دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

کارشناسی ارشد
مهندسی نرم‌افزار

عنوان

الگوها در مهندسی نرم‌افزار

تمرین اول

نگارش

مهدی عیدی

استاد

دکتر رامان رامسین

بهار ۱۴۰۴

چکیده

در این تمرین، ۱۲ الگوی مطرح شده با رویکرد مبتنی بر معیار، تحلیل و ارزیابی شدند. برای این ارزیابی، معیارهایی شامل اصول شیگرایی و الگوهای GRASP استفاده شدند. این تحلیل، به شفافسازی ویژگیها و نقاط قوت و ضعف هر الگو کمک میکند.

کلیدواژهها: الگوها در مهندسی نرمافزار، الگوهای طراحی، ارزیابی مبتنی بر معیار

فهرست مطالب

فهرست تصاویر

فصل ۱

معیارهای ارزیابی

در این تمرین، هدف تحلیل، ارزیابی و مقایسه مبتنی بر معیار[۱] الگوهای طراحی است. در ادامه این فصل، معیارهای ارزیابی که مبنای کار خواهند بود معرفی می‌گردند.

- دسته
- هدف
- حوزه
- ساختار و رفتار
- جفت‌شدگی[۲]
- انسجام[۳]
- پیکربندی
- انعطاف‌پذیری
- کارایی
- شی جعلی
- انتشار تغییرات
- سهولت پیاده‌سازی
- موارد کاربرد

[۱] criteria-based
[۲] coupling
[۳] cohesion

- کپسوله‌سازی[۴]
- جداسازی دغدغه‌ها[۵]
- مزایا و معایب (تبعات)
- الگوهای مرتبط
- اصول شی‌گرایی:
  - OCP
  - LSP
  - ISP
  - DIP
  - CRP
  - PLK
- الگوهای GRASP[6]:
  - Information Expert
  - Creator
  - Low Coupling
  - High Cohesion
  - Controller
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations

---

[۴] encapsulation
[۵] separation of concerns
[6] general responsibility assignment software patterns

فصل ۲

ارزیابی الگوهای Builder، Command و Event Logging

در این فصل، سه الگوی Builder، Command و Event Logging مبتنی بر معیارهایی که در فصل یک معرفی شدند، تحلیل، ارزیابی و مقایسه می‌شوند.

۲-۱ دسته

Builder: این الگو در دسته الگوهای طراحی آفرینشی[۱] قرار دارد. این دسته از الگوها کار نمونه‌گیری[۲] و پیکربندی کلاس‌ها و اشیا را بر عهده دارند. به این سوال می‌پردازند که چگونه نمونه‌گیری انجام دهیم که انعطاف‌پذیری بالا بوده و جفت‌شدگی[۳] در حد معمول پایین باشد. به دو دغدغه اصلی توجه می‌کنند که چگونه اشیا ساخته شوند و همچنین دانش در مورد اینکه سیستم از چه کلاس‌های concrete استفاده می‌کند را پنهان سازند.

Command: این الگو در دسته الگوهای رفتاری[۴] قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام[۵] و انعطاف‌پذیری.

---

[۱] creational
[۲] instantiation
[۳] coupling
[۴] behavioral
[۵] cohesion

Event Logging: این الگو جزو الگوهای هفت‌گانه پیتر کُد[۶] است. نسبتا ریزدانه بوده و بیشتر وجه رفتاری داشته و مسائلی مانند آغاز زنجیره‌ای از تعاملات برای پاسخ‌دهی به رویداد بیرونی در آن دیده می‌شود.

مقایسه: الگوی اول در دسته آفرینشی، الگوی دوم در دسته رفتاری و الگوی سوم از مجموعه الگوهای هفت‌گانه پیتر کد است که وجه رفتاری دارد.

۲-۲ هدف

Builder: در این الگو، هدف ساخت اشیای پیچیده است، با این نکته که فرایند ساخت (الگوریتم کلی) بین تمام اشیا مشترک است، اما نتیجه‌ی نهایی می‌تواند ساختارهای داخلی[۷] متفاوتی داشته باشد. الگوریتم کلی که ترتیب کلی کار را مشخص میکند مشابه است. به بیان دیگر، گویا یک الگوریتم ساخت ثابت داریم که می‌تواند محصولات مختلف تولید کند. فرایند ساخت از جزئیات جدا شده است. انعطاف‌پذیری از این جهت که با الگوریتم مذکور بشود اشیا با ساختارهای متفاوت ساخت، وجود دارد.

Command: این الگو، با هدف تبدیل یک درخواست (یا پیغام) به یک شی مستقل معرفی شده است. به‌طور معمول، وقتی یک شی پیغامی به شی دیگر می‌فرستد، این پیغام یک رویداد آنی و بی‌هویت است که بلافاصله منتقل و اجرا می‌شود. اما وقتی این پیغام را به صورت یک شی درآوریم، هویت‌دار و مانا[۸] می‌شود. این تغییر باعث می‌شود که بتوان پیغام‌ها را مانند هر شی دیگری مدیریت کرد. برای مثال می‌توان آن‌ها را در صف قرار داد، ذخیره یا بازیابی کرد، لاگ کرد یا پس از وقوع خطا یا کرش در سیستم، مجدداً به همان ترتیب قبلی اجرا نمود تا وضعیت سیستم به حالت قبل بازگردد. با این شیوه، پیغام‌ها می‌توانند حالت سیستم را در خود نگه دارند و در صورت نیاز، عملیات انجام‌شده را با undo برگردانند. در این مدل، خود شی command مسئول بازگرداندن وضعیت به حالت قبل از اجرای دستور است. این ساختار به ما اجازه می‌دهد که یک پردازش کننده command طراحی کنیم که مجموعه‌ای از فرمان‌ها را دریافت کرده و به‌ترتیب دلخواه اجرا کند. از طرفی، چون command به‌عنوان یک شی مستقل تعریف می‌شود، می‌توان مشتری‌ها را نیز پارامتریزه کرد. به این معنا که در زمان اجرا، بر اساس شرایط مختلف، تصمیم بگیریم کدام فرمان ارسال شود.

Event Logging: معمولا در سیستم‌های real-time که با پردازش رویدادهای بیرونی سروکار دارند مطرح می‌شود، اما می‌توان آن را به سایر انواع سیستم‌ها نیز تعمیم داد و نباید صرفا محدود به سیستم‌های real-time دانست. در این‌جا، کلاس‌هایی به نام Device تعریف می‌شوند که وظیفه‌شان پایش یک دستگاه بیرونی، عمدتا یک حسگر است. این‌ها باید تشخیص دهند که یک رویداد خارجی رخ داده (یعنی حسگر

[۶] Peter Coad
[۷] representation
[۸] persistent

تحریک شده) و سپس زنجیره‌ای از تعاملات را برای پاسخ‌دهی آغاز کنند. برای نمونه، در یک سیستم اعلام خطر، حسگرهای حریق باید بررسی شوند. در این سیستم، برای هر حسگر یا گروهی از حسگرها، یک شی از کلاس Device در زمان اجرا ایجاد می‌شود که وظیفه پایش آن‌ها را بر عهده دارد. می‌تواند به صورت وقفه[۹] یا از طریق سر زدن دوره‌ای[۱۰] انجام شود تا در صورت تغییر وضعیت حسگر، تحریک آن تشخیص داده شود. پس از تشخیص رویداد، این شی ممکن است مستقیما واکنش نشان ندهد، بلکه پیامی به سایر اشیا مانند شی مربوط به آژیر ارسال کند تا آن‌ها اقدامات لازم مانند فعال‌سازی یک دستگاه آژیر بیرونی را انجام دهند. در بسیاری از سیستم‌ها، یکی از جنبه‌های مهم پاسخ‌دهی به رویدادها، ثبت log آن‌هاست. به‌طوری که اطلاعات مربوط به رویداد ذخیره شود تا بتوان بعدا عملیاتی بر اساس آن‌ها انجام داد. در این الگو، برای هر رویداد، یک شی از کلاسی به نام EventRemembered ساخته می‌شود که نماینده‌ی آن رویداد است. این شی شامل اتریبیوت‌هایی است که اطلاعات مربوط به رویداد در آن‌ها ذخیره می‌شود. این اشیا نه تنها داده‌های مربوط به رویداد را در خود نگه می‌دارند، بلکه ممکن است خودشان بتوانند رفتارهایی از خود نشان دهند. به‌عنوان مثال، می‌توانند گزارش تولید کنند، یا حتی خودشان عملیات مربوط به آن رویداد را آغاز نمایند. یعنی بسته به محتوای اتریبیوت‌هایشان، می‌توانند تشخیص دهند که چه پاسخی باید داده شود و اجرای بخشی از این پاسخ را نیز بر عهده بگیرند. برخلاف حالتی که فقط یک شی دیگر مانند Device، مسئول واکنش باشد، این اشیای رویدادی می‌توانند خود نقش فعالی در پردازش داشته باشند. همچنین، از آن‌جا که این اشیا دارای هویت و ماندگاری هستند، می‌توان آن‌ها را برای پردازش آتی ذخیره کرد. بدین صورت که در یک بخش از سیستم ساخته و نگهداری شوند و سپس در زمان دیگری، حتی به صورت آسنکرون، توسط بخش دیگری از سیستم پردازش گردند.

