به نام خدا

مقدمه‌ای بر بازآرایی(Refactoring)

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

by: einali{at}[ce.sharif.edu | gmail.com]

فهرست مطالب

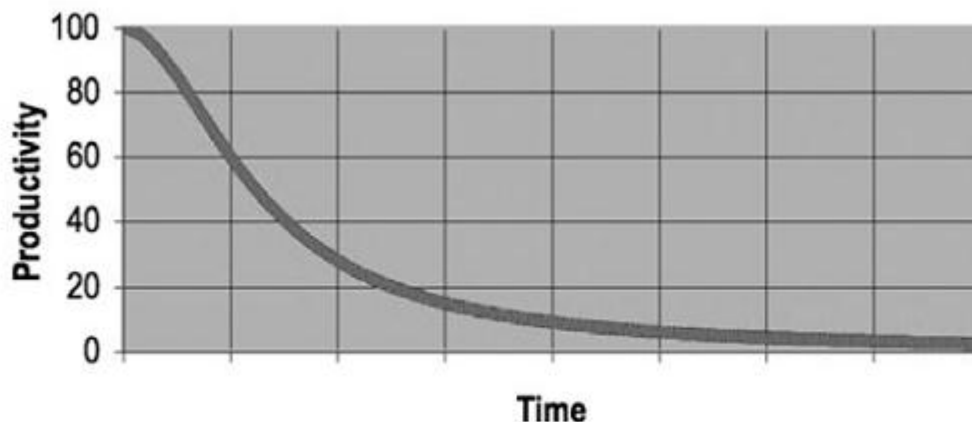
مقدمه ۳
داستان کد کثیف ۴
ریشه کد کثیف ۷
بازآرایی چیست؟ ۷
آزمون واحد ۸
مزایای بازآرایی ۹
کلاه جدیدی برای بازآرایی ۱۰
خشت های خام ۱۱
کد تکراری ۱۲
متد طولانی ۱۳
کلاس بزرگ ۱۴
فهرست بلند پارامترها ۱۵
تغییر واگرا ۱۶
جراحی ساچمه ها ۱۶
چشم داشتن به امکانات دیگران ۱۷
خوشه های داده ۱۸
توصیه های مربوط به خوانایی کد ۱۹
کد تمیز ۲۰
منابع ۲۱

مقدمه

برنامه نوشته شده یا همان کد، خروجی اصلی بسیاری از پروژه‌های نرم‌افزاری است. شاید بتوان گفت حتی با پیشرفت تولید نرم‌افزارها و ایجاد ابزارهایی برای تولید نرم‌افزار از سطح تجرید بالاتر، در آینده نیز همچنان کدی بوسیله برنامه‌نویس نوشته شده‌است و توسط ماشین خوانده و اجرا شود به عنوان خروجی اصلی کار مهندسان نرم‌افزار باقی خواهد ماند. از این روست که توجه به کیفیت این خروجی اهمیت زیادی برای مهندسان دارد.

کدی که با اصول درستی برنامه‌نویسی نوشته نشده باشد، کد کثیف خوانده می‌شود.

نمودار زیر یک رابطه بین بهره‌وری تیم تولید و زمان برای یک کد کثیف را نشان می‌دهد. منظور از بهره‌وری میزان هزینه و زمانی است که باید برای افزودن ویژگی‌های جدید به برنامه صرف شود.



همانطور که مشاهده می‌کند با گذشت زمان برای یک کد کثیف[1] بهره‌وری به شدت پایین می‌آید. زیرا کد کثیف ماهیت مسری دارد. یعنی آنکه هنگام افزودن کد جدید به کد کثیف، احتمال اینکه کد جدید نیز کثیف نوشته شود خیلی بیشتر از آن است که به صورتکد تمیز نوشته شود. به عبارت دیگر می‌توان گفت که میزان کثیف بودن یک کد به شکل نمایی افزایش پیدا می‌کند. و از این رو بهره‌وری تیم به شکل نمایی کاهش پیدا می‌کند و به مرور به صفر میل می‌کند.

این مستند تلاش می‌کند یک مقدمه در خصوص کیفیت کد و روش‌های بهبود آن ارایه دهد.

[1] در این مستند اصطلاح کد کثیف(Messy Code) در مقابل کد تمیز(Clean Code) برای خروجی برنامه‌نویسی بد در مقابل برنامه‌نویسی خوب استفاده شده است.

داستان کد کثیف

قطعن کسی دوست ندارد کد کثیف بنویسد. ولی چرا بسیاری از پروژه‌ها به دلیل کد کثیف شکست می‌خورند؟ چرا بسیاری از نرم‌افزارها قبل از آنکه به اندازه سرمایه‌گذاری انجام شده بر روی آن‌ها استفاده شوند، دور ریخته و دوباره نوشته می‌شوند؟ چرا بسیاری سازمان‌های قادر به تغییرات حتی کوچک بر روی کدهای خود نیستند و حاضرند که مبالغ بسیاری از ضرر کنند ولی تن به تغییر کدهای موجود خود ندهند؟

نمی‌توان تنها فرض کرد که دانش کم برنامه‌نویسان نسبت به کد و طراحی بد آنها است که چنین وضعی را پیش‌آورده است. (هرچند که این دلیل سهم زیادی در کیفیت کم نرم‌افزارها دارد!!) در ادامه داستان چرخه حیات یک پروژه نمونه را پی‌می‌گیریم تا کمی دقیق‌تر این مشکلات را بررسی کنیم.

روزی روزگاری در یک سازمان تیمی به نام (الف) برای انجام یک پروژه نرم‌افزاری تشکیل شد. این تیم پس از ارایه نسخه اولیه با استقبال بخش‌های مختلف سازمان (یا مشتریان) مواجه شد. حالا تیم در فشار شدید کاربران برای افزودن ویژگی‌ها و امکانات جدید و پشتیبانی از ارتباطات جدید قرار داشت. مدیران تصمیم گرفتند که با وضع زمان‌بندی‌های سخت و افزایش هزینه‌های در کمترین زمان ممکن تغییرات زیادی در کد داده شود.

