به نام او

درس الگوها در مهندسی نرم‌افزار

استاد: دکتر رامان رامسین

تمرین سوم

پدرام شاطری - ۴۰۰۲۱۱۳۹۸

نیمسال دوم ۰۰-۰۱

فهرست مطالب

بررسی موقعیت اول

همان‌طور که از اسم این موقعیت پیدا است ما در این موقعیت, ساختار توارثی یا بهتر است بگوییم کلاس‌های پدر و فرزندی داریم که فرزندها به تعهدات پدر عمل نمی‌کنند, یعنی رابطه is-a نداریم و بیشتر برای Reuse از توارث استفاده کرده‌ایم.

یعنی فرزند یک Attribute یا Operation از پدر را نمی‌خواهد, یا ممکن است مثلاً پیش‌شرط‌ها را قوی تر یا پس شرط‌ها را ضعیف‌تر می‌کنند. (این مصداق یک Refused Bequest است).

انتشار تغییرات در این شرایط بسیار زیاد است, ممکن است Operationهایی در پدر تعریف کنیم و در فرزند این عملیات را خالی کنیم. بیشتر برای رفع این مشکلات در ساختار توارثی مثلاً رابطه بین پدر و فرزند رابطه Delegation تعریف می‌کنیم و پدر Server و فرزند Client باشد.

این Bad Smell باعث می‌شود ساختار Gen/Spec به هم بریزد, هم چنین باعث نقض LSP می‌شود. (زیرا با هر تغییر در پدر باید تغییرات را در فرزندان اعمال کنیم, به‌این‌ترتیب یک نمونه از فرزند نمی‌تواند به‌جای نمونه‌ای از پدر استفاده شود.)

الگوی اول: جایگزینی فوق کلاس با واسپاری

بررسی الگوی اول

این الگو در شرایطی استفاده می‌شود که Refused Bequest داریم و LSP برقرار نیست و همان‌طور که در بالا اشاره شد, باید توارث را از بین ببریم و رابطه Client-Server با استفاده از واسپاری ایجاد کنیم. در واقع هرجا که مشکل Refused Bequest داریم دچار مشکل اسب و عنکبوت شده‌ایم. فوق کلاس, یک کلاس جداگانه می‌شود, سپس سرویس یا رفتاری که در زیر کلاس به ارث می‌رسیده است, حال از طرف زیر کلاس که یک کلاس جداگانه است این سرویس را درخواست می‌کند و به‌عنوان یک کلاینت برای اوست و کار را از طریق Delegation انجام می‌دهد.

می‌توان گفت وقتی رابطه is-a نداریم, این ساختار توارثی توجیهی برای نگه داشته شدن ندارد و اگر از طریق واسپاری انجام شود از لحاظ Maintainability سود خواهیم داشت.

زمان اعمال الگو

این الگو که قبلاً در کتاب 1999 با نام Replace Inheritance With Delegation شناخته می‌شود, زمانی می‌تواند به ما کمک کند که در واقع نگهداری یک ساختار توارثی به دلیل نقض is-a توجیهی ندارد, زیرا که با هر تغییر در فوق کلاس تغییرات به زیر کلاس منتشر شده و باید عملیات یا فیلدهایی که در زیر کلاس نیاز نیست را به‌صورت خالی پیاده‌سازی کنیم (مصداق Refused Bequest). در چنین شرایطی LSP نیز نقض شده است, زیرا Instanceهای کلاس فرزند به‌جای Instanceهای کلاس پدر نمی‌توانند

استفاده شوند. در این شرایط اگر کار یا سرویس به‌صورت واسپاری به زیر کلاس برسد, انتشار تغییرات و در نتیجهٔ آن Maintainability افزایش پیدا می‌کند.

خود Fowler می‌گوید "حتی در زمان‌هایی که یک Subclass توجیه وجودی و مدلینگ درست دارد, از واسپاری به‌جای توارث استفاده می‌کند, زیرا که رابطه فوق کلاس و کلاس‌هایی که از آن ارث می‌برند بسیار Coupled هست و هم تغییرات در فوق کلاس به زیر کلاس منتشر می‌شود, اما مشکلی که این کار دارد این است که برای متدهایی که هم در فوق کلاس و هم در زیر کلاس یکسان هستند باید به‌اصطلاح Forwarding Functionهایی در زیر کلاس بنویسد تا آن‌ها را صدا کند, اما این کار بسیار ساده است و خیلی احتمال خطای پایینی دارد."

چگونگی مفید بودن الگو

ازآنجایی‌که ما Refused Bequest که یک Bad Smell هست را برطرف کرده‌ایم, هم چنین دیگر LSP و رابطهٔ is-a بین فوق کلاس و زیر کلاس را نقض نمی‌کنیم, این الگو می‌تواند مفید باشد. هم چنین از نظر Maintainability هم به ما کمک می‌کند, زیرا که تغییرات منتشر شوند بین فوق کلاس و زیر کلاس (که مثلاً زیر کلاس, یک سری از عملیات یا Attributeها را نمی‌خواهد) نخواهیم داشت.

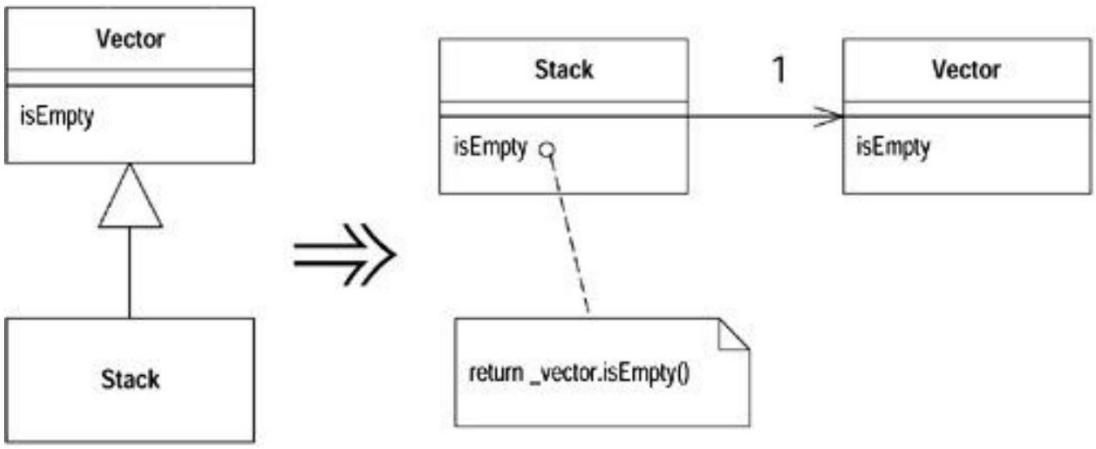
حال دیگر این زیر کلاس فقط متدها و عملیات ها یا حتی Attributeهایی که درون خود دارد که با آن‌ها کار می‌کند یا به آن‌ها احتیاج دارد.

چگونگی پیاده‌سازی یا اعمال الگو

۱. یک فیلد در زیر کلاس می‌سازیم که یک رفرنس به فوق کلاس دارد و آن را از طریق متد سازنده مقداردهی می‌کنیم.
۲. متدهای زیر کلاس را تغییر می‌دهیم تا به‌جای استفاده از متدهای فوق کلاس با استفاده از کلمه کلیدی this از فیلد مذکور در بالا استفاده کند.
۳. برای متدهایی که در کد کلاینت وجود دارد و از طریق زیر کلاس, در حال استفاده از متدهای به ارث رسیده در فوق کلاس است, متدهایی به‌اصطلاح Forwarding Functions می‌سازیم تا از طریق همان فیلد مذکور و با Delegation این کار را انجام دهند.
۴. ارث‌بری را از بین می‌بریم.

