مهندسی نرم افزار ۲

ویراست هفتم

تألیف:

راجر اس. پرسمن

ترجمه:

عین‌الله جعفرنژاد قمی، ابراهیم عامل محرابی

دانش نگار

1

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.

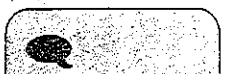
با چه سرعتی از نیویورک به لس‌آنجلس برسد، آیا سفر شامل تماشای مناظر و اطراق بین راهی هم می‌شود، او چقدر پول می‌تواند خرج کند، آیا قصد خاصی از سفر دارد، و چه وسایل نقلیه شخصی در اختیار خود دارد. با توجه به این نیروها، مسأله‌ی (سفر از نیویورک به لس آنجلس) را بهتر می‌توان تعریف کرد. برای مثال، بررسی (جمع 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.

نکته‌ی کلیدی
الگوی مولد، نه تنها مسأله، حیطه و نیروها را توصیف می‌کند بلکه راهکاری عملی برای این مسأله شرح می‌دهد.

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.

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.

آیا راهی برای گروه‌بندی انواع الگوها وجود دارد؟

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.

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 تم‌پر 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 الگویی بیابید که مناسب دامنه‌ی مسأله‌ی شما باشد، به دنبال تشابه 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

---

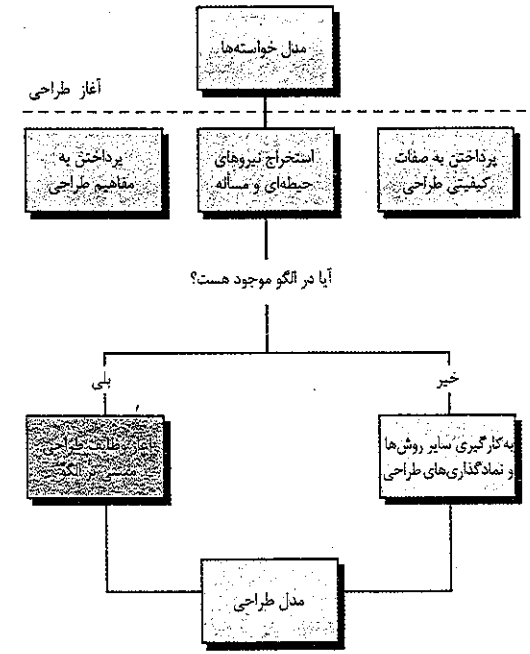
[1] کریستوفر الکساندر در آغاز برای معماری و نقشه کشی شهری، زبان‌های الگوها را پیشنهاد کرد. امروزه، زبان‌های الگوها برای هر چیزی از علوم اجتماعی گرفته تا فرایند مهندسی نرم‌افزار توسعه یافته‌اند.

[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.

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.

طراحی مبتنی بر الگو برای مسأله‌ای که باید حل کنم، جالب به نظر می‌رسد؛ از کجا شروع کنم؟

[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 (سطوح 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.

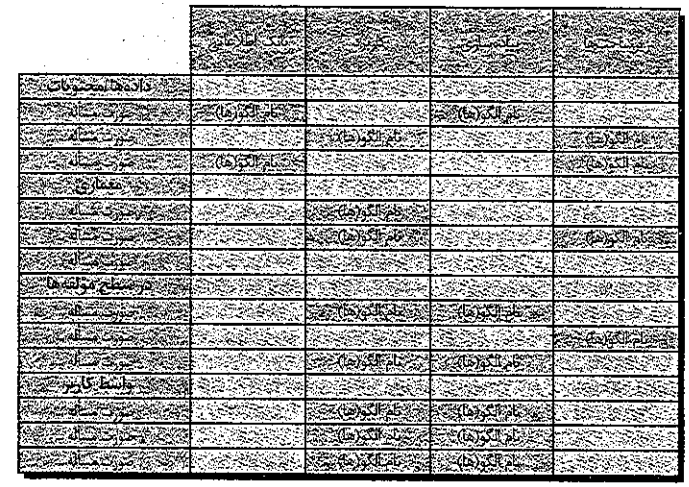
به‌علاوه، رویکرد مبتنی بر الگوها را باید همراه با سایر مفاهیم و تکنیک‌های طراحی به‌کار برد.

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.



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 حیطه 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 حیطه 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

[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.

نکته‌ی کلیدی

هر معماری نرم‌افزار ممکن است چند الگوی معماری داشته باشد که به مسائل گوناگون از قبیل همروندی، مانیتورکاری و توزیع مربوط شوند.

چند نمونه از دامنه‌های الگوی معماری را نام ببرید.

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 جس‌گرها 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 جس‌گرهای 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 می‌کند.

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.

۱۲-۵ 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 ویرایشگرها

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 یک عملیات پیچیده

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 سبد خرید

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.

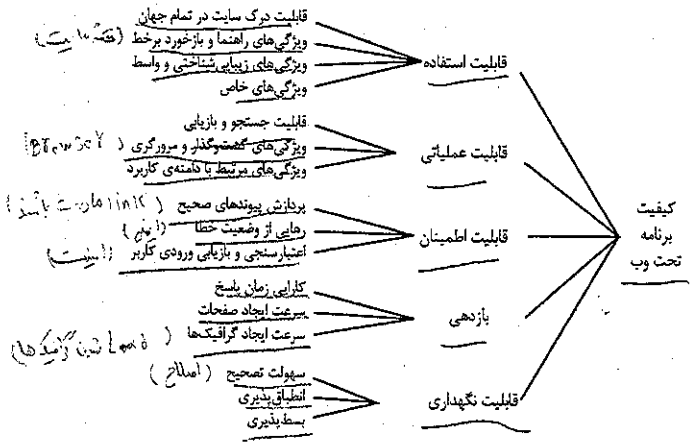
ولی کیفیت برنامه‌ی تحت وب را چگونه باید درک کرد؟ در برنامه‌ی تحت وب چه صفاتی باید وجود داشته باشد تا خوب بودن به چشم کاربران نهایی بیاید و در عین حال، خصوصیات فنی کیفیت را از خود نشان دهد که شما را در درازمدت قادر به تصحیح، تطبیق، به‌سازی و پشتیبانی از آن سازد؟

> «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:

۱) امنیت ۲) اوازهوبت ۳) سهولت‌سنجی ۴) قابلیت‌بررسی ۵) انعکاس‌پذیری

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:

- 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.

هنگام ارزیابی محتوا چه چیزهایی را باید مد نظر داشت؟

۱۳-۲ 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.

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

«صرف توانایی، به معنای لزوم نیست.»
جین کیسر

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.

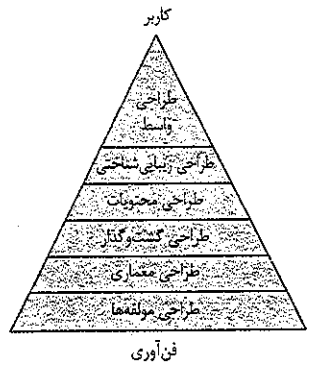
> «از نظر برخی، طراحی وب، شکل و شمایل بصری را کانون توجه قرار می‌دهد. از نظر برخی دیگر، طراحی وب به ساختاردهی اطلاعات و گشت‌وگذار در میان فضای مستندات مربوط می‌شود. سایرین حتی ممکن است طراحی وب را به فن‌آوری مربوط بدانند. در واقع، طراحی شامل همه‌ی این چیزها و حتی بیشتر از این می‌شود.»
>
> **تامس پاول**

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 دستور العمل‌های 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 طراقت 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 آنان زیباتر باشید 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.grophic-design.com**، **www.grantasticdesign.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.

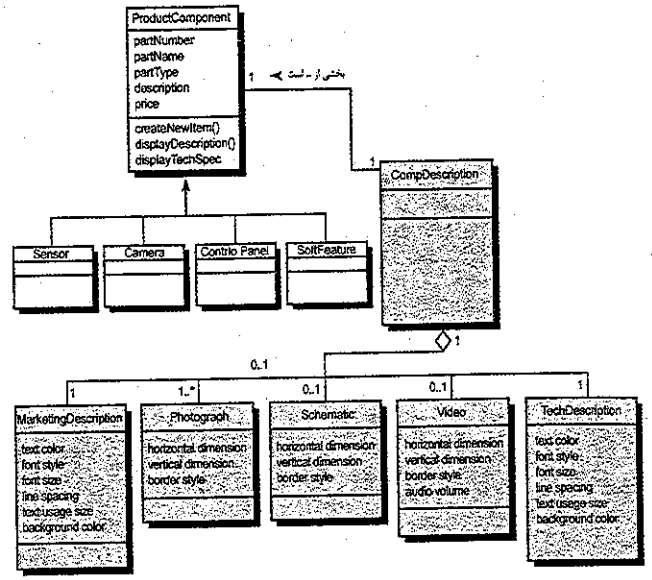
۱-۶-۱۳ اشیای محتوایی (Content objects)

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**

> «طراحان خوب می‌توانند از آشفتگی، نظم بیافرینند، آن‌ها می‌توانند به‌وضوح ایده‌ها را از طریق سازمان‌دهی و دستکاری واژه‌ها و تصاویر منتقل کنند.»
>
> **جفری وین**

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*، *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.

---

[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.

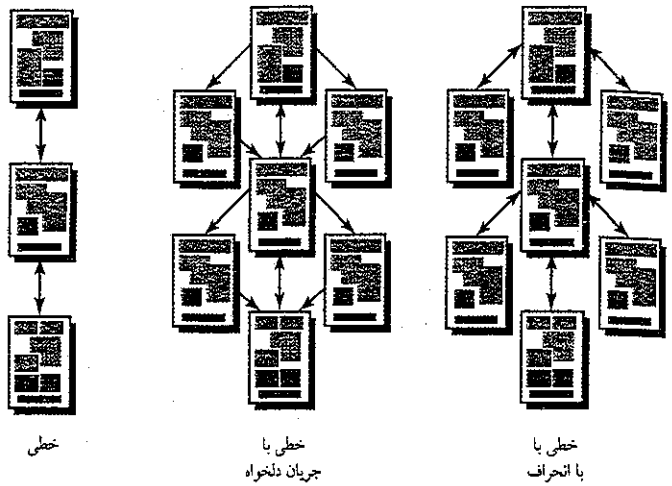
۱۳-۷-۱ معماری محتوا (Content Architecture)

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).

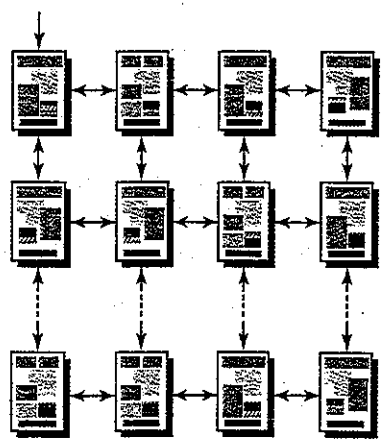
[1] عبارت *معماری اطلاعات* برای ساختارهایی به‌کار می‌رود که به سازمان‌دهی، نشان‌گذاری، گشت‌وگذار و جستجوی بهتر اشیای محتوایی منجر می‌شود.