شاید بتوان این داستان را به گونه‌ای دیگر برای تیم (ب) تعریف کرد. پس از ارایه نسخه اولیه که با تاخیر همراه بود، تیم (ب) متوجه شد که به دلیل تحلیل نادرست و فرایند اشتباه تولید نرم‌افزار، محصول تولید شده منطبق با نیازمندی‌های کاربران نیست. پروژه به دلیل تاخیرهای مکرر کاملا از چهارچوب هزینه و زمان پیش‌بینی شده خارج شده است. حالا پروژه خطر شکست کامل روبروست و نیاز به یک اقدام ضربتی برای خروج از بحران شدیدا احساس می‌شود.

یک روایت دیگر از داستان برای تیم (ج) وجود دارد. این تیم برای انجام یک پروژه تشکیل شد که در ابتدا درک درستی از ابعاد و زمان‌بندی پروژه وجود نداشت. به همین دلیل زمان‌بندی بسیار کمتری برای پروژه در نظر گرفته شد. حالا نزدیک سررسید[1] انجام پروژه است و تیم با یک وضعیت بحرانی روبروست.

[1] Deadline

از اینجا قصه تیم (الف) ، (ب) و (ج) به یک شکل ادامه پیدا می‌کند. یک جلسه بحران تشکیل می‌شود و همه اعضای تیم متعهد می‌شوند که تمام تلاش خود را برای درست شدن اوضاع بکنند. با تلاش شبانه‌روزی تیم سعی می‌کند که از وضعیت بحرانی خارج شود و همه چیز را به حالت عادی برگرداند. این اتفاق می‌افتد و حالا تیم به همه سررسیدهای خود رسیده است و مدیران بسیار راضی هستند.

شب سررسید که مدیر تیم دیروقت خسته به سمت خانه خود می‌رود، متوجه یک اتفاق عجیب می‌شود. او با خود فکر می‌کند که به طرز عجیبی در روزهای منتهی به سررسید بهره‌وری تیم به شدت افزایش پیدا کرده بود و اعضای تیم کدهای محول شده به خودشان را بسیار سریع‌تر از چند وقت پیش تحویل می‌دادند. مدیر تیم با خود فکر کرد که حتما احساس مسئولیت بیشتر اعضای تیمش باعث افزایش بهره‌وری آنها شده است.

بعد از مدتی (مثلا چند ماه یا یک سال) کم‌کم همه توجه بخش‌هایی از کد شدند که به طرز کثیفی نوشته‌شده بود. کم‌کم بهره‌وری تیم در حال کاهش بود. وقتی برنامه‌نویسان برای دیرکرد سررسیدهای خود مورد مواخذه قرار می‌گرفتند، زمان زیاد یکپارچه‌سازی با کد فعلی یا فهمیدن کد موجود و بازبکارگیری[1] آن را به عنوان دلیل ارایه می‌کردند.

حالا مدیران برای کاهش بهره‌وری افراد را مقصر می‌دانند و سعی می‌کنند با جابجایی افراد یا استخدام نیروهای جدید وضعیت را بهتر کنند. افراد جدیدی که به پروژه اضافه می‌شوند به دلیل کد کثیفی که می‌بینند نمی‌توانند دید درستی از طراحی کد به دست آورند و با کدهایی جدیدی که با چشمان بسته می‌نویسند، موجب کثیف‌تر شدن کد می‌شوند.

---
[1] Reuse

سرانجام یک شورش در سازمان اتفاق می‌افتد. بلاخره اعضای تیم از این وضعیت به تنگ آمده و مدیران را مجاب می‌کنند که با تشکیل یک تیم جدید و استفاده از فناوری‌های جدید می‌توان طرحی نو در انداخت و تمامی این مشکلات را به راحتی در زمان کمی حل کرد. سازمان بلاخره راضی می‌شود که یک تیم حرفه‌ای[1] برای این کار تشکیل دهد.

همه دوست دارند که در این تیم‌باشند زیرا آنجا می‌توانند با فناوری‌های جدید کار کنند و آزادی عمل بیشتری داشته باشند اما عموما این تیم‌ها با اعضای جدید که فناوری‌های جدید را می‌شناسند و ولی با منطق کسب‌وکار سازمان بیگانه هستند تشکیل می‌شود. اعضای پروژه قدیمی کنار گذاشته می‌شوند و از این رو حس بعدی نسبت به اعضای تیم جدید پیدا می‌کنند.

حالا هر دو تیم در حال رقابت هستند. تیم جدید باید نرم‌افزاری بسازد که نه‌تنها همه کارهای نرم‌افزار قدیمی را انجام می‌دهد و امکانات جدیدتری را نیز خواهد داشت بلکه همه درخواست‌های تغییر[2] که در حال حاضر بر روی سیستم قدیمی در حال اجرا است، باید در سیستم جدید حین ساخته‌شدن اعمال شود.

مدیریت اعلام کرده است که تنها زمانی که سیستم جدید کامل شود و بتواند همه کارهای مربوط به سیستم قبلی را به درستی انجام دهد اجازه به‌کاراندازی[3] آن را خواهد داد.

زمان تولید نرم‌افزار جدید بسیار طول می‌کشد و کم‌کم نرم‌افزار جدید به یک سیستم زیبا ولی ناکارآمد در مقابل نرم‌افزار قدیمی زشت ولی کارا در سازمان شناخته می‌شود مجبور کردن کاربران به استفاده از آن سخت و سخت‌تر می‌شود. کم‌کم اعمال تغییرات روی سیستم قدیمی به امید آنکه سیستم جدید عنقریب با تمامی ویژگی‌های جدید آماده خواهد شد متوقف می‌شود و همه منتظر سیستم جدید می‌مانند.

اما سیستم جدید با مشکلات زیادی برای پیاده سازی روبروست و نمی‌تواند سررسیدهای معین شده را محقق کند. بلاخره این بار مدیران شورش می‌کنند و تیم جدید را مجبور می‌کنند که تا سررسید مشخصی نرم‌افزار جدید را به کاربیندازند وگرنه پروژه متوقف خواهد شد.

بار دیگر جلسه بحران این بار در تیم جدید تشکیل می‌شود. برای ادامه داستان این بخش را دوباره از اول این بار برای تیم‌جدید بخوانید!!

---

[1] Tiger Team

[2] Change Request – اصطلاحن به درخواست‌هایی توسط کاربران برای تغییر بر روی سیستم نرم‌افزاری داده می‌شود، می‌گویند. در فاز نگهداری نرم‌افزار تغییرات از طریق درخواست‌های تغییر و پس از طی یک فرایند مدون تایید و برنامه‌ریزی بر روی سیستم اعمال می‌شوند.