مثال

مثلاً برای استفاده از بعضی متدهای یک Vector ما این کلاس را به‌عنوان فوق کلاسی برای Stack در نظر گرفته‌ایم تا از بعضی از متدهای آن استفاده کنیم (استفاده از توارث برای Reuse و نه Is-a). به‌جای نگه‌داشتن این ساختار توارثی, باید کار را به‌صورت واسپاری انجام داد و Stack از Vector سرویس بگیرد. (رابطه Client-server)
همان‌طور که گام‌های انجام این الگو در بالا توضیح داده شد, می‌توان وضعیت قبل و بعد اعمال این الگو را مشاهده خواهید کرد.



شکل ۱

مزایای اعمال این الگو

۱. دیگر زیر کلاس‌ها متدهایی که استفاده نمی‌کنند را نخواهند داشت.

۲. می‌توان پیاده‌سازی‌های مختلف از فوق کلاس را در فیلدی که در زیر کلاس قراردادیم بگذاریم و در نتیجهٔ آن الگوی طراحی Strategy خواهیم داشت.

۳. دیگر LSP را نقض نمی‌کنیم و در صورت توسعهٔ هرکدام از زیر کلاس‌ها یا فوق کلاس‌ها (به‌صورت درست و داشتن رابطهٔ is-a) این اصل را خواهیم داشت.

۴. انتشار تغییرات از فوق کلاس به زیر کلاس قطع می‌شود.

۵. Maintainability بالا می‌رود زیرا تغییرات از فوق کلاس به زیر کلاس منتشر نمی‌شود.

۶. برقرار سازی CRP.

۷. چون از اینترفیس کلاس Server استفاده می‌کنیم, DIP و در نتیجهٔ آن OCP حفظ شده است.

۸. کاهش Coupling شدید بین زیر کلاس و فوق کلاس (این گفتهٔ خود Fowler است.).
۹. این الگو Code Smellهایی مثل Refused Bequest و یا Divergent Change (به دلیل Refused Bequest تغییرات بی‌دلیل به زیر کلاس منتشر خواهد شد) را از بین می‌بریم.

معایب اعمال این الگو

۱. سربار پیام‌ها به‌خاطر واسپاری زیاد می‌شود.
۲. ممکن است تعداد زیادی Forwarding Function که بسیار ساده است و فقط کار را واسپاری می‌کند خواهیم داشت.

الگوهای مرتبط

۱. می‌تواند پیاده ساز برای الگوی طراحی Strategy باشد.
۲. ضد الگوی بازآرایی این الگو Replace Delegation With Inheritance/Superclass است.

الگوی دوم: استخراج اینترفیس

بررسی الگوی دوم

این الگو یعنی استخراج اینترفیس دو کاربرد خواهد داشت. محدودکردن دید (یعنی یک زیرمجموعه Operationهای خود را در اختیار یک سری از کلاینت‌ها و زیرمجموعه دیگر رو در اختیار دیگر کلاینت‌ها قرار می‌دهد). اگر قرار باشد کل اینترفیس رو در اختیار همه کلاینت‌ها قرار دهیم, Coupling زیاد می‌شود و هم چنین احتمال بروز فراخوانی غلط وجود خواهد داشت. محدودکردن دید Coupling را کم خواهد کرد و با تعریف چندین اینترفیس روی کلاس می‌توان این کار را کرد. این موضوع می‌توان در حوزه کلاینت‌هایی با نیازهای متفاوت برقرار باشد.

حالت دیگر اشتراک‌گذاری اینترفیس بین کلاس‌هاست, یعنی ممکن است دو کلاس اینترفیس مشابه, یا تعدادی از عملیات هایی که دارند یکسان باشد (بهتر است بگوییم دو پیاده‌سازی از یک عملیات را دارند) و می‌توان هرکدام از این کلاس‌ها را در اختیار کلاینت قرار داد و کلاینت با همین اینترفیس با آن‌ها کار خواهد کرد. یعنی نمی‌داند کدام کلاس در اختیار او هست و فقط با اینترفیس آن کار می‌کند.

در هر دو حالت باعث کاهش Coupling می‌شود.

زمان اعمال الگو

اگر از این الگو صرفاً برای کم‌کردن Coupling یا برقراری LSP استفاده شود کمک خاصی نمی‌تواند بکند, اما اگر در این Context خاص که می‌خواهیم در واقع یک Code Smell که اینجا Refused Bequest هست را از بین ببریم, می‌توان با تولید اینترفیس‌های

Narrow و Wide ایـــن کـــار را انجـــام دهـــیم. یعنـــی یـــک اینتـــرفیس Narrow بـــرای Operationهـایی کـه هـم در Superclass و هـم در Subclass هسـت تعریـف کنـیم و هـر دو Superclass و Subclass ایـــن اینتـــرفیس را محقـــق ســـازند, و یـــک اینتـــرفیس Wide کـــه در واقـــع نســـخه کامـــل‌تر بـــرای Operationهـــای Superclass هســـت و در واقـــع تمـــام Operationهـــای او را دارد می‌ســـازیم و فقـــط Superclass ایـــن اینتـــرفیس را محقـــق می‌سازد.

ایـن الگـو بـرای به‌اشـتراک‌گذاری یـک سـری عملیـات هـای مشـترک بـین دو کـلاس نیـز کـاربرد دارد, هـم چنـین بـرای محـدودکردن دیـد کلاینت‌هـا (یعنـی یـک زیرمجموعـه از operationهـا در اختیـار یـک کلاینـت و یـک زیرمجموعـه دیگـر در اختیـار دیگـر کلاینت‌هـا قـرار می‌دهـد) نیـز استفاده می‌شود و باعث کاهش Coupling و مانع از انتشار تغییرات می‌شود.

چگونگی مفید بودن الگو

همـان‌طور کـه در بـالا اشـاره شـد در شـرایطی کـه Refused Bequest داریـم و ایـن مشـکل را می‌خـواهیم بـا اسـتخراج اینتـرفیس برطـرف کنـیم, بایـد ۲ اینتـرفیس Narrow & Wide تعریـف کـرد. حـــال ایـــن الگـــو عـــلاوه بـــر کـــاهش شـــدید Coupling و هـــم چنـــین برقـــراری LSP و ازبین‌بـــردن رابطـــه تـــوارثی کـــه خیلـــی توجیـــه وجـــودی نـــدارد و نـــاقض LSP و is-a هســـت, می‌تواند به ما کمک کند.

هم چنین کلاینت دیگر به تمام متدهایی که در فوق کلاس بود, دید ندارد.

یـا متوجـه نمی‌شـود کـه کـدام آبجکـت در اختیـار او قـرار گرفتـه اسـت زیـرا LSP برقـرار اسـت و فقط با اینترفیس آن کار می‌کند. در واقع این کار را بیشتر در الگوهای طراحی دیده‌ایم.

حال برای محدودکردن دید, همان‌طور که در بالا اشاره شد, می‌توان چندین اینترفیس تعریف کرد تا یک کلاینت به تمام Operationها دسترسی نداشته باشد و یک اشتباه بالقوه که فراخوانی اشتباه است را به یک اشتباه بالفعل تبدیل نکند.

