده) از اشاره‌گرهایی در نظر گرفت که اشیای مرکبی نظیر **ArchitecturalModel** و **DataModel** و اشیای پایه‌ای نظیر **UMLClassDiagramN** و **ComponentN** را مشخص می‌سازد.

هر شیء دارای مجموعه‌ای از ویژگی‌های متمایز است که آن را مشخص می‌سازد: نام، توصیف، فهرستی از منابع، و یک «عینیت‌بخشی». نام شیء یک رشته کاراکتری است که شیء را بدون هیچ ابهامی مشخص می‌کند. توصیف شیء، فهرستی از آیتم‌های داده‌ای است که موارد زیر را مشخص می‌کند: نوع SCI (مثل سند، برنامه یا داده‌ها) که توسط شیء نشان داده می‌شود؛ شناسه‌ی پروژه؛ اطلاعات مربوط به تغییر و/یا نسخه. منابع، « موجودیت‌هایی هستند که توسط شیء فراهم می‌شوند، پردازش می‌شوند، ارجاع داده می‌شوند، یا موردنیاز شیء هستند» [Cho89]. به‌عنوان مثال، انواع داده‌ها، توابع مشخص یا حتی نام متغیرها را می‌توان در ردیف منابع اشیا دانست. عینیت‌بخشی، اشاره‌گری به «واحد متن» برای یک شیء پایه و مقدار صفر برای یک شیء مرکب است.

در شناسایی شیء پیکربندی، روابط میان اشیای با نام را نیز باید در نظر گرفت. برای مثال، با استفاده از نمادگذاری ساده زیر، می‌توانید سلسله مراتبی از SCIها را تشکیل دهید:

Class diagram <part-of> requirements model;
requirements model diagram <part-of> requirements Specification

در بسیاری از موارد، میان شاخه‌های این درخت نیز روابطی برقرار است. این روابط ساختاری را به‌شیوه‌ی زیر می‌توان نشان داد:

DataModel < > DataFlowModel
DataModel < > TestCaseClassM

در مورد اول، ارتباط داخلی بین یک شیء مرکب برقرار است، حال آنکه در مورد دوم ارتباط میان یک شیء مرکب (**DataModel**) و یک شیء پایه (**TestCaseClassM**) برقرار است.

در طرح شناسایی برای اشیای نرم‌افزار، باید توجه داشت که اشیاء در سرتاسر فرایند نرم‌افزار تکامل می‌یابند. هر شیء پیش از تبدیل به خط مبنا ممکن است بارها تغییر کند و حتی پس از تثبیت یک خط مبنا، ممکن است تغییرات فراوانی رخ دهد.

۲۲-۳-۲ کنترل نسخه‌ها (Version Control)

کنترل نسخه‌ها، برای مدیریت نسخه‌های گوناگون اشیای پیکربندی که طی فرایند نرم‌افزار ایجاد می‌شوند، روال‌ها و ابزارهایی را با هم ترکیب می‌کند. سیستم کنترل نسخه‌ها، چهار قابلیت اصلی را پیاده‌سازی می‌کند: (۱) یک بانک اطلاعاتی (مخزن) پروژه که کلیه اشیای پیکربندی مرتبط را ذخیره می‌کند، (۲) یک قابلیت *مدیریت نسخه‌ها* که همه‌ی نسخه‌های یک شیء پیکربندی را ذخیره می‌کند (یا ساخت هر نسخه را با استفاده از اختلاف‌های نسخه‌های قبلی میسر می‌سازد)، (۳) یک تسهیلات ساخت‌وساز که به کمک آن می‌توانید همه‌ی اشیای پیکربندی مرتبط را جمع‌آوری کنید و نسخه‌ی خاصی از نرم‌افزار را ایجاد کنید. به علاوه، سیستم‌های کنترل نسخه‌ها و کنترل تغییرات، غالباً قابلیت *ردگیری مسائل* (یا همان *ردگیری اشکال‌ها*) را پیاده‌سازی می‌کنند که تیم به کمک آن می‌تواند وضعیت کلیه‌ی مسائل برجسته‌ی مرتبط با هر شیء پیکربندی، ثبت و ردگیری کند.

نکته‌ی کلیدی
به کمک روابط وضع‌شده، برای اشیای پیکربندی می‌توانید تأثیر تغییرات را بسنجید.

اندرز
حتی اگر بانک اطلاعاتی پروژه توانایی برقراری این روابط را در اختیار قرار دهد، برقراری این روابط و بهنگام نگه داشتن آن‌ها کاری وقت‌گیر است. گرچه برای مدیریت کلی تغییرات ضروری نیستند، برای تحلیل تأثیرات ناشی از این تغییرات بسیار مفید واقع می‌شوند.

چند سیستم کنترل نسخه، یک *مجموعه‌ی تغییرات* را تشکیل می‌دهند- مجموعه‌ای از کلیه تغییرات (که روی یک پیکربندی خط مبنا اعمال می‌شوند) و برای ایجاد نسخه‌ی مشخصی از نرم‌افزار مورد نیازند. دارت [Dar91] خاطر نشان می‌سازد که هر مجموعه تغییرات «همه‌ی تغییرات به عمل آمده در تمامی فایل‌های موجود در پیکربندی و نیز دلیل اعمال این تغییرات و جزئیات مربوط به عامل این تغییرات و زمان آن را در بر می‌گیرد.» برای هر سیستم یا برنامه کاربردی، ممکن است چند مجموعه تغییرات با نام‌های مشخص شناسایی شود. به این ترتیب می‌توانید نسخه‌ای از نرم‌افزار را با مشخص‌کردن چند مجموعه تغییرات (که هر یک نام مشخصی دارد) ایجاد کنید که باید در پیکربندی خط مبنا اعمال شوند. برای دستیابی به این هدف، از رویکرد *مدل‌سازی* سیستمی استفاده می‌شود. مدل سیستمی حاوی این موارد است: (۱) *قالبی* که شامل یک سلسله مراتب از مؤلفه‌ها و یک «سفارش ساخت» برای مؤلفه‌هایی می‌شود که شرح می‌دهد سلسله مراتب چگونه باید ایجاد شود، (۲) قواعد ساخت و (۳) قواعد وارسی[1].

طی چند دهه‌ی گذشته، چند روش خودکار متفاوت برای کنترل نسخه‌ها پیشنهاد شده است. اختلاف اصلی این روش‌ها در پیچیدگی صفاتی که برای ایجاد نسخه‌ها و تنوع‌های مشخصی از یک سیستم به‌کار می‌روند، و نیز مکانیک فرایند ساخت است.

ابزارهای نرم‌افزاری

سیستم نسخه‌های همروند (CVS)

استفاده از ابزارها برای دستیابی به کنترل نسخه‌ها برای مدیریت اثربخش تغییرات، اهمیت اساسی دارد. سیستم نسخه‌های همروند (CVS) یک ابزار پرکاربرد برای کنترل نسخه‌هاست. سیستم CVS که در ابتدا برای کدهای منبع طراحی شده بود، برای هر فایل متنی مفید است و (۱) یک مخزن ساده ایجاد می‌کند، (۲) همه‌ی نسخه‌های یک فایل را با ذخیره‌سازی اختلاف‌های میان نسخه‌های تدریجی فایل اولیه در فایلی با نام مشخص حفظ می‌کند و (۳) با تعیین دایرکتوری‌های متفاوت برای هر توسعه‌دهنده، از اِعمال تغییرات همزمان روی یک فایل جلوگیری می‌کند. CVS تغییرات را پس از کامل شدن کار همه‌ی توسعه‌دهندگان در هم ادغام می‌کند.

شایان ذکر است که CVS یک سیستم «ساخت و ساز» نیست؛ یعنی نسخه‌ی مشخصی از نرم‌افزار را ایجاد نمی‌کند. ابزارهای دیگری (مانند Makefile) باید با CVS منسجم شوند تا این منظور نیز برآورده شود. CVS یک فرایند کنترل تغییرات (مانند درخواست‌های تغییر، گزارش‌های تغییر، ردگیری اِشکال‌ها) را پیاده‌سازی نمی‌کند.

CVS حتی با وجود این محدودیت‌ها نیز یک سیستم کنترل نسخه‌های غالب و متن‌باز (open source) است که در شبکه شفاف است و برای هر کسی از تک تک توسعه‌دهندگان گرفته تا تیم‌های بزرگ و توزیع شده مفید واقع می‌شود» [CVS07]. معماری کلاینت-سرور آن به کاربران این امکان را می‌دهد تا از طریق اتصال‌های اینترنتی به فایل‌ها دست پیدا کنند و کدباز بودن آن باعث می‌شود که روی اکثر سکوهای پرطرفدار، در دسترس باشد.

CVS به‌طور رایگان برای محیط‌های Windows، Mac OS، LINUX و UNIX در دسترس است. برای جزئیات بیشتر، [CVS07] را ببینید.

[1] امکان درخواست وضعیت از مدل سیستمی برای ارزیابی چگونگی تأثیرگذاری یک مؤلفه بر سایر مؤلفه‌ها نیز وجود دارد.

۳-۳-۲۲ کنترل تغییرات

واقعیتِ کنترل تغییرات در حیطه مهندسی نرم‌افزار نوین را جیمز بَک به‌طرزی زیبا خلاصه کرده است [Bac98]:

> کنترل تغییرات حیاتی است، ولی نیروهایی که آن را ضروری می‌سازند، آن را خسته کننده هم می‌کنند. ما نگران تغییرات هستیم زیرا یک اختلال جزئی در کد می‌تواند شکستی عظیم در محصول ایجاد کند، ولی می‌تواند یک شکست بزرگ را نیز برطرف کند یا باعث ایجاد قابلیت‌های جالب و جدیدی شود. ما از آن رو نگران تغییرات هستیم که یک نرم‌افزارنویس ناشی می‌تواند پروژه را معدوم کند؛ با این حال، ایده‌های درخشان در ذهن همین ناشی نقش می‌بندند و یک فرایند تحتِ کنترل تغییرات ممکن است آنها را از انجام‌دادن کارهای خلاقانه باز دارد.

یک تشخیص داده است که باید متعادل رفتار کنیم. اگر بیش از حد به کنترل تغییرات بپردازیم، مشکل ایجاد می‌کنیم و اگر کم باشد، مشکلات دیگری به بار خواهد آمد.

برای یک پروژه مهندسی نرم‌افزار بزرگ، تغییرات کنترل نشده به سرعت منجر به آشوب خواهد شد. در چنین پروژه‌هایی، کنترل تغییرات، رویه‌های بشری و ابزارهای خودکار را با هم ترکیب کرده سازوکاری برای کنترل تغییرات فراهم می‌آورند. فرایند کنترل تغییرات به‌طور شماتیک در شکل ۵-۲۲ نشان داده شده است. *درخواست تغییر*، پس از تسلیم مورد ارزیابی قرار می‌گیرد تا شایستگی فنی، اثرات جانبی بالقوه، تأثیر کلی آن بر دیگر اشیای پیکربندی و عملکردهای سیستم سنجیده شود. نتایج این ارزیابی به صورت یک *گزارش تغییر* ارائه می‌شود که مورد استفاده *مسؤول کنترل تغییر* (CCA) قرار می‌گیرد - یعنی شخص یا گروهی که درباره وضعیت و اولویت تغییر، تصمیم نهایی را اتخاذ می‌کند. برای هر تغییر مصوب، یک *سفارش تغییر مهندسی* (ECO) تولید می‌شود. ECO تغییری را که باید اعمال شود؛ شرایط حدی که باید رعایت شوند، و ملاک‌های مرور و ممیزی را توصیف می‌کند.

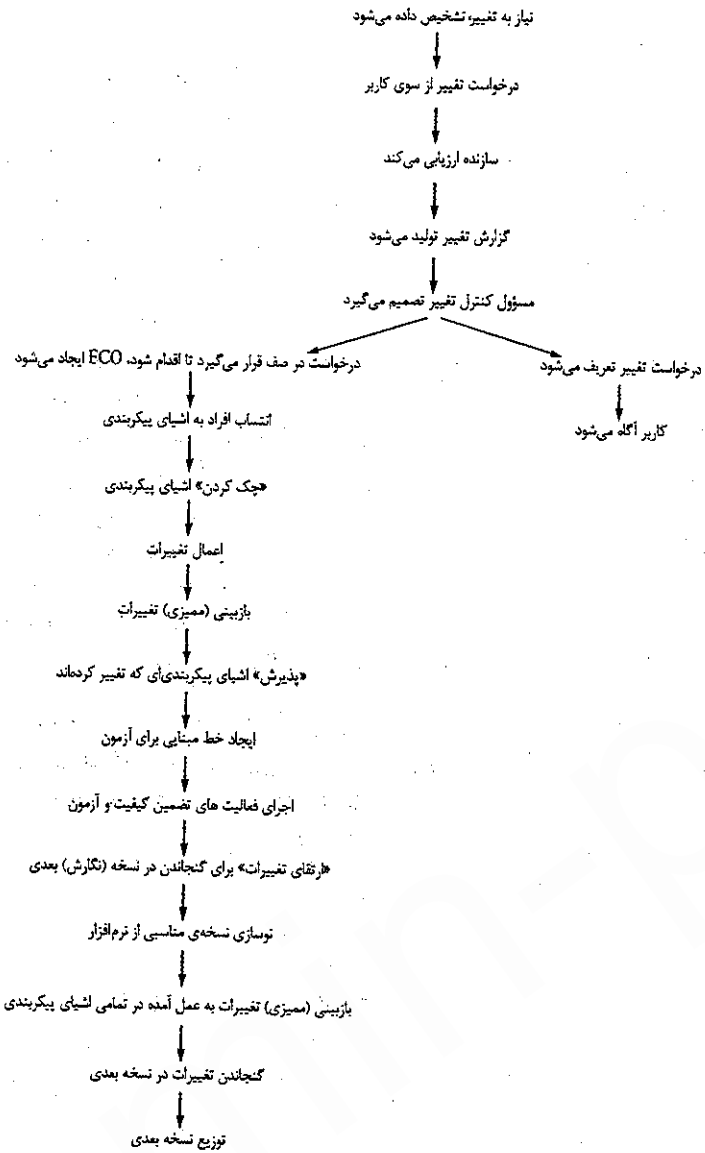
نکته‌ی کلیدی

لازم به ذکر است که چند درخواست تغییر را می‌توان با هم ترکیب کرد و تنها یک ECO به دست آورد و ECO-ها معمولاً به تغییرات در چند شیء پیکربندی می‌شوند.

اشیایی که قرار است تغییر داده شوند، در دایرکتوری‌ای قرار داده می‌شوند که فقط توسط مهندس نرم‌افزاری که تغییر را اعمال می‌کند، قابل کنترل است. یک سیستم کنترل نسخه‌ها (کادر مربوط به CVS را ببینید) پس از اعمال تغییر، فایل اولیه را بهنگام‌سازی می‌کند. به‌عنوان یک راه دیگر، شیء (هایی) را که باید تغییر داده شوند، می‌توان از بانک اطلاعاتی (مخزن) پروژه «خارج کرده»، تغییر را اعمال کرد و فعالیت‌های SQA مناسب را اعمال نمود. سپس این شیء(ها) دوباره وارد بانک اطلاعاتی می‌شوند و برای ایجاد نسخه‌ی بعدی نرم‌افزار از سازوکارهای مناسبی برای کنترل نسخه‌ها (بخش ۲-۳-۲۲) استفاده می‌شود.

این سازوکارهای کنترل نسخه‌ها، که در فرایند کنترل تغییرات منسجم می‌شوند، دو عنصر مهم از مدیریت تغییرات را پیاده‌سازی می‌کنند- کنترل دستیابی و کنترل همزمان سازی. *کنترل دستیابی* تعیین می‌کند که کدام مهندسان نرم‌افزار اجازه دستیابی به یک شیء پیکربندی خاص و اصلاح آن را دارند. به کمک *کنترل همگام‌سازی* می‌توان اطمینان حاصل کرد که تغییرات موازی و انجام شده توسط دو نفر متفاوت، روی یکدیگر نوشته نمی‌شوند.

ممکن است از این همه کاغذبازی که کنترل تغییر به دنبال دارد (شکل ۵-۲۲)، احساس ناراحتی کنید. این احساس عادی است. بدون تدابیر امنیتی مناسب، کنترل تغییر می‌تواند باعث تشریفات زائد بشود.



شکل ۵-۲۲ فرایند کنترل تغییرات.

اکثر نرم‌افزارنویسانی که دارای سازوکارهای کنترل تغییرات هستند، چند لایه کنترلی پدید آورده‌اند که به پرهیز از مشکلات ذکر شده در بالا کمک می‌کند.

پیش از تبدیل یک SCI به خط مبنا، فقط به *کنترل غیررسمی* تغییر نیاز است. سازنده‌ی شیء پیکربندی موردنظر، ممکن است هرگونه تغییراتی را که مورد تأیید خواسته‌های فنی و پروژه باشد،

اعمال کند (تا هنگامی که این تغییرات بر خواسته‌های وسیع‌تری که در خارج از دامنه کاری نرم‌افزارنویس قرار دارند، تأثیری نداشته باشد). هنگامی که شیء دستخوش مرور فنی رسمی شد و به تصویب رسید، یک خط مبنا ایجاد می‌شود[1]. هنگامی که SCI به یک خط مبنا تبدیل شد، کنترل تغییر سطح پروژه به اجرا درمی‌آید. اکنون، برای اعمال تغییر، نرم‌افزارنویس باید نظر مثبت مدیر پروژه را تأمین کند (اگر تغییر «محلی» است) یا آن را به تصویب CCA برساند (اگر تغییر بر SCIهای دیگر تأثیر می‌گذارد). در برخی موارد، تولید رسمی درخواست‌های تغییرات، گزارش‌های تغییرات و ECOها مستثنی می‌شود. ولی، همه‌ی تغییرات، مورد سنجش، پیگیری و مرور قرار می‌گیرند. هنگامی که محصول نرم‌افزاری به مشتری عرضه شد، کنترل رسمی تغییرات نهادینه می‌شود. رویه‌ی کنترل و تغییرات در شکل ۵-۲۲ ترسیم شده است.

اندرز

خود را برای قدری تغییرات، پیش از آن چه در ذهن دارید، آماده کنید. احتمالاً مقدار درست، بیش از حد است.

مسؤول کنترل تغییرات (CCA) در لایه‌های دوم و سوم کنترل، نقش فعالی دارد. بسته به اندازه و ویژگی‌های پروژه نرم‌افزاری، CCA ممکن است از یک نفر - مدیر پروژه - یا چند نفر (مثلاً نماینده‌هایی از طرف مهندسی نرم‌افزار، سخت‌افزار، بانک اطلاعاتی، پشتیبانی، بازاریابی و غیره) تشکیل شود. هدف CCA به‌دست آوردن یک دید کلی و سرتاسری است، یعنی ارزیابی تأثیر تغییرات در ورای SCI موردنظر. تغییر چه تأثیری بر سخت‌افزار دارد؟ چه تأثیری بر کارایی دارد؟ درک مشتری از محصول را چگونه اصلاح می‌کند؟ کیفیت و قابلیت اطمینان محصول چگونه تحت‌تأثیر قرار می‌گیرند؟ CCA باید این پرسش‌ها و بسیاری از پرسش‌های دیگر را پاسخ دهد.

> «تغییر، اجتناب‌ناپذیر است جز برای ماشین‌های فروش کالا.»
>
> یک برچسب تجاری

۴-۳-۲۲ ممیزی پیکربندی (Configuration Audit)

شناسایی، کنترل نسخه و کنترل تغییرات به نرم‌افزارنویس کمک می‌کند تا نظم امور را حفظ کند ولی، حتی مؤثرترین سازوکارهای کنترل نیز یک تغییر را تا جایی پیکربندی می‌کنند که یک ECOتولید شود. چگونه می‌توان اطمینان یافت که تغییر به‌طور مناسب پیاده‌سازی شده است؟ پاسخ دو وجهی است: (۱) مرورهای فنی رسمی و (۲) ممیزی پیکربندی نرم‌افزار.

مرور فنی رسمی (که به تفصیل در فصل ۸ بحث شد) بر درستی فنی شیء پیکربندی اصلاح شده تأکید دارد. مسؤولان مرور، SCI را مورد سنجش قرار می‌دهند تا سازگاری با SCIهای دیگر، جزئیات یا اثرات جانبی دیگر را تعیین کنند. همه‌ی تغییرات را باید مورد مرور فنی رسمی قرار داد، مگر آن دسته از تغییراتی که بسیار کم اهمیت هستند.

در *ممیزی پیکربندی نرم‌افزار*، مرور فنی رسمی با سنجش یک شیء پیکربندی برای ویژگی‌هایی صورت می‌پذیرد که عموماً طی مرور در نظر گرفته نمی‌شوند. ممیزی، سؤالات زیر را مطرح کرده پاسخ آنها را پیدا می‌کند:

۱. آیا تغییر مشخص شده در ECO اعمال شده است؟ آیا اصلاحی صورت پذیرفته است؟
۲. آیا مرور فنی رسمی برای سنجش صحت فنی اجرا شده است؟
۳. آیا فرایند نرم‌افزار دنبال شده است و استانداردهای مهندسی نرم‌افزار به‌طور مناسب به اجرا درآمده‌اند؟

[1] ایجاد خط مبنا دلایل دیگری نیز می‌تواند داشته باشد. برای مثال، هنگامی که «ساخت‌های روزانه» ایجاد می‌شوند، همه‌ی مؤلفه‌های پذیرفته‌شده در زمانی خاص، به خط مبنایی برای روز بعد تبدیل شده باشند.

SafeHome

مسائل SCM

صحنه: دفتر داگ میلر، در شروع پروژه نرم‌افزار *SafeHome*

بازیگران: داگ میلر (مدیر تیم مهندسی نرم‌افزار *SafeHome*) و وینود رامان، جیمی لازار و سایر اعضای تیم مهندسی نرم‌افزار محصول.

گفتگو:

داگ: می‌دانم که زود است، ولی باید درباره مدیریت تغییرات صحبت کنیم.

وینود (با خنده): خیلی هم زود نیست. همین امروز صبح، از بازاریابی تماس گرفتند و چند تا فکر جدید داشتند، چیز مهمی نبود، ولی این شروع کار است.

جیمی: ما در پروژه‌های قبلی، مدیریت تغییرات را خیلی غیر رسمی جلو بردیم.

داگ: می‌دانم، ولی این پروژه بزرگ‌تر است، بیشتر توی چشم است و آن طور که یادم هست...

وینود (سری تکان می‌دهد): در پروژه کنترل روشنایی منزل، تغییرات کنترل نشده امان ما را بریده بود... یادت هست... تأخیرهایی که...

داگ (با اخم): کابوسی که ترجیح می‌دهم فراموش کنم.

جیمی: پس می‌خواهی چه کار کنی؟

داگ: آن‌طوری که من می‌بینم، سه تا کار. اول باید یک فرایند کنترل تغییرات تهیه کنیم یا قرض بگیریم.

جیمی: منظورت، شیوه‌ی درخواست تغییرات است؟

وینود: بله، و البته این که چطور تغییرات را ارزیابی کنیم، تصمیم بگیریم که چه زمانی آن‌ها را اعمال کنیم (اگر به اعمال آن تصمیم گرفته باشیم) و چطور از چیزهایی که از آن تغییر تأثیر گرفته اند، سوابقی را تهیه کنیم.

داگ: دوم باید یک ابزار SCM واقعاً خوب برای کنترل تغییرات و نسخه‌ها داشته باشیم.

جیمی: می‌توانیم برای همه‌ی محصولات کاری یک بانک اطلاعاتی بسازیم.

وینود: در این حیطه به آن SCI می‌گویند و اکثر ابزارهای خوب، آن را به نحوی پشتیبانی می‌کنند.

داگ: این نقطه‌ی شروع خوبی است و حالا ما باید...

جیمی: داگ، تو گفتی سه تا کار...

داگ (با لبخند): سوم- همه‌ی ما باید متعهد شویم که فرایند مدیریت تغییرات را دنبال کنیم و از ابزارها استفاده کنیم.

۴. آیا تغییر در SCI «برجسته» شده است؟ آیا داده‌های تغییر و صاحب تغییر مشخص شده است؟ آیا صفات شیء پیکربندی، این تغییر را منعکس می‌کنند؟
۵. آیا روال‌های SCM برای ذکر تغییر، ثبت آن و گزارش آن دنبال شده‌اند؟
۶. آیا همه‌ی SCIهای مربوط به‌طور مناسب بهنگام‌سازی شده‌اند؟

در برخی موارد، پرسش‌های ممیزی به‌عنوان بخشی از مرور فنی رسمی پرسیده می‌شوند، ولی هنگامی که SCM یک فعالیت رسمی است، ممیزی SCM به‌طور جداگانه توسط گروه تضمین کیفیت اجرا می‌شود. با این گونه ممیزی‌های رسمی پیکربندی، می‌توان اطمینان یافت که SCIهای صحیح (از نظر شماره نسخه) در یک ساخت مشخص قرار می‌گیرند و همه‌ی مستندات، به‌هنگام بوده با نسخه‌ای که ساخته می‌شود، سازگاری دارند.

