به نام خدا

مقدمه‌ای بر طراحی– روش CRC

فهرست مطالب

تذکر مهم

این مستند به عنوان راهنمای طراحی و پیاده‌سازی برای دانشجویان درس برنامه‌سازی پیشرفته تهیه شده است و فرایند ارایه شده در آن مبتنی بر هیچ متودولوژی رسمی تولید نرم‌افزار نیست. کلیه مطالب مطرح شده در این مستند شامل ساده‌سازی در حل مورد نیاز برای درس جاری بوده است. لطفن برای جزئیات مربوط به روش‌ها و نمودارها به کتاب‌های مرجع یا درس‌های مربوط به آن‌ها مراجعه نمایید

طراحی چیست؟

طراحی بخش از فرایند ایجاد نرم‌افزار است که به تهیه یک نقشه و طرح از نرم‌افزاری که باید پیاده‌سازی شود، اختصاص دارد. ورودی فرایند طراحی توصیف مربوط به نیازمندی‌های نرم‌افزار(در این درس شرح مسئله یا پروژه) و خروجی آن توصیف اجزای نرم‌افزار(در این درس کلاس‌ها، ویژگی‌ها و متدهایشان و روابط بین آن‌هاست).

در یک جمله می‌توان گفت که توصیف نیازمندی‌های یک نرم‌افزار درباره چیستی نیاز برای آن نرم‌افزار و توصیف طراحی نرم‌افزار، چگونگی برآوردن آن نیازها را بیان می‌کند. طراحی را می‌توان به روش‌های مختلفی مستند که در این درس هدف مستندسازی آن با مدل‌های CRC و نمودار ساده‌شده کلاس از نمودارهای UML است.

دقت نمایید که استفاده از زبان شی‌گرا برای ایجاد راه‌حل نرم‌افزاری، شرط کافی برای شی‌گرا بودن نرم‌افزار نیست. بلکه این نحوه طراحی و پیاده‌سازی است که مشخص می‌کند از اصول شی‌گرایی رعایت شده است یا خیر.

فرایند عمومی طراحی

گام‌های زیر یک فرایند عمومی در حد مطالبی که در این مستند ارایه شده است و متناسب برای محتوای این درس را ارایه می‌دهد. توجه نمایید که این فرایند تنها شامل بخش طراحی تا پیاده‌سازی بوده و جزئیات مربوط به قبل یا بعد از این مراحل را در نظر نگرفته است.

۱- شناسایی رفتار سیستم

- شناسایی رفتار سیستم خود یک موضوع مفصل به نام مهندسی نیازمندی‌ها و تحلیل نیازمندی‌ها است. توصیف رفتار سیستم باید به زبان مشتری بین استفاده‌کنندگان سیستم و مهندسان تولیدکننده سیستم نوشته شود.
- در حد این درس منظور از این قسمت توصیف مسئله یا پروژه‌ای است که به عنوان سوال مطرح می‌شود.

۲- پالایش نیازمندی‌ها

- هدف از این گام توصیف سناریوهایی گام‌به‌گام برای استفاده از سیستم است. این سناریوها در عین این که رفتار درست نرم‌افزار را بتوانند توصیف می‌کنند بلکه برای ارزیابی درستی رفتار نرم‌افزار بعد از پیاده‌سازی می‌توانند به کار روند.
- خروجی این کار مستند توصیف نیازمندی‌های نرم‌افزار است. یک فرایند رفت و برگشتی دریافت بازخورد بین تهیه‌کننده این مستند و استفاده‌کننده سیستم می‌تواند برای حصول توافق و دید واحد از نرم‌افزار بین تولیدکننده و استفاه‌کننده آن بسیار مفید باشد.

۳- شناسایی کلاس‌های نرم‌افزار

- در این مرحله کلاس‌های اصلی و رفتارهای مهم هر کدام شناسایی می‌شوند.
- استفاده از روش CRC در این مرحله آغاز می‌شود.(برای جزئیات بخش روش CRC مراجعه نمایید.)

۴- شروع مستندسازی طراحی

- این مرحله شامل کلاس‌هایی است که در قالب مدل‌های CRC در گام مرحله نهایی آن و پس از چند تکرار پالایش استخراج شده‌اند.

۵- مستندسازی طراحی

- در اینجا منظور مستندسازی کلاس‌ها در قالب نمودار کلاس است. در این مرحله روابط بین کلاس‌ها و شکل رابطه به صورت دقیق‌تر توصیف می‌شود. روابط پدرفرزندی مورد نیاز شناسایی و مستندسازی می‌شود.

۶- پیاده‌سازی کلاس‌ها

- در این مرحله نمودار کلاس به کلاس‌های جاوا تبدیل می‌شود و جزئیات بیشتری بسته به نیاز به آن‌ها اضافه می‌شود. کلاس‌هایی جدیدی بسته به اقتضائات پیاده‌سازی مورد نیاز باشد که باید نوشته شوند.

طراحی بد

طراحی‌ای خوب است که ساده، کوتاه و مختصر، قابل فهم، قابل تغییر و قابل بازبکارگیری[1] باشد و پیاده‌سازی آن به سادگی صورت گیرد.

طبق قاعده کلی که می‌توان چیزها را اضداد آنها شناخت یک راه توصیف طراحی خوب، توصیف طراحی بد است. معمولا نشانه یک اتفاق بد در نرم‌افزار را بوی بد[2] می‌نامند[3]. به عبارت دیگر فهرست زیر را می‌توان از بوهای بد طراحی ارایه داد.

۱- سفت و سخت بودن طراحی(Rigidity): تغییر سیستم سخت است زیرا هر بار که یک بخشی تغییر می‌کند باید بخش‌های دیگر نیز تغییر کند تا با تغییر جدید سازگار باشد. این زنجیره تغییر به صورت بازتاب‌های بی‌انتها در سیستم منتشر می‌شود(تغییر منتشرشونده).
۲- شکننده بودن طراحی(Fragility): تغییر در یک قسمت سیستم موجب از کار افتادن بخش‌های دیگری که ربط مستقیمی به بخش تغییر یافته ندارند، می‌شود.

[1] Reusability
[2] Bad Smell
[3] ریشه این اصطلاح به کتاب معروف refactoring نوشته مارتین فاولر و کنت بک برمی‌گردد. نقل قولی از مادر بزرگ کنت بک که می‌گفت وقتی بوی بدی می‌شنوی وقت عوض کردن کهنه بچه است، الهام بخش این اصطلاح بود به این معنا که وقتی نشانه‌هایی از طراحی یا برنامه نویسی بد را می‌بینی وقت بازنویسی(refactoring) است. شاید بتواند دلیل چنین نگاهی را به مسئله کیفیت طراحی یا پیاده‌سازی ، قابل فهم بودن و ملموس‌تر بودن توصیف ویژگی‌های بد و خطاهای طراحی و پیاده‌سازی در مقابل توصیف خشک ، انتزاعی و غیرقابل فهم طراحی یا پیاده‌سازی خوب برای برنامه‌نویسان دانست.

۳- غیرقابل جابجایی بودن طراحی(Immobility): بسیار سخت است که بتوان یک مولفه سیستم را برای بازبکارگیری از کل سیستم جدا کرد و جای دیگری مورد استفاده مجدد قرار داد.

۴- گرانروی زیاد طراحی(Viscosity): گرانروی در مایعات به مقاومت مایع در برابر شکل‌پذیری گفته می‌شود. به عنوان مثال گرانروی عسل بیشتر از گرانروی آب است. در طراحی نرم‌افزار منظور گرانروی، میزان مقاومت طراحی در برابر افزوده شدن کد جدید یا تغییر کد قبلی است. اگر در طراحی افزودن کد جدید به سیستم مستلزم تغییر زیادی و تصحیح زیاد خطاهای ایجاد شده باشد، طراحی گرانروی زیادی دارد.