[3] Deployment

ریشه کد کثیف

شاید بتوان مهم‌ترین علت ایجاد کد کثیف توسط برنامه‌نویسان باسواد حرفه‌ای[1] را، بودن تحت فشار سررسیدها دانست. هنگام نزدیک شدن به سررسیدهای برنامه‌نویسان به صورت ناخوداگاه بخشی ای فیلترهای ذهنی خود در خصوص کیفیت برنامه‌نویسی را خاموش می‌کنند تا بتوانند با آزادی عمل بیشتری به سررسید مورد نظر برسند.

این جمله "فعلن بنویسمش بعدن درستش می‌کنم" نشانه این وضعیت است. بعدنی که هیچ‌گاه فرانمی‌رسد. مسئله فقط فراموشی یا نبود اراده برای بازگشت به گذشته و تغییر کد نیست بلکه از انجایی که برنامه‌نویسی یک کار پیچیده ذهنی است، برگشت به وضعیت ذهنی گذشته برای فهمیدن درست یا غلط بودن تصمیمات نیز برخی از اوقات بسیار سخت می‌شود.

پذیرفتن فشار سرسید به عنوان ریشه کد کثیف نمی‌تواند توجیهی برای سلب مسئولیت برنامه‌نویسان از کار در برنامه و چهارچوب‌های زمان و هزینه مشخص باشد. صد البته که قبول مدیریت‌پذیری و برنامه‌ریزی به عنوان یک اصل برای پیش‌برد هر کار مهندسی یک اصل غیرقابل خدشه است. ولی مسئله مطرح شده در خصوص تاثیرات بد زمانبندی تنها تاکیدی بر سختی این کار و دقت مورد نیاز برای آن را در خصوص یک پروژه نرم‌افزاری است.

بازآرایی چیست؟

بازآرایی(Refactoring) یک روش مدون برای بهبود کیفیت کد موجود بدون تغییر در واسطهای بیرونی و تاثیر در استفاده‌های آن است. به عبارت دیگر بازنوسی یک فرایند تعریف شده برای تبدیل کد کثیف[2] به کد تمیز[3] است.

یک تعریف دیگر از بازآرایی بهبود طراحی بعد از نوشته‌شدن کد است. شاید این تعریف کمی عجیب به نظر برسد زیرا که عموما کدنویسی بعد از طراحی انجام می‌شود. مسئله این است که یک کدنویسی درست بعد از یک طراحی خوب اتفاق می‌افتد و پس از مدتی که کد تغییر می‌کند، یکپارچگی[4] سیستم و ساختار طراحی‌شده برای آن، کم‌کم کمرنگ و کمرنگ‌تر شده و پس از مدتی محو می‌شود. در این هنگام افزودن کد به چنین برنامه‌ای به جای یک فرایند مهندسی بیشتر به یک کار وصله‌پینه کردن با چشمان بسته تبدیل می‌شود.

[1] برای برنامه‌نویسان مبتدی و بی‌سواد همچنان نداشتن دانش در خصوص کیفیت کد و طراحی، دلیل اصلی کد بد است.

[2] Messy Code
[3] Clean Code
[4] Integrity

فرایند بازآرایی کد برای بهبود طراحی و حاکم‌کردن دوباره یک تفکر منسجم طراحی شده به آن است. این کار از طریق تغییرات عموما کوچکی بر روی بخش‌های مختلف کد اتفاق می‌افتد که تاثیر جمعی این تغییرات کوچک به ایجاد کدی با کیفیت بیشتر می‌انجامد.

بخش‌هایی از کد که بهتر است مورد بازآرایی قرارگیرند، را می‌توان از نشانه‌ایی از دیگر کدها بازشناخت. در کتاب بازآرایی مارتین فاولر به چنین نشانه‌هایی بوی بد[۱] گفته می‌شود که در این مستند این اصطلاح با اصطلاح خشت خام[۲] جایگزین شده است.

آزمون واحد

گام اول برای بازآرایی کد نوشتن آزمون‌های واحد[۳] مناسب برای کد است. بازآرایی کد مانند هر تغییر دیگری در آن محتمل است که مشکلات و خطاهای جدیدی را در کد موجب شود. نوشتن آزمون این امکان را می‌دهد که پس از تغییرات از عدم ایجاد خطا[۴] مطمئن شویم.

باید توجه داشت که هدف بازآرایی تغییر ساختار و طراحی اجزای برنامه بدون تغییر در واسطهای بیرونی آن‌هاست. آزمون واحد دقیقا وظیفه اطمینان از صحت رفتار بیرونی اجزا را بر عهده دارد. هر چند که اگر کد با فرایند درستی نوشته و نگهداری شده باشد قطعا موارد آزمون[۵] کامل و دقیقی دارد که بایستی با هر تغییر کد دوباره اجرا و ارزیابی شوند.

[۱] Bad Smell – مبنای این اصطلاح گفته مادربزرگ کنت بک در خصوص کهنه بچه است: وقتی بوی بد شنیدی باید بچه را عوض کنی. منظور نشانه‌ای از یک اتفاق بد است.

[۲] خشتی که خوب پخته نشده باشد، می‌تواند با یک بارندگی از بین برود و چه بسا ساختمانی را فرو بریزد. خشت خام می‌تواند در یک بنای بزرگ به عنوان یک نقطه ضعف بالقوه باقی بماند و چه بسا زمانی موجب فروریختن بنا شود. از این رو شاید بتوان گفت که خشت خام معنای دقیق‌تری را از اصلاحی که فاولر در کتابش به کار برده است، در خود دارد. جایگزین کردن قطعه‌کدهایی با طراحی بد در یک نرم‌افزار بسیار شبیه جایگزین کردن خشت‌خام در یک ساختمان است.
یک خشت خام در یک بنای بزرگ می‌تواند زمینه‌ساز وجود آجر سِنِمّار باشد. سِنِمّار معمار ایرانی زمان ساسانیان بود که قصر بسیار باشکوهی برای نعمان شاه حیره ساخت و چون بنا به اتمام رسید شاه از معمار پرسید آیا نقطه ضعفی در این بنا می‌شناسی؟ سِنِمّار پاسخ داد که آجری در بنا می‌شناسد که اگر آن را بردارد، کل قصر فرو می‌ریزد. پس پادشاه دستور داد، که او را از بالای قصر به زیر بیاندازند. در برخی روایت‌ها چون بنا به اتمام رسید، نعمان چنان در شکوه بنای سِنِمّار متحیر شد که از بیم آنکه سِنِمّار بنای شبیه به آن را برای دیگری بسازد، دستور قتل او را داد .منظور از آجر سنمار نقطه ضعفی کوچک در ساختمانی بزرگ است که می‌تواند ویرانی‌های بزرگ به بار آورد. بهرام بیضایی نمایشنامه زیبایی به نام "مجلس قربانی سِنِمّار" در خصوص این مهندس هنرمند ایرانی نوشته است.

