دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

کارشناسی ارشد
مهندسی نرم‌افزار

عنوان

الگوها در مهندسی نرم‌افزار

تمرین سوم

نگارش

مهدی عیدی

استاد

دکتر رامان رامسین

تابستان ۱۴۰۴

چکیده

در این تمرین، سه موقعیت توصیف شدند که نشان دهنده مشکلاتی در سیستم هستند. برای هرکدام از موقعیت‌ها دو الگو ارائه شدند. الگوهای ارائه شده برای هر موقعیت تحلیل شده و بیان شد که این الگوها چگونه و در چه شرایطی مشکلات موقعیت مذکور را رفع می‌کنند. سپس الگوهای ارائه شده مقایسه شدند.

کلیدواژه‌ها: بازآرایی، بازمهندسی

فهرست مطالب

فهرست تصاویر

فصل ۱

موقعیت اول

در این فصل، موقعیت اول ارائه شده در تمرین شرح داده شده و بررسی می‌شود. سپس دو الگوی اشاره شده در این موقعیت مورد بررسی قرار می‌گیرند. [۱] [۲] [۳]

۱-۱ بررسی موقعیت

موقعیت: تغییر در ساختارهای توارثی به دلیل انتشار تغییرات مشکل شده است.

انتشار تغییرات، مشکلی در سیستم‌های نرم‌افزاری است که به‌ویژه در ساختارهای توارثی اهمیت می‌یابد. چرا که توارث یکی از انواع قوی جفت‌شدگی[۱] است و می‌تواند منجر به گسترش وسیع تغییرات شود که به نوبه خود تغییرپذیری و قابلیت گسترش برنامه را کاهش می‌دهد. استفاده نادرست از توارث زمانی رخ می‌دهد که توارث صرفا برای استفاده مجدد[۲] از کد به کار رود، بدون توجه جدی به انتشار تغییرات. در این موارد، ممکن است رابطه "Is-A" نقض شده و سلسله‌مراتب کلاس‌ها بیش از حد عمیق یا وسیع می‌شود که توسعه و اعمال تغییرات را بسیار دشوار می‌کند. همچنین، نادیده گرفتن استفاده از چندریختی[۳] نیز می‌تواند به تکرار کد و استفاده مکرر از دستورات شرطی منجر شود که هر دو از عوامل اصلی انتشار تغییرات هستند. در نهایت، این مسائل می‌تواند به بروز نشانه‌های "کد بد"[۴] مانند Refused Bequest منجر شود، جایی که زیرکلاس‌ها مجبور به نادیده گرفتن یا تغییر پیاده‌سازی پدر می‌شوند، که همگی نشان‌دهنده مشکلات جدی در ساختار توارثی و عامل انتشار تغییرات هستند و اصل OCP را نقض می‌کنند. نشان دهنده طراحی نامناسب انتزاع و تلاش

[۱] coupling
[۲] reuse
[۳] polymorphism
[۴] bad smell

برای مدل‌سازی چندین مفهوم جدا به‌صورت یکجا هستند.

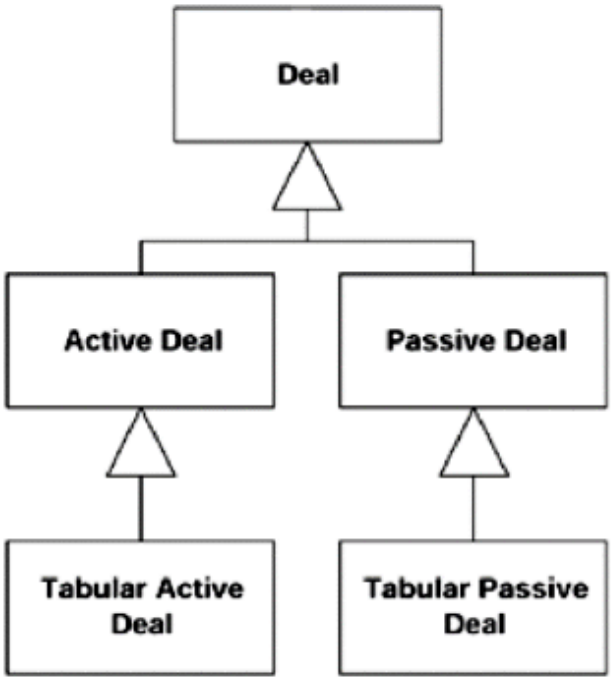
۱-۲ الگوی اول - Tease Apart Inheritance

۱-۲-۱ بررسی الگو

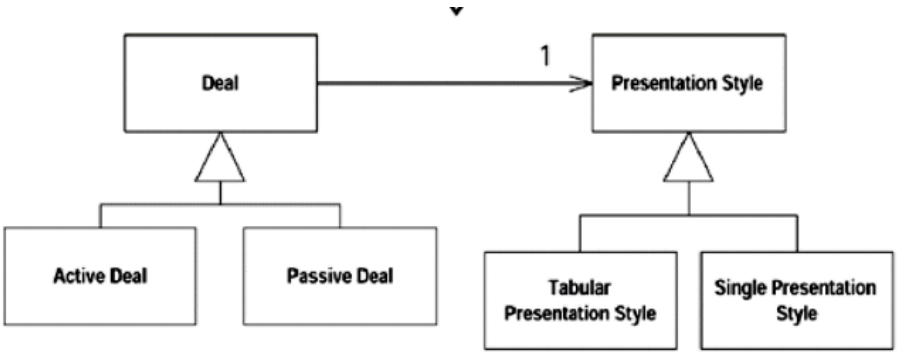
این الگو جزو الگوهای بازآرایی درشت‌دانه بوده و در زمینه‌ی ساختارهای توارثی مرکب مطرح می‌شود. زمانی که چند ساختار توارثی با یکدیگر ترکیب شده‌اند، هدف آن است که این ساختارها از یکدیگر تفکیک شوند. اتفاقی مشابه را می‌توان در الگوی Bridge از مجموعه‌ی GoF مشاهده کرد. در آنجا، specification و پیاده‌سازی در قالب یک ساختار توارثی واحد ترکیب شده بودند. اما الگوی Bridge این دو را از یکدیگر جدا می‌سازد و ارتباط میان آن‌ها را در سطح کلاس پایه برقرار می‌کند. به‌این‌ترتیب، یک سلسله‌مراتب توارث برای abstraction و یک سلسله‌مراتب مستقل برای implementorها تعریف می‌شود. الگوی Bridge در واقع حالت خاصی از این رویکرد است. در این الگو، ممکن است ساختارهای توارثی بنا به دلایلی با هم ترکیب شده باشند. همانند مثالی که در آن موجودیت ''دانشجو'' بررسی شده است. در چنین حالتی، وجوه مختلف یک موجودیت در قالب ساختار توارثی به‌صورت مرکب ظاهر شده‌اند. هدف، جداسازی این وجوه (یا سلسله‌مراتب‌ها) و برقرار کردن ارتباط میان آن‌ها از طریق کلاس پایه است.

در این مسئله، با یک سلسله‌مراتب توارث مواجه هستیم که به‌طور هم‌زمان در حال انجام دو وظیفه‌ی مجزا است. به‌عبارتی، این ساختار دووجهی یا چندوجهی عمل می‌کند و مسئولیت‌های متعددی را به عهده دارد. راه‌حل پیشنهادی آن است که این وظایف یا وجوه مختلف از یکدیگر تفکیک شوند. سپس، ارتباط میان آن‌ها از طریق کلاس پایه برقرار گردد. تعامل بین این ساختارهای تفکیک‌شده نیز از طریق delegation انجام می‌پذیرد. به این معنا که هر بخش با واگذار کردن وظایف به بخش دیگر، خدمات موردنیاز را فراهم می‌سازد و همکاری میان آن‌ها ایجاد می‌شود.

در شکل ۱-۱ مثالی از وضعیت قبل از اعمال این الگو مشاهده می‌شود. در شکل ۱-۲ نیز وضعیت پس از اعمال الگو را می‌توان مشاهده کرد. در این بخش، انواع قرارداد یا معامله تعریف شده‌اند که به دو دسته‌ی کلی «فعال» و «غیرفعال» تقسیم می‌شوند. همچنین، نحوه‌ی نمایش این قراردادها نیز می‌تواند متفاوت باشد. برای مثال، به‌صورت جدول‌محور یا در قالب نمایش ساده. در اینجا، دو جنبه‌ی مجزا، یکی مربوط به ماهیت قرارداد (فعال یا غیرفعال بودن) و دیگری مربوط به شیوه‌ی نمایش آن در یک ساختار توارثی ترکیب شده‌اند. در نتیجه، یک سلسله‌مراتب توارثی دووجهی شکل گرفته است که به‌طور هم‌زمان دو بعد متفاوت از سیستم را پوشش می‌دهد. در وضعیت پس از اعمال الگو، موجودیت «Deal» دارای یک سلسله‌مراتب توارثی مستقل



شکل ۱-۱: مثال پیش از اعمال الگوی Tease Apart Inheritance



شکل ۱-۲: مثال پس از اعمال الگوی Tease Apart Inheritance

است و «نحوه نمایش» نیز در قالب یک ساختار توارثی جداگانه تعریف می‌شود. در این طراحی، وظایف مرتبط با نمایش، از طریق delegation، از سوی شی «Deal» به یک شی از نوع «Presentation Style» که در اختیار دارد، محول می‌گردد. بدین‌ترتیب، مسئولیت‌های نمایشی از ماهیت اصلی قرارداد تفکیک شده‌اند و به یک عنصر مستقل واگذار می‌شوند.

۱-۲-۲ زمان و شرایط مفید بودن اعمال الگو

یکی از نشانه‌های نیاز به جداسازی ساختارهای توارثی، وجود سلسله‌مراتب‌های تودرتو و ترکیبی در برنامه است. این ساختارها معمولا منجر به تکرار پیاده‌سازی ویژگی‌ها در کلاس‌های مختلف می‌شوند و در نتیجه،

اعمال تغییر در یکی از آن‌ها نیازمند تغییر در سایر کلاس‌ها نیز خواهد بود که مصداقی از Shotgun Surgery است. انتشار تغییرات ممکن است مستقیما ناشی از سلسله‌مراتب‌های توارثی تودرتو باشد، که خود نتیجه‌ای از استفاده‌ی نادرست از توارث است. اگر تغییری که مورد نیاز است، افزودن یک زیرکلاس جدید باشد، ممکن است لازم شود که در شاخه‌های مختلفی از سلسله‌مراتب کلاسی صورت گیرد. همچنین، اگر هدف آن باشد که مجموعه‌ای از قابلیت‌ها در انواع خاصی از اشیا تغییر یابند، ممکن است ابتدا نیاز به شناسایی مکان دقیق این قابلیت‌ها در کلاس‌ها داشته باشیم و سپس تغییرات موردنظر را در شاخه‌های مختلف اعمال کنیم. هر دوی این وضعیت‌ها نمونه‌هایی از انتشار تغییرات هستند. ترکیب دو سلسله‌مراتب توارثی مستقل می‌تواند موجب تولید تعداد زیادی زیرکلاس شود که هرکدام حالت خاصی از ترکیب دو ویژگی مستقل را نمایش می‌دهند. این وضعیت باعث افزایش پیچیدگی، تکرار کد، کاهش انسجام و ظهور کلاس‌های بزرگ و فاقد تمرکز مشخص شود (Large Class و Divergent Change). در چنین مواردی، به‌کارگیری این الگو راه‌حل خوبی برای حل مسئله‌ی انتشار تغییرات به شمار می‌رود.