۵- پیچیدگی بی‌فایده(Needless Complexity): شاید بسیاری ساختمان داده‌ها یا الگوریتم‌های پیچیده در طراحی پیشبینی شده است که در حال حاضر نیازی آن‌ها نیست. شاید طراح این شیوه حل مسئله را جالب‌تر، هیجان‌انگیزتر یا هوشمندانه‌تر دانسته است اما در توصیف نیازمندی‌های هیچ دلیل موجهی برای این پیچیدگی به صراحت وجود ندارد.

۶- برنامه‌نویسی برای آینده(Future Programming): طراحی و برنامه نویسی نه برای نیازمندی‌های امروز بلکه برای نیازمندی‌های آینده اینکه به صراحت قید نشده‌اند بلکه طراح یا برنامه نویس پیش‌بینی می‌کند که ممکن است در آینده پیش بیاید انجام شده است. دقت کنید که این ویژگی حالت خاصی از ویژگی قبلی است و برخی اوقات برای پیچیدگی بی‌فایده توجیهاتی از منظر برنامه‌نویسی برای آینده آورده می‌شود.

نکته جالب آن است که این ویژگی با ویژگی اول یعنی سفت وسخت بودن طراحی در تضاد است و مثل بسیاری موقعیت‌های مهندسی، تصمیم درست نه تبعیت از قواعد قطعی کلی بلکه تشخیص درست نقطه تعادل مناسب بین دو قاعده کلی متضاد است.

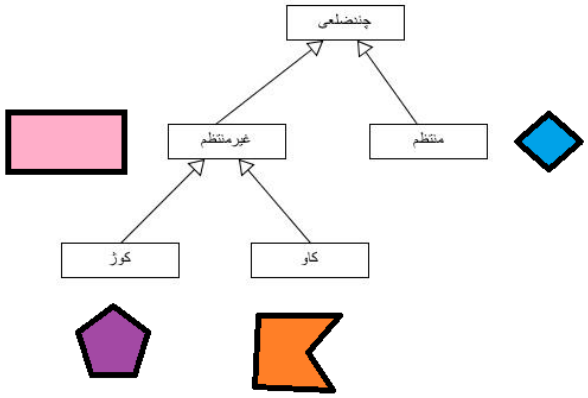
۷- تکرار بی‌فایده(Needless Repetition): به نظر می‌رسد که بخش‌های مختلف کد توسط دو برنامه نویس متفاوت و با استفاده از copy/paste نوشته شده است.

۸- طراحی برای استثنا: هنگام طراحی رفتار مورد انتظار سیستم و ساختار مورد نیاز آن برای نه حالت‌های عمومی استفاده از سیستم بلکه موارد استثنا و خاص مد نظر قرار گرفته است. این مسئله برای طراحی ساختمان مانند آن است که برای روبروی ورودی ساختمان به جای پلکان نفررو، سطح شیبدار عریض(برای استفاده از ویلچر) پیش-بینی شود و پلکان کم‌عرضی در کنار سطح شیبدار پیاده‌شود.

۹- دوختن کت‌وشلوار برای تکمه: این مسئله خصوصا برای برنامه‌نویسان و طراحان جوان بسیار پیش می‌آید. در هنگام طراحی بخش یا بخش‌هایی از سامانه و یا جنبه‌هایی از طراحی که بیشترین ابهام یا پیچیدگی یا تازگی را از نظر طراح دارند به صورت خودآگاه یا ناخودآگاه به عنوان مرکز طراحی در نظر گرفته‌شده است و بقیه اجزا حول

آنها و در خدمت آنها طراحی می‌شوند. در چنین طراحی‌هایی ارتباطات غیرضروری و کلاس‌های قادرمطلق(Omnipotent)[۴] بسیار دیده می‌شوند.

۱۰-ابهام در طراحی: نمی‌توان از خود طراحی به هدف آن از یک ساختار یا ارتباط پی‌برد. جنبه یا منظری که طراح بر اساس آن اقدام به افزار بخش‌ها و یا برقراری ارتباطات کرده است یه صورت واضح و ثابت نیست و گاهی درگیر تناقض است. به عنوان مثال در ساختار پدرفرزندی جنبه‌ای انتزاع[۵] که سلسله مراتب بر اساس آن تشخیص داده شده است، در سطوح مختلف با هم سازگار نیست. مثال زیر وضعیتی از چنین شرایط را در هندسه نشان می‌دهد.



نمودار کلاس UML به ما کمک می‌کند تا پیش از پیاده‌سازی بتوانیم طراحی انجام شده را برای پیدا کردن بوهای بد طراحی مورد ارزیابی قرار دهیم و با دیگران در خصوص آن صحبت کنیم.

[۴] کلاس‌های قادر مطلق به عنوان یک پادالگو(Antipattern) شناخته می‌شوند. پادالگوها الگوهای تکراری هستند که همواره باید از آنها اجتناب کرد. کلاس قادر مطلق به کلاس‌هایی گفته می‌شوند که وظایف، رفتارها و ویژگی‌های زیاد، متنوع و عموما بی‌ربط به همی دارند. این کلاس‌ها ارتباطات زیادی با دیگر کلاس‌ها ایجاد می‌کنند. تغییر در چنین کلاس‌هایی عموما منتشر می‌شوند و خود این کلاس‌ها هم در معرض انتشار تغییرات دیگران هستند. چنین کلاس‌هایی به صورت ستاره(کلاسی و ارتباطات زیاد) یا کلاسی با رفتارهای مفصل در نمودار کلاس به چشم می‌آیند.

[5] Abstraction

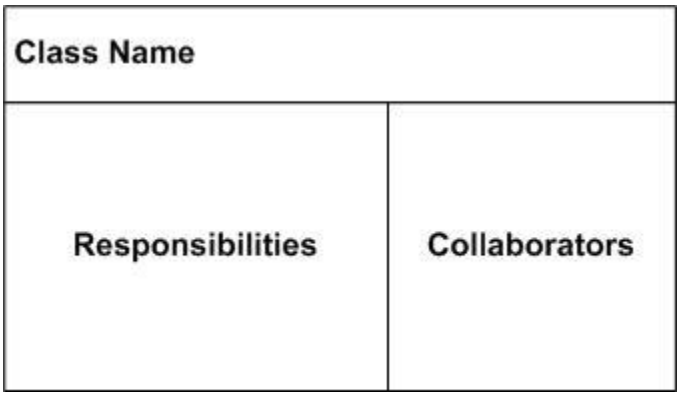
روش CRC

روش Class Responsibility Collaborator ، روشی برای توفان ذهنی به منظور شناسایی کلاس‌های طراحی است. این روش توسط وارد کانینگام[۶](مبدع مفهوم ویکی) و کنت بکت[۷](نویسنده معروف شی‌گرایی) در ابتدا به عنوان یک روش آموزشی معرفی شد اما بعدها بیشتر مورد توجه طراحان قرار گرفت. این روش مبتنی بر ایجاد مدل‌های CRC برای طراحی است.

مدل CRC شامل سه بخش نام کلاس، وظایف کلاس و همکار کلاس است.

