بسمه‌تعالی

تمرین شماره ۳

استفاده از الگوهای بازآرایی و بازمهندسی در موقعیت های مختلف

استاد درس:

دکتر رامان رامسین

تهیه‌کننده:

[عادل وحدتی ۹۳۲۰۸۲۸۹]

نیمسال بهار ۹۵-۱۳۹۴

فهرست مطالب

۱ تغییرات در سلسله مراتب ارث بری ، به دلیل انتشار تغییرات ، دشوار است

هنگامی که در تعریف ساختارهای توارثی ، از مکانیسم ارث بری به درستی استفاده نکنیم ممکن است دچار عوارضی همچون Refused Bequest شویم که یکی از نشانه های کد بد[1] است . تغییرات در ساختار سلسله مراتبی با دشواری همراه می شود وشاهد انتشار تغییرات خواهیم بود. همچنین عدم استفاده از ارث بری می تواند باعث ایجاد Duplication و پیچیده گی شود و انجام تغییر منجر به دوباره کاری و انتشار تغییرات خواهد شد.

۱-۱ استفاده از الگوی Extract Superclass

این الگو یکی از الگوهای بازآرایی[2] است که با کمک آن می توان برخی از انواع Duplication را برطرف نمود تا به این ترتیب اعمال تغییرات راحت تر شود.

۱-۱-۱ شرایط اعمال الگو

کلاسهایی داریم که دارای عملیات و صفات مشترکی هستند ولی فوق کلاس مشترکی ندارند . در نتیجه قبل از اعمال الگو هر یک از این کارکردها در همه این کلاسها تعریف شده است و شاهد Duplication خواهیم بود و تغییر در یکی از کارکردها باید در سایر کلاسها نیز اعمال شود . برای حذف Duplication می توان از الگوی Extract Superclass استفاده کرد .

یکی از مواردی که باعث دشوار شدن انجام تغییرات می شود این است که ساختار سلسله مراتبی و ارث بری ، چند کاره باشد و رشد درخت توارث بر اساس ترکیبی از حالتهای مختلف باشد. انجام تغییر در این حالت با دشواری همراه است زیرا اداره یک مسئله در سطح کلاسهای مختلف توزیع شده است . یکی دیگر از مسائلی که باعث می شود اعمال تغییرات در ساختار سلسله مراتبی مشکل باشد این است که سطوح انتزاعی به درستی تعریف نشده باشد. مثلا فرض کنید که کلاسهایی مثل ، دانشجوی کارشناسی ، دانشجوی ارشد ، دانشجوی دکتری ، استاد ، مسئول آموزش ، مسئول امور مالی و... همه ذیل یک فوق کلاس به نام کاربر تعریف شده باشد . در این حالت مشکل Refused Bequest می تواند ایجاد شود.

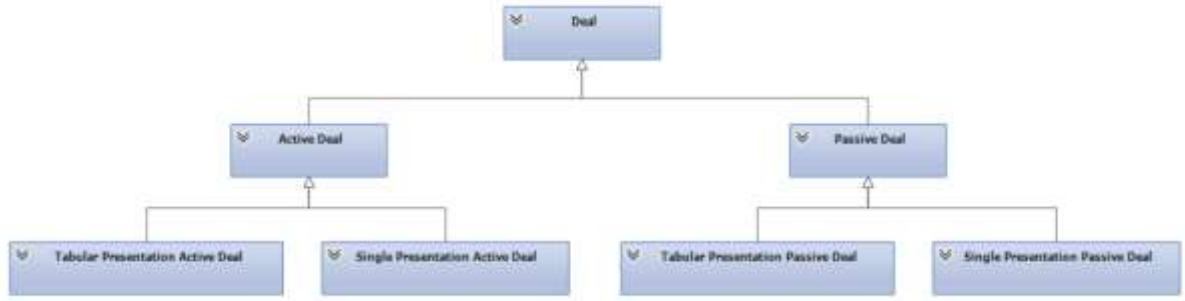
۱-۱-۲ نحوه حل مسئله

برای حل مشکل چند کاره بودن ساختار سلسله مراتبی ، نیازمند اعمال یک Big Refactoring هستیم . اگرچه راه حل این مشکل ، استفاده از الگوی Tease Apart Inheritance است اما در خلال اعمال این الگوی کلان می توانیم الگوی Extract Superclass را نیز به کار گیریم. در ادامه سعی می شود این مسئله را در قالب یک مثال توضیح

---

[1] Bad Smell
[2] Refactoring

دهیم. در اینجا نیاز داریم که یک ساختار سلسله مراتبی را به دو ساختار سلسله مراتبی تقسیم کرده و ارتباط بین آنها را از طریق Delegation برقرار کنیم.



شکل ۱- ساختار توارثی چند کاره

برای این منظور در ساختار سلسله مراتبی کار فرعی (تابع[۳]) را شناسایی می کنیم . در مثال بالا سبک نمایش اطلاعات یک کار فرعی است . در اینجا می توان دو کلاس برای هر یک از انواع سبکهای نمایشی ایجاد کرده (Tabular Presentation و Single Presentation) و برای آنها فوق کلاس مشترک ایجاد کرده و آنها را ذیل فوق کلاس Presentation Style تعریف کنیم . سپس با کمک Move Method ، متدهای مربوط به هر سبک را از ساختار قدیمی به کلاس جدید مرتبط به آن منتقل کنیم و سپس با استفاده از Pull up Method و Pull up Field و یا Pull up Constructor Body عملیات و ویژگی های مشترک را در فوق کلاس تعریف می کنیم . حال زیر کلاسهای فرعی را از درخت سلسله مراتب اصلی حذف کرده و ارتباط این دو درخت توارثی را از طریق Delegation برقرار می سازیم.