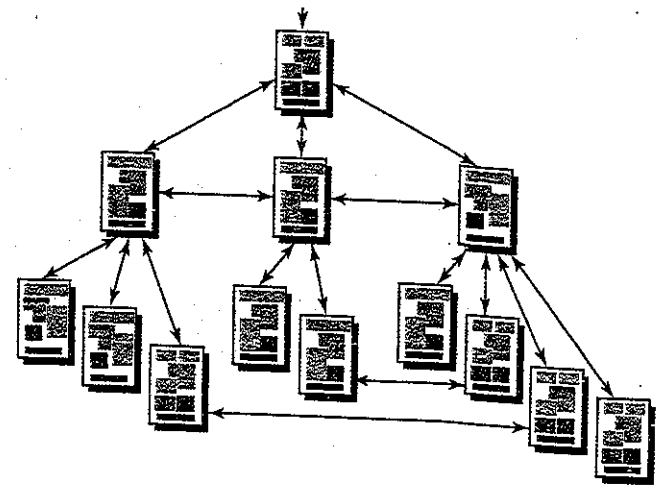
شکل ۴-۱۳ ساختارهای خطی.

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].

نکته‌ی کلیدی
معماری MVC واسط کاربر را از قابلیت‌های عملیاتی برنامه‌ی تحت وب و محتوای اطلاعاتی آن منفک می‌کند.

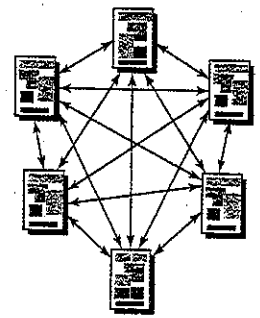


شکل ۵-۱۳ ساختار مشبک.



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 آبرمتنی 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

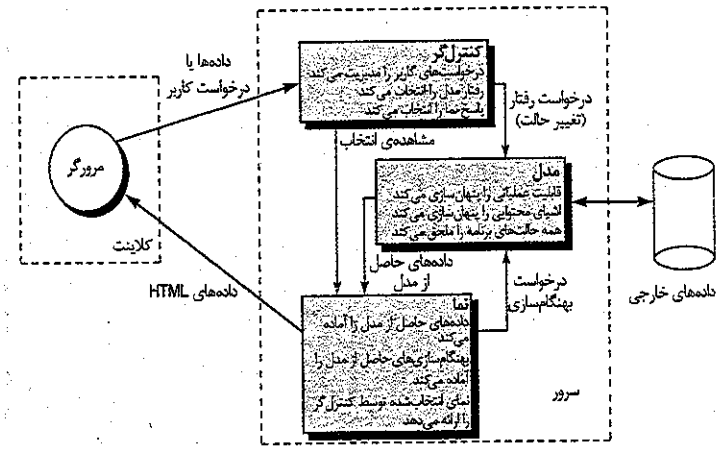
معماری برنامه‌های تحت وب، زیرساختاری را توصیف می‌کند که سیستم یا برنامه کاربردی مبتنی بر وب را قادر می‌سازد تا به اهداف تجاری خود دست پیدا کند. جاسیتو و همکاران [Jac02b] خصوصیات پایه‌ی این زیرساخت را به شیوه زیر توصیف می‌کند:

> 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.

۱۳-۸ 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.

۱۳-۸-۱ معناشناسی گشت‌وگذار (Navigation Semantics)

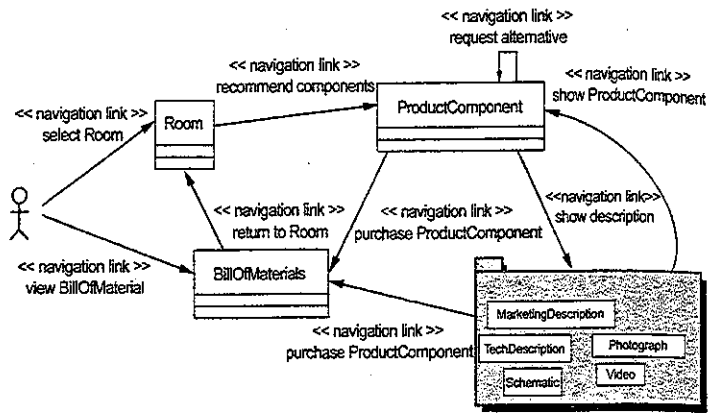
همانند بسیاری از کنش‌های طراحی برنامه‌های تحت وب، طراحی گشت‌وگذار با در نظر گرفتن سلسله مراتب کاربران و پرونده‌های کاربرد مرتبط (فصل ۵)، که برای هر گروه از کاربران (کنش گران) تهیه شده‌اند، آغاز می‌شود. هر کنش‌گر ممکن است از برنامه‌ی تحت وب با قدری تفاوت نسبت به دیگران استفاده کند و بنابراین، خواسته‌های 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 خرده‌نان‌هایی که پشت سرمان ریخته، پدیدار شوند؛ آن‌ها راه بازگشت به خانه را نشان می‌دهند.»
>
> **هانسل و گرتل**

تکته‌ی 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.

> «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**

[1] این پیوندها را گاهی پیوندهای معناشناختی گشت‌وگذار (NSL) نیز می‌نامند [Cac02].

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.

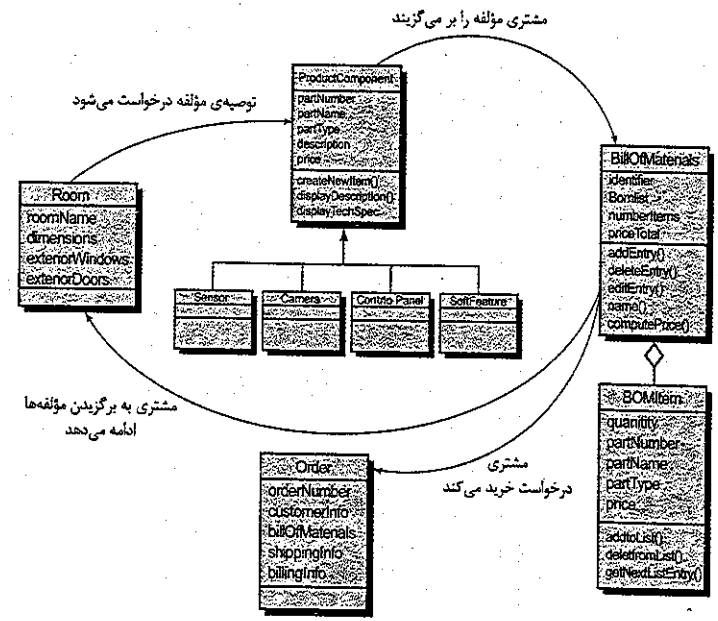
۱۳-۱۰-۱ طراحی مفهومی برای OOHDM

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:

> «مردم فراموش می‌کنند که چقدر سریع کاری را انجام داده‌اید- ولی همیشه به‌خاطر خواهند سپرد که چقدر خوب آن را انجام داده‌اید.»
> **هاوارد نیوتن**

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.

۱۴-۲ 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.

۱۴-۲-۱ 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؟

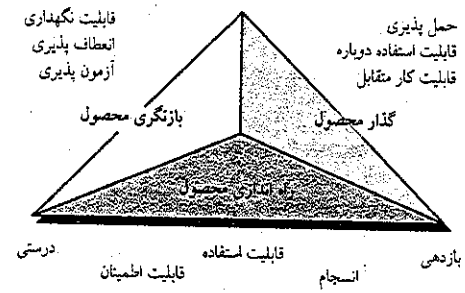
دوام. آیا نرم‌افزار را می‌توان نگهداری کرد (تغییر داد) یا تصحیح (اشکال‌زدایی) کرد بدون این‌که اثرات جانبی ایجاد شود؟ آیا تغییرات باعث می‌شود که میزان خطا یا قابلیت اطمینان با زمان کاهش یابد؟

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.

۲-۲-۱۴ 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 سرِ وقت 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.

۳-۲-۱۴ 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.

۴-۲-۱۴ 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.

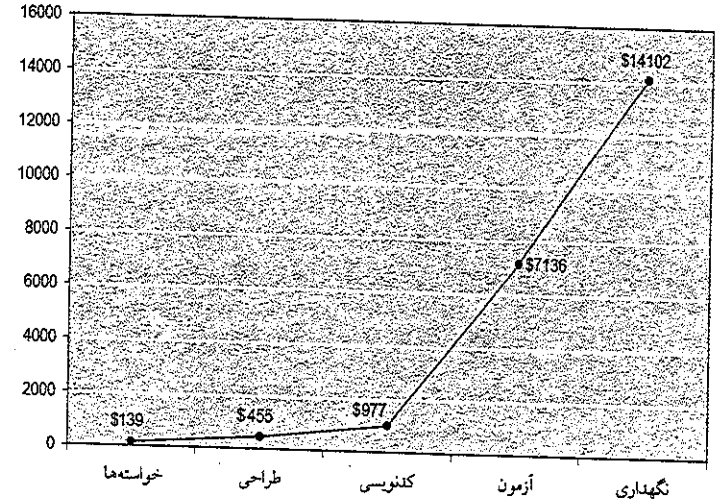
هزینه‌های شکست خارجی به نقایصی مربوط می‌شود که پس از رسیدن محصول به‌دست مشتری کشف می‌شوند. مثال‌هایی از هزینه‌های شکست خارجی عبارتند از جلب رضایت شاکی، مرجوع و جایگزین کردن محصول، کمک به پشتیبانی خطی و هزینه‌های کار مرتبط با ضمانت. بدنامی و ضرر و زیان نیز یک هزینه شکت خارجی دیگر است که تعیین کمّی آن 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.

۳-۳-۱۴ 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.

۴-۳-۱۴ 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.

۵-۳-۱۴ 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.

۶-۳-۱۴ 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.

۴-۱۴ 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.

۱-۴-۱۴ 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.

۸-۱۴ 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.

۹-۱۴ با در نظر گرفتن هر کدام از چهار جنبه‌ی هزینه کیفیت، تصور می‌کنید کدام یک بیشترین هزینه را در بر دارد و چرا؟

۱۰-۱۴ 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.

۱۱-۱۴ آیا *کیفیت* و *امنیت* یک مقوله‌اند؟ توضیح دهید.

۱۲-۱۴ 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

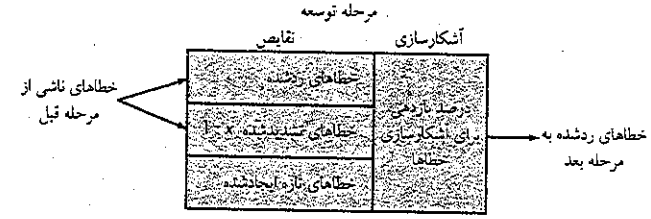
تا ۷۵٪ در کشف معایب طراحی مؤثر واقع می‌شوند [Jon86]. فرایند مرور با یافتن و حذف درصد بزرگی از این خطاها، هزینه‌ی مراحل بعدی را در فازهای توسعه و پشتیبانی، به‌طور چشمگیر کاهش می‌دهد.

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.