[3] Unit Test
[4] Bug
[5] TestCase

آین آزمون‌های بایستی طی فرایند بازآرایی به صورت مداوم و پس از هر تغییری دوباره اجرا شوند. یکی از مهم‌ترین مزایای آزمون خودکار، تکرارپذیری آن است پس اگر آن را یکبار و پس از انجام همه تغییرات اجرا کنید از مزیت اصلی آن بهره نبرده‌اید.

مزایای بازآرایی

فهرست زیر مزایای بازآرایی کد را احصا کرده است:

۱. بازآرایی طراحی را بهبود می‌بخشد.
کیفیت طراحی یک برنامه با افزوده شدن امکانات جدید و اعمال تغییرات همیشه رو به افول است و بازآرایی تلاشی است در جهت بارگرداندن تفکر طراحی به برنامه

۲. بازآرایی برنامه را قابل فهم‌تر می‌کند.
کامپیوترهایی که برنامه‌ها را اجرا می‌کنند، تنها استفاده‌کنندگان کدها نیستند. بلکه برنامه‌نویسانی که در آینده نیز باید روی کد کار کنند و آن را بفهمند، نیز استفاده‌کننده کد محسوب می‌شوند و باید مد نظر قرار گیرند. هیچ برنامه‌نویسی جزئیات کدی را که مدتی از نوشتن آن گذشته است، به خاطر ندارد. لذا خود نویسنده نیز بعد از مدتی از این دست استفاده‌کنندگان کد خواهد بود خصوصن اینکه هیچ برنامه‌نویسی پس از زمان کوتاهی جزئیات برنامه‌ای را که نوشته‌است، را به خاطر ندارد. بازآرایی کمک می‌کنند تا کد از شکل یک معمای غیرقابل فهم به صورت یک نثر شیوا برای توصیف هدف نویسنده‌اش در آید.

۳. بازآرایی به کشف خطاها کمک می‌کند.
خود فرایند بازآرایی از آن رو که مستلزم فهم کامل کد موجود است، گونه‌ای آزمون ایستای نرم‌افزار[۱] یا همان بازبینی کد[۲] به شمار می‌آید.
از طرف دیگر بازآرایی موجب می‌شود، هدف نویسنده کد را حد ممکن به شکل شفافی در کد بیان شود و این مسئله به یافتن خطاها کمک می‌کند. خطاها بسیاری از اوقات جایی اتفاق می‌افتد که هدف برنامه‌نویس به روشنی بیان نشده است و این ابهام، زمینه استفاده نابجا و یا تلقی اشتباه از بخش‌های مختلف کد را ایجاد می‌کند.
توصیف بیارنه استروستروپ از کد تمیز دقیقا حول مفهوم بیان شده است. او می‌گوید کد تمیز کدی است که پنهان شدن را برای خطاها سخت کند.

[1] Software Static Test
[2] Code Review

۴. بازآرایی برنامه‌نویسی را سریع‌تر می‌کند.

شاید در ابتها کمی عجیب به نظر برسد ولی بازآرایی، توسعه برنامه را سریع‌تر می‌کند. شاید توسعه کد کثیف در ابتدا سریع‌تر به نظر برسد ولی اگر در بازه زمانی طولانی به تغییر کدی که به طرز کثیفی توسعه پیدا کرده، نگاه کنیم، زمانی به دلیل کثیف بودن کد به زمان توسعه افزوده شده است، بسیار بیشتر از زمانی است که می‌توانست به مرور برای تمیز نگهداشتن کد صرف شود.

زیرا می‌توان گفت که زمان مورد نیاز برای بازآرایی همیشه نسبت تقریبا خطی خود را با حجم کد و تغییرات اعمال شده بر روی آن حفظ می‌کند ولی زمان مورد نیاز برای تغییر کد کثیف به صورت نمایی افزایش پیدا می‌کند.

از این رو تغییر یک کد بازآرایی‌شده تمیز و با طراحی درست و شفاف همواره آسان‌تر، سریع‌تر و هم‌هزینه‌تر از تغییر در یک کد کثیف با طراحی محوشده است.

کلاه جدیدی برای بازآرایی

کنت بک پیشنهاد می‌دهد که برنامه‌نویسان بین نقش بازآرایی و نقش توسعه‌گری[1] خود تمایز قایل شوند. یعنی بخشی از وقت خود را تنها به افزودن امکانات جدید و نوشتن مورد آزمون به آن امکانات اختصاص دهند و در بخش دیگری از وقت خود تنها به بازآرایی کدهای نوشته شده موجود بپردازند. این مفهوم را با گذاشتن مداوم کلاه توسعه‌گری یا بازآرایی بر سر توصیف می‌کند.

این مفهوم در واقع از کارهای ادوارد دوبونو پزشک، روانشناس و نویسنده و کتاب معروف او به نام شش کلاه برای تفکر اخذ شده است. منظور از کلاه در نوشته‌های او منظرگاه‌های مختلفی هستند که می‌توان از دریچه آنها به یک مسئله نگاه کرد. این مفهوم کمک می‌کند در حین حل مسئله بتوان با کمترین تاثیر چهارچوب‌های ذهنی از پیش‌تعیین شده، به صورت خلاقانه یک مسئله را از جنبه‌های مختلف مورد نقادی قرار داد. در واقع کلاه‌های مختلف تفکر راهی است برای کم‌تر کردن تاثیر منفی چهارچوب‌های ذهنی از پیش تعیین شده برای مواجه با یک موقعیت ناآشنا که نیاز به تفکر خلاق دارد.

در موضوع حاضر صحبت از کلاه جدید برای بازآرایی در مقابل کلاه توسعه‌گری تاکید بر جداسازی جنبه‌های مربوط به بازآرایی از فرایند معمول برنامه‌نویسی است. یعنی هنگام توسعه‌گری چهارچوب ذهنی طوری تنظیم نمود که به انجام درست امکان جدید تمرکز کرد و در وقت بازآرایی کیفیت کل طراحی برنامه هدف انجام دهنده است.