شکل ۲ - حل مشکل ساختار توارثی چند کاره

با اینکار هر یک از ساختار های توارثی پیاده سازی یک دغدغه را برعهده دارند و همچنین Duplicated Code ایجاد شده که ناشی از نیاز به اداره یک سبک از نمایش برای حالتهای مختلف است را از بین می بریم و با گذاشتن اعمال مشترک در فوق کلاس ایجاد شده می توانیم تغییرات را راحت تر مدیریت کنیم.

[3] subsidiary

مسئله عدم وجود یا تعریف سطوح مناسبی از انتزاع را نیز می توان با کک Extract Superclass حل کرد این مسئله را نیز در قالب یک مثال توضیح می دهیم. فرض کنید که کلاسهایی مثل ، دانشجوی کارشناسی ، دانشجوی ارشد ، دانشجوی دکتری ، استاد ، مسئول آموزش ، مسئول امور مالی و... همه ذیل یک فوق کلاس به نام کاربر تعریف شده باشد . در این حالت کلاسهای مرتبط با دانشجو دارای عملی مثل مشاهده کارنامه یا نمره هستند که در همه این سه کلاس یکسان است ولی برای کلاسهای مسئول امور مالی ، این ویژگی مورد نیاز نیست . در نتیجه این عمل را نمی توان در فوق کلاس کاربر تعریف کرد . در یک چنین شرایطی مسئله Refused Bequest می تواند پیش بیاید. برای حل این مسئله می توان کلاسهای دانشجوی کارشناسی ، ارشد و دکتری را بیرون کشید و با استفاده از Extract Superclass یک فوق کلاس مشترک به نام دانشجو برای آنها تعریف کرد و ارتباط بین کلاس کاربر و کلاس دانشجو را بر قرار کرد. رویکرد دیگر که چندان مورد پسند نیست این است که می توان فوق کلاس دانشجو رابه عنوان subclass کلاس "کاربر" تعریف کرد این مسئله باعث افزایش عمق درخت توارث می شود و پیچیده گی و مشکلات نگهداری را به وجود می آورد. مراحل اعمال الگوی Extract Superclass به شرح زیر است.

۱. ابتدا یک Superclass ایجاد می کنیم .
۲. کلاسهای موجود را به عنوان زیر کلاس های این Superclass تعریف می کنیم.
۳. با استفاده از Pull up Method , Pull up Field,Pull up Constructor Body ویژگی ها مشترک را به Superclass منتقل می کنیم.
   a. معمولا ساده تر این است که اول فیلدها و صفات را به بالا منتقل کنیم
   b. در برخی موارد متدهای موجود در Subclass ها ممکن است اهداف یکسانی داشته باشند ولی Signature آنها متفاوت باشد به همین دلیل قبل از استفاده از Pull up Method باید از Rename Method استفاده کرد.
   c. اگر Signature متدها یکسان بود ولی بدنه متد فرق می کرد ، می توانیم Signature مشترک را به عنوان Abstract Method در بالا تعریف کنیم.
۴. بعد از هر Pull up Method برنامه را Compile کنید.
۵. اگر در متدهای باقیمانده ، بخشهایی از بدنه مشترک است می توان از Extract Method استفاده کرد و بعد از آن Pull Up Method را انجام دهیم.
۶. در پایان ، مشترهایی که از کلاسهای قبلی استفاده می کردند را اصلاح کرده و دید آنها را به فوق کلاس برقرار می کنیم.

۱-۱-۳ مزایای به کارگیری این الگو

۱. مشکل Duplicated Code با استفاده از این الگو بر طرف می شود .

۲. با توجه به اینکه ویژگیهای مشترک در یک جا تعریف می شود تغییر دادن آنها راحت تر است و بر خلاف وضعیت قبلی ، انتشار تغییرات از این بابت نداریم. بنابراین مسئله Shotgun Surgery نیز به نوعی حل می شود

۳. با به کار گیری این الگو به همراه سایر الگوهای مورد نیاز ، ساختار سلسله مراتبی اصلاح می شود و از داشتن ساختارهای توارثی با عمق زیاد جلوگیری می شود. بنابراین خوانایی و قابلیت نگهداری ارتقاء می یابد

۴. در سناریوهای ذکر شده در بالا مسئله Refused Bequest وجود داشت که این مشکل نیز با ایجاد یک ساختار توارثی جدید با کمک Extract Superclass و جدا کردن دو ساختار توارثی و برقراری رابطه Delegation برطرف گردید.

۵. با کمک این الگو و ترکیب آن با سایر الگوهای بازآرایی ، می توان ساختارهای توارثی داشت که تک-کاره هستند و فقط یک جنبه را پوشش می دهند. در نتیجه پیچیدگی کاهش می یابد و امکان تغییر راحت تر می شود. مسئله Divergent Change نیز حل می شود زیرا هر ساختار سلسله مراتبی یک مفهوم منطقی واحد را پیاده سازی می کند و وظایف مختلف و نامرتبط بین کلاسها پراکنده نیست.

۱-۱-٤ معایب استفاده از این الگو

۱. هنگام استفاده از الگوی Extract Superclass باید مراقب بود که رابطه IS-A برقرار باشد.

۲. اگر ساختار توارثی از قبل وجود نمی داشت ، راحت تر می توانستیم در میان کلاسهای موجود ، با ایجاد یک فوق کلاس و بردن ویژگیهای مشترک به فوق کلاس ، مسئله Duplication و دشوار بودن اعمال تغییر را حل کنیم . اما در شرایطی که درخت سلسله مراتبی از قبل وجود دارد ، تشخیص کلاسهای به هم مرتبط که حول یک مفهوم منطقی بتوان آنها را جمع کرد و Extract Superclass را اعمال کرد دشوار است . معمولا این ساختارهای توارثی چندکاره هستند و در نتیجه یک کارکرد به ازای حالتهای مختلف پیاده سازی شده و در درخت توزیع شده است.