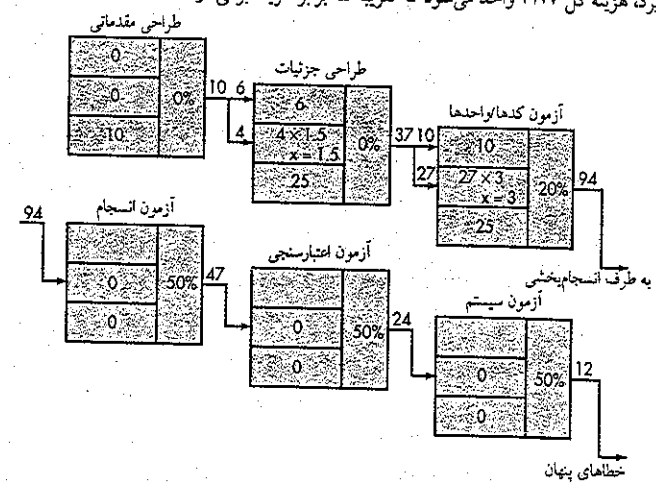
[1] اگر بهبود فرایند نرم افزار مد نظر باشد، مشکلی کیفیتی را که از یک فعالیت چارچوبی فرایند (مثلا مدل‌سازی) به فعالیت چارچوبی دیگر مثلاً ساخت) منتشر می شود نیز می‌توان «نقص» نامید (چون این مشکل می‌بایست قبل از «تحویل» یک محصول کاری به فعالیت دیگر یافته می‌شده است.)

> «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 شده باشند، با گذر زمان، تشخیص آن‌ها آسان، ولی معالجه‌ی آن‌ها دشوار خواهد شد.»
>
> **نیکولو ماکیاولی**



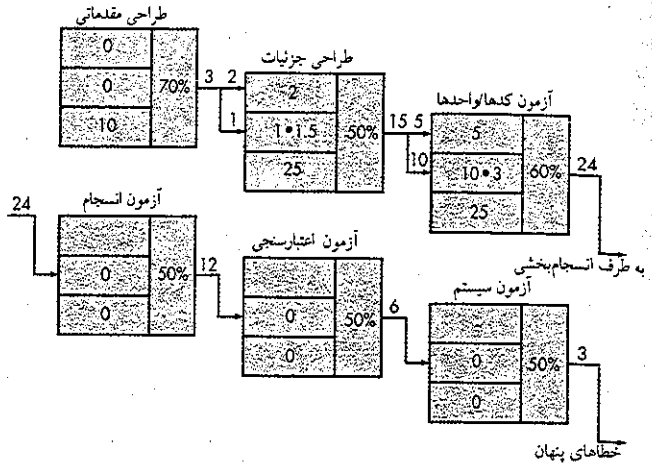
شکل ۱-۱۵ مدل تشدید نقایص.

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:

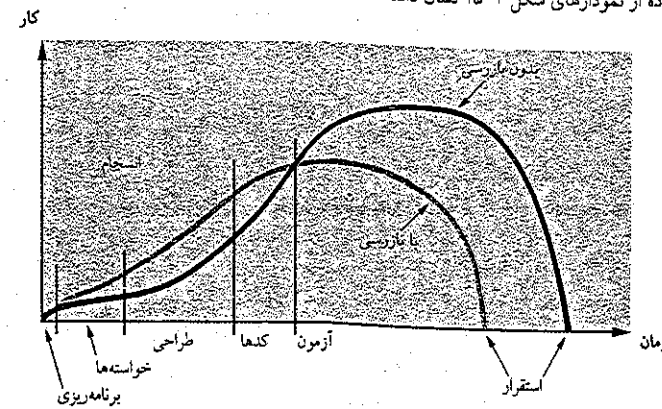
$$\text{تلاش صرفه‌جویی‌شده به‌ازای هر خطا} = E_{testing} - E_{reviews}$$

$$= 45 - 6 = 30 \text{ (نفر ساعت به‌ازای هر خطا)}$$

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.

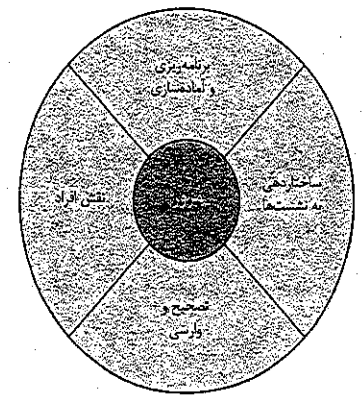


شکل ۴-۱۵ تلاش‌های صرف شده با مرور و بدون مرور.

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 زواند 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 عبارتنداز: (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.

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.

> «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

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.

> «تعداد اشتباهات شما بیش از حد زیاد بوده است.»
>
> یوگی برا

[1] اگر فصل ۱۴ را نخوانده‌اید، اکنون آن را بخوانید.

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.

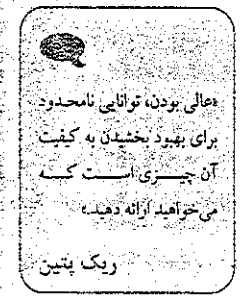
مدیریت منابع خرید. سه گروه از نرم‌افزارها از منابع خارجی تأمین نرم‌افزار خریداری می‌شوند- *پکیج‌های بسته‌بندی شده* (مثل Microsoft Office)، قطعات *نیمه‌آماده* [Hor03] که ساختار اسکلتی پایه را فراهم می‌آورند و می‌توان آن را مطابق با میل و سلیقه‌ی خریدار تکمیل کرد و نرم‌افزارهای *قراردادی* که از روی مشخصات 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 عبارتنداز 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 وارسیِ 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 وارسیِ 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. هوارتی

«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 منتهی 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 اجتناب‌ناپذیری 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.

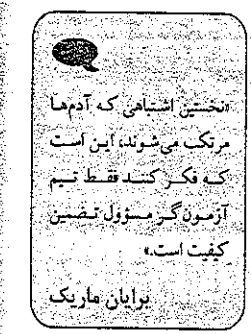
نقش *گروه آزمون مستقل* (ITG) برطرف کردن مشکلات ذاتی است که در واگذاری آزمون به شخص سازنده وجود دارد. آزمون مستقل، اختلاف سلیقه‌ها را برطرف می‌سازد. به هر حال، برای یافتن خطاها، به اعضای این گروه مستقل پول پرداخت می‌شود.

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.

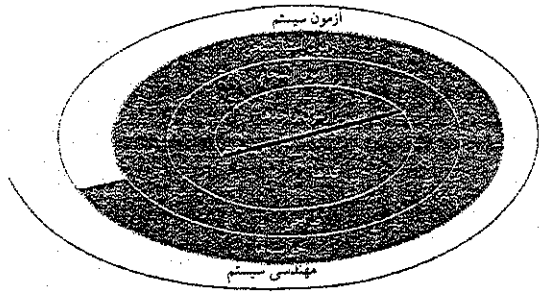


«نخستین اشتباهی که آدم‌ها مرتکب می‌شوند، این است که فکر کنند فقط تیم آزمون‌گر مسؤول تضمین کیفیت است.»

برایان ماریک

۳-۱-۱۷ راهبردی برای آزمون نرم‌افزار

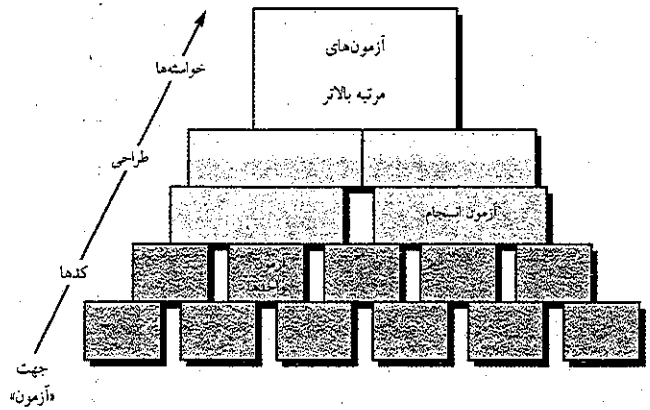
فرایند مهندسی نرم‌افزار را می‌توان به صورت مارپیچ شکل ۱-۱۷ در نظر گرفت. در ابتدا، مهندسی سیستم یا مهندسی اطلاعات، نقش نرم‌افزار را تعیین می‌کند و به تحلیل خواسته‌های نرم‌افزار می‌انجامد، که در آن دامنه اطلاعاتی، عملکرد، رفتار، کارایی، شرایط حدی و ملاک‌های اعتبارسنجی مربوط به نرم‌افزار برقرار می‌شوند. 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 دیگر 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.

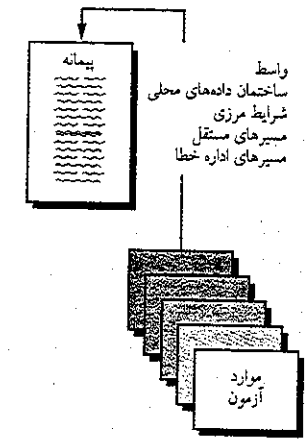
اهداف و آزمون را به وضوح بیان کنید. اهداف مشخص آزمون باید به‌صورتی قابل اندازه‌گیری بیان شوند. مثلاً، اثربخشی آزمون، پوشش‌دهی آزمون، میانگین زمان شکست، هزینه یافتن و برطرف‌ساختن نقایص، چگالی نقایص باقیمانده یا فراوانی وقوع و تعداد ساعت‌های کار به ازای هر آزمون رگرسیون، همگی باید در طرح آزمون بیان شوند [Gil95].

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



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.

«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.

کدام خطاها معمولاً طی آزمون واحد کشف می‌شوند؟

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!

مرجع وب

درباره انواع مقالات و منابع مربوط به «آزمون چابک» در نشانی testing.com/agile اطلاعات مفیدی خواهید یافت.

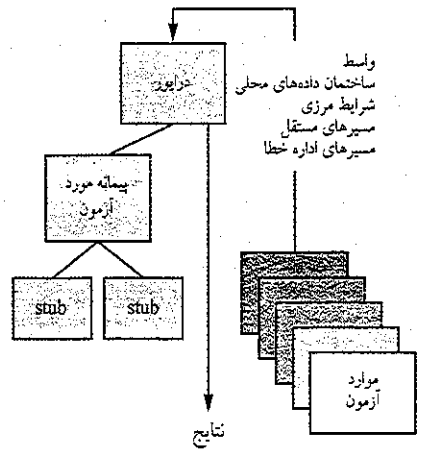
از میان خطاهای متداول‌تر در محاسبات، می‌توان به موارد زیر اشاره نمود: (۱) تقدم محاسباتی نادرست، یا آنهایی که به‌درستی درک نشده‌اند؛ (۲) گوناگونی عملیات؛ (۳) مقداردهی اولیه نادرست؛ (۴) دقت نادرست؛ (۵) ارائه نمادهای نادرست برای یک رابطه یا عبارت.

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.

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).

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.

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.

۱۷-۳-۲ آزمون انسجام (Integration Testing)

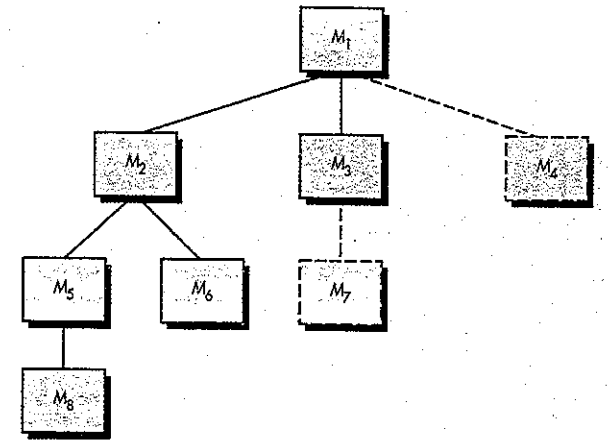
یک فرد مبتدی در جهان نرم‌افزار ممکن است پس از انجام آزمون واحدها روی همه‌ی پیمانه‌ها این پرسش را مطرح کند: «اگر همه‌ی پیمانه‌ها به‌تنهایی کار می‌کنند، چه دلیلی دارد که در کنار هم کار نکنند؟» البته در کنار هم کار کردن، مسأله‌ی ایجاد رابطه میان آنها را مطرح می‌سازد. ممکن است داده‌ها در گذر از یک واسط از 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 گام‌به‌گام 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 به‌پایین، 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.