[1] Add features

این مسئله به این معنی نیست که در هنگام افزودن امکانات جدید باید از اصول حاکم بر کیفیت کد که در بازآرایی مد نظر است غافل شد، بلکه بیشتر تاکید بر این نکته است که گاهی از اوقات خصوصن زمانی که نشانه‌هایی از خشت‌های خام در کد قابل مشاهده است، کلاه توسعه‌گری را به کناری گذاشت و با کلاه بازآرایی بدون دغدغه اینکه کار بازآرایی از نظر کاربر یا مدیر، خروجی قابل لمسی ندارد به تعویض خشت‌های خام با خشت‌های پخته پرداخت.

یک برنامه‌نویس حین کار خود به صورت مداوم کلاهای مختلفی را بر سر می‌گذارد و از منظرگاه‌های مختلف نتیجه کار خود را ارزیابی می‌کند. به گفته کنت بک همواره برنامه‌ها از دو منظر ارزش‌گذاری می‌شوند. ارزشی که برنامه در زمان حال دارد و ارزشی که برنامه برای آینده دارد. کلاه توسعه‌گری برای افزایش ارزش فعلی برنامه است و کلاه بازآرایی برای بهبود ارزش آتی برنامه.

برنامه‌نویس همواره در خصوص کاری که برنامه باید بکند، مطمئن است اما کاملا نمی‌داند که فردا باید برنامه چه کار جدیدی را بر عهده بگیرد. ولی اگر تنها به فکر تغییر امروز باشد، قطعن برای تغییرات فردا دچار مشکل خواهد شد. بازآرایی اصلاح تصمیمات درست دیروز است که امروز اشتباه به نظر می‌رسد، برای کمک در جهت کارهایی که فردا باید انجام شود. کلاه‌ی توسعه‌گری برای ارزیابی ارزش فعلی و کلاه بازآرایی برای ارزیابی ارزش آتی برنامه است.

خشت‌های خام

تشخیص درست خشت‌های خام نیاز به تجربه و دانش دارد. دانش از آن رو که چنین مشکلاتی در برنامه‌های نرم‌افزاری در منابع مختلف به شکل مدونی جمع‌آوری و تحلیل شده‌اند و دلایل مشکلات محتملی که ممکن است ایجاد کنند، بر اساس اصول و قواعد مشخص تبیین شده‌اند.

برای تشخیص خشت‌های خام، تجربه از آن رو نقش کلیدی دارد که چنین تصمیماتی مثل تمامی تصمیمات طراحی بسیار وابسته به فضای مسئله و زمینه‌ای[1] است که طراحی در آن مطرح شده است. یعنی درست یا غلط بودن یک طراحی در بسیاری از مواقع به زمینه و فضای مسئله‌ای که طراحی برای راه‌حل آن ارایه شده است ارتباط دارد و احکام کلی برای درستی یا نادرستی یک قطعه کد اکثر اوقات وجود ندارد.

[1] Context

مسئله مهم دیگر ارتباط بین اصول حاکم بر شناسایی خشت‌های خام است. برخی از اوقات فلسفه و اصولی یک خشت بر اساس آن خام ارزیابی می‌شود، با برخی دیگر از اصول شناسایی خشت‌های خام در تبایل و گاه در تضاد است. به عنوان مثال تلاش برای انعطاف‌پذیری کد برخی از اوقات کار پسندیده و گاهی کاری ناپسند تلقی شده است. و باز هم این زمینه و فضای مسئله است که مشخص می‌کند در هر موقعیت، کدام یک از اصول باید بیشتر مورد توجه قرار گیرد.

کد تکراری[۱]

تکرار را شاید بتوان ام‌الخبائث دنیای نرم‌افزار دانست[۲]. مشکلی که ریشه بسیاری از مشکلات دیگر است. بسیاری از متدها و روش‌های مهندسی نرم‌افزار برای از بین بردن این مشکل ابداع شده‌اند. به عنوان مثال مفهوم بهنجاری[۳] در دادگان‌ها[۴] که توسط ادگار کاد[۵] توسعه پیدا کرده است، حول مفهوم افزودنگی یا تکرار داده است. یا رویکرد شی‌گرایی با محوریت بازبکارگیری[۶] برای فرار از تکرار توسعه پیدا کرده است.

اگر در برنامه ساختار یا رفتاری مشاهده کردید که در چندین جای مختلف تکرار شده‌است، مطمئن باشید که اگر بتوان راهی برای یکی کردن این تکرارها پیدا کرد، کیفیت برنامه بهتر خواهد شد.

برخی نویسندگان به عنوان یک حساب سرانگشتی قاعده‌ای را بیان می‌کنند که تکرار تا دوبار می‌تواند طبیعی باشد ولی بار سوم بایستی فکری به حال جایگزینی تکرار با بازبه‌کارگیری کرد. از آنجایی که طراحی کاری شدیدا وابسته به فضای مسئله است بهتر است که همیشه زمینه و فضای مسئله مد نظر قرار گیرد. شاید برای یک مسئله حتی تحمل دوبار تکرار نیز منطقی نباشد و برای مسئله‌ای دیگر بتوان پذیرفت که بخشی از کد بیش از سه‌بار تکرار شود.

ساده‌ترین شکل تکرار، تکراری بخشی از کد در دو یا چند متد یک کلاس است. برای حل چنین مشکلی کافیست که با استخراج متد، کد تکراری در قالب یک متد جدید دو یا چند بار فراخوانی کنید. گونه دیگر تکرار، تکرار بخشی از کد در دو کلاس است که در یک سلسله مراتب پدر–فرزندی در یک سطح قرار دارند. استخراج متد و سپس بالاکشیدن به کلاس پدر می‌تواند گزینه مناسبی باشد.