مقایسه: اهداف هر الگو به‌صورت مفصل شرح داده شده است و ارتباط الگوها نیز در قسمت الگوهای مرتبط توضیح داده شده است.

۳-۲ حوزه

Builder: این الگو در حوزه شی[۱۱] قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

Command: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation

[۹] interrupt
[۱۰] polling
[۱۱] object scope

استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

Event Logging: این الگو روابط پیچیده‌ای تعریف نمی‌کند و نسبتا ریزدانه است. کلاس‌های Device و EventRemembered تعریف می‌شوند و به منظور ساخت اشیای EventRemembered، دید از سمت Device به آن برقرار است که این قسمت در زمان کامپایل محقق می‌شود. اما نمونه‌گیری از خود Device و تخصیص آن برای پایش یک دستگاه بیرونی، در زمان اجرا باید انجام شود.

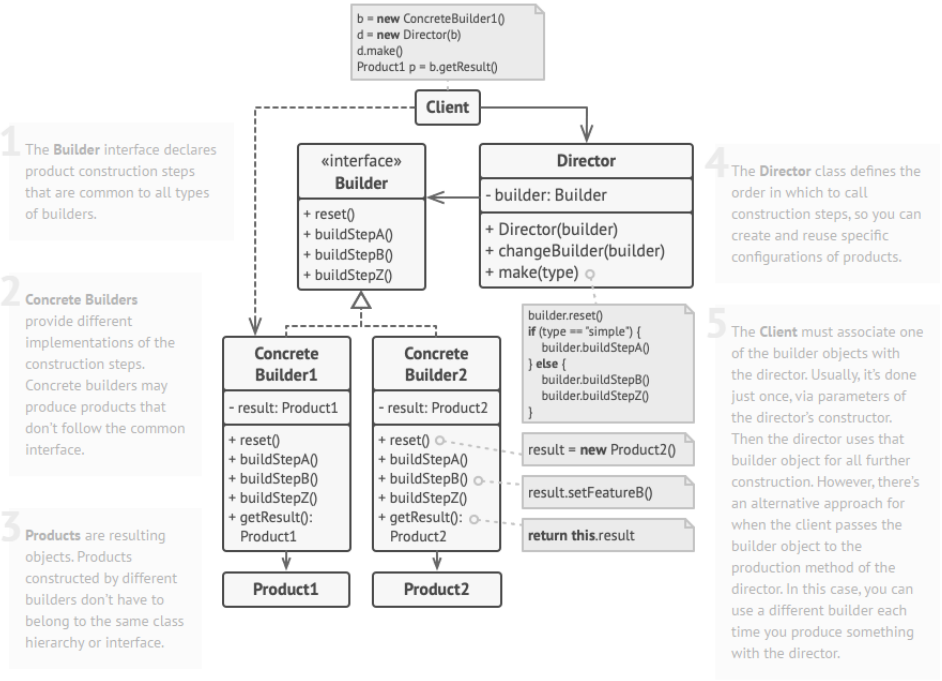
مقایسه: دو الگوی اول در حوزه شی قرار دارند و برای الگوی سوم توضیح مفصل‌تر ارائه شده است.

۲-۴ ساختار و رفتار

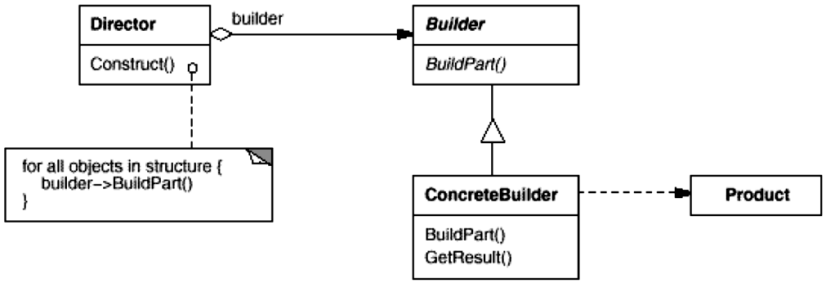
Builder: ساختار این الگو در شکل ۲-۱ و ۲-۲ قابل مشاهده است. در این ساختار، مشتری نقش پیکربند را ایفا می‌کند. مشتری دید از نوع وابستگی[۱۲] با زیرکلاس[۱۳] های Builder دارد که دید مانا نبوده و برای پیکربندی کارگردان استفاده می‌شود. مشتری از یک زیرکلاس Builder نمونه‌گیری کرده و به عنوان پارامتر در هنگام نمونه‌گیری از کارگردان استفاده می‌کند و به آن ارسال می‌کند. واسط[۱۴] Builder عملیات ساخت که بین تمام انواع مشترک هستند را تعریف می‌کند که زیرکلاس‌های آن پیاده‌سازی‌های متفاوتی از آن‌ها محیا می‌کنند که منجر به ساخت محصولات متفاوتی می‌شود. محصولات، اشیایی هستند که در نتیجه ساخته می‌شوند. کلاس کارگردان ترتیب و فرایند کلی فراخوانی عملیات‌های ساخت را تعریف می‌کند که این امکان را می‌دهد که پیکربندی‌های مشخصی از محصولات ساخته و استفاده مجدد[۱۵] شوند. رابطه association بین کارگردان و Builder وجود دارد و برخلاف شکل ۲-۲ نیازی به وجود رابطه aggregation یا composition نیست و در اینجا رابطه association وجود دارد. کارگردان از سازنده‌ای که توسط مشتری به آن تخصیص یافته است استفاده می‌کند برای آفرینش‌های آتی. دید در رابطه association از سوی کارگردان به Builder است و در نتیجه Builder کارگردان را نمی‌شناسد. با استفاده از gen spec انواع زیرکلاس برای Builder تعریف شده است. در متد Construct در کارگردان، الگوریتم ساخت بیان شده است و برای ساخت محصول نهایی، ترتیب خاص اجرای عملیات را به Builder دستور می‌دهد. معمولا عملیات builder را ریزدانه می‌گیرند و ریزدانگی باعث افزایش تنوع الگوریتم سمت کارگردان می‌شود. کارگردان دید به محصول نهایی ندارد و محصول نهایی مورد استفاده مشتری است. بنابراین متد getResult در واسط Builder تعریف نشده است که جلوی یک

[۱۲] dependency
[۱۳] subclass
[۱۴] interface
[۱۵] reuse

جفت‌شدگی زائد را می‌گیرد. تغییرات در زیرکلاس‌های Builder به کارگردان متنشر نمی‌شود چون کارگردان از واسط آن استفاده می‌کند و دید به زیرکلاس‌ها ندارد. دید مشتری به زیرکلاس‌های Builder نیز از نوع وابستگی و غیر مانا هست که در نهایت با فراخوانی متد getResult محصول نهایی ساخته شده را برمی‌گرداند. همچنین نمودار توالی[۱۶] این الگو نیز در شکل ۲-۳ قابل مشاهده است که به نکات تعامل اشیا نیز اشاره شد.