۱-۲-۳ چگونگی مفید بودن الگو در موقعیت

استفاده از این الگو باعث افزایش انسجام، کاهش جفت‌شدگی ناشی از ترکیب پیچیده‌ی چند ساختار توارثی و بهبود انطباق با اصولی مانند OCP، DIP و LSP می‌شود. این الگو با کپسوله‌سازی مناسب، از انتشار تغییرات در میان ساختارهای توارثی جلوگیری کرده و قابلیت نگهداری، خوانایی، تغییرپذیری و توسعه‌پذیری سیستم را افزایش می‌دهد. در مجموع، به‌کارگیری این الگو زمانی مفید است که ترکیب دو ساختار توارثی پیچیده موجب انتشار ناخواسته‌ی تغییرات در سراسر سیستم شده باشد. همچنین نگهداری چندین سلسله‌مراتب توارثی کوچک به وضوح بسیار ساده‌تر از نگهداری یک سلسله‌مراتب بزرگ و عمیق تودرتو است. گسترش قابلیت‌ها نیز به‌سادگی از طریق توسعه‌ی همین سلسله‌مراتب‌های کوچک یا افزودن یک سلسله‌مراتب جدید برای مسئولیت‌های تازه قابل انجام است. نکته‌ی مهم آن است که ساختار حاصل، امکان بهره‌گیری از delegation به‌جای توارث را فراهم می‌سازد که مطابق با اصل CRP است و در نتیجه، سیستم از نظر توسعه‌پذیری بسیار انعطاف‌پذیرتر خواهد بود.

۱-۲-۴ چگونگی پیاده‌سازی یا اعمال الگو

در این الگو، ساختارهای توارثی تودرتو از یکدیگر تفکیک شده و به‌جای آن، چند سلسله‌مراتب توارثی مستقل ایجاد می‌گردد. ارتباط میان این ساختارها به‌جای توارث، از طریق delegation برقرار می‌شود. بدین صورت که هر ساختار، در صورت نیاز به یک سرویس، آن را به ساختار مرتبط دیگری محول می‌کند. این رویکرد

به افزایش انعطاف‌پذیری و کاهش پیچیدگی منجر می‌شود. روند اجرای این الگو به‌صورت مرحله‌ای است: ابتدا عملکرد اصلی موجود در ساختار توارثی تودرتو شناسایی شده و برای آن یک سلسله‌مراتب مستقل تعریف می‌شود. سپس سایر عملکردها به‌تدریج استخراج و در کنار سلسله‌مراتب‌های پیشین قرار می‌گیرند. نتیجه‌ی این فرایند، جلوگیری از تکرار کد، افزایش خوانایی، کاهش وابستگی‌ها و رعایت اصول شی‌گرایی خواهد بود. در نهایت، این تغییر موجب بهبود توسعه‌پذیری ساختار می‌شود.

مراحل:

۱. شناسایی مسئولیت‌ها: ابتدا وظایف مختلفی که سلسله‌مراتب توارثی فعلی بر عهده دارد را مشخص کنید.

۲. انتخاب مسئولیت اصلی: یکی از ابعاد، که مهم‌ترین وظیفه‌ی سلسله‌مراتب فعلی است، نگه داشته می‌شود و سایر ابعاد باید استخراج شوند.

۳. ایجاد کلاس‌های همکار: با استفاده از الگوی Extract Class، کلاس‌هایی را از ساختار فعلی و کلاس پایه‌ی آن استخراج کنید. کلاس استخراج شده به‌عنوان کلاس پایه برای سلسله‌مراتب جدید مورد استفاده قرار می‌گیرد.

۴. ایجاد زیرکلاس‌های جدید: برای هر بعد استخراج شده، زیرکلاس‌هایی مستقل در ساختار جدید تعریف کنید و در همین حین، زیرکلاس‌های متناظر در سلسله‌مراتب اصلی را حذف نمایید.

۵. انتقال رفتارها: با استفاده از الگوی Move Function، متدهای مرتبط با هر بُعد را به سلسله‌مراتب جدید منتقل کنید. پیش از آن اطمینان حاصل کنید که به‌درستی از الگوی Extract Function استفاده شده تا مسئولیت‌ها به‌خوبی تفکیک شده باشند.

۶. بررسی نهایی: روند بالا را ادامه داده و در پایان، اطمینان حاصل کنید که ساختارهای توارثی جدید نیاز به بازآرایی بیشتر مانند Pull Up Method ندارند.

۱-۲-۵ مزایای اعمال الگو

- افزایش انسجام[۵] کلاس‌ها
- کاهش جفت‌شدگی[۶] در ساختار توارث
- رعایت اصل ISP

[۵] cohesion
[۶] coupling

- برقرار ارتباط از فوق کلاس‌های ابسترکت و برقراری DIP
- کمک به برقرار OCP پس از برقراری DIP
- فراهم شدن امکان رعایت اصل CRP
- کاهش تکرار کد
- افزایش قابلیت استفاده مجدد در ساختار جدید
- سادگی و افزایش خوانایی
- کاهش انتشار تغییرات در قسمت ذکر شده
- توسعه‌پذیری و انعطاف‌پذیری ساختار جدید

۱-۲-۶ معایب اعمال الگو

- پیچیدگی در اعمال: از مهم‌ترین مشکلات این الگو، دشواری در پیاده‌سازی آن است. هرچه تعداد ساختارهای توارثی درهم‌تنیده بیشتر باشد، فرایند جداسازی آن‌ها پیچیده‌تر و زمان‌برتر خواهد بود و هرچقدر دیرتر اقدام به این بازآرایی شود، اعمال این الگو به‌مراتب سخت‌تر می‌شود. همچنین پیچیدگی ساختار حاصل از اعمال این الگو باعث افزایش احتمال خطا می‌شود. لازم است پس از اجرای الگو، سیستم به‌دقت بازبینی شده و در صورت مشاهده نشانه‌های کد بد، از الگوهای بازآرایی ریزدانه‌تر برای اصلاح آن‌ها استفاده گردد.
- خطر نقض اصل PLK: در صورت بی‌دقتی، همکاری زنجیره‌ای کلاس‌ها ممکن است منجر به نقض این اصل شود.
- کاهش اندکی کارایی: عملکردی که پیش‌تر در یک کلاس بزرگ متمرکز بود، اکنون از طریق همکاری چندین کلاس کوچک با delegation انجام می‌شود. این امر موجب افزایش تبادلات پیام و در نتیجه کاهش کارایی می‌شود.
- خطر ایجاد وابستگی بالا میان کلاس‌های کوچک: اگر شکست ساختار به‌درستی انجام نشود، ممکن است کلاس‌هایی با وابستگی بالا به‌وجود آید که منجر به بروز مشکلاتی نظیر Feature Envy و Insider Trading شود.
- افزایش تعداد کلاس‌ها: ممکن است باعث افزایش پیچیدگی ساختار کلی شود.

۱-۲-۷ Bad Smell هایی که کاهش می‌یابند

- **Refused Bequest:** با کوچک‌سازی و ساده‌سازی سلسله‌مراتب‌ها، فهم ساختار آسان‌تر شده و احتمال استفاده نادرست از توارث در زیرکلاس‌ها کاهش می‌یابد.
- **Divergent Change:** با جداسازی وظایف در قالب سلسله‌مراتب‌های مستقل، تغییر در یک وظیفه فقط بر سلسله‌مراتب مرتبط با آن تأثیر می‌گذارد.
- **Shotgun Surgery:** در ساختارهای تودرتو، تغییر در یک بخش ممکن است نیازمند تغییرات مشابه درکلاس‌های متعدد باشد. این الگو با شکستن آن ساختارها از چنین تغییرات پراکنده‌ای جلوگیری می‌کند.
- **Repeated Switches:** اگر برای یک مسئولیت خاص از دستور switch استفاده شود، در صورت تکرار زیرکلاس‌سازی، این دستورات در چندین بخش تکرار می‌شوند. استفاده از این الگو باعث حذف این تکرارها می‌شود.
- **Duplicated Code:** با حذف زیرکلاس‌گیری در شاخه‌های مختلف سلسله‌مراتب، پیاده‌سازی‌های مشابه دیگر درکلاس‌های مختلف تکرار نمی‌شوند.
- **Large Class:** تقسیم مسئولیت‌ها به کلاس‌های مستقل، باعث افزایش انسجام و کاهش پیچیدگی کلاس‌های بزرگ می‌شود.
- **Data Class:** کلاس‌هایی که فقط داده نگه می‌دارند و رفتاری ندارند، اغلب محصول وجود کلاس‌های بزرگ هستند. با افزایش انسجام و جابجایی رفتار به کنار داده‌ها، از ایجاد چنین کلاس‌هایی جلوگیری می‌شود.

۱-۲-۸ Bad Smell هایی که ممکن است بروز یابند

- Feature Envy
- Insider Trading
- Lazy Element
- Message Chains
- Middle Man
- Long Parameter List

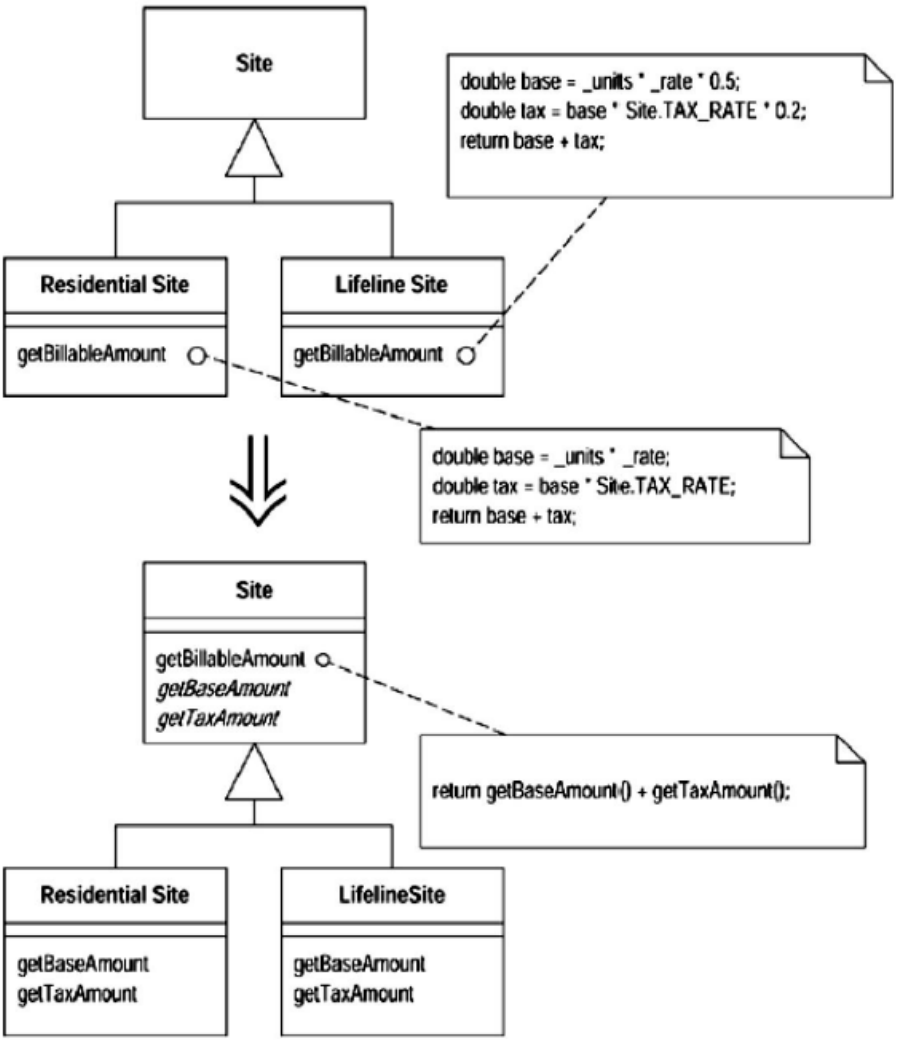
۱-۲-۹ الگوهای مرتبط

- Extract Class
- Extract Function
- Move Function
- Pull-Up Method/Field
- Replace Superclass with Delegate
- Bridge