۳. معمولا در سناریوهای مطرح شده نیاز به یک بازآرایی در مقیاس بزرگ داریم که این الگو بخشی از آن است

۴. معمولا بعد از Extract Superclass ارتباط بین درخت سلسله مراتبی جدید و قدیم باید از طریق Delegation صورت گیرد که همین مسئله باعث افزوده شدن یک سطح Indirection و درنتیجه کاهش کارایی می شود.

۱-۲ استفاده از الگوی Replace Inheritance with Delegation

این الگو یکی از الگوهای بازآرایی است که زمانی مورد استفاده قرار می گیرد که یک زیر کلاس فقط بخشی از واسط فوق کلاس را مورد استفاده قرار می دهد و به همه داده ها و ویژگی های آن نیاز ندارد.

۱-۲-۱ شرایط اعمال الگو

گاهی اوقات یک زیر کلاس ، فقط به بخشی از واسط[۴] فوق کلاس خود نیاز دارد. در نتیجه اگر عملیات جدیدی در فوق کلاس تعریف شود که زیر کلاس به آن نیاز نداشته باشد ، مجبور است بدنه متد را در پائین خالی کند و Refused Bequest اتفاق می افتد . در این حالت بهتر است به جای ارث بری ، زیر کلاس را به عنوان مشتری سرویسهای فوق کلاس تعریف کنیم و از Delegation به جای Inheritance استفاده کنیم . رابطه ارث بری را از بین می بریم و در زیر کلاس متدهای مورد نیاز موجود در واسط فوق کلاس را تعریف می کنیم و انجام کار را از طریق Delegation ، به فوق کلاس واسپاری می کنیم. در این الگو پیاده سازی عمل در کلاس سرویس دهنده تعریف شده است و کلاس مشتری کار را واسپاری می کند . در نتیجه تغییر پیاده سازی عملیات ، تا زمانی که منجر به تغییر در واسط سرویس دهنده نشود ، به کلاس مشتری منتقل نمی گردد.

گاهی اوقات نیز ساختار سلسله مراتبی چند کاره است و در نتیجه اعمال تغییرات مشکل می شود . برای حل این مسئله یک ساختار سلسله مراتبی دو بعدی (و یا چند بعدی) ایجاد می کنیم که هر یک حول یک مفهوم یا کارکرد منطقی باشد. در این حالت کارکرد فرعی را در قالب یک ساختار توارثی جدید بیرون کشیده و به جای ارث بری چند سطحی ، ارتباط بین ساختار سلسله مراتبی باقیمانده و ساختار سلسله مراتبی جدید را از طریق Delegation برقرار می کنیم . در اینجا نیز واسپاری جایگزین ارث بری در ساختار سلسله مراتبی چند کاره شده است.

۱-۲-۲ نحوه حل مسئله

۱. به ازای هر متد موجود در Superclass که توسط مشتری مورد استفاده قرار می گیرد ، یک متد ساده برای واسپاری کار به سرویس دهنده تعریف می کنیم.
۲. در Subclass (مشتری) یک فیلد که به Superclass (سرویس دهنده) اشاره می کند تعریف می کنیم . به این شکل دید مشتری به سرویس گیرنده برقرار می شود
۳. متد موجود در Subclass را اصلاح می کنیم تا سرویس مورد نظر خود را از طریق فیلدی که تعریف شده است به کلاس سرویس دهنده واسپاری کند و سرویس مورد نظر را فراخوانی کند.

[4] Interface

۴. ساختار توارثی را با حذف رابطه ارث بری از بین می بریم و ارتباط بین مشتری و سرویس دهنده را از طریق Delegation برقرار می کنیم.
۵. برنامه را تست و کامپایل می کنیم.

۱-۲-۳ مزایای به کارگیری این الگو

۱. مشکل Refused Bequest در این حالت حل می شود و به جای ارث بردن یک سرویس ، مشتری استفاده از آن سرویس می شویم و هر موقع آنرا نیاز داریم فراخوانی می کنیم.
۲. مسئله اعمال تغییرات راحت تر می شود . زیر تعریف سرویسهای جدید در سرویس دهنده به کلاس مشتری منتقل نمی شود . در حالیکه قبل از استفاده از این الگو باید مسئله متدهایی که مورد نیاز زیر کلاسها نیستند را مدیریت می کردیم.
۳. در ساختارهای سلسله مراتبی چند کاره ، با شکستن و تبدیل آن به دو ساختار و بهره گیری از Delegation هر یک از این ساختارهای توارثی می توانند به طور مستقل رشد کنند و می توانیم DIP و OCP را برقرار کنیم.

۱-۲-٤ معایب استفاده از این الگو

۱. به دلیل استفاده از Delegation یک سطح Indirection ایجاد شده است که می تواند باعث کاهش کارایی شده و تعداد پیامهای رد و بدل شده را افزایش دهد .
۲. اگر کلاس مشتری ، به عنوان یک سرویس دهنده برای یک کلاس ثالث باشد باید مراقب بود که قانون دیمیتر نقض نشود .

۱-۳ مقایسه دو الگو

هر دو این الگوها اعمال تغییرات را راحت تر می کنند و انتشار تغییرات با اعمال آنها کاهش خواهد یافت . برای اعمال برخی از الگوهای درشت دانه مثل جدا سازی ساختارهای توارثی ، هر دوی این الگو ها را به طور همزمان باید مورد استفاده قرار دهیم. الگوی اول مسئله Duplicated Code را به شکل قابل توجهی حل می کند و الگو دوم مسئله Refused Bequest را حل می کند. استفاده از هر دو این الگوها می تواند به اصلاح ساختار سلسله مراتبی کمک کند.