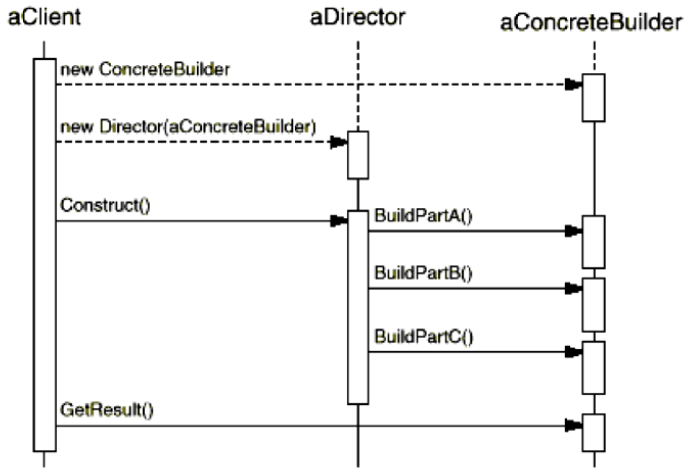


شکل ۲-۱: ساختار الگوی Builder [۱]



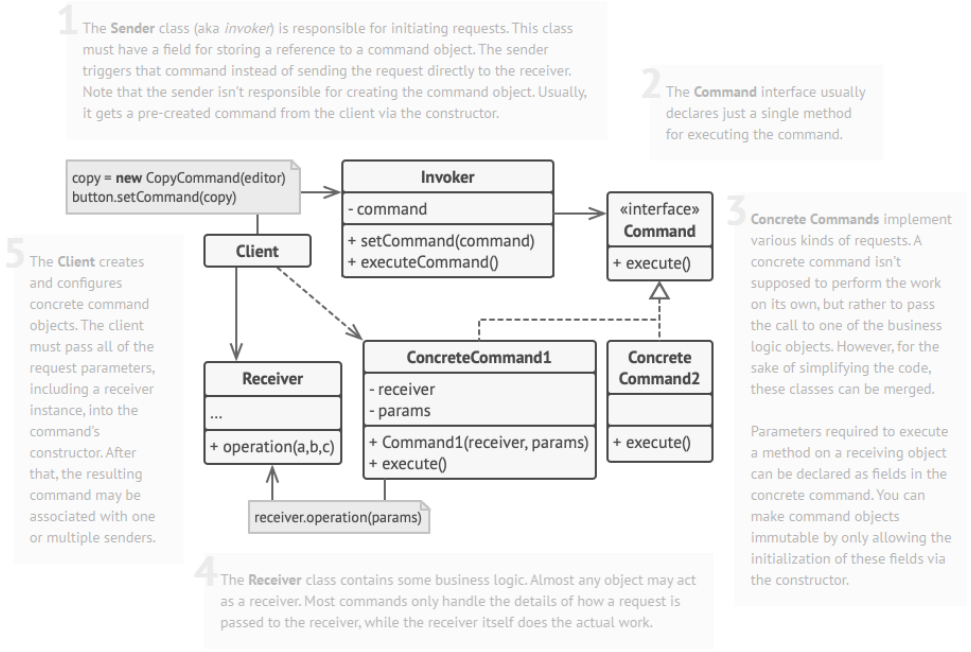
شکل ۲-۲: ساختار الگوی Builder در کتاب GoF [۲]

[۱۶] sequence diagram



شکل ۲-۳: نمودار توالی الگوی Builder [۲]

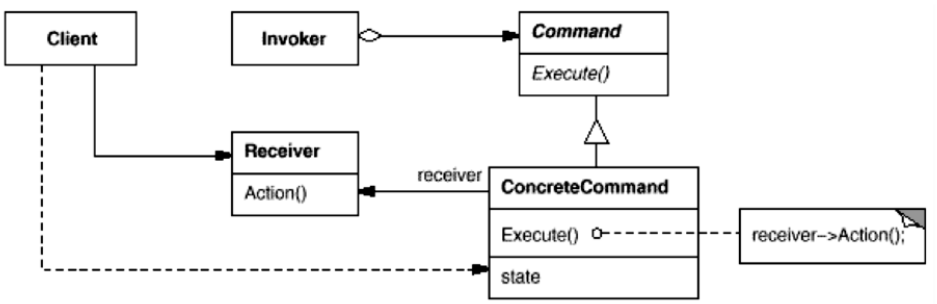
Command: ساختار این الگو در شکل ۲-۴ و ۲-۵ قابل مشاهده است.



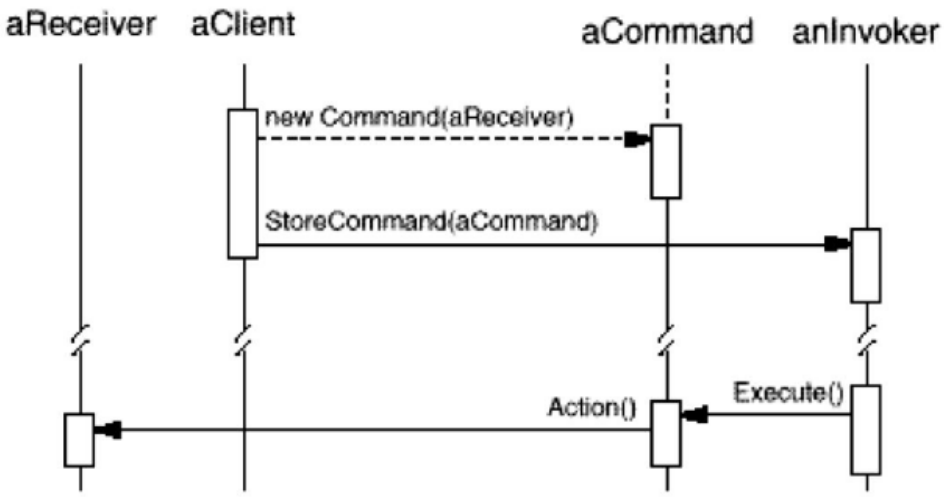
شکل ۲-۴: ساختار الگوی Command [۱]

در این الگو، ساختار کلی شامل چهار جزء اصلی است: Client، ConcreteCommand، Receiver و Invoker. در این ساختار، مشتری وظیفه پیکربندی و برقرار کردن ارتباط بین اجزای دیگر را بر عهده دارد. او ابتدا یک نمونه از Receiver می‌سازد، سپس یک نمونه از ConcreteCommand ساخته و آن را با Receiver پیکربندی می‌کند. ConcreteCommand از وجود Receiver آگاه است و می‌داند هنگام اجرای دستور باید به آن پیغام action بدهد، اما خود مشتری است که تعیین می‌کند ConcreteCommand با کدام Receiver در ارتباط باشد. بعد از پیکربندی Command، این شی در اختیار Invoker قرار می‌گیرد؛ یعنی کلاینت Invoker

را هم با این Command پیکربندی می‌کند. در زمان اجرا، وقتی Invoker قصد انجام کاری را دارد، پیام execute را به ConcreteCommand می‌فرستد و ConcreteCommand نیز متد مناسب را در Receiver صدا می‌زند تا کار انجام شود. نکته مهم اینجاست که وابستگی بین مشتری و Receiver یا بین مشتری و Invoker می‌تواند موقتی از نوع use باشد و نیازی به ارتباط دائمی یا association نیست. همچنین، رابطه بین Invoker و Command نیز صرفاً از نوع association است و نه composition یا aggregation؛ زیرا نیازی به رابطه "کل به جزء" در اینجا وجود ندارد. به همین دلیل استفاده از نماد لوزی در دیاگرام بی‌مورد است. همچنین نمودار توالی این الگو نیز در شکل ۲-۶ قابل مشاهده است که به نکات تعامل اشیا نیز اشاره شد.