[1] Duplicate Codes
[2] "Duplication may be the root of all evil in software" Robert Cecil Martin
[3] Normal Form
[4] Database
[5] Edgar Frank Codd
[6] Reusability

در صورتی که دو کلاسی که متدهای تکراری دارد در رابطه پدر–فرزندی شریک نباشند، گزینه های زیر می‌تواند مطرح باشد:

- استخراج کلاس جدید از متدها و کدهای تکراری و فراخوانی در هر دو کلاس
- فراخوانی یکی از کلاس‌های شامل کد مشترک توسط کلاس دیگر
- بازبینی مجدد جنبه مشترکی که موجب ایجاد کدتکراری در دو کلاس شده است و ایجاد پدر مشترک برای دو کلاس در صورت نیاز

متد طولانی[1]

در نرم‌افزار اصل کوچک زیباست[2]، همیشه مطلوب بوده است. اجزای کوچک و طبعن ساده‌ای که به خوبی در کنار هم قرار گرفته‌اند همیشه نسبت به اجزای بزرگی و پیچیده‌ای که کارهای متنوع و زیادی را بر عهده دارند، ترجیح دارند.

در حال حاضر تقریبا همه زبان‌های مدرن سرباره ناشی از توالی فراخوانی متدهای مختلف به حد قابل چشم‌پوشی کم کرده‌اند. از این رو یک متد بلند از دیدگاه اجرا هیچ مزیتی نسبت به چندین متد کوتاه که هم‌دیگر را فراخوانی می‌کنند، ندارد.

از این رو خوانایی کد تنها معیار مقایسه بین متدهای بلند و کوتاه است. شکستن یک متد بلند به متدهای کوتاه و اسم گذاری دقیق و خوب برای آن‌ها در واقع رمزگشایی از معمای فهم ساختار و هدف متد و تبدیل آن به یک متن خوانا از اجزایی وظیفه‌مندی مورد نیاز برای اجرای برنامه است.

می‌توان گفت که باید متدهای به حتی کوتاه باشند و اسم‌های مناسب برای آن‌ها انتخاب شده‌باشد که خواننده بدون نیاز به خواندن بدنه متد با خواندن امضای آن[3] هرچیزی که لازم است راجع به آن بداند را متوجه شود.

برخی از اوقات شکستن متد به متدهای کوچک‌تر حتی می‌تواند جایگزین توضیحاتی[4] شود که در خلال کد نوشته شده است. کافیست بخشی از کد که نیاز به توضیح دارد در قالب یک متد کوچک با نامی با مسما که توصیف درستی از کار متد ارایه می‌دهد، جایگزین شود.

[1] Long Method
[2] Small is beautiful
[3] Method Signature(منظور خط اول تعریف متد شامل نام، اعلان پارامترهای ورودی، نوع داده بازگشتی و برخی دیگر از پیکربندی‌های تعریف متد است)
[4] Comments

حتی ابایی نداشته باشید که متدی را استخراج کنید نام مفصل آن از کدی که در درون بدنه آن وجود دارد، طول بیش‌تری داشته باشد. مسئله اصلی معنایی است که نام متد برای توصیف هدف آن متد در خود دارد، نه صرفن کوتاهی کد.

وجود توضیحات در برنامه نشانه خوبی برای جایی است که می‌توان با شکستن کد به اجزای کوچک‌تر و استفاده از اسامی با معنا کد را تمیزتر کرد. در واقع توضیحات درون کد فاصله معنایی بین هدف نویسنده و خوانایی کد را پر می‌کند. کاری که متدهای کوتاهی که برای انجام وظایف کوچک ولی واضح نوشته شده‌اند، به طرز بهتری انجام می‌دهند.

قاعده سر انگشتی وجود دارد که بایستی تلاش کرد بدنه هر متد بیشتر از چند خط نباشد.

به یاد داشته باشید که متدها باید همواره یک کار انجام دهند و باید آن کار را درست انجام دهند و باید تنها همان کار را انجام دهند.

کلاس بزرگ

کلاس‌های بزرگ با همان مبنایی که در خصوص متدهای طولانی گفته شد، بایستی به کلاس‌های کوچک‌تر افزار شوند. معمولا چنین کلاس‌های متغیرهای داده‌ای[۱] زیادی دارند.

اگر کمی دقیق‌تر به متغیرهای چنین کلاس‌های نظر مورد بررسی قرارگیرند، بعید نیست که ارتباط معنایی بین دسته‌هایی از آن‌ها وجود داشته باشد که نشان‌دهنده آن است که شاید بتوان هر دسته‌ای را در قالب یک کلاس جدید سازمان داد. در این صورت به تبع جاینمایی متغیرهای داده‌ای در کلاس‌های جدید، متدهای کلاس بزرگ بین کلاس‌های کوچکی از شکستن آن ایجاد شده‌اند قابل تقسیم هستند.

کلاس‌های بزرگ به جایی زمینه بدی برای تکرار کردن کد هستند. زیرا کدهایی که با متغیرهای داده‌ای آن‌ها کار می‌کنند به دلیل آن که در متدهای مختلف کلاس بزرگ پخش‌شده‌اند امکان بازبکارگیری[۲] زیادی ندارند.

برخی از اوقات کلاس‌های بزرگ به شکل کلاس‌های قادرمطلق[۳] در می‌آیند که وظایف زیاد و متنوعی را بر عهده می‌گیرند. کلاس‌های قادر مطلق از پادالگوهای[۴] معروف در حوزه شی‌گرایی است که باید از آن اجتناب کرد.

---

[1] Instance Variable(Fields)
[2] Reusability
[3] Omni potent Classes
[4] Anti-Pattern

همواره به یاد داشته باشید هر چیزی که در یک داخل کلاس (الف) است باید به مفهومی که کلاس (الف) حول آن تعریف شده ، ربط مستقیم داشته باشد و هر چیزی که در فضای حل مسئله به این مفهوم ربط مستقیم دارد، بایستی در کلاس (الف) باشد.

فهرست بلند پارامترها[1]

زبان‌های شی‌گرا متغیرهای سراسری[2] که در نسل‌های قدیمی‌تر زبان‌های برنامه‌نویسی ریشه مشکلات زیادی بودند، پشتیبانی نمی‌کنند . در عوض با امکان بسته‌بندی داده‌ها در قالب اشیا امکان خوبی برای ارسال داده برای فراخوانی متدها ایجاد کرده‌اند.

فهرست بلند پارامترهای مورد نیاز یک متد می‌تواند با اشیایی که آن داده‌ها را به خوبی بسته‌بندی کرده‌اند جایگزین شود.

مسئله دیگر در فراخوانی متدها اجتناب از فراخوانی تودرتوی متدهاست. برخی از برنامه نویسان عادت دارند که خروجی یک متد را بدون آنکه به عنوان یک متغیر مشخص نام گذاری کنند به محض فراخوانی به عنوان ارگمان یک متد دیگر ارسال کنند.