۱-۳ الگوی دوم - Form Template Method

۱-۳-۱ بررسی الگو

این الگو یکی از الگوهای رفتاری GoF و در دسته Dealing with Inheritance الگوهای بازآرایی است که در شرایطی کاربرد دارد که چند متد مختلف، اگرچه در جزئیات پیاده‌سازی تفاوت دارند، اما در ساختار کلی اجرای خود مشترک هستند. فرض کنید در چند کلاس مختلف، متدهایی با نام و معنای مشابه (مثلا draw) وجود دارد که هر کدام شامل سه بخش مشخص هستند: عملیاتی قبل از یک حلقه، عملیاتی داخل حلقه و عملیاتی بعد از حلقه. اگرچه این سه بخش در کلاس‌های مختلف متفاوت پیاده‌سازی شده‌اند، اما ساختار کلی این متدها کاملا مشابه است. این الگو این امکان را فراهم می‌کند که ساختار کلی الگوریتم به فوق کلاس منتقل شود. برای این کار، ابتدا در کلاس‌های مختلف، بخش‌های متفاوت متد مثلا به صورت سه متد مجزا (مثلا a، b، c) تعریف شده و سپس در متد draw این سه متد به‌ترتیب (قبل، حین، و بعد از حلقه) فراخوانی می‌شوند. در نتیجه، نسخه‌های مختلف متد draw در تمام کلاس‌ها یکسان شده و می‌توانند به‌عنوان یک متد مشترک در فوق کلاس قرار گیرند. در این مرحله، متد draw به کلاس پایه منتقل شده و متدهای a، b، c به صورت تعاریف انتزاعی در کلاس پایه تعریف می‌شوند. پیاده‌سازی این متدها به کلاس‌های فرزند واگذار شده و از طریق polymorphism رفتار خاص هر کلاس حفظ می‌شود. هدف این الگو، انتقال شباهت‌های ساختاری موجود در متدهای مختلف به کلاس پایه است، حتی اگر این شباهت تنها در حد چارچوب کلی الگوریتم باشد. در نتیجه، تکرار (duplication) ساختاری حذف می‌شود. این الگو به‌صورت class scope عمل می‌کند و مبتنی بر وراثت بوده و در زمان کامپایل تحقق می‌یابد. متدی که در کلاس پایه ساختار کلی الگوریتم را مشخص می‌کند، template method نامیده می‌شود.



شکل ۱-۳: مثال الگوی Form Template Method

در شکل ۱-۳ مثالی از اعمال این الگو مشاهده می‌شود. در این مسئله، با کلاسی به نام Site روبرو هستیم که نمایانگر یک آدرس است. این آدرس می‌تواند مربوط به محل سکونت یا محل کسب‌وکار باشد. در هر دو حالت، هدف محاسبه مبلغ قابل پرداخت است، اما جزئیات محاسبه در هر حالت متفاوت است. متدی به نام getBillableAmount به کلاس پایه منتقل شده که ساختار کلی محاسبه را مشخص می‌کند. در نتیجه، ساختار کلی محاسبه در کلاس پایه متمرکز شده و تنها جزئیات متغیر در کلاس‌های فرزند تعریف می‌شوند.

۱-۳-۲ زمان و شرایط مفید بودن اعمال الگو

در مواردی که متدهای موجود در زیرکلاس‌های یک ساختار توارثی، از نظر معنایی و ترتیب کلی گام‌های پردازشی مشابه هستند، اما در جزئیات تفاوت دارند، تکرارهایی در کد مشاهده می‌شود که نشانه‌ای از کد بد

(مانند تکرار کد) است. این تکرار می‌تواند نشان دهد که زمان به‌کارگیری این الگو فرارسیده است. در چنین شرایطی، می‌توان ساختار کلی مشترک این متدها را به یک فوق کلاس انتزاعی منتقل کرد. این کار باعث حذف تکرار، بهبود خوانایی و افزایش قابلیت نگهداری کد می‌شود. در غیر این صورت، هرگونه تغییر در ساختار کلی پردازش ناچارا باید در تمامی زیرکلاس‌ها اعمال شود، که موجب انتشار تغییرات و کاهش انعطاف‌پذیری خواهد شد. بنابراین، این الگو زمانی مناسب است که وجود تکرار در مراحل کلی متدها در زیرکلاس‌ها مشاهده شود.

۱-۳-۳ چگونگی مفید بودن الگو در موقعیت

پس از اعمال الگو، هرگاه نیاز به تغییر در منطق کلی و ساختار کلی باشد، این تغییرات تنها در template method موجود در کلاس پایه اعمال می‌شوند و این از انتشار تغییرات به زیرکلاس‌ها جلوگیری می‌کند. در حالت قبل نیاز بود تا این تغییر در تمام زیرکلاس‌ها اعمال شود که اکنون جلوی این کار گرفته شده است. همچنین اگر این ساختار کلی مدام نیاز به تغییر داشته باشد، حسن این الگو بیشتر به چشم می‌آید. با این حال، در برخی موارد ممکن است این رویکرد کافی نباشد. به‌ویژه اگر تغییرات مستلزم افزودن یک گام جدید باشند که در همه زیرکلاس‌ها به‌صورت یکسان قابل پیاده‌سازی نیست. در چنین شرایطی، گام جدید باید به‌صورت پلی‌مورفیک در تمامی زیرکلاس‌ها پیاده‌سازی شود.

۱-۳-۴ چگونگی پیاده‌سازی یا اعمال الگو

پس از شناسایی تکرار کد با ساختار کلی مشابه در متدهای زیرکلاس‌ها، این ساختار کلی به کلاس پدر منتقل می‌شود. اگر بخش مشترک تنها بخشی از بدنه متد را شامل شود، ابتدا باید آن بخش به کمک الگوی Extract Function جدا شود. سپس متدی که ساختار کلی را در بر دارد، به‌عنوان یک متد قالب در فوق کلاس تعریف می‌شود. گام‌های پردازشی این متد نیز به‌صورت specification در فوق کلاس تعریف شده و پیاده‌سازی آن‌ها در زیرکلاس‌ها انجام می‌شود. هر زیرکلاس بر اساس نیاز خود، این عملیات را پیاده‌سازی می‌کند.

مراحل:

۱. ابتدا با به‌کارگیری الگوی Extract Function، ساختار کلی مشترک کد موجود در زیرکلاس‌ها را به گام‌های مجزا تقسیم کرده و هر گام را در قالب یک متد مستقل پیاده‌سازی کنید.

۲. با استفاده از الگوی Pull-Up Method، متد اصلی ساختار مشترک را به فوق کلاس منتقل کنید.

۳. متدهایی که همچنان متفاوت باقی مانده‌اند و گام‌ها را تشکیل می‌دهند را به یک نام یکسان و معنادار تغییر نام دهید تا هماهنگی بین آن‌ها حفظ شود.

۴. برای هر یک از این متدهای متفاوت، در فوق کلاس یک specification تعریف کنید. پیاده‌سازی این متدها باید همچنان در زیرکلاس‌ها باقی بماند.

۵. زیرکلاس‌ها را بازنویسی کنید تا فراخوانی‌ها تصحیح شوند.

۶. در نهایت کد جدید را کامپایل و تست کنید.

۱-۳-۵ مزایای اعمال الگو

- از بین بردن duplication در حد ساختار کلی مشابه
- انتقال ساختار کلی مشابه به فوق کلاس باعث می‌شود تغییرات لازم در این ساختار صرفا در فوق کلاس اعمال شده و این تغییر به زیرکلاس‌ها منتشر نمی‌شود
- افزایش انعطاف‌پذیری و قابلیت گسترش
- امکان اضافه کردن پیاده‌سازی‌های جدید از گام‌ها در زیرکلاس‌ها به‌صورت منعطف (OCP)
- رعایت چندریختی
- افزایش خوانایی در ساختار کلی کد

۱-۳-۶ معایب اعمال الگو

- کاهش اندکی کارایی در اثر افزایش فراخوانی‌ها و تبادل پیام
- ممکن است استخراج ساختار کلی و گام‌ها دشوار و پیچیده باشد
- نیاز به نام‌گذاری دقیق و معنا دار در specification ها که می‌تواند دشوار باشد
- ممکن است ساختار کلی مشترک به اندازه کافی نباشد یا دقیقا مشترک نباشد و محاسن این الگو به‌صورت کامل محقق نشوند
- نیاز به دقت به رعایت اصل LSP
- کاهش خوانایی گام‌ها زیرا درک منطق هر گام نیاز به مطالعه جداگانه متد پیاده‌سازی شده آن گام دارد

۱-۳-۷ Bad Smell هایی که کاهش می‌یابند

- Duplicated Code: با استخراج ساختار کلی مشابه

- Long Function: گام‌های اجرایی در ساختار کلی در قالب متدهای کوچک‌تر پیاده‌سازی می‌شوند
- Repeated Switches: در صورت وجود سوییچ در ساختار کلی مشترک در متدها در زیرکلاس‌ها، این ساختار به فوق کلاس منتقل شده و منطق کلی فقط در فوق کلاس تعریف می‌شود
- Shotgun Surgery: ساختار کلی در فوق کلاس تعریف شده و اعمال تغییرات در این ساختار به فرزندان منتشر نمی‌شود

۱-۳-۸ Bad Smell هایی که ممکن است بروز یابند

- Refused Bequest
- Long Parameter List
- Data Clumps

۱-۳-۹ الگوهای مرتبط

- Pull-Up Method
- Extract Function
- Template Method (GoF)

۱-۴ مقایسه الگوها

الگوی Tease Apart Inheritance زمانی به‌کار می‌رود که مرکب و تو در تو بودن سلسله‌مراتب وراثت عامل اصلی انتشار تغییرات باشد. در مقابل، الگوی Form Template Method هنگامی استفاده می‌شود که تغییرات در ساختار کلی مشترک متدها در میان زیرکلاس‌ها انتشار می‌یابد و در تمام زیرکلاس‌ها باید اعمال شود، ولی این ساختار کلی قابلیت استخراج و تبدیل به یک متد قالب را دارد. الگوی اول به جداسازی ساختارهای توارثی و استفاده از delegation برای دریافت خدمات تاکید داشته و الگوی دوم به استخراج ساختار کلی مشترک و استفاده از چندریختمی تاکید دارد. مزایا و معایب اعمال هر الگو در بخش‌های جداگانه بررسی شده‌اند. الگوی اول از الگوهای درشت‌دانه بازآرایی است. اجرای الگوی اول دشوارتر ولی تأثیرگذاری گسترده‌تری در جلوگیری از انتشار تغییرات دارد.

فصل ۲

موقعیت دوم

در این فصل، موقعیت دوم ارائه شده در تمرین شرح داده شده و بررسی می‌شود. سپس دو الگوی اشاره شده در این موقعیت مورد بررسی قرار می‌گیرند.

۲-۱ بررسی موقعیت

موقعیت: دستورات شرطی سوئیچ در متدهای کلاس‌ها تکرار شده‌اند.

در برنامه‌نویسی شی‌گرا، استفاده‌ی مکرر از ساختارهای سوییچ معمولا نشانه‌ای از عدم بهره‌گیری صحیح از Polymorphism است. هنگامی که منطق برنامه بر اساس بررسی نوع شی پیاده‌سازی می‌شود، چه به‌صورت بررسی نوع شی جاری یا نوع شی مشتری، و بر اساس آن رفتار متفاوتی انتخاب می‌گردد، در واقع می‌توانستیم از قابلیت‌های چندریختی به‌جای آن استفاده کنیم. در چنین شرایطی، وجود یک یا چند سوییچ، به‌ویژه سوییچ‌های تکراری که ساختار تصمیم‌گیری مشابهی دارند اما بدنه‌های پیاده‌سازی‌شان متفاوت است، می‌تواند نشانه‌ای از یک طراحی ناکارآمد باشد. این تکرار ممکن است به بروز Code Duplication یا Code Bloat بینجامد. بنابراین، در طراحی شی‌گرا، مشاهده‌ی سوییچ باید ما را به دقت وادارد. شاید بتوان با اعمال اصول شی‌گرایی، به‌ویژه چندریختی، کد را تمیزتر، منعطف‌تر و قابل توسعه‌تر طراحی کرد. همچنین اگر سوئیچ‌های تکرار شده دارای منطق مشابه یا یکسان باشند، این امر باعث افزایش جفت‌شدگی [۱] در قسمت‌های بکار رفته می‌شود. استفاده بی‌رویه از سوئیچ و تکرار آن در متدها باعث کاهش شدید خوانایی و درک منطق برنامه نیز می‌شود.