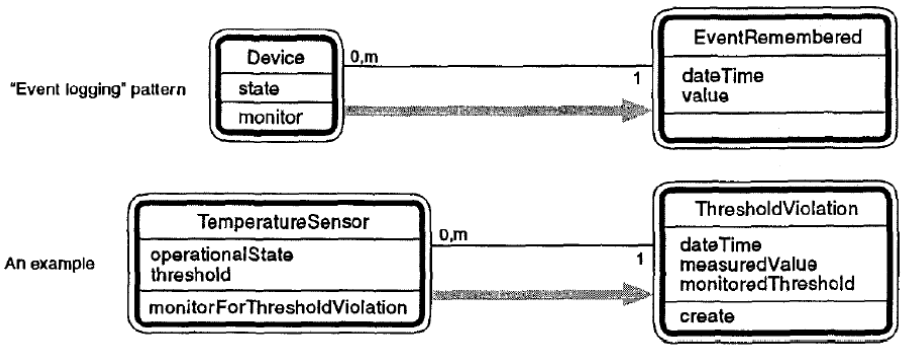
شکل ۲-۵: ساختار الگوی Command در کتاب GoF [۲]



شکل ۲-۶: نمودار توالی الگوی Command [۲]

Event Logging: ساختار این الگو در شکل ۲-۷ قابل مشاهده است. در این بخش، منظور از value، مقدار به‌دست آمده از دستگاه خارجی مرتبط با سیستم است. یعنی مقداری که مثلا یک سنسور اندازه‌گیری کرده است. البته در برخی موارد، مانند سنسورهای دوحالتی (مثلا روشن/خاموش)، value اهمیت چندانی ندارد. در مثالی که به آن اشاره شده، نمونه‌های کلاس TemperatureSensor وظیفه پایش سنسورهای

بیرونی را دارند. زمانی که سنسور مقدار بحرانی‌ای را نشان دهد (یعنی تریگر شود)، شی متناظر با آن سنسور این وضعیت را تشخیص داده و وقوع آن را ثبت می‌کند. این ثبت از طریق ایجاد یک نمونه از کلاس ThresholdViolation انجام می‌شود که مقادیر اتریبیوت‌های آن بر اساس وضعیت رویداد تنظیم می‌گردد. عملیات monitorForThresholdViolation یک تابع فعال است که به‌صورت مداوم سنسور بیرونی را بررسی می‌کند. اگر مقدار سنجیده شده از مقدار آستانه‌ای[۱۷] که در اتریبیوت‌ها تعریف شده بیشتر باشد، باید رویداد را ثبت کند. یعنی یک شی از کلاس ThresholdViolation بسازد. کلاس ThresholdViolation ممکن است عملیات متعددی داشته باشد. از جمله تولید گزارش، تحلیل پس از وقوع و موارد مشابه. در این‌جا عملیات create که منظور همان سازنده[۱۸] است، نمایانگر ایجاد یک شی جدید است. گرچه در نمودار به‌خوبی دیده نمی‌شود، اما پیغام از یک شی به کلاس ارسال می‌شود یعنی به کادر داخلی. یعنی نوک پیکان به کادر داخلی اشاره دارد. این نشان می‌دهد که یکی از نمونه‌های کلاس TemperatureSensor به خود کلاس ThresholdViolation پیام می‌فرستد تا یک نمونه جدید بسازد. چون سازنده‌ها در سطح کلاس تعریف می‌شوند و نه در سطح شی، عملیاتی class-level یا static محسوب می‌شوند.



شکل ۲-۷: ساختار الگوی Event Logging [۳]

۲-۵ جفت‌شدگی

Builder: در این الگو، دید یک‌سویه از سمت کارگردان به واسط Builder وجود دارد. در نتیجه، اولا دید معکوس و جفت‌شدگی زائد از سمت سازنده به کارگردان وجود ندارد و دوما کارگردان از تغییرات زیرکلاس‌های Builder منتزع است و تغییرات آن‌ها به کارگردان منتشر نمی‌شود و در این مجموعه، جفت‌شدگی مطلوب است. همچنین کارگردان هیچ دیدی نسبت به محصولات ندارد. اما در سمت مشتری که پیکربند نیز هست، مشتری باید تمام زیرکلاس‌های Builder را بشناسد و از آن‌ها نمونه‌گیری می‌کند و این به معنی جفت‌شدگی بالا از سمت مشتری به مجموعه است. ولی در سمت کارگردان، تا زمانی که واسط Builder تغییر نکند، کارگردان

[۱۷] threshold
[۱۸] constructor

نیز تغییر نمی‌کند. مشتری با فراخوانی getResult روی زیرکلاس Builder، محصول را دریافت می‌کند.

Command: در این الگو، رابطه بین Invoker و Command از طریق واسط است و DIP برقرار شده است. در نتیجه زیرکلاس‌های کامند می‌توانند تغییر کنند و Invoker از آن بی‌تاثیر است. همچنین در واقع کامند به عنوان یک موجودیت میانی بین Invoker و Receiver نشسته است و یک سطح indirection ایجاد می‌کند و Invoker به Receiver دید مستقیم ندارد. در نتیجه در این بخش‌ها جفت‌شدگی مطلوب است. البته مشتری که نقش پیکربند را نیز ایفا می‌کند، به Invoker، ConcreteCommand و Receiver دید دارد و باید آن‌ها را بشناسد هرچند که این دید نیازی به مانا بودن ندارد ولی به دلیل دید مستقیم، در این قسمت جفت‌شدگی مطلوب نیست. همچنین ConcreteCommand و Receiver نیز دید مستقیم دارند و انتزاع وجود ندارد.

Event Logging: در این الگو، همان‌طور که در شکل ساختار آن مشخص است، Device ها به EventRemembered ها دید مستقیم دارند و در زمان اجرا، اشیایی از EventRemembered نمونه‌گیری می‌کنند. بنابراین جفت‌شدگی میان این دو موجودیت بالا است. از واسط، انتزاع و در کل روش‌هایی برای کاهش جفت‌شدگی استفاده نشده است.

مقایسه: جفت‌شدگی در الگوی سوم بالا است به‌دلیل توضیحات ارائه شده. اما در دو الگوی اول همان‌طور که شرح داده شده است، در قسمت‌هایی جفت‌شدگی مطلوب و در قسمت‌هایی نامطلوب است.

۶-۲ انسجام

Builder: در این الگو، کارگردان و سازنده‌ها از انسجام خوبی برخوردار هستند و عملیات بی‌ربط در کلاس‌ها دیده نمی‌شود. مسولیت اعمال الگوریتم کلی ساخت در کارگردان تعریف شده است و پیاده‌سازی عملیاتی که در واسط سازنده تعریف شده‌اند، بر عهده زیرکلاس‌های سازنده است که جزئیات ساخت هر محصول را پیاده‌سازی می‌کنند. درواقع این‌ها اشیا متخصص آفرینش هستند که این ساختار به افزایش انسجام کمک می‌کند. مشتری نیز از تعریف الگوریتم ساخت و همچنین جزئیات ساخت هر محصول منتزع شده و این عملیات در کلاس‌های خود تعریف شده‌اند که نشانه انسجام مناسب در این الگو است.

Command: در این الگو، Command یک شی متخصص است که به درخواست یا پیغام، هویت داده است و همچنین در سایر کلاس‌ها نیز عملیات نامرتبط به هم انجام نمی‌پذیرد و موجودیت‌ها انسجام مطلوب دارند.

Event Logging: در این الگو، کلاس‌هایی به نام Device تعریف می‌شوند که وظیفه‌شان پایش یک دستگاه بیرونی است. این‌ها باید تشخیص دهند که یک رویداد خارجی رخ داده و سپس زنجیره‌ای از تعاملات

را برای پاسخ‌دهی آغاز کنند. برای هر رویداد، یک شی از کلاسی به نام EventRemembered ساخته می‌شود که نماینده‌ی آن رویداد است. این شی شامل اتریبیوت‌هایی است که اطلاعات مربوط به رویداد در آن‌ها ذخیره می‌شود. این اشیا نه تنها داده‌های مربوط به رویداد را در خود نگه می‌دارند، بلکه ممکن است خودشان بتوانند رفتارهایی از خود نشان دهند. به‌عنوان مثال، می‌توانند گزارش تولید کنند، یا حتی خودشان عملیات مربوط به آن رویداد را آغاز نمایند. یعنی بسته به محتوای اتریبیوت‌هایشان، می‌توانند تشخیص دهند که چه پاسخی باید داده شود و اجرای بخشی از این پاسخ را نیز بر عهده بگیرند. در نتیجه، به این شکل انسجام افزایش یافته است و جداسازی دغدغه‌ها دیده می‌شود. اما باید توجه کرد که اشیای EventRemembered ممکن است به مرور زمان بزرگ‌تر و پیچیده‌تر شوند و باید مواظب از دست رفتن انسجام آن با اضافه شدن داده و رفتار نامرتبط و حرکت به سمت God Class جلوگیری کرد. همینطور برای Device نیز به همین شکل باید مواظب انسجام آن بود.