ابزارهای نرم‌افزاری

پشتیبانی SCM

هدف: ابزارهای SCM یک یا چند مورد از فعالیت‌های فرایندی بحث شده در بخش ۳-۲۲ را پشتیبانی می‌کند.

مکانیک: اکثر ابزارهای SCM مدرن در ارتباط با یک مخزن (یک سیستم بانک اطلاعاتی) کار می‌کنند و سازوکارهایی برای شناسایی، کنترل تغییرات و نسخه‌ها، ممیزی و گزارش دهی فراهم می‌آورند.

ابزارهای نمونه

CCC/Harvest، که توسط Computer Associates (www.cai.com) توسعه یافته است، یک سیستم SCM چند سکویی است.

Clear Case، که توسط Rationale توسعه یافته است، گروهی از قابلیت‌های SCM را فراهم می‌آورد (www.306.ibm.com/software/awdtools/clearcase/index.html).

Serena Change Man ZMF که توسط Serena توزیع شده است مجموعه کاملی از ابزارهای SCM را فراهم می‌آورد که هم برای نرم‌افزارهای سنتی و هم برای برنامه‌های تحت وب قابل استفاده است (www.serena.com/US/prooducts/zmf/index.aspx).

Source Forge که توسط VA Software (sourceforge.net) توزیع می‌شود، مدیریت نسخه‌ها، قابلیت‌های ساخت و ساز، ردگیری مسائل/ اشکال‌ها و بسیاری ویژگی‌های مدیریتی را فراهم می‌سازد.

Surround SCM که توسط Seapine Software توسعه یافته است و قابلیت‌های کامل مدیریت تغییرات را فراهم می‌سازد (www.seapine.com).

Vesta، که توسط Compac توزیع می‌شود، یک سیستم SCM با دامنه عمومی است که می‌تواند پروژه‌های کوچک (<10KLOC) و بزرگ (10000 KLOC) را مدیریت کنند (www.vestasys.org) فهرست کاملی از محیط‌ها و ابزارهای تجاری SCM را می‌توان در نشانی زیر یافت.

www.cmtoday.com/yp/commercial.html

۵-۳-۲۲ گزارش وضعیت

گزارش وضعیت پیکربندی (که گاهی صورت *وضعیت* نیز خوانده می‌شود) یک فعالیت SCM است که به سؤالات زیر پاسخ می‌گوید: (۱) چه اتفاقی رخ داده است؟ (۲) چه کسی مسبب آن بوده است؟ (۳) چه زمانی رخ داده است؟ (۴) چه چیزهای دیگری از آن تأثیر خواهند پذیرفت؟

جریان اطلاعات برای گزارش وضعیت پیکربندی (CSR) در شکل ۵-۲۲ نشان داده شده است. هر بار که به یک SCI هویتی جدید یا بازسازی شده داده می‌شود، یک مدخل CSR ساخته می‌شود. هربار که تغییری به تصویب CCA می‌رسد (یعنی یک ECO صادر می‌شود)، یک پیمانه CSR ساخته می‌شود. هربار که یک ممیزی انجام می‌شود، نتایج به‌عنوان بخشی از وظیفه CSR گزارش می‌شوند. خروجی یک CSR را می‌توان در یک بانک اطلاعاتی آنلاین قرار داد، به‌طوری که نرم‌افزارنویسان یا نگهدارنده‌ها بتوانند توسط گروهی از واژه‌های کلیدی به اطلاعات مربوط به تغییرات دست پیدا کنند. علاوه‌بر این، به‌طور منظم یک گزارش CSR تولید می‌شود و در اختیار مدیران و دست‌اندرکاران قرار می‌گیرد تا تغییرات مهم را ارزیابی کنند.

اندرز

برای هر کدام از اشیای پیکربندی، فهرستی از موارد ضروری تهیه کنید و آن را به‌روز نگه دارید. هنگامی که تغییری به‌عمل آوردید، حتماً آن را در این فهرست مشخص سازید.

۴-۲۲ مدیریت پیکربندی برای برنامه‌های تحت وب

قبلاً در این کتاب، درباره ماهیت خاص برنامه‌های تحت وب و روش‌های تخصصی (موسوم به روش‌های مهندسی وب[1]) بحث شد که برای ساخت این نوع برنامه‌ها مورد نیاز است. از جمله خصوصیات فراوانی که برنامه‌های تحت وب را از نرم‌افزارهای سنتی متمایز می‌سازند، ماهیت همه‌گیری تغییر است.

سازندگان برنامه‌های تحت وب غالباً از یک مدل فرایند افزایشی استفاده می‌کنند که در آن بسیاری از اصول به‌دست آمده از توسعه نرم‌افزار چابک (فصل ۳) را به‌کار بسته می‌شود. با استفاده از این رویکرد، تیم مهندسی غالباً هر نسخه از برنامه‌ی تحت وب را در دوره‌ی زمانی بسیار کوتاه با استفاده از یک رویکرد مشتری‌گرا توسعه می‌دهد. در نسخه‌های بعدی، محتوا و قابلیت‌های دیگر اضافه می‌شوند و این احتمال وجود دارد که در هر کدام تغییراتی پیاده‌سازی شود که به بهبود محتوا، قابلیت استفاده‌ی بهتر، بهبود زیبایی، گشت‌وگذار بهتر، بهبود کارایی و امنیت بیشتر منجر شود. بنابراین، در دنیای برنامه‌های تحت وب چابک، به تغییر نگاهی نسبتاً متفاوت می‌شود.

تغییرات کنترل‌نشده چه تأثیری بر برنامه‌ی تحت وب دارند؟

اگر عضو تیم برنامه‌ی تحت وب هستید، باید با آغوش باز پذیرای تغییرات باشید. در عین حال، یک تیم چابک نمونه، از همه‌ی چیزهایی که باعث سنگین شدن فرایند و بوروکراسی می‌شوند، پرهیز می‌کند. غالباً تصور می‌شود (البته به غلط) که پیکربندی نرم‌افزار واجد این خصوصیات هست. این تضاد ظاهری نه با ردکردن اصول، ابزارها و روش‌های کاری SCM، بلکه با شکل‌دهی دوباره به آن‌ها برای برآورده ساختن نیازهای خاص پروژه‌های برنامه‌ی تحت وب قابل حل است.

۱-۴-۲۲ مسائل غالب (Dominant Issues)

با رشد اهمیت برنامه‌های تحت وب در ادامه‌ی حیات شرکت‌های تجاری، نیاز به مدیریت پیکربندی نیز رشد پیدا می‌کند. چرا؟ چون بدون کنترل‌های اثربخش، تغییرات نامناسب در برنامه‌ی تحت وب (به‌خاطر دارید که فوریت و تکامل پیوسته‌ی صفات غالب در بسیاری از برنامه‌های تحت وب هستند) می‌تواند به اعلان غیر مجاز اطلاعات محصول جدید، قابلیت‌های عملیاتی خطادار یا به خوبی آزموده‌نشده‌ای که بازدید کنندگان سایت را به زحمت می‌اندازند، و سایر عواقب ناگوار اقتصادی یا حتی مصیبت بار شود.

[1] برای بحث جامعی در خصوص [Pre08]، روش‌های مهندسی وب را ببینید.

از همان راهبردهای عمومی برای مدیریت پیکربندی نرم‌افزار (SCM) که در این فصل شرح داده شدند، می‌توان استفاده کرد، ولی تاکتیک‌ها و ابزارها را باید تطبیق داد تا با ماهیت منحصربه‌فرد برنامه‌های تحت وب همخوانی داشته باشند. هنگام توسعه‌ی تاکتیک‌هایی برای مدیریت پیکربندی نرم‌افزار برنامه‌های تحت وب، چهار مسأله [Dar99] را باید مد نظر داشت.

محتوا. یک برنامه‌ی تحت وب معمولی حاوی آرایه گسترده‌ای از محتوا- متون، تصاویر گرافیکی، اسکریپت‌ها، فایل‌های ویدیویی/صوتی، فرم‌ها، عناصر صفحه‌ای فعال، جداول، داده‌های جریان‌دار (streaming) و بسیاری موارد دیگر- می‌شود. چالش پیش روی، سازمان‌دهی این دریای محتوا در قالب مجموعه‌ای از اشیای پیکربندی (بخش ۴-۱-۲۲) و سپس ایجاد سازوکارهای کنترلی پیکربندی برای این اشیاست. یک رویکرد برای این منظور، مدل‌سازی محتوای برنامه‌ی تحت وب با به‌کارگیری تکنیک‌های سنتی مدل‌سازی داده‌ها (فصل ۶) و متصل‌کردن مجموعه‌ای از خواص تخصصی به هر شیء است. ماهیت ایستا/ پویای هر شیء و طول عمر پیش بینی شده برای آن (مثلاً شیء موقت، با طول عمر مشخص یا دائمی) مثال‌هایی از خواص مورد نیاز برای ایجاد یک رویکرد SCM اثربخش هستند. برای مثال، اگر یک آیتم محتوایی به‌صورت ساعتی تغییر می‌کند، طول عمر آن، موقتی است. سازوکارهای کنترلی برای این آیتم با آن‌چه که برای یک مؤلفه‌ی فرم‌ها به‌کار می‌رود (و شیئ‌ای دائمی به شمار می‌رود) تفاوت دارد (کم‌تر رسمی است).

افراد. از آن‌جا که درصد چشمگیری از توسعه‌ی برنامه‌های تحت وب همچنان به شیوه‌ای برنامه‌ریزی نشده انجام می‌شوند، همه‌ی افراد دخیل در کار برنامه‌ی تحت وب می‌توانند به امر ایجاد محتوا، مبادرت ورزند (و می‌ورزند). بسیاری از کسانی که محتوا ایجاد می‌کنند، فاقد اطلاعات مهندسی نرم‌افزار بوده از نیاز به مدیریت پیکربندی کاملاً ناآگاهند. در نتیجه، برنامه به شیوه‌ای کنترل نشده رشد می‌کند و تغییر می‌یابد.

گسترش‌پذیری (Scalability). تکنیک‌ها و کنترل‌های به‌کار رفته در برنامه‌های تحت وب کوچک، به‌خوبی قابل تعمیم به مقیاس‌های بزرگ نیستند و در واقع توسعه‌پذیر نیستند. رشد چشمگیر یک برنامه‌ی تحت وب ساده به موازات پیاده‌سازی روابطی میان سیستم‌های اطلاعاتی موجود، بانک‌های اطلاعاتی و مسیرهای پورتال، امری متداول است. با رشد اندازه و پیچیدگی، تغییرات کوچک ممکن است اثراتی دور از انتظار و ناخواسته به بار آورند که باعث بروز مشکل شوند. بنابراین، دشواری سازوکارهای کنترل پیکربندی باید با ابعاد برنامه متناسب باشد.

سیاست. چه کسی «مالک» برنامه‌ی تحت وب است؟ این پرسش در شرکت‌های بزرگ و کوچک مطرح می‌شود و پاسخ آن تأثیری چشمگیر بر فعالیت‌های مدیریتی و کنترلی دارد. در برخی موارد، سازندگان برنامه‌ی تحت وب در خارج از سازمان IT جای دارند که این باعث بروز مشکلات ارتباطی بالقوه می‌شود. دارت [Dar99] پرسش‌های زیر را برای کمک به درک خط مشی‌های مربوط به مهندس وب مطرح می‌سازد:

- مسؤولیت صحت اطلاعات عرضه شده روی وب‌سایت را چه کسی می‌پذیرد؟
- چه کسی اطمینان حاصل می‌کند که فرایندهای کنترل کیفیت، قبل از انتشار اطلاعات روی سایت، دنبال شده‌اند؟

چگونه تعیین کنیم چه‌کسی مسؤولیت مدیریت پیکربندی برنامه‌ی تحت وب را بر عهده دارد؟

- مسؤولیت اعمال تغییرات بر عهده‌ی چه کسی است؟
- چه کسی هزینه تغییرات را تقبل می‌کند؟

پاسخ به این پرسش‌ها به تعیین افراد داخل سازمان، که باید فرایند مدیریت پیکربندی را بپذیرند، کمک می‌کند.

مدیریت پیکربندی برای برنامه‌های تحت وب همچنان در حال تکامل است (مثلاً [Ngu06]). فرایند SCM سنتی ممکن است بیش از حد دشوار باشد، ولی نسل جدیدی از *ابزارهای مدیریت محتوا* که به‌طور ویژه برای مهندسی وب طراحی شده‌اند، طی چند سال اخیر ظهور یافته است. این ابزارها فرایندی را وضع می‌کنند که اطلاعات موجود را (از آرایه گسترده‌ای از اشیای برنامه‌ی تحت وب) می‌گیرد، تغییرات اشیا را مدیریت می‌کند، آن‌ها را طوری سازمان‌دهی می‌کند که برای کاربر نهایی قابل ارائه باشند و سپس آن‌ها را برای نمایش در اختیار محیط طرف کلاینت قرار می‌دهد.

۲-۴-۲۲ اشیای پیکربندی برنامه‌ی تحت وب

برنامه‌های تحت وب شامل گستره وسیعی از اشیای پیکربندی- اشیای محتوایی (مثلاً متون، تصاویر گرافیکی، عکس، ویدیو، صوت)، مؤلفه‌های عملیاتی (مانند اسکریپت‌ها و اپلت‌ها) و اشیای واسط (مثلاً COM یا CORBA) می‌شوند. اشیای برنامه‌ی تحت وب را می‌توان به هر شیوه‌ای که برای سازمان مناسب باشد، شناسایی کرد (با نسبت دادن نام فایل‌های مناسب). به هر حال، برای حصول اطمینان از حفظ سازگاری میان سکوها، قراردادهای زیر توصیه می‌شود: نام فایل‌ها باید به طول ۳۲ کاراکتر محدود شود، از ترکیب حروف بزرگ و کوچک و همه‌ی حروف بزرگ و نیز از زیر خط باید پرهیز شود. به علاوه در مراجع URL (پیوندها) در داخل یک شیء پیکربندی، باید همواره از مسیرهای نسبی مثلاً (.../products/alarmsensor.htm) استفاده شود.

همه‌ی محتوای برنامه‌ی تحت وب دارای فرمت و ساختار هستند. فرمت‌های فایل داخلی، تحت کنترل محیط کامپیوتری‌ای هستند که محتوا در آن نگهداری می‌شود. به هر حال *قالب پرداختی* (rendering format که *قالب نمایشی* نیز خوانده می‌شود)، توسط سبک زیبایی‌شناختی و قواعد طراحی وضع‌شده برای برنامه‌ی تحت وب تعیین می‌شود. معماری محتوا در *ساختار محتوا* تعریف می‌شود؛ یعنی، طریقه‌ی مونتاژشدن اشیای محتوایی برای ارائه اطلاعات با معنی به کاربر نهایی را تعریف می‌کند. بویکو [Boi04] ساختار را چنین تعریف می‌کند «نقشه‌ای که روی یک مجموعه اشیای محتوایی پهن می‌شود تا آن‌ها را سازمان‌دهی کند و آن‌ها را برای افرادی که به آن‌ها نیاز دارند، قابل دستیابی می‌سازد.»

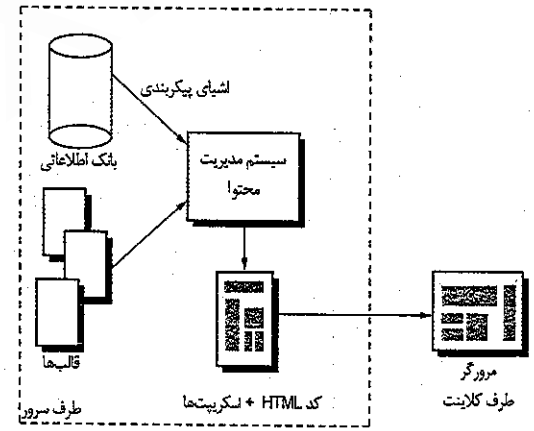
۳-۴-۲۲ مدیریت محتوا (Content Management)

مدیریت محتوا از این حیث با مدیریت پیکربندی در ارتباط است که سیستم پیکربندی نرم‌افزار (CMS)، فرایندی (پشتیبانی شده توسط ابزارهای مناسب) وضع می‌کند که محتوای موجود را (از آرایه گسترده‌ای از اشیای پیکربندی برنامه‌ی تحت وب) می‌گیرد، آن‌ها را طوری سازمان‌دهی می‌کند که برای کاربر نهایی قابل ارائه باشند و سپس آن‌ها را برای نمایش در اختیار محیط طرف کلاینت قرار می‌دهد.

«مدیریت محتوا، یادزهر آشفتگی اطلاعاتی در دنیای امروز است.»

باب بویکو

رایج‌ترین کاربرد سیستم مدیریت، در برنامه‌ی تحت وب پویاست. برنامه‌های تحت وب پویا صفحات وب را «داغ داغ» ایجاد می‌کنند به این معنی که کاربر معمولاً از برنامه‌ی تحت وب اطلاعات خاصی را درخواست می‌کند. برنامه‌ی تحت وب هم از یک بانک اطلاعاتی، اطلاعات را درخواست می‌کند، آن‌ها را فرمت‌بندی می‌کند و به کاربر عرضه می‌نماید. برای مثال، یک شرکت موسیقی، کتابخانه‌ای از CDها را برای فروش فراهم ساخته است. هنگامی که کاربر درخواست CD یا هم‌ارز الکترونیکی آن را می‌کند، یک بانک اطلاعاتی مورد درخواست قرار می‌گیرد و انواع اطلاعات درباره خواننده، CD (مثلاً تصویر جلد آن)، محتوای موسیقیایی و نمونه‌ی صوتی، همگی دانلود شده به‌صورت یک قالب محتوایی استاندارد پیکربندی می‌شوند. صفحه وب حاصل، در طرف سرور ساخته می‌شود و برای بررسی توسط کاربر به مرورگر نصب‌شده در طرف کلاینت تحویل می‌شود. نمایش کلی این فرایند در شکل ۶-۲۲ نشان داده شده است.



شکل ۶-۲۲ سیستم مدیریت محتویات.

در عمومی‌ترین حالت، CMS محتوا را برای کاربر نهایی و با فراخواندن سه زیرسیستم منسجم «پیکربندی» می‌کند: یک زیرسیستم جمع آوری، یک زیرسیستم مدیریتی و یک زیرسیستم انتشار [Boi04].

زیرسیستم جمع‌آوری. محتوا از روی داده‌ها و اطلاعاتی که باید ایجاد شود، یا توسط یک سازنده‌ی محتوا، به‌دست می‌آید. *زیرسیستم جمع‌آوری*، شامل کلیه‌ی کنش‌هایی که برای ایجاد و/یا به‌دست آوردن محتوا مورد نیازند و نیز قابلیت‌های فنی‌ای می‌شود که برای موارد زیر لازم هستند: (۱) تبدیل محتوا به شکلی قابل ارائه توسط یک زبان علامت‌گذاری (مانند HTML یا XML) و (۲) سازماندهی محتوا در قالب بسته‌هایی که به‌طور اثربخش در طرف کلاینت قابل نمایش باشند.

ایجاد و به‌دست‌آوردن محتوا (که غالباً تألیف نامیده می‌شود) معمولاً به موازات سایر فعالیت‌های توسعه‌ی برنامه‌ی تحت وب رخ می‌دهد و غالباً توسط افراد غیر فنی انجام می‌شود. این فعالیت، عناصر خلاقیت و پژوهش را در هم می‌آمیزد و با ابزارهایی پشتیبانی می‌شود که مؤلف را قادر می‌سازند تا طوری محتوا را سامان‌دهی کند که برای استفاده در داخل برنامه‌ی تحت‌وب، قابل استانداردسازی باشد.

نکته‌ی کلیدی

زیرسیستم جمع آوری، شامل کلیه‌ی کنش‌هایی می‌شود که برای ایجاد، کسب و/یا تبدیل محتوا به شکلی که به طرف کلاینت قابل ارائه باشد، مورد نیازند.

هنگامی که محتوا موجود باشد، باید آن‌ها را تبدیل کرد تا با خواسته‌های یک CMS مطابقت داشته باشند. این بدان معناست که محتوای خام باید از هرگونه اطلاعات بیهوده (مثلاً نمایش‌های گرافیکی زائد) مبّری باشد؛ محتوا طوری فرمت‌بندی شود که با خواسته‌های CMS همخوانی داشته باشد و نتایج در یک ساختار اطلاعاتی نگاشت شوند که مدیریت و نشر آن را امکان‌پذیر سازد.

زیرسیستم مدیریت. هنگامی که محتوا موجود باشد، باید آن را در مخزنی نگهداری کرد که برای استفاده‌های بعدی، فهرست‌برداری و کاتالوگ شده است، به‌طوری که موارد زیر در آن تعریف شود:

(۱) وضعیت فعلی (مثلاً آیا شیء محتوایی کامل یا در حال توسعه است؟)، نسخه مناسب شیء محتوایی و (۳) اشیای محتوایی مرتبط. بنابراین، *زیرسیستم مدیریت*، مخزنی را پیاده‌سازی می‌کند که شامل عناصر زیر می‌شود:

- *بانک اطلاعاتی محتوا*- ساختار اطلاعاتی که برای نگهداری کلیه‌ی اشیای محتوایی وضع شده است.
- *قابلیت‌های بانک اطلاعاتی*- قابلیت‌هایی که CMS را قادر می‌سازد تا اشیای محتوایی خاص (یا گروه‌هایی از اشیا) را جستجو، نگهداری و بازیابی کند و ساختار فایل وضع شده برای محتوا را مدیریت کند.
- *قابلیت‌های عملیاتی مدیریت پیکربندی*- عناصر عملیاتی و جریان کاری مرتبط که شناسایی شیء محتوایی، کنترل نسخه‌ها، مدیریت تغییرات و ممیزی تغییرات و گزارش‌دهی را پشتیبانی می‌کنند.

زیرسیستم مدیریت، علاوه بر این عناصر، یک قابلیت دیگر نیز پیاده‌سازی می‌کند که شامل شبه داده‌ها و قواعدی برای کنترل ساختار کلی محتوا و شیوه‌ی پشتیبانی از آن‌ها می‌شود.

زیرسیستم انتشار. محتوا باید از مخزن استخراج شود، به شکلی مناسب برای انتشار تبدیل شود و طوری فرمت‌بندی گردد که به مرورگرهای نصب شده در طرف کلاینت قابل انتقال باشد. زیرسیستم انتشار با به‌کارگیری یک سری قالب، موفق به انجام این وظایف می‌شود. هر قالب، قابلیتی است که مطالب قابل انتشار را با به‌کارگیری یکی از سه مؤلفه متفاوت زیر می‌سازد [Boi04]:

- *عناصر ایستا*- متون، گرافیک‌ها، رسانه‌ها و اسکریپت‌هایی که نیاز به پردازش بیشتر ندارند، مستقیماً به طرف کلاینت انتقال داده می‌شود.
- *سرویس‌های انتشار* - فراخوانی خدمات ویژه بازیابی و فرمت‌بندی که محتوا را (با استفاده از قواعد از پیش تعیین شده) شخصی می‌کنند، تبدیل داده‌ها را انجام می‌دهند و پیوندهای مناسب برای گشت‌وگذار می‌سازند.
- *سرویس‌های خارجی*- دستیابی به زیر ساخت اطلاعاتی خارجی نظیر داده‌های شرکتی یا برنامه‌های «اتاق پشتی».

یک زیرسیستم مدیریت محتوا که شامل هر سه زیرسیستم فوق می‌شود، برای پروژه‌های بزرگ برنامه‌ی تحت وب قابل استفاده است. به هر حال، فلسفه‌ی پایه و عملکرد مرتبط با یک CMS، برای تمامی برنامه‌های تحت وب قابل استفاده است.

نکته‌ی کلیدی

زیرسیستم‌مدیریت،برای‌همه‌ی محتواها یک مخزن ایجاد می‌کند. مدیریت پیکربندی در داخل این زیرسیستم انجام می‌شود.

نکته‌ی کلیدی

زیرسیستم انتشار، محتوا را از مخزن استخراج کرده آن‌ها را به مرورگر نصب‌شده در طرف کلاینت تحویل می‌دهد.

ابزارهای نرم‌افزاری

مدیریت محتوا

هدف: کمک به مهندس نرم‌افزار و تهیه‌کننده‌ی محتوا در مدیریت محتوایی که در برنامه‌ی تحت وب قرار داده خواهد شد.

مکانیک: ابزارهای این گروه به مهندس نرم‌افزار و تهیه کننده محتوا این امکان را می‌دهند تا محتوای برنامه‌ی تحت وب را به شیوه‌ای کنترل شده، بهنگام‌سازی کنند. اکثر این ابزارها یک سیستم مدیریت فایل ساده برقرار می‌سازند که برای انواع محتوای برنامه‌ی تحت وب، اجازه‌نامه‌های بهنگام‌سازی و ویرایش صفحه به صفحه صادر می‌کند. برخی حاوی یک سیستم ثبت نسخه‌اند، به‌طوری که نسخه‌های قبلی محتوا را می‌توان برای حفظ سوابق، بایگانی کرد.

ابزارهای نمونه

Vignette Content Management، که توسط شرکت Vignette توسعه یافته است و مجموعه‌ای از ابزارهای مدیریت محتوای تجاری را در بر دارد (www.vignette.com/us/Products).