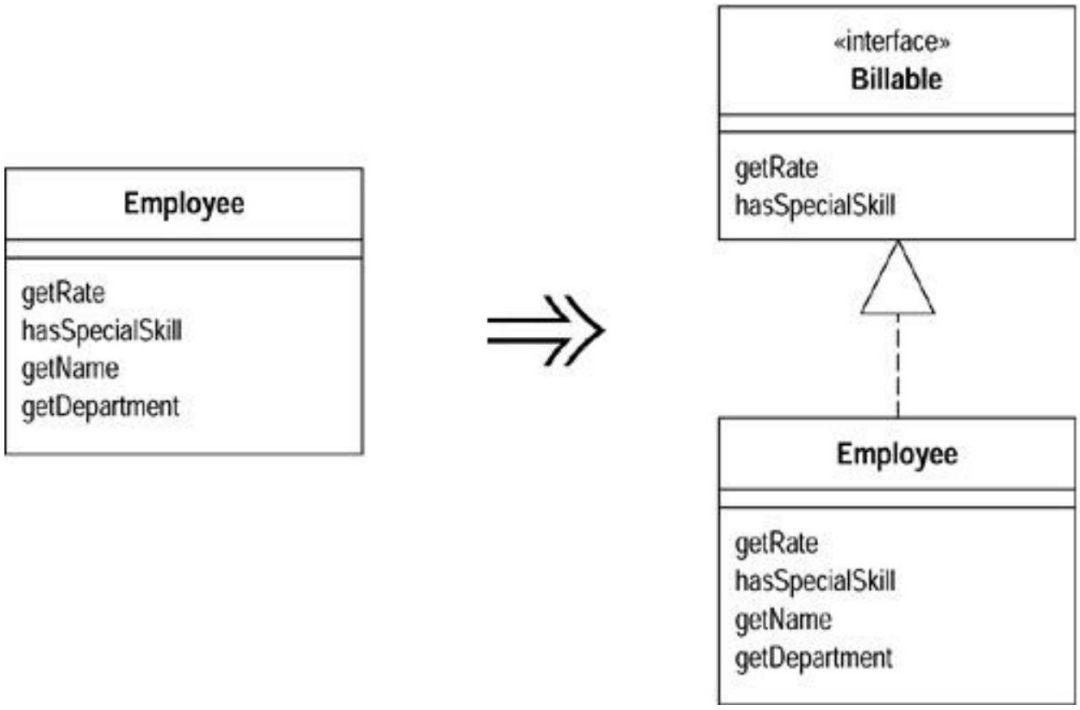
چگونگی پیاده‌سازی یا اعمال الگو

۱. ابتدا یک اینترفیس Narrow برای عملیات هایی که هم در Superclass و هم Subclass هست تعریف می‌کنیم.
۲. سپس یک اینترفیس برای عملیات هایی که جزو اشتراکات این دو کلاس نیست تعریف کرده که از اینترفیس مذکور در مورد بالا ارث می‌برد.
۳. سپس Superclass از اینترفیس Wide و Subclass از اینترفیس Narrow ارث می‌برد و آن‌ها را محقق می‌سازند.
۴. رابطه توارثی را قطع می‌کنیم.
۵. در کد کلاینت هرجا که زیر کلاس نمونه‌سازی شده و در متغیری از جنس Superclass قرار داده شده, به متغیری از جنس اینترفیس Narrow تبدیل می‌کنیم.

مثال

مثلاً برای کاهش دید یا حتی اشتراک‌گذاری یک سری عملیات ها بین آبجکت‌ها برای کلاس کارمند می‌توان اینترفیس تعریف کرد و این کلاس, این اینترفیس را محقق سازد, همان‌طور که در شکل زیر مشاهده می‌کنید, می‌توان گفت که هم LSP برقرار شده,

ممکـن اســت در آینـده انـواع مختلفـی از Employee بـه سیســتم اضـافه شــود, و هـم چنـین دیـد کلاینـت بـه همـین اینتـرفیس محـدود شـده اسـت. وضـعیت قبـل و بعـد اعمـال ایـن الگـو را در شـکل زیر می‌بینید.



شـکل ۲

مزایای اعمال این الگو

۱. محدودکردن دید کلاینت‌ها.
۲. اشتراک‌گذاری عملیات ها بین کلاس‌ها.
۳. کم‌کردن Coupling.

۴. می‌توان به‌واسطه اینترفیس‌ها DIP را برقرار کرد و در نتیجهٔ آن OCP نیز برقرار شود.
۵. برقرار کردن LSP: زیرا دیگر کلاینت نمی‌داند کدام نمونه از کدام آبجکت در اختیار او است و فقط با اینترفیس آن کار می‌کند و همچنین این نمونه می‌تواند عوض شود.
۶. با استفاده از تعریف اینترفیس‌های متعدد و کوچک و Cohesive می‌توان ISP را نیز برقرار کرد.

معایب اعمال این الگو

۱. یک سری پیاده‌سازی‌ها که در فوق کلاس مشترک بوده و به‌درستی در زیر کلاس از آن استفاده شده است باید در هر دو کلاس به‌واسطه اینترفیس‌ها پیاده‌سازی شود. (این مشکل به دلیل این است که ما برای ازبین‌بردن Refused Bequest داریم از الگوی Extract Interface استفاده می‌کنیم و مجبوریم ساختار توارثی را از بین ببریم).
۲. به دلیل ذکر شده در مورد اول می‌تواند باعث تولید Duplicated Code شود.

الگوهای مرتبط

این الگو شباهت زیادی با Extract Superclass دارد, اما تفاوتی با این الگو دارد که در بعضی زبان‌ها نمی‌توان Multiple Inheritance از کلاس‌ها داشت و هم چنین اگر پیاده‌سازی‌های یکسان در فوق کلاس‌ها و زیر کلاس‌ها داریم می‌توان در فوق کلاس

قـرارداد و از آن ارث بـرد, امـا در Extract Interface نمی‌تـوان پیاده‌سـازی مشـترک در آنجـا داشت (در C# به‌تازگی Default Implementation در اینترفیس‌ها اضافه شده است.)

مقایسه اعمال دو الگو

بـا بررسـی‌های انجـام شـده می‌تـوان گفـت هـر دو الگـو بـه نحـوی می‌تواننـد مشـکل موقعیـت فعلـی کـه بیشـتر بـه‌خاطر Refused Bequest هسـت را برطـرف کننـد و شـباهت‌هایی مثـل افـزایش Maintainability کـاهش Coupling و کـاهش انتشـار تغییـرات دارنـد, امـا تفاوت‌هـایی هـم دارنـد: اولـین و مهم‌تـرین تفـاوت نحـوه اعمـال ایـن دو الگـو اسـت کـه الگـوی اول از طریـق واسـپاری و الگـو دوم از طریـق تعریـف اینتـرفیس ایـن کـار را انجـام می‌دهـد. تفـاوت بعـدی ایـن اسـت کـه در الگـوی اول بـه دلیـل واسـپاری سـربار انتقـال پیـام زیـاد می‌شود اما در الگوی دوم این‌گونه نیست.

می‌تـوان گفـت الگـوی باعـث دوم کـاهش وابسـتگی, کم‌کـردن دیـد کلاینت‌هـای از طریـق تعریـف اینترفیس‌هـای Narrow & Wide کمـک کنـد امـا همـان‌طور کـه در اینجـا اشـاره شـد می‌تواند باعث تولید Duplicated Code شود.

الگـوی اول نیـز تـا حـدی Coupling و انتشـار تغییـرات را کـاهش می‌دهـد و بـه نظـر می‌رسـد در ایـن مـوقعی, نسـبت بـه الگـوی دوم بهتـر عمـل می‌کنـد. زیـرا الگـوی دوم ممکـن اسـت باعـث تولیـد Duplicated Code بشـود. البتـه در مـورد الگـوی اول نیـز می‌تـوان گفـت کـه کـاری برای کاهش دید کلاینت به آبجکت‌ها انجام نمی‌دهد.

پـس بایـد باتوجه‌بـه Context و مـوقعیتی کـه در آن هسـتیم و بـا درنظرداشـتن Trade-Off ها, الگوی مناسب را انتخاب کنیم.

بررسی موقعیت دوم

مشکلی که در این موقعیت داریم, تغییرات متعدد و بی‌ربط نسبت به یک تغییر دیگر است, مثلاً تکنولوژی UI تغییر می‌کند, اما باید یک سری تغییرات در مدل‌ها یا لاجیک واکشی و نوشتن مدل‌ها صوت گیرد تا به‌درستی کار انجام شود. می‌توان گفت که برعکس Shotgun Surgery است, یعنی ما در Divergent Change باید تغییرات زیاد در یک کلاس (به دلایل بی‌ربط) انجام بدهیم, اما در Shotgun Surgery برای یک تغییر, باید تغییرات زیاد به‌صورت هم‌زمان در کلاس‌های متعدد صورت بگیرد.

Divergent Change یعنی یک کلاس وجوه مختلف دارد و Cohesive نیست, اما Shotgun Surgery یعنی Coupling به‌شدت بالاست.