مقایسه: الگوها از انسجام مطلوبی برخوردار هستند. توضیحات مفصل در هر قسمت ارائه شده است.

۲-۷ پیکربندی

Builder: در این الگو، یک پیکربند ثالث که همان مشتری است، وظیفه نمونه‌گیری از زیرکلاس Builder و همچنین نمونه‌گیری از کارگردان را دارد که شی ساخته شده از زیرکلاس سازنده را به کارگردان به عنوان پارامتر در هنگام نمونه‌گیری ارسال می‌کند و بدین شکل پیکربندی انجام شده و الگو در زمان اجرا محقق می‌شود.

Command: در این الگو، یک پیکربند ثالث که همان مشتری است، وظیفه نمونه‌گیری از زیرکلاس کامند و Receiver را داشته و همینطور شی زیرکلاس کامند و Invoker را نیز پیکربندی می‌کند و الگو در زمان اجرا محقق می‌شود.

Event Logging: در این الگو، دید از سمت Device به EventRemembered وجود دارد به منظور نمونه‌گیری از آن‌ها در زمان اجرا برای آغاز زنجیره تعاملی. این دید در زمان کامپایل محقق است و نیاز به پیکربندی ندارد. اما نمونه‌گیری از خود کلاس‌های Device، تخصیص آن‌ها به دستگاه بیرونی و اجرای متد مربوط به پایش، نیاز دارد تا موجودیتی این کار را انجام دهد که می‌توان به نوعی پیکربندی در نظر گرفت.

مقایسه: دو الگوی اول نیاز به پیکربند ثالث دارد. در رابطه با الگوی سوم توضیح مفصل‌تری آمده است.

۲-۸ انعطاف‌پذیری

Builder: با جداسازی الگوریتم و فرایند کلی ساخت از جزئیات ساخت هر محصول و همچنین برقراری DIP بین کارگردان و سازنده، امکان تغییر در محصولات و جزئیات پیاده‌سازی وجود دارد و کارگردان از آن تاثیر نمی‌پذیرد. همچنین با تعریف ریزدانه عملیات ساخت، تنوع بیش‌تری برای کارگدان محیا می‌شود تا الگوریتم‌های متنوع تعریف کند. امکان تغییر سازنده در زمان اجرا نیز وجود دارد و کارگردان از آن تاثیر نخواهد برد و متوجه آن نمی‌شود. در نتیجه انعطاف‌پذیری در حد مطلوب وجود دارد.

Command: در این الگو، ارتباط بین Invoker و کامند از طریق واسط است و DIP برقرار بوده و Invoker از تغییرات زیرکلاس‌های کامند تغییر نمی‌پذیرد. همچنین دید مستقیم از Invoker به Receiver نیز وجود ندارد. این که پیغام به یک شی مستقل تبدیل شده و هویت پیدا کرده است، قابلیت‌های مختلفی نظیر صف‌بندی، زمان‌بندی، لاگ کردن و غیره را میسر کرده است. قابلیت ذخیره تاریخچه و همچنین عملیات undo فراهم شده است.

Event Logging: در این الگو، دید از سمت Device به EventRemembered وجود دارد به منظور نمونه‌گیری از آن‌ها و آغاز زنجیره تعاملی بر اساس تغییرات دستگاه بیرونی و این دید در زمان کامپایل محقق است که باعث صلب بودن این ساختار می‌شود. برای مثال نمی‌توان در زمان اجرا EventRemembered ها را تغییر داد یا جایگزین کرد. بنابراین انعطاف‌پذیری نامطلوب است. اما از این جهت که می‌توان به EventRemembered ویژگی‌ها و عملیات جدید به مرور زمان اضافه کرد، این قسمت تاحدی قابل توسعه است.

مقایسه: دو الگوی اول با روش‌های شرح داده شده انعطاف‌پذیری را میسر می‌کنند. در الگوی سوم انعطاف‌پذیری تا حدی نامطلوب است.

۲-۹ کارایی

Builder: پس از اعمال این الگو، کلاس کارگردان، واسط سازنده و زیرکلاس‌های سازنده پدیدار شده‌اند. در نتیجه تعداد اشیا پس از اعمال این الگو افزایش می‌یابد. همچنین سربار ناشی از فراخوانی‌های عملیات ریزدانه برای ساخت و ارتباطات میان اشیا نیز وجود دارد و تعداد پیام‌های رد و بدل شده افزایش یافته است.

Command: در این الگو، پیامی که قبلا به صورت مستقیم ارسال می‌شد، اکنون به شکل شی مستقل در آمده و می‌توان مانند یک شی معمولی با آن برخورد کرد. ممکن است به مرور زمان تعداد اشیا زیاد شده و حالت خود را نیز ذخیره کنند که سربار زیادی به وجود می‌آید. همچنین وجود یک سطح indirection نیز

کارایی را کاهش می‌دهد نسبت به قبل که انتقال پیام آنی بود و پیام هویت مستقل نداشت.

Event Logging: در این الگو، Device حسگر بیرونی را پایش می‌کند و در موارد نیاز، از EventRemembered نمونه‌گیری می‌کند برای آغاز زنجیره تعاملی. همچنین این اشیای ایجاد شده می‌توانند در سیستم مانا شده و به دلایل مختلفی مثل پردازش‌های آتی، ثبت شوند. به مرور زمان تعداد این اشیا می‌تواند زیاد شده و روی منابع سیستم فشار وارد کند.

مقایسه: هر سه الگو کارایی را کاهش می‌دهند باتوجه به توضیحات ارائه شده برای هرکدام.

۲-۱۰ شی جعلی

Builder: هیچ‌کدام از کلاس‌های کارگردان، واسط سازنده و زیرکلاس‌هایش ناشی از تحلیل قلمرو مسئله نیستند و بنابراین اشیا جعلی محسوب می‌شوند.

Command: در این الگو، شی Command ناشی از تحلیل قلمرو مسئله نبوده و قبلا که پیام به صورت آنی انتقال می‌یافت اکنون در کامند هویت مستقل یافته است تا اهدف این الگو محقق شود و این شی جعلی است.

Event Logging: در این الگو، EventRemembered ناشی از تحلیل قلمرو مسئله نیست و بنابراین شی جعلی محسوب می‌شود.

مقایسه: هر سه الگو اشیای جعلی تعریف می‌کنند.

۲-۱۱ انتشار تغییرات

Builder: کارگردان با واسط سازنده رابطه دارد، بنابرین در این بخش DIP رعایت شده است و این باعث می‌شود تغییرات زیرکلاس‌های سازنده به کارگردان منتشر نشوند و کارگردان صرفا به واسط وابسته است و تا زمانی که واسط تغییر نکند، کارگردان نیز تغییر نمی‌کند. اما مشتری به مجموعه دید دارد و با تغییر کارگردان یا زیرکلاس‌های سازنده، مشتری نیز بالقوه تغییر می‌کند. همچنین تغییر واسط سازنده مانند اضافه کردن یک متد جدید، باعث می‌شود تمام زیرکلاس‌هایش تغییر کنند و آن متد را پیاده‌سازی کنند که این مصداق تغییر منتشر شونده است.

Command: در این الگو، تغییرات Invoker و Receiver به یکدیگر منتشر نمی‌شوند زیرا ارتباط مستقیم وجود ندارد و یک سطح indirection شکل گرفته است. همچنین ارتباط بین Invoker و Command

از طریق واسط بوده و DIP برقرار است. در نتیجه تغییرات زیرکلاس‌های کامند به آن منتشر نمی‌شود. البته مشتری که پیکربند نیز هست، به Invoker، Receiver و ConcreteCommand دید دارد و از تغییرات تاثیر می‌پذیرد.