Ektron-CMS300 که توسط ektron (www.ektron.com) تهیه شده است و مجوعه‌ای از ابزارهاست که قابلیت‌های مدیریت محتوا و نیز ابزارهای توسعه‌ی تحت وب را فراهم می‌سازد.

OmniUpdate، که توسط Website ASP,Inc. (www.omniupdate.com) تهیه شده است و ابزاری است که تأمین کنندگان مجاز محتوا به کمک آن می‌توانند محتوای یک برنامه‌ی تحت وب مشخص را به طریقی کنترل شده، بهنگام‌سازی کنند.

اطلاعات اضافی درباره ابزارهای مدیریت محتوا و SCM برای مهندسی وب را می‌توانید در یکی از چند وب‌سایت زیر بیابید:

Web Developer's Encyclopedia (**www.wdlv.com**)
WebDeveloper (**www.webdeveloper.com**)
Developer Shed (**www.devshed.com**)
webknowhow (**www.webknowhow.com**)
WebReference (**www.webreference.com**)

۲۲-۴-۴ مدیریت تغییرات

جریان کاری (workflow) مرتبط با کنترل تغییرات برای نرم‌افزارهای سنتی (بخش ۳-۳-۲۲)، عموماً برای توسعه‌ی برنامه‌های تحت وب بیش از حد سنگین است. این احتمال که درخواست تغییرات، گزارش تغییرات و مهندسی ترتیب سفارش تغییرات به شیوه‌ای چابک، قابل انجام باشد و در عین حال برای اکثر پروژه‌های توسعه برنامه‌های تحت وب قابل قبول باشد، چندان زیاد نیست. پس چگونه می‌توان جریان پیوسته‌ی تغییرات را که برای محتوا و عملکردهای برنامه‌های تحت وب درخواست می‌شود، مدیریت کرد؟

برای پیاده‌سازی مدیریت اثربخش تغییرات در فلسفه‌ی «کد بنویس و برو»، که همچنان بر توسعه برنامه‌های تحت وب حکم فرماست، فرایند کنترل تغییرات باید اصلاح گردد. هر تغییر باید در قالب یکی از چهار کلاس زیر دسته‌بندی شود:

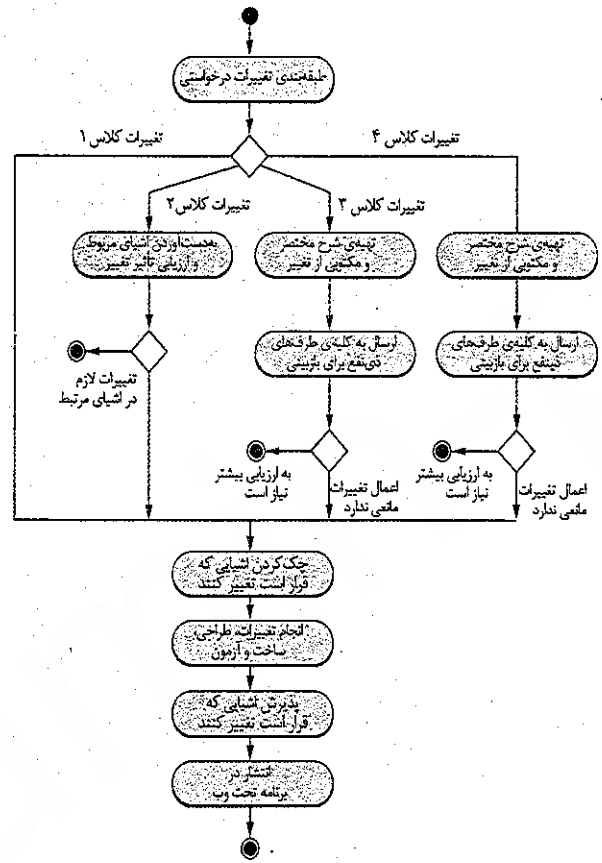
کلاس ۱ - تغییری در محتوا یا عملکردهایی که با یک خطا در ارتباط است یا محتوا و عملکردهای محلی را بهبود می‌بخشد.

کلاس ۲ - تغییری در محتوا یا عملکردهایی که بر سایر مؤلفه‌های عملیاتی یا اشیای محتوایی تأثیر می‌گذارد.

کلاس ۳ - تغییری در محتوا یا عملکردهایی که تأثیری گسترده بر برنامه‌ی تحت وب دارد (مثلاً بسط عمده‌ی قابلیت عملیاتی، بهبود چشمگیر یا کاهش محتوا، تغییرات عمده‌ی لازم در گشت‌وگذار)

کلاس ۴ - تغییری عمده در طراحی (مثلاً تغییر در طراحی واسط یا رویکرد گشت‌وگذار) که بلافاصله نزد یک یا چند گروه از کاربران قابل توجه است.

هنگامی که تغییرات درخواست شده دسته‌بندی شدند، می‌توان آن‌ها را مطابق با الگوریتم نشان‌داده‌شده در شکل ۷-۲۲ پردازش کرد.



شکل ۷-۲۲ مدیریت تغییرات برای برنامه‌های تحت وب.

همان طور که در این شکل مشاهده می‌شود، به تغییرات کلاس ۱ و کلاس ۲ به‌طور غیر رسمی پرداخته می‌شود و شیوه‌ی چابک برای آن‌ها برگزیده می‌شود. برای تغییری از کلاس ۱ باید تأثیر آن تغییر را بسنجید، ولی هیچ گونه مرور یا مستندسازی بیرونی مورد نیاز نیست. به موازاتی که تغییر انجام می‌شود، روال‌های استاندارد check-in و check-out توسط ابزارهای مخزن پیکربندی به اجبار اجرا می‌شوند. برای تغییرات کلاس ۲، باید تأثیر تغییر بر اشیای مرتبط را مرور کنید (یا از دیگر سازندگان مسؤول برای آن اشیا بخواهید که چنین کنند). اگر تغییر را بتوان اعمال کرد، بدون این که نیاز به تغییرات چشمگیری در اشیای دیگر باشد، اصلاحات بدون مرور یا مستندسازی اضافی انجام خواهد شد. اگر تغییرات اساسی مورد نیاز باشد، ارزیابی و برنامه‌ریزی بیشتری مورد نیاز خواهد بود.

به تغییرات کلاس‌های ۳ و ۴ نیز به شیوه‌ای چابک پرداخته می‌شود، ولی قدری مستندسازی توصیفی و روال‌های مرور رسمی‌تر مورد نیاز است. یک شرح تغییر- توصیفی از تغییر که به‌طور خلاصه اثر تغییر را ارزیابی می‌کند- برای تغییرات کلاس ۳ تهیه می‌شود. این شرح در میان تمامی اعضای تیم مرور توزیع می‌شود تا تأثیر آن را بهتر بسنجند. برای تغییرات کلاس ۴ نیز یک شرح تغییر تهیه می‌شود، ولی در این مورد، مرور توسط همه‌ی طرف‌های ذی‌نفع اجرا می‌شود.

ابزارهای نرم‌افزاری

مدیریت تغییرات

هدف: کمک به مهندسان وب و تهیه‌کنندگان محتوا در مدیریت تغییرات به موازات اعمال این تغییرات در اشیای پیکربندی برنامه‌ی تحت وب.

مکانیک: ابزارهای این گروه در آغاز برای نرم‌افزارهای سنتی تهیه شدند، ولی می‌توان آن‌ها را برای استفاده توسط مهندسان وب و تهیه کنندگان محتوا نیز تطبیق داد تا تغییرات به شیوه‌ای کنترل شده در برنامه‌های تحت وب اعمال شود. این ابزارها check-in و check-out خودکار، کنترل و لغو نسخه‌ها، گزارش‌دهی و سایر وظایف SCM را پشتیبانی می‌کنند.

ابزارهای نمونه

Change Man WCM، که توسط Serena (**www.serena.com**) توسعه یافته است و مجموعه‌ای از ابزارهای مدیریت تغییرات است که قابلیت‌های کامل SCM را فراهم می‌سازد.

Clear Case، که توسط Rational تهیه شده است و مجموعه‌ای از ابزارهاست که قابلیت‌های کامل مدیریت پیکربندی برای برنامه‌های تحت وب را فراهم می‌سازد:
www-306.ibm.com/software/rational/sw-atoz/indexC.html

Source Integrity، که توسط mks (**www.mks.com**) تهیه شده است و یک ابزار SCM است که می‌توان آن را با محیط‌های توسعه انتخابی منسجم ساخت.

۵-۴-۲۲ کنترل نسخه‌ها

به موازاتی که برنامه‌ی تحت وب از طریق یک سری «نسخه» تکامل پیدا می‌کند، ممکن است چند نسخه‌ی متفاوت همزمان موجود باشد. یک نسخه (همان برنامه‌ی تحت وب که در حال حاضر عملیاتی است) از طریق اینترنت برای کاربران نهایی در دسترس قرار دارد؛ نسخه دیگر (نسخه‌ی بعدی برنامه‌ی تحت وب) ممکن است در مراحل نهایی آزمون، قبل از استقرار باشد؛ نسخه سومی در حال توسعه باشد و از

لحاظ محتوا، زیبایی‌شناسی واسط و قابلیت‌های عملیاتی، به‌طور عمده بهنگام‌سازی شده باشد. اشیای پیکربندی باید به‌وضوح تعریف شوند، به‌طوری که هر کدام را بتوان با نسخه‌ی مناسب ربط داد. به‌علاوه، سازوکارهای کنترلی باید برقرار شوند. درایلینگر [Dre99] اهمیت کنترل نسخه‌ها (و تغییرات) را چنین مطرح می‌کند:

> در این سایت کنترل نشده که در آن چند مؤلف، مجاز به‌دستیابی و ویرایش محتوا هستند، احتمال بروز مشکلات و تضادها بالا می‌رود- به ویژه هنگامی که این مؤلفان از دفاتر متفاوت و در زمان‌های متفاوتی از شب و روز کار کنند. ممکن است شما روز را صرف بهبود بخشیدن به فایل index.html برای یک مشتری کنید. پس از اعمال تغییرات، سازنده‌ی دیگری که در منزل یا دفتری دیگر کار می‌کند، ممکن است شب را صرف آپلود نسخه‌ی جدیدی از index.html کند و به‌طور کامل روی کار شما نوشته شود، به‌طوری که هیچ راه بازگشتی هم نباشد!

این احتمال وجود دارد که شما نیز وضعیت مشابهی را تجربه کرده باشید. برای پرهیز از این وضعیت، به یک فرایند کنترل نسخه‌ها نیاز دارید.

۱. *یک مخزن مرکزی برای پروژه برنامه‌ی تحت وب باید ایجاد شود.* این مخزن نسخه‌های فعلی، همه‌ی اشیای پیکربندی برنامه‌ی تحت وب (محتوا، مؤلفه‌های عملیاتی و غیره) را در خود نگه خواهد داشت.

۲. *هر مهندس وب، پوشه کاری خودش را ایجاد می‌کند.* این پوشه حاوی آن دسته از اشیایی است که در هر زمان معینی ایجاد یا تغییر داده می‌شوند.

۳. *ساعت روی همه‌ی ایستگاه‌های کاری که سازندگان روی آن‌ها کار می‌کنند باید با هم تنظیم شده باشد.* این کار برای پرهیز از تضادهای ناشی از رونویسی‌هایی انجام می‌شود که در زمان‌های نزدیک به هم توسط دو نفر متفاوت رخ می‌دهند.

۴. *با توسعه‌ی اشیای پیکربندی جدید یا تغییر یافتن اشیای موجود، این اشیا وارد مخزن مرکزی می‌شوند.* ابزار کنترل نسخه‌ها (بحث CVS در کادر صفحه‌ی ۶۲۹ را ببینید) همه‌ی وظایف check-in و check-out را از دایرکتوری‌های کاری هر سازنده برنامه‌ی تحت وب مدیریت می‌کنند. این ابزار همچنین بهنگام‌سازی خودکار نامه‌های الکترونیکی به تمامی طرف‌های ذی‌نفع را به هنگام اعمال تغییرات روی مخزن بر عهده دارد.

۵. *با واردات یا صادرات اشیا از مخزن، یک پیام خودکار و دارای مهر تاریخ و زمان تهیه می‌شود که وقایع در آن ثبت شده اند.* به این ترتیب اطلاعات مفیدی برای ممیزی فراهم می‌آید و می‌تواند به بخشی از یک الگوی گزارش‌دهی اثربخش تبدیل شود.

این ابزار کنترل نسخه‌ها، نسخه‌های متفاوتی از برنامه‌ی تحت وب را حفظ می‌کند و در صورت نیاز می‌تواند به یک نسخه قدیمی‌تر باز گردد.

۶-۴-۲۲ ممیزی و گزارش‌دهی

در حیطه‌ی فرایندهای چابک، دیگر بر وظایف ممیزی و گزارش‌دهی در کارهای مهندسی وب تأکید

اطلاعات

استانداردهای SCM

فهرست استانداردهای SCM (که از www.12207.com استخراج شده است) جامع است:

استانداردهای IEEE

IEEE 828

EEE 1042

استانداردهای ISO

ISO 10007-1995

ISO/IEC 12207

ISO/IEC TR 15271

ISO/IEC TR 15846

استانداردهای EIA

EIA 649

EIA CMB4-1A

EIA CMB4-2

EIA CMB4-3

EIA CMB4-4

EIA CMB6-1C

EIA CMB6-3

EIA CMB6-4

EIA CMB6-5

EIA CMB7-1

استانداردهای نظامی آمریکا

DoD MIL STD-973

MIL-HDBK-61

سایر استانداردها

DO-178B

ESA PSS-05-09

AECL CE-1001-STD rev. 1

DOE SCM checklist

BS-6488

Best Practice – UK

CMII

standards.ieee.org/catalog/olis/

طرح‌های مدیریت پیکربندی نرم‌افزار

مدیریت پیکربندی نرم‌افزار

www.iso.ch/iso/en/ISOOnline.frontpage

مدیریت کیفیت، راهنمای CM

فن‌آوری اطلاعات – فرایند چرخه حیات نرم‌افزار

راهنمای ISO/IEC 12207

مهندسی نرم‌افزار – فرایند چرخه حیات نرم‌افزار – مدیریت پیکربندی برای سفارش نرم‌افزار

www.eia.org/

استاندارد اجماع ملی برای مدیریت پیکربندی

تعاریف مدیریت پیکربندی برای برنامه‌های کامپیوتری دیجیتال

شناسایی پیکربندی برای برنامه‌های کامپیوتری دیجیتال

کتابخانه‌های نرم‌افزارهای کامپیوتری

کنترل تغییر پیکربندی برای برنامه‌های کامپیوتری دیجیتال

سفارش مراجع مدیریت پیکربندی و داده‌ها

شناسایی پیکربندی

کنترل پیکربندی

حسابداری وضعیت پیکربندی

تبادل الکترونیکی داده‌های مدیریت پیکربندی

www-library.itsi.disa.mil

مدیریت پیکربندی

راهنمای مدیریت پیکربندی

راهنمای توسعه نرم‌افزارهای هوانوردی

راهنمای مدیریت پیکربندی نرم‌افزار

استاندارد برای مهندسی نرم‌افزارهایی با اهمیت ایمنی بحرانی

http://cio.doe.gov/ITReform/sqse/download/cmcklst.doc

استانداردهای بریتانیا، مدیریت پیکربندی سیستم‌های کامپیوتری

دفتر تجارت دولتی: www.ogc.gov.uk

مؤسسه CM Best Practice :www.icmhq.com

A Configuration Management Resource Guide اطلاعات مکملی برای علاقه مندان به فرایندهای CM فراهم می‌آورد. آن را از www.quality.org/config/cm-guide.html می‌توانید دریافت کنید.

نمی‌شود[1]، به هر حال، این‌طور هم نیست که کاملاً حذف شوند. همه‌ی اشیایی که وارد مخزن یا از آن خارج می‌شوند، در یک فایل کارنامه ثبت می‌شوند که در هر زمان می‌توان آن‌ها را مرور کرد. یک گزارش کارنامه‌ی کامل را می‌توان طوری ایجاد کرد که همه‌ی اعضای تیم برنامه‌ی تحت وب، ترتیب زمانی تغییرات را در یک دوره‌ی زمانی تعریف شده در اختیار داشته باشند. به علاوه، یک تذکر خودکار در قالب نامه‌ی الکترونیکی (به آدرس سازندگان و سایر افراد ذی‌نفع) را می‌توان با هر بار ورود یک شیء به مخزن یا خروج از آن ارسال کرد.

۲۲-۵ خلاصه

مدیریت پیکربندی نرم‌افزار یک فعالیت چتری است که در سرتاسر فرایند نرم‌افزار اجرا می‌شود. وظیفه SCM، شناسایی، کنترل، ممیزی و گزارش اصلاحاتی است که در اثنای توسعه نرم‌افزار و پس از ارائه آن به مشتری رخ می‌دهند. همه‌ی اطلاعاتی که به‌عنوان بخشی از مهندسی نرم‌افزار تولید می‌شوند، بخشی از پیکربندی نرم‌افزار را تشکیل می‌دهند. پیکربندی به شیوه‌ای سازمان‌دهی می‌شود که کنترل منظم تغییر را میسر سازد.

پیکربندی نرم‌افزار، مجموعه‌ای از اشیای مرتبط به یکدیگر تشکیل می‌شود که آنها را آیتم‌های پیکربندی نرم‌افزار (SCI) نیز می‌نامند؛ این اشیا به‌عنوان نتیجه‌ای از یک فعالیت مهندسی نرم‌افزار تولید می‌شوند. علاوه بر مستندات، برنامه‌ها و داده‌ها، محیط توسعه‌ای را که برای ایجاد نرم‌افزار تولید می‌شوند، می‌توان تحت کنترل پیکربندی درآورد.

هنگامی که یک شیء پیکربندی توسعه یافت و مرور شد، به خط مبنا تبدیل می‌شود. تغییراتی که در یک خط مبنا اعمال می‌شوند، منجر به ایجاد نسخه جدیدی از آن شیء می‌شود. تکامل یک برنامه را می‌توان با بررسی تاریخچه‌ی مرورهای همه‌ی اشیای پیکربندی پیگیری نمود. اشیای پایه و مرکب یک مخزن اشیاء تشکیل می‌دهند که تنوع‌ها و نسخه‌های گوناگونی از آن قابل ایجاد است. کنترل نسخه، مجموعه‌ای از روال‌ها و ابزارها برای مدیریت استفاده از این اشیاست.

کنترل تغییر، یک فعالیت رویه‌ای است که تضمین کیفیت را به موازات اعمال تغییرات در یک شیء پیکربندی بر عهده دارد. فرایند کنترل تغییر با یک درخواست تغییر آغاز می‌شود که پاسخ آن پذیرش یا رد درخواست تغییر است و به دنبال آن در صورت قبول تغییر، بهنگام‌سازی کنترل شده SCI انجام می‌شود.

ممیزی پیکربندی، یک فعالیت SQA است که به حفظ تضمین کیفیت، پس از اعمال تغییرات، کمک می‌کند. گزارش وضعیت، اطلاعات مربوط به هر تغییر را در اختیار افراد ذی‌صلاح قرار می‌دهد.

مدیریت پیکربندی برای برنامه‌های تحت وب از بیشتر جهات مشابه با SCM برای نرم‌افزارهای سنتی است، ولی هر کدام از وظایف هسته‌ای SCM باید به‌طور روان اجرا شوند و تدابیر ویژه‌ای برای مدیریت محتوا باید پیاده‌سازی شود.

---

[1] این رویکرد در حال تغییر است. به‌عنوان یک عنصر از امنیت برنامه تحت وب [Sar06] تأکیدی که بر SCM گذاشته می‌شود رو به افزایش است. یک ابزار مدیریت تغییرات با فراهم ساختن سازوکاری برای ردگیری و گزارش دهی هر کدام از تغییرات به عمل آمده در هر شیء برنامه‌ی تحت وب، می تواند محافظت ارزشمندی در برابر تغییرات زیان‌بار فراهم سازد.

مسائل و نکاتی برای تعمق

۲۲-۱ چرا قانون اول مهندسی نرم‌افزار صحت دارد؟ این قانون چگونه درک ما از الگوهای مهندسی نرم‌افزار را تحت‌تأثیر قرار می‌دهد؟

۲۲-۲ چهار عنصری که هنگام پیاده‌سازی یک سیستم SCM اثربخش وجود دارند کدام‌اند؟

۲۲-۳ دلایل مربوط به خط مبنا را به زبان ساده شرح دهید.

۲۲-۴ فرض کنید مدیر یک پروژه‌ی کوچک هستید، چه خطوط مبنایی برای پروژه تعریف می‌کنید و چگونه آنها را کنترل می‌کنید؟

۲۲-۵ یک سیستم بانک اطلاعاتی پروژه طراحی کنید که مهندس نرم‌افزار را قادر به نگهداری، ارجاع متقابل، پیگیری، بهنگام‌سازی، تغییر و کلیه آیتم‌های پیکربندی مهم دیگر سازد. این بانک اطلاعاتی چگونه با نسخه‌های گوناگون یک برنامه برخورد می‌کند؟ آیا شیوه رفتار با مستندات و کدهای منبع متفاوت است؟ چگونه دو نرم‌افزارنویس از اعمال تغییرات متفاوت روی یک SCI برحذر می‌مانند؟

۲۲-۶ یک ابزار SCM موجود را مورد پژوهش قرار داده، شرح دهید این ابزار چگونه کنترل نسخه‌ها، تنوع‌ها و به‌طور کلی اشیای پیکربندی را پیاده‌سازی می‌کند؟

۲۲-۷ روابط <بخشی از> و <مرتبط با> نشان‌گر روابطی ساده میان اشیای پیکربندی هستند. پنج رابطه دیگر را شرح دهید که ممکن است در حیطه‌ی بانک اطلاعاتی پروژه مفید واقع شوند.

۲۲-۸ یک ابزار SCM موجود را مورد پژوهش قرار داده شرح دهید چگونه مکانیک کنترل نسخه را پیاده‌سازی می‌کند. دو یا سه مقاله درباره SCM مطالعه کنید و ساختمان داده‌ها و سازوکارهای آدرس‌دهی به‌کار رفته در کنترل نسخه را توصیف کنید.

۲۲-۹ فهرست کنترلی تهیه کنید که در اثنای ممیزی پیکربندی قابل استفاده باشد.

۲۲-۱۰ اختلاف میان ممیزی SCM و مرور فنی رسمی در چیست؟ آیا عملکرد آنها را می‌توان به یک مرور خلاصه کرد؟ محاسن و معایب آن کدام است؟

۲۲-۱۱ اختلاف میان SCM برای نرم‌افزارهای سنتی و SCM برای برنامه‌های تحت وب را به اختصار شرح دهید.

۲۲-۱۲ مدیریت محتوا چیست؟ با استفاده از وب، ویژگی‌های یک ابزار مدیریت محتوا را جستجو کنید و خلاصه‌ای فراهم آورید.

فصل ۲۳

معیارهای محصول

نگاهی گذرا

معیارهای نرم‌افزار چیست؟ مهندسی، رشته‌ای با ماهیت کمّی است. مهندسان برای طراحی و ارزیابی محصولی که می‌سازند، از اعداد کمک می‌گیرند. تا همین اواخر، مهندسان نرم‌افزار، راهنمایی کمّی کمتری در کار خود داشتند - ولی وضع دارد عوض می‌شود. معیارهای فنی به مهندسان نرم‌افزار کمک می‌کنند تا طراحی و ساختمان محصولی را که می‌سازند، بهتر درک کنند.

چه کسی آن را انجام می‌دهد؟ مهندسان نرم‌افزار از معیارهای فنی استفاده می‌کنند تا به آنها در ساخت نرم‌افزاری با کیفیت بهتر کمک کنند.

چرا اهمیت دارد؟ همواره یک عنصر کیفی در ایجاد نرم‌افزارهای کامپیوتری وجود دارد. مشکل اینجاست که ارزیابی کیفی ممکن است کافی نباشد. مهندس نرم‌افزار به ملاک‌های عینی نیاز دارد که در طراحی داده‌ها، معماری، واسط‌ها و مؤلفه‌ها به او کمک کنند. آزمون‌گر به راهنمایی کمّی نیاز دارد که او را در گزینش موارد آزمون و اهداف آنها یاری دهد. معیارهای فنی مبنایی فراهم می‌آورد که از روی آن می‌توان تحلیل، طراحی، کدنویسی و آزمون را به‌طور عینی‌تر اجرا کرد و به‌طور کمّی‌تر مورد سنجش قرار داد.

مراحل کار کدام است؟ نخستین مرحله از فرایند اندازه‌گیری، به‌دست آوردن معیارها و موازین نرم‌افزاری است که برای نمایش نرم‌افزار موردنظر مناسب باشند. سپس، داده‌های لازم برای به‌دست آوردن معیارهای تدوین‌شده، جمع‌آوری می‌شوند. معیارهای مناسب پس از محاسبه، براساس دستورالعمل‌های ازپیش‌تعیین‌شده و داده‌های قبلی تحلیل می‌شوند. نتایج تحلیل مورد تفسیر قرار می‌گیرند تا دیدی از کیفیت نرم‌افزار به‌دست آید و نتایج تفسیر، به اصلاح محصولات کاری‌ای منجر می‌شوند که از تحلیل، طراحی، کدنویسی یا آزمون به‌دست آمده‌اند.