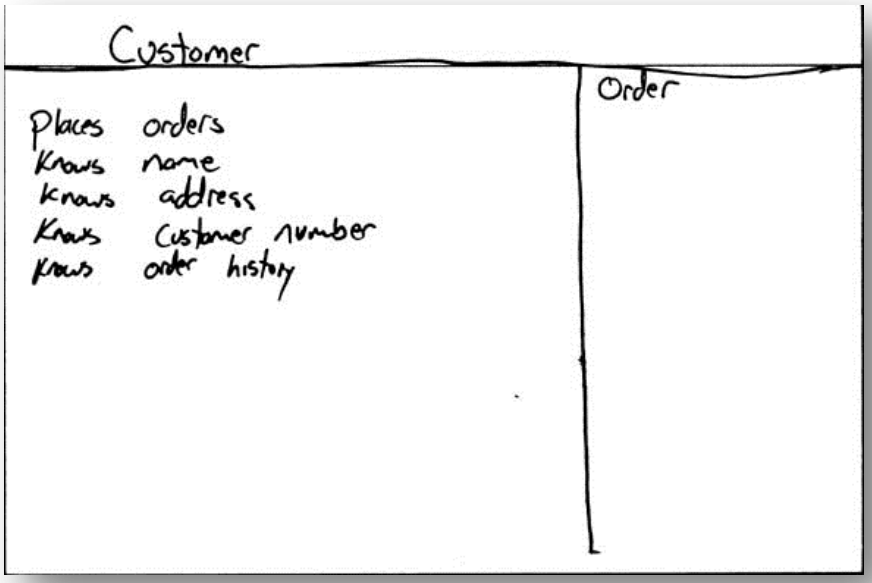
- منظور از کلاس همان مفهوم کلاس در شی‌گرایی است که تجریدی از نوع دسته‌ای از اشیای مشابه می‌باشد.
- منظور از وظایف کلاس چیزی است که کلاس می‌داند یا کاری است که باید انجام دهد.
- همکار کلاس شامل اسامی دیگر کلاس‌هایی است که کلاس مورد نظر برای انجام وظایف خود به تعامل با انها احتیاج دارد.

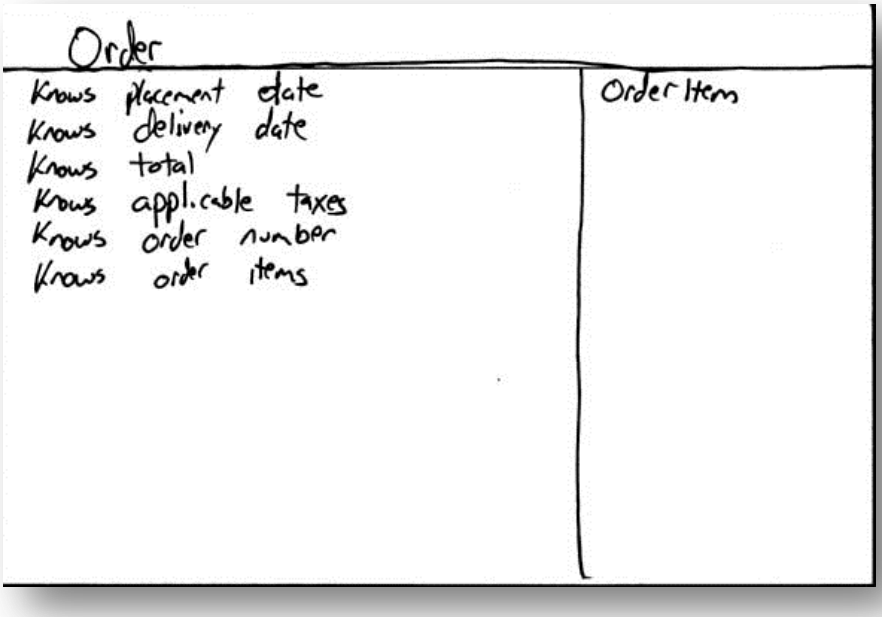
در این روش اطلاعات مربوط به هر کلاس در کارت‌هایی به شکل زیر نوشته می‌شود:



[۶] Ward Cunningham
[۷] Kent Beck

چندین مثال پر شده برای کارت‌های CRC را می‌توانید در زیر ببینید:





اجرای CRC

برای انجام روش CRC گام‌های زیر به صورت تکراری و افزایشی انجام می‌شود. منظور از تکراری بودن این است که بعد از دور کامل اجرا دوباره از گام اول بایستی شروع کرد و منظور از افزایشی آن است که خروجی دورهای قبلی اجرا به عنوان ورودی در دور جاری استفاده می‌شوند و نتیجه کار از تکمیل و پالایش تدریجی خروجی حاصل می‌شود.

۱. پیدا کردن کلاس‌ها: فرایند پیدا کردن کلاس در واقع بخشی از کار تحلیل است که وظیفه مشخص کردن اجزای اصلی سازنده نرم‌افزار را بر عهده دارد. ساده ترین روش شناسایی کلاس‌ها تمرکز بر اسامی مطرح شده در فضای مسئله است. از اسامی کلاس باید ساده، واضح و به صورت مفرد باشد.
۲. مشخص کردن وظایف: پس از مشخص شدن کلاس‌های بایستی در خصوص هر کدام از آنها به این دو سوال پاسخ دارد. این کلاس چه اطلاعاتی را بایستی نگهداری کند و چه وظایفی را بر عهده دارد.
برخی از اوقات وظایف کلاس‌های از خدماتی که باید به کلاس‌های همکار خود ارایه دهند مشخص می‌شود. به عبارت دیگر بایستی به دیگر کارت‌های که کلاس جاری به عنوان همکار آورده شده است، دقت کرد و مشخص نمود که کلاس جاری برای اینکه بتواند وظایف خود در قبال آن کلاس‌ها را انجام دهد چه اطلاعاتی را باید نگهداری کند و چه وظایفی را باید بر عهده بگیرد.
۳. مشخص کردن همکار: اغلب یک کلاس برای انجام وظایف خود تمامی اطلاعات را در اختیار ندارد. بلکه باید از همکاری دیگر کلاس‌های برای انجام این کار استفاده کند. همکاری بین کلاس‌های به یکی از این دو صورت انجام می‌شود: درخواست اطلاعات یا درخواست برای انجام کاری.
برای مشخص کردن همکارهای یک کلاس همیشه می‌توان این سوال را پرسید: آیا کلاس به تنهایی می‌تواند تمامی وظایف مشخص شده برای آن را انجام دهد؟ اگر پاسخ این سوال منفی است یکی از دو حالت زیر پیش می‌آید.

- در حال حاضر کلاسی وجود دارد که می‌تواند وظیفه مورد نیاز کلاس جاری را انجام دهد یا اینکه می-توان چنین وظیفه‌ای را برای آن کلاس تعریف کرد.
- باید کلاس یا کلاس‌های جدیدی شناسایی کرد و وظیفه مورد نیاز را به آن کلاس یا کلاس‌ها واگذار نمود.

۴. کارت‌ها را به اشتراک بگذارید: برای بهبود درک تمامی کسانی که در طراحی درگیر هستند کارت‌ها باید روی یک میز به شکل مرتبی چیده شده باشند. هر دو کارتی که با یکدیگر همکاری می‌کنند باید کنار هم چیده شوند و کارت‌هایی که ارتباطی به یکدیگر ندارند جدا از باشند.

این فرایند بایستی چندین بار به صورت تکراری بر روی کارت‌ها تکرار شوند. تا زمانی که مطمئن شوید که همه تغییر جدیدی مورد نیاز نیست.