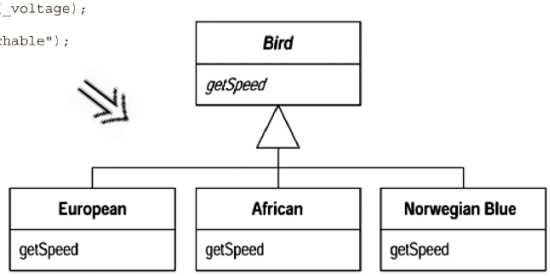
[۱] coupling

۲-۲ الگوی اول - Replace Conditional with Polymorphism

۲-۲-۱ بررسی الگو

در طراحی شیگرا، این الگو از اهمیت بالایی برخوردار است. در برخی موارد، وجود عبارات شرطی باید ما را به استفاده از چندریختی سوق دهد. بهویژه زمانی که در کد، نوع یک شی بررسی میشود و بر اساس آن، رفتارهای متفاوتی اعمال میگردد. در چنین شرایطی، مشاهدهی چککردن نوع شی باید زنگ خطری باشد که نشان دهد امکان پیادهسازی این رفتارها به صورت چندریختی وجود دارد. در واقع، کلاسی که اکنون حاوی این منطق شرطی است، باید دارای زیرکلاسهایی شود که هر یک از آنها، بخشی از این رفتار را بهتناسب نوع خود پیادهسازی کنند. به این ترتیب، منطق شرطی به رفتارهای توزیع شده در زیرکلاسها تبدیل میشود و از طریق چندریختی مدیریت میگردد. این کار ضمن افزایش خوانایی و انعطافپذیری، از تکرار کد و پیچیدگی غیرضروری نیز جلوگیری میکند. در این الگو شما عبارتی شرطی دارید که بسته به نوع شی یا ویژگیهای آن، اقدامات مختلفی را انجام میدهد. برای حل آن، زیرکلاسهایی مطابق با شاخههای عبارت شرطی ایجاد کنید. در هر زیرکلاس، یک متد مشترک تعریف کرده و کد مربوط به شاخهی متناظر شرط را به آن منتقل کنید. سپس عبارت شرطی را با فراخوانی آن متد جایگزین نمایید. نتیجه این خواهد بود که پیادهسازی مناسب بر اساس کلاس شی و از طریق چندریختی بهدرستی انجام خواهد شد. در بازمهندسی، موضوع Client Type Check نیز مطرح میشود. در این وضعیت، معمولا ساختار زیرکلاسها از پیش موجود است، اما مشتری اقدام به بررسی نوع زیرکلاسهای سرویسدهنده میکند و بر اساس نوع آنها، دستورات مربوطه را اجرا مینماید. در این حالت، میتوان منطق مربوط به این دستورات را به خود زیرکلاسها منتقل کرد. در نتیجه، مشتری دیگر نیازی به شناسایی زیرکلاسها و بررسی نوع آنها ندارد و تنها با کلاس پدر در تعامل خواهد بود. این کار وابستگی را کاهش داده و اصل DIP را برقرار میکند.

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
           return getBaseSpeed();
        case AFRICAN:
           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
           return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```



شکل ۲-۱: مثال اعمال الگوی Replace Conditional with Polymorphism

در شکل ۲-۱ مثالی از اعمال این الگو مشاهده میشود. در حالت قبل، داخل متد getSpeed دستور سوییچ

وجود داشت که براساس نوع پرنده، عملیات متناظر اجرا می‌شود. در حالت پس از اعمال الگو، زیرکلاس‌هایی برای پرنده تعریف شده و متد getSpeed در زیرکلاس‌ها چندریختی می‌شود و پیاده‌سازی‌های متناسب اعمال می‌شود.

۲-۲-۲ زمان و شرایط مفید بودن اعمال الگو

این الگوی بازآرایی زمانی مفید است که کد شامل شرط‌هایی باشد که بر اساس نوع کلاس شی یا اینترفیس پیاده‌سازی‌شده، مقدار یکی از فیلدهای شی یا نتیجه فراخوانی یکی از متدهای شی رفتارهای متفاوتی را اجرا می‌کنند. در چنین حالتی، اگر ویژگی جدیدی به شی اضافه شود یا نوع جدیدی تعریف گردد، لازم است که تمام شرط‌های مشابه در بخش‌های مختلف کد را یافته و تغییر دهید. بنابراین، هرچه این شرط‌ها در متدهای مختلف شی پراکنده‌تر و تکراری باشند، سودمندی این الگو (یعنی جایگزینی شرط‌ها با چندریختی) بیشتر خواهد بود. زیرا تغییرات متمرکز و قابل‌مدیریت‌تر می‌شوند. در زمانی هم که بررسی نوع مشتری انجام شود و ساختار توارثی موجود باشد، اعمال این الگو و بردن رفتار متناسب به زیرکلاس‌های موجود و حذف عبارات شرطی از سمت مشتری نیز مفید خواهد بود.

۲-۲-۳ چگونگی مفید بودن الگو در موقعیت

این الگو از اصل Tell-Don’t-Ask پیروی می‌کند. به جای آنکه وضعیت یک شی را بررسی کرده و سپس براساس آن تصمیم‌گیری کنید، بهتر است وظیفه مورد نظر را مستقیما به خود شی بسپارید تا براساس منطق داخلی‌اش آن را اجرا کند. از سوی دیگر، این تکنیک باعث حذف کد تکراری می‌شود، زیرا بسیاری از شرط‌های تقریبا مشابه و تکراری را حذف می‌کند. همچنین، در صورت نیاز به افزودن رفتار جدید، تنها کافی است یک زیرکلاس جدید تعریف کنید، بدون آنکه نیازی به تغییر در کد موجود باشد. این ویژگی در راستای اصل OCP عمل می‌کند. همچنین در وضعیت بررسی نوع مشتری، معمولا ساختار زیرکلاس‌ها از پیش موجود است، اما مشتری اقدام به بررسی نوع زیرکلاس‌های سرویس‌دهنده می‌کند و بر اساس نوع آن‌ها، دستورات مربوطه را اجرا می‌نماید. در این حالت، می‌توان منطق مربوط به این دستورات را به خود زیرکلاس‌ها منتقل کرد. در نتیجه، مشتری دیگر نیازی به شناسایی زیرکلاس‌ها و بررسی نوع آن‌ها ندارد و تنها با کلاس پدر در تعامل خواهد بود. این کار وابستگی را کاهش داده و اصل DIP را برقرار می‌کند. در مجموع این الگو باعث بکارگیری چندریختی، تحقق OCP و DIP و افزایش انعطاف‌پذیری و خوانایی می‌شود.

۲-۲-۴ چگونگی پیاده‌سازی یا اعمال الگو

برای استفاده از این الگو، باید یک سلسله‌مراتب از کلاس‌ها در اختیار داشته باشید که رفتارهای جایگزین در آن‌ها پیاده‌سازی شوند. اگر چنین ساختاری وجود ندارد، باید ابتدا آن را ایجاد کنید. برای این کار، می‌توان از این تکنیک‌ها کمک گرفت: ۱ـ جایگزینی کد نوع با زیرکلاس‌ها: برای هر مقدار از یک ویژگی مشخص در شی، یک زیرکلاس ساخته می‌شود. این روش ساده است، اما انعطاف‌پذیری کمتری دارد، زیرا تنها برای همان ویژگی قابل استفاده است و نه سایر ویژگی‌های شی. ۲ـ جایگزینی کد نوع با الگوی حالت یا استراتژی: در این روش، یک کلاس مجزا برای یک ویژگی خاص از شی تعریف می‌شود و زیرکلاس‌هایی برای مقادیر مختلف آن ویژگی ساخته می‌شوند. کلاس اصلی به این کلاس‌ها ارجاع می‌دهد و اجرای رفتار را به آن‌ها واگذار می‌کند.

مراحل بعدی اجرای بازآرایی، فرض را بر این می‌گذارند که این سلسله‌مراتب از پیش ایجاد شده است.

مراحل:

۱. اگر عبارت شرطی درون متدی قرار دارد که علاوه‌بر آن، اعمال دیگری نیز انجام می‌دهد، ابتدا با استفاده از الگوی Extract Method بخش مرتبط با شرط را جدا کنید.

۲. سپس برای هر زیرکلاس در سلسله‌مراتب، متد حاوی شرط را بازنویسی کرده و کد متناظر با شاخه‌ی مربوط به آن زیرکلاس را در آن قرار دهید.

۳. پس از انتقال کد هر شاخه به زیرکلاس مربوطه، آن شاخه را از عبارت شرطی حذف کنید.

۴. این روند را تا زمانی ادامه دهید که شرط کاملا خالی شود. در این مرحله، عبارت شرطی را حذف کرده و متد اصلی را به‌صورت انتزاعی در کلاس پایه اعلام کنید.

۲-۲-۵ مزایای اعمال الگو

- افزایش خوانایی کد به دلیل جداکردن رفتارها و انتقال به زیرکلاس‌های مربوطه. سوییچ‌های تکراری و پیچیده خوانایی پایینی دارند.
- امکان رعایت اصل DIP و OCP. به‌عبارتی می‌توان زیرکلاس‌های جدید که حاوی رفتار جدید هستند به مجموعه اضافه کرد و مشتری با فوق کلاس در ارتباط خواهد بود.
- کاهش جفت‌شدگی که ناشی از تکرار کد به‌وجود آمده بود.
- کاهش انتشار تغییرات به‌دلیل این که تغییری ممکن بود نیاز شود در شاخه‌های مختلف سوییچ‌های مختلف اعمال شود.

- تسهیل قابلیت نگهداری و قابلیت آزمون کد.

۲-۲-۶ معایب اعمال الگو

- افزایش تعداد کلاس‌ها. انواع متغیر حتی کوچک و شاخه‌های شرطی نیاز به تعریف زیرکلاس خواهند داشت.
- افزایش تعداد کلاس‌ها به معنی افزایش بار مراقبت و نگهداری است.
- برای اعمال الگو نیاز است تا ساختار توارثی وجود داشته باشد و اگر موجود نباشد باید چنین ساختاری را ایجاد کرد.
- رفتار یک شی توسط کلاس آن هنگام ایجاد تعیین می‌شود و در زمان اجرا قابل تغییر نیست. اگر یک مهندس به مدیر ارتقا یابد، نمی‌توانید به سادگی رفتار شی را تغییر دهید. باید یک شی مدیر جدید ایجاد کنید تا جایگزین شی مهندس قدیمی شود.
- پخش شدن منطق. منطقی که قبلا در سوییچ جمع شده بود، اکنون بین زیرکلاس‌های متعدد پخش می‌شود و برای درک تمام منطق باید پیاده‌سازی‌های تمام زیرکلاس‌ها را خواند.

۲-۲-۷ Bad Smell هایی که کاهش می‌یابند

- Repeated Switches: توضیح داده شد که عبارات شرطی پیچیده با چندریختی جایگزین می‌شوند.
- Duplicated Code: کدهای تکراری در شاخه‌ها و سوییچ‌های تکراری حذف می‌شوند.
- Shotgun Surgery: ممکن بود یک تغییر نیاز به اعمال آن در شاخه‌های مختلف شرطی داشته باشد به‌دلیل جفت‌شدگی و تکرار کد.

۲-۲-۸ Bad Smell هایی که ممکن است بروز یابند

- Speculative Generality: در مواردی که تنوع واقعی در رفتار وجود ندارد یا فقط یک یا دو مورد خاص هستند، استفاده از این الگو ممکن است به انتزاع‌های بی‌مورد و آماده‌سازی برای تغییرات غیرواقعی منجر شود.