محصول کاری چیست؟ معیارهای نرم‌افزاری که از داده‌های به‌دست‌آمده از مدل‌های تحلیل و طراحی، کد منبع و موارد آزمون جمع‌آوری می‌شوند.

چگونه اطمینان حاصل کنم که درست از عهده امور برآمده‌ام؟ باید اهداف اندازه‌گیری را قبل از شروع جمع‌آوری داده‌ها مشخص کنید و هر یک از معیارهای فنی را به شیوه‌ای عاری از ابهام تعریف کنید. تنها چند معیار تعریف کنید و سپس آنها را برای به‌دست آوردن دیدی از محصول کاری مهندسی نرم‌افزار، مورد استفاده قرار دهید.

یکی از عناصر کلیدی در فرایند مهندسی، اندازه گیری است. برای درک بهتر صفات مدل‌های ایجادشده و سنجش کیفیت محصولات مهندسی شده یا سیستم‌های ساخته‌شده می‌توانید از موازین ویژه‌ای استفاده کنید. ولی برخلاف رشته‌های دیگر مهندسی، مهندسی نرم‌افزار ریشه در قوانین کمّی پایه‌ای نداشته موازین مطلق نظیر ولتاژ، جرم، سرعت، یا دما در جهان نرم‌افزار متداول نیستند. درعوض، کوشش می‌کنیم تا مجموعه‌ای از موازین غیرمستقیم را به‌دست آوریم که منجر به معیارهایی می‌شوند که شاخصی از کیفیت نرم‌افزار فراهم می‌آورند. از آنجا که موازین و معیارهای نرم‌افزار، مطلق نیستند، جای بحث دارند. فنتون [Fen91] این مسأله را چنین بیان می‌کند:

> اندازه‌گیری، فرایندی است که توسط آن اعداد و نمادها به صفات موجودیت‌هایی از جهان واقع نسبت داده می‌شوند، به شیوه‌ای که آنها را طبق قواعد کاملاً واضح تعریف کنند ... در علوم فیزیکی، پزشکی، اقتصاد و اخیراً علوم اجتماعی قادر به اندازه‌گیری صفاتی هستیم که قبلاً تصور می‌رفت قابل اندازه‌گیری نباشند... البته چنین اندازه‌گیری‌هایی همانند اندازه‌گیری‌های علوم فیزیکی دقیق نیستند ... ولی وجود دارند (و تصمیم‌گیری‌های پایه‌ای براساس آنها انجام می‌شود). احساس می‌کنیم که اجبار در اندازه‌گیری موارد غیرقابل اندازه‌گیری، به منظور بهبود بخشیدن به درک ما از موجودیت‌های مشخص، در مهندسی نرم‌افزار نیز به اندازه‌ی هر رشته دیگر اهمیت دارد.

ولی برخی اعضای جامعه نرم‌افزار، همچنان بر این عقیده‌اند که نرم‌افزار، غیرقابل اندازه‌گیری است یا کوشش در اندازه‌گیری باید تا زمانی که نرم‌افزار و صفات توصیف کننده‌ی آن بهتر شناخته شوند، به تعویق افتد. این اشتباه است.

معیارهای فنی برای نرم‌افزارهای کامپیوتری، مطلق نیستند ولی شیوه‌ای سیستماتیک برای ارزیابی کیفیت، براساس مجموعه‌ای از قواعد مشخص و واضح، در اختیارمان قرار می‌دهند. آنها همچنین به‌جای یک دید بَعدی، دیدی فوری در اختیار مهندس نرم‌افزار قرار می‌دهند. این امر، مهندس را قادر می‌سازد تا مشکلات بالقوه را قبل از تبدیل آنها به نقایص فاجعه‌بار، کشف و تصحیح کند.

در فصل ۴، کاربرد معیارها را در سطح فرایند و پروژه مورد بحث قرار دادیم. در این فصل، توجه خود را به موازینی معطوف خواهیم کرد که برای ارزیابی کیفیت محصول به موازات مهندسی شدن آن به‌کار می‌روند. این موازین صفات درونی محصول، یک شاخص زمان حقیقی از بازدهی مدل‌های تحلیل، طراحی و کدنویسی، اثربخشی موارد آزمون و کیفیت کلی نرم‌افزاری که ساخته می‌شود، در اختیار مهندس نرم‌افزار قرار می‌دهد.

> «بلوغ یک علم به اندازه‌ی بلوغ ابزارهای اندازه‌گیری آن است.»
>
> لویی پاستور

۲۳-۱ چارچوبی برای معیارهای تکنیکی محصول

چنان که در مقدمه این فصل گفتیم، اندازه‌گیری یعنی نسبت دادن اعداد یا نمادهایی به صفاتی از موجودیت‌های موجود در جهان واقعی. برای نیل به این مقصود، به یک مدل اندازه‌گیری با مجموعه‌ای از قواعد سازگار نیاز است. گرچه نظریه اندازه‌گیری (مثلاً [Kyb84]) و کاربرد آن در نرم‌افزارهای کامپیوتر (مثلاً [Zus97]) مباحثی خارج از حوصله این بحث هستند، بد نیست یک چارچوب بنیادی و مجموعه‌ای از اصول پایه را برای اندازه‌گیری معیارهای تکنیکی نرم‌افزار بنا کنیم.

۲۳-۱-۱ موازین، معیارها و شاخص‌ها

گرچه واژه‌های *میزان* (measure)، *اندازه‌گیری* (measurement) و *معیار* (metric) غالباً مترادف بوده به‌جای هم به‌کار می‌روند، توجه به تفاوت‌های ظریف میان آنها اهمیت دارد. در حیطه مهندسی نرم‌افزار، میزان عبارت از کمیتی است که نشان‌گر حد، مقدار، ابعاد، ظرفیت یا اندازه صفتی از محصول یا فرایند است. اندازه‌گیری، عمل تعیین میزان است. در *فرهنگ واژه‌های مهندسی نرم‌افزار* (IEE93b)، معیار چنین تعریف می‌شود: «میزانی کمّی از حدی که یک سیستم، مؤلفه، یا فرایند می‌تواند دارای یک صفت مفروض باشد».

هنگامی که یک نقطه‌ی داده‌ای منفرد جمع‌آوری شده باشد (مثل تعداد خطاهای کشف شده در مرور یک پیمانه‌ی منفرد)، یک میزان ایجاد شده است. اندازه‌گیری در نتیجه‌ی جمع‌آوری میزان‌هایی از تعداد خطاها برای هر مورد رخ می‌دهد. معیار نرم‌افزاری به نحوی با تک‌تک میزان‌ها در ارتباط است (مثل تعداد میانگین خطاهای یافت شده به‌ازای هر مرور یا تعداد میانگین خطاهای یافت شده به‌ازای هر نفر - ساعت صرف شده روی مرورها).

مهندس نرم‌افزار معیارهایی را جمع‌آوری و ایجاد می‌کند، به‌طوری که شاخص‌هایی از آنها به‌دست می‌آید. شاخص، معیار یا ترکیبی از معیارهاست که درکی از فرایند نرم‌افزار، پروژه نرم‌افزاری یا خود محصول به‌دست می‌دهد [RAG95]. شاخص، دیدی فراهم می‌آورد که مدیر پروژه یا مهندسان نرم‌افزار را قادر به تنظیم فرایند می‌سازد تا شرایط پروژه یا فرایند بهتر شود.

> **تفاوت میان میزان و معیار در چیست؟**

> **نکته‌ی کلیدی**
>
> شاخص به معیار یا مجموعه معیارهایی گفته می‌شود که دیدی از فرایند، محصول یا پروژه به‌دست می‌دهند.

۲۳-۱-۲ چالش معیارهای تکنیکی

طی چهار دهه گذشته، بسیاری از پژوهش‌گران کوشیده‌اند معیاری توسعه دهند که میزانی قابل درک از پیچیدگی نرم‌افزار ارائه دهد. فنتون [Fen94] این پژوهش را به‌عنوان جستجویی به دنبال «جام مقدس» توصیف می‌کند. گرچه ده‌ها میزان از پیچیدگی پیشنهاد شده است [Zus90]، هر یک دیدگاهی نسبتاً متفاوت به پیچیدگی و صفاتی از سیستم دارد که به پیچیدگی می‌انجامد. به‌طور مشابه، معیاری برای ارزیابی یک خودروی «جذاب» در نظر بگیرید. برخی ناظران ممکن است بر طراحی بدنه تأکید ورزند؛ عده‌ای ممکن است خصوصیات مکانیکی را مد نظر قرار دهند؛ برخی نیز ممکن است هزینه، یا کارایی یا استفاده از سوخت‌های متنوع یا حتی قابلیت بازیافت در هنگام اوراق کردن خودرو را مورد توجه قرار دهند. ازآنجا که هرکدام از این خصوصیات ممکن است با بقیه در تضاد باشد، به‌دست آوردن یک مقدار منفرد برای «جذاب‌بودن» دشوار می‌شود. برای نرم‌افزار نیز همین اتفاق رخ می‌دهد.

به هر حال نیاز به اندازه‌گیری و کنترل پیچیدگی همچنان به قوت خود باقی است. اگر به‌دست آوردن مقداری از این معیار کیفی دشوار باشد، توسعه موازینی از صفات داخلی برنامه (مثل پیمانه‌های اثربخش، استقلال عملیاتی و صفات دیگری که در فصل ۸ بحث شد) باید امکان‌پذیر باشد. این موازین و معیارهای به‌دست آمده از آنها را می‌توان به‌عنوان شاخص‌های مستقلی از کیفیت مدل‌های تحلیل و طراحی به‌کار برد. باز هم مشکلاتی ایجاد می‌شود. فنتون [Fen94] به این مسأله اشاره کرده است: «خطر کوشش برای یافتن موازینی که این تعداد صفات متفاوت را مشخص می‌سازند، آن‌است‌که موازین باید اهداف متضاد را به‌طور اجتناب‌ناپذیری برآورده سازند. این موضوع با *نظریه‌ی نمایشی اندازه‌گیری* در تضاد است.» گرچه گفته‌ی فنتون درست است، بسیاری چنین استدلال می‌کنند که اندازه‌گیری‌های تکنیکی اجراشده طی مراحل اولیه فرایند نرم‌افزار، سازوکاری سازگار و عینی برای سنجش کیفیت در اختیار مهندسان نرم‌افزار قرار می‌دهد.

> «درست همان‌طور که اندازه‌گیری دما با انگشت اشاره آغاز می‌شود... و به ابزارها، مقیاس‌ها و تکنیک‌های پیچیده تکامل می‌یابد، اندازه‌گیری در نرم‌افزار نیز به همین منوال رشد و بلوغ پیدا می‌کند.»
>
> شاری فلیگر

ولی خوب است ببرسیم این معیارهای تکنیکی تا چه حد اعتبار دارند. یعنی معیارهای تکنیکی چقدر با قابلیت اطمینان و کیفیت سیستم کامپیوتری همسویی دارد؟ فنتون [Fen91] با این پرسش چنین برخورد می‌کند:

> علی‌رغم روابط شهودی بین ساختار داخلی محصولات نرم‌افزاری [معیارهای تکنیکی] و صفات فرایند و محصول خارجی آن، تلاش علمی زیادی برای برقراری روابط خاص انجام نشده است. چند دلیل برای آن وجود دارد که مهمترین علتش این‌است که تجربیات کافی در این زمینه وجود ندارد.

هر یک از «چالش‌های» ذکرشده در بالا، دلیلی برای احتیاط است، ولی دلیلی برای از دست دادن معیارهای تکنیکی به‌شمار نمی‌رود.[1] اگر هدف، دستیابی به کیفیت است، اندازه‌گیری ضروری است.

۲۳-۱-۳ اصول اندازه‌گیری

پیش از آن‌که به معرفی معیارهای تکنیکی بپردازیم که (۱) به ارزیابی مدل‌های تحلیل و طراحی کمک کنند؛ (۲) شاخصی از پیچیدگی طراحی‌های رویه‌ای و کد منبع فراهم آورند و (۳) طراحی آزمون‌های اثربخش‌تر را تسهیل کنند، درک اصول اندازه‌گیری پایه، اهمیت دارد. روشه [Roc94] یک فرایند اندازه‌گیری پیشنهاد می‌کند که توسط پنج فعالیت مشخص می‌شود:

- *تدوین.* به‌دست آوردن موازین و معیارهای نرم‌افزاری برای نمایش دادن نرم‌افزار موردنظر
- *جمع‌آوری.* سازوکارهای مورداستفاده برای انباشتن داده‌های لازم جهت به‌دست آوردن معیارهای تدوین‌شده
- *تحلیل.* محاسبه معیارها و به‌کارگیری ابزارهای ریاضی
- *تفسیر.* ارزیابی معیارهایی که منجر به کوشش برای به‌دست آوردن دیدی از کیفیت نمایش می‌شود
- *بازخورد.* توصیه‌هایی که از تفسیر معیارهای تکنیکی منتقل شده به تیم نرم‌افزاری به‌دست می‌آیند

معیارهای نرم‌افزار تنها در صورتی مفید واقع می‌شوند که به‌طرزی اثربخش مشخص و اعتبارسنجی شده باشند به‌طوری که ارزش آن‌ها به اثبات رسیده باشد. اصول زیر [Let03b] نمونه‌هایی از چندین اصلی هستند که می‌توان برای مشخص کردن و اعتبارسنجی معیارها پیشنهاد کرد:

- *معیار باید خواص ریاضی مطلوب داشته باشد.* یعنی، ارزش معیار باید در گستره‌ای معنی‌دار قرار داشته باشد (مثلاً صفر یا یک که صفر واقعاً به معنای نبود آن کیفیت یا صفت، یک نشان‌گر حداکثر مقدار و ۰/۵ نمایان‌گر «نقطه میانه» است). همچنین، معیاری که ادعا می‌شود در یک مقیاس عددی تنظیم شده است نباید از اجزایی تشکیل شده باشد که در مقیاس ترتیبی (ordinal scale) قابل تعریف هستند.

[2] گرچه نقد معیارهای خاص در متون رایج است، بسیاری از متقدان، مسائل محرمانه را کانون توجه قرار داده از هدف اصلی معیار در جهان واقعیت غافل می‌مانند: کمک به مهندس نرم‌افزار در پی افکندن راهی سیستماتیک و عینی برای به‌دست آوردن دیدی از کارش و در نتیجه، بهبودبخشیدن به کیفیت محصول.

مرجع وب

اطلاعات فراوانی در خصوص معیارهای محصول توسط هورست زوس در آدرس زیر جمع آوری شده است:
Irb.cs.tu-berlin.de/~zuse/

مراحل یک فرایند اندازه‌گیری
اثربخش از چه قرار است؟

اندرز

در واقع، بسیاری از معیارهای محصول که امروزه به کار گرفته می‌شوند، آن چنان که باید، از این اصول پیروی نمی‌کنند ولی این بدان معنا نیست که این معیارها فاقد ارزش هستند. فقط هنگام استفاده از آن‌ها مراقب باشید و بدانید هدف آن‌ها فراهم آوردن یک دید است نه وارسی اکید علمی.

- *هنگامی یک معیار، خصوصیتی از نرم‌افزار را نمایش می‌دهد که با رخ‌دادن صفت مثبت، افزایش می‌یابد یا با مشاهده‌ی صفت نامطلوب، کاهش می‌یابد، ارزش معیار به همان شیوه افزایش یا کاهش پیدا کند.*
- *هر معیاری باید به‌صورت تجربی و در انواع متفاوتی از حیطه‌ها اعتبارسنجی گردد و بعداً منتشر شود یا در تصمیم گیری‌ها به‌کار گرفته شود.* معیار باید عامل مورد نظر را مستقل از سایر عوامل اندازه‌گیری کند. باید این قابلیت را داشته باشد که بتوان آن را وسعت داد تا سیستم‌ها و کارهای بزرگ را در انواع زبان‌های برنامه‌نویسی و دامنه‌های سیستمی پوشش دهد.

گرچه تدوین، مشخص‌سازی و اعتبارسنجی اهمیتی حیاتی دارند، جمع آوری و تحلیل نیز فعالیت‌هایی هستند که فرایند اندازه‌گیری را به پیش می‌رانند. روچ [Roc94] برای این فعالیت‌ها اصول زیر را پیشنهاد می‌کند: (۱) هرگاه که امکان داشت، جمع آوری و تحلیل داده‌ها باید خودکار شود؛ (۲) برای برقراری رابطه میان صفات داخلی محصول و خصوصیات کیفیتی خارجی (مثلاً این که آیا سطح پیچیدگی معماری با تعداد نقایص گزارش شده در استفاده از محصول همبستگی دارد) باید تکنیک‌های آماری به‌کار برده شود؛ و (۳) دستورالعمل‌های تفسیری و توصیه‌ها باید برای هر معیار مشخص شوند.

۲۳-۱-۴ سنجش هدف‌گرای نرم‌افزار

الگوی هدف/پرسش/معیار (GQM) به‌عنوان تکنیکی برای شناسایی معیارهای با معنی در بخشی از فرایند نرم‌افزار توسط باسیلی و وایس [Bas84] توسعه یافت. در GQM بر سه چیز تأکید می‌شود: (۱) تعیین یک *هدف* اندازه‌گیری که ویژه‌ی خصوصیتی از محصول یا فعالیت فرایندی است که قرار است ارزیابی شود، (۲) تعریف مجموعه‌ای از *پرسش‌ها* که باید برای دستیابی به هدف به آن‌ها پاسخ گفته شود و (۳) شناسایی معیارهایی با تدوین مناسب که به پاسخ گفتن به این پرسش‌ها کمک کنند.

برای تعریف هر کدام از اهداف اندازه‌گیری می‌توان از *قالب تعریف اهداف* [Bas94] بهره برد. این قالب به شکل زیر است:

تحلیل {نام فعالیت یا صفتی که قرار است اندازه‌گیری شود} **با هدف** {هدف کلی تحلیل[1]} **نسبت به** {جنبه‌ای از فعالیت یا صف که در نظر گرفته می‌شود} **از دیدگاه** {کسانی که به این اندازه‌گیری علاقه دارند} **در حیطه‌ی** {محیطی که در آن اندازه‌گیری رخ می‌دهد}.

به‌عنوان یک مثال، قالب تعریف اهداف را برای *SafeHome* در نظر بگیرید.

تحلیل معماری نرم‌افزار *SafeHome* **با هدف** ارزیابی مؤلفه‌های ساختاری **نسبت به** توانایی بسط‌پذیری بیشتر *SafeHome* **از دیدگاه** انجام کارهای مهندسی نرم‌افزار **در حیطه‌ی** بهبود بخشیدن به محصول طی سه سال آینده.

با تعریف واضح هدف اندازه گیری، مجموعه‌ای از پرسش‌ها توسعه می‌یابد. پاسخ این پرسش‌ها به تیم نرم‌افزار (یا ذی‌نفع‌ها) کمک می‌کند تا تعیین کنند که آیا هدف اندازه‌گیری برآورده شده است یا خیر. از جمله پرسش‌هایی که می‌توان پرسید، عبارتند از:

[1] وان سولینگن و برگوت [Sol99] معتقدند که هدف تقریباً همیشه «شناخت، کنترل یا بهبودبخشیدن» به یک فعالیت از فرایند یا صفتی کیفی است.

مرجع وب

بحث مفیدی درباره‌ی GQM را می‌توان در وب‌سایت زیر یافت:
www.thedacs.com/GoldPractices/practices/gqma.html

Q_1: آیا مؤلفه‌های معماری طوری مشخص شده‌اند که توابع و داده‌های مرتبط با آن از هم تفکیک شده باشند؟

Q_2: آیا پیچیدگی هر مؤلفه در مرزهایی هست که اصلاح و بسط را تسهیل کند؟

هر کدام از این پرسش‌ها باید به طریق کمّی و با به‌کارگیری یک یا چند معیار پاسخ گفته شود. برای مثال، معیاری که نشان‌گر یکپارچگی یک مؤلفه معماری است (فصل ۸) ممکن است در پاسخ گفتن به پرسش Q_1 مفید واقع گردد. معیارهای بحث شده در ادامه‌ی این فصل ممکن است دیدی برای Q_2 فراهم سازند. در هر حال، معیارهایی که انتخاب (یا به‌دست آورده) می‌شوند باید با اصول اندازه‌گیری بحث شده در بخش ۳-۱-۲۳ و صفات اندازه‌گیری بحث شده در ۵-۱-۲۳ همخوانی داشته باشند.

۵-۱-۲۳ صفات معیارهای نرم‌افزاری اثربخش

صدها معیار برای نرم‌افزارهای کامپیوتری پیشنهاد شده است، ولی همه‌ی آنها عملاً به‌کار مهندس نرم‌افزار نمی‌آیند. برخی مستلزم اندازه‌گیری‌های بسیار پیچیده هستند، برخی دیگر چنان اسرارآمیز هستند که تعداد اندکی از حرفه‌ای‌ها امید به درک آنها دارند و عده‌ای دیگر خالی از ایده‌های هوشمندانه‌اند.

اجیوگو [Eji91] مجموعه‌ای از صفات را تعریف می‌کند که معیارهای نرم‌افزاری اثربخش باید شامل این صفات باشند. معیارها و موازینی که به این امر می‌انجامند، باید:

- *ساده و قابل محاسبه باشند.* چگونگی به‌دست آوردن معیار باید به آسانی قابل یادگیری باشد و محاسبات آن نباید مستلزم زمان و انرژی زیاد باشد.
- *از نظر تجربی و از نظر شهودی قانع کننده باشند.* معیار باید تصورات شهودی مهندس را در مورد صنعت محصول موردنظر برآورده کند (مثلاً معیاری که انسجام پیمانه را اندازه‌گیری می‌کند، باید با افزایش سطح انسجام، افزایش یابد).
- *سازگار و هدفمند باشد.* نتیجه‌ی معیار نباید مبهم باشد. شخص ثالث مستقل باید بتواند با استفاده از همان اطلاعات راجع به نرم‌افزار، به همان معیار برسد.
- *سازگاری در استفاده از واحدها و ابعاد.* محاسبه ریاضی معیار باید از میزانی استفاده کند که منجر به ترکیب نادرستی از واحدها نشود. به‌عنوان مثال، افزایش افراد موجود در تیم پروژه به وسیله متغیرهای زبان برنامه‌نویسی در یک برنامه، ترکیب نادرستی از واحدها را ایجاد می‌کنند.
- *استقلال زبان برنامه‌سازی.* معیارها باید براساس مدل تحلیل، مدل طراحی، یا ساختار برنامه باشد. نباید به معناشناسی و نحو زبان برنامه‌سازی وابسته باشند.
- *سازوکار مؤثری برای بازخورد کیفیت.* یعنی، معیار باید اطلاعاتی را در اختیار مهندس نرم‌افزار قرار دهد که منجر به محصولی با کیفیت بالا شود.

گرچه اکثر معیارهای نرم‌افزاری، صفات فوق را دارا هستند، برخی از معیارهای پرکاربرد ممکن‌است واجد یک یا دو مورد نباشند. یک مثال، امتیاز عملکرد (Function Point) است– میزانی از عملکرد نرم‌افزار که در بخش ۱-۲-۲۳ بحث شد. می‌توان استدلال کرد[1] که این صفت سازگار و عینی به

چگونه باید کیفیت یک معیار نرم‌افزار پیشنهادی را ارزیابی کنیم؟

اندرز

تجربه نشان می‌دهد که یک معیار محصول تنها در صورتی به‌کار برده خواهد شد که مبتنی بر عقل سلیم باشد و به‌راحتی بتوان آن را محاسبه کرد. اگر به ده‌ها شمارش نیاز باشد و محاسبات پیچیده‌ای باید انجام شود، احتمال پذیرفته‌شدن گسترده‌ی آن معیار کم خواهد بود.

[1] استدلالی در جهت مخالف نیز می‌توان ارائه داد. ماهیت معیارهای نرم‌افزاری چنین است.

SafeHome

بحث بر سر معیارهای محصول

صحنه: کابین وینود.

نقش‌آفرینان: وینود، جیمی و اد- اعضای تیم نرم‌افزاری که کار طراحی در سطح مؤلفه‌ها و طراحی موارد آزمون را ادامه می‌دهند.

گفتگو:

وینود: داگ [داگ میلر، مدیر مهندسی نرم‌افزار] به من گفت که همه باید از معیارهای محصول استفاده کنیم، ولی یک جورهایی مبهم حرف می‌زد. تازه می‌گفت خیلی هم فشار نیاوریم... این که استفاده از آن‌ها به خودمان بستگی دارد.

جیمی: خوب است، چون اصلاً راهی نیست که اندازه‌گیری را شروع کنیم. همین طوری هم کلی وقت کم داریم.

اد: من با جیمی موافقم. با مشکل مواجه می‌شویم. بفرمایید... وقتی نماند.

وینود: بله، می‌دانم، ولی احتمالاً استفاده از آن‌ها یک مزیت‌هایی هم دارد.

جیمی: بحثی ندارم وینود، ولی مسأله وقت است... و من یکی که اصلاً وقت اضافی ندارم.