پس از اتمام کار می‌توان به روش‌های مختلف طراحی انجام شده را ارزیابی نمود که روش‌های زیر از آن جمله هستند:

۱. وظایف نوشته شده برای تک‌تک کلاس‌ها را مرور کنید. درخصوص هر وظیفه از خود بپرسید که آیا همه اطلاعاتی که برای انجام این وظیفه مورد نیاز است برای کلاس در نظر گرفته اید؟ آیا همه کلاس‌های همکاری که برای انجام این وظیفه مورد نیاز است به عنوان همکار کلاس در کارت ذکر شده است؟
۲. در خصوص تک‌تک اطلاعاتی که بایستی برنامه برای انجام کار خود نگهداری کند از خود سوال بپرسید و مطمئن شوید که کلاسی برای نگهداری و مدیریت آن اطلاعات پیش‌بینی کرده اید.
۳. سناریوهای اصلی استفاده از برنامه را در نظر بگیرد و فرایند گام‌به‌گام انجام آن‌ها را در نظر بگیرید. حال برای تک‌تک گام‌ها سوالات زیر را از خود بپرسید:
    a. آیا تک‌تک گام‌ها به عنوان وظیفه کلاسی پیشبینی شده است؟
    b. آیا ارتباطات لازم برای انجام هر گامی بین کلاس‌ها به عنوان همکار وجود دارد؟

نکات مهم

- دقت کنید که کلاس‌های شناسایی شده در روش CRC شامل نسخه اولیه از کلاس‌های شناسایی شده هستند و ممکن است در مرحله بعدی یعنی رسم نمودار کلاس در UML کلاس‌های کم یا زیاد شوند. به عنوان مثال روابط پدرفرزندی در مدل CRC نشان داده نمی شود در حالی که در مدل کلاس UML بایستی به صورت دقیق مشخص شوند.
- دقت کنید که کلاس‌های شناسایی شده در مدل CRC یا حتی کلاس‌های مدون شده در مدل کلاس UML در واقع کلاس‌های طراحی هستند و باز ممکن است که در پیاده‌سازی به دلایل مختلف نیاز به کلاس‌های دیگری نیز باشد که کم‌کم با تکمیل شدن راه حل شما برای مسئله ارایه شده خود را نشان می‌دهند. به عنوان مثال شما کلاسی احتیاج دارید که متد main برنامه جاوا در باشد برنامه از آنجا آغاز شود. چنین کلاسی الزاما از توصیفات ارایه شده در توصیف مسئله قابل استخراج نیست.
- در یک پروژه بزرگ واقعی عملا سه نسخه کلاس به نام‌های کلاس‌های تحلیل، طراحی و پیاده‌سازی شناسایی و پیاده‌سازی می‌شوند که بیشتر به جنبه تکاملی بودن گام‌به‌گام ایجاد راه‌حل نرم‌افزاری برای یک مسئله اشاره می‌کند که شامل گام‌های پایه تحلیل، طراحی و پیاده‌سازی است. جزئیات مربوط به آن‌ها خارج از محتوای این آموزش است. قطعا برای مسائلی که شما در درس برنامه‌سازی پیشرفته با آن‌ها مواجه می‌شوید، مراحل و جزئیات ارایه شده در این مستند برای فرایند طراحی کافیست.

مثال برای روش CRC

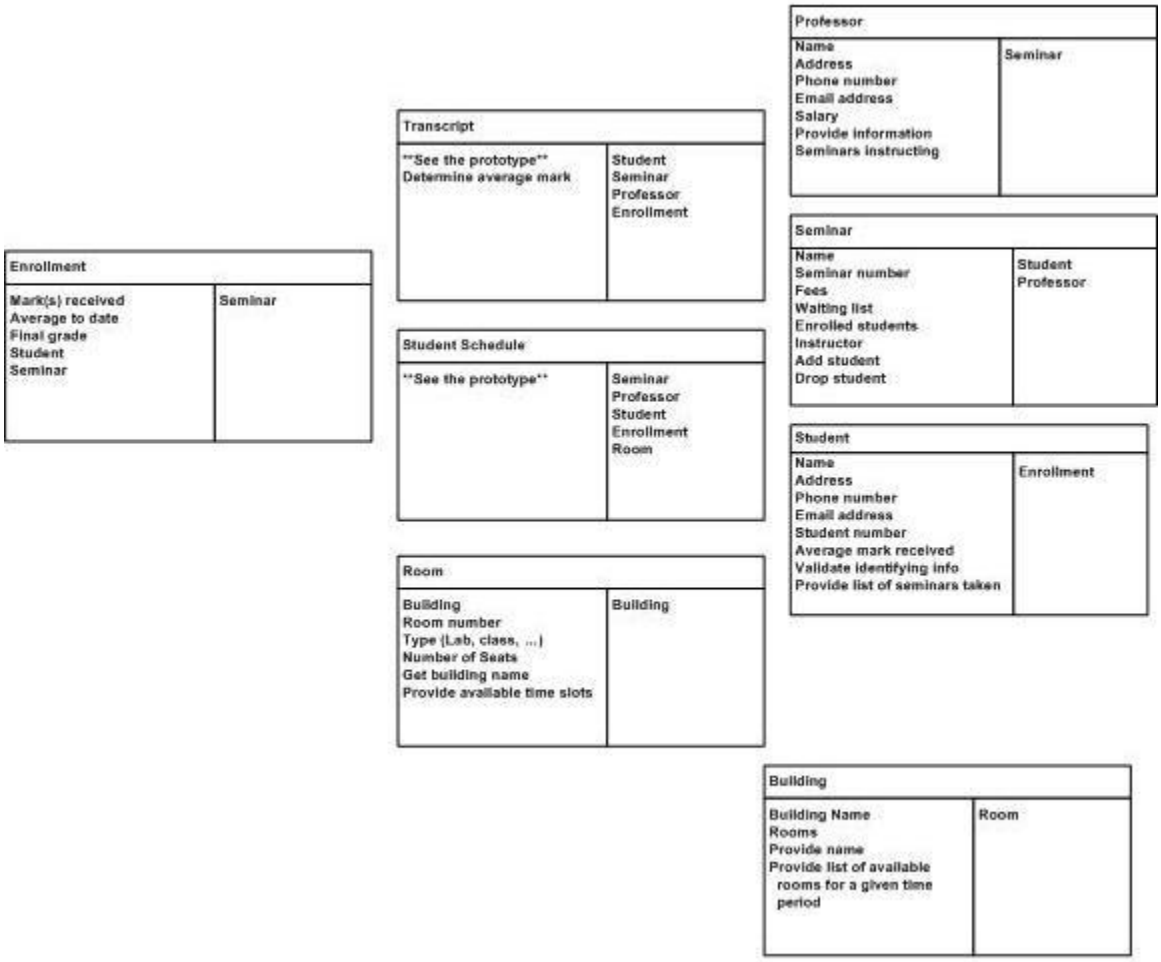
صورت مسئله

یک برنامه مدیریت زمانبندی کلاس‌های برای برپایی مدیریت ارایه‌ها(سمینار) در یک دوره فشرده را در نظر بگیرید.هر سمینار با شرکت یک استاد و چنین دانشجو برگذار می‌شود. برای هر سمینار یک گزارش سمینار(Transcript) تهیه می‌شود که شامل نمره متوسط دانشجویان در سمینار است. در پایان هر سمینار از دانشجویان ارزیابی می‌شود و نمره‌ای برای دانشجو در نظر گرفته می‌شود. بایستی برای هر سمینار یک یک کلاس(Room) اختصاص داده شود. کلاس‌ها ممکن است که در ساختمان‌های مختلفی قرار داشته باشند.