الگوی اول: شکستن/تکه‌تکه کردن **God Class**

بررسی الگوی اول

این الگو از دسته الگوهای بازمهندسی و در فاز Redistribute Responsibilities است.

همان‌طور که از نام این الگو مشخص است, در اینجا ما یک کلاسی داریم که بسیار بزرگ و چندوجهی است, در نتیجه آن نیز تعداد زیادی Data Class هم خواهیم داشت. وجود چنین کلاسی در این موقعیت باعث می‌شود که برای تغییرات در یک فیچر سیستم (یعنی دست‌کاری در این God Class) مجبور باشیم جاهای دیگری از این

کلاس که هیچ ارتباطی با این فیچر ندارند را تغییر دهیم, این مشکل به دلیل Coupling بالا و Cohesion پایین در این کلاس به وجود آمده.

باید این God Class شکسته شود و رفتارها به نزدیک داده‌های منتقل شوند و کپسول‌های داده رفتار داشته باشیم و کلاس‌ها با هم تعامل داشته باشند, در این فرآیند ممکن است رفتارها از این کلاس به کلاس‌های داده‌های مرتبط خود منتقل شوند (Move Behavior Close To Data) , یا کلاس‌های جدید تعریف کنیم (Extract Class). گاهی این رفتارها بسیار درشت‌دانه هستند و باید از الگوی Extract Method نیز استفاده کرد. همانطور که جلوتر در مثال هم خواهیم گفت, این الگو به صورت Incremental انجام می‌شود. در نهایت یک Façade از God Class باقی می‌ماند که می‌توان آن را حذف کرد.

زمان اعمال الگو

هنگامی که در یک سیستم تعداد زیادی Data Class و یک کلاس بزرگ چندوجهی که مجموعه بزرگی از رفتار دارد را می‌بینیم می‌توانیم این الگو را اعمال کنیم. این الگو باعث می‌شود تا به‌شدت Divergent Change داشته باشیم, قابلیت تغییر و نگهداری بسیار پایین است, هم چنین Cohesion پایین و Coupling بالا باعث می‌شود تا تغییرات در این کلاس بسیار سخت باشد و برای یک تغییر ساده در یک از عملیات ها مجبور باشیم در سرتاسر این کلاس تغییرات را اعمال کنیم درصورتی‌که این تغییرات هیچ ربطی به تغییر اعمال‌شدهٔ اولیه ندارد (Divergent Change).

پس از اعمال الگو کپسول‌های داده رفتار Cohesive داریم که با یکدیگر تعامل دارند و به دلیل Coupling پایین آن‌ها تغییرپذیری افزایش و انتشار تغییرات کاهش پیدا می‌کند.

چگونگی مفید بودن الگو

این الگو باعث می‌شود تا رفتارهایی که در God Class داریم و از داده‌های خود دورافتاده‌اند, به کنار داده‌های خود منتقل شوند و کپسول‌های داده رفتار داشته باشیم تا این کپسول‌ها Cohesion بالا و Coupling پایین داشته باشند و به‌تبع آن تغییرپذیری در سیستم افزایش پیدا کند و تغییرات منتشر شونده نداشته باشیم و این کلاس‌ها فقط از طریق تعامل با یکدیگر با هم کار کنند و همهٔ این رفتارها در یک کلاس واحد نباشند.

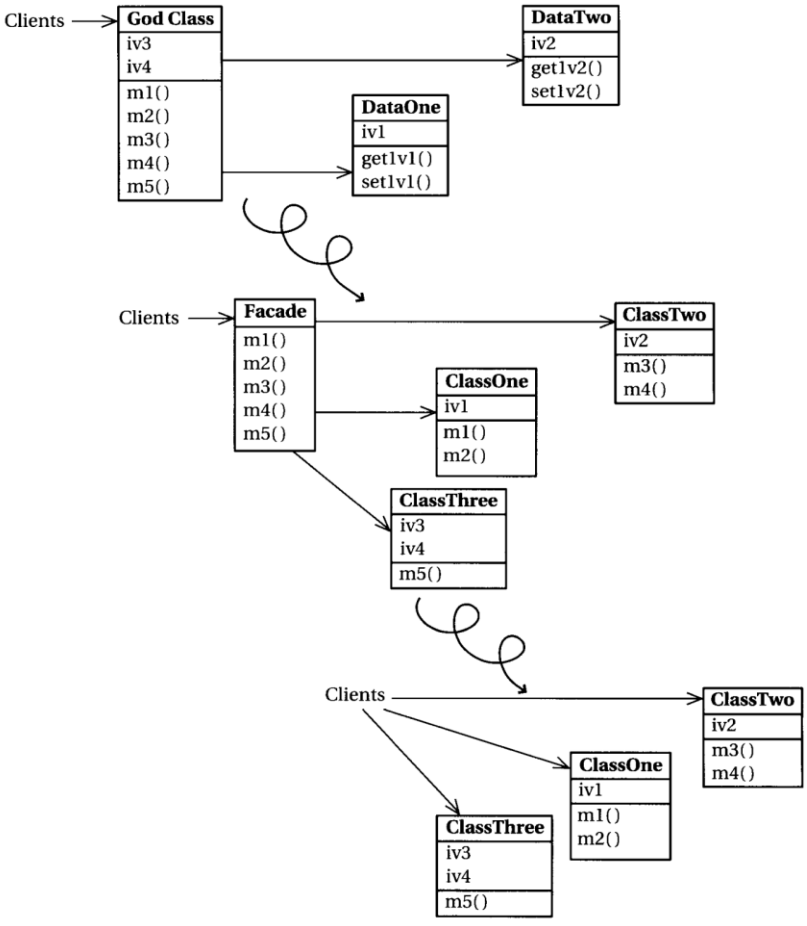
با این الگو که به‌صورت Incrementally اعمال می‌شود, می‌توان رفتارها را به کلاس‌های داده‌ای خود منتقل کرد, یا کلاس‌های جدید ساخت که همگی Cohesive هستند و باعث می‌شود Maintainability و Flexibility افزایش پیدا کند.

چگونگی پیاده‌سازی یا اعمال الگو

۱. ابتدا باید God Class و کلاس‌هایی که با آن ارتباط دارند را شناسایی کنیم.
۲. رفتارها و داده‌هایی که داده‌های آن در کلاس‌های دیگر است به همان جا منتقل کنیم. (Move Behavior Close To Data) و (Move Field/ Move Method). هم چنین ممکن است این رفتارها درشت‌دانه باشد که باید Extract Method هم انجام دهیم.
۳. شناسایی رفتار و داده‌هایی که در حال حاضر کلاسی در سیستم ندارند و باید برای آن‌ها کلاس ساخت. (Extract Class).

بـدیهی اســت کــه فراینــد Initialization تمــام ایــن کلاس‌هــا بایــد در God Class انجــام گیـرد. یـا مـثلاً فراینـد Delegation بایـد به‌درسـتی انجـام شـود. بعـد از انجـام قـدم‌های بـالا به‌صـورت Incremental, God Class فقـط به‌صـورت یـک Facade در می‌آیـد کـه فقـط کـار واسـپاری انجـام می‌دهـد, کـه می‌تـوان آن را نیـز حـذف کـرد و کلاینت‌هـا را مسـتقیم بـه اینترفیس کلاس‌های جدید متصل کرد.

مثال



شکل ۳

همان‌طور که در شکل بالا به‌وضوح مشخص است, قدم‌هایی که در قسمت پیاده‌سازی گفته شد به‌صورت Incremental انجام شده و کم‌کم یا کلاس‌های جدید ظهور پیدا کردند و یا رفتار به کلاس‌های داده‌ای خود منتقل شده‌اند و سپس ازآنجایی‌که God Class به‌صورت یک Lazy Class درآمده و توجیه وجودی خاصی ندارد, حذف شده است.