۲ عملیاتهای یک کلاس به شدت وابسته به عملیات و صفات کلاس دیگری است

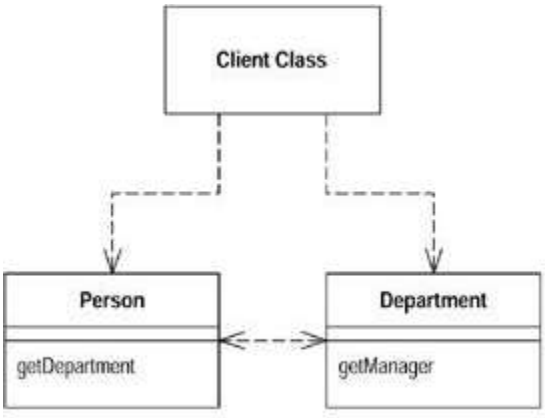
بالاخره حدی از وابستگی بین کلاسها وجود دارد اما آنچه که نـامطلوب اسـت وابسـتگی شـدید و ناخواسـته در اثـر عـدم رعایت اصول طراحی شی گرا و Encapsulation است . Inappropriate Intimacy یکی از نشـانه هـای کـد بـد است که باید تلاش کنیم با استفاده از الگوهای بازآرایی و رعایت قواعد شی گرایی آنرا رفع کنیم.

۲-۱ استفاده از الگوی Hide Delegate

این الگو یکی از الگوهای بازآرایی است که تلاش می کند با حـذف دیـد ترایـا و رعایـت قـانون دیمیتـر ، مـانع از نقـض Encapsulation شود.

۲-۱-۱ شرایط اعمال الگو

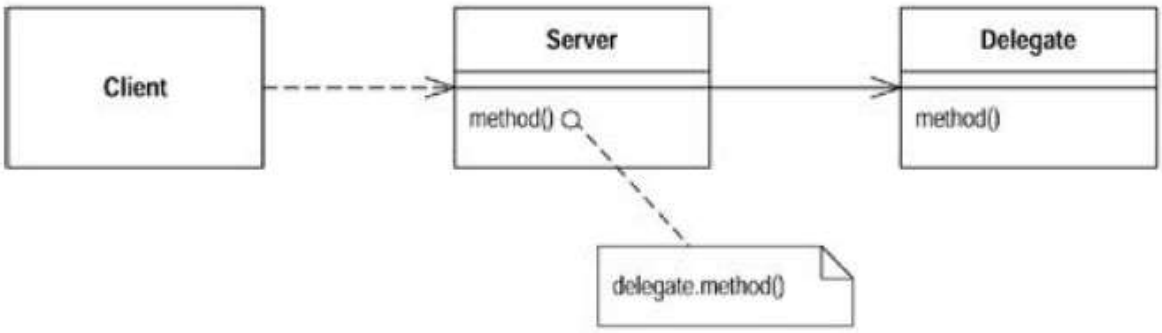
گاهی اوقات عدم رعایت Encapsulation باعث ایجاد وابستگی شدید به سایر کلاسها می شود در حالیکه می توان بـا یک طراحی بهتر این وابستگی را به نحو چشم گیری کاهش داد . رعایت Encapsulation به این منجـر مـی شـود کـه اشیاء نیاز به اطلاعات کمتری از سایر بخشهای سیستم داشته باشند و در نتیجه هنگامی که یک بخش از سیستم تغییـر کـرد ، تعداد کمتری از اشیاء از این تغییرات آگاه خواهند شد و بنابراین انتشار تغییـرات کمتـر اسـت و اعمـال تغییـر در سیسـتم راحـت تـر صـورت مـی گیـرد. بـه وجـود آمـدن دیـد ترایـا ، باعـث ایجـاد وابسـتگی مـی شـود کـه ناشـی از نقـض Encapsulation است . فرض کنید مشتری ، از طریق یکی از صفات سـرور ،بـه یـک Delegate Class دسترسـی پیدا می کند و به طور مستقیم درخواست خود را به Delegate Class ارسال می کند . این مسئله باعث می شـود کـه بـا تغییر Delegate Class نه تنها کلاسهایی که بـه طـور مسـتقیم بـه Delegate Class وابسـته بودنـد تغییـر کننـد بلکـه کلاس مشتری که دید آن از طریق مقدار برگشتی برقرار شده است نیز دچار تغییر شود . دلیـل ایـن وابسـتگی نـامطلوب، نقض قانون دیمیتر و Encapsulation است . شکل زیر نحوه نقض قانون PLK را نشان می دهد.



شکل ۳-نقض قانون PLK

۲-۱-۲ نحوه حل مسئله

راه حل این است که Indirection را به شکل درست پیاده سازی کنیم به نحوی که اجازه نقض قانون PLK را نداشته باشیم . برای این کار ، متدهای موجود در Delegate Class را که مورد نیاز مشتری می باشد ، در سرور تعریف می کنیم و کلاس مشتری را فقط به کلاس سرور وابسته می کنیم و امکان دسترسی مستقیم به Delegate Class را به مشتری نمی دهیم. بنابراین مشتری متد موجود در سرور را فراخوانی می کند و سرور نیز با دریافت درخواست مشتری ، انجام کار را به Delegate Class ، واسپاری می کند . این کار باعث از بین رفتن دید ترایا می شود و Encapsulation برقرار می گردد. با اینکار ، تغییر در Delegate Class به مشتری منتقل نمی شود و حداکثر کلاس سرور ممکن است دچار تغییر شود. شکل زیر نحوه اعمال این الگو را نشان می دهد.