راه حل

به نظر می‌رسد که برای مسئله فوق عبارت‌های دانشجو، استاد، سمینار،گزارش سمینار، کلاس و ساختمان نشان‌گرهای مناسبی برای اسامی کلاس‌ها باشد. اگر فرایند را چندین بار اجرا نمایید به کلاس‌های دیگری نیز می‌رسید که در ابتدا چندان بدیهی نیستند.

شکل زیر وضعیت کارت‌ها را بعد از چندین بار اجرا کردن فرایند CRC نشان می‌دهد. لطفا به چینش کارت‌ها دقت کنید.



نمودار کلاس در UML

هیچ روشی برای طراحی بدون ابزار و زبان مستندسازی کامل نیست. این بخش به توضیح نمودار کلاس UML به عنوان زبان مستندسازی طراحی می‌پردازیم.

UML چیست؟

UML یک زبان نشانه‌گذاری[۸] گرافیکی کشیدن نمودار[۹] برای نشان‌دادن مفاهیم مختلف نرم‌افزار است.از این زبان می‌توان برای کشیدن نمودارهایی برای توصیف فضای مسئله، طراحی نرم‌افزاری و حتی جزئیات نرم‌افزار پیاده‌سازی شده استفاده کرد.

نمودارهای UML بر اساس استاندارد نسخه ۲ آن شامل ۱۳ نمودار است که مهمترین آنها عبارتند از:

Class Diagram, Usecase Diagram, State Diagram, Activity Diagram

Sequence Diagram, Package Diagram, Deployment Diagram

برای رسم نمودارهای Uml می‌توانید از ابزارهای زیر استفاده کنید:

نام	توضیحات	پیوند دسترسی
IBM Rational Rose	ابزار رسمی UML و کمی قدیمی	www.ibm.com/software/rational/uml/products/
Enterprise Architect	ابزار کامل و پیچیده	www.sparxsystems.com.au/products/ea/
Visual Paradigm	ابزار کامل و پیچیده	http://www.visual-paradigm.com/
Umlet	ابزار ساده و بازمتن	http://www.umlet.com/

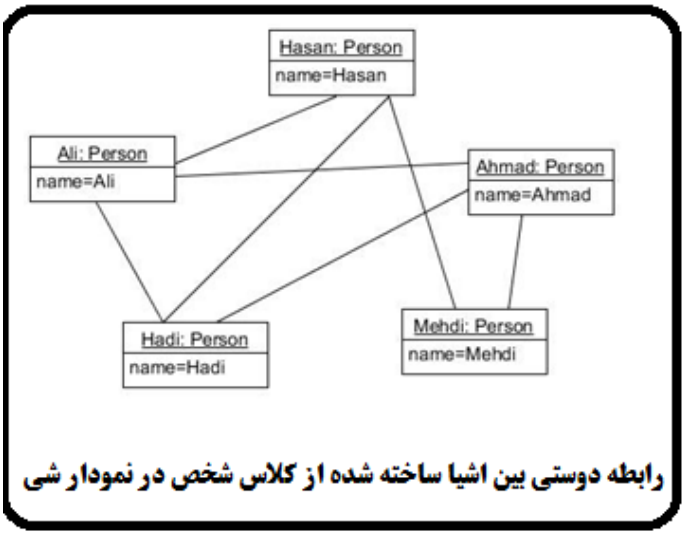
برای نمودارهای کلاس این درس استفاده از نرم‌افزار Umlet توصیه می‌شود.

---

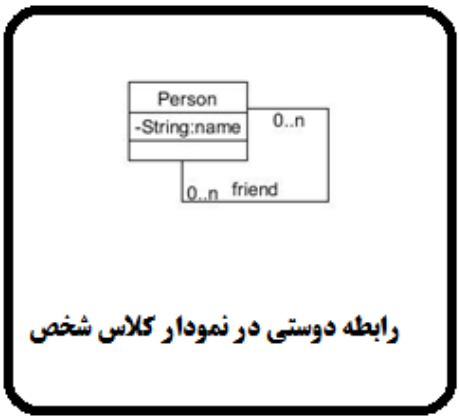
[۸] Notation
[۹] Diagram

نمودار کلاس

نمودار کلاس برای نشان‌دادن کلاس‌ها، ویژگی‌ها و ارتباطات آن‌هاست. منظور از ارتباطات بین کلاس‌های ارتباطات ایستا بین انواعی است که کلاس‌ها آن‌ها را نمایندگی می‌کنند نه رابطه پویا بین اشیایی که از آن‌کلاس‌ها ساخته می‌شوند. به عنوان مثال رابطه دوستی روابط ایستا بین کلاس‌ها و رابطه پویا بین اشیا به دو صورت زیر نشان داده می‌شود.

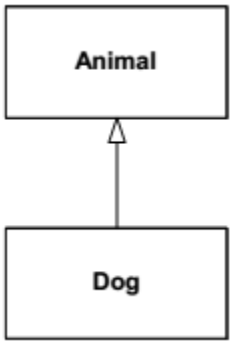


رابطه دوستی بین اشیا ساخته شده از کلاس شخص در نمودار شی

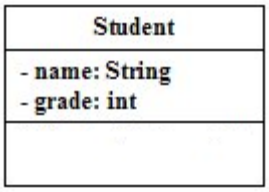


رابطه دوستی در نمودار کلاس شخص

در این نمودار هر کلاس با یک مستطیل نشان داده می‌شود که نام کلاس درون آن نوشته شده است. به عنوان مثال در شکل زیر دو کلاس دیده می‌شوند.



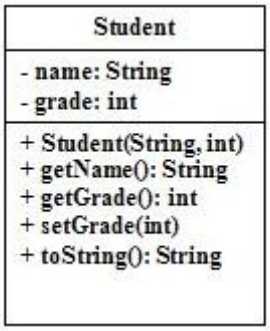
هر کلاس می‌تواند ویژگی‌هایی[۱۰] داشته باشد که با خطی زیر نام کلاس مشخص می‌شود. ویژگی‌های کلاس در واقع جزئیات اطلاعاتی است که کلاس نگهداری می‌کند. به عنوان مثال نمودار زیر نشان دهنده کلاس دانشجو است.



کد جاوای نمودار فوق به شرح زیر است:

```
public class Student {
    private String name;
    private int grade;
}
```

حال اگر رفتارهایی به کلاس دانشجو اضافه کنیم، شکل نمودار کلاس آن به شرح زیر می‌شود.

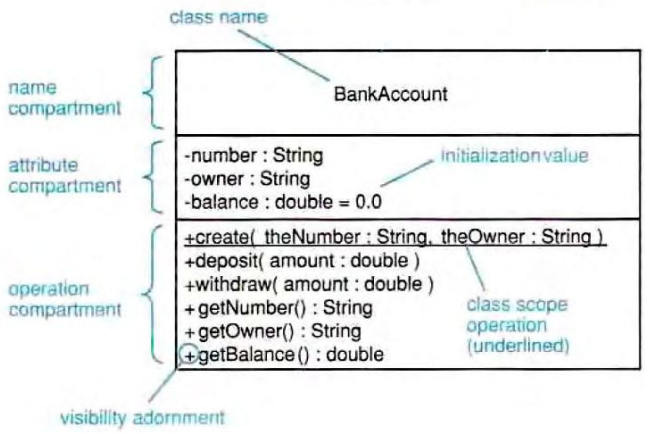


[۱۰] Attributes

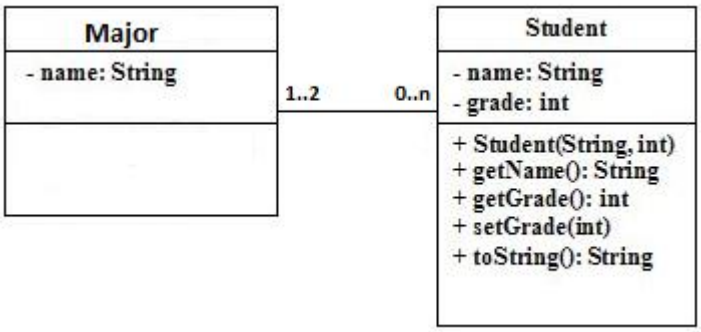
کد جاوای معادل نمودار فوق به صورت زیر است:

```java
public class Student {
    private String name;
    private int grade;

    public Student(String name, int grade) {
        this.name = name;
        this.grade = grade;
    }

    public String getName() { return name; }
    public int getGrade() { return grade; }
    public void setGrade(int grade) { this.grade = grade; }
    @Override
    public String toString() {
        return "Student{ name='" + name + '\'' +", grade=" + grade + '}';
    }
}
```