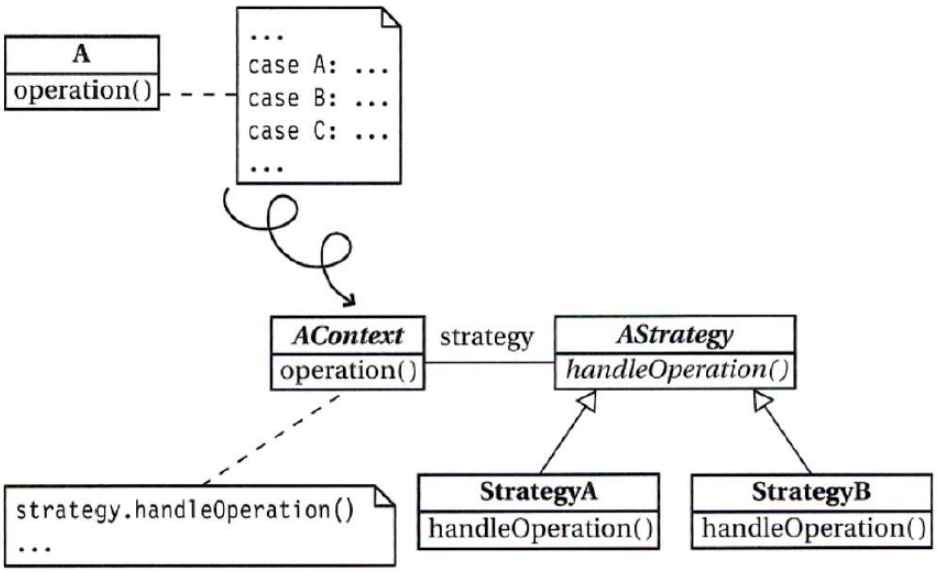
۲-۲-۹ الگوهای مرتبط

- State (GoF)
- Strategy (GoF)
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Transform Conditionals to Polymorphism Patterns (Reengineering)

۲-۳ الگوی دوم - Factor Out Strategy

۲-۳-۱ بررسی الگو

در این الگو، هدف اصلی تغییر الگوریتم اجرا شده بر اساس مقدار یک attribute از شی یا شرایط محیطی است. برای دستیابی به این هدف، از الگوی Strategy استفاده می‌شود. در پیاده‌سازی این الگو، یک شی استراتژی تعریف می‌شود که الگوریتم به آن منتقل می‌گردد. انواع مختلف الگوریتم‌ها به‌صورت زیرکلاس‌هایی از کلاس Strategy پیاده‌سازی شده و در زمان اجرا، یک نمونه‌ی مناسب از آن‌ها به Context اختصاص داده می‌شود تا بتواند رفتار متناظر را اجرا کند. در الگوی Strategy، معمولا انتخاب و تخصیص استراتژی به Context بر عهده یک عامل ثالث است، برخلاف الگوی State که در آن، تغییر وضعیت معمولا درون خود شی وضعیت صورت می‌گیرد. البته در برخی موارد، خود Context می‌تواند استراتژی مناسب را بر اساس وضعیت داخلی‌اش انتخاب کند، مشروط بر اینکه نمونه‌های استراتژی‌ها از پیش موجود باشند. در چنین حالتی، اگرچه الگوریتم از Context جدا شده است، ولی عبارات سوییچ حذف نمی‌شوند و همچنان وجود دارند، که با چندریختی سازگار نیست و ما میل داریم از چندریختی استفاده کنیم. برای حذف کامل عبارات شرطی و دستیابی به چندریختی واقعی، باید وظیفه‌ی انتخاب استراتژی مناسب را به عاملی مستقل از Context واگذار کرد. در این صورت، Context صرفا با رابط Strategy تعامل دارد و از جزئیات زیرکلاس‌های آن بی‌اطلاع است، که موجب رعایت اصل DIP می‌شود. با این حال، این روش باعث ایجاد جفت‌شدگی بین عامل ثالث و کل سلسله‌مراتب استراتژی‌ها خواهد شد. در شکل ۲-۲ مثالی از این الگو مشاهده می‌شود.



شکل ۲-۲: مثال الگوی Factor Out Strategy

۲-۳-۲ زمان و شرایط مفید بودن اعمال الگو

این الگو زمانی مفید است که در قسمت‌هایی از کد نیاز به تغییر الگوریتم اجرایی وجود دارد که بر اساس مقدار اتریبیوت، شرایط محیطی مانند زمان و حافظه یا حتی ورودی کاربر انجام می‌شود. ولی این کار در قسمت‌های مختلف کد با استفاده از دستورات شرطی یا سوییچ‌ها انجام می‌شوند که وضعیت را بررسی کرده و الگوریتم متناسب اجرا می‌شود. در این موارد استفاده از این الگو باعث می‌شود که چندریختی را بکار برده و الگوریتم‌های مختلف را به عنوان زیرکلاس‌هایی از یک فوق کلاس به نام استراتژی منتقل کنیم و رابطه بین کانتکست و استراتژی را نیز در سطح فوق کلاس برقرار کنیم که باعث تحقق اصول DIP و OCP شده و الگوریتم‌ها توسعه‌پذیر می‌شوند. دستورات سوییچ نیز در کانتکست از بین می‌روند البته مشروط بر این که تغییر الگوریتم توسط ثالث انجام بپذیرد که توضیح آن قبل‌تر داده شد. در نتیجه وجود الگوریتم‌های متنوع و شرایط مختلف برای اجرای آن‌ها باعث مفید بودن اعمال این الگو و حذف دستورات شرطی پیچیده، تکرار کد و جفت‌شدگی می‌شود.

۲-۳-۳ چگونگی مفید بودن الگو در موقعیت

در حالت پیشین، دستورات شرطی پیچیده یا سوییچ‌ها در قسمت‌های مختلف کد برای بررسی شرایط مانند مقادیر اتریبیوت‌ها یا شرایط محیطی وجود داشتند تا براساس شرایط، الگوریتم اجرایی متناسب را اجرا کنند. در این شرایط تکرار کد، ناخوانایی، جفت‌شدگی، عدم انعطاف‌پذیری و قابلیت توسعه و غیره وجود داشت که پس از اعمال این الگو، از چندریختی استفاده شد و الگوریتم‌های متعدد به عنوان زیرکلاسی از فوق کلاس استراتژی

تعریف شدند. کانتکست نیز با این فوق کلاس در ارتباط بوده و در نتیجه DIP و OCP برقرار می‌شود که باعث کاهش جفت‌شدگی و افزایش انعطاف‌پذیری و قابلیت توسعه می‌شود. همچنین به‌دلیل حذف سوییچ‌ها و دستورات شرطی پیچیده، خوانایی نیز افزایش یافته و از تکرار این سوییچ‌ها نیز جلوگیری می‌کند که بدین شکل این الگو در این شرایط مفید واقع می‌شود.

۲-۳-۴ چگونگی پیاده‌سازی یا اعمال الگو

برای اعمال این الگو، مراحل زیر انجام می‌شود:

۱. شناسایی واسط استراتژی: ابتدا باید رفتارهایی که قرار است قابل تغییر باشند (الگوریتم‌ها) شناسایی شوند و رابط مشترک آن‌ها استخراج گردد.

۲. تعریف کلاس Strategy: یک کلاس انتزاعی با نام مثلا Strategy ایجاد می‌شود که رابط شناسایی‌شده را نمایش می‌دهد.

۳. ایجاد پیاده‌سازی‌های مختلف الگوریتم: برای هر الگوریتم شناسایی‌شده، یک زیرکلاس از استراتژی تعریف می‌شود.

۴. انتقال پیاده‌سازی الگوریتم‌ها: هر یک از کلاس‌های استراتژی باید متدهای واسط را پیاده‌سازی کرده و کد مربوط به شاخه‌ی شرطی معادل خود را به درون این متدها منتقل کنند.

۵. افزودن متغیر Strategy به کلاس Context: در کلاس کانتکست (کلاسی که از استراتژی استفاده می‌کند)، یک متغیر نمونه افزوده می‌شود تا به استراتژی جاری اشاره کند.

۶. (در صورت نیاز) افزودن ارجاع معکوس: ممکن است لازم باشد در کلاس‌های استراتژی، ارجاعی به Context نیز موجود باشد تا بتوانند به اطلاعات Context دسترسی یابند.

۷. مقداردهی اولیه به استراتژی پیش‌فرض: متغیر استراتژی باید با یک نمونه پیش‌فرض مقداردهی اولیه شود.

۸. بازنویسی متدهای Context: متدهایی که پیش‌تر شامل عبارت شرطی بودند باید اصلاح شوند تا به جای بررسی شرط، فراخوانی به متغیر استراتژی صورت گیرد و وظیفه‌ی تصمیم‌گیری به شی استراتژی واگذار شود.

گام چهارم می‌تواند با الگوی Extract Function انجام شود. پس از هر مرحله، تست‌های رگرسیون باید بدون خطا اجرا شوند. مرحله حیاتی، آخرین مرحله است که رفتار به اشیا استراتژی واگذار می‌شود.

۲-۳-۵ مزایای اعمال الگو

- اعمال این الگو تأثیر کمی بر رابط عمومی کلاس اصلی دارد، چرا که اشیای استراتژی از طریق delegation توسط کلاس Context فراخوانی می‌شوند. در نتیجه، کدهای مشتری معمولا نیازی به تغییر ندارند.
- چون پیاده‌سازی الگوریتم‌ها به کلاس‌های Strategy منتقل می‌گردد، کلاس Context کوچک‌تر می‌شود.
- انسجام در کانتکست و استراتژی‌ها افزایش می‌یابد.
- جفت‌شدگی میان کانتکست و الگوریتم‌ها کاهش می‌یابد. همچنین جفت‌شدگی به‌دلیل تکرار دستورات شرطی نیز کاهش می‌یابد و جلوی انتشار تغییرات در این قسمت گرفته می‌شود.
- اصول DIP و OCP میان کانتکست و استراتژی برقرار می‌شود که باعث کاهش جفت‌شدگی و افزایش قابلیت توسعه می‌شود که الگوریتم‌های متعدد را بتوان اضافه کرد با تبعات اندک.
- خوانایی کانتکست و الگوریتم‌ها افزایش می‌یابد و دستورات شرطی پیچیده و سوییچ‌ها از بین می‌رود.
- استفاده از چندریختی.

۲-۳-۶ معایب اعمال الگو

- جفت‌شدگی بالای عامل ثالث با کل مجموعه.
- درصورت سپردن مسئولیت تغییر الگوریتم به خود کانتکست، سوییچ‌ها از بین نمی‌روند پس وجود ثالث الزامی است.
- افزایش تعداد کلاس‌ها برای پیاده‌سازی الگوریتم‌ها باعث افزایش سربار مراقبت و نگهداری می‌شود.
- استفاده از delegation باعث کاهش کارایی و افزایش تبادل پیام می‌شود.
- افزایش اشیا در حین اجرا باعث فشار بر منابع حافظه می‌شود.
- نیاز به ثالث برای مدیریت چرخه‌حیات استراتژی‌ها.

۲-۳-۷ Bad Smell هایی که کاهش می‌یابند

- Repeated Switches: توضیح داده شد که عبارات شرطی پیچیده با چندریختی جایگزین می‌شوند.
- Duplicated Code: کدهای تکراری در شاخه‌ها و سوییچ‌های تکراری حذف می‌شوند.

- Shotgun Surgery: ممکن بود یک تغییر نیاز به اعمال آن در شاخه‌های مختلف شرطی داشته باشد به‌دلیل جفت‌شدگی و تکرار کد.