شکل ۴- نتیجه اعمال الگوی Hide Delegate

۲-۱-۳ مزایای به کارگیری این الگو

۱. Shotgun Surgery یکی از نشانه های کد بد قبل از اعمال الگو در برنامه بود که با به کار گیری الگوی Hide Delegate بر طرف شده است

۲. Message Chains یا زنجیره دید ترایا یکی دیگر از نشانه های کد بد در برنامه بود که با اعمال الگو از بین رفت.

۳. قبل از به کار گیری این الگو ، وابستگی کلاس مشتری به Delegate Class به طور مستقیم بود و در نتیجه اعمال تغییرات در آن باعث انتشار تغییرات به کلاس مشتری می شد . با اعمال این الگو ، وابستگی به طور غیر مستقیم شده است که برای ما مطلوب تر است و همچنین جلوی انتشار تغییرات به کلاس مشتری گرفته شده است.

۴. الگوی Indirection در GRASP در اینجا دیده می شود و به درستی پیاده سازی شده است و به دلیل غیر مستقیم شدن وابستگی ، الگوی Low Coupling در GRASP را نیز می توانیم مشاهده کنیم.

۲-۱-٤ معیاب استفاده از این الگو

۱. استفاده از این الگو باعث ایجاد یک سطح Indirection شده است و در نتیجه تعداد پیامهای رد و بدل شده نسبت به قبل افزایش یافته است و همین مسئله باعث افت کارایی می شود.

۲. همه متدهای Delegate Class که مورد نیاز مشتری است باید در سرور تعریف شود زیرا مشتری فقط به سرور دید دارد و درخواستهای مشتری از طریق سرور به Delegate Class واسپاری می شود. این مسئله باعث سنگین شدن و آلوده شدن واسط سرور با این متدها خواهد شد و می تواند بر روی چسبندگی اثر منفی بگذارد.

۲-۲ استفاده از الگوی Extract Interface

این الگو یکی از الگوهای بازآرایی است که تلاش می کند اشتراک گذاری در سطح واسط را محقق کند تا بتوان وابستگی ها را در سطح واسط تعریف کرد.

۲-۲-۱ شرایط اعمال الگو

مشتری نیاز دارد تا با مجموعه ای از کلاسها تعامل داشته باشد و یک سری سرویس مشخص که توسط همه آنها ارائه می شود را دریافت کند و از نظر کلاس مشتری فرقی نمی کند که کدام یک از این کلاسها برای سرویس گیری در اختیار او قرار گیرد.در نتیجه تغییر کلاس سرویس دهنده ، باعث انتشار تغییرات به کلاس مشتری خواهد شد زیرا عملا وابستگی مشتری به کلاسهای سرویس دهنده در سطح Concrete تعریف شده است. همچنین ممکن است بخشی از واسط یک کلاس ، مورد استفاده گروهی از کلاسهای مشتری قرار گیرد. در این حالت نیز تغییر کلاس سرویس دهنده ، همه مشتریان آن کلاس را تحت تاثیر قرار خواهد داد . گاهی اوقات می خواهیم دید به یک کلاس را محدود کنیم و تنها اجازه استفاده از سرویسهای مشخصی از سرویس دهنده را به مشتری بدهیم تا به این ترتیب ، مشتری به سرویسهایی که به آن نیاز ندارد دید نداشته باشد و وابستگی ناخواسته ایجاد نشود

۲-۲-۲ نحوه حل مسئله

برای جلوگیری از انتشار تغییرات باید وابستگی مشتری به کلاس یا کلاسهای سرویس دهنده را در سطح واسط تعریف کنیم . این مسئله باعث خواهد شد که مشتری فقط به واسط وابستگی داشته باشد و عملا با هر یک از کلاسهای سرویس دهنده که آن واسط را محقق می کند ، بتواند کار کند. همچنین تغییر در کلاسهای سرویس دهنده به مشتری منتقل نخواهد شد زیرا این کلاسها باید واسط مربوطه را محقق نمایند و تغییرات داخلی آنها به بیرون منعکس نمی شود زیرا واسط با تعریف مرز ، دنیای بیرون را از تغییرات داخلی منتزع کرده است. بنابراین Extract Interface ، مکانیسمی است که در این راه به ما کمک می کند.

هر چه دانش ما نسبت به یک کلاس کمتر باشد ، میزان وابستگی ما به آن کلاس نیز کمتر خواهد بود. بنابراین اگر یک کلاس مشتری فقط نیازمند بخشی از سرویس های یک کلاس دیگر است ، باید دید او را نسبت به کلاس سرویس دهنده محدود کنیم و همه سرویسهای ارائه شده را در دسترس او قرار ندهیم. یکی از روشهای محدود کردن دید ، استفاده از واسط است و در نتیجه الگوی Extract Interface می تواند در این زمینه به ما کمک کند.

برای اعمال این الگو ، ابتدا یک واسط خالی ایجاد می کنیم و سپس همه عملیاتهای مشترک بین کلاسهای سرویس دهنده را در آنجا معرفی می کنیم . کلاسهای سرویس دهنده مسئولیت پیاده سازی واسط را دارند و آنرا محقق می کنند. در نهایت ، وابستگی کلاس مشتری را نیز در سطح واسط تعریف می کنیم.

همچنین برای محدود کردن دید مشتری به برخی از متدهای یک کلاس ، واسط خالی را ایجاد کرده و آن دسته از متدها و سرویسهایی را که می خواهیم در اختیار مشتری قرار دهیم را در آن واسط ، اعلان می کنیم و کلاس سرویس دهنده را به عنوان پیاده ساز واسط ، معرفی می کنیم تا آنرا محقق کند. در پایان ، دید مشتری برای سرویس گیری را به واسط برقرار می کنیم.