مزایای اعمال این الگو

۱. کنترل اپلیکیشن به‌صورت مرکزی نیست و کلاس‌های متعدد با هم تعامل می‌کنند و کار سیستم را انجام می‌دهند و هدف آن را محقق می‌سازند.
۲. قسمت‌های God Class مشخص‌تر و ساده‌تر و قابل‌فهم‌تر در کلاس‌های ریزدانه هستند.
۳. کلاس‌ها Stable تر هستند, زیرا Divergent Change اتفاق نمی‌افتد.
۴. CRP محقق می‌شود.
۵. PLK نقض نمی‌شود (مگر نقض غرض).
۶. Encapsulation رعایت می‌شود.
۷. Data Class, Divergent Change, Large Class و خود God Class حذف می‌شوند.
۸. Cohesion افزایش و Coupling کاهش و به‌تبع آن تغییرپذیری و Flexibility افزایش و انتشار تغییرات کاهش میابد.
۹. قابلیت استفاده مجدد از کلاس‌های Cohesive استخراج‌شده فراهم می‌گردد.

معایب اعمال این الگو

۱. فرایند طولانی و خسته‌کننده است.
۲. شکستن رفتارها, شناسایی موجودیت‌ها و کلاس‌ها به‌صورت Cohesion و یا Extract Method/Class سخت است.
۳. تعداد کلاس‌ها افزایش می‌یابد.
۴. تعداد آبجکت‌های Runtime افزایش میابد.
۵. کارایی به دلیل تبادل پیام و تعامل کلاس‌ها به یکدیگر کاهش میابد.
۶. دیباگ سخت‌تر است, زیرا به‌جای یک God Class چندین و چند کلاس داریم که باید مشکل را در آن‌ها پیدا کنیم
۷. اگر God Class را حذف نکنیم می‌تواند باعث نقض ISP شود چون Cohesion بسیار پایینی دارد. اگر آن را حذف کنیم باعث انتشار تغییرات به کلاینت می‌شود. (Trade-Off)

الگوهای مرتبط

در اعمال این الگو مکرراً Extract Method, Extract Class, Move Field/Method و Move Behavior Close To Data اتفاق می‌افتد.

الگوی دوم جداسازی ساختارهای توارثی

بررسی الگوی دوم

این الگو در دسته الگوهای Big Refactoring هست و زمانی کاربرد دارد که ما ساختارهای توارثی تودرتو داریم و این نشان‌دهندهٔ پیاده‌سازی مسئولیت‌ها یا حالت‌ها است که ترکیبات مختلف دارند و با استفاده از Subclassing پیاده‌سازی شده‌اند, به‌جای آن باید از الگوهایی مثل Strategy یا State یا Bridge باید استفاده می‌شد تا از انفجار ترکیبی جلوگیری کرد. این الگو می‌تواند ساختارهای ترکیبی را جدا کند و باعث افزایش قابلیت نگهداری, کاهش انتشار تغییرات و ساده شدن ساختار کد (کم‌کردن Complexity) شود. در واقع Subclassing بر مبنای Functionality اتفاق افتاده و ما در این موقعیت Nested Generalization داریم و این الگو سعی می‌کند تا این مشکل را برطرف سازد.

این الگو با افزایش Cohesion و جداسازی این ساختار توارثی و ارتباط آن‌ها با استفاده از Delegation (در ریشهٔ هر سلسله‌مراتب) باعث می‌شود کلاس‌ها تک کاره شوند و از انفجار ترکیبی و تغییرات منتشر شونده جلوگیری کرد.

این الگو می‌تواند آن‌قدر بزرگ باشد تا به یک الگوی بازمهندسی تبدیل شود.

تفاوت آن با Bridge این است که در این الگو Abstraction و Implementation جدا نمی‌شود, اما در Bridge جدا می‌شود.

زمان اعمال الگو

یکی از اصلی‌ترین نشانه‌های زمان اعمال این الگو می‌تواند Nested Generalization باشد, خود این استفاده نادرست از توارث باعث می‌شود مشکلاتی مثل انتشار تغییرات یا Divergent Change به وجود بیاید, زیرا که ما مثلاً در یک کلاس تکنولوژی نمایش UI را تغییر می‌دهیم اما در همان کلاس لازم هست تغییرات دیگر و بی ربطی بدهیم تا Functionality اصلی دچار مشکل نشود.

نشانه دیگر می‌تواند Duplicated Code باشد, همان‌طور که در بالا اشاره کردیم, ممکن است انفجار ترکیبی باعث شود مثلاً برای نمایش انواع و اقسام Entity ها, روش‌های مختلفی داشته باشیم, و در این ساختار همهٔ ترکیبات ممکن را داریم و باعث می‌شود که برای هر Entity, هر یک از روش‌های نمایش را پیاده‌سازی کنیم (مثلاً روش پرینت, روش نمایش در صفحه‌نمایش و...) همچنین خود این ترکیبات باعث تغییرات منتشر شونده می‌شود, یعنی با تغییر در هر از Entityها یا روش‌ها نمایش, این تغییر در بقیه ترکیبات موجود باید اعمال شود.

چگونگی مفید بودن الگو

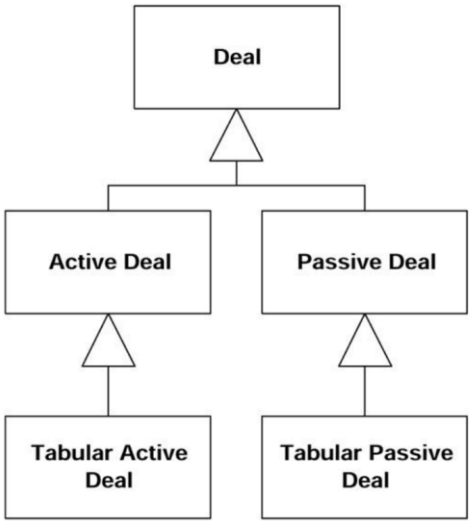
این الگو با جداسازی ساختارهای توارثی تودرتو در برقرار کردن ارتباطات آن‌ها در ریشهٔ ساختارها از طریق Delegation باعث می‌شود تا تغییرات منتشر شونده, انفجار ترکیبی, سلب بودن ساختار توارثی و تغییرات بی‌ربط که همان Divergent Change است, از بین بروند. زیرا ما برای پیاده‌سازی این ترکیبات (همانند الگوی State) می‌توانیم مثلاً تکنولوژی UI را با یک فیلد در یک Entity که قرار هست نمایش بدهیم, و

نیاز نیست که این ترکیب به‌صورت ساختار توارثی سلب در بیاید. (در مثال این موضوع روشن‌تر توضیح داده شده است) , لازم به ذکر است این رابطه Association در ریشه‌ها یا Superclassها انجام می‌شود که برای همه زیر کلاس‌ها اعمال شوند.

چگونگی پیاده‌سازی یا اعمال الگو

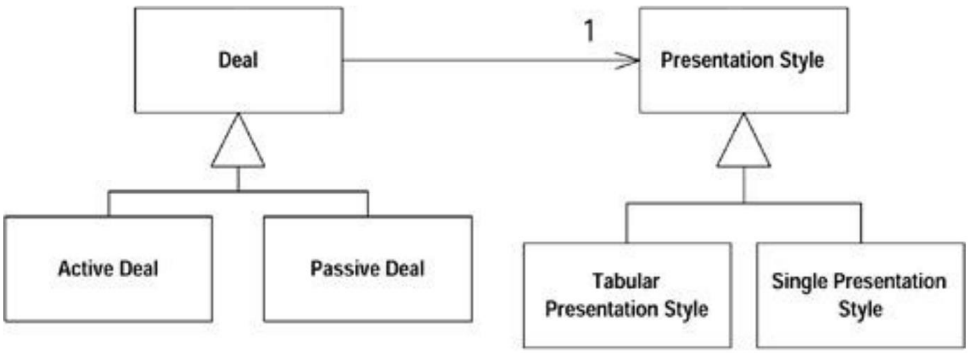
۱. کارهای متفاوتی که در ساختار توارثی در حال انجام هست را شناسایی کنیم.
۲. اصلی‌ترین کار را شناسایی کرده تا در ساختار فعلی نگه داریم.
۳. تا زمانی که ساختار توارثی اولیه Cohesive و تک کاره نشده باید با استفاده از الگوی Extract Class وظایف دیگر را شناسایی کرده و به‌صورت کلاس در بیاوریم تا در ساختار توارثی دیگری قرار دهیم.
۴. در ریشه یا Superclass ساختار اولیه یک یا چند فیلد برای Delegate کردن کارها به ساختارهای دیگر ایجاد می‌کنیم.
۵. در تمام این ساختارها می‌توان برای کاهش کد تکراری و افزایش Flexibility از الگوهای Pull Up Method/Field استفاده کرد تا فقط در فوق کلاس باشند.