وینود: ولی اگر این اندازه‌گیری‌ها باعث صرفه‌جویی در وقت بشود، چطور؟

اد: اشتباه می‌کنی، این کار وقت می‌برد و همان‌طور که جیمی گفت...

وینود: نه صبر کن... چی باعث صرفه جویی در وقت می‌شود؟

جیمی: چطور؟

وینود: دوباره کاری... این طوری. اگر میزانی که از آن استفاده می‌کنیم، ما را در پرهیز از یک خطای عمده یا حتی معتدل کمک کند و این باعث شود که مجبور به دوباره کاری روی بخشی از سیستم نشویم، در زمان صرفه جویی می‌شود. نه؟

اد: امکان دارد، فکر می‌کنم، ولی می‌توانی تضمین بدهی که یک معیار محصول به ما در پیداکردن مسأله‌ای کمک کند؟

وینود: تو می‌توانی تضمین بدهی که نمی‌کند؟

جیمی: پس پیشنهادت چیست؟

وینود: فکر کنم باید چند معیار طراحی، احتمالاً شیء گرا، انتخاب کنیم و برای هر کدام از مؤلفه‌هایی که توسعه می‌دهیم از آن‌ها به‌عنوان بخشی از فرایند مرور استفاده کنیم.

اد: من خیلی با معیارهای شیءگرا آشنا نیستم.

وینود: من قدری وقت برای بررسی آن‌ها می‌گذارم و چندتای آن‌ها را توصیه می‌کنم... مشکلی نیست؟

[اد و جیمی بدون اشتیاق سر تکان می‌دهند.]

شکست می‌انجامد زیرا شخص ثالث ممکن است قادر نباشد همان مقدار امتیاز عملکرد را به‌دست آورد که همکارش با همان اطلاعات درخصوص نرم‌افزار به‌دست آورده است. پس آیا باید میزان FP را رد کرد؟ پاسخ البته منفی است. FP دیدی سودمند به‌دست می‌دهد و بنابراین مقداری متمایز ارائه می‌کند، حتی اگر به‌طور کامل واجد یک صفت نباشد.

۲-۲۳ معیارهایی برای مدل خواسته‌ها

کار تکنیکی در مهندسی نرم‌افزار با خلق مدل تحلیل آغاز می‌شود. در همین مرحله است که خواسته‌ها به‌دست می‌آید و مبنایی برای طراحی بنا گذاشته می‌شود. بنابراین معیارهای فنی که دیدی از کیفیت مدل تحلیل به‌دست دهند، مطلوب هستند.

گرچه تعداد معیارهای تحلیل و مشخصات نسبتاً اندکی در متون به چشم می‌خورد، می‌توان معیارهای به‌دست آمده برای براورد پروژه را به منظور استفاده در این حیطه تطبیق داد. با این معیارها می‌توان مدل تحلیل را به قصد پیش‌بینیِ «اندازه» سیستم حاصل مورد بررسی قرار داد. اندازه گاهی (ولی نه همواره) شاخصی از پیچیدگی طراحی است و تقریباً همواره شاخصی از افزایش تلاش‌های لازم برای کدنویسی، انسجام‌بخشی و آزمون است.

۱-۲-۲۳ معیارهای مبتنی بر عملکرد

معیار امتیاز عملکرد (FP) را می‌توان به‌طور اثربخش، به‌عنوان ابزاری برای اندازه‌گیری عملکردهای ارائه‌شده توسط سیستم به‌کار گرفت[1]. در این صورت، با استفاده از داده‌های تاریخی می‌توان از معیار FP برای (۱) برآورد هزینه یا تلاش لازم برای طراحی، کدنویسی و آزمایش نرم‌افزار، (۲) پیش‌بینی تعداد خطاهایی که طی آزمون مشاهده خواهد شد، و (۳) پیش‌بینی تعداد مؤلفه‌ها و/یا تعداد خطوط کد لازم برای پیاده‌سازی سیستم استفاده کرد.

امتیاز عملکرد با به کارگیری یک رابطه‌ی تجربی مبتنی بر موازین قابل شمارش (مستقیم) از دامنه‌ی اطلاعاتی و ارزیابی کیفی پیچیدگی نرم‌افزار به‌دست می‌آیند. مقادیر دامنه‌ی اطلاعاتی به شیوه‌ی زیر تعریف می‌شوند[2]:

تعداد ورودی‌های خارجی (IE)، هر *ورودی خارجی* که منشاء آن یک کاربر است یا از یک برنامه کاربردی دیگر صادر می‌شود که داده‌های کاربرد-گرای متمایز یا اطلاعات کنترلی فراهم می‌آورد. ورودی‌ها غالباً برای بهنگام‌سازی *فایل‌های منطقی داخلی* (ILF ها) به‌کار می‌روند. ورودی‌ها باید از درخواست‌ها متمایز و جداگانه شمارش شوند.

تعداد خروجی‌های خارجی (EOها)، هر *خروجی خارجی*، داده‌های به‌دست آمده در داخل برنامه کاربردی است که اطلاعات مورد نیاز کاربر را در اختیارش قرار می‌دهد. در این حیطه، خروجی خارجی به گزارش‌ها، صفحات نمایش، پیام‌های خطا و غیره اطلاق می‌شود. آیتم‌های داده‌ای منفرد در داخل گزارش جداگانه شمارش نمی‌شوند.

تعداد درخواست‌های خارجی (EQها)، *درخواست خارجی* به‌عنوان یک ورودی آنلاین تعریف می‌شود که به تولید پاسخ فوری نرم‌افزار به شکل یک خروجی آنلاین منجر می‌شود (و غالباً به شکل یک ILF بازیابی می‌شود).

مرجع وب

اطلاعات بسیار مفیدی درباره امتیاز عملکرد را می‌توان در دو آدرس زیر یافت:

www.functionpoints.com

و

www.ifpug.org

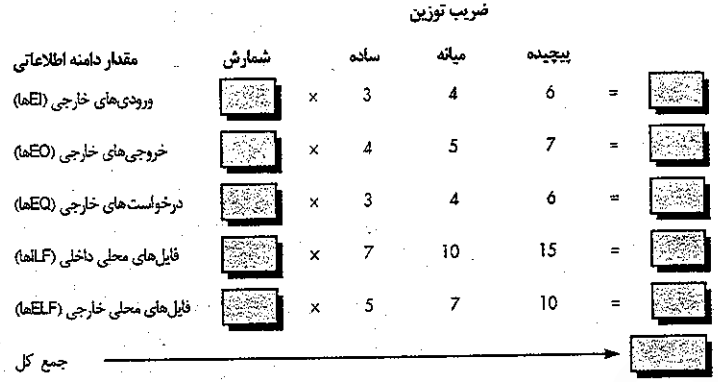
[1] صدها کتاب و مقاله درباره معیارهای FP نوشته شده است. فهرست خوبی از این نوشته‌ها را در [IFP05] می‌توانید بیابید.

[2] در واقع، تعریف مقادیر دامنه‌ی اطلاعاتی و شیوه شمارش آنها قدری پیچیده‌تر است. خواننده‌ی علاقه‌مند می‌تواند به [IFP01] رجوع کند.

تعداد فایل‌های منطقی داخلی (ILFها). هر *فایل منطقی داخلی*، گروه‌بندی منطقی داده‌هایی است که درون برنامه کاربردی قرار دارند و از طریق ورودی‌های خارجی حفظ می‌شود.

تعداد فایل‌های واسط خارجی (EIFها). هر فایل واسط خارجی، گروه‌بندی منطقی داده‌هایی است که در بیرون از برنامه‌ی کاربردی قرار دارند، ولی اطلاعاتی را در اختیار می‌گذارند که ممکن است در برنامه کاربردی قابل استفاده باشند.

هنگامی که این داده‌ها جمع آوری شد، جدول شکل ۱-۲۳ کامل می‌شود و یک مقدار پیچیدگی به هر شمارش نسبت داده می‌شود. سازمان‌هایی که از روش‌های امتیاز عملکرد استفاده می‌کنند، برای تعیین این که آیا یک مدخل معین، ساده، متوسط یا پیچیده است، ملاک‌هایی تهیه می‌کنند. با این وجود، تعیین پیچیدگی قدری موضوعی است.



شکل ۱-۲۳ محاسبه امتیاز عملکرد.

برای محاسبه امتیازهای عملکرد (FP)، از رابطه‌ی زیر استفاده می‌شود:

$$FP = \text{شمارش کل} \times [0.65 + 0.01 \times \Sigma(F_i)] \qquad (۲۳-۱)$$

که در آن، شمارش کل، حاصل جمع همه‌ی مدخل‌های FP به‌دست آمده از شکل ۱-۲۳ است.

F_iها (i برابر با ۱ تا ۱۴) «ضرایب تنظیم» (VAF) بوده براساس پاسخ‌هایی که به پرسش‌های زیر داده می‌شوند، قابل تعیین هستند [Lon02]:

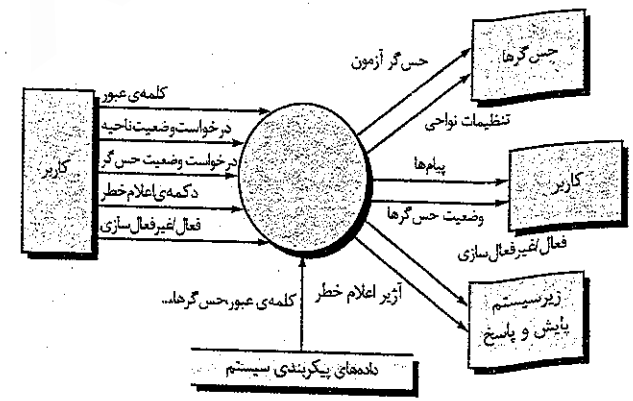
۱. آیا سیستم به تهیه‌ی پشتیبان و بازیابی قابل اطمینان نیاز دارد؟
۲. آیا ارتباطات داده‌ای موردنیاز است؟
۳. آیا عملیات‌های پردازشی توزیع شده وجود دارند؟
۴. آیا کارایی اهمیت دارد؟
۵. آیا سیستم در یک محیط عملیاتی موجود و پرکاربرد اجرا می‌شود؟
۶. آیا سیستم به مدخل داده‌های آنلاین نیاز دارد؟
۷. آیا مدخل داده‌های آنلاین نیاز به ساخت تراکنش ورودی روی عملیات یا صفحات نمایش چندگانه دارد؟

نکته‌ی کلیدی

برای فراهم‌ساختن شاخصی از پیچیدگی مسأله، از ضرایب تنظیم ارزش استفاده می‌شود.

۸. آیا فایل‌های اصلی به‌صورت آنلاین بهنگام می‌شوند؟
۹. آیا ورودی‌ها، خروجی‌ها، فایل‌ها و درخواست‌ها پیچیده‌اند؟
۱۰. آیا پردازش داخلی پیچیده است؟
۱۱. آیا کد طوری طراحی شده است که دوباره قابل استفاده باشد؟
۱۲. تبدیل و نصب در طراحی لحاظ شده است؟
۱۳. آیا سیستم برای نصب چندگانه در سازمان‌های متفاوت طراحی شده است؟
۱۴. آیا برنامه کاربردی طوری طراحی شده است که تغییرات را تسهیل کند و به آسانی توسط کاربر استفاده شود؟

با استفاده از مقیاسی که از صفر (عدم اهمیت یا لزوم اجرا) تا ۵ (مطلقاً ضروری) تغییر می‌کند، به هر یک از این پرسش‌ها پاسخ داده می‌شود. مقادیر ثابت در معادله (۱-۲۳) و ضرایب وزنی که در شمارش دامنه اطلاعاتی به‌کار برده شده‌اند، به‌طور تجربی به‌دست آمده‌اند.



شکل ۲-۲۳ مدل جریان داده‌ها برای نرم افزار *SafeHome*

برای نشان دادن کاربرد FP در این زمینه، نمایشی از مدل تحلیل را در نظر می‌گیریم که در شکل ۲-۲۳ نشان داده شده است. همان‌طور که در شکل می‌بینید، یک نمودار جریان داده‌ها (فصل ۷) برای یکی از عملکردهای نرم‌افزار *SafeHome* نشان داده شده است. این عملکرد، تعامل با کاربر را مدیریت می‌کند، یعنی کلمه‌ی عبور را از کاربر پذیرفته سیستم را فعال / غیرفعال می‌کند و اجازه می‌دهد تا وضعیت نواحی امنیتی و حس‌گرهای امنیتی به استحضار کاربر رسانده شود. این عملکرد، پیام‌هایی به نمایش درمی‌آورد و سیگنال‌های کنترلی مناسبی را به قطعات گوناگون سیستم امنیتی ارسال می‌کند.

نمودار جریان داده‌ها مورد ارزیابی قرار می‌گیرد تا مجموعه‌ای از موازین دامنه اطلاعات کلیدی مورد نیاز برای محاسبه‌ی معیار امتیاز عملکرد، تعیین گردد. سه ورودی کاربر: **کلمه‌ی عبور، دکمه اعلام خطر، و فعال‌سازی / غیرفعال‌سازی** در شکل به همراه دو درخواست خارجی (**درخواست وضعیت نواحی و درخواست وضعیت حس‌گرها**) نشان داده شده‌اند. یک فایل (**فایل پیکربندی سیستم**) نیز نشان داده شده است. دو خروجی کاربر (**پیام‌ها و وضعیت حس‌گرها**) و چهار واسط خارجی (**آزمون حس‌گرها، تنظیم نواحی، فعال‌سازی/غیرفعال‌سازی و آژیر**) نیز موجود هستند. این داده‌ها همراه با پیچیدگی مناسب در شکل ۳-۲۳ نشان داده شده‌اند.

مرجع وب

یک محاسبه‌گر آنلاین FP را می‌توانید در آدرس زیر بیابید: lrb.cs.uni-magdeburg.de/sw-eng/us/java/fp/

شمارش کل که در شکل ۳-۲۳ نشان داده شده است باید با استفاده از معادله (۱-۲۳) تنظیم شود. برای مثالی که دنبال می‌کنیم، فرض می‌کنیم $\Sigma(F_i)$ برابر با ۴۶ باشد (محصولی با پیچیدگی معتدل). لذا:

$$FP = 50 \times [0.65 + 0.01 \times 46] = 56$$

مقدار دامنه اطلاعاتی	شمارش		ضریب توزین: ساده	میانه	پیچیده		
ورودی‌های خارجی (EIها)	3	×	3	4	6	=	9
خروجی‌های خارجی (EOها)	2	×	4	5	7	=	8
درخواست‌های خارجی (EQها)	2	×	3	4	6	=	6
فایل‌های محلی داخلی (ILFها)	1	×	7	10	15	=	7
فایل‌های محلی خارجی (ELFها)	4	×	5	7	10	=	20
جمع کل							50

شکل ۳-۲۳ محاسبه امتیاز عملکرد برای یکی از قابلیت‌های عملیاتی *SafeHome*

تیم پروژه براساس مقدار FP به‌دست آمده از مدل تحلیل، می‌تواند اندازه‌ی پیاده‌سازی‌شده‌ی کلی عملکرد تعامل کاربر در *SafeHome* را برآورد کند. فرض کنید داده‌های گذشته نشان می‌دهد که یک FP حاکی از ۶۰ خط کد است (اگر از یک زبان شیءگرا استفاده شود) و به ازای هر نفر ماه کار، دوازده FP ساخته می‌شود. این داده‌های تاریخی، اطلاعات برنامه‌ریزی مهمی را در اختیار مدیر پروژه قرار می‌دهند که مبتنی‌بر مدل تحلیل است نه برآوردهای مقدماتی. به‌علاوه، فرض کنید که در پروژه‌های گذشته در اثنای تحلیل و مرورهای طراحی، به ازای هر FP سه خطا و در اثنای آزمون واحدها و آزمون انسجام به ازای هر FP چهار خطا پیدا شده باشد. این داده‌ها می‌توانند مهندسان نرم‌افزار را در سنجش کامل بودن مرورهایشان و فعالیت‌های آزمون آنها یاری دهد.

امورا و همکاران [Uem99] پیشنهاد می‌کنند که امتیازهای عملکرد را نیز می‌توان از نمودارهای ترتیب و کلاس‌های UML محاسبه کرد. در صورت علاقه بیشتر به این موضوع می‌توانید [Uem99] را ببینید.

> «به‌جای اینکه که فقط به این فکر باشیم چه معیار جدیدی ممکن است کاربرد داشته باشد، این پرسش اساسی‌تر را نیز از خود بپرسیم که با این معیارها چه خواهیم کرد؟»
> مایکل ماه و لری پوتنام

۲-۲-۲۳ معیارهایی برای کیفیت مشخصات

دیویس و همکاران وی [Dav93] فهرستی از ویژگی‌ها را پیشنهاد می‌کنند که می‌توان برای ارزیابی کیفیت مدل تحلیل و مشخصات خواسته‌های مربوط به‌کار برد: *مشخص بودن، کامل بودن، درست بودن، قابلیت درک، قابلیت وارسی، سازگاری داخلی و خارجی، قابلیت بایگانی، فشردگی، قابلیت ردگیری، اصلاح‌پذیری، دقت* و *قابلیت استفاده دوباره*. به‌علاوه، آنها متذکر می‌شوند که مشخصاتی با کیفیت بالا که به‌طور الکترونیکی ذخیره‌سازی می‌شوند، قابل اجرا یا حداقل قابل تفسیر هستند، اهمیت نسبی آنها مشخص می‌شود، پایدارند، شماره‌ی نسخه در آنها مشخص است، سازمان‌دهی شده‌اند، حاوی پی‌نوشت‌های مناسب هستند و در سطح مناسبی از جزئیات قرار دارند.

گرچه بسیاری از ویژگی‌های بالا ماهیتی به ظاهر کیفی دارند، دیویس و سایرین [Dav93] پیشنهاد می‌کنند که هر یک از آنها را می‌توان با استفاده از یک یا چند معیار نمایش داد. برای مثال، فرض می‌کنیم که در مشخصات نرم‌افزار n_r خواسته وجود دارد به‌طوری که

$$n_r = n_f + n_{nf}$$

که n_f تعداد خواسته‌های عملیاتی و n_{nf} تعداد خواسته‌های غیرعملیاتی (مثلاً کارایی) است.

دیویس برای تعیین ویژگی مشخص بودن خواسته‌ها، معیاری پیشنهاد می‌کند که مبتنی بر سازگاری تفسیر مسؤول مرور از خواسته‌هاست:

$$Q_1 = \frac{n_{ui}}{n_r}$$

n_{ui} تعداد خواسته‌هایی است که همه‌ی مسؤولان مرور تفسیر یکسانی از آنها داشته‌اند. هر چه مقدار Q به ۱ نزدیک‌تر شود، ابهام کمتری در مشخصات وجود دارد.

کامل بودن خواسته‌های عملیاتی را می‌توان با محاسبه نسبت زیر تعیین کرد:

$$Q_2 = \frac{n_u}{n_i \times n_s}$$

که n_u تعداد خواسته‌های عملیاتی منحصر به‌فرد، n_i تعداد ورودی‌ها (محرک‌های) تعریف شده یا گرفته شده از مشخصات و n_s تعداد حالت‌های مشخص شده است. نسبت Q_2 درصد قابلیت‌های عملیاتی لازمی را اندازه‌گیری می‌کند که برای سیستم مشخص شده است. ولی با خواسته‌های غیرعملیاتی کاری ندارد. برای تلفیق آنها در قالب یک معیار کلی برای ویژگی کامل بودن، باید حدی را در نظر بگیریم که خواسته‌ها اعتبارسنجی شده‌اند:

$$Q_3 = \frac{n_c}{n_c \times n_{nv}}$$

که n_c تعداد خواسته‌هایی است که درست تشخیص داده شده‌اند و n_{nv} تعداد خواسته‌هایی است که هنوز اعتبارسنجی نشده‌اند.

نکته‌ی کلیدی

با اندازه‌گیری خصوصیات مشخصه، این امکان وجود دارد که دیدی کمّی از مشخص‌بودن و کامل‌بودن به‌دست آید.

۲۳-۳ معیارهایی برای مدل طراحی

نمی‌توان باور کرد که طراحی یک هواپیمای نو، یک تراشه کامپیوتری جدید یا یک ساختمان اداری تازه، بدون تعریف موازین طراحی، تعیین معیارهایی برای جنبه‌های گوناگون کیفیت نرم‌افزاری و استفاده از آنها به‌عنوان راهنمایی برای شیوه تکامل یافتن طراحی، اجرا شود. با این حال، طراحی سیستم‌های نرم‌افزاری پیچیده، بدون انجام هرگونه اندازه‌گیری انجام می‌شود. جالب اینجا است که معیارهای طراحی برای نرم‌افزار در دسترس هستند، ولی اغلب مهندسان نرم‌افزار همچنان از وجود آنها ناآگاه هستند.

معیارهای طراحی برای نرم‌افزارهای کامپیوتری، همانند معیارهای نرم‌افزاری دیگر، کامل نیستند. بحث بر سر اثربخش بودن آنها و شیوه به‌کارگیری آنها همچنان ادامه دارد. بسیاری از کارشناسان استدلال می‌کنند که پیش از به‌کارگیری موازین طراحی باید تجربه بیشتری اندوخت. با این حال، طراحی بدون اندازه‌گیری، راهی غیرقابل قبول است.

در بخش‌هایی که به‌دنبال خواهد آمد، برخی معیارهای طراحی متداول‌تر برای نرم‌افزارهای کامپیوتری را بررسی می‌کنیم، هر یک می‌تواند دید بهتری در اختیار طراح بگذارد و همگی می‌توانند به طراح کمک کنند تا به سطح بالاتری از کیفیت دست پیدا کند.

۲۳-۳-۱ معیارهای طراحی معماری

در معیارهای طراحی معماری آنچه کانون توجه قرار می‌گیرد، خصوصیات معماری برنامه و اثربخشی پیمانه‌هاست. این معیارها، از نوع جعبه سیاه هستند، زیرا نیازی به اطلاع از کارکرد داخلی پیمانه‌های داخل سیستم ندارند.

کارد و گلاس [Car90] سه میزان برای پیچیدگی طراحی نرم‌افزار تعریف می‌کنند: پیچیدگی ساختاری، پیچیدگی داده‌ای و پیچیدگی سیستمی.

برای معماری‌های سلسله مراتبی (مانند معماری‌های فراخوانی و بازگشت)، *پیچیدگی ساختاری* پیمانه‌ی i به شیوه زیر تعریف می‌شود:

$$S(i) = f_{out}^2(i)$$

که در آن $f_{out}(i)$ توان خروجی پیمانه‌ی i است[1].

پیچیدگی داده‌ای، شاخصی از پیچیدگی را در واسط داخلی پیمانه‌ی i ارائه می‌کند و به‌صورت زیر تعریف می‌شود:

$$D(i) = \frac{V(i)}{f_{out}(i)+1}$$

که $V(i)$ تعداد متغیرهای ورودی و خروجی است که به پیمانه‌ی i تحویل داده می‌شوند یا از آن تحویل گرفته می‌شوند.

پیچیدگی سیستمی به‌صورت مجموع پیچیدگی داده‌ای و ساختاری تعریف می‌شود:

$$C(i) = S(i) + D(i)$$

با افزایش هر یک از این مقادیر پیچیدگی، مقدار کل پیچیدگی معماری سیستم نیز فزونی می‌یابد. به این ترتیب، احتمال افزایش کار لازم برای انسجام و آزمون بالا می‌رود.

فنتون [Fen91] چند معیار ریخت‌شناسی (مانند شکل) پیشنهاد می‌کند که مقایسه‌ی معماری‌های متفاوت را با استفاده از مجموعه‌ای از ابعاد صحیح، امکان‌پذیر می‌سازد. با توجه به معماری بازگشت و فراخوان شکل ۴-۲۳، معیارهای زیر را می‌توان تعریف کرد:

$$\text{اندازه} = n + a$$

که n تعداد گره‌ها (nodes) و a تعداد یال‌ها (arcs) ست. برای معماری نشان داده شده در شکل ۴-۲۳ داریم،

$$\text{اندازه} = 17 + 18 = 35$$

عمق = طولانی‌ترین مسیر از گره ریشه (بالا) تا یک گره برگ. برای معماری شکل ۴-۲۳، عمق برابر ۴ است.

پهنا = حداکثر تعداد گره‌ها در هر سطح از معماری. برای معماری شکل ۴-۲۳، پهنا برابر ۶ است.

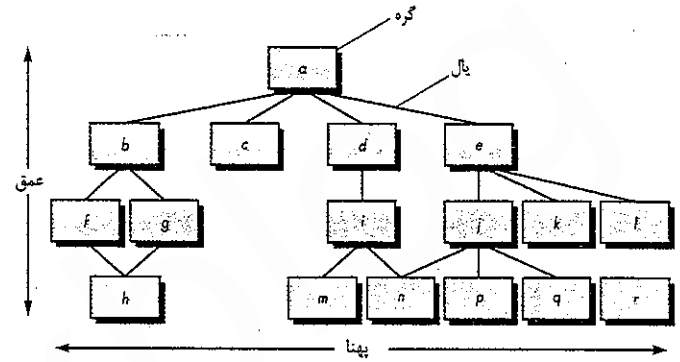
نکته‌ی کلیدی

معیارها می‌توانند دیدی از پیچیدگی داده‌ای و مرتبط با طراحی معماری در اختیار قرار دهند.