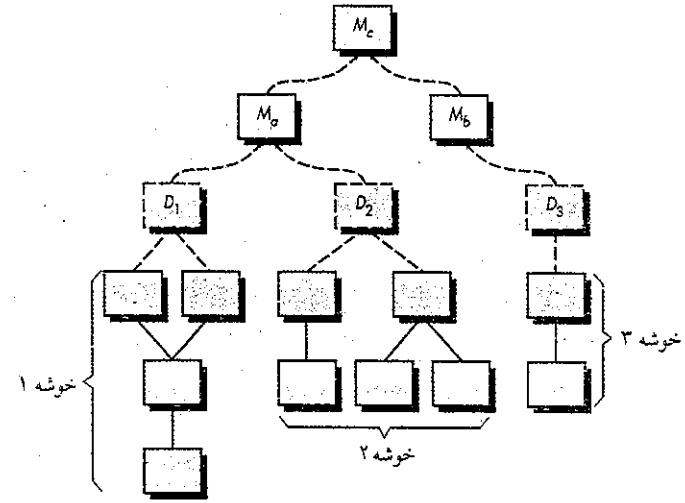
۲. یک درایور (برنامه کنترلی برای آزمون) نوشته می‌شود تا ورودی و خروجی موارد آزمون را هماهنگ کند.

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 کاهش‌دادن «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. 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 سخت‌افزار، 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.

۶-۱۷ 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؟

۷-۱۷ 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 اوضوع 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 [Bci84] 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 تخصص‌یافته‌ی 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)

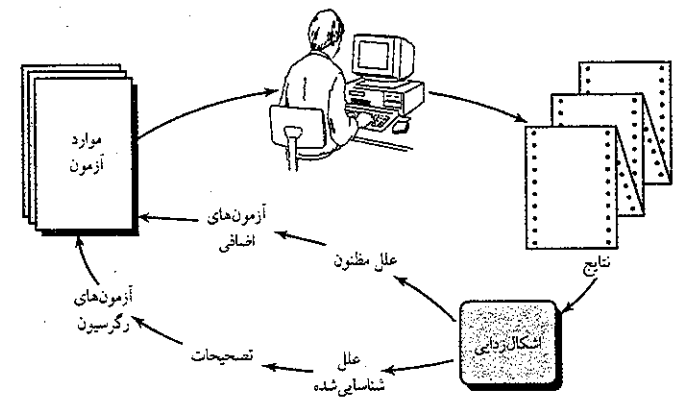
آزمون نرم‌افزار، فرایندی است که می‌توان آن را به‌طور سیستماتیک برنامه‌ریزی و مشخص نمود. مواردآزمون را می‌توان طراحی کرد، راهبردی را تعیین نمود و نتایج را در مقابل انتظارات تجویز شده مورد سنجش قرار داد.

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.

[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!

> «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.

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 مقطعی 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 این‌قدر 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 بارها 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.»
>
> بروس کانر و سایرین

[1] مفهوم برنامه‌نویسی جفتی (که به‌عنوان بخشی از مدل برنامه‌نویسی حذفی 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 گردهمایی 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

۱۷-۱ تفاوت میان وارسی و اعتبارسنجی را به زبان ساده بیان کنید. آیا هر دو از راهبردهای آزمون و روش‌های طراحی موارد آزمون استفاده می‌کنند؟

۱۷-۲ 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 به‌دنبال 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.

«خطاها در نرم‌افزارها، متداول‌تر، شایع‌تر و مشکل‌آفرین‌تر از خطاهای موجود در سایر فن‌آوری‌ها هستند»
دیوید پارناس

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.

آزمون «خوب» چیست؟

خصوصیات آزمون. درباره خودِ آزمون‌ها چه می‌توان گفت؟ کینر، فالک و نگوین [Kan93] برای آزمون «خوب» صفات زیر را برمی‌شمرند:

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. بی‌آن‌که 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.

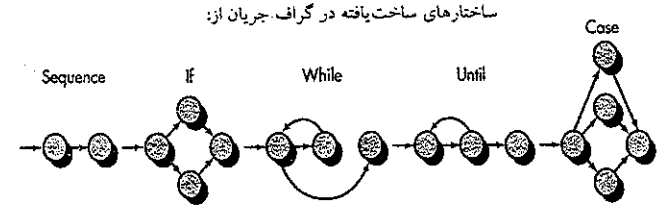
> «اشکال‌ها در گوشه و کنارها در کمین هستند و در نقاط مرزی گرد هم مجتمع می‌شوند.»
>
> بوریس بیزر

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)

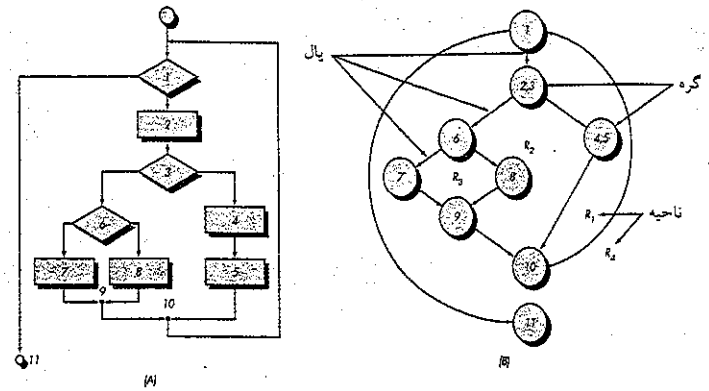
پیش از آن‌که بتوان روش مسیرهای پایه را معرفی نمود، یک نمادگذاری ساده، موسوم به *گراف جریان* یا (*گراف برنامه*) را باید برای نمایش دادن جریان کنترل معرفی کرد[1]. گراف جریان، جریان کنترل منطقی را با استفاده از نمادگذاری در شکل ۱-۱۸ تصویر می‌کند. متناظر با هر ساختار ساخت‌یافته (فصل ۱۰) یک نماد گراف جریان 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.

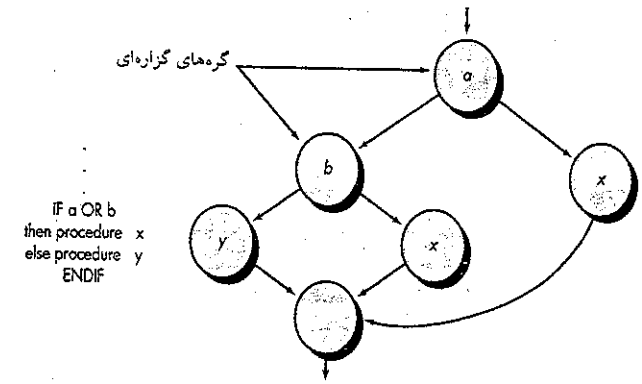
لوزی‌های تصمیم‌گیری نمودار گردشی وجود نداشته باشد). با توجه به شکل ۲-۱۸ب، هر دایره که گره *گراف جریان* خوانده می‌شود، یک یا چند دستور رویه‌ای را نشان می‌دهد. ترتیبی از مستطیل‌های پردازشی و یک لوزی تصمیم‌گیری را می‌توان در یک گره منفرد خلاصه نمود. پیکان‌های روی گراف جریان، که *یال* یا *پیوند* خوانده می‌شوند، 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. عملیات‌هایی‌که $V(G)$ 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;
    total.input = total.valid = 0;   (2)
(3) sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100   (4)
(4) increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   (6)
(5)       THEN increment total.valid by 1;
(7)            sum = s sum + value[i]
          ELSE skip
(8)    ENDIF
       increment i by 1;
(9) ENDDO
    IF total.valid > 0   (10)
(11)   THEN average = sum / total.valid;
(12)   ELSE average = -999;
(13) 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

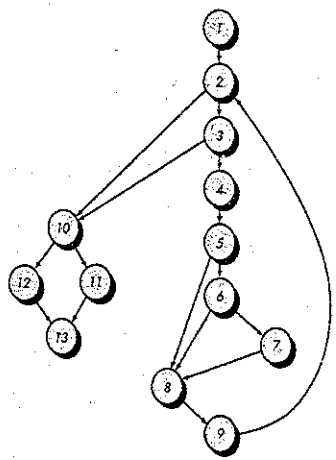
مسیر ۳: ۱-۲-۳-۱۰-۱۱-۱۳

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.**

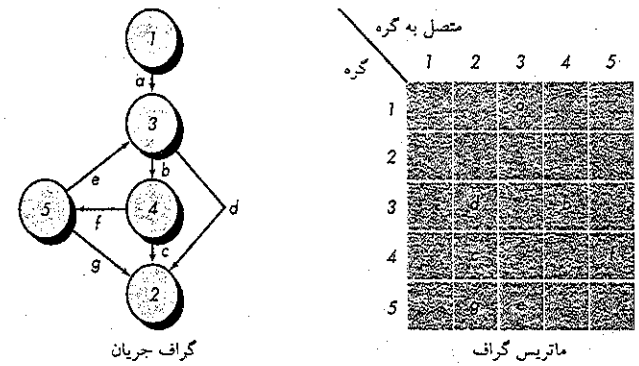
داده‌ها را باید طوری انتخاب کرد که به موازات آزموده شدن هر مسیر، شرایط در گره‌های گزاره‌ای به‌طور مناسب تنظیم شود. هر مورد آزمون اجرا می‌شود و با نتایج مورد انتظار مقایسه می‌شود. هنگامی که همه‌ی موارد آزمون کامل شدند، آزمون‌گر می‌تواند مطمئن شود که همه‌ی دستورهای برنامه حداقل یک بار اجرا شده‌اند.

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.

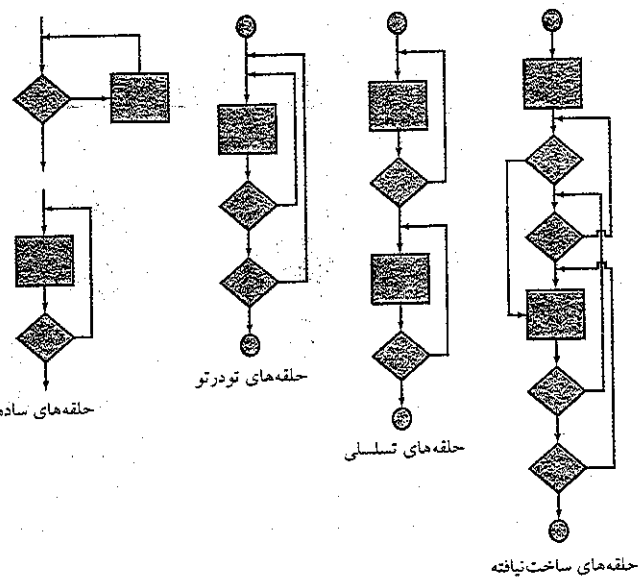
۳. دوبار گذر از حلقه.

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.

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).

اندرز

حلقه‌های غیرساخت‌یافته را نمی‌توانید به‌طور اثربخش آزمایش کنید. آن‌ها را بازآرایی کنید.