شکل زیر توصیفی از یک نمودار کلاس برای یک کلاس تنها را نشان می‌دهد:



برای نشان دادن روابط بین کلاس‌ها یک یک خط ساده از بین آن‌ها استفاده می‌شود. به عنوان مثال شکل زیر رابطه بین دانشجو و رشته را نشان می‌دهد.



کد جاوای مربوط به این ارتباط در کلاس دانشجو با حذف متدهای این کلاس به شکل زیر است:

```
public class Student {
    private String name;
    private int grade;
    private List<Major> studentMajor;
}
```

بر روی خط مربوط به رابطه بین کلاس‌ها می‌توان چندی[11] رابطه را مشخص کرد. به عنوان مثال برای رابطه بین دانشجو و رشته که در بالا نشان داده شده است، هر دانشجو می‌تواند یک یا دو رشته داشته باشد و هر رشته‌ای می‌تواند هیچ یا چند دانشجو داشته باشد.

[11] Multiplicity

بخش مربوط به ویژگی‌های کلاس از قالب زیر پیروی می‌کند:

```
visibility name: type multiplicity = default {property-string}
```

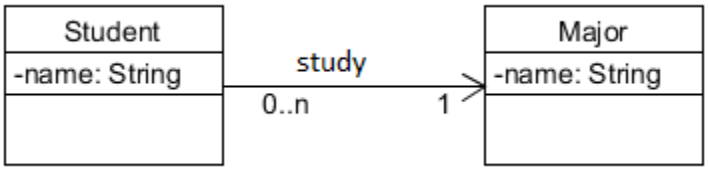
visibility ویژگی در واقعی محدوده دسترسی به ویژگی را مشخص می‌کند که شامل موارد زیر است:

نشانه در Class Diagram	کلید واژه جاوا	توضیحات
–	private	دسترسی خصوصی کلاس
~	-	دسترسی تنها در بسته جاری
#	protected	دسترسی در بسته جاری و فرزندان
+	public	دسترسی عمومی

متدها و کلاس‌های استاتیک به صورت Underlined نوشته می‌شوند.

روابط بین کلاس‌ها association نامیده می‌شوند که به صورت یک پاره خط و یا یک بردار نشان داده می‌شوند. پیکان برداری روی یک رابطه جهت رابطه را نشان می‌دهد. اگر تنها پاره خط دو کلاس را بهم متصل کند، به معنای رابطه دوطرفه است. نقش رابطه را می‌توان روی پاره‌خط رابطه نوشت.

به عنوان مثال نمودار زیر را در نظر بگیرید:

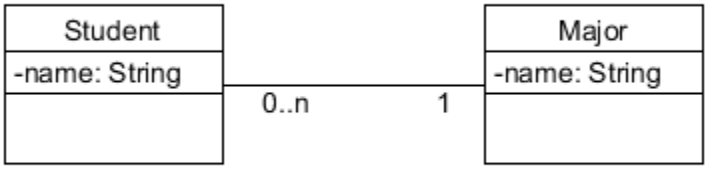


این نمودار نشان می‌دهد که :

- هر دانشجویی تنها یک رشته دارد.
- هر رشته‌ای می‌تواند مال هیچ یا چند دانشجو باشد.
- می‌توان مفهوم رابطه را در قالب یک فعل بر روی پاره‌خط رابطه نوشت.

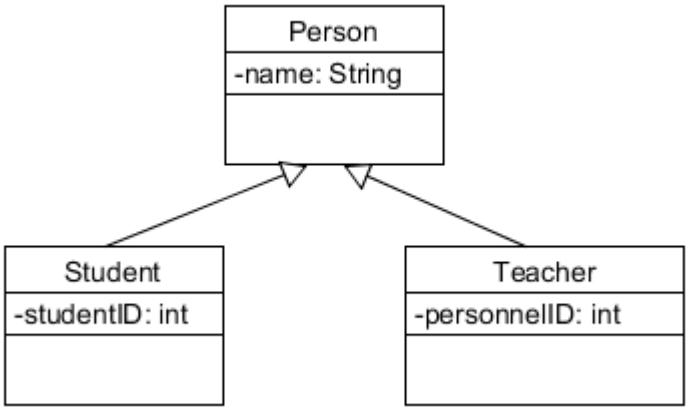
- در کلاس دانشجو فیلدی از نوع رشته وجود دارد ولی در کلاس رشته فیلدی از نوع فهرستی از دانشجوها وجود ندارد.

در حالی که برای نمودار زیر



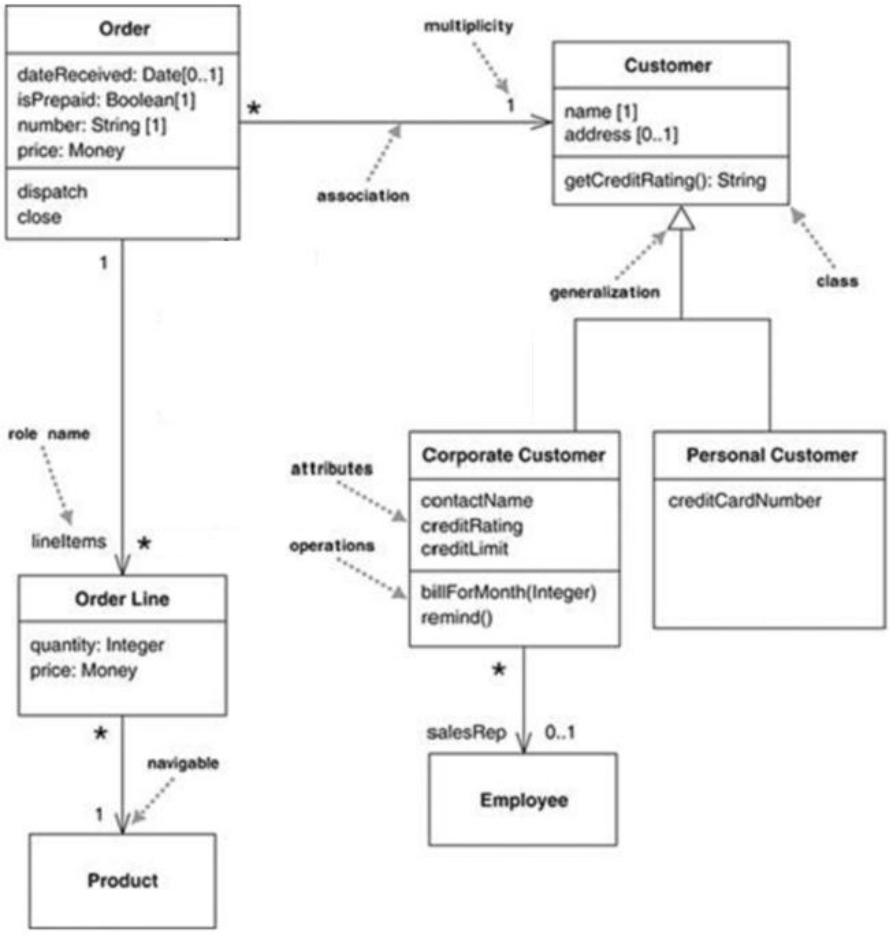
در کلاس دانشجوی فیلدی از نوع رشته وجود دارد و در کلاس رشته فیلدی از نوع ارایه یا لیستی از دانشجویان آن رشته وجود دارد.