۲-۳-۸ Bad Smell هایی که ممکن است بروز یابند

- Speculative Generality: در مواردی که تنوع واقعی در رفتار وجود ندارد یا فقط یک یا دو مورد خاص هستند، استفاده از این الگو ممکن است به انتزاع‌های بی‌مورد و آماده‌سازی برای تغییرات غیرواقعی منجر شود.
- Long Parameter List: نیاز است تا در فوق کلاس استراتژی، واسط مشترک زیرکلاس‌ها در نظر گرفته شود که ممکن است باعث شود پارامترهای زیادی به آن ارسال شوند که حتی شاید برخی از آن‌ها برای استراتژی فعلی نیاز نباشند.

۲-۳-۹ الگوهای مرتبط

- Factor Out State
- Extract Function
- State (GoF)
- Strategy (GoF)
- Replace Conditional with Polymorphism

۲-۴ مقایسه الگوها

هر دو الگو به‌منظور حذف ساختارهای شرطی و سوییچ‌های تکراری و بهبود طراحی شیگرا به‌کار می‌روند، اما تفاوت‌هایی در زمان انتخاب رفتار، ساختار پیاده‌سازی و انعطاف‌پذیری دارند. هر دو الگو شباهت‌هایی دارند از جمله: جایگزینی دستورات شرطی با استفاده از چندریختی، کاهش تکرار کد، کاهش جفت‌شدگی و افزایش خوانایی و تست‌پذیری. اما در Replace Conditional with Polymorphism، رفتار وابسته به نوع شی است و در قالب زیرکلاس‌های متفاوت در یک ساختار توارثی پیاده‌سازی می‌شود و compile-time binding وجود دارد، در حالی‌که در Factor Out Strategy، رفتار به‌شکل قابل تعویض در زمان اجرا به‌صورت runtime binding از طریق delegation به کلاس‌های استراتژی جداگانه منتقل می‌شود. بنابراین،

استراتژی الگوی منعطف‌تری است که برای تغییر رفتار در زمان اجرا مناسب‌تر است، در حالی‌که الگوی اول برای سناریوهایی مناسب است که ساختار توارثی مشخص و ثابت بین نوع‌ها وجود دارد. در الگوی اول با استفاده از تشکیل ساختار توارث و استفاده از چندریختی، بررسی نوع با به‌کارگیری دستورات شرطی پیچیده و سوییچ‌های تکراری را از بین می‌برد. در الگوی دوم نیز با تشکیل ساختار توارثی استراتژی‌ها که الگوریتم‌های مختلف را پیاده‌سازی می‌کنند و با بهره‌گیری از چندریختی و delegation، الگوریتم مناسب در زمان اجرا براساس شرایط به کانتکست منتسب می‌شود و خدمت‌رسانی می‌کند و به این شکل دستورات شرطی را از بین می‌برد.

فصل ۳

موقعیت سوم

در این فصل، موقعیت سوم ارائه شده در تمرین شرح داده شده و بررسی می‌شود. سپس دو الگوی اشاره شده در این موقعیت مورد بررسی قرار می‌گیرند.

۳-۱ بررسی موقعیت

موقعیت: برخی از کلاس‌ها Data Class هستند.

Data Classها معمولا در کنار Large Class یا God Class ظاهر می‌شوند. این کلاس‌ها عمدتا شامل مجموعه‌ای از اتریبیوت‌ها و متدهای ساده‌ای مانند get/set هستند، اما فاقد رفتار معنادار مرتبط با داده‌های خود هستند. اگر کلاسی از نظر رفتاری فقیر باشد، باید بررسی شود که رفتارهای مرتبط با داده‌های آن در کجا قرار گرفته‌اند. معمولا در چنین مواردی این رفتارها به اشتباه به کلاس‌هایی نامربوط منتقل شده و God Class تشکیل شده است و در نتیجه، تقسیم وظایف به درستی انجام نشده است. بنابراین، به محض مشاهده‌ی یک Data Class باید بررسی کرد که آیا رفتار مرتبط با داده‌هایش در کلاس دیگری است؟ در این صورت باید رفتار به کلاس داده بازگردانده شود. یا اگر سیستم در حال تکامل است، برنامه‌ریزی شود که در آینده این رفتار در همان کلاس اضافه گردد. داده و رفتار مرتبط باید در کنار یکدیگر قرار گیرند. برخلاف برنامه‌نویسی رویه‌ای که در آن داده و رفتار جدا از هم نگهداری می‌شوند، در شی‌گرایی باید رفتاری که به داده مربوط است، در همان کلاسی قرار گیرد که داده در آن تعریف شده است. از دلایل بروز این مشکل آشنایی ناکافی طراح سیستم با مفاهیم شی‌گرایی و طراحی سیستم به سبک رویه‌ای است. یا تنبلی تیم نگهداری سیستم، که به جای توزیع درست مسئولیت بین کلاس‌ها، رفتار را به راحت‌ترین کلاس موجود اختصاص می‌دهند و داده‌های مورد نیاز را از سایر کلاس‌ها جمع‌آوری می‌کنند. پس در چنین شرایطی احتمال زیادی وجود دارد که God Class نیز در سیستم تشکیل شده

باشد.

۳-۲ الگوی اول - Split Up God Class

۳-۲-۱ بررسی الگو

در برخی سیستم‌ها، ممکن است کلاسی به‌وجود آید که مسئولیت‌های زیادی را بر عهده دارد و رفتارهای مختلف را از کلاس‌های داده‌ای جمع کرده و در خود متمرکز کرده است. این کلاس که به آن God Class گفته می‌شود، به دلیل بزرگی و عدم وجود انسجام[۱]، فهم و تغییر آن بسیار دشوار است. برای رفع این مشکل، ابتدا رفتارها از God Class به سایر کلاس‌های موجود یا کلاس‌های جدید منتقل می‌شوند. ممکن است برخی کلاس‌های جدید نیز مستقیما از دل God Class استخراج شوند. در نهایت، God Class به یک Facade تبدیل می‌شود که صرفا درخواست‌ها را از کلاینت‌ها دریافت کرده و به کلاس‌های دیگر واگذار می‌کند. با این حال، اگر این Facade نیز بزرگ و غیرمنسجم باشد، باید طبق اصل ISP، آن را به چند Facade یا interface منسجم تقسیم کرد. در برخی موارد نیز ممکن است وجود همین Facade نیز ضروری نباشد، و در این صورت، God Class به‌کلی حذف شده و ارتباط کلاینت‌ها با کلاس‌ها به‌صورت مستقیم برقرار می‌شود. این یک الگوی بازمهندسی است برای فاز توزیع مجدد مسئولیت‌ها[۲]. در حین اعمال این فاز به‌دفعات انتقال رفتار به نزدیک داده[۳] انجام شده و ممکن است الگوهای Extract Class، Extract Function یا Move Method اعمال شوند.

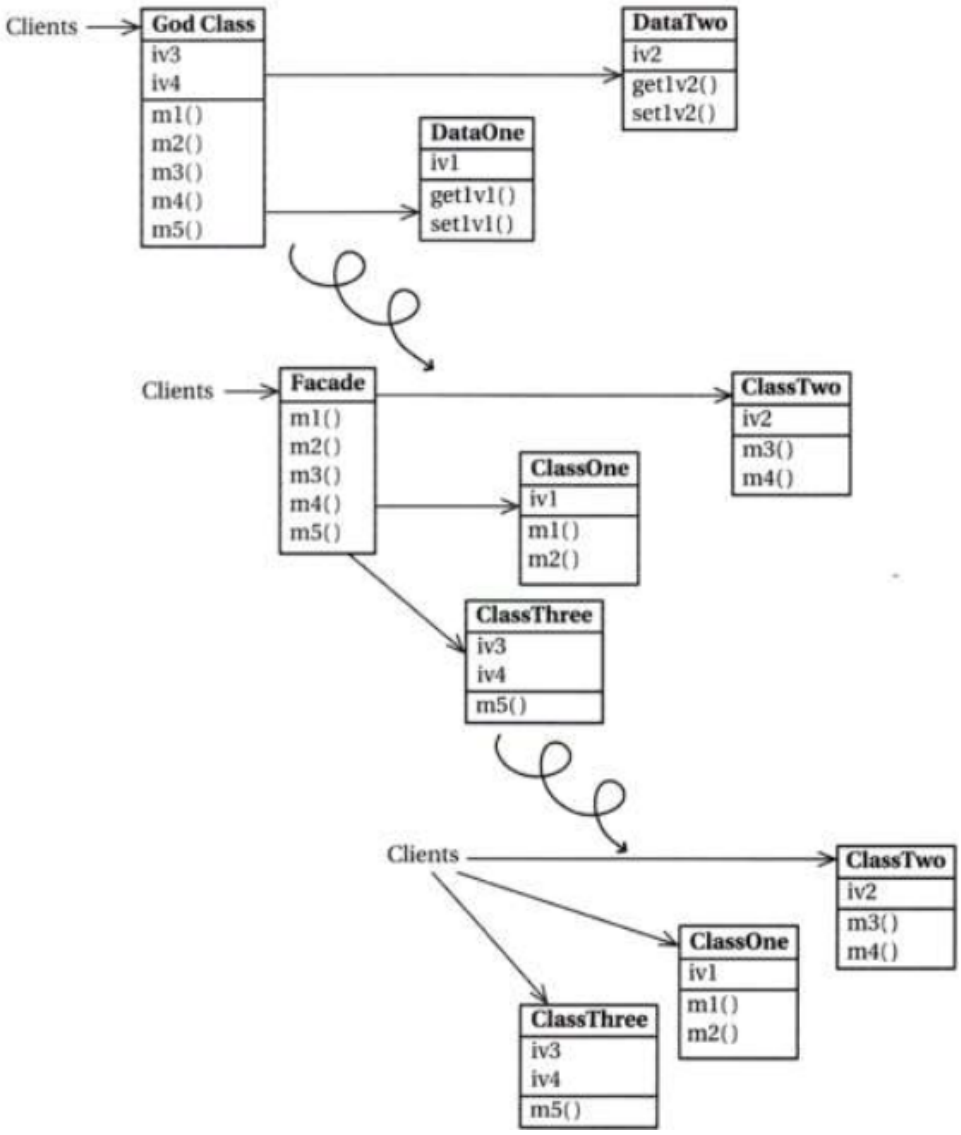
در شکل ۳-۱ مثالی از اعمال این الگو مشاهده می‌شود. در این سناریو، دو Data Container و یک God Class وجود دارند. درگام نخست، یک کلاس جدید (کلاس سوم) نیز استخراج شده است. در این فرایند، اتریبیوت‌ها و عملیات بین کلاس‌ها توزیع شده‌اند به‌طوری‌که رفتارها به کلاس‌هایی منتقل شده‌اند که داده‌های مرتبط را در خود دارند. متدهای باقی‌مانده در God Class (m1 تا m5) صرفا وظیفه‌ی delegation دارند. در نتیجه، God Class به یک Facade تبدیل شده است. در ادامه، می‌توان تصمیم گرفت که این Facade نیز حذف شود و کلاینت‌ها مستقیما با کلاس‌های حاوی داده و رفتار در ارتباط باشند، که باعث کاهش پیچیدگی و افزایش شفافیت ارتباطات می‌شود.

---

[۱] cohesion

[۲] Redistribute Responsibilities

[۳] move behavior close to data



شکل ۳-۱: مثال اعمال الگوی Split Up God Class

۳-۲-۲ زمان و شرایط مفید بودن اعمال الگو

همان‌طور که قبلا بیان شد، وجود Data Class در سیستم به احتمال زیاد نشان دهنده وجود God Class نیز است. زیرا زمانی که با یک Data Class مواجه شویم، این سوال پیش می‌آید که اکنون رفتار مربوط به این داده‌ها کجا قرار داد؟ در بسیاری از مواقع، رفتارها در یک کلاس غیرمنسجم God Class جمع شده‌اند. به این شکل کلاسی خالی از رفتار بامعنی شده (Data Class) و کلاسی نیز حاوی بیش از اندازه رفتار نامربوط شده است (God Class). بنابرین با اعمال این الگو، این کلاس بزرگ غیرمنسجم شکسته شده، رفتارها به نزدیک داده‌های خود رفته و کلاس‌های داده‌ای حاوی رفتار بامعنی مربوط به خود می‌شوند و God Class نیز یا تبدیل به Facade شده یا از بین می‌رود. پس در چنین شرایطی، اعمال الگوی مناسب ضروری است تا این کلاس

بزرگ شکسته شود و رفتارهای مربوط به هر مجموعه داده، به کلاس‌های داده‌ای مرتبط با آن داده‌ها بازگردانده شود. این مسئله به‌ویژه در صورت بروز Divergent Change اهمیت دارد، چرا که تغییر دادن چنین کلاس‌هایی دشوار و پرهزینه می‌شود.