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

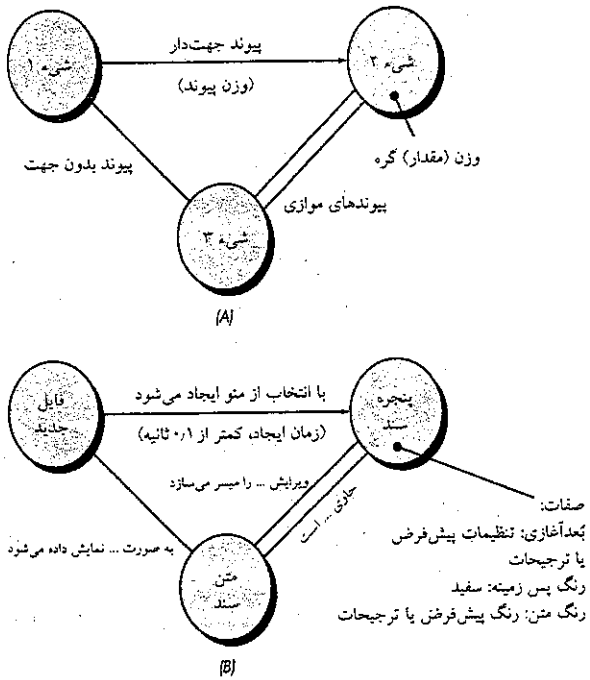
نخستین گام در آزمون جعبه سیاه، شناخت اشیایی[1] است که در نرم‌افزار مدل‌سازی می‌شوند و نیز روابط میان این اشیاست. هنگامی که این منظور برآورده شد، مرحله بعدی تعیین تعدادی آزمون است که ثابت کنند همه‌ی اشیاء، رابطه‌ی موردانتظار را با یکدیگر دارند [Bei95]. به بیان دیگر، آزمون نرم‌افزار با ایجاد گرافی از اشیای 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 $FTW = 0.62 \times GW$ 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.

۱۸-۶-۲ 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.

اندرز

کلاس‌های ورودی در فرایند نرم‌افزار نسبتاً زودهنگام شناسایی می‌شوند. به این دلیل، تفکر درباره افراز هم‌ارزی را به مراحل ایجاد طراحی شروع کنید.

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:

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.

کلاس‌های هم‌ارزی را برای آزمون‌ها چگونه تعریف کنم؟

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.

۱۸-۶-۳ 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.

یک راه مؤثر برای آزمایش کدها، آزمای آن‌ها در مرزهای طبیعی است.»
برایان کرنیگان

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] .

نکته‌ی کلیدی

BVA افراز هم‌ارزی را با تأکید بر «لبه‌های» یک دسته‌ی هم‌ارزی بسط می‌دهد.

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.

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.

۱۸-۶-۴ 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.

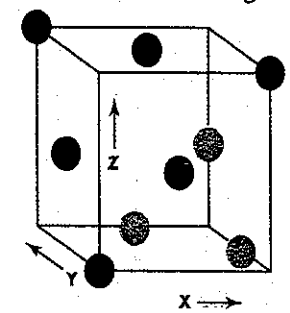
برای نشان دادن کاربرد آرایه متعامد L9 عمل ارسال را برای نمابر در نظر بگیرید. چهار پارامتر P۱، P۲، P۳ و P۴ به «عمل ارسال» تحویل داده می‌شود. هر یک از آنها سه مقدار می‌گیرد. مثلاً P۱ مقادیر زیر را می‌گیرد:

P۱=۱ همین الان ارسال کن

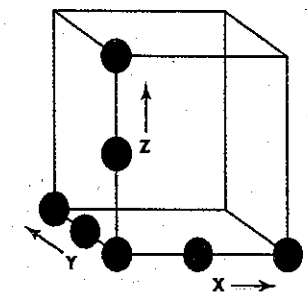
P۱=۲ یک ساعت بعد ارسال کن

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 است؛ 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 caseها 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 اقداماتی 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 زفتاری 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

۱-۱۸ 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.

۲-۱۸ 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.

۳-۱۸ آیا می‌توانید اهداف دیگری برای آزمون برشمرید که در بخش ۱-۱-۱۸ ذکر نشده باشد؟

۴-۱۸ 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.

۵-۱۸ 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.

۶-۱۸ 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.

۷-۱۸ یک ابزار خودکار طراحی کنید که حلقه‌ها را شناسایی کرده آنها را مانند بخش ۳-۵-۱۸ گروه‌بندی کند.

۸-۱۸ 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.

۹-۱۸ 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.

۱۰-۱۸ آیا آزمون جامع (حتی اگر برای برنامه‌های بسیار کوچک امکان‌پذیر باشد) متضمن آن است که برنامه ۱۰۰٪ درست است؟

۱۱-۱۸ 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.

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.

[1] تکنیک‌های مدل‌سازی خواسته‌ها و طراحی در بخش دوم این کتاب ارائه شدند. مفاهیم پایه‌ی شیءگرایی در پیوست ۲ ارائه خواهد شد.

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.

۱۹-۳-۱ 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.

۱۹-۳-۲ 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.

نکته‌ی کلیدی
در آزمون انسجام برای نرم‌افزارهای شیءگرا، آنچه که آزمایش می‌شود، مجموعه کلاس‌هایی است که باید به رویدادی معین پاسخ بدهند.

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.

۱۹-۳-۳ 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 use caseهایی (فصل‌های ۵ و ۶) را در نظر بگیرد که بخشی از مدل تحلیل هستند. use case 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.

۱۹-۴ 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 بک

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.

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.

نکته‌ی کلیدی

رامبرد 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؟

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 آزمونی 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.

۱۹-۴-۵ 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 caseها) 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 caseها 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

آندرژ

گرچه آزمون مبتنی بر سناریو مزیت‌هایی در بر دارد، با صرف زمان روی مرور use caseها به‌هنگام تهیه‌ی آنها به‌عنوان بخشی از مدل تحلیل، نتیجه‌ی بیشتری عایدتان خواهد شد.

۱۹-۴-۶ 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).

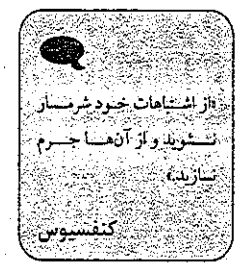
مطمئناً در جزئیات تفاوت وجود دارد. برای مثال، در یک سیستم سنتی با واسطی که از طریق صدور فرمان کار می‌کند، کاربر ممکن است فهرستی از فرمان‌ها را به‌عنوان یک فهرست کنترلی آزمون استفاده کند. اگر هیچ سناریوی آزمونی برای امتحان فرمان وجود نداشته باشد، احتمالاً آزمون برخی از وظایف کاربر را نادیده گرفته است (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 عمیقی مشابه با آزمون جعبه سفید است.



آندرز
تعداد تبدیلات جایگشتی برای آزمون تصادفی می‌تواند بسیار بزرگ باشد. برای بهبود بخشیدن به بازدهی آزمون می‌توان از راهبردی مشابه با آزمون آرایه‌های متعامد بهره برد.

۵-۱۹ 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].

۱-۵-۱۹ 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

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.

۲-۵-۱۹ 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( )*، *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.

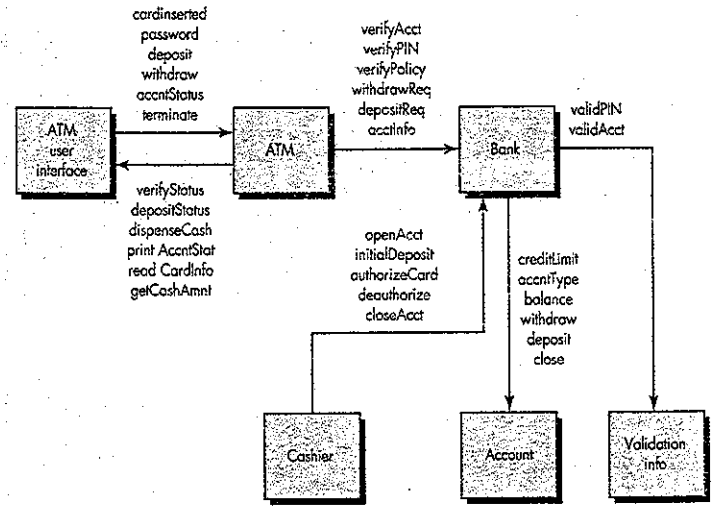
(۲) عملیات‌هایی که *creditLimit* را اصلاح می‌کنند، و (۳) عملیات‌هایی که *creditLimit* را اصلاح نمی‌کنند. سپس دنباله‌های آزمون برای هر افراز طراحی می‌شود.

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:

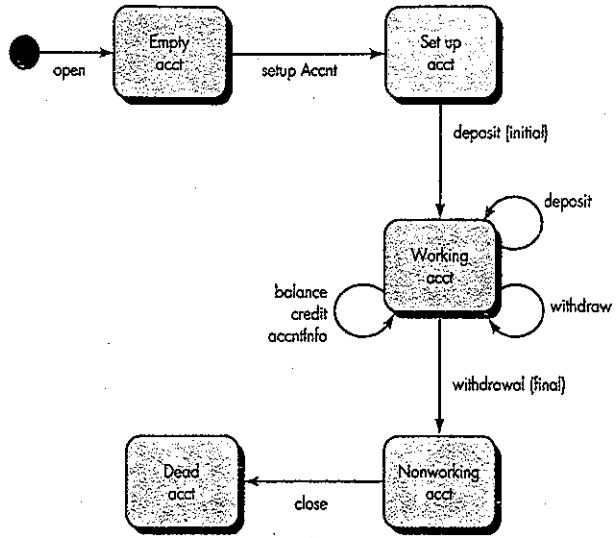
verifyAcctBank[validAcctValidationInfo].verifyPINBank

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 ۱۹-۵ 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.

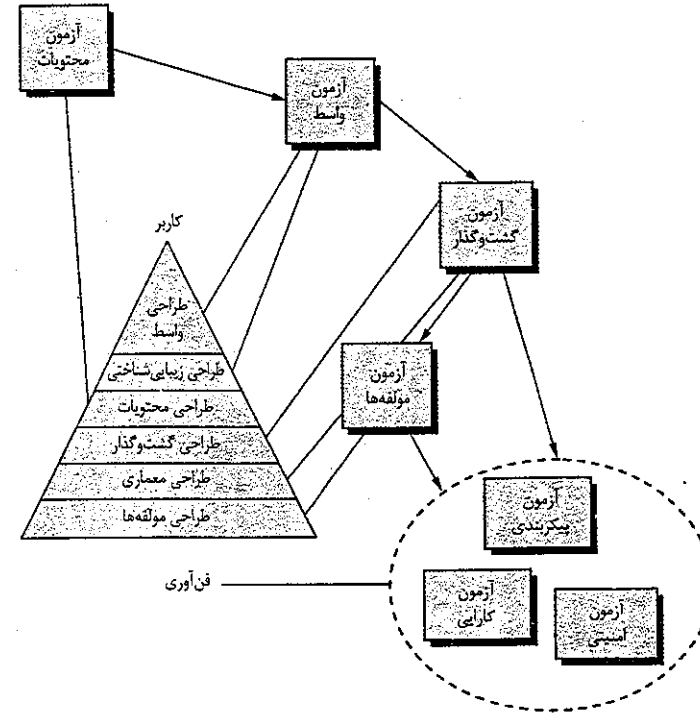
نکته‌ی کلیدی
در طرح آزمون، مجموعه وظایف آزمون، محصولات کاری‌ای که باید توسعه داده شوند و روش ارزیابی ثبت و استفاده‌ی مجدد از نتایج تعیین می‌شود.