برای نشان دادن رابطه پدرفرزندی از یک پیکان توخالی از سمت کلاس فرزند به کلاس پدر استفاده می‌شود.



در نمودار بالا کلاس‌های دانشجو و مدرس فرزند کلاس شخص هستند.

شکل زیر نمونه‌ای از نمودار کلاس مفصل‌تری را نشان می‌دهد:



اصول حاکم بر طراحی کلاس‌ها

وقتی که به یک مدل نمودار کلاس نگاه می‌کنیم دنبال چه چیزی هستیم؟ چگونه یک طراحی خوب را می‌توان از طراحی بد تشخیص داد؟ این بخش در خصوص معیارهایی صحبت می‌کند که می‌تواند برای این کار استفاده کرد.

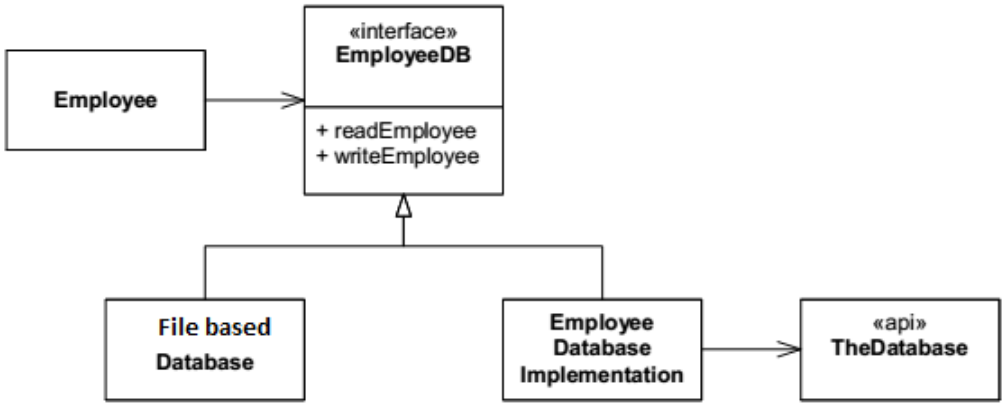
اصل بازِبسته[12](باز بودن در عین بستگی)

هر کلاسی باید برای توسعه باز باشد و برای دستکاری(modification) بسته باشد.

شاید بتوان گفت که این اصل مهم‌ترین اصل طراحی کلاس‌هاس و دیگر اصل‌ها بیان‌های دیگر یا تاکیداتی بر جنبه‌های مختلف این اصل است.

معنای این اصل ساده است. باید بتواند محیط عملکرد یک کلاس را بتون نیاز به تغییر در پیاده‌سازی داخلی یک کلاس تغییر داد(بازبکارگیری). همچنین باید بتوان رفتارهای جدیدی به کلاس افزود یا پیاده‌سازی رفتارهای قبلی را بدون تغییر واسطه خارجی آن رفتارهای تغییر داد، بدون آنکه خللی در استفاده دیگر کلاس‌های وابسته به این کلاس از آن ایجاد شود.

مثال زیر را در نظر بگیرید. فکر کنید که کلاس کارمند تا کنون برای ذخیره‌سازی کارمندان از فایل‌ها استفاده می‌کرد. اکنون امکان اتصال و ذخیره سازی در دادگان(Database) فراهم شده است. با طراحی زیر و رعایت این اصل بدون اینکه تغییری در کلاس کارمند اینجاد شود، می‌تواند از امکان جدید استفاده کند. زیرا که به کلاس concrete بلکه به یک کلاس abstract برای وظایف ذخیره‌سازی وابسته است.



[12] Open Closed Principle(OCP)

اصل یکتایی وظیفه کلاس[13]

هر کلاس باید طوری طراحی شود که تنها یک دلیل برای تغییر آن وجود داشته باشد.

بایستی کلاس‌های طوری طراحی شوند که تنهای یک مولفه مشخص از فضای راه حل را که بر اساس جنبه مشخصی شناسایی شده است، را نمایندگی کنند. نباید یک کلاس جنبه‌ها و وظیفه‌های غیرمرتبط با مفهومی که بر اساس آن شناسایی شده است را بر عهده بگیرد.

به عنوان مثال اگر در یک فضای مسئله کلاس دانشجو شناسایی شده است. اگر قرار باشد یک فایل Xml حاوی اطلاعات دانشجو خوانده شود و بر اساس آن یک شی دانشجو ساخته شود، نباید متد مربوط به خواندن فایل Xml در خود کلاس دانشجو پیش‌بینی شود. کلاس دانشجو وظیفه مدیریت داده‌ها و رفتار مفهوم دانشجو را بر عهده دارد نه چیز دیگر. در غیر اینصورت در صورت نیاز به پشتیانی قالب فایل دیگری مانند JSON ، باید کلاس دانشجو هم تغییر کند که بسیار بد است.

اصل جایگزینی لیسکوف[14]

باید بتوان کلاس فرزند را جایگزین کلاس پدر کرد بدون انکه استفاده‌کنندگان از کلاس پدر با مشکلی مواجه شوند.

این اصل در واقع یک گونه خاصی از اصل OCP است. از این رو نقض LSP به نقض OCP منجر می‌شود و عکس این گزاره صحیح نیست.

این اصل که در جاوا امضای متدهای در فرزندان بایستی از قواعد خاصی نسبت به متد پدر پیروی کند از این اصل نشات می‌گیرد. به عنوان مثال دسترسی در متد فرزند نمی‌تواند محدودتر از دسترسی متدی که override می‌کند باشد زیرا در این صورت امکان جایگزینی رفتار پدر با رفتار فرزند را برای استفاده‌کنندگان با مشکل مواجه می‌کند.

اصل لیسکوف بر اساس استوار است که هر فراخوانی متدی یک پیش‌شرط[15] و پس‌شرط[16] دارد که در صورت برآورده بودن پیش‌شرط قبل از فراخوانی، بعد از فراخوانی پس‌شرط برقرار خواهد بود. بر اساس لیکوف هر فراخواهی متد پیاده‌سازی override شده از فرزند روابط بین پیش‌شرط و پس‌شرط فراخوانی متد پدر را رعایت می‌کند.

[13] Single Responsibility Principle(SRP)
[14] Liskov Substation Principle(LSP)

پیاده‌سازی override در فرزند نمی‌تواند پیش‌شرطی قوی‌تری از پیش‌شرط متد پدر داشته باشد و همچنین نمی‌تواند پس‌شرط ضعیف‌تری از کلاس‌پدر داشته باشد. برای مثال بالا حق دسترسی متد پدر جز پیش‌شرط متد پدر است ولی متد فرزند نمی‌تواند حق دسترسی محدودتری از چیزی که متد پدر اجبار می‌کند، داشته باشد.

هر چند که در خصوص برخی قواعد مانند مثال فوق کامپایلر می‌تواند از برقراری اصل لیسکوف مطمئن شود ولی در خصوص پیاده‌سازی داخلی متدهای جاوا یا هر زبان دیگری نمی‌تواند برنامه‌نویس را مجبور کند که اصل لیسکوف را رعایت نماید. بلکه خود برنامه‌نویس باید هنگام استفاده از رابطه پدرفرزندی به این نکته توجه کند.