۳-۲-۳ چگونگی مفید بودن الگو در موقعیت

این الگو با شکستن God Class و انتقال رفتارها به نزدیک داده‌های خود، باعث می‌شود بازتوزیع مسئولیت‌ها صورت گرفته و Data Classها حاوی رفتارهای بامعنی مربوط به خود شوند. همچنین God Class نیز یا از بین می‌رود یا صرفا تبدیل به یک Facade می‌شود که بدین صورت، انسجام افزایش یافته، خوانایی و قابلیت درک کد افزایش یافته، مسئولیت‌ها به‌درستی توزیع شده، تغییرپذیری افزایش یافته و Divergent Change موجود در کلاس بزرگ نیز از بین می‌رود.

۳-۲-۴ چگونگی پیاده‌سازی یا اعمال الگو

پس از تشخیص God Class راه‌حل بر این اساس استوار است که به‌صورت تدریجی، رفتارها از God Class جدا شوند. در طول این فرایند، کلاس‌های داده‌ای به‌تدریج به کلاس‌هایی شیگرا و مستقل‌تر تبدیل می‌شوند، چراکه عملیاتی را که پیش‌تر God Class روی داده‌های آن‌ها انجام می‌داد، اکنون خودشان انجام می‌دهند. همچنین، در این مسیر کلاس‌های جدیدی نیز از درون God Class استخراج می‌شوند. مراحل زیر توصیف می‌کنند که این فرایند در حالت ایده‌آل چگونه پیش می‌رود. با این حال، باید توجه داشت که God Classها از نظر ساختار داخلی ممکن است بسیار متفاوت باشند، بنابراین برای اجرای این مراحل ممکن است تکنیک‌های متفاوتی نیاز باشد. همچنین، باید روشن باشد که یک God Class را نمی‌توان به‌صورت یک‌باره اصلاح کرد. بنابراین، مسیر امن این است که ابتدا God Class را به یک نسخه سبک‌تر Lightweight God Class تبدیل کنیم، سپس به یک Facade که رفتارها را به کلاس‌ها واگذار می‌کند. در نهایت، کلاینت‌ها به کلاس‌های داده‌ای بازنویسی‌شده و سایر اشیای جدید هدایت می‌شوند و Facade نیز می‌تواند حذف شود. فرایند در شکل ۳-۱ قابل مشاهده است.

مراحل زیر به صورت تکراری اعمال می‌شوند. حتما باید پس از هر تغییر، آزمون رگرسیون انجام شود:

۱. زیرمجموعه‌های منسجم از متغیرهای نمونه در God Class را شناسایی کرده و آن‌ها را به کلاس‌های داده‌ای خارجی تبدیل کنید. سپس، متدهای مقداردهی اولیه را در God Class تغییر دهید تا به نمونه‌هایی از این کلاس‌های داده‌ای جدید ارجاع دهند.

۲. تمام کلاس‌هایی که توسط God Class به‌عنوان کلاس‌های داده‌ای استفاده می‌شوند (از جمله آن‌هایی که در مرحله‌ی ۱ ایجاد شده‌اند) را شناسایی کرده و الگوی Move Behavior Close to Data را روی آن‌ها اعمال کنید تا این کلاس‌های داده‌ای به سرویس‌دهنده‌ها ارتقا یابند. در این حالت، متدهای اصلی موجود در God Class صرفا به متدهای انتقال‌یافته delegate می‌کنند.

۳. پس از تکرار مکرر مراحل ۱ و ۲، God Class تنها به یک Facade با یک متد بزرگ مقداردهی اولیه تبدیل خواهد شد. حالا مسئولیت مقداردهی اولیه را به کلاسی جداگانه منتقل کنید تا یک Facade خالص باقی بماند. سپس به‌صورت تکراری کلاینت‌ها را به کلاس‌هایی هدایت کنید که God Class اکنون به آن‌ها Facade می‌زند و در نهایت یا Facade را منسوخ کنید (با استفاده از الگوی Deprecate Obsolete Interfaces) یا به‌سادگی آن را حذف کنید.

۳-۲-۵ مزایای اعمال الگو

- افزایش انسجام در کلاس‌ها.
- رفع مشکل Divergent Change در God Class.
- کاهش جفت‌شدگی پس از انتقال عملیات به نزدیک داده خود.
- افزایش خوانایی و قابلیت درک برنامه.
- افزایش قابلیت تغییر در برنامه.
- بهبود طراحی شی‌گرای سیستم.
- رفع Data Classها با اضافه کردن عملیات بامعنی مربوط به آن‌ها.
- مسئولیت‌ها از یک نقطه متمرکز و شکننده به قسمت‌های مختلف توزیع شده‌اند.
- پس از انتقال رفتار به نزدیک داده و ایجاد کلاس‌های بامعنی و منسجم، قابلیت استفاده مجدد افزایش می‌یابد.
- به‌دلیل حذف کلاس بزرگ و توزیع مسئولیت‌ها به کلاس‌های کوچک‌تر، مراقبت و نگهداری آسان‌تر شده است.

۳-۲-۶ معایب اعمال الگو

- اعمال این الگو می‌تواند فرایندی طولانی و کند باشد.

- دیگر نمی‌توان به یک کلاس واحد برای تشخیص رفتار برای حل یک مشکل مراجعه کرد.
- تعداد کلاس‌ها افزایش می‌یابند که سربار مراقبت و نگهداری را افزایش می‌دهد.
- برای رعایت ISP در Facade پس از اعمال الگو باید دقت کرد.
- تعداد اشیا در حافظه ممکن است افزایش یابد که سربار حافظه دارد.
- تعداد تبادل پیام‌ها ممکن است افزایش یابد.
- پس از جداسازی رفتار و توزیع در کلاس‌ها، ممکن است همکاری و جفت‌شدگی افزایش یابد.

۳-۲-۷ Bad Smell هایی که کاهش می‌یابند

- Divergent Change: قبلا توضیح داده شد که به‌دلیل وجود کلاس غیرمنسجم بزرگ این مشکل وجود داشت که با اعمال این الگو حل می‌شود.
- Feature Envy: قبلا که عملیات در کلاس بزرگ غیرمنسجم اجرا می‌شد، همواره به کلاس‌های داده‌ای برای دریافت داده نیاز داشت.
- Long Function: در حین شکستن مسئولیت‌ها و انتقال آن‌ها به نزدیک داده خود، ممکن است عملیاتی شکسته شده و فقط قسمت‌های مربوط منتقل شوندن که در این صورت تابع کوچک‌تر می‌شود.
- Lazy Element: کلاس‌های داده‌ای حاوی رفتار بامعنی می‌شوند.
- Data Clumps: قبلا که دسته‌ای از داده‌ها با هم در کلاس بزرگ غیرمنسجم در عملیات‌ها استفاده می‌شد، اکنون عملیات‌ها نزدیک داده‌های خود رفته‌اند.
- Large Class
- Data Class

۳-۲-۸ Bad Smell هایی که ممکن است بروز یابند

- Middle Man
- Lazy Element
- Shotgun Surgery

۳-۲-۹ الگوهای مرتبط

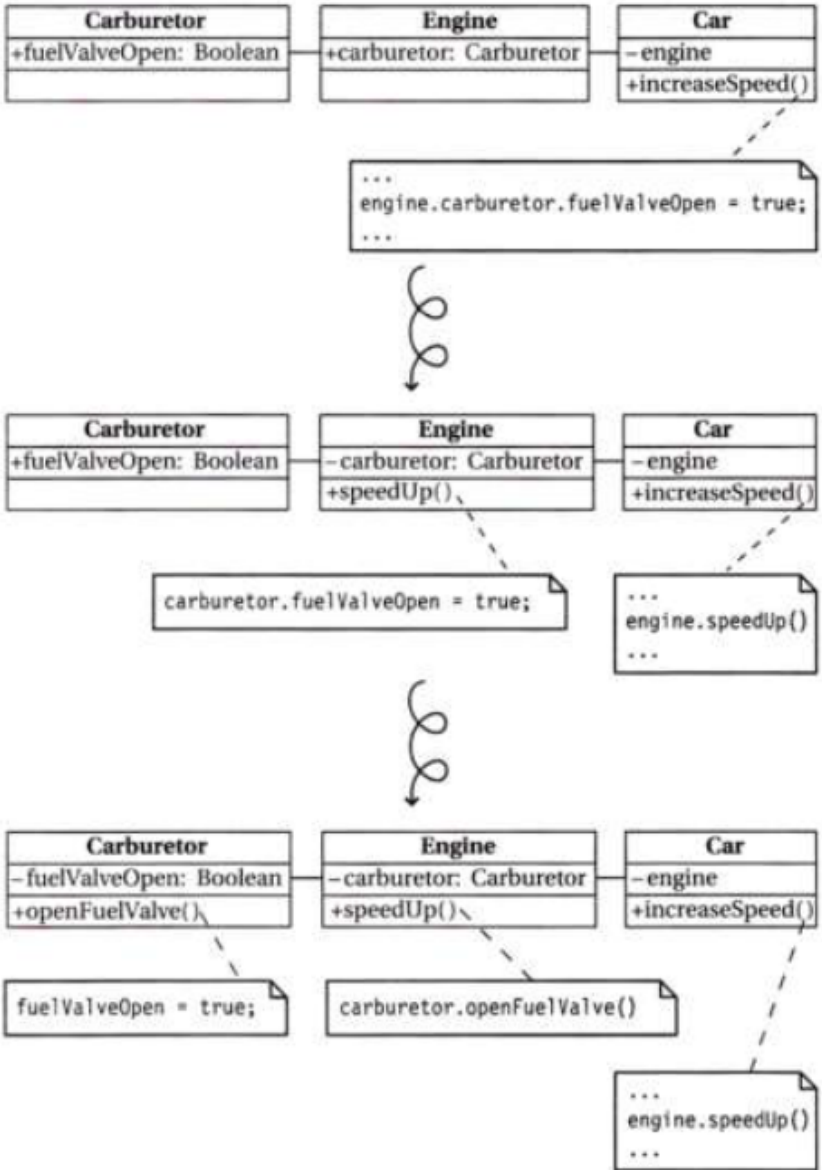
- Extract Class
- Move Behavior Close to Data
- Move Method
- Extract Function
- Remove Middle Man

۳-۳ الگوی دوم - Eliminate Navigation Code

۳-۳-۱ بررسی الگو

این الگوی بازمهندسی از فاز بازتوزیع مسئولیت‌ها، نوعی مشابه الگوی Move Behavior Close to Data است که در یک ساختار زنجیره‌ای از کلاس‌ها اعمال می‌شود. در این حالت، داده مورد نیاز کلاینت در انتهای زنجیره‌ای از آبجکت‌ها قرار دارد و کلاینت برای دستیابی به آن، باید این زنجیره را طی کند. بنابراین، به‌صورت تدریجی و مرحله‌به‌مرحله، رفتار از کلاینت به‌سمت داده حرکت داده می‌شود. به‌گونه‌ای که هر بار این رفتار به کلاس بعدی منتقل می‌شود، میزان encapsulation تقویت می‌گردد و transitive visibility از بین می‌رود. در نهایت، کلاینت تنها با آبجکت ابتدایی زنجیره ارتباط برقرار می‌کند و هیچ‌گونه دید مستقیمی به سایر اجزای زنجیره ندارد. در نتیجه، تنها آبجکت اول به‌عنوان نقطه تعامل کلاینت عمل می‌کند و کل پیمایش و اعمال رفتار، درون زنجیره انجام می‌شود. گرچه این الگو شباهت‌هایی به الگوی Move Behavior Close to Data دارد، اما با آن یکسان نیست. چراکه مسئله مطرح‌شده در اینجا متفاوت بوده و به‌جای یک جابه‌جایی ساده بین دو کلاس، با ساختاری زنجیره‌ای و نیاز به اعمال مکرر الگو مواجه‌ایم. با اعمال مکرر الگوی انتقال رفتار به نزدیک داده، دید ترایا را از بین برده و اصل PLK را برقرار می‌کنیم.