به مثال زیر توجه کنید. متد محاسبه حقوق، متد محاسبه ساعات کاری را به به شکل تودرتو فراخوانی کرده است.

```
double salary = calculateSalary(employee, contract, calculateWorkingHours(employee, attendanceList));
```

با استفاده از امکان استخراج متغیر می‌توان کد بسیار خواناتری نوشت:

```
double workingHoursForThisEmployee = calculateWorkingHours(employee, attendanceList);
double salary = calculateSalary(employee, contract, workingHoursForThisEmployee);
```

[1] Long Parameter List
[2] Global Variables

تغییر واگرا[1]

برخی از اوقات در با کلاس‌هایی مواجه می‌شوید که به دلایلی متنوع ممکن است که تغییر کند. مثلا ممکن است در بازبینی یک کلاس به خود بگویید که اگر قرار باشد نوع ذخیره‌سازی جدید استفاده کنم، باید این سه متد را تغییر دهم و اگر نحوه نمایش تغییر کنم این دو متد این کلاس باید تغییر کند. دلایلی که برای تغییر این کلاس وجود دارد، ارتباط نزدیکی با یکدیگر ندارند و این شاید به این معنی باشد که بایستی این کلاس به دو کلاس منسجم‌تر[2] شکسته شود.

به عنوان یک قاعده کلی باید همواره دلایلی که ممکن است یک کلاس نیاز به تغییر داشته باشد، بسیار بهم شبیه باشند. مفهوم این خشت خام با اصل یکتایی وظیفه[3] رابطه نزدیکی دارد و به عبارت دیگر نقض اصل یکتایی وظیفه کلاس منجر به ایجاد خشت خام تغییر واگرا می‌شود.

جراحی ساچمه‌ها[4]

برای جراحی کسی که مورد اصابت تفنگ ساچمه‌ای قرار گرفته است، جراح باید تعداد زیادی قطعات کوچک ساچمه را از نواحی مختلف بدن او خارج کند. این دقیقا شبیه کاری است که یک برنامه‌نویس ممکن است برای اعمال یک تغییر، تغییرات کوچکی در جاهای مختلف کند به تعداد زیاد انجام دهد.

به عنوان مثال تصور کنید که قطعه کد زیر برای رویدادنگاری[5] استثناها در بخش‌های مختلف کد استفاده شده است:

```
}catch (Exception e){
    System.out.println("Type:"+ Log.BRIEF+"Error:"+e.getMessage());
}
```

حال اگر تصمیم بگیرید که از این پس اطلاعات دیگری نظیر نام‌کاربری کاربر جاری و یا زمان وقوع خطا را به خروجی که چاپ می‌شود اضافه نمایید، مجبور خواهید بود که در کلاس‌های زیادی(هر جا که رویدادنگاری انجام می‌شد) تغییرات کوچکی را تکرار نمایید.

[1] Diverge Change
[2] Coherent
[3] Single Responsibility Principle(SRP)
[4] Shotgun Surgery
[5] Logging

خشت جراحی ساچمه‌های در واقع مفهوم مقابل تغییر واگرا است. در تغییر واگرا تنوع دلیل برای تغییر یک کلاس واحد نکوهش شده است، در جراحی ساچمه‌ها تغییرهای متعدد در کلاس‌های گوناگون برای یک دلیل واحد مورد توجه قرار گرفته است. حالت ایده‌آل آن است که همیشه بین دلایل تغییر و کلاسی که تغییر در آن انجا می‌شود، رابطه یک‌به‌یک وجود داشته باشد.

راه حل برطرف کردن این خشت خام، جایگزینی کدتکراری در کلاس‌های مختلف با یک کلاس یا متد واحد است که جاهای مختلف فراخوانی شده است. رویکرد برنامه‌نویسی جنبه‌گرا[1] با تاکید بر حل این مشکل ابداع شده است. در این رویکرد جنبه‌هایی تکراری از نرم‌افزار که در کل نرم‌افزار پراکنده شده‌اند، به صورتی روشمند مدیریت می‌شوند.

چشم داشتن به امکانات دیگران[2]

اگر بخشی از یک کلاس برای انجام کارهای خود نیاز به اطلاعاتی از دیگر کلاس‌ها داشته باشد، به نظر می‌رسد که یک جای کار ایراد دارد. متد زیر را در کلاسی در غیر از کلاس Employee در نظر بگیرید:

```
public double calculateSalary(Employee employee){
```

این متد وظیفه محاسبه حقوق یک کارمند را دارد و در درون بدنه خود به اطلاعاتی که در کلاس کارمند است، احتیاج دارد.

چنین متدی منطقن نباید جایی جز خود کلاس کارمند باشد. اکثر اوقات این خشت خام به صورت چشم داشتن به داده دیگر کلاس‌ها دیده می‌شود. متدی که برای انجام وظیفه خود نیازمند به داده‌های زیادی از کلاس دیگری دارد، باید نزدیک به همان داده‌ها و در کلاس مربوط به آن‌ها باشد.

گاهی از اوقات بخش‌هایی از یک متد است که چشم به دیگر کلاس‌ها دارد.در این صورت چنین بخشی باید در ابتدا به صورت یک متد جدید استخراج شود و سپس به کلاسی که به آن ارتباط بیشتری دارد، منتقل شود.

در شی‌گرایی مفهوم شی با در کنار هم دیگر و در یکجا بودن داده و رفتار معنا پیدا می‌کند و این دو باید در کنار یکدیگر تغییر کند. در صورتی که تغییر رفتار در جایی غیر از تغییر داده مربوط به آن اتفاق بیفتد، بایستی رفتار به جایی نزدیک به داده منتقل شود. این خشت خام با خشت تغییر واگرا ارتباط معنایی نزدیکی دارد.

[1] Aspect Oriented Programming
[2] Feature Envy

خوشه‌های داده[1]

در یک برنامه معمولا داده‌های هم‌جنس در کنار هم ظاهر می‌شوند. از این رو است که می‌توان گفت داده‌ها تمایل به خوشه‌بستن در کنار داده‌هایی مرتبط با خود دارند. آرگومان‌های ورودی یک متد یا قطعه کدی که بخشی از داده‌ها کنار یکدیگر تغییر می‌کنند و یا داده‌هایی که همیشه همزمان تغییر می‌کنند، نشان دهنده خوشه‌هایی از داده است.