۲-۲-۳ مزایای به کار گیری این الگو

۱. با بکار گیری این الگو می توانیم DIP را بر قرار کنیم و وابستگی مشتری را در سطح واسط تعریف کنیم. در نتیجه تغییر در کلاسهای سرویس دهند باعث انتشار تغییرات به مشتری نخواهد شد و بنابراین مشکل Shotgun Surgery را نخواهیم داشت.

۲. امکان افزوده شدن سرویس دهنده های جدید که آن واسط را محقق کنند نیز وجود دارد و مشتری بدون نیاز به تغییر می تواند با آنها کار کند و در نتیجه OCP نیز برقرار است.

۳. برای محدود کردن دید به یک کلاس ، امکان تعریف چندین واسط وجود دارد . در نتیجه این قابلیت وجود دارد که بخشی از سرویسهای یک کلاس را به یک مشتری نشان دهیم در حالیکه یک مشتری دیگر به آن سرویسها دسترسی ندارد و آنها را نمی بیند .

۲-۲-٤ معایب به کار گیری این الگو

۱. اشتراک گذاری فقط در سطح واسط وجود دارد ، اما اگر یک متد یا بخشی از آن ، در همه سرویس دهنده ها یکسان باشد ، نمی توان در سطح پیاده سازی[5] اشتراک گذاری داشت. در نتیجه در این شرایط Duplication وجود دارد که می تواند باعث انتشار تغییرات بین کلاسهای سرویس دهنده شود.

[5] Implementation

۲. یکی از مشکلاتی که در هنگام اعمال این الگو پیش می آید این است که کلاسهای مختلفی که می خواهیم برای آنها واسط تعریف کنیم ، دارای متدهایی با کارکرد یکسان ولی Signature متفاوت هستند . برای استفاده از این الگو باید بتوان با تغییر نام متدها ، به یک Signature یکسان دست یافت.

۲-۳ مقایسه دو الگو

در هر دو این الگو ها به نوعی Indirection قابل مشاهده است و این مسئله در مورد الگوی اول به روشنی مشخص است . در هر دوی این الگوها توصیف در قالب یک واسط به مشتری ارائه شده است در حالیکه پیاده سازی اصلی عمل در جایی دیگر است . در الگوی اول این کار از طریق واسپاری صورت می گیرد و در الگوی دوم توسط کلاسی که آن واسط را محقق می کند . در هر دوی این الگو ها وابستگی کلاس مشتری به عملیات کلاسهای دیگر کاهش می یابد زیرا در الگوی اول با تغییر Delegate Class فقط ممکن است سرور تغییر کند و کلاس مشتری از این مسئله با خبر نمی شود . در الگوی دوم نیز تغییرات داخلی یک کلاس از دید بیرون مخفی می ماند زیرا آن کلاس همچنان واسط را محقق کرده و وابستگی مشتری نیز در سطح واسط تعریف شده است. درمجموع هر دوی این الگوها وابستگی که از دید ما مورد پسند نیست را کاهش داده و مانع از انتشار تغییرات به کلاس مشتری می شوند.

۳ بعضی از کلاسها ، Data Class هستند

Data Class یکی از نشانه های کد بد است و به این دلیل ایجاد می شود که داده از رفتار خود دور مانده است . برای حل این مسئله باید الگوی Information Expert محقق شود و برای این کار از مکانیسم Move Behaviour Close to Data استفاده می کنیم.

۳-۱ استفاده از الگوی Split Up God Class

Split Up God Class یکی از الگوهای بازمهندسی[6] است که برای توزیع مجدد مسئولیتها[7] پیشنهاد می شود. هدف این الگو بهبود Cohesion است که از طریق شکستن God Class و بردن رفتار به کنار داده مربوطه محقق می شود.

۳-۱-۱ شرایط اعمال الگو

اغلب موارد وقتی در یک سیستم Data Class داریم ، باید انتظار داشته باشیم که یک جای دیگر حتما God Class ایجاد شده است . در این حالت رفتارها و مسئولیتها در God Class جمع شده اند و از داده مورد نیاز خود دور مانده اند. بنابراین برای حل مشکل Data Class می توانیم God Class های موجود را شناسایی کرده و رفتارهایی که دور از داده خود قرار گرفته اند را به Data Class مربوطه منتقل کنیم .

۳-۱-۲ نحوه حل مسئله

اولین گام برای حل مسئله ، شناسایی God Class است . به طور کلی God Class کلاسی است که حاوی تعداد زیادی مسئولیت است و کنترل کلی و محاسبات مورد نیاز در برنامه را در اختیار دارد و به نوعی قلب سیستم محسوب می شوند. رهنمود های زیر می تواند به ما در شناسایی God Class کمک کند.

۱. کلاسهای بسیار بزرگ می توانند به عنوان کاندیدای God Class بودن ، مورد بررسی قرار گیرند.
۲. کلاسهایی که در عنوان آنها ، عباراتی نظیر "System" ، "Subsystem" ، "Manager" ، "Controller" ، "Driver" دیده می شود در معرض God Class شدن قرار دارند و باید بررسی شوند.
۳. تغییرات در سیستم که گاهی نامرتبط هم هستند ، منجر به تغییر یک یا دسته ای خاص از کلاسها می شود . آنها به احتمال زیاد God Class هستند.
۴. استفاده مجدد از یک کلاس بسیار دشوار است زیرا دغدغه های متعدد طراحی در آنها وجود دارد . احتمالا این کلاس یک God Class خواهد بود.