گالیله

«چیزی را اندازه بگیرید که قابل اندازه‌گیری باشد و آنچه را که قابل اندازه‌گیری نیست، قابل اندازه‌گیری کنید.»

[1] توان خروجی (fan-out) به‌عنوان تعداد پیمانه‌هایی تعریف می‌شود که بلافاصله در زیردست پیمانه‌ی i قرار دارند؛ یعنی تعداد پیمانه‌هایی که پیمانه‌ی i مستقیماً آنها را فرا می‌خواند.



شکل ۴-۲۳ معیارهای ریخت‌شناسی.

$r = a/n$ (نسبت یال‌ها به گره‌ها)، دانسیته‌ی اتصال معماری را اندازه‌گیری می‌کند و ممکن است شاخص ساده‌ای از اتصال معماری را به‌دست دهد. برای معماری شکل ۴-۲۳، $r = 18/17 \approx 1.06$.

فرمان‌دهی سیستم‌های نیروی هوایی ایالات متحده [USA81]، چند شاخص برای کیفیت نرم‌افزار توسعه داده است که مبتنی بر خصوصیات طراحی قابل سنجش یک برنامه‌ی کامپیوتری هستند. نیروی هوایی با به‌کارگیری مفاهیمی مشابه با مفاهیم پیشنهاد شده در استاندارد IEEE std.982.1-1988 [IEE94]، از اطلاعات به‌دست آمده از داده‌ها و طراحی معماری، یک شاخص کیفیت ساختار طراحی (DSQI) به‌دست می‌آورد که از صفر تا یک در تغییر است. مقادیر زیر برای محاسبه DSQI باید تعیین شود [Chs89]:

S_1 = تعداد کل پیمانه‌های تعریف شده در معماری برنامه

S_2 = تعداد پیمانه‌هایی که عملکرد درست آن‌ها به منبع ورود داده‌ها بستگی دارد یا این که داده‌هایی را تولید می‌کنند که در جای دیگر استفاده می‌شوند (به‌طور کلی، پیمانه‌های کنترلی و برخی پیمانه‌های دیگر، به‌عنوان بخشی از S_2 شمارش نمی‌شوند).

S_3 = تعداد پیمانه‌هایی که عملکرد درست آن‌ها به پردازش قبلی بستگی دارد.

S_4 = تعداد آیتم‌های بانک اطلاعاتی (شامل اشیای داده‌ای و همه‌ی صفاتی که اشیا را تعریف می‌کنند).

S_5 = تعداد کل آیتم‌های بانک اطلاعاتی منحصر به فرد.

S_6 = تعداد قطعات بانک اطلاعاتی (رکوردهای متفاوت یا اشیای منفرد)

S_7 = تعداد پیمانه‌هایی با تنها یک نقطه‌ی ورود و یک نقطه‌ی خروج (پردازش استثناها به‌عنوان خروجی چندگانه در نظر گرفته نمی‌شود)

هنگامی که مقادیر S_1 تا S_7 برای یک برنامه کامپیوتری تعیین شد، مقادیر میانی زیر را می‌توان محاسبه کرد:

ساختار برنامه: D_1، که D_1 به این صورت تعریف می‌شود: اگر طراحی معماری با به‌کارگیری یک روش متمایز توسعه یافته باشد (مثل طراحی جریان گرا یا طراحی شیءگرا)، آن گاه $D_1 = 1$ و در غیر این صورت، $D_1 = 0$.

«اندازه‌گیری را می‌توان نوعی انحراف از مسیر دانست. این انحراف از مسیر، ضروری است چرا که اکثر انسان‌ها قادر به تصمیم‌گیری هدف‌مند و واضح نیستند [بدون پشتیبانی کمّی]»

هورست زوس

استقلال پیمانه‌ها: $D_2 = 1 - \frac{S_2}{S_1}$

پیمانه‌هایی که به پردازش قبلی وابستگی ندارند: $D_3 = 1 - \frac{S_3}{S_1}$

اندازه بانک اطلاعاتی: $D_4 = 1 - \frac{S_5}{S_4}$

بخش‌بخش‌کردن بانک اطلاعاتی: $D_5 = 1 - \frac{S_6}{S_4}$

مشخصه‌ی ورود/خروج پیمانه: $D_6 = 1 - \frac{S_7}{S_1}$

با تعیین این مقادیر میانی، DSQI به شیوه‌ی زیر قابل محاسبه است:

$$DSQI = \sum w_i D_i$$

که i برابر با ۱ تا ۶ و w_i وزن نسبی هر کدام از مقادیر میانی بوده اهمیت آن را منعکس می‌سازد و $\sum w_i = 1$ (اگر همه D_iها به یک اندازه اهمیت داشته باشند، $w_i = 0.167$).

مقدار DSQI برای طراحی‌های قبلی را می‌توان تعیین کرد و با طراحی‌ای که در حال حاضر در حال توسعه است، مقایسه کرد. اگر DSQI به‌طور چشمگیری کوچک‌تر از میانگین باشد، کار طراحی و مرور بیشتری لازم است. به‌طور مشابه، اگر تغییرات عمده‌ای قرار است روی یک طراحی موجود اعمال شود، اثر آن تغییرات بر DSQI قابل محاسبه است.

۲-۳-۲۳ معیارهایی برای طراحی شیءگرا

در این خصوص که طراحی شیءگرا، امری موضوعی است، زیاد بحث شده است - طراح کارآزموده «می‌داند» که چگونه خصوصیات یک سیستم شیءگرا را مشخص کند تا خواسته‌های مشتری به‌طور اثربخش پیاده‌سازی شود. ولی به موازاتی که مدل طراحی شیءگرا از لحاظ اندازه و پیچیدگی رشد می‌کند، نمای عمیق‌تری از خصوصیات طراحی، می‌تواند برای یک طراح کارآزموده (که دیدی اضافی به‌دست می‌آورد) و طراح تازه‌کار (که شاخصی از کیفیت به‌دست می‌آورد) مفید باشد.

هنگام ارزیابی یک طراحی شیءگرا، کدام خصوصیات را می‌توان اندازه‌گیری کرد؟

ویتمایر [Whi97] طی یک بررسی مشروح درباره معیارهای نرم‌افزاری برای سیستم‌های شیءگرا، نُه خصوصیت قابل اندازه‌گیری را برای طراحی شیءگرا برمی‌شمرد:

اندازه. اندازه برحسب چهار دیدگاه تعریف می‌شود: جمعیت، حجم، طول و عملکرد. جمعیت با در نظر گرفتن یک شمارش آماری از موجودیت‌های شیءگرا از قبیل کلاس‌ها یا عملیات سنجیده می‌شود. موازین حجمی با موازین جمعیتی همسان هستند، ولی به‌صورت پویا - در یک بازه زمانی مفروض - جمع‌آوری می‌شوند. طول، میزانی از زنجیره‌ی عناصر طراحی متصل به یکدیگر است - مثلاً عمق یک درخت وراثت، میزانی از طول است). معیارهای عملکرد، شاخصی غیرمستقیم از مقداری هستند که از جانب برنامه کاربردی شیءگرا به مشتری تحویل شده است.

پیچیدگی (Complexity). همانند اندازه، چندین دیدگاه متفاوت برای پیچیدگی نرم‌افزار وجود دارد [Zus97]. ویتمایر، پیچیدگی را با بررسی چگونگی ارتباط میان کلاس‌ها در طراحی و برحسب خصوصیات ساختاری در نظر می‌گیرد.

اتصال (Coupling). اتصالات فیزیکی میان عناصر طراحی شیءگرا (مثل تعداد مشارکت‌های میان کلاس‌ها یا تعداد پیام‌های تبادل شده میان اشیاء) نشان‌گر اتصال در سیستم شیءگرا است.

کافی بودن (Sufficiency). ویتمایر، «کافی بودن» را به‌عنوان «حدی که یک انتزاع ، ویژگی‌های موردنیاز خود را پردازش می‌کند یا حدی که مؤلفه‌ای از طراحی خصوصیات موجود در انتزاع خود را پردازش می‌کند، (از دیدگاه یک کاربرد کنونی) تعریف می‌کند». به عبارت دیگر، می‌پرسیم: «این انتزاع (کلاس) چه خواصی را باید پردازش کند که برای من مفید واقع شود؟» [Whi97] در اصل، یک مؤلفه‌ی طراحی (مثلاً کلاس) در صورتی کافی خواهد بود که کلیه خواص شیء دامنه کاربردی را که مدل‌سازی می‌کند، به‌طور کامل منعکس سازد - به عبارت دیگر، انتزاع (کلاس) ویژگی‌های موردنیاز خود را پردازش می‌کند.

کامل بودن (Completeness). تنها تفاوت میان کامل بودن و کافی بودن «مجموعه ویژگی‌هایی است که انتزاع یا مؤلفه‌ی طراحی را در مقابل آنها مقایسه می‌کنیم.» [Whi97]. کافی بودن، انتزاع را از دیدگاه کاربرد کنونی مقایسه می‌کند. کامل بودن با پرسیدن این سؤال که «چه خواصی برای نمایش کامل شیء دامنه مسأله موردنیاز است؟» چند دیدگاه متفاوت را در نظر می‌گیرد. از آنجا که ملاک مربوط به کامل‌بودن، دیدگاه‌های متفاوتی را در نظر می‌گیرد، به‌طور غیرمستقیم، به قابلیت استفاده‌ی مجدد یک انتزاع اشاره دارد.

یکپارچگی (Cohessiveness). هر مؤلفه شیءگرا همانند همتای خود در نرم‌افزارهای سنتی، باید به شیوه‌ای طراحی شود که همه‌ی عملیات‌ها با یکدیگر کار کنند و به یک هدف معین و واحد دست پیدا کنند.

مقدماتی بودن (Primitiveness). مقدماتی بودن که خصوصیتی مشابه با سادگی است (و هم در مورد عملیات و هم کلاس‌ها به‌کار می‌رود)، میزان اتمی بودن یک عمل را می‌رساند - منظور از اتمی بودن آن است که عمل را نتوان از طریق یک سری عملیات موجود در کلاس ایجاد کرد. کلاسی که درجه بالایی از مقدماتی بودن را از خود نشان دهد، فقط عملیات مقدماتی را بسته‌بندی می‌کند.

مشابهت (Similarity). این میزان، نشان‌گر میزان شباهت دو یا چند کلاس از لحاظ ساختار، عملکرد، رفتار و/یا هدف آنها است.

ناپایداری (Volatility). چنان‌که پیش از این در این کتاب دیدیم، تغییرات طراحی هنگامی انجام می‌شود که خواسته‌ها اصلاح شوند یا هنگامی که اصلاحاتی که در بخش‌های دیگر یک برنامه کاربردی رخ داده‌اند، منجر به تطابق اجباری مؤلفه‌ی طراحی شوند. ناپایداری یک مؤلفه‌ی طراحی شیءگرا ، میزانی از احتمال وقوع تغییر را ارائه می‌دهد.

در حالت واقعی، معیارهای تکنیکی برای سیستم‌های شیءگرا را می‌توان نه تنها در مدل طراحی، بلکه در مدل تحلیل نیز به‌کار برد. در بخش‌های بعد، به دنبال معیارهایی خواهیم بود که شاخصی از کیفیت در سطح کلاس‌های شیءگرا و در سطح عملیات ارائه دهد. علاوه بر این، معیارهایی نیز مورد بررسی قرار می‌گیرند که برای مدیریت پروژه و آزمون قابل استفاده باشند.

> «بسیاری از تصمیم گیری‌هایی که قبلاً می‌بایست با اتکا بر نظر عامه و اساطیر انجام شوند، اکنون با به کارگیری داده‌های کمی قابل انجام هستند.»
>
> - آشکات ویتمایر

۳-۳-۲۳ معیارهای شیءگرا - مجموعه معیارهای CK

کلاس، واحد بنیادی سیستم‌های شیءگراست. بنابراین موازین و معیارهای یک کلاس شیءگرا، سلسله مراتب کلاس‌ها و مشارکت‌های میان کلاس‌ها برای مهندس نرم‌افزاری که باید کیفیت طراحی را ارزیابی کند، بسیار ارزشمند است. در فصول قبل دیدیم که کلاس، عملیات (پردازش) و صفات (داده‌ها) را بسته‌بندی می‌کنند. این کلاس غالباً «والد» زیرکلاس‌هایی (که گاه فرزند خوانده می‌شوند) است که صفات و عملیات آن را به ارث می‌برند. کلاس غالباً با کلاس‌های دیگر مشارکت دارد. هر یک از این خصوصیات را می‌توان به‌عنوان مبنایی برای اندازه‌گیری به‌کار برد.[1]

یکی از مجموعه معیارهای شیءگرا که بسیار به آن رجوع می‌شود، توسط چیدامبر و کمرر [Chi94] پیشنهاد شده است. در این مجموعه که غالباً به اختصار آن را مجموعه معیارهای CK می‌خوانند، شش معیار مبتنی بر کلاس برای سیستم‌های شیءگرا ذکر شده است.[2]

متدهای موزون به‌ازای هر کلاس (WMC). فرض کنید n متد با پیچیدگی‌های C_1، C_2، ... C_n برای کلاس $\mathbf{C}$ تعریف می‌شوند. معیار پیچیدگی مشخصی که انتخاب می‌شود (مثلاً پیچیدگی سیکلوماتیک) باید طوری بهنجار شود که پیچیدگی اسمی یک متد، مقدار ۱٫۰ را به خود بگیرد. به‌ازای i برابر با ۱ تا n

$$\mathrm{WMC} = \Sigma C_i$$

تعداد متدها و پیچیدگی آنها، شاخص مناسبی از کار لازم برای پیاده‌سازی و آزمون یک کلاس است. به‌علاوه، هر چه تعداد متدها بیشتر باشد، درخت وراثت پیچیده‌تر می‌شود (همه زیرکلاس‌ها، وارث متدهای والدین خود هستند). سرانجام، هنگامی که تعداد متدها برای یک کلاس رشد می‌کند، این احتمال وجود دارد که هر چه بیشتر ویژه‌ی برنامه کاربردی شود و از این رو، استفاده‌ی مجدد را محدود سازد. به همه‌ی این دلایل، WMC را باید هر چه کمتر نگه داشت.

گرچه شمارش تعداد متدهای یک کلاس، نسبتاً صریح به‌نظر می‌رسد، مسأله درواقع پیچیده‌تر از آن چیزی است که به نظر می‌رسد و یک روش شمارش سازگار [Chu94] باید پیش گرفته شود.

عمق درخت وراثت (DIT). این معیار به‌عنوان «حداکثر طول مابین گره و ریشه درخت» تعریف می‌شود [Chi94] . مقدار DIT برای سلسله مراتب کلاس‌هایی که در شکل ۱-۲۴ نشان داده شده است برابر با ۴ است. به موازاتی که DIT رشد می‌کند، این احتمال وجود دارد که کلاس‌های سطح پایین‌تر متدهای بسیاری را به ارث ببرند. این موضوع، هنگام پیش‌بینی رفتار یک کلاس منجر به مشکلات بالقوه‌ای می‌شود. سلسله مراتبی با عمق زیاد (DIT بزرگ) نیز منجر به افزایش پیچیدگی طراحی می‌شود. از دیدگاه مثبت، مقادیر DITنشان‌گر آن هستند که امکان استفاده‌ی مجدد از متدها زیاد است.

---

[1] شایان ذکر است که اعتبار برخی معیارهای ذکرشده در این فصل در حال حاضر در متون فنی موضوع بحث و جدل است. مدافعان نظریه اندازه‌گیری در پی درجه‌ای از رسمیت هستند که معیارهای شیءگرا فاقد آن هستند. به هرحال، منطقی است که بگوییم معیارهای ذکرشده، دید مفیدی از نرم‌افزار به‌دست می‌دهند.

[2] چیدامبر، دارسی و کمرر به جای *عملیات* از *متد* استفاده می‌کنند. ما نیز به تبع آن‌ها در این بخش از این واژه استفاده کردیم.

تعداد فرزندان (NOC). زیرکلاس‌هایی که زیردست بلافصل یک کلاس هستند، فرزندان آن کلاس نامیده می‌شوند. در شکل ۵-۲۳ کلاس C_2 سه فرزند دارد: C_{21}، C_{22} و C_{23}. با رشد تعداد فرزندان، استفاده‌ی مجدد فزونی می‌یابد، ولی این هم درست است که با افزایش NOC، انتزاع نشان داده شده توسط کلاس والد، ممکن است کمرنگ شود. یعنی این احتمال هست که برخی از فرزندان واقعاً اعضای مناسبی از کلاس والد نباشند. با افزایش NOC، مقدار آزمون (موردنیاز برای امتحان کردن هر فرزند در حیطه عملیاتی آن) نیز افزایش می‌یابد.

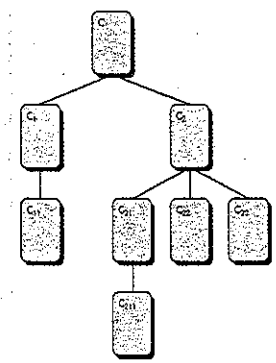
اندرز

دو مفهوم اتصال و یکپارچگی در هر دو نوع نرم‌افزار سنتی و شیءگرا کاربرد دارند. اتصال کلاس‌ها را در سطحی پایین و یکپارچگی عملیاتی را در سطحی بالا حفظ کنید.

اتصال میان کلاس‌های اشیا (CBO). مدل CRC (فصل ۶) را می‌توان برای تعیین مقدار CBO به‌کار برد. در اصل، CBO برابر با تعداد مشارکت‌های هر کلاس در کارت شاخص CRC آن است.[1] با افزایش CBO این احتمال وجود دارد که قابلیت استفاده‌ی مجدد یک کلاس کاهش یابد. مقادیر بالای CBO باعث دشوار شدن اصلاحات و آزمون می‌شود. به‌طور کلی، مقادیر CBO برای هر کلاس را باید در حدی منطقی پایین نگه داشت. این موضوع با دستورالعمل کلی مبنی بر کاهش اتصال در نرم‌افزارهای سنتی سازگاری دارد.

پاسخ کلاس (RFC). مجموعه پاسخ‌های یک کلاس عبارت از «مجموعه متدهایی است که می‌توانند در پاسخ به پیام دریافت شده توسط یک شیء از آن کلاس، اجرا شوند». با افزایش RFC، کار لازم برای آزمون نیز افزایش می‌یابد، زیرا دنباله‌ی آزمون‌ها (فصل ۱۹) رشد می‌کند. همچنین اگر RFC رشد کند، پیچیدگی کلی طراحی افزایش می‌یابد.

فقدان یکپارچگی در متدها (LCOM). هر متد در کلاس C به یک یا چند صفت (که به آنها متغیرهای نمونه نیز گفته می‌شود) دستیابی دارد. LCOM عبارت از تعداد متدهایی است که به یک یا چند صفت مشترک دستیابی دارند.[2] اگر هیچ متدی به یک صفت مشترک دستیابی نداشته باشد، در آن صورت $LCOM = 0$ است. برای نشان دادن مواردی که در آن $LCOM \neq 0$ است، کلاسی با شش متد را در نظر بگیرید. چهارتا از این متدها دارای یک یا چند متد مشترک هستند (یعنی به صفات مشترکی دستیابی دارند). بنابراین، $LCOM = 4$ است. اگر LCOM بزرگ باشد، متدها ممکن است از طریق صفات با یکدیگر پیوسته شوند. این موضوع باعث افزایش پیچیدگی طراحی کلاس می‌شود. به‌طور کلی، مقادیر بالای LCOM نشان می‌دهد که کلاس را می‌توان با تقسیم آن به دو یا چند کلاس جداگانه بهتر طراحی کرد. گرچه مواردی وجود دارد که در آنها مقدار بالایی از LCOM قابل توجیه است، بالا نگه داشتن یکپارچگی، و در نتیجه پایین نگه داشتن LCOM مطلوب است.[3]



شکل ۵-۲۳ سلسله مراتب کلاس‌ها.

[1] اگر از کارت‌های شاخص CRC به‌طور دستی استفاده می‌کنید، کامل‌بودن و سازگاری را باید پیش از تعیین مطمئن CBO ارزیابی کنید.

[2] تعریف رسمی، قدری پیچیده‌تر است. برای جزئیات بیشتر، [Chi94] را ببینید.

[3] معیار LCOM در برخی شرایط دید مفیدی به‌دست می‌دهد، ولی در شرایطی هم می‌تواند گمراه کننده باشد. برای مثال، با بسته‌بندی اتصال در داخل یک کلاس، یکپارچگی کل سیستم بالا می‌رود. بنابراین، از یک لحاظ بسیار مهم، LCOM بالا درواقع به این معنی خواهد بود که کلاس ممکن است همبندی بالاتری داشته باشد نه پایین‌تر.

SafeHome

همکارگیری معیارهای *CK*

صحنه: کابین وینود

نقش‌آفرینان: وینود، جیمی، شکیرا و اد- اعضای تیم مهندسی نرم‌افزار که همچنان به‌کار روی طراحی در سطح مؤلفه‌ها و طراحی موارد آزمون مشغول هستند.

گفتگو:

وینود: شماها فرصت کردید شرح مجموعه معیارهای CK را که چهارشنبه برایتان ارسال کردم، بخوانید و اندازه‌گیری‌ها را انجام بدهید؟

شکیرا: خیلی پیچیده نبود. من، همان‌طور که پیشنهاد کرده بودی، به نمودارهای ترتیب و کلاس‌های UML خودم برگشتم و DIT، REC و LCOM را به‌طور خودی شمردم. مدل CRC را نتوانستم پیدا کنم و به همین خاطر CBO را شمارش نکردم.

جیمی (با لبخند): تو نتوانستی مدل CRC را پیدا کنی چون پیش من بود.

شکیرا: برای همین عاشق این تیم هستم. ارتباطات عالی.

وینود: من هم شمارش‌هایم را انجام دادم... شماها برای معیارهای CK اعدادی به‌دست آورید؟ [جیمی و اد به نشانه پاسخ مثبت سری تکان می‌دهند]

جیمی: چون من کارت‌های CRC داشتم، یک نگاهی به CBO کردم و در اکثر کلاس‌ها، کاملاً یکنواخت به نظر می‌رسید. یک استثنا وجود داشت که متوجه‌اش شدیم.

اد: چند تا کلاس هست که RFC در آن‌ها در مقایسه با میانگین‌ها خیلی بالاست. شاید لازم باشد به نحوه ساده کردن آن‌ها نگاهی بیندازیم.

جیمی: شاید بله، شاید هم نه. من هنوز نگران وقتم و نمی‌خواهم چیزهایی را درست کنم که اصلاً خراب نشده‌اند.

وینود: موافقم. شاید بهتر باشد به دنبال کلاس‌هایی بگردیم که حداقل در دو یا چند معیار CK اعداد خوبی ندارند.

شکیرا (نگاهی به فهرست کلاس‌های اد با RFC بالا می‌اندازد): ببین این کلاس را نگاه کن، هم LCOM بالایی دارد و هم RFC بالا.

وینود: بله. این طور فکر می‌کنم... پیاده‌سازی‌اش به دلیل پیچیدگی، سخت می‌شود و البته آزمون‌اش هم به همین دلیل مشکل می‌شود. احتمالاً بهتر است دو کلاس جداگانه طراحی کنیم تا به همان رفتار دست پیدا کنیم.

جیمی: فکر می‌کنی اصلاح این کلاس باعث صرفه‌جویی در وقت بشود؟

وینود: در درازمدت، بله.

۲۳-۴-۳ معیارهای شیءگرا - مجموعه معیارهای MOOD

هریسون، کانسل و نیتی [Har98] مجموعه‌ای از معیارها برای طراحی شیءگرا (MOOD) پیشنهاد کرده‌اند که برای خصوصیات طراحی شیءگرا شاخص‌هایی کمّی فراهم می‌آورند. نمونه‌هایی از معیارهای MOOD در زیر آمده است.

فاکتور وراثت متدها (MIF). میزانی که معماری کلاس‌ها در یک سیستم شیءگرا ، برای متدها (عملیات) و صفات از وراثت استفاده می‌کند و به‌صورت زیر تعریف می‌شود:

$$MIF = \frac{\Sigma M_i(C_i)}{\Sigma M_a(C_i)}$$

که در آن جمع روی i برابر با ۱ تا TC بسته می‌شود. TC به‌عنوان تعداد کل کلاس‌ها در معماری تعریف می‌شود، C_i کلاس موجود در معماری است و

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

که $M_a(C_i)$ تعداد متدهایی که می‌توان همراه با C_i اجرا کرد، $M_d(C_i)$ تعداد متدهای اعلان شده در کلاس و $M_i(C_i)$ تعداد متدهای به ارث برده شده (و نه همنام) در C_i است. مقدار MIF (فاکتور وراثت صفت، AIF نیز به شیوه‌ای مشابه قابل تعریف است). شاخصی از تأثیر وراثت در نرم‌افزار شیءگرا است.