چنین خوشه‌هایی باید در قالب کلاس‌های جدید سامان‌دهی شوند. استخراج کلاس، جایگزینی ارگمان‌های متعدد یک متد با یک کلاس محتوی همان داده‌ها راه‌هایی برای مدیریت خوشه‌های داده است.

حالت‌هایی از خشت خام فهرست بلند پارامترها گونه‌ای خاص از این خشت خام خوشه‌های داده است.

برای مطالعه بیشتر در خصوص خشت‌های خام به منبع شماره یک مراجعه نمایید.

[1] Data Clumps

توصیه‌های مربوط به خوانایی کد

- اسامی با معنا

دنیای نرم‌افزار، دنیای کارکردن با مفاهیم، مدل‌ها، سوژه های ذهنی و تجریدها[1] است. در این دنیا زبان و ارتباط از اهمیت فوق‌العاده‌ای برخودارند. چه زبان و ارتباط میان مهندسان و کاربران نرم‌افزار و چه زبان‌های برنامه‌نویسی و رسمی برای ارتباط با ماشین‌هایی که نرم‌افزارها را اجرا می‌کنند.
از این روست که مهندسان نرم‌افزار مانند مترجمان چیره‌دست یا شاعران به واژه‌ها به عنوان امانت‌داران گنجینه ارزشمند معنا احترام بگذارند و در به کار بردن و انتخاب آن‌ها برای اسم‌گذاری و برقراری ارتباط در قالب مستندات، وسواس و دقت زیادی نشان دهند.
استفاده از اسامی که در راه را برای سوءتعبیر، ابهام، خلط معنا و یا اشتراک معنا باز می‌گذارند، خوانایی کد را بسیار کم می‌کند. اسامی بلند که به شکل دقیق، کامل و یکتایی مسمی خود را توصیف می‌کنند همیشه توصیه شده است. طولانی شدن اسم هیچ‌گاه نباید برنامه‌نویس از گنجادن معنی کامل منظور خود در آن منصرف کند.

- استفاده از اکولاد

هر چند که در جاوا استفاده از اکولاد برای بلاک‌های یک خطی اختیاری است، همواره عادت کنید که برای چنین دستوراتی نیز از اکولاد استفاده کنید. چنین کاری به IDE کمک می‌کند که دندانه‌گذاری[2] واضح‌تری از کد شما انجام دهد. همواره پس از چند خط کد نوشتن گزینه Reformat مربوط به IDE را فعال کنید[3] تا برنامه شما را مرتب نماید.

- ترجیح بیان تصریح به معانی ضمنی

همواره بیان صریح فرض‌های انجام شده در برنامه‌نویسی باید مد نظر قرار گیرد. مثلا استفاده از مقدار پیشفرض متغیرهای بدون مقداردهی اولیه موجب ابهام خوانندگان کرد خواهد شد. مثلا هرچند که هر برنامه نویسی جاوایی میداند که متغیر عددی که مقداردهی اولیه نشده باشد مقدار صفر می‌گیرد، ولی تصریف مقدار صفر برای این متغیر ترجیح دارد.

[1] Abstraction
[2] Indenting
[3] برای InteliJ IDEA از کلید میانبر Alt+Ctrl+L و برای Eclipse از Ctrl+Shift+F استفاده کنید.

بسیاری از اوقات برنامه‌نویسان فرض‌هایی برای دامنه یا مقادیر معتبر برای متغیرها، مقدار اولیه متغیرهای داده‌ای و ترتیب فراخوانی متدها می‌کنند که آن‌ها را بدیهی فرض کرده اند. کد تمیز کدی است که از چنین فرض‌هایی از طریق امکاناتی که زبان برنامه نویسی در اختیار قرار می‌دهد، محافظت شده باشد.

- بسته‌بندی درست انواع و حالت‌ها در شمردنی‌ها[1]

در خصوص انواع و حالت‌های ثابت برنامه به جای اینکه فرض‌های خود را در قالب توضیحات خلال کد بنویسید دسته‌بندی‌های بر روی انواع کوچک و حالت‌های مختلف را در قالب شمردنی‌های مدل‌سازی کنید. جاوا امکانات بسیار خوبی برای پشتیبانی از شمردنی‌های در اختیار قرار می‌دهد که از پخش‌شدن منطق مربوط به حالت‌های مختلف در جاهای مختلف برنامه و نوشتن کد تکراری جلوگیری می‌کند.

کد تمیز

کد تمیز در واقع فاصله کدی است که کار می‌کند با کد ایده‌آل. یعنی کارکردن کد لازم برای کد تمیز است نه شرط کافی. از این رو این استدلال که وقتی این کد کار می‌کند چرا باید آن را تغییر دهم[2]، قابل قبول نیست. کیفیت کد شامل بسیار از ویژگی‌هایی است که در تمامی آن‌های کار کردن کرد بدیهی فرض شده است.

می‌توان گفت که مثل معروف مهندسی "چیزی که خراب نشده‌است را تعمیر نمی‌کنند"[3]، در مهندسی نرم‌افزار چندان کاربرد ندارد. (هر چند دقیق‌تر آن است که بگوییم تعریف خرابی در مهندسی نرم‌افزار پیچیده‌تر از دیگر رشته‌های مهندسی و فراتر از مسئله کار نکردن است.)

تشخیص کد بدون خطا با کامپایلر، تشخیص کدی که کار می‌کند با کاربر و تشخیص کد تمیز با برنامه‌نویس است.

---

[1] Enumeration

[2] این استدلال بیشتر توسط برنامه‌نویسان تجربی مطرح می‌شود. از دید برنامه نویسان تجربی خود نوشته شدن برنامه هدف است و با انجام آن یک پروژه نرم‌افزاری به نتیجه رسیده است. این رویکرد در واقع فروکاستن دانش مربوط به مهندسی نرم‌افزار به مهارت برنامه‌نویسی است.

[3] If it ain't broke, don't fix it

“I'm not a great programmer; I'm just a good programmer with great habits.”

Kent Beck

منابع

1. Refactoring: Improving the Design of Existing Code, Martin Fowler , Kent Beck and others, Addison-Wesley Professional,1999

2. Clean code,A handbook of agile software craftmanship, Robert C Martin,2008, Prentice Hall

3. The Essence of Object Oriented Programming with Java and UML, Antonio Maña Gómez, University of Malaga, http://www.lcc.uma.es/~amg/ISE/OOP-Java-UML/Chapter8.html