---

[6] Reengineering
[7] Redistribute Responsibility

۵. God Class شامل تعداد زیادی متد غیر مرتبط است که بر روی مجموعه های جداگانه ای از متغییر ها کار می کنند.

۶. تست کردن این کلاسها دشوار است و زمان کامپایل آنها ، حتی اگر تغییرات کوچک باشد ، زیاد است.

بعد از شناسایی God Class باید به صورت گام به گام ، با استفاده از Move Method رفتار را از God Class به سمت کلاسی که حاوی داده مورد نیاز آن رفتار است ببریم . به این صورت Data Class ها نیز حاوی رفتار خواهند شد. همچنین ممکن است از دل God Class ، کلاس دیگری بیرون کشیده شود . بنابراین برای اعمال الگوی Split Up God Class ، گامهای زیر را انجام می دهیم :

۱. مجموعه ای از متغییرها که از چسبندگی مطلوبی برخوردانـد را از درون God Class انتخاب کرده و به صورت یک Data Container بیرون می کشیم.

۲. همه کلاسهایی که God Class از آنها در قالب Data Container استفاده می کند را شناسایی کرده و با استفاده از Move Behaviour Close to Data و مکانیسم بازآرایی Move Method ، رفتار را در سمت داده آن تعریف می کنیم . متد باقیمانده در سمت God Class می تواند فقط نقش واسپاری کار را بر عهده بگیرد و اجرای کار را به کلاس مربوطه محول کند.

۳. با اعمال مکرر گامهای ۱ و ۲ ، آنچه که از God Class باقی می ماند ، یک Facade است که در صورت عدم نیاز می توان آنرا حذف کرد که در این صورت نیازمند اصلاح کلاسهای مشتری هستیم زیرا آنها قبلا به God Class وابسته بوده اند.

۳-۱-۳ مزایای استفاده از این الگو

۱. داشتن Data Class به گونه ای که رفتار آن در جایی دیگر تعریف شده باشد ، یکی از نشانه های کد بد است که با شکستن God Class و بردن رفتار نزد داده آن، این مسئله حل می شود.

۲. قبل از اعمال این الگو ، به دلیل وجود God Class به احتمال زیاد شاهد متدهای طولانی خواهیم بود که یکی از نشانه های کد بد است و با اعمال این الگو رفع می شود

۳. God Class ها معمولا کلاسهای بزرگی هستند و کلاسهای بزرگ یکی از نشانه های کد بد است که با شکستن God Class این مسئله رفع می شود

۴. معمولا در God Class ، متدها برای انجام عملیات خود داده ها را در قالب پارامتر مورد استفاده قرار می دهند. این مسئله می تواند منجر به لیست طولانی از پارامترها شود که یکی از نشانه های کد بد است . با شکستن God Class و بردن رفتار به نزد داده مربوطه ، عملا نیاز به پارامتر منتفی می شود و لیست طولانی پارامتر ها از بین می رود.

۵. به دلیل افزایش چسبندگی ، Divergent Change که یکی دیگر از نشانه های کد بد است را نخواهیم داشت و کلاسها به دلایل مختلف و بی ربط عوض نمی شوند

۶. مسئله Feature Envy حل می شود زیرا قبل از اعمال این الگو ، متدهای موجود در God Class برای انجام عملیات مربوطه نیازمند داده ها و صفاتی بودند که در Data Class ها تعریف شده بود.

۷. Data Clumps نیز بر طرف می شود زیرا دسته ای از متغییرهایی که در God Class معمولا با هم مورد استفاده قرار می گیرند و به عنوان پارامتر به متدهای این کلاس پاس می شوند ، عملا در قالب یک کلاس جداگانه بیرون کشیده شده و رفتار آنها نیز در کنار آنها تعریف می شود

۸. با برطرف شدن Data Class ، مسئله Lazy Class که یکی از نشانه های کد بد است نیز بر طرف می شود.

۹. پیچیدگی کلی کاهش می یابد و خوانایی و قابلیت نگهداری و استفاده مجدد افزایش می یابد.

۱۰. Cohesion بهبود می یابد.

۳-۱-٤ معایب استفاده از این الگو

۱. فرایند شکستن God Class فرایندی زمانبر و خسته کننده است.

۲. تعداد کلاسهای موجود افزایش می یابد.

۳. به دلیل عدم وجود تمرکز و توزیع رفتار بین کلاسهای مختلف ، مسئول نگهداری دیگر نمی تواند برای حل مشکل فقط به یک محل رفته و اصلاحات را انجام دهد.

۳-۲ استفاده از الگوی Eliminate Navigation Code

این الگو یکی از الگوهای بازمهندسی است که برای توزیع مجدد مسئولیتها پیشنهاد می شود و هدف آن این است که با کاهش وابستگی بین زنجیره ای از کلاسهای به هم مرتبط ، اثرات تغییر در این زنجیره را به حداقل برساند. گاهی اوقات تغییر یک کلاس ، نه تنها مشتریان مستقیم آن کلاس را تحت تاثیر قرار می دهد بلکه عدم رعایت Encapsulation و وجود دید ترایا درزنجیره ای از کلاسهای در حال تعامل ، باعث می شود که مشتریانی که به طور غیر مستقیم و از طریق پیمایش[8] صفات و متد های دسترسی[9] عمومی[10] به آن کلاس دید پیدا کرده اند نیز دچار تغییر شوند.