فاکتور اتصال (CF). قبلاً در این فصل متذکر شدیم که اتصال، شاخصی از اتصال میان عناصر طراحی شیءگراست. در مجموعه معیارهای MOOD ، اتصال به‌صورت زیر تعریف می‌شود:

$$CF = \Sigma_i \Sigma_j is\_client \frac{(C_i, C_j)}{T_C^2 - T_C}$$

که در آن، جمع روی i برابر با ۱ تا T_C و روی jبرابر با ۱ تا T_C بسته می‌شود. اگر رابطه‌ای میان کلاس کلاینت، یعنی C_C و کلاس سرور، یعنی C_S وجود داشته باشد، تابع is_client به‌صورت $is\_client = 1$ تعریف می‌شود و در غیر این صورت $is\_client = 0$ تعریف می‌شود.

گرچه عوامل فراوانی بر پیچیدگی، قابلیت درک و قابلیت نگهداری نرم‌افزار مؤثرند، منطقی است که نتیجه بگیریم با افزایش CF، پیچیدگی نرم‌افزار شیءگرا نیز افزایش می‌یابد و قابلیت درک، قابلیت نگهداری و پتانسیل استفاده‌ی مجدد نیز ممکن است کاهش یابد.

هریسون و همکاران [Har98b] تحلیل مفصلی از MIF، CF و PF همراه با معیارهای دیگر ارائه می‌دهند و اعتبار آنها را برای استفاده در ارزیابی کیفیت طراحی، مورد بررسی قرار می‌دهند.

۵-۳-۲۳ معیارهای شیءگرا پیشنهادشده توسط لورنتس و کید

لورنتس و کید در کتاب خود که در باب معیارهای شیءگراست [Lor94]، معیارهای مبتنی بر کلاس را به چهار گروه گسترده تقسیم می‌کنند: اندازه، وراثت، داخلی و خارجی. معیارهای اندازه‌گرا برای کلاس شیءگرا ، بر شمارش صفات و عملیات مربوط به یک کلاس و مقادیر میانگین برای کل سیستم شیءگرا تأکید دارند. معیارهای مبتنی بر وراثت، بر شیوه استفاده‌ی مجدد عملیات از طریق سلسله مراتب کلاس‌ها تأکید می‌ورزند. معیارهای داخلی کلاس به یکپارچگی (بخش ۳-۳-۲۳) و مسائل مرتبط با کدنویسی مربوط می‌شوند و معیارهای خارجی، اتصال و استفاده‌ی مجدد را مورد بررسی قرار می‌دهند. نمونه‌هایی از معیارهای پیشنهادی لورنتس و کید در زیر می‌آید:

اندازه کلاس (CS). اندازه کلی کلاس را می‌توان با تعیین موازین زیر سنجید:

- تعداد کل عملیات (چه ارثی و چه خصوصی) که در کلاس بسته‌بندی شده‌اند.
- تعداد صفات (چه ارثی و چه خصوصی) که توسط کلاس بسته‌بندی شده‌اند.

اندرز

طی مرور مدل تحلیل، کارت‌های CRC، شاخصی منطقی از مقادیر قابل انتظار برای CS فراهم می‌آورند. اگر به کلاسی با تعداد زیاد مسؤولیت‌ها برخوردید، به افراز کردن آن بیندیشید.

معیار WMC که چیدامبر و کمرر پیشنهاد کرده‌اند (بخش ۳-۳-۲۳) نیز یک میزان وزنی از اندازه کلاس است. همان‌طور که پیش از این ذکر شد، مقادیر بزرگ CS نشان می‌دهد که کلاس ممکن است مسؤولیت‌های بسیار بزرگی داشته باشد. این موضوع باعث کاهش قابلیت استفاده‌ی مجدد کلاس و پیچیده‌شدن پیاده‌سازی و آزمون می‌شود. به‌طور کلی، در تعیین اندازه کلاس، باید به صفات و عملیات ارثی یا عمومی، وزن بیشتری داده شود [Lor94] . صفات و عملیات‌های خصوصی، تخصصی‌تر شده در طراحی متمرکزترند. میانگین‌های مربوط به تعداد صفات و عملیات کلاس را نیز می‌توان محاسبه کرد. هرچه مقادیر میانگین بیشتر باشد، احتمال آنکه کلاس‌های داخل سیستم را بتوان بیشتر مورد استفاده‌ی مجدد قرار داد، بیشتر است.

۶-۳-۲۳ معیارهای طراحی در سطح مؤلفه‌ها

این معیارها بر خصوصیات داخلی مؤلفه‌های نرم‌افزار تأکید دارند و شامل موازین یکپارچگی، اتصال و پیچیدگی پیمانه می‌شوند. این موازین می‌توانند به مهندس نرم‌افزار کمک کنند تا درباره کیفیت یک طراحی در سطح مؤلفه‌ها قضاوت کند.

معیارهای ارائه شده در این بخش، حکم «جعبه شیشه‌ای» را دارند، زیرا به آگاهی از کارکرد داخلی پیمانه نیاز است. هنگامی که طراحی رویه‌ای به پایان رسید، معیارهای طراحی در سطح مؤلفه‌ها را می‌توان به‌کار برد. به طریق دیگر، ممکن است استفاده از آنها را تا پایان مرحله کدنویسی به تعویق انداخت.

معیارهای یکپارچگی. بیمان و اوت [Bie94] مجموعه‌ای از معیارها را تعریف می‌کند که شاخصی از یکپارچگی (فصل ۱۳) پیمانه به‌دست می‌دهد. این معیارها برحسب پنج مفهوم و میزان تعریف می‌شوند:

برش داده‌ها (Data Slice). به بیان ساده، برش داده‌ها عقبگرد در پیمانه‌ای است که در جستجوی مقادیری است که بر مکانی از پیمانه مؤثرند که عقبگرد از آنجا شروع شده است. لازم به ذکر است که هم برش‌های داده‌ای و هم برش‌های برنامه‌ای را (که بر دستورها و شرط‌ها تأکید دارند) می‌توان تعریف کرد.

نشانه‌های داده‌ای (Data Tokens). متغیرهای تعریف شده برای یک پیمانه را می‌توان به‌صورت نشانه‌های داده‌ای آن پیمانه تعریف کرد.

نشانه‌های چسبی (Glue Tokens). مجموعه‌ای از نشانه‌های داده‌ای که روی یک یا چند برش داده‌ای قرار می‌گیرند.

نشانه‌های اَبَرچسبی (Superglue Tokens). مجموعه‌ای از نشانه‌های داده‌ای که در همه‌ی برش‌های داده‌ای یک پیمانه مشترک است.

استحکام (Stickiness). استحکام نسبی یک نشانه چسبی، با تعداد برش‌های داده‌ای که آنها را به هم پیوند می‌زند، نسبت مستقیم دارد.

بیمان و اوت، معیارهایی جهت *یکپارچگی عملیاتی قوی* (SFC)، *یکپارچگی عملیاتی ضعیف* (WFC) و *چسبندگی* (حد نسبی پیوند دادن برش‌های داده‌ای توسط نشانه‌های چسبی) توسعه داده‌اند. بحث مفصلی از معیارهای بیمان و اوت را خود آنها [Bie94] ارائه داده‌اند.

نکته‌ی کلیدی

امکان محاسبه‌ی موازین مربوط به استقلال عملیاتی- اتصال و یکپارچگی- برای یک مؤلفه و به‌کارگیری این موازین در ارزیابی کیفیت طراحی وجود دارد.

معیارهای اتصال (Coupling Metrics). اتصال پیمانه، شاخصی از «درستی» پیمانه در مقابل پیمانه‌های دیگر، داده‌های سراسری و محیط خارجی به‌دست می‌دهد. در فصل ۹، اتصال را به‌طور کیفی مورد بحث قرار دادیم.

داما [Dha95] معیاری برای اتصال پیمانه‌ها پیشنهاد کرده است که شامل اتصال جریان کنترل و داده‌ای، اتصال سراسری و اتصال محیطی می‌شود. موازین لازم برای محاسبه اتصال پیمانه‌ها، برحسب هر یک از سه نوع اتصال فوق‌الذکر تعریف می‌شوند.

برای اتصال جریان کنترلی و داده‌ای:

d_i=تعداد پارامترهای داده‌ای ورودی

c_i=تعداد پارامترهای کنترلی ورودی

d_o=تعداد پارامترهای داده‌ای خروجی

c_o=تعداد پارامترهای کنترلی خروجی

برای اتصال سراسری:

g_d=تعداد متغیرهای سراسری که به‌عنوان داده به‌کار می‌روند

g_c=تعداد متغیرهای سراسری که به‌عنوان کنترل به‌کار می‌روند

برای اتصال محیطی:

w=تعداد پیمانه‌هایی که فراخوانده می‌شود (توان خروجی)

r= تعداد پیمانه‌های که پیمانه موردنظر را فرا می‌خوانند (توان ورودی)

با استفاده از این موازین، یک شاخص اتصال پیمانه، m_c، به‌صورت زیر تعریف می‌شود:

$$m_c = \frac{k}{M}$$

که در آن k یک ثابت تناسب است و

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$$

مقادیر a، b، c و k باید به‌طور تجربی به‌دست آید.

هرچه مقدار m_c بزرگتر شود، اتصال کلی پیمانه کوچکتر است. برای آنکه معیار اتصال متناسب با خود اتصال افزایش یابد، معیار اتصال به‌صورت زیر تعریف می‌شود:

$$C = 1 - m$$

که در آن درجه اتصال با افزایش مقدار M بالا می‌رود.

معیارهای پیچیدگی. انواع معیارهای نرم‌افزاری را می‌توان محاسبه کرد تا پیچیدگی جریان کنترلی تعیین شود. بسیاری از آنها مبتنی بر گراف جریان هستند. همان‌طور که در فصل ۱۸ بحث شد، گراف، نمایشی است متشکل از گره‌ها و پیوندها (یال‌ها). هنگامی که پیوندها (یال‌ها) جهت‌دار باشند، گراف جریان، یک گراف جهت‌دار می‌شود.

مک کیب [McC94] چند کاربرد مهم برای معیارهای پیچیدگی برمی‌شمرد:

معیارهای پیچیدگی را می‌توان برای پیش‌بینی اطلاعات مهم مربوط به قابلیت اطمینان و قابلیت نگهداری سیستم‌های نرم‌افزاری از روی تحلیل خودکار کد منبع (یا اطلاعات طراحی رویه‌ای) به‌کار برد. معیارهای پیچیدگی همچنین در اثنای پروژه نرم‌افزاری، بازخوردی به‌دست می‌دهند که به کنترل (فعالیت طراحی) کمک می‌کند. در اثنای آزمون و نگهداری، اطلاعات مفصلی درباره پیمانه‌ها فراهم می‌آورند که نقاط ضعف را مشخص می‌کنند.

پرکاربردترین (و بحث‌انگیزترین) معیار پیچیدگی برای نرم‌افزارهای کامپیوتری، پیچیدگی سیکلوماتیک است که نخستین بار توماس مک کیب [McC89] و [McC76] آن را مطرح کرده است و در فصل ۱۸ به‌طور مفصل بحث شد.

نکته‌ی کلیدی
پیچیدگی سیکلوماتیک تنها یکی از چندین معیار پیچیدگی است.

زوس ([Zus97]، [Zus97]) بحثی همه‌جانبه در خصوص بیش از هجده گروه متفاوت از معیارهای پیچیدگی نرم‌افزار ارائه می‌دهد. نویسنده، در هر گروه تعاریفی اساسی برای معیارها ارائه می‌دهد (مثلاً چند شکل متفاوت از معیار پیچیدگی سیکلوماتیک وجود دارد) و سپس هر کدام را تحلیل و نقد می‌کند. کار زوس جامع‌ترین کار منتشر شده تاکنون است.

۲۳-۳-۷ معیارهای عمل‌گرا (Operation-Oriented Metrics)

از آنجا که کلاس، واحد اصلی در سیستم‌های شیءگراست، برای عملیاتی که در یک کلاس جای دارند، معیارهای معدودتری پیشنهاد شده است. چرچر و شپرد [Chu95] در این باره چنین می‌گویند: «نتایج مطالعات اخیر نشان می‌دهد که متدها تمایل دارند که هم از نظر تعداد دستورها و هم از نظر پیچیدگی منطقی، کوچک باقی بمانند [Wil93] و این نشان می‌دهد که ساختار اتصالی یک سیستم ممکن است مهمتر از محتویات تک‌تک پیمانه‌ها باشد.» ولی با بررسی خصوصیات میانگین مربوط به متدها (عملیات‌ها) به چند مورد می‌توان پی برد. سه معیار ساده که توسط لورنتس و کید [Lor94] پیشنهاد شده‌اند، در زیر آورده شده‌اند:

اندازه میانگین متد (OS_{avg}). گرچه تعداد خطوط کد را می‌توان به‌عنوان شاخصی برای اندازه متد در نظر گرفت، موازین LOC ، دچار مشکلات بحث شده در فصل ۴ هستند. از این رو، تعداد پیام‌های ارسال شده توسط متد، راه دیگری برای سنجش اندازه‌ی عمل فراهم می‌آورد. با افزایش تعداد پیام‌های ارسال شده توسط یک متد، این احتمال وجود دارد که مسؤولیت‌ها به خوبی در داخل کلاس تخصیص داده نشده باشند.

پیچیدگی عملیات (OC) . پیچیدگی یک عملیات را می‌توان با به کارگیری هر کدام از معیارهای پیچیدگی محاسبه کرد که برای نرم‌افزارهای سنتی پیشنهاد شده است [Zus90] . از آنجا که عملیات‌ها را باید به یک مسؤولیت خاص محدود کرد، طراح باید بکوشد تا OC را تا حد امکان پایین نگه دارد.

تعداد میانگین پارامترها به‌ازای هر متد (NP_{avg}).هر چه تعداد پارامترهای مدل بزرگتر باشد، مشارکت میان اشیاء پیچیده‌تر می‌شود. NP_{avg} تا حد امکان باید پایین نگه داشته شود.

۲۳-۳-۸ معیارهای طراحی واسط کاربر

گرچه درباره طراحی واسطهای انسان - کامپیوتر، مطالب فراوانی نگاشته شده است (فصل ۱۱)، درباره معیارهای مربوط به کیفیت و قابلیت استفاده از واسط اطلاعات نسبتاً کمی منتشر شده است.

سیرز [Sea93] پیشنهاد می‌کند که *مناسب بودن چیدمان* (LA) را می‌توان به‌عنوان یک معیار طراحی باارزش برای واسط‌های انسان – کامپیوتر به‌کار برد. در یک GUI متداول، از موجودیت‌های چیدمان – آیکون‌های گرافیکی، متون، منوها، پنجره‌ها و نظایر آن – برای کمک به کاربر در انجام امور استفاده می‌شود. برای انجام یک وظیفه با استفاده از GUI، کاربر باید از یک موجودیت چیدمان به بعدی حرکت کند. موقعیت مطلق و نسبی هر موجودیت از چیدمان، فراوانی استفاده از آن و «هزینه» گذار از یک موجودیت چیدمان به دیگری، در مناسب بودن کاربرد سهیم هستند.

مطالعه‌ای روی معیارهای مربوط به صفحات وب [Ivo01] نشان می‌دهد که خصوصیات ساده‌ی عناصر چیدمان نیز ممکن است تأثیر چشمگیری بر کیفیت مشاهده شده از سوی طراحی GUI داشته باشد، تعداد واژه‌ها، پیوندها، گرافیک‌ها، رنگ‌ها و فونت‌ها (و سایر مشخصات) موجود در یک صفحه وب بر پیچیدگی و کیفیت آن صفحه تأثیر می‌گذارد.

لازم به ذکر است که انتخاب یک طراحی GUI را می‌توان به کمک معیارهایی همچون LA انجام داد، ولی آنچه که در نهایت انتخاب می‌شود باید براساس نیاز کاربر روی نمونه اولیه باشد. نیلسن و لوی [Nie94] گزارش می‌کند که «... اگر انتخاب از میان (طراحی‌های) واسط براساس آرای کاربران صورت پذیرد، امکان موفقیت بسیار زیاد است. کاربران، میانگینی از کارایی را ارائه می‌دهند و رضایت آنها از GUI بسیار مهم است».

> می‌توانید حداقل یک اصل از طراحی واسط کاربر را با پر کردن یک ماشین ظرف‌شویی باد بگیرید. اگر زیادی آن را پر کنید، هیچ چیز پاکیزه نخواهد شد.
>
> تاشنایین

۲۳-۴ معیارهای طراحی برای برنامه‌های تحت وب

مجموعه‌ی مفیدی از موازین و معیارها برای برنامه‌های تحت وب، پاسخ کمّی به سؤالات زیر فراهم می‌سازد:

- آیا واسط کاربر باعث بالا رفتن قابلیت استفاده می‌شود؟
- آیا زیبایی‌شناسی برای دامنه کاربرد مناسب است و کاربر از آن لذت می‌برد؟
- آیا محتویات به گونه‌ای طراحی شده است که حداکثر اطلاعات را با کمترین تلاش در اختیار بگذارد.
- آیا گشت‌وگذار به‌طور صریح انجام می‌شود و بازدهی دارد؟
- آیا معماری برنامه‌های تحت وب طوری طراحی شده است که اهداف و مقاصد ویژه‌ی کاربران برنامه تحت وب، ساختار محتویات و قابلیت‌های عملیاتی و جریان گشت‌وگذاری لازم برای به‌کارگیری اثربخش سیستم را پاسخ گو باشد؟
- آیا مؤلفه‌ها به گونه‌ای طراحی شده‌اند که پیچیدگی روالی را کاهش دهند و صحت، قابلیت اطمینان و کارایی را بهبود بخشند؟

امروزه، هر کدام از این پرسش‌ها را می‌توان تنها به‌طور کیفی پاسخ داد، زیرا هنوز مجموعه‌ای اعتبارسنجی شده از معیارها که پاسخ‌های کمّی فراهم آورد، وجود ندارد.

در پاراگراف‌هایی که به دنبال خواهد آمد، روشی برای نمونه‌برداری از معیارهای طراحی برنامه‌های تحت وب ارائه شد که در متون پیشنهاد شده است. شایان ذکر است که بسیاری از این معیارها هنوز اعتبارسنجی نشده‌اند و باید آن‌ها را با احتیاط به‌کار برد.

> **اندرز**
>
> بسیاری از این معیارها در تمامی معیارهای کاربر قابل استفاده بوده باید آن‌ها را همراه با معیارهای ارائه‌شده در بخش ۸-۳-۲۳ به‌کار برد.

معیارهای واسط. برای برنامه‌های تحت وب، موازین زیر را برای واسط‌ها می‌توان در نظر بگیرید:

معیار پیشنهادی	شرح
مناسب بودن چیدمان	بخش ۸-۳-۲۳
پیچیدگی چیدمان	تعداد نواحی متمایزی[1] که برای یک واسط تعریف می‌شوند.
پیچیدگی نواحی چیدمان	تعداد میانگین پیوندهای به‌ازای هر ناحیه
پیچیدگی شناختی	تعداد میانگین آیتم‌های متمایزی که کاربر باید پیش از انجام یک گشت‌وگذار یا تصمیم‌گیری برای واردکردن داده‌ها به آن‌ها نگاه‌کند.
زمان شناخت	زمان میانگینی (برحسب ثانیه) که به طول می‌انجامد تا کاربری کنش مناسب برای وظیفه ای معین را برگزیند.
کار تایپی	تعداد میانگین ضربات صفحه‌کلید برای هر عملکرد.
کار با ماوس	تعداد میانگین کلیک‌های ماوس برای هر عملکرد.
پیچیدگی انتخاب‌ها	تعداد میانگین پیوندهایی که به‌ازای هر صفحه می‌توان انتخاب کرد.
زمان دریافت محتویات	تعداد میانگین واژه‌های متن به‌ازای هر صفحه وب.
بار حافظه	تعداد میانگین آیتم‌های داده‌ای متمایزی که کاربر باید به‌خاطر بسپارد تا به هدفی خاص دست پیدا کند.

معیارهای زیبایی‌شناختی (طراحی گرافیکی). طراحی گرافیکی مبتنی بر قضاوت‌های کیفی است و عموماً قابلیت اندازه‌گیری و نسبت‌دادن معیار به آن وجود ندارد. به هر حال، ایوری و همکاران وی [Ivo01] مجموعه‌ای از موازین را پیشنهاد می‌کنند که ممکن است در ارزیابی تأثیر طراحی زیبایی‌شناسی مفید واقع شوند:

معیار پیشنهادی	شرح
شمارش کلمات	تعداد کل کلماتی که در صفحه ظاهر می‌شوند.
درصد متن بدنه	درصد کلماتی که بدنه را تشکیل می‌دهند در مقایسه با کلمات نمایش (یعنی تیترها و عنوان‌ها)
درصد متن بدنه‌ی تأکیدشده	بخشی از متن بدنه که بر آن تأکید می‌شود (با حروف ضخیم یا ایتالیک)
شمارش‌تعیین موقعیت متون	تغییرات در موقعیت متون از حالت چپ‌چین.
شمارش خوشه‌های متون	نواحی از متن که با رنگ، حاشیه‌بندی، خطوط یا فهرست برجسته‌نمایی می‌شوند.
شمارش کلمات	تعداد کل کلماتی که در صفحه ظاهر می‌شوند.
درصد متن بدنه	درصد کلماتی که بدنه را تشکیل می‌دهند در مقایسه با کلمات نمایش (یعنی تیترها و عنوان‌ها)

معیار پیشنهادی	شرح
شمارش پیوندها	تعداد پیوندهای موجود در صفحه
اندازه صفحه	تعداد کل بایت‌ها به‌ازای صفحه و نیز عناصر، گرافیک‌ها و برگه‌های سبک نگارش
درصد گرافیک‌ها	درصدی از تعداد کل بایت‌های صفحه که به گرافیک‌ها تعلق دارند
شمارش گرافیک‌ها	تعداد گرافیک‌های موجود در صفحه
شمارش رنگ‌ها	تعداد رنگ‌های به‌کاررفته
شمارش فونت‌ها	تعداد کل فونت‌های به‌کاررفته (نوع + اندازه + ضخیم + ایتالیک)

معیارهای محتوا. در معیارهای این گروه، پیچیدگی محتوا و خوشه‌های اشیای محتوایی که در قالب صفحات، سازمان‌دهی شده‌اند، کانون توجه قرار می‌گیرد [Men01].

معیار پیشنهادی	شرح
انتظار صفحه	زمان میانگین لازم برای دانلود یک صفحه در سرعت‌های متفاوت.
پیچیدگی صفحه	تعداد میانگین انواع رسانه‌های به‌کاررفته در یک صفحه به غیر از متون.
پیچیدگی گرافیکی	تعداد میانگین رسانه‌های گرافیکی به‌ازای هر صفحه.
پیچیدگی صوتی	تعداد میانگین رسانه‌های صوتی به‌ازای هر صفحه.
پیچیدگی ویدیویی	تعداد میانگین رسانه‌های ویدیویی به‌ازای هر صفحه.
پیچیدگی پویانمایی	تعداد میانگین رسانه‌های پویانمایی به‌ازای هر صفحه.

معیارهای گشت‌وگذار. معیارهای این گروه به پیچیدگی جریان گشت‌وگذار می‌پردازند [Men01]. به‌طور کلی، این‌ها تنها برای وب ایستا قابل استفاده‌اند و شامل پیوندها و صفحاتی نمی‌شوند که به‌صورت پویا تولید می‌شوند.

معیار پیشنهادی	شرح
پیچیدگی پیوند صفحات	تعداد پیوندها به‌ازای هر صفحه.
اتصال (connectivity)	تعداد کل پیوندهای درونی؛ به‌غیر از پیوندهایی که به‌طریق پویا ایجاد می‌شوند.
دانسیته اتصال	اتصال تقسیم بر شمارش صفحات.

با به‌کارگیری زیرمجموعه‌ای از معیارهای پیشنهاد شده، ممکن است بتوان روابطی تجربی به‌دست آورد که به تیم توسعه‌ی برنامه تحت وب کمک کنند تا کیفیت را ارزیابی کنند و تلاش‌های مبتنی بر برآوردهای پیچیدگی را پیش‌بینی کند. در این زمینه هنوز خیلی کارها باقی مانده است که باید انجام شود.

ابزارهای نرم‌افزار

معیارهای فنی

هدف: کمک به مهندسان وب در تهیه‌ی معیارهای یا معنی برای برنامه‌های تحت وب که دیدی از کیفیت کلی یک برنامه‌ی کاربردی به‌دست می‌دهد.

مکانیک: مکانیک این ابزارها متغیر است.

ابزارهای نمونه

Netmechanic Tools که توسط Netmechanic (www.netmechanic.com) توسعه یافته است، مجموعه‌ای از ابزارهاست که به بهبود کارایی وب‌سایت با تأکید ورزیدن بر مسائل خاص پیاده‌سازی کمک می‌کند.

Nist Web Metrics Testbed که توسط The National Institute Of Standards and Technology (zing.ncsl.nist.gov/web Tools/) توسعه یافته است، شامل مجموعه ابزارهای مفید زیر می‌شود که برای دانلودکردن در دسترس هستند:

Web Static Analyzer Tool (Web SAT)- صفحه‌ی وب HTML را در مقابل دستورالعمل‌های متداول مربوط به قابلیت استفاده چک می‌کند.

Web Category Analysis Tool (Web CAT)- به مهندسی که مسؤولیت قابلیت استفاده را بر عهده دارد، این امکان را می‌دهد تا تحلیل گروهی وب را بناسازی و اجرا کند.