مثال



شکل ۴

همان‌طور که در شکل بالا می‌بینید, برای هر روش نمایش یک Deal و ترکیب آن با انواع Deal زیر کلاس تعریف شده است و ما باید سعی کنیم تا این دو ساختار با استفاده از این الگو به دو یا چند ساختار تبدیل شوند و از طریق ریشه و از طریق Delegation ارتباط داشته باشند.



شکل ۵

وضعیت After اعمال این الگو را در شکل بالا مشاهده می‌کنید.

مزایای اعمال این الگو

۱. باعث کاهش Duplicate Code می‌شود.
۲. CRP برقرار و در نتیجهٔ آن Flexibility افزایش میابد.
۳. DIP و در نتیجهٔ آن OCP داریم (زیرا هرکدام از ساختارها فقط با فوق کلاس یا اینترفیس ساختار توارثی دیگر کار می‌کند.)
۴. افزایش Cohesion
۵. کاهش Coupling
۶. انتشار تغییرات بسیار کم و نزدیک به صفر می‌رسد.
۷. Large Class حذف می‌شود.
۸. کلاس‌ها بسیار ساده‌تر خواهند شد.
۹. توسعه‌پذیری هر ساختار توارثی بسیار ساده‌تر می‌شود.
۱۰. امکان استفاده مجدد از هر ساختار توارثی افزایش می‌یابد.

معایب اعمال این الگو

۱. می‌تواند آن‌قدر سخت و بزرگ باشد تا تبدیل به یک پروژهٔ Reengineering شود.
۲. از نظر Performance به دلیل Delegation و تبادل پیام, کارایی کاهش میابد.
۳. شناسایی ساختار توارثی از ترکیبات متعدد بسیار سخت است (مخصوصاً اگر به‌جای ۲ ساختار چندین ساختار توارثی درهم‌تنیده داشته باشیم.)

۴. اگر به‌درستی انجام نشود می‌تواند باعث نقض PLK شود و زنجیره دید ترایا ایجاد شود. (نقض غرض)
۵. در صورت اشتباه در جداسازی ساختارهای توارثی ممکن است باعث تولید Feature Envy شود.

الگوهای مرتبط

برای اعمال این الگو از الگوهایی مثل Extract method, Extract Class و یا Pull up Field/Method باید استفاده شود.

ارتباط نزدیکی با الگوهای State, Strategy یا Bridge دارد, البته با تفاوت‌هایی که بالاتر به آن‌ها اشاره شد.

می‌توان گفت به طریقی با Shotgun Surgery ارتباط دارد (تفاوت), فقط Shotgun Surgery باعث می‌شود یک تغییر به‌صورت هم‌زمان به جاهای دیگر منتشر شود, اما Divergent Change باعث می‌شود که یک تغییر باعث تغییر بی‌ربط به همان کلاس شود.

مقایسه اعمال دو الگو

با بررسی‌های انجام شده می‌توان گفت هر دو الگو می‌توانند باعث جلوگیری از تغییرات بی‌ربط یا همان Divergent Change شوند, اما باتوجه‌به Context مسئله می‌توان از

بین این دو الگو انتخاب کرد. هر دو در کاهش انتشار تغییرات, افزایش Maintainability و Flexibility و همچنین کاهش Coupling نقش به سزایی ایفا می‌کنند.

اما همان‌طور که گفته شد الگوی اول در جایی اعمال می‌شود که یک God Class و چندین Data Class داریم که با هم کار می‌کنند (به‌جای این که چندین کلاس Cohesive باهم تعامل داشته باشند).

اما در الگوی دوم دو یا چند ساختار توارثی درهم‌تنیده شده‌اند و تغییر در یک قسمت باعث تغییرات بی‌ربط در همان کلاس خواهد شد (نمود Divergent Change) و یا باعث ایجاد انفجار ترکیبی شود و لازم باشد برای تولید یک الگوریتم نمایش جدید, زیر کلاس‌های متعددی برای هر Entity بسازیم (همان‌طور که در مثال گفته شد.)

بررسی موقعیت سوم

در این موقعیت تکه کدهای مشابه و تکراری در قسمت‌های مختلف متدهای کلاس‌ها متفاوت وجود دارد. این مشکل باعث کاهش Maintainability و Flexibility و باعث افزایش تغییرات منتشر شونده می‌شود. در واقع Code Smell یا Bad Smell که اینجا وجود دارد Duplicated Code هست که یکی از مهم‌ترین Code Smellها است. مثلاً فرض کنید کدی که با Copy-Paste ساخته شده است, در صورت تغییر در یک کلاس باید تغییرات را در کلاس دیگر اعمال کنیم. هم چنین ممکن است دو تیم که روی یک پروژه کار می‌کنند, سهواً دو متد یا دوتکه کد برای کار یکسان با الگوریتم یکسان بنویسند و تفاوت‌های جزئی داشته باشند اما به‌صورت Duplicated Code آن‌ها را متیوان در نظر گرفت البته تشخیص این نوع Duplicate Code معمولاً سخت است. می‌توان الگوهای بازآرایی مانند Extract Method و یا Pull up Method و یا Extract Superclass انجام داد (بسته به شرایط) و از شر این کدهای تکراری راحت شد. با برطرف‌کردن این مشکل قابلیت نگهداری افزایش‌یافته و کد خواناتر و ساده‌تر شده و هم چنین تغییرات منتشر شونده کاهش میابند.

الگوی اول: معرفی کیس خاص

بررسی الگوی اول

این الگو به این صورت عمل می‌کند که مثلاً ما در جاهایی لازم داریم تا چک کنیم آیا آبجکتی که داریم (فرض کنیم یکی از آبجکت‌های بیزینس یا دیتابیس هست) null هست یا نه, اگر نال بود مثلاً یک مقدار دیفالت برگردانیم تا یک سری فیلدها پر شوند, یا اگر نال نبود عملیات موردنظر را انجام دهیم. این عمل باعث چکینگ های تکراری در سراسر برنامه خواهد شد. می‌توان یک آبجکت برای این کار تعریف کرد تا از کلاس اصلی ارث ببرد و Subclass کلاس اصلی باشد و رفتار دیفالت را درون خود داشته باشد (می‌تواند مثلاً برای موجودیت‌های دیتابیسی فقط یک سری فیلدها با مقادیر دیفالت باشد.).

اعمال این الگو می‌تواند از تکه کدهای تکراری در سراسر برنامه جلوگیری کند, کد را خوناتر کند, قابلیت نگهداری افزایش یابد, تغییرات منتشر شونده کم شود (ممکن است مثلاً یک فیلد به این موجودیت اضافه شود و مثلاً باید تمام جاهایی که چکینگ نال انجام شده و رفتار دیفالت پیاده‌سازی شده, باید بازنویسی شود.) هم چنین Duplicated Code را کاهش دهد.

زمان اعمال الگو

همان‌طور که در بالاتر گفته شد, مثلاً برای چکینگ هایی از نال آبجکت یک کلاس تعریف کرد که رفتار پیش‌فرض را درون خود دارد و این کلاس را زیر کلاس برای کلاس اصلی

قـرار دهـیم, سـپس تغییـرات را در همـهٔ قسـمت‌های چکینـگ اعمـال کنـیم تـا Duplicated Code کاهش یابد.

در واقـع تعریـف یـا معرفـی کـیس نـال, حالـت خـاص‌تر از معرفـی کـیس Special اسـت و مـا برای سادگی کیس نال را در نظر می‌گیریم (گرچه تفاوت خاصی ندارد).