Event Logging: در این الگو، Device به EventRemembered دید دارد به منظور نمونه‌گیری و آغاز زنجیره تعاملی که این دید در زمان کامپایل محقق می‌شود که صلب است. از مکانیزم‌های کنترل جفت‌شدگی و انتزاع استفاده نشده و این باعث می‌شود تغییرات EventRemembered به Device منتشر شود.

مقایسه: توضیحات مفصل برای هرکدام از الگوها بیان شده است و هرکدام در قسمت‌هایی جلوی انتشار تغییرات را گرفته و در قسمت‌هایی انتشار وجود دارد.

۱۲-۲ سهولت پیاده‌سازی

Builder: با استفاده از یک زبان شی‌گرا می‌توان این الگو را بدون مشکل پیاده‌سازی کرد. البته باید فرایند ساخت یعنی الگوریتم کلی، اجازه ساخت اشیا با ساختارهای داخلی مختلف را بدهد. همچنین کارگردان دید یک‌سویه به سازنده دارد و پیاده‌سازی آن نسبت به یک دید دوسویه آسان‌تر است. همچنین متدهای BuildPart در سازنده باید از سطح ریزدانگی مناسبی برخوردار باشند تا تنوع الگوریتم در سمت کارگردان بیش‌تر شود.

Command: این الگو رایج است و در یک زبان شی‌گرا می‌توان به آسانی آن را پیاده‌سازی کرد که ساختار و رفتار الگو در بخش مربوطه شرح داده شده است.

Event Logging: پیاده‌سازی این الگو چالش بخصوصی نداشته و با یک زبان شی‌گرا می‌توان آن را پیاده‌سازی کرد.

۱۳-۲ موارد کاربرد

Builder:

- زمانی که می‌خواهیم الگوریتم ایجاد یک شی پیچیده را از این که چه قطعاتی برای محصول نهایی استفاده و چگونه به هم وصل و پیکربندی بشوند، مستقل کنیم. الگوریتم ساخت در سمت کارگردان ثابت بوده و همان الگوریتم می‌تواند محصولات مختلفی بسازد. درواقع الگوریتم را جدا کرده و از فرایند دقیق ساخت مستقل کردیم. فرایند دقیق ساخت بر عهده سازنده است.
- زمانی که باید فرایند ساخت یعنی الگوریتم، اجازه ساخت اشیا با ساختارهای داخلی مختلف را بدهد.

Command:

- وقتی می‌خواهیم اشیا را با اعمال مختلف پارامتریزه کنیم، هدف این است که به آن‌ها درجه‌ای از آزادی بدهیم تا به جای اجرای همیشگی یک عمل مشخص، بتوانند بر اساس یک کامند که در اختیارشان قرار می‌گیرد، عمل متفاوتی انجام دهند. این کامند از پیش مشخص نیست و در زمان اجرا تعیین و به شی متناظر می‌شود، یعنی شی با آن پیکربندی می‌شود. بنابراین، نحوه اجرای عمل نیز در همان زمان اجرا مشخص می‌گردد و حتی می‌توان این پیکربندی را در ادامه تغییر داد. به جای این که پیغامی مستقیماً از فرستنده به گیرنده ارسال شود، این پیغام اکنون به یک شی میانی تبدیل شده که همان کامند است. گویا خود پیام را به یک آبجکت تبدیل کرده‌ایم.
- زمانی که بخواهیم درخواست‌ها را در زمان اجرا مشخص کنیم، به صف کنیم و با ترتیبی که مورد نظر هست در زمان‌های مختلف اجرا کنیم.
- با استفاده از این الگو می‌توان تغییرات سیستم را لاگ کرد، به این صورت که هر بار که یک کامند اجرا می‌شود، امکان ثبت آن وجود دارد. این کار از طریق افزودن عملیاتی مانند load و store به اینترفیس کامند انجام می‌شود تا بتوان این اشیا را ذخیره‌سازی و در صورت نیاز مجدداً بارگذاری کرد. در نتیجه، اگر سیستم دچار کرش شود، می‌توان از آخرین وضعیت معتبر سیستم (که می‌شناسیم) شروع کرد و کامندهایی را که پس از آن اجرا شده‌اند، به ترتیب دوباره load و اجرا کرد تا سیستم به همان وضعیت پیش از کرش بازیابی شود. این مکانیسم، قابلیت restoration را ممکن می‌سازد.
- سیستم را می‌توان حول محور عملیات درشت‌دانه‌ای طراحی کرد که خود از چند عملیات ریزتر تشکیل شده‌اند؛ به این ساختار Macro Command گفته می‌شود. در این رویکرد، به‌جای آنکه عملیات سیستم تنها از طریق تبادل مستقیم پیام بین اشیا انجام شود، همه چیز از طریق اجرای کامندها صورت می‌گیرد. Macro Command در واقع کامندی است که شامل چندین کامند دیگر است و می‌تواند یک عملیات پیچیده را در قالب یک واحد مستقل مدیریت کند. این ساختار به ویژه در سیستم‌هایی مانند سامانه‌های پردازش تراکنش و سیستم‌های بانکی بسیار مفید است، چرا که امکان تعریف پویا و ترکیبی کامندها در زمان اجرا را فراهم می‌کند و نیازی به پیش‌بینی همه حالت‌ها از قبل نیست. همچنین، با استفاده از این ساختار می‌توان قابلیت‌هایی نظیر undo، redo، صف‌بندی، لاگ کردن و بازگردانی سیستم را به‌صورت یکپارچه و قدرتمند پیاده‌سازی کرد. در نتیجه، کل عملیات سیستمی می‌توانند از طریق این کامندها و اجرای آن‌ها مدیریت شوند، که سطح بالایی از انعطاف‌پذیری را فراهم می‌سازد.

Event Logging: از این الگو زمانی استفاده می‌شود که بخواهیم وقوع یک رویداد را با اهداف گوناگون ثبت کنیم. این اهداف ممکن است شامل تحلیل‌های پس از وقوع یا after-the-fact analysis باشند. مانند استخراج آمار، شناسایی سناریوها و یافتن الگوها میان رویدادها. چنین تحلیل‌هایی به‌ویژه در سیستم‌های

real-time، مانند سامانه‌های هشدار یا کنترل صنعتی که مبتنی بر رویداد[۱۹] هستند، اهمیت زیادی دارند. در برخی موارد نیز ثبت رویداد الزام قانونی دارد و سیستم باید طبق مقررات، وقوع برخی رویدادها را مستندسازی کند. بنابراین دلایل متنوعی می‌تواند پشت نیاز به ثبت وقایع باشد، از تحلیل‌های فنی گرفته تا رعایت الزامات قانونی. به‌محض اینکه نیاز به ثبت یک رویداد مطرح شود، باید ذهن به سمت این الگو رود. در این الگو، هر رویداد به‌صورت یک شی مستقل مدل می‌شود که دارای هویت و مانایی در طول زمان است. این مدل امکان توسعه‌پذیری فراهم می‌کند. یعنی می‌توان ویژگی‌ها و عملیات جدیدی به این اشیا افزود و حتی مسئولیت برخی عملیات را به خود آن‌ها واگذار کرد.

مقایسه: موارد کاربرد هر الگو به صورت مفصل در هر قسمت شرح داده شده است و ارتباط آن‌ها در قسمت الگوهای مرتبط بیان شده‌اند.

۱۴-۲ کپسوله‌سازی

Builder: کلاس کارگردان با واسط سازنده در ارتباط است و هیچ دیدی نسبت به زیرکلاس‌ها و جزییات داخلی ندارد. بنابرین کپسوله‌سازی[۲۰] در این قسمت رعایت شده است. البته از سمت پیکربند یعنی مشتری، به دلیل این که مشتری به زیرکلاس‌ها دید داشته و مستقیما متد getResult را فراخوانی می‌کند، در این بخش کپسوله‌سازی به خوبی دیده نمی‌شود.

Command: در این الگو Invoker و Receiver از هم خبر ندارند. Invoker صرفا با واسط کامند ارتباط دارد. کامند و Receiver نیز از طریق ارسال پیام ارتباط می‌گیرند و محتوای درونی و جزئیات آشکار نمی‌شود پس کپسوله‌سازی رعایت شده است.