اصل وارونگی وابستگی[۱۷]

کلاس‌ها باید به واسطه بیرون یکدیگر وابسته باشند نه به جزئیات پیاده‌سازی یک دیگر

واسطه‌های بیرونی نباید به جزئیات پیاده‌سازی وابسته باشد بلکه جزئیات پیاده‌سازی باید به واسطه‌های بیرونی وابسته باشد.

این اصل هم باز یک بیان دیگر حالت خاصی از اصل OCP است و درواقع راهی را نشان می‌دهد که از طریق آن می‌توان از برقراری OCP اطمینان حاصل کرد. به زبان ساده این اصل استفاده از واسطه‌ها[۱۸] را برای کاهش وابستگی نابجا تشویق می‌کند راه درست استفاده از آن‌ها را نشان می‌دهد.

به وابستگی به واسطه‌ها به جای وابستگی به پیاده‌سازی[۱۹] وابستگی مجرد[۲۰] می‌گویند. استفاده از وابستگی مجرد تقریبا هرجایی که برنامه نویس احساس می‌کنند ممکن است که پیاده‌سازی در معرض تغییرات آینده باشد، توصیه می‌شود. در مثال مربوط به OCP دیدیم که در واقع استفاده از وابستگی مجرد بود که امکان تغییر فراهم نموده بود. هر چند که نباید در استفاده از این اصل زیاده‌روی کرد یعنی تبدیل همه وابستگی‌های پیاده‌سازی به وابستگی مجرد در صورتی که در افراط-گرایانه باشد، نتیجه‌ای جز افزودن کار برنامه‌نویس و پیچیده‌کردن بیهوده برنامه نخواهد داشت.

---

[15] Precondition
[16] Postcondition
[17] Dependency Inversion Principle(DIP)
[18] Interfaces
[19] Concrete Coupling
[20] Abstract Coupling

به این موقعیت‌هایی که برنامه‌نویس در استفاده از اصول طراحی افراط می‌کند با افزودن گزینه‌های اضافه و مولفه‌های زیاد موجب پیچیدگی‌ای نابجا در برنامه می‌شود، بیش‌طراحی[۲۱] می‌گویند.

اصل تفکیک واسطه‌ها[۲۲]

چندین واسط خاص بهتر از یک واسط عمومی برای یک کلاس است.

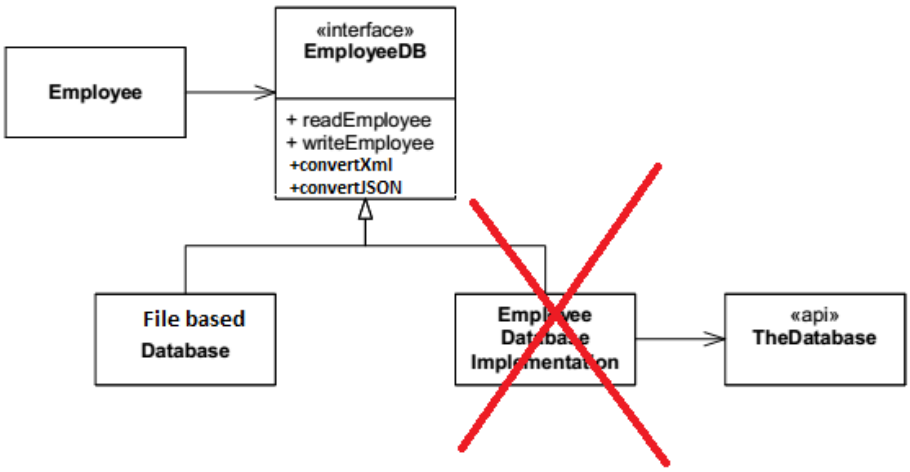
به عبارت دیگر از طرف استفاده کنندگان واسط‌ها، بهتر است استفادکنندگان واسط‌ها به واسط‌هایی وابسته شوند که از همه متدهای انها استفاده می‌کنند.

در اصل قبلی در خصوص اهمیت وابستگی مجرد و مزیت‌های استفاده از واسطه‌ها صحبت شد. تمامی مزایای فوق زمانی ممکن است که واسطه‌های تعریف شده تا حد ممکن خاص منظوره طراحی شده باشند با به عبارت دیگر اصل یکتایی وظیفه کلاس یا SRP در آن‌ها رعایت شده باشد.

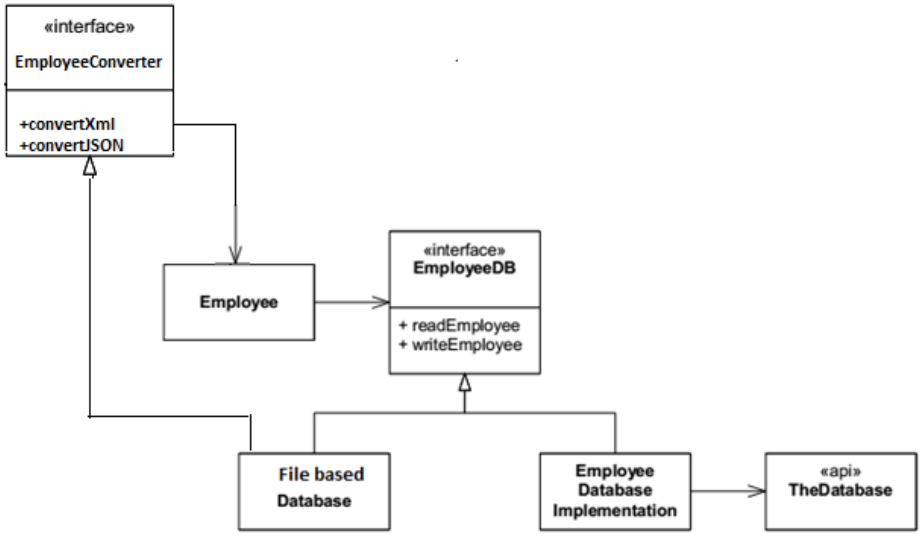
امکان استفاده از وراثت چندگانه برای واسطه‌ها در جاوا که امکان تعریف واسطه‌های پدر گوناگون برای یک کلاس را فراهم کرده است، می‌تواند در رعایت این اصل کمک‌کننده باشد.

[21] Over Design
[22] Interface Segregation Principle(ISP)

مثال مربوط به OCP در خصوص وابستگی مجرد را در نظر بگیرید. اگر برای ذخیره‌سازی در فایل به تبدیل فایل Xml و JSON به شی کامند احتیاج می‌داشتیم و چنین ویژگی‌هایی را در واسطه EmployeeDB وارد می‌کردیم، دیگر نمی‌توانستیم پیاده‌سازی جدیدی از آن برای پشتیبانی دادگان ارایه دهیم.



برای رعایت این اصل بایستی طراحی زیر را انجام داد.



منابع

1. Object oriented programming with java, Kirk Knoernschild, Addison Wesley,2001
2. Object oriented programming with java , ADVANCED TOPICS, EUGENE AGEENKO,2003, http://cs.joensuu.fi/pages/intra/ageenko/OOP_with_Java.pdf
3. UML for Java Programmers, Robert C Martin, Prentice-Hall,2002
4. Refactoring: Improving the Design of Existing Code, Martin Fowler , Kent Beck and others, Addison-Wesley Professional,1999
5. CRC method page on agilemodeling: http://www.agilemodeling.com/artifacts/crcModel.htm