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 ــ اختلاف ــ 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].

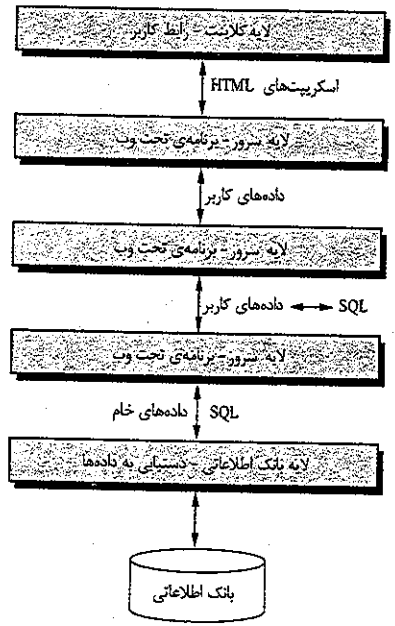
۳. داده‌های خام به‌دست آمده از بانک اطلاعاتی باید به سرور برنامه‌ی تحت وب ارسال شوند و برای انتقال بعدی به کلاینت به فرمت مناسب درآیند. بنابراین، آزمون‌هایی باید فراهم آورده شوند که اعتبار داده‌های خامِ دریافت شده توسط سرور برنامه‌ی تحت وب را به نمایش بگذارند و علاوه بر آن، آزمون‌هایی نیز باید ایجاد 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 متون 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؟

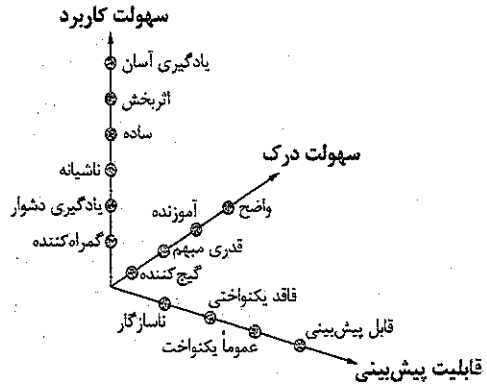
دسترس‌پذیری- آیا برنامه‌ی تحت وب برای افرادی با ناتوانی‌های خاص قابل دستیابی هست؟

کدام خصوصیات قابلیت استفاده در آزمون، کانون توجه قرار می‌گیرند و چه اهداف خاصی باید دنبال شوند؟

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

[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.

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.

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

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] پیچیدگی‌منطقی‌را می‌توان با محاسبه‌ی پیچیدگی‌سیکلوماتیک الگوریتم تعیین‌کرد.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 دفترچه‌ای 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 جالبترین 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

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؟

با آزمایش هر NSU چه پرسش‌هایی باید پرسیده و پاسخ داده شوند؟

- 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:

به هنگام اجرای آزمون پیکربندی در طرف سرور، چه پرسش‌هایی باید پرسیده و پاسخ داده شوند؟

- 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؟

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 سودجود 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.

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.

نکته‌ی کلیدی

آزمون‌های امنیتی باید طوری طراحی شوند که دیوارهای آتش، صدور مجوز، پنهان‌سازی و احراز هویت را تمرین دهند.

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.

۲۰-۹-۱ 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 دیدِ کاربر نهایی، دشوار است. ازدحام بار شبکه، تغییرات سخت‌افزارهای واسط و مسائل مشابه در سطح برنامه‌ی تحت وب به آسانی قابل آزمایش نیستند.

۲۰-۹-۲ 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.

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.

اندرز

اگر برنامه‌ی تحت وبی برای فراهم آوردن ظرفیت بالا از چند سرور استفاده می‌کند، آزمون ازدحام بار باید در محیط چندسروری اجرا شود.

۲۰-۹-۳ 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:

- 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؟

نکته‌ی کلیدی

هدف آزمون فشار، درک بهتر چگونگی شکست‌خوردن یک سیستم با اعمال فشارهایی ورای حدود عملیاتی آن است.

- 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:

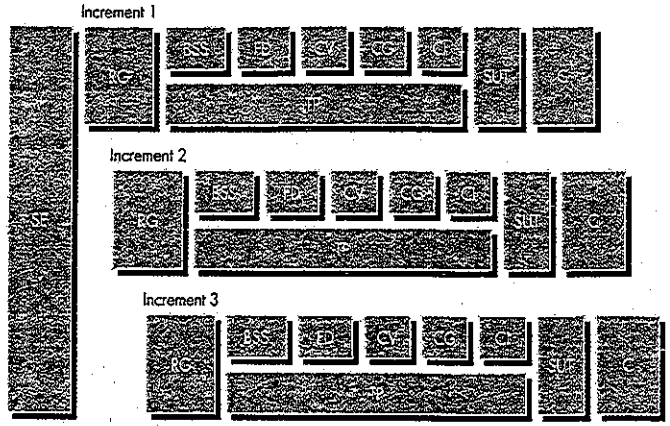
> روش‌های رسمی [و مهندسی نرم‌افزار اتاق تمیز] بحث برانگیزند. مدافعان این روش‌ها ادعا می‌کنند که می‌توانند [نرم‌افزار] را متحول سازند و مخالفان این روش‌ها معتقدند که بسیار دشوار و غیر ممکن هستند. در ضمن، برای اکثر افراد، روش‌های رسمی [و مهندسی نرم‌افزار اتاق تمیز] چنان ناآشنا هستند که قضاوت درباره این 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.

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

«از نکات جالب زندگی این است که اگر از پذیرفتن هر چیزی مگر بهترین آن سرباز بزنید، معمولاً بهترین‌ها نصیبتان خواهد شد.»

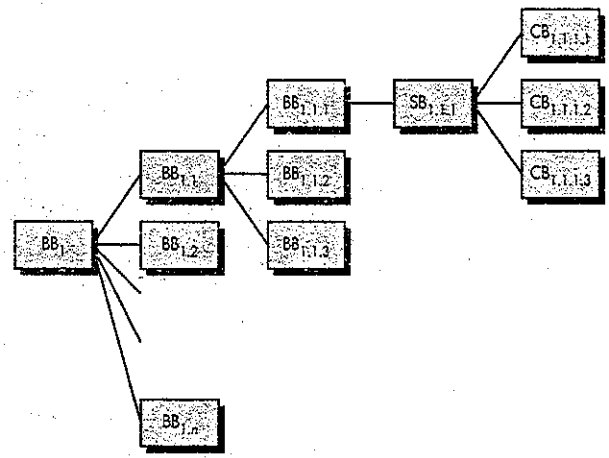
سامرست موام

پالایش چگونه به‌عنوان بخشی از مشخصات ساختار چهارگوش عملی می‌شود؟

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.

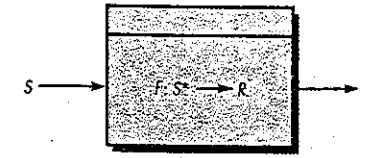
نکته‌ی کلیدی

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 *f* روی 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.

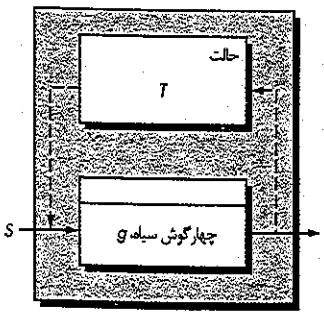


شکل ۳-۲۱ یک مشخصه‌ی چهارگوش سیاه.

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.

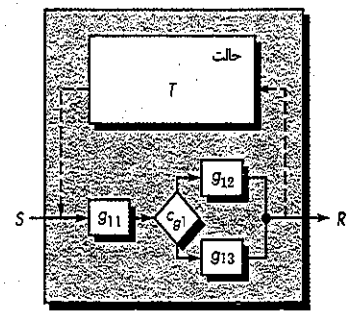
$$g : S^* \times T^* \rightarrow R \times T$$

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.

۲۱-۳-۱ 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 را انجام می‌دهد؛ و هرگاه <c> نادرست باشد، آیا r 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

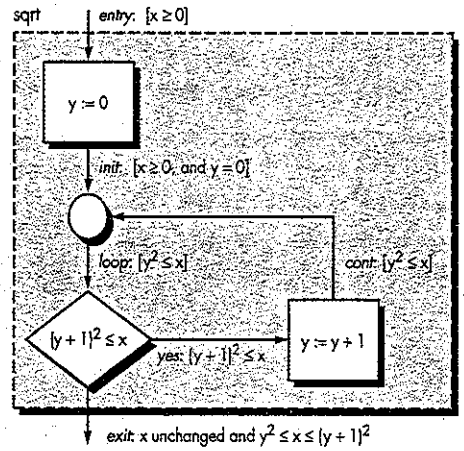
اگر در توسعه‌ی طراحی روالی، فقط خودتان را به ساختارهای ساخت‌یافته محدود کنید، اثبات درستی، کاری صریح خواهد بود. اگر از این ساختارها عدول کنید، اثبات درستی ممکن است دشوار یا حتی غیرممکن شود.

۲۱-۳-۲ 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 $x \leq (y+1)^2$ 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 $x \leq y^2$). بنابراین، برای برقرار شدن شرط exit می‌توان $x > (y+1)^2$ و $x \leq y^2$ 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.

> «کیفیت، نه یک عمل بلکه عادت است.»
>
> ارسطو

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 ها) 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.

> **اندرز**
>
> حتی اگر تصمیم گرفتید که از رویکرد اتاق تمیز استفاده نکنید، خوب است آزمون کاربرد آماری را به‌عنوان بخشی جدایی‌ناپذیر 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

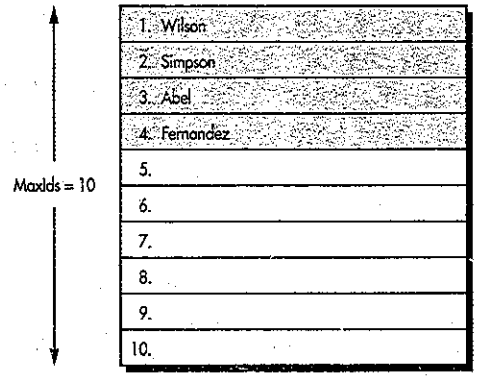
در فرهنگ بزرگ مهندسی نرم‌افزار [Mar01]، روش‌های رسمی به شیوه زیر تعریف می‌شوند:

> 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.

روش‌های رسمی توان بالقوه‌ی عظیمی برای بهبود بخشیدن به وضوح و دقت مشخصات خواسته‌ها و یافتن خطاهای مهم و ظریف دارند.
استیو استربروک



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.

نکته‌ی کلیدی
ثابت داده‌ای، مجموعه‌ای از شرایط است که در سرتاسر اجرای سیستمی، حاوی مجموعه داده‌های درست است.

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

[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.