در شکل ۳-۲ نمونه‌ای از اعمال این الگو مشاهده می‌شود. در این مثال، سه کلاس داریم: Car، Engine و Carburetor. کلاس ماشین شامل یک نمونه از موتور است و موتور نیز به نوبه خود به Carburetor دسترسی دارد. در ابتدا، متدی در کلاس ماشین به نام increaseSpeed وجود دارد که مستقیما به موتور دسترسی پیدا می‌کند، سپس از طریق موتور به کاربراتور دسترسی می‌یابد و در نهایت مستقیما مقدار اتریبیوت fuelValveOpen درون کاربراتور را به true تغییر می‌دهد تا دریچه سوخت باز شده و سرعت افزایش یابد. این طراحی ناقض encapsulation است، چرا که ماشین مستقیما به اجزای داخلی موتور و حتی عمیق‌تر،



شکل ۳-۲: مثال الگوی Eliminate Navigation Code

به ویژگی‌های داخلی کاربراتور دسترسی دارد. این نوع وابستگی زنجیره‌ای و دسترسی عمیق به اشیا داخلی، نمونه‌ای از transitive visibility است که طراحی شیگرا را تضعیف می‌کند. در گام اول بازآرایی، مسئولیت افزایش سرعت به موتور منتقل می‌شود. حالا ماشین فقط متد speedUp را در موتور فراخوانی می‌کند و دیگر با کاربراتور یا ویژگی‌های آن کاری ندارد. اما هنوز هم موتور مستقیما اتریبیوت fuelValveOpen را تغییر می‌دهد، که همچنان ناقض encapsulation است. در گام دوم، این مشکل نیز با انتقال رفتار به کلاس کاربراتور حل می‌شود. اکنون موتور متدی مثل openFuelValve را فراخوانی می‌کند که درون خود کاربراتور پیاده‌سازی شده است و تغییر مقدار ویژگی به‌صورت داخلی انجام می‌شود. در نتیجه، پس از بازآرایی تدریجی هر کلاس فقط با اجزای سطح خودش کار می‌کند، دسترسی مستقیم به ویژگی‌های داخلی سایر کلاس‌ها حذف می‌شود.

encapsulation رعایت شده و وابستگی‌های بین کلاس‌ها کاهش می‌یابد.

۳-۳-۲ زمان و شرایط مفید بودن اعمال الگو

در برخی از موارد، منشا به‌وجود آمدن Data Class زنجیره‌ای از کلاس‌هاست که داده در انتهای زنجیره قرار دارد، ولی رفتار مربوط به آن داده در ابتدای زنجیره تعریف شده است. این مسئله منجر به نقض Encapsulation می‌شود، چرا که برای اجرای عملیات، داده‌ها باید از مسیرهایی در زنجیره پیمایش شوند و کلاس‌ها به ویژگی‌هایی دید دارند که نباید داشته باشند. علت اصلی این وضعیت، تنبلی در طراحی است. به‌جای انتقال رفتار به نزدیکی داده، آن را در نزدیک‌ترین محل قابل دسترس قرار می‌دهیم و سپس داده‌های لازم را از کلاس‌های مختلف جمع‌آوری می‌کنیم. در نتیجه، سیستم دارای کلاس‌هایی می‌شود که صرفا داده را نگهداری می‌کنند و رفتاری ندارند، در حالی که عملیات مربوط به آن داده‌ها در جای دیگری از سیستم پیاده‌سازی شده است و همچنین دید ترایا وجود دارد. برای حل این مشکل، می‌توان از این الگو استفاده کرد.

۳-۳-۳ چگونگی مفید بودن الگو در موقعیت

با اعمال این الگو، به‌مرور رفتار در زنجیره حرکت کرده و به سمت کلاسی می‌رود که داده مربوطه در آن قرار دارد و به این شکل کلاس‌های داده‌ای که تهی از رفتار معنی‌دار بودند اکنون حاوی عملیات مربوط به داده خود می‌شوند. همچنین این فرایند باعث بهبود encapsulation و کاهش جفت‌شدگی می‌شود و مشتری صرفا عملیاتی را از ابتدای زنجیره فراخوانی می‌کند و تحقق عملیات در طول زنجیره رخ می‌دهد که عملیات‌ها بازتوزیع شده و نزد داده خود قرار گرفته‌اند و مشتری از داخل زنجیره خبر ندارد.

۳-۳-۴ چگونگی پیاده‌سازی یا اعمال الگو

رفتارهایی که توسط یک کلاینت غیرمستقیم تعریف شده‌اند را به‌صورت تکراری به کلاسی منتقل کنید که داده‌ی مورد استفاده‌ی آن رفتار را در خود نگه می‌دارد. توجه داشته باشید که مراحل بازمهندسی واقعی در این الگو، اساسا مشابه الگوی Move Behavior Close to Data هستند، اما نمود مسئله در اینجا متفاوت است. در هر مرحله از زنجیره، یک گام به منبع داده نزدیک‌تر شده و رفتار مربوطه به کلاس مناسب منتقل می‌گردد. این فرایند تکراری در نهایت الگو را محقق می‌سازد.

برای اعمال این الگو، ابتدا باید کلاس‌های داده‌ای را به‌عنوان تأمین‌کنندگان داده شناسایی کرده و کلاینت‌هایی که به‌صورت غیرمستقیم از طریق زنجیره‌ای از اتریبیوت‌ها یا متدهای دسترسی به آن‌ها رجوع می‌کنند را تشخیص

دهیم. در هر دو حالت، داده نهایی به‌عنوان یک کلاس داده‌ای رفتار می‌کند که عملیات مورد نظر روی آن اعمال می‌شود. وضعیتی که نشان‌دهنده‌ی نقض اصول Encapsulation و اصل PLK است. فرایند Eliminate Navigation Code در واقع به‌صورت بازگشتی از الگوی انتقال رفتار به نزدیکی داده استفاده می‌کند. شکل ۳-۲ این تبدیل را نشان می‌دهد.

مراحل:

۱. ابتدا زنجیره و عملیاتی را که باید منتقل شود، شناسایی کنید.

۲. با اعمال الگوی Move Behavior Close to Data یک سطح از Navigation را حذف کنید. (در این مرحله باید تست‌های رگرسیون اجرا شوند.)

۳. تکرار مراحل قبل تا رفع مشکل.

۳-۳-۵ مزایای اعمال الگو

- افزایش تحقق Encapsulation.
- رفع مشکل دید ترایا و رعایت اصل PLK.
- کاهش جفت‌شدگی به‌وجود آمده به‌دلیل نقض Encapsulation و دید ترایا و کاهش انتشار تغییرات.
- انتقال عملیات مرتبط به کلاس‌های داده‌ای و رفع Data Class.
- افزایش انسجام کلاس‌ها با انتقال رفتار به مکان مناسب.
- افزایش خوانایی، قابلیت درک و نگهداری.
- به‌دلیل مستقل شدن و جمع شدن داده و رفتار مرتبط در کلاس‌ها پس از اعمال الگو، امکان استفاده مجدد افزایش یافته است.

۳-۳-۶ معایب اعمال الگو

- اعمال این الگو می‌تواند کند، طولانی و پیچیده باشد.
- اگر سیستم در حال تکامل باشد، ممکن است مدام رفتار جدید معرفی شود که انتقال و تعریف این رفتار در جای درست ممکن است دشوار باشد.

- کاربرد سیستماتیک این الگو ممکن است منجر به ایجاد واسط‌های بزرگ شود. به‌ویژه، اگر یک کلاس دارای متغیرهای نمونه‌ای زیادی از نوع Collection باشد، استفاده از این الگو شما را مجبور می‌کند تا تعداد زیادی متد اضافی تعریف کنید تا دسترسی مستقیم به این مجموعه‌ها را پنهان کرده و آن‌ها را کپسوله کنید.
- استفاده از delegation و indirection باعث می‌شود اندکی تعداد تبادل پیام افزایش و کارایی اندکی کاهش یابد.
- در بعضی موارد ممکن است عناصری در میانه زنجیره صرفا نقش واسطه را بازی کنند و صرفا delegate کنند.

۳-۳-۷ Bad Smell هایی که کاهش می‌یابند

- Divergent Change: منسجم‌تر شدن کلاس‌ها و انتقال رفتار به نزدیکی داده مربوطه.
- Shotgun Surgery: به‌دلیل کاهش جفت‌شدگی نسبت به حالت قبل و کاهش انتشار تغییرات.
- Data Class: رفتار مربوطه به نزدیکی داده منتقل می‌شود و کلاس‌های داده‌ای حاوی رفتار بامعنی می‌شوند.
- Message Chains
- Lazy Element: کلاس‌هایی که قبلا ممکن بود رفتار خاصی نداشته باشند، اکنون حاوی رفتار بامعنی شده‌اند.

۳-۳-۸ Bad Smell هایی که ممکن است بروز یابند

- Middle Man: ممکن است برخی عناصر در زنجیره، پس از اعمال این الگو، تقریبا تمام مسئولیت‌های خود را از دست داده و صرفا نقش واسپاری کننده را بازی کرده و تبدیل به واسطه شوند.
- Lazy Element: مشابه توضیح مورد قبل.
- Shotgun Surgery: به‌دلیل پخش شدن رفتار در زنجیره به‌منظور نزدیک شدن به داده خود، تغییر رفتار کلی می‌تواند منجر به تغییر در بخش‌های مختلف شود.

۳-۳-۹ الگوهای مرتبط

- Move Behavior Close to Data
- Move Method
- Extract Function
- Encapsulate Field

۳-۴ مقایسه الگوها

هردو، الگوی فاز Redistribute Responsibilities بازمهندسی هستند و زمانی به کار می‌روند که داده‌ها از رفتارهای مرتبط جدا شده‌اند و سعی می‌کنند الگوی Information Expert را محقق کنند. اگر این رفتارها در یک کلاس بزرگ غیرمنسجم متمرکز باشند، باید از الگوی اول استفاده کرد. اما اگر این رفتارها در کلاس‌های مختلف توزیع شده و موجب دید ترایا شده‌اند، الگوی دوم مناسب‌تر است. هردو از الگوی انتقال رفتار به نزدیکی داده بهره می‌گیرند. اعمال این الگوها باید به‌صورت تدریجی و مرحله‌به‌مرحله انجام شود و پس از هر مرحله، آزمون رگرسیون لازم است تا از صحت عملکرد برنامه اطمینان حاصل گردد. در نهایت، استفاده از این الگوها Data Classها را رفع کرده و طراحی برنامه را به طراحی شی‌گرا نزدیک‌تر می‌کند و سایر مزایا که در قسمت‌های قبل توضیح داده شد.

Bibliography

[1] A. Shvets. https://refactoring.guru.

[2] R. Ramsin. Patterns in software engineering, 2025. Lecture slides, Sharif University of Technology, Tehran, Iran.

[3] O. Nierstrasz. https://www.oscar.nierstrasz.org/oorp/.

Abstract

In this study, three situations were described, each representing certain problems within the system. For each situation, two patterns were proposed. The proposed patterns for each case were analyzed, explaining how and under what conditions they address the identified issues. Finally, the proposed patterns were compared.

Keywords: refactoring, reengineering

Sharif University of Technology
Department of Computer Engineering

Patterns in Software Engineering

Third Assignment

By:

Mehdi Eidi

Professor:

Dr. Raman Ramsin

Summer 2025