چگونگی مفید بودن الگو

فـرض کنیـد مرتبـاً در یـک سیسـتم قـرار اسـت بـه دیتـابیس درخواسـت بـزنیم و ببینـیم آیـا یـک موجودیـت خـاص بـا فیلترهـایی کـه دادیـم رکـوردی دارد یـا نـه, در ایـن صـورت بایـد در بیـزینس لاجیـک بررسـی کنـیم کـه اگـر نـال نبـود رفتـار خـود را اجـرا کنـد و اگـر نـال بـود یـک رفتـار پیش‌فـرض را اجـرا کنـد, ایـن خـود باعـث تولیـد تکـه کـدهای تکـراری شـده و قابلیـت نگهـداری را کـم می‌کنـد, چـرا کـه اگـر بعـدها لازم باشـد تـا ایـن رفتـار پیش‌فـرض تغییـر کنـد, بایـد در تمـام قسمت‌ها این تغییر را اعمال کنیم (مصداق تغییرات منتشر شونده).

خـوب همـان‌طور کـه بـالاتر هـم اشـاره شـد بـا تعریـف یـک Subclass از کـلاس اصـلی کـه کـار آبجکـت نـال را می‌کنـد و تعریـف رفتـار پیش‌فـرض در آن می‌تـوان حتـی از Polymorphism اسـتفاده کـرد تـا باتوجه‌بـه type ایـن رفتـار را اجـرا کـرد و تغییـرات منتشـر شـونده یـا Duplicated Code نداشته باشیم.

چگونگی پیاده‌سازی یا اعمال الگو

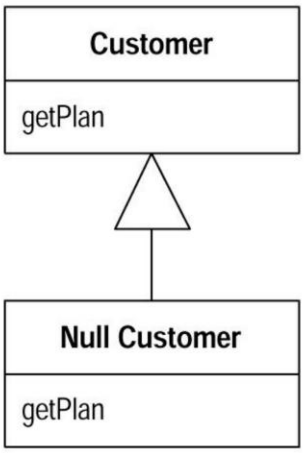
۱. یک فیلد در زیر کلاس اصلی می‌سازیم و مقدار isNull را درون خود دارد که False هست. بهتر است Getter فقط برای آن تعریف شود و Setter هم لازم ندارد. بهتر است کلاً فیلد هم نباشد و فقط متد باشد.
۲. یک زیر کلاس از کلاس اصلی می‌سازیم که کار رفتار خاص (که در اینجا رفتار در زمان نال بودن را در نظر گرفته‌ایم) انجام می‌دهیم, متد isNull برای این کلاس True هست.
۳. تمام جاهایی که چک کردیم و کلاس اصلی نال بوده, این کلاس را جایگزین می‌کنیم.
۴. تمام جاهایی که کلاس را با null مقایسه کرده‌ایم, با مقدار isNull مذکور در بالا چک می‌کنیم.

مثال

فرض کنید ما آبجکت مشتری داریم و می‌خواهیم برای او فاکتور صادر کنیم (این فاکتور بسته به کلاس‌های مشتری متفاوت است یعنی رفتار وابسته به type یا polymorphism دارد)

اما متوجه می‌شویم که این آبجکت نال است, حال باید به‌صورت هاردکد رفتار دیفالت را پیاده‌سازی کنیم که منجر به Duplicated Code خواهد شد.

به‌جای این کار همان‌طور که در بالا اشاره شد یک Null Object تعریف می‌کنیم که از کلاس اصلی ارث می‌برد.



شکل ۶

همانند شکل بالا این کلاس تعریف شده و مثال استفاده از آن به شکل زیر است و وضعیت Before/After آن را در شکل زیر مشاهده می‌کنیم.

```
if (customer == null)
{
  plan = BillingPlan.Basic();
}
else
{
  plan = customer.GetPlan();
}
```

```
public sealed class NullCustomer: Customer
{
  public override bool IsNull
  {
    get { return true; }
  }

  public override Plan GetPlan()
  {
    return new NullPlan();
  }
  // Some other NULL functionality.
}

// Replace null values with Null-object.
customer = order.customer ?? new NullCustomer();

// Use Null-object as if it's normal subclass.
plan = customer.GetPlan();
```

شکل ۷

مزایای اعمال این الگو

۱. باعث ازبین‌رفتن Duplicated Code می‌شود.
۲. باعث ازبین‌رفتن Repeated Switch Statement می‌شود.
۳. در بعضی موارد می‌تواند باعث ازبین‌رفتن Temporary Field شود.
۴. LSP به‌خوبی برقرار می‌شود (از شیء NullObject می‌توان به‌عنوان شیء اصلی در زمان نال بودن آن استفاده کرد).
۵. قابلیت نگهداری افزایش میابد.
۶. کد تمیزتر, ساده تر و خواناتر خواهد شد.

معایب اعمال این الگو

۱. معمولاً تعداد این نال آبجکت‌ها زیاد می‌شود (بهتر از Singleton استفاده کرد).
۲. شاید سلسه‌مراتب توارثی کمی پیچیده‌تر شود (در بعضی موارد).
۳. هنگامی که آبجکت null را زیرکلاس برای کلاس اصلی قرار می‌دهیم, ممکن است بعضی از متد ها را خالی کند, این مصداق Refused Bequest است.

الگوهای مرتبط

۱. یکی از الگوهای مشابه بازآرایی Replace Conditional With Polymorphism است.

۲. در شرایطی که کدهای تکراری ربطی به ساختار توارثی ندارد می‌توان از Extract Function استفاده کرد و مثلاً در یک Class Helper قرار دارد (مثلاً DateTimeProvider که همهٔ کلاس‌ها ممکن است از آن استفاده کنند.)

۳. با الگوی طراحی Singleton پیاده‌سازی شود بهتر است.

الگوی دوم **Form template method**

بررسی الگوی دوم

این الگو می‌تواند در جاهایی که کدهای تکراری دقیقاً مشابه هم نیستند اما یک ساختار مشابه (مثلاً قبل از loop, داخل loop و بعد از loop قرار هست کارهایی انجام شود) دارند و این کارها در زیر کلاس‌های متفاوت, بسته به نوع این زیر کلاس‌ها متفاوت هستند. یعنی می‌خواهیم ساختار را به اشتراک بگذاریم, از Duplicated Code جلوگیری می‌کنیم.

یک متد Template در فوق کلاس می‌سازیم که در واقع یک Placeholder برای کارهایی است که به‌صورت Polymorphic در زیر کلاس‌ها پیاده‌سازی می‌شوند (منظور همان کارهای قبل و بعد و داخل loop است).

زمان اعمال الگو

این الگو زمانی می‌تواند باعث کاهش یا ازبین‌بردن کدهای تکراری شود که این کد هرچند که پیاده‌سازی‌های متفاوت هستند, اما یک ساختار یا روال مشخص دارند. این

ساختار را در Superclass تعریف کرده و جزئیات و متدهای ریزدانهٔ این ساختار وابسته به type در زیر کلاس‌ها تعریف می‌شود.

اگر به هر دلیلی این ساختار به‌صورت کلی تغییر کند, نیاز نیست این تغییر در همهٔ زیر کلاس‌ها منتشر شود و آن‌ها را نیاز نیست تغییر دهیم, بلکه در زیر کلاس‌ها فقط جزئیات پیاده‌سازی قسمت‌های مختلف این ساختار وجود دارد. (پس به‌نوعی از تغییرات منتشر شونده جلوگیری کردیم).

چگونگی مفید بودن الگو

وقتی در زیر کلاس‌ها در واقع یک ساختار مشخص برای انجام یک عملیات درشت‌دانه داریم و این ساختار مدام در حال تکرار هست, علاوه بر اینکه مصداق کد تکراری است, مصداق تغییرات منتشر شونده نیز هست.