۳-۲-۱ شرایط اعمال الگو

گاهی اوقات اگر ریشه به وجود آمدن Data Class را در سیستم بررسی کنیم مشاهده خواهیم کرد که زنجیره ای از کلاسها وجود دارد به نحوی که رفتار ها در ابتدای زنجیره تعریف شده است و Data Class ها در نقش تامین کننده

[8] Navigation
[9] Access method
[10] Public

داده قرار دارند. این داده در طول زنجیره از سمت Data Class ، حرکت می کند و در ابتدای زنجیره رفتار مورد نظر بر روی داده اعمال می گردد . علت وقوع این رخداد نقض Encapsulation است که باعث ایجاد دید ترایا شده است. در واقع هنگام تعریف یک عملیات ، به جای اینکه عملیات مربوطه را بشکنیم و هر تکه عملیات را نزدیک داده خود قرار دهیم ، به دلیل تنبلی ، رفتار را در نزدیک ترین جایی که می شود قرار داد ، قرار می دهیم و سپس داده ها را از کلاسهای مختلف جمع آوری کرده و عملیات مورد نظر را روی آن انجام می دهیم . این مسئله باعث می شود که در سیستم شاهد کلاسهای داده رفتار باشیم که خالی از متد هستند در حالیکه عملیاتهای مربوط به آن داده ها در جای دیگری از سیستم تعریف شده است. برای حل این مشکل می توان از الگوی Eliminate Navigation Code استفاده کرد که در بخش بعدی به توضیح آن می پردازیم

۳-۲-۲ نحوه حل مسئله

برای حل این مسئله باید رفتار را نزدیک داده خودش بیاوریم و این با بهره گیری از ایده موجود در الگوی Move Behaviour Close to Data صورت می گیرد . گاهی اوقات این داده در طول یک زنجیره از کلاسها حرکت می کند تا به سر زنجیره برسد . بنابراین باید به صورت یک فرایند تکراری الگوی Move Behaviour Close to Data را اجرا کنیم و رفتار را در طول زنجیره توزیع کرده و در هر مرحله رفتار را یک گام به داده نزدیک تر کنیم . این دقیقا همان کاری است که الگوی Eliminate Navigation Code انجام می دهد که سعی می کند از طریق تکرار الگوی Move Behaviour Close to Data زنجیره دید ترایا را از بین ببرد.

برای اعمال این الگو باید Data Class ها را به عنوان تامین کننده داده شناسایی و مشخص کنیم . سپس مشتریانی که به طور غیر مستقیم و از طریق Navigation Code به این Data Class ها دسترسی دارند را شناسایی و مشخص می کنیم. به طور کلی Navigation Code به دو صورت ظاهر می شود :

۱. از طریق دنباله ای از صفات عمومی و قابل دسترسی که به صورت a.b.c.d خود را نشان می دهد. در این حالت ، b صفت a است ، c صفت b است و d صفت c است .
۲. از طریق دنباله ای از فراخوانی متدهای دسترسی که به صورت object.m1().m2().m3() خود را نشان می دهد. خروجی هر یک از متدهای m1,m2,m3 یک شی است.

اگر آخرین حلقه زنجیره یک شی داده رفتار[11] باشد ، آنگاه می توان با دسترسی به صفات آن ، عملیات مورد نظر را روی داده ها انجام داد و مقادیر جدید را جایگزین مقادیر قبلی صفات کرد. همین مسئله نشان دهنده چگونگی نقض Encapsulation و همچنین قانون دیمیتر است.

بنابراین در Eliminate Navigation Code گامهای زیر را به صورت تکرار شونده انجام می دهیم

۱. Navigation code را شناسایی کنید
۲. با به کارگیری Move Behaviour Close to Data ، یک سطح از Navigation Code را حذف کنید .
۳. در صورت نیاز این فرایند را تکرار کنید.

۳-۲-۳ مزایای استفاده از این الگو

۱. با حذف زنجیره دید ترایا ، وابستگی بین کلاسها کاهش یافته و Encapsulation بهبود یافته است و Message Chains که یکی از نشانه های کد بد است از بین رفته است.
۲. به دلیل کاهش وابستگی و انجام کار از طریق مکانیسم واسپاری ، Shotgun Surgery و انتشار تغییرات به کلاسهایی که به طور غیر مستقیم از یک کلاس دیگر سرویس می گیرند را شاهد نخواهیم بود
۳. با قرار گرفتن رفتار در کنار داده مربوطه ، چسبندگی بهبود می یابد و در نتیجه کلاسها به دلایل بی ربط تغییر نخواهند کرد.
۴. Feature Envy به دلیل توزیع رفتار در طول زنجیره و قرار گرفتن رفتار و داده مورد نیاز در کنار هم ، رفع خواهد شد
۵. Data Class ها صاحب رفتار مرتبط با داده می شوند و نشانه کد بد Lazy Class بهبود می یابد.

۳-۲-٤ معایب استفاده از این الگو

۱. به کارگیری Delegation در این الگو باعث ایجاد سطوح Indirection می شود و این مسئله حجم پیامهای رد و بدل شده را افزایش می دهد و می تواند باعث کاهش کارایی گردد.
۲. این یک فرایند گام به گام است که انجام آن زمان بر و سخت تر از حالت قبل از به کارگیری این الگو است . در این الگو نیاز به شکستن رفتار و توزیع آن در طول زنجیره داریم. به همین دلیل تنبلی یکی از عوامل بروز این مشکلات است که باعث ایجاد زنجیره دید ترایا Data Class می شود.

[11] Data Class

۳-۳ مقایسه دو الگو

در هر دو الگو سعی کردیم که Information Expert را محقق کنیم و رفتار در کنار داده مربوط به آن قرار گیرد. مبنای هر دو الگو ، Move Behaviour Close to Data است . اعمال هر دو الگو یک فرایند تدریجی است که به مرور باعث می شود کلاسهای داده ای حاوی رفتار مربوط به خود شوند . این الگوها باعث بهبود Cohesion می شوند. هر دو الگو ، نشانه های به وجود آمدن Data Class را شناسایی کرده و در پی حل ریشه بروز مشکل هستند.