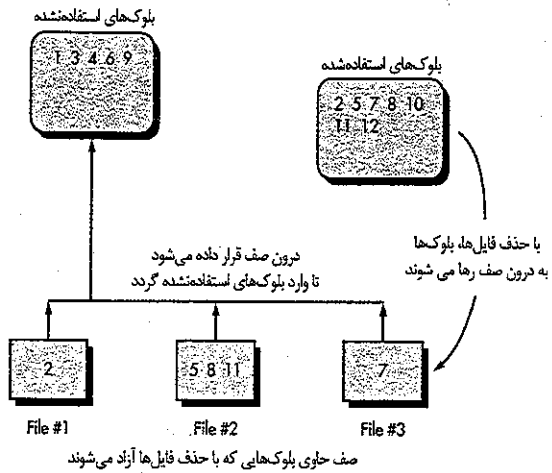
مفهوم نهایی، *عملیات* است. عملیات، فعالیتی است که در داخل سیستم انجام می‌شود و داده‌ها را می‌خواند یا می‌نویسد. اگر افزودن یا حذف‌کردن نام‌ها از جدول نمادها از جمله وظایف برنامه‌ی جدول نمادها باشد، این برنامه با دو عملیات مرتبط خواهد بود: یک عملیات *add( )* برای افزودن نام مشخصی به جدول نمادها و یک عملیات *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 قیودشان 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 موثقی 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:

$$used' = used \setminus head\ BlockQueue \wedge$$

$$free' = free \cup head\ BlockQueue \wedge$$

$$BlockQueue' = tail\ BlockQueue$$

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.

پس‌شرط‌ها و پیش‌شرط‌ها را چگونه نمایش می‌دهیم؟

$$Ablocks \subseteq used$$

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.
$C->forAll$(elem|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.
$C->forAll$(elem1,elem2|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.
$C->isUnique$(elem|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.

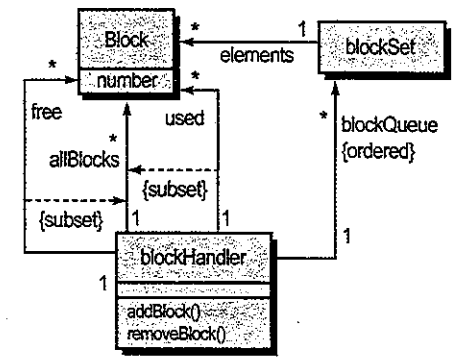
شیء **Self** عنصر نمودار UML است که عبارت OCL در حیطه‌ی آن تعیین می‌شود. سایر اشیا را می‌توان با *گشت و گذار* و با استفاده از نماد نقطه (.) از شیء **Self** به‌دست آورد. برای مثال:

- 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.

۲۱-۷-۲ 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

مرجع وب

اطلاعات مشروح درباره زبان Z را می‌توانید در وب سایت زیر یابید:

www.users.cs.york.ac.uk/~susan/abs/z.htm

[1] به‌خاطر دارید که در فصل های دیگر، *حالت* برای شناسایی یک شیوه رفتاری سیستم به‌کار رفته است که از بیرون قابل مشاهده است.

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 دریک عملیات تغییر نمی‌کند.

مثال زیر از یک شِما، حالت مدیریت بلوک‌ها (*BlockHandler*) و ثابت داده‌ای را توصیف می‌کند:

BlockHandler

$used,\ free:\ \mathbb{P}\ BLOCKS$

$BlockQueue: \text{seq}\ \mathbb{P}\ BLOCKS$

$used \cap free = \varnothing \land$

$used \cup free = AllBlocks \land$

$\forall i: dom\ BlockQueue \bullet BlockQueue\ i \subseteq used \land$

$\forall i, j: \text{dom}\ BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$

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

$\Delta BlockHandler$

$Ablocks? : \text{BLOCKS}$

$Ablocks? \subseteq used$

$BlockQueue' = BlockQueue \cap \langle Ablocks \rangle \land$

$used' = used \land$

$free' = free$

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.

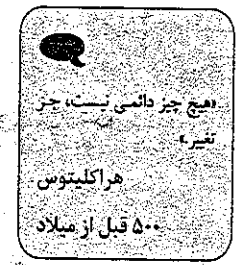
درک تفاوت میان پشتیبانی نرم‌افزار و مدیریت پیکربندی نرم‌افزار حائز اهمیت است. پشتیبانی عبارت از یک مجموعه فعالیت‌های مهندسی نرم‌افزار است که پس از تحویل نرم‌افزار به مشتری و به‌کار انداختن آن رخ می‌دهد. مدیریت پیکربندی نرم‌افزار، مجموعه‌ای از فعالیت‌های پیگیری و کنترل است که با شروع‌شدن پروژه مهندسی 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.

۱-۲۲ 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.»



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؟

۱-۱-۲۲ 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.

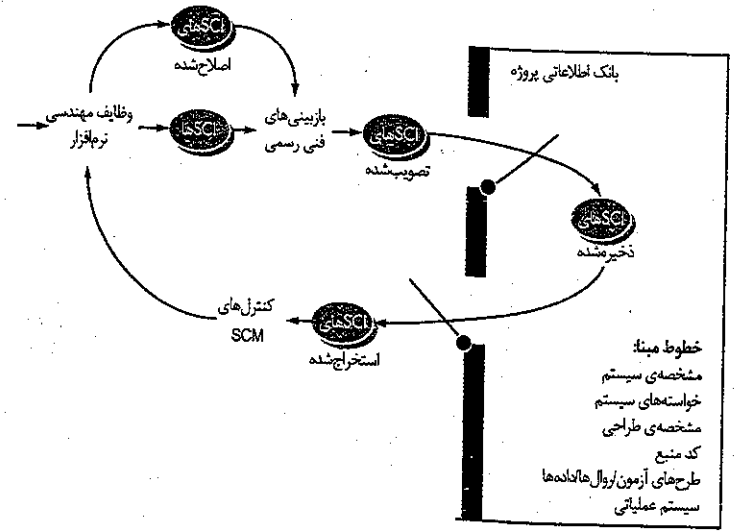
خط مبنا یک مفهوم مدیریت پیکربندی نرم‌افزار است که به کنترل تغییرات، بدون جلوگیری کردن از تغییراتِ موجه، کمک می‌کند. در استاندارد IEEE 610.12-1940 خط مبنا به صورت زیر تعریف می‌شود:

> 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.

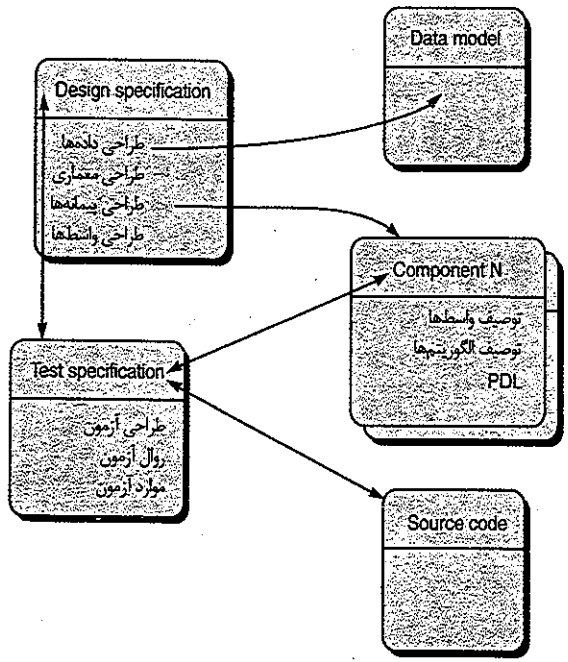
بسیاری از سازمان‌های مهندسی نرم‌افزار، علاوه بر SCIهای به‌دست آمده از محصولات کاری نرم‌افزار، ابزارهای نرم‌افزاری تحت کنترل پیکربندی را هم قرار می‌دهند. یعنی، نسخه‌های مشخصی از ویراستارها، کامپایلرها، مرورگرها و سایر ابزارهای خودکار شده به‌عنوان بخشی از پیکربندی نرم‌افزار «منجمد» می‌شوند. از آن‌جا که 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 را به بانک اطلاعاتی پروژه در یک مکان 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.

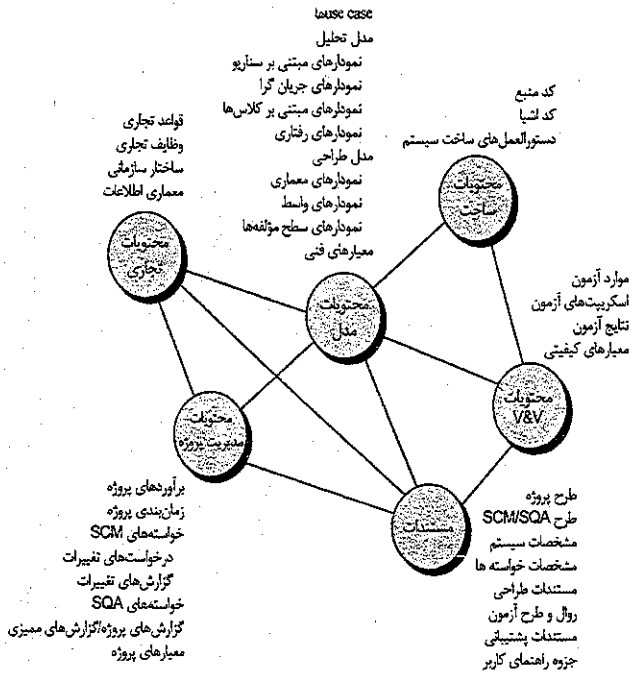
مخزنی که به تیم مهندسی نرم‌افزار خدمات می‌دهد، همچنین باید (۱) با قابلیت‌های مدیریت فرایند پشتیبانی منسجم باشند یا مستقیماً آن‌ها را پشتیبانی کند، (۲) قواعد ویژه‌ای را که بر عملکرد

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 ۳-۲۲ 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 ۲-۳-۲۲) 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 موثرکردن 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.

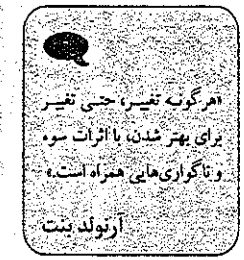
جلسات ممیزی. در جلسه ممیزی، اطلاعات اضافی درباره زمان، علت و عامل انجام دهنده‌ی تغییرات فراهم می‌آید. اطلاعات مربوط به منبع تغییرات را می‌توان به‌عنوان صفات اشیای خاص در مخزن وارد کرد. یک سازوکار آغازگر مخزن می‌تواند در پیام دادن به سازنده یا ابزار مورد استفاده کمک کند تا واردکردن اطلاعات ممیزی (از قبیل دلیل تغییر) به هنگام اصلاح یک عنصر طراحی آغاز شود.

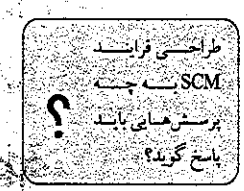
۲۲-۳ فرایند SCM

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؟

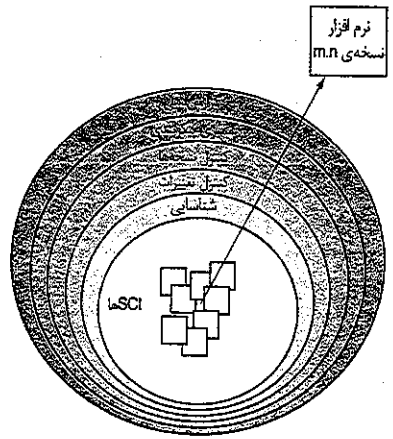




- چگونه می‌توان اطمینان حاصل کرد که تغییرات به‌طور مناسب اِعمال شده است؟
- چه سازوکاری برای آگاه ساختن دیگران از اِعمال تغییرات به‌کار می‌رود؟

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> requiremenis 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، ابده‌های درخشان در ذهن همین ناشی نقش می‌بندد 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.