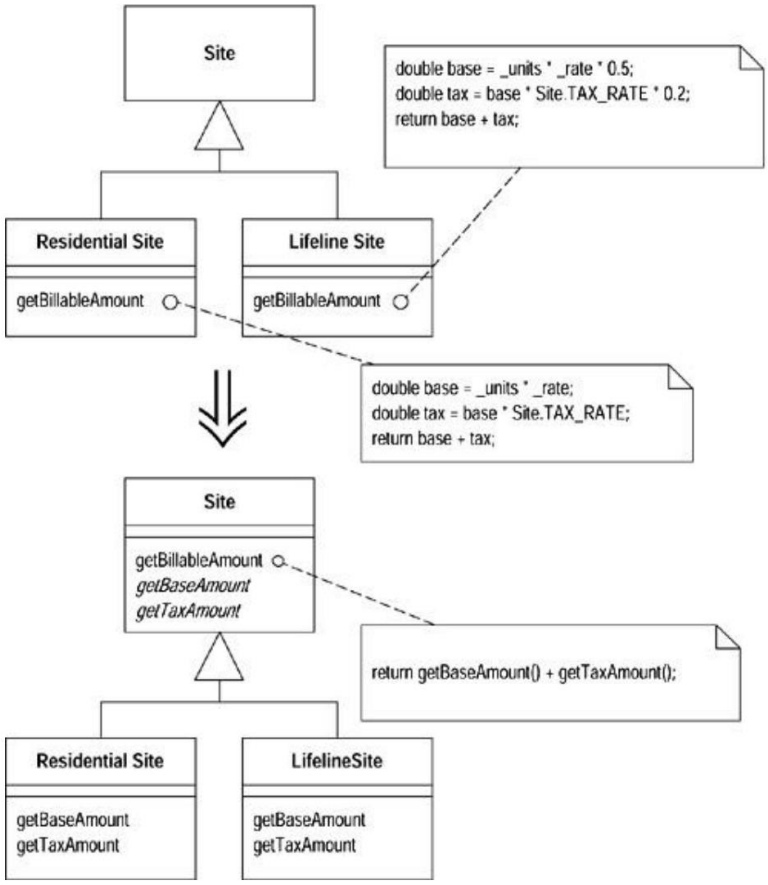
مثلاً فرض کنید کارهای قبل از loop را متد A کارهای درون loop را متد B و کارهای بعد از loop را متد C در زیر کلاس‌ها پیاده‌سازی کنند. سپس این ساختار یعنی A,B و C پشت‌سرهم با نام Template Method در Superclass تعریف شوند و پیاده‌سازی‌های A,B و C فقط در زیر کلاس‌ها باشد, به‌این‌ترتیب هنگامی که ساختار کلی تغییر کند, این تغییر به‌جایی منتشر نمی‌شود, هم چنین از کد تکراری هم جلوگیری کرده‌ایم.

چگونگی پیاده‌سازی یا اعمال الگو

۱. ساختار کلی لاجیک را مشخص کرده و برای هر قسمت آن یک متد Abstract در Superclass تعریف کنیم (Extract Method)

۲. قسمت‌های مشترک در متد مذکور در مورد ۱ و هم چنین ساختار کلی که قرار است آن را Template Method بنامیم با الگوی Pull up Method به Superclass منتقل می‌شوند.
۳. پیاده‌سازی‌های مختلف متدهای ریزدانه که قسمت‌های مختلف ساختار کلی را تشکیل می‌دهند در زیرکلاس‌ها نوشته می‌شود.
۴. فراخوانی‌های مرتبط را تصحیح می‌کنیم تا متد Template Method از Superclass را صدا بزنند. یعنی هرجایی یکی از Subclassها می‌خواهد عملیات انجام دهد این متد را فراخوانی می‌کند.

مثال



شکل ۸

همان‌طور که در شکل بالا می‌بینید هرکدام از Siteها ساختار کلی مشابهی برای محاسبه Billable دارند, این ساختار می‌تواند به‌صورت کلی در فوق کلاس تعریف شود و هرکدام از زیر کلاس‌ها پیاده‌سازی‌های قسمت‌های ریز دانهٔ این متد که با زیر کلاس‌های دیگر فرق دارد را, درون خود به‌صورت Polymorphic پیاده‌سازی کنند, به‌این‌ترتیب Duplicate Code حتی روی ساختار هم نداریم و فقط قسمت‌های ریزدانه با هم تفاوت دارند که داخل زیر کلاس‌ها هستند.

مزایای اعمال این الگو

۱. باعث کاهش Duplicate Code می‌شود.
۲. تغییرات منتشر شونده به سبب تغییر در ساختار کلی که در ساختار کلی و فوق کلاس است, نداریم.
۳. ایجاد الگوریتم‌های جدید بر مبنای ساختار کلی, به‌راحتی انجام می‌شود, زیرا که در واقع OCP برقرار است.
۴. استفاده صحیح از Polymorphism رعایت میشود.
۵. Extensibility افزایش میابد.
۶. Shotgun Surgery نداریم, همانطور که در بالا گفتیم, دیگر تغییرات کلی در ساختار کلی نیاز نیست در تمام قسمت‌های زیر کلاس‌ها اعمال شود.

معایب اعمال این الگو

۱. زمان اجرا به دلیل فراخوانی‌های متعدد متدها و هم چنین فراخوانی متد فوق کلاس افزایش میابد.
۲. باید متدهایی که قرار هست فقط در زیر کلاس‌ها تعریف شوند, به‌صورت Protected باشند, اگر این‌گونه نباشد می‌تواند ناقض ISP شود, یا فراخوانی‌های غیرمجاز و غلط به متدهای ریزدانهٔ زیر کلاس‌ها اتفاق بیفتد.
۳. نامگذاری قسمت و متدهای مختلف Template Method باید بسیار مرتبط و معنی‌دار باشند.

۴. در صورت تغییر در متد Template Method و تغییر ساختار کلی این الگوریتم, این تغییر به تمام زیر کلاس ها منتشر خواهد شد (مثلا افزایش یا کاهش یک step در Template Method).
۵. میتواند باعث تولید Refused Bequest و در نتیجه نقض LSP شود.

الگوهای مرتبط

این الگو همان‌طور که در بالاتر اشاره شد شباهت‌هایی به Extract Method و Pull up Method دارد و یا از آن‌ها استفاده می‌کند و همچنین پیاده‌سازی الگوی طراحی Template Method شباهت زیادی با آن دارد.

مقایسه اعمال دو الگو

همان‌طور که در بالاتر به آن اشاره شد, هر دو الگو به‌خوبی بسته به شرایط و Context می‌تواند باعث کاهش Duplicate Code افزایش Maintainability و کاهش تغییرات منتشر شونده بشوند. هم چنین هر دو به خوانایی کد کمک می‌کنند.

الگوی Introduce Special Case در موقعیت‌های خاص چک‌کردن‌های متعدد و یکسان در سراسر کد را کم می‌کند, برای مثال چک کردن Null بودن یک آبجکت و داشتن یک رفتار پیش‌فرض در چنین شرایطی و یا خواندن یک سری فیلدهای پیش‌فرض در این شرایط, و الگوی Form Template Method می‌تواند با مشخص‌کردن ساختار کلی عملیات و Pull Up کردن آن به فوق کلاس باعث استفاده مجدد از ساختار کلی شود, و به‌این‌ترتیب

فقط متدهای ریز دانهٔ این ساختار در زیر کلاس‌ها به‌صورت Polymorphic پیاده‌سازی می‌شوند, به کاهش کد تکراری کمک کنند.

به‌این‌ترتیب هر دو الگو به کاهش Code Smell اصلی مدنظر در این موقعیت که Duplicated Code هست, کمک شایانی می‌کنند.

منابع

1. Class slides and lectures
1. 2.Fowler, M., Addison Fowler, M., Refactoring: Improving the Design of Existing Code, Wesley, 1999
2. 3.Refactoring: Improving the Design of Existing Code, 2nd Edition, Addison-Wesley, 2019.
3. Demeyer, S., Ducasse, S., and Nierstrasz, O., Reengineering Patterns, Object Elsevier Science, 2003 .
4. refactoring.guru