Event Logging: در این الگو، از واسط و انتزاع استفاده نشده و Device مستقیما به EventRemembered دید داشته و می‌تواند در مواقع نیاز، از آن به منظور آغاز زنجیره تعاملی نمونه‌سازی کند که این دید مستقیم بوده و نمایش داخلی در صورت پنهان نکردن، در دسترس است. اما Device نوعا از آن نمونه‌گیری می‌کند و زنجیره را آغاز می‌کند و از نمایش داخلی آن استفاده نمی‌کند.

مقایسه: در الگوی اول و سوم امکان نقض کپسوله‌سازی وجود دارد.

---

[۱۹] event-driven
[۲۰] encapsulation

۲-۱۵ جداسازی دغدغه‌ها

Builder: در این الگو جداسازی دغدغه‌ها به خوبی دیده می‌شود و فرایند کلی ساخت از جزئیات و ساختار داخلی جدا شده است. همچنین مفهوم شی متخصص در این الگو دیده می‌شود.

Command: مسئولیت ارسال درخواست، مدیریت اجرای آن و نگهداری تاریخچه و غیره در کلاس‌های جدای از هم تعریف می‌شوند و در کلاس‌ها رفتار نامرتبط و غیرمنسجم وجود ندارد. حتی Invoker اطلاعی از گیرنده پیام ندارد.

Event Logging: در این الگو، کلاس‌هایی به نام Device تعریف می‌شوند که وظیفه‌شان پایش یک دستگاه بیرونی است. این‌ها باید تشخیص دهند که یک رویداد خارجی رخ داده و سپس زنجیره‌ای از تعاملات را برای پاسخ‌دهی آغاز کنند. برای هر رویداد، یک شی از کلاسی به نام EventRemembered ساخته می‌شود که نماینده‌ی آن رویداد است. این شی شامل اتریبیوت‌هایی است که اطلاعات مربوط به رویداد در آن‌ها ذخیره می‌شود. این اشیا نه تنها داده‌های مربوط به رویداد را در خود نگه می‌دارند، بلکه ممکن است خودشان بتوانند رفتارهایی از خود نشان دهند. به‌عنوان مثال، می‌توانند گزارش تولید کنند، یا حتی خودشان عملیات مربوط به آن رویداد را آغاز نمایند. یعنی بسته به محتوای اتریبیوت‌هایشان، می‌توانند تشخیص دهند که چه پاسخی باید داده شود و اجرای بخشی از این پاسخ را نیز بر عهده بگیرند. در نتیجه، به این شکل انسجام افزایش یافته است و جداسازی دغدغه‌ها دیده می‌شود.

مقایسه: هر الگو در قسمت‌هایی با استفاده از روش‌های بیان شده به جداسازی دغدغه‌ها پرداخته‌اند.

۲-۱۶ مزایا و معایب (تبعات)

:Builder

مزایا:

- اجازه ساخت محصولات با ساختارهای داخلی متفاوت را می‌دهد.
- کد ایجاد و کد ساخت نهایی یعنی ساختار داخلی از هم جدا هستند. یعنی الگوریتم ساخت جدا در سمت کارگردان و الگوریتم ساختار داخلی جدا در سمت سازنده.
- این الگو یک کنترل دقیق و ریزدانه به فرایند ساخت ارائه می‌کند. زیرا نوعا در سازنده، عملیات ریزدانه تعریف می‌شوند و این اجازه انجام گام به گام را می‌دهد تحت نظر کارگردان.

:Command

مزایا:

- این الگو باعث می‌شود که فرستنده‌ی پیام (یعنی شی فراخواننده) از گیرنده‌ی نهایی پیام (آن که واقعاً کار را انجام می‌دهد) جدا و مستقل شود. این یعنی یک نوع decoupling یا گسستگی ایجاد می‌شود بین مبدا و مقصد عملیات. فرستنده دیگر نیاز ندارد بداند که چه کسی دقیقاً قرار است عمل را انجام دهد یا حتی اینکه عمل دقیقاً چیست. تنها کاری که می‌کند این است که یک کامند را اجرا می‌کند. این خود کامند است که اطلاعات لازم برای اجرای عمل را دارد و می‌داند به کدام گیرنده پیام بدهد. این وابستگی تا قبل از پیکربندی مشخص نیست و تازه در زمان اجراست که کامند به گیرنده‌ی مناسب متصل می‌شود. در نتیجه، ارتباطات در سیستم انعطاف‌پذیرتر و غیرمستقیم می‌شوند.
- کامندها در این الگو خودشان به‌صورت واقعی شی هستند. یعنی هر کامند یک شی مستقل با هویت مشخص در سیستم محسوب می‌شود. این اشیا فقط یک پیام گذرا نیستند، بلکه در طول زمان در سیستم حضور دارند و می‌توان مانند سایر اشیا با آن‌ها تعامل داشت، آن‌ها را ذخیره کرد، انتقال داد یا حتی تغییر داد. به دلیل این که کامندها شی هستند، می‌توان زیرکلاس‌های مختلفی برای انواع عملیات مختلف تعریف کرد.
- می‌شود کامندها را با یکدیگر تجمیع و ترکیب کرد و کامندهای درشت‌دانه مرکب ساخت.
- افزودن کامندهای جدید در این الگو بسیار ساده و بدون ایجاد تغییر در بخش‌های دیگر سیستم انجام‌پذیر است. همان‌طور که در نمودار کلاس دیده می‌شود، اگر یک کامند جدید به سیستم اضافه شود، این تغییر تنها بر پیکربند یا همان مشتری اثر می‌گذارد که وظیفه‌ی آشنا کردن کامند با سایر اجزای سیستم را دارد. سایر بخش‌ها، مانند Invoker، هیچ نیازی به آگاهی از جزئیات کامند جدید ندارند و تغییری در آن‌ها ایجاد نمی‌شود. این ساختار به‌خوبی اصل DIP را رعایت کرده و امکان گسترش سیستم را بدون آسیب زدن به ساختار فعلی فراهم می‌کند. بنابراین می‌توان مجموعه‌ای از کامندهای متنوع را به مرور به سیستم افزود، بدون اینکه نیاز به بازنویسی کدهای موجود باشد.

عیب: به دلیل آن که پیغام‌ها تبدیل به شی مستقل شده و هویت پیدا کرده‌اند و می‌توانند حالت ذخیره کنند، تعداد اشیا می‌تواند زیاد شده و روی منابع سیستمی فشار وارد کند. همچنین وجود یک سطح indirection نیز کارایی را کاهش می‌دهد.

Event Logging: از این الگو به عنوان اولین مورد هویت دادن به عناصر ساده یاد می‌شود. یعنی عناصر داده‌ای ساده که به آن‌ها هویت آبجکتی داده شده است. اطلاعاتی که در ابتدا فقط به‌صورت داده‌های خام هستند، با این الگو به صورت یک شی با هویت مستقل در سیستم مدل می‌شوند. در نگاه شی‌گرا، نباید از تعریف کلاس برای چنین داده‌های ساده‌ای ترسید. هرچند ممکن است در ابتدا کلاس‌های حاصل بسیار ریزدانه و ابتدایی به نظر برسند، اما در عمل این کلاس‌ها در طول زمان رشد می‌کنند و با توسعه‌ی سیستم، اتریبیوت‌ها

و عملیات بیشتری به آن‌ها افزوده می‌شود. بنابراین، همین اجزا ساده می‌توانند به اجزایی غنی و پیچیده تبدیل شوند. رویدادهایی که در سیستم رخ می‌دهند، حتی اگر اطلاعات محدودی درباره آن‌ها موجود باشد، وقتی به شکل اشیا در سیستم مدل شوند، زمینه‌ای فراهم می‌شود تا عملیات متنوعی بر روی آن‌ها تعریف گردد. در واقع، با تبدیل یک رویداد به شی، می‌توان آن را در سیستم نگه داشت، از آن گزارش‌گیری کرد، وضعیت آن را تحلیل نمود یا حتی واکنش‌هایی مبتنی بر خصوصیاتش تعریف کرد. این رویکرد نشان می‌دهد که حتی داده‌هایی که در ابتدا ساده و محدود به نظر می‌رسند، در قالب شی می‌توانند امکانات و قابلیت‌هایی فراتر از تصور اولیه به سیستم اضافه کنند. پس با این جداسازی شی مخصوص پایش و اشیایی که طبق رویدادها ساخته می‌شوند، دغدغه‌ها جدا شده و همچنین مزایای تبدیل رویداد به شی که قبلا اشاره شد، به‌دست می‌آیند. از معایب الگو این است که از مکانیزم‌های کنترل جفت‌شدگی مانند انتزاع، واسط و غیره استفاده نشده است و همینطور الگو در زمان کامپایل محقق شده و ساختار صلبی دارد. یعنی امکان تخصیص و تعویض اشیای مربوط به رویداد در زمان اجرا وجود ندارد. همچنین تعداد اشیای ساخته شده به مرور زمان بیشتر می‌شود و در صورت مدیریت نکردن بهینه حافظه، بر روی منابع سیستم فشار وارد می‌کند. همچنین به این اشیای پایشگر و رویداد، به مرور زمان می‌توان قابلیت‌ها و ویژگی‌های جدید اضافه شود و در صورت مراقبت نکردن ممکن است انسجام از دست رفته و به سمت God Class حرکت کند.