Web Variable Instrumenter Program (Web VIP)- وب‌سایت را به این امکان مجهز می‌کند تا از تعامل کاربر، ثبت وقایع کند.

Framework for Logging Usability Data (FLUD)- یک تجزیه‌گر و قالب دهنده‌ی فایل برای نمایش ثبت وقایع تعامل کاربر پیاده‌سازی می‌کند.

VisVIP Tool- یک تجسم سه بعدی از مسیرهای گشت‌وگذار در وب‌سایت ایجاد می‌کند.

Tree Dec- کمک‌هایی برای گشت‌وگذار در صفحات وب‌سایت اضافه می‌کند.

۲۳-۵ معیارهایی برای کد منبع (Source Code)

نظریه‌ی هالستد درخصوص «علم نرم‌افزار» [Hal77] نخستین «قوانین» تحلیلی را برای نرم‌افزارهای کامپیوتری پیش می‌کشد[1]. هالستد با استفاده از مجموعه‌ای از موازین اولیه، قوانین کمّی را به توسعه نرم‌افزار نسبت می‌دهد. این موازین اولیه پس از تولید کد به‌دست می‌آیند یا اینکه پس از کامل شدن طراحی برآورد می‌گردند:

n_1 = تعداد عملگرهای متمایز که در برنامه ظاهر می‌شوند.

n_2 = تعداد عملوندهای متمایز که در برنامه ظاهر می‌شوند.

N_1 = تعداد کل فراوانی عملگرها.

N_2 = تعداد کل فراوانی عملوندها.

«مغز انسان از یک مجموعه قواعد اکید پیروی می‌کند [برای توسعه الگوریتم‌ها] تا این که این کار را خود آگاهانه انجام دهد.»

موریس هالستد

[1] لازم به ذکر است که «قوانین» هالستد باعث در گرفتن بحث و جدل‌های اساسی بوده‌اند و بسیاری بر این باورند که نظریه‌ای که زیربنای این قوانین را تشکیل می‌دهد، نقص دارد. به هرحال، این قوانین برای برخی زبان‌های برنامه‌نویسی به‌طور تجربی وارسی شده‌اند [Fei89].

هالستد از موازین اولیه برای توسعه عبارت‌هایی جهت طول کلی برنامه؛ حجم کمینه‌ی بالقوه برای یک الگوریتم؛ حجم واقعی (تعداد بیت‌های لازم برای مشخص کردن یک برنامه)؛ سطح برنامه (میزانی از پیچیدگی برنامه)؛ سطح زبان (ثابتی که به زبان مفروض بستگی دارد) و ویژگی‌های دیگری از قبیل میزان کار توسعه، زمان توسعه و حتی تعداد خطاهای پیش‌بینی شده در نرم‌افزار استفاده می‌کند.

هالستد نشان می‌دهد که طول N را می‌توان برآورد کرد:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

و حجم برنامه را تعریف کرد:

$$V = N \log_2 (n_1 + n_2)$$

لازم به ذکر است که V به زبان برنامه‌نویسی بستگی دارد و حجم اطلاعات لازم برای مشخص کردن یک برنامه (برحسب بیت) را نشان می‌دهد.

به لحاظ نظری، باید یک حجم کمینه برای هر الگوریتم وجود داشته باشد. هالستد نسبت حجمی L را به‌صورت نسبت حجم فشرده‌ترین شکل یک برنامه به حجم برنامه واقعی تعریف می‌کند. درواقع، L باید همواره کوچکتر از واحد باشد. برحسب موازین اولیه، نسبت حجمی را چنین می‌توان بیان نمود:

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

کارهای هالستد جای آزمون زیاد دارد و پژوهش‌های فراوانی روی علم نرم‌افزار انجام شده است. بحث در این خصوص خارج از دید این کتاب است، ولی می‌توان گفت که بین نتایج تحلیلی و تجربی به‌دست‌آمده توافق و همخوانی خوبی حاصل شده است. برای اطلاعات بیشتر می‌توانید به [Zus90] و [Fen91] رجوع کنید.

اندزر

عملگرها شامل همه‌ی جریان ساخت‌های کنترلی، شرط‌ها و عملیات‌های ریاضی می‌شوند. عمل‌وندها شامل همه‌ی متغیرها و ثابت‌های برنامه می‌شوند.

۲۳-۶ معیارهایی برای آزمون

گرچه مطالب فراوانی درخصوص معیارهای آزمون نرم‌افزار نوشته شده است (مثل [Het93])، اکثر معیارهای پیشنهادی بر فرایند آزمون تأکید دارند، نه بر ویژگی‌های فنی خود آزمون‌ها. به‌طور کلی، آزمون‌گران باید متکی بر معیارهای تحلیل، طراحی و کدنویسی باشند تا راهنمای آنها در طراحی و اجرای موارد آزمون شوند.

معیارهای طراحی معماری نیز اطلاعاتی درباره سهولت یا دشواری آزمون انسجام (بخش ۳-۲۳) و نیاز به آزمون تخصصی نرم‌افزار (مثلاً stub‌ها و راه‌اندازها) به‌دست می‌دهند. پیچیدگی سیکلوماتیک (معیاری در طراحی سطح مؤلفه‌ها) اساس آزمون مسیرهای پایه را تشکیل می‌دهد و می‌توان آن را برای پیمانه‌های هدف، به‌عنوان کاندیدایی جهت آزمون گسترده‌ی واحدها به‌کار برد (فصل ۱۸). پیمانه‌هایی با پیچیدگی سیکلوماتیک بالا، احتمالاً بیشتر از پیمانه‌هایی با پیچیدگی سیکلوماتیک پایین در معرض خطا هستند. به همین دلیل، آزمون‌گر پیش از الحاق پیمانه به سیستم، باید انرژی زیادی برای کشف خطاها در آن پیمانه صرف کند.

نکته‌ی کلیدی

معیارهای آزمون به دو گروه عمده تقسیم‌بندی می‌شوند: (۱) معیارهایی که سعی در پیش‌بینی محتمل‌بودن تعداد آزمون‌های مورد نیاز در سطوح متفاوت آزمون دارند و (۲) معیارهایی که پوشش‌دهی به مولفه‌ای مفروض را کانون توجه قرار می‌دهند.

۲۳-۶-۱ کاربرد معیارهای هالستد در آزمون

مقدار کار لازم برای آزمون را می‌توان با استفاده از معیارهای به‌دست آمده از موازین هالستد تعیین کرد

(بخش ۵-۲۳). با استفاده از تعاریف حجم برنامه، V، سطح برنامه، PL، کار عملی نرم‌افزار، e، را می‌توان به‌صورت زیر محاسبه کرد:

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)} \qquad (۲-۲۳الف)$$

$$e = \frac{V}{PL} \qquad (۱-۲۳ب)$$

درصد کل کاری که باید به پیمانه k تخصیص داده شود، با استفاده از رابطه زیر برآورد می‌شود:

$$\text{درصد کار آزمون } (k) = \frac{e(k)}{\Sigma e(i)} \qquad (۳-۲۳ب)$$

که $e(k)$ برای پیمانه k با استفاده از معادلات (۲-۲۳) محاسبه می‌شود و مجموعی که در مخرج معادله (۳-۲۳) ظاهر می‌شود، برابر با مجموع کار عملی در میان کلیه پیمانه‌های سیستم است.

۲۳-۶-۲ معیارهای مربوط به آزمون‌های شیءگرا

معیارهای طراحی ذکرشده در بخش‌های ۳-۲۳، شاخصی از کیفیت طراحی ارائه می‌دهند. این معیارها همچنین شاخصی کلی از مقدار کار لازم برای آزمون سیستم شیءگرا ارائه می‌دهند. بایندر [Bin94] انواع معیارها را پیشنهاد می‌کند که بر «آزمون‌پذیری» سیستم شیءگرا تأثیری مستقیم دارند. در این معیارها جنبه هایی از بسته‌بندی و وراثت مد نظر قرار می‌گیرند.

فقدان انسجام در متدها (LCOM)[1]. هر چه مقدار LCOM بزرگتر باشد، حالت‌های بیشتری را باید آزمود تا اطمینان حاصل شود که متدها اثرات جانبی تولید نمی‌کنند.

درصد عمومی و محافظت شده (PAP). صفات عمومی از کلاس‌های دیگر به ارث برده می‌شوند و لذا در معرض دید آن کلاس‌ها قرار دارند. صفات محافظت شده، شکل تخصص‌اند و خاص یک زیرکلاس معین هستند. این معیار نشان‌گر درصدی از صفات کلاس است که عمومی هستند. مقادیر بالای PAP باعث افزایش احتمال بروز اثرات جانبی در میان کلاس‌ها می‌شود[2]. باید آزمون‌هایی طراحی شود که به حصول اطمینان از کشف این اثرات جانبی منجر شوند.

دستیابی عمومی به اعضای داده‌ای (PAD). این معیار، تعداد کلاس‌ها یا متدهایی را نشان می‌دهد که می‌توانند به صفات کلاس‌های دیگر دستیابی داشته باشند. این موضوع، عدول از بسته‌بندی است. مقادیر بالای PAD منجر به افزایش اثرات جانبی در میان کلاس‌ها می‌شود. باید آزمون‌هایی طراحی شود که به حصول اطمینان از کشف این اثرات جانبی کمک کند.

تعداد کلاس‌های ریشه (NOR). این معیار، شمارشی از سلسله مراتب‌های متمایز است که در مدل طراحی توصیف می‌شوند. برای هر کلاس ریشه و سلسله مراتب کلاس‌ها، باید مجموعه‌ای از آزمون‌ها توسعه داده شود. با افزایش NOR، کار آزمون فزونی می‌یابد.

اندزر

آزمون شیءگرا می‌تواند بسیار پیچیده باشد. معیارها می‌توانند شما را در هدف‌قراردادن منابع آزمون در نخ‌ها، سناریوها و پکج‌هایی از کلاس‌ها که بر اساس خصوصیات اندازه‌گیری شده «مظنون» شمرده می‌شوند، یاری دهند.

---

[1] برای شرح LCOM بخش ۳-۳-۲۳ را ببینید.

[2] برخی طراحی را ارتقا می‌دهند بدون اینکه صفات عمومی یا خصوصی باشند، یعنی PAP = 0. این بدان معناست که همه‌ی صفات باید در کلاس‌های دیگر از طریق متدها ارزیابی شوند.

توان ورودی (FIN) . توان ورودی هنگامی که در حیطه شیءگرا به‌کار گرفته شود، شاخصی از وراثت چندگانه است. FAN > 1 نشان‌گر آن‌است‌که، کلاس، صفات و عملیات خود را از بیش از یک کلاس ریشه به ارث می‌برد. تا حد امکان باید از FAN < 1 پرهیز کرد.

تعداد فرزندان (NOC) و عمق درخت وراثت (DIT)[1]. همان‌طور که در فصل ۱۹ بحث شد، متدهای کلاس پایه را باید برای هر زیرکلاس دوباره آزمود.

ابزارهای نرم‌افزاری

معیارهای محصول

هدف: برای کمک به مهندسان نرم‌افزار در توسعه‌ی معیارهای با معنی که محصولات کاری تولید شده طی مراحل مدل‌سازی تحلیل و طراحی، تولید کد منبع و آزمون را ارزیابی کنند.

مکانیک: ابزارهای موجود در این گروه شامل آرایه گسترده‌ای از معیارها و به‌صورت برنامه‌های کاربردی مستقل (یا به‌طور متداول‌تر) به‌صورت یک قابلیت عملیاتی در داخل ابزارهای مربوط به تحلیل و طراحی، کدنویسی یا آزمون پیاده‌سازی می‌شوند. در اکثر موارد، ابزار معیاری، نمایشی از نرم‌افزار (مثلاً یک مدل UML یا کد منبع) را تحلیل می‌کند و یک یا چند معیار را توسعه می‌دهد.

ابزارهای نمونه:

Krakatua Metrics که توسط Power Software (**www.powersoftware.com/products**) توسعه یافته است، معیارهای پیچیدگی، هالستد و معیارهای مرتبط با آن‌ها را برای C/C++ و جاوا محاسبه می‌کند.

Metrics4C که توسط Software Engineering 1+ توسعه یافته است (**www.plus-one.com/Metrics4C_fact_sheet.html**) ، انواع معیارهای معماری، طراحی، و کدمحور و نیز معیارهای پروژه-محور را محاسبه می‌کند.

Rational Rose که توسط IBM توزیع می‌شود، ابزاری جامع برای مدل‌سازی UML است که شامل چند ویژگی تحلیل معیار نیز می‌شود.
(**www-304.ibm.com/jct03001c/software/awdtools/developer/rose**)

RSM که توسط شرکت M-Squared Technologies توسعه یافته است، (**www.m-squaredtechnologies.com/m2rsm/index.html**) انواع معیارهای کدمحور را برای C/C++ و جاوا محاسبه می‌کند.

۲۳-۷ معیارهایی برای نگهداری

کلیه معماری‌های نرم‌افزاری معرفی شده در این فصل را می‌توان برای توسعه دادن نرم‌افزارهای جدید و نگهداری نرم‌افزارهای موجود به‌کار برد. ولی، معیارهایی نیز برای فعالیت‌های نگهداری، طراحی شده‌اند.

[1] برای شرح NOC و DIT، بخش ۳-۳-۲۳ را ببینید.

در استاندارد IEEE Std. 982.1-19889 [IEE93] یک *شاخص بلوغ نرم‌افزار* (SMI) پیشنهاد شده است که (براساس تغییراتی که برای هر بار ارائه نسخه جدیدی از محصول رخ می‌دهد) نشان‌گر پایداری محصول نرم‌افزاری است. اطلاعات زیر تعیین می‌شود:

M_T = تعداد پیمانه‌ها در نسخه اصلی

F_c = تعداد پیمانه‌هایی که در نسخه اصلی تغییر یافته‌اند

F_a = تعداد پیمانه‌هایی که در نسخه فعلی اضافه شده‌اند

F_d = تعداد پیمانه‌هایی که در نسخه‌های قبلی بوده‌اند و در نسخه فعلی حذف شده‌اند

شاخص بلوغ نرم‌افزار به شیوه زیر محاسبه می‌شود:

$$\text{SMI} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

با نزدیک شدن SMI به ۱٫۰، محصول پایدارتر می‌شود. SMI را می‌توان به‌عنوان معیاری برای برنامه‌ریزی فعالیت‌های نگهداری نرم‌افزار به‌کار برد. زمان میانگین برای تولید نسخه‌ای از یک محصول نرم‌افزار را می‌توان با SMI مرتبط ساخت و مدل‌هایی تجربی را برای کار لازم جهت نگهداری، توسعه داد.

۲۳-۸ خلاصه

معیارهای نرم‌افزار یک راه کمّی برای ارزیابی کیفیت صفات داخلی محصول به‌دست می‌دهند و درنتیجه مهندس نرم‌افزار را قادر می‌سازد تا کیفیت را پیش از تولید محصول ارزیابی کند. معیارها دید لازم برای ایجاد مدل‌های تحلیل و طراحی اثربخش، کدهای منسجم و آزمون‌های کامل را فراهم می‌آورند.

برای آنکه معیار نرم‌افزاری در جهان واقعی مفید باشد، باید ساده و قابل محاسبه، مستدل، سازگار و عمیق باشد. باید مستقل از نوع زبان برنامه‌نویسی باشد و بازخورد مؤثری در اختیار مهندس نرم‌افزار قرار دهد.

معیارهای مربوط به مدل خواسته‌ها، عملکرد، داده‌ها و رفتار- سه مؤلفه‌ی مدل- را کانون توجه قرار می‌دهند. در معیارهای مربوط به طراحی معماری، جنبه‌های ساختاری مدل طراحی مد نظر قرار می‌گیرد. معیارهای طراحی در سطح مؤلفه‌ها، با فراهم آوردن موازین مستقیم برای یکپارچگی، اتصال و پیچیدگی، شاخصی از کیفیت پیمانه‌ها فراهم می‌آورند. معیارهای طراحی واسط کاربر، شاخصی از سهولت استفاده از یک GUI فراهم می‌آورند. در معیارهای برنامه‌های تحت وب، جنبه‌هایی از واسط کاربر و نیز زیبایی‌شناسی برنامه‌های تحت وب، محتویات و گشت‌وگذار در نظر گرفته می‌شوند.

معیارهای مربوط به سیستم‌های شیءگرا، اندازه‌گیری‌هایی را کانون توجه قرار می‌دهند که می‌توان آن‌ها در خصوصیات کلاس‌ها و طراحی به‌کار برد؛ این خصوصیات عبارتند از محلی‌سازی، پنهان‌سازی اطلاعات، وراثت و تکنیک انتزاع اشیا که کلاس را منحصر به فرد می‌سازند. در مجموعه معیارهای CK، شش معیار برای نرم‌افزارهای کلاس گرا تعریف می‌شوند که کلاس و سلسله مراتب کلاس‌ها را کانون توجه قرار می‌دهند. در این مجموعه معیارها همچنین معیارهایی برای ارزیابی همکاری‌های میان کلاس‌ها و یکپارچگی متدهای درون هر کلاس فراهم می‌آورند. مجموعه معیارهای CK در یک سطح کلاس گرا را می‌توان با معیارهای پیشنهاد شده توسط لورنتز و کید و مجموعه معیارهای MOOD تکمیل کرد.

هالستد، مجموعه‌ای فزاینده از معیارها را در سطح کد منبع فراهم می‌آورد. علم نرم‌افزار با به‌کارگیری عملکرد و عمل‌وندهای موجود در کدها، انواع معیارهای قابل استفاده در ارزیابی کیفیت برنامه را در اختیار می‌گذارد.

چند معیار محصولی معدود برای استفاده مستقیم در آزمون و نگهداری نرم‌افزار پیشنهاد شده است. به هر حال، معیارهای محصولی دیگری را نیز می‌توان در هدایت و راهبری فرایند آزمون و به‌عنوان سازوکاری برای ارزیابی قابلیت نگهداری یک برنامه کامپیوتری به‌کار برد. انواع معیارهای شیءگرا برای ارزیابی قابلیت آزمون یک سیستم شیءگرا پیشنهاد شده است.

مسائل و نکاتی برای تعمق

۲۳-۱ نظریه اندازه‌گیری یک مبحث پیشرفته است که تأثیری قوی بر معیارهای نرم‌افزاری داشته است. با استفاده از [Zus97]، [Fen91]، [Zus90] یا منابع موجود در وب، مقاله مختصری بنویسید و باورهای اصلی نظریه اندازه‌گیری را در آن مطرح کنید. پروژه انفرادی: این موضوع را سر کلاس ارائه دهید.

۲۳-۲ چرا نمی‌توان یک معیار منفرد و جهان‌شمول برای پیچیدگی برنامه یا کیفیت برنامه توسعه داد؟

۲۳-۳ سیستمی ۱۲ ورودی خارجی، ۲۴ خروجی خارجی، ۳۰ درخواست خارجی متفاوت دارد، ۴ فایل منطقی داخلی را مدیریت می‌کند و با ۶ سیستم قدیمی متفاوت ایجاد واسط می‌کند (شش EIF). همه‌ی این داده‌ها، پیچیدگی میانگین دارند و کل سیستم نسبتاً کوچک است. FP را برای این سیستم محاسبه کنید.

۲۳-۴ نرم‌افزار مربوط به سیستم X دارای ۲۴ خواسته‌ی علمیاتی و ۱۴ خواسته‌ی غیر عملیاتی است. میزان مشخص‌بودن و کامل‌بودن را تعیین کنید.

۲۳-۵ یک سیستم اطلاعاتی بزرگ دارای ۱۱۴۰ پیمانه است. ۹۶ پیمانه، عملیات کنترلی و هماهنگ‌سازی را انجام می‌دهند و ۴۹۰ پیمانه هستند که عملکرد آنها به پردازش قبلی بستگی دارد. سیستم حدوداً ۲۲۰ شیء داده‌ای را پردازش می‌کند که هر یک به‌طور میانگین ۳ صفت دارد. ۱۴۰ عنصر بانک اطلاعاتی و ۹۰ مؤلفه متفاوت در بانک اطلاعاتی وجود دارد. ۶۰۰ پیمانه دارای نقاط ورود و خروج یگانه‌اند. DSQI را برای این سیستم محاسبه کنید.

۲۳-۶ یک کلاس به نام X، ۱۲ عملیات دارد. پیچیدگی سیکلوماتیک برای کلیه عملیات سیستم شیءگرا محاسبه شده است و مقدار میانگین پیچیدگی پیمانه‌ها برابر ۴ است. برای کلاس X، پیچیدگی، برای عملیات ۱ تا ۱۲ به‌ترتیب عبارت است از: ۵، ۴، ۳، ۳، ۶، ۸، ۲، ۲، ۵، ۵، ۴، ۴. WMC را محاسبه کنید.

۲۳-۷ یک ابزار نرم‌افزاری توسعه دهید که پیچیدگی سیکلوماتیک را برای یک پیمانه زبان برنامه‌نویسی محاسبه کند. انتخاب زبان با خودتان است.

۲۳-۸ ابزار نرم‌افزاری کوچکی توسعه دهید که تحلیل هالستد را برای کد منبعی به انتخاب خودتان انجام دهد.

۲۳-۹ یک سیستم قدیمی ۹۴۰ پیمانه دارد. آخرین نسخه مستلزم جایگزینی ۹۰ پیمانه است. به‌علاوه ۴۰ پیمانه جدید اضافه و ۱۲ پیمانه حذف شده‌اند. شاخص بلوغ نرم‌افزار را برای این سیستم محاسبه کنید.

پیوست ۱

آشنایی با UML[1]

زبان مدل‌سازی یکپارچه (UML) «زبانی استاندارد برای نوشتن نقشه‌های نرم‌افزار است. از UML می‌توان برای تجسم بخشیدن، مشخص‌کردن، ساخت و مستندسازی محصولات یک سیستم نرم‌افزاری استفاده کرد.» [Boo05] به بیان دیگر، درست همان طور که معماران ساختمانی، نقشه‌هایی تهیه می‌کنند که شرکت‌های ساخت‌وساز از آنها استفاده می‌کنند، معماران نرم‌افزار نیز نمودارهای UML را برای کمک به سازندگان نرم‌افزار در ساخت نرم‌افزار تهیه می‌کنند. اگر واژگان UML (یعنی عناصر تصویری نمودار و معانی آنها) را فرابگیرید، بسیار آسان‌تر می‌توانید سیستمی را بفهمید و مشخص کنید و طراحی آن سیستم را برای دیگران توضیح دهید. گرادی بوچ، جیم رومباف و ایوار جیکابسون، UML را در میانه‌ی دهه ۱۹۹۰ با گرفتن بازخوردهای فراوان از جامعه‌ی نرم‌افزاری توسعه دادند. UML، چند نمادگذاری مدل‌سازی رقیب را که در صنعت نرم‌افزار آن زمان مورد استفاده قرار می‌گرفتند، در هم ادغام نمود. در سال ۱۹۹۷ UML 1.0، تسلیم گروه مدیریت اشیا (یک کنسرسیوم غیرانتفاعی دخیل در حفظ و نگهداری مشخصات مورد استفاده در صنعت نرم‌افزار) شد. UML 1.0 بازبینی و به UML 1.1 ارتقا داده شد و سال بعد پذیرفته شد. استاندارد فعلی، UML 2.0 است اکنون یکی از استانداردهای ISO است. از آنجا که این استاندارد بسیار جدید است، بسیاری از مراجع قدیمی نظیر [Gam95] از نمادگذاری UML استفاده نمی‌کنند.

UML 1.0 سیزده نمودار متفاوت برای استفاده در مدل‌سازی نرم‌افزار فراهم می‌آورد. در این پیوست، تنها به بحث درباره نمودارهای کلاس‌ها، استقرار، use case، ترتیب (sequence)، ارتباطات، فعالیت و حالت خواهیم پرداخت. در این ویرایش از کتاب *مهندسی نرم‌افزار* از همین نمودارها استفاده شده است.

باید توجه داشته باشید که در نمودارهای UML، ویژگی‌های اختیاری فراوانی موجود است. زبان UML این گزینه‌ها را (که گاهی اسرارآمیز هستند) فراهم می‌آورد تا بتوانید همه‌ی جنبه‌های مهم یک سیستم را بیان کنید. در عین حال، این انعطاف را دارید که بخش‌هایی از نمودار را که به جنبه‌ی مدل‌سازی شده مربوط نمی‌شوند، بپوشانید به‌طوری که از ازدحام نمودار با جزئیات نامربوط جلوگیری شود. بنابراین، حذف یک ویژگی خاص به این معنی نیست که آن ویژگی وجود ندارد. بلکه ممکن است به این معنی باشد که آن ویژگی پوشانده شده است. در این پیوست، همه‌ی ویژگی‌های UML به‌طور کامل ارائه نشده است. در عوض، گزینه‌های استاندارد کانون توجه قرار گرفته‌اند به ویژه آن دسته از گزینه‌هایی که در این کتاب استفاده شده‌اند.

[1] این پیوست با کسب اجازه از دیل سرکین و از کتاب ایشان با عنوان *An Introduction to Object-Oriented Design Patterns in Java* آورده شده است.