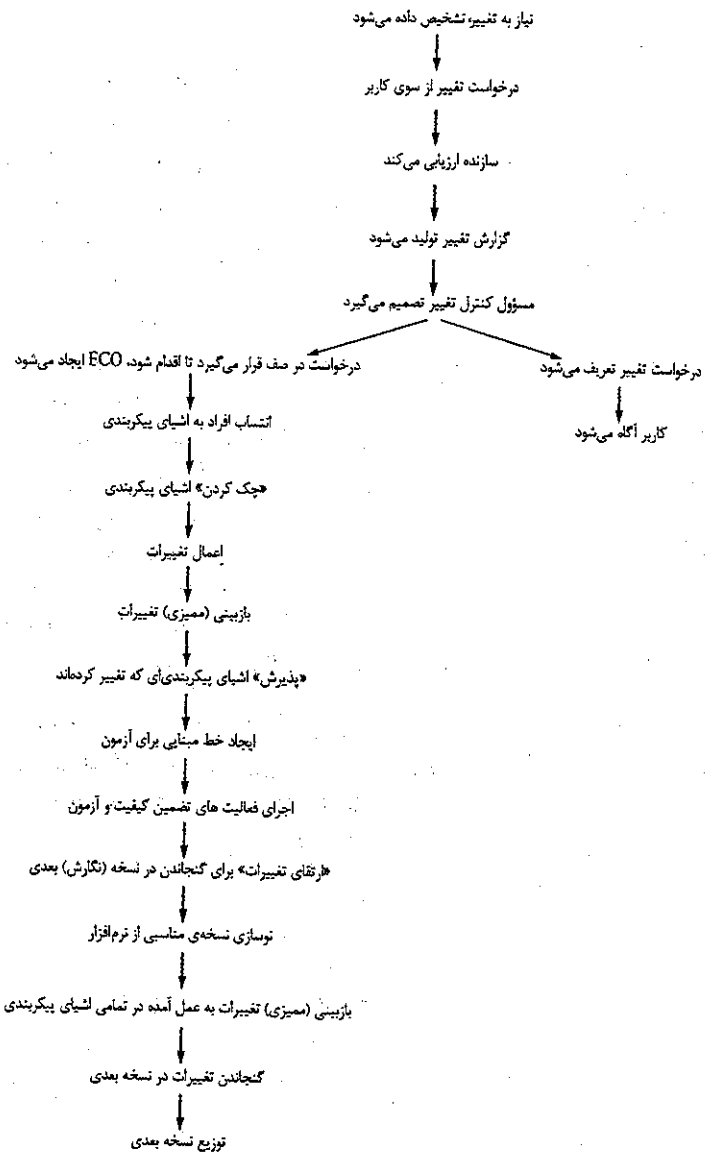
نکته‌ی کلیدی

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، بیش 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 زامان، 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 نیزا دارند، قابل دستیابی می‌سازد.»

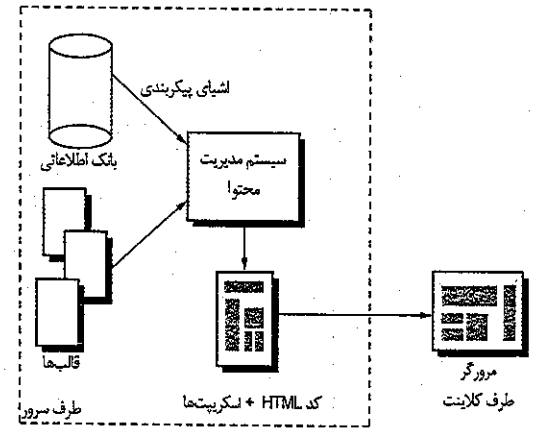
۳-۴-۲۲ مدیریت محتوا (Content Management)

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
زیرسیستم مدیریت، برای همه‌ی 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:

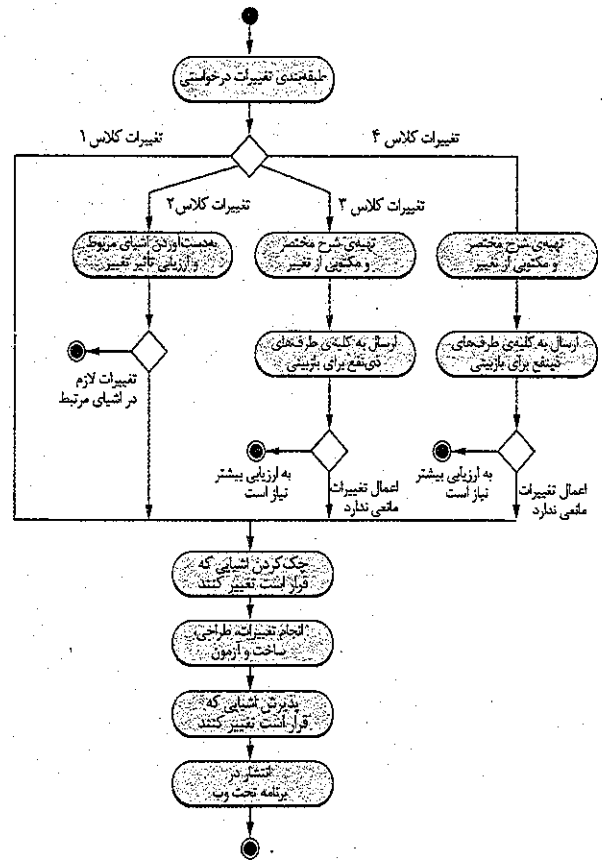
کلاس ۱ - تغییری در محتوا یا عملکردهایی که با یک خطا در ارتباط است یا محتوا و عملکردهای محلی را بهبود می‌بخشد.

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:

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

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

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 بُعدی، 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.

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.

> «بلوغ یک علم به اندازه‌ی بلوغ ابزارهای اندازه‌گیری آن است.»
>
> لویی پاستور

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 بپرسیم این معیارهای تکنیکی تا چه حد اعتبار دارند. یعنی معیارهای تکنیکی چقدر با قابلیت اطمینان و کیفیت سیستم کامپیوتری همبستگی 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.

مرجع وب

اطلاعات فراوانی در خصوص معیارهای محصول توسط هورست زوس در آدرس زیر جمع آوری شده است:
Irb.cs.tu-berlin.de/~zuse/

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

مراحل یک فرایند اندازه‌گیری اثربخش از چه قرار است؟

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.

اندرز

در واقع، بسیاری از معیارهای محصول که امروزه به‌کار 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.

[1] گرچه نقد معیارهای خاص در متون رایج است، بسیاری از منتقدان، 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.

- *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.

مرجع وب

بحث مفیدی درباره‌ی GQM را می‌توان در وب‌سایت زیر یافت:
www.thedacs.com/GoldPractices/practices/gqma.html

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.

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

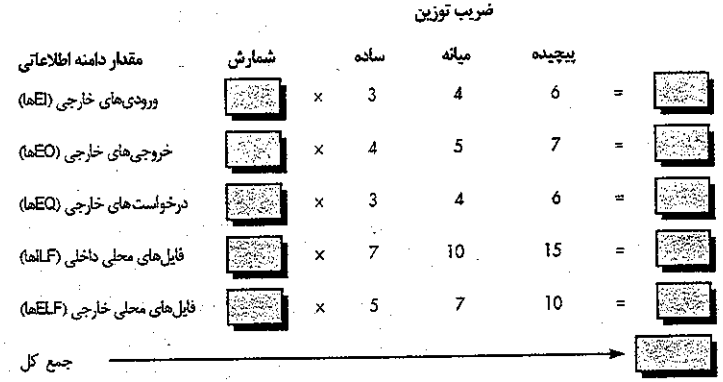
[1] صدها کتاب و مقاله درباره معیارهای FP نوشته شده است. فهرست خوبی از این نوشته‌ها را در [IFP05] می‌توانید بیابید.

[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]:

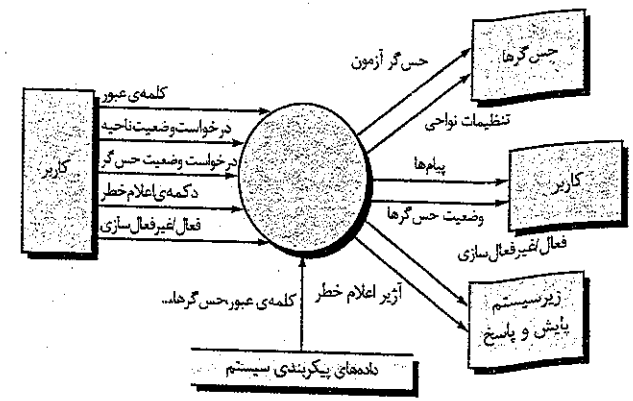
۱. آیا سیستم به تهیه‌ی پشتیبان و بازیابی قابل اطمینان نیاز دارد؟
۲. آیا ارتباطات داده‌ای موردنیاز است؟
۳. آیا عملیات‌های پردازشی توزیع شده وجود دارند؟
۴. آیا کارایی اهمیت دارد؟
۵. آیا سیستم در یک محیط عملیاتی موجود و پرکاربرد اجرا می‌شود؟
۶. آیا سیستم به مدخل داده‌های آنلاین نیاز دارد؟
۷. آیا مدخل داده‌های 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:
Irb.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.

نکته‌ی کلیدی

معیارها می‌توانند دیدی از پیچیدگی داده‌ای و مرتبط با طراحی معماری در اختیار قرار دهند.

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.

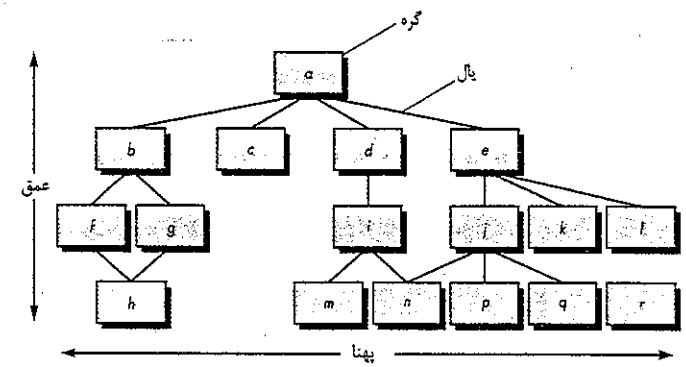
پهنا = حداکثر تعداد گره‌ها در هر سطح از معماری. برای معماری شکل ۴-۲۳، پهنا برابر ۶ است.

---

[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.

گالیله

«چیزی را اندازه بگیرید که قابل اندازه‌گیری باشد و آنچه را که قابل اندازه‌گیری نیست، قابل اندازه‌گیری کنید.»



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

هنگام ارزیابی یک طراحی شیءگرا، کدام خصوصیات را می‌توان اندازه‌گیری کرد؟

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.

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 قلاً می‌بایست با اتکا بر نظر عامه و اساطیر انجام شوند، اکنون با به کارگیری داده‌های کمی قابل انجام هستند.»
>
> اشکات ویتمایر

۲۳-۳-۳ 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 **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.

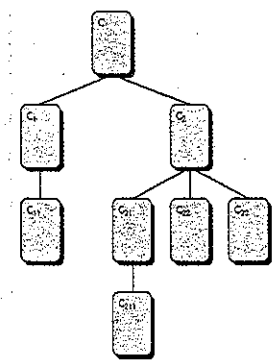
اندرز

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 موردی 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 شدم.

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.

۲۳-۳-۵ 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 برخوردند، 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.

۲۳-۳-۶ 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.

نکته‌ی کلیدی

پیچیدگی سیکلوماتیک تنها یکی از چندین معیار پیچیدگی است.

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.

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 با 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 [Fel89].

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 پکیج‌هایی 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.