۲-۱۷ الگوهای مرتبط

Builder: بسیاری از طراحی‌ها با استفاده از الگوی Factory Method آغاز می‌شوند (چون ساده‌تر بوده و از طریق زیرکلاس‌ها قابلیت سفارشی‌سازی بیشتری دارد) و به‌مرور به سمت الگوهای Abstract Factory، Prototype یا Builder پیش می‌روند (که انعطاف‌پذیرتر اما پیچیده‌تر هستند). الگوی Builder بر ساخت مرحله‌به‌مرحله‌ی اشیای پیچیده تمرکز دارد. در مقابل، Abstract Factory در ایجاد خانواده‌ای از اشیای مرتبط تخصص دارد. Abstract Factory بلافاصله محصول را برمی‌گرداند، در حالی که Builder اجازه می‌دهد قبل از دریافت محصول، چند مرحله‌ی اضافی در ساخت اجرا شوند. می‌توان از Builder برای ساخت درخت‌های ترکیبی (Composite) پیچیده استفاده کرد، چون مراحل ساخت آن را می‌توان طوری برنامه‌ریزی کرد که به صورت بازگشتی کار کند. می‌توان Builder را با الگوی Bridge ترکیب کرد. به گونه‌ای که کلاس کارگردان نقش انتزاع را دارد و سازنده‌های مختلف نقش پیاده‌سازی‌ها را بازی می‌کنند. الگوهای Abstract Factory، Builder و Prototype را می‌توان به‌صورت Singleton نیز پیاده‌سازی کرد.

Command: الگوهای Chain of Responsibility، Command، Mediator و Observer هر کدام روش‌های متفاوتی برای اتصال فرستنده‌ها و گیرنده‌های درخواست ارائه می‌دهند. در الگوی زنجیره مسئولیت،

می‌توان هندلرها را به‌صورت Command پیاده‌سازی کرد. در این حالت، می‌توان عملیات‌های مختلفی را روی یک شی زمینه واحد که به‌صورت درخواست نمایان شده است، اجرا کرد. اما رویکرد دیگری هم وجود دارد که در آن، خود درخواست به‌صورت یک شی کامند تعریف می‌شود. در این حالت، می‌توان یک عملیات یکسان را روی مجموعه‌ای از زمینه‌های مختلف که در یک زنجیره به هم متصل شده‌اند، اجرا کرد. برای پیاده‌سازی قابلیت undo، می‌توان الگوهای کامند و Memento را با یکدیگر ترکیب کرد. در این حالت، کامندها مسئول انجام عملیات مختلف روی یک شی هدف هستند، در حالی که ممنتو وضعیت شی را درست قبل از اجرای کامند ذخیره می‌کنند. الگوهای کامند و استراتژی ممکن است مشابه به‌نظر برسند، زیرا هر دو امکان پارامتریزه کردن یک شی با یک عمل خاص را فراهم می‌کنند، اما هدف‌های آن‌ها بسیار متفاوت است. الگوی کامند به شما امکان می‌دهد هر عملیاتی را به یک شی تبدیل کنید. پارامترهای آن عملیات به فیلدهای آن شی تبدیل می‌شوند. این تبدیل به شما اجازه می‌دهد اجرای عملیات را به تعویق بیندازید، آن را در صف قرار دهید، تاریخچه‌ای از کامندها ذخیره کنید، کامندها را به سرویس‌های راه دور بفرستید و غیره. در مقابل، الگوی استراتژی معمولاً روش‌های مختلفی برای انجام یک کار خاص را توصیف می‌کند و به شما اجازه می‌دهد این الگوریتم‌ها را در چارچوب یک کلاس زمینه، با یکدیگر تعویض کنید. زمانی که نیاز دارید نسخه‌هایی از کامندها را برای تاریخچه ذخیره کنید، می‌توانید از الگوی Prototype کمک بگیرید. الگوی Visitor را می‌توان نسخه‌ای قدرتمند از الگوی کامند دانست، زیرا اشیای آن می‌توانند عملیاتی را روی اشیای مختلف از کلاس‌های گوناگون اجرا کنند. از الگوی Composite نیز برای ساخت کامندهای ترکیبی یا همان Macro Command استفاده می‌شود.

Event Logging: در این الگو، اشیای Observers به تغییرات در شی سابجکت واکنش نشان می‌دهند. در زمینه الگوی Event Logging، می‌توان عناصر ثبت‌کننده را به‌عنوان ناظرهایی در نظر گرفت که به تغییرات در سیستم واکنش نشان داده و رویدادها را ثبت می‌کنند. یا آن‌ها می‌توانند باعث تغییر در سابجکت شوند. الگوی Command نیز با الگوی Event Logging مرتبط و مشابه است. در این الگو، پیام‌ها به‌صورت اشیا کپسوله می‌شوند. با ترکیب این الگو با ثبت رویداد، می‌توان هر فرمان اجراشده را به‌عنوان یک رویداد ثبت کرد، که برای مقاصدی مانند undo/redo یا حسابرسی مفید است. الگوی Chain of Responsibility نیز می‌تواند در ساختاردهی به سیستم‌های ثبت رویداد مفید باشد. در این الگو، درخواست‌ها از طریق زنجیره‌ای از اشیا عبور می‌کنند تا یکی از آن‌ها آن را پردازش کند. الگوی Memento نیز در زمینه ثبت وضعیت سیستم مرتبط است. این الگو امکان ذخیره و بازیابی وضعیت اشیا را فراهم می‌کند، که می‌تواند در ترکیب با ثبت رویداد برای بازسازی وضعیت سیستم در زمان‌های گذشته مفید باشد. الگوی Strategy می‌تواند برای تعیین نحوه پردازش رویدادها بر اساس نوع آن‌ها به کار رود. با استفاده از این الگو، می‌توان منطق‌های مختلفی برای پردازش انواع مختلف رویدادها تعریف کرد و در زمان اجرا استراتژی مناسب را انتخاب نمود.

۱۸-۲ اصول شی‌گرایی

OCP ۱-۱۸-۲

Builder: رابطه بین کارگردان و سازنده در سطح انتزاع است یعنی کارگردان با واسط سازنده رابطه دارد. بنابراین DIP برقرار بوده که OCP را فراهم می‌کند و این یعنی هرگونه تغییر در زیرکلاس‌های سازنده تاثیری بر کارگردان نمی‌گذارد و کارگردان از آن منتزع است. اما مشتری که بیرون مجموعه بوده و نقش پیکربند دارد، دید مستقیم به زیرکلاس‌های سازنده داشته و از تغییر آن‌ها منتزع نیست و OCP در این قسمت مشکل دارد.

Command: در این الگو، Invoker با واسط کامند ارتباط دارد و DIP در این قسمت برقرار است و در نتیجه OCP برقرار میشود. هرگونه تغییر در زیرکلاس‌های کامند به Invoker منتشر نمی‌شود و می‌توان کامندهای جدید اضافه کرد. تا زمانی که واسط کامند تغییر نکند Invoker نیز تغییر نمی‌کند. ولی در سمت مشتری، دید مستقیم به زیرکلاس‌های کامند وجود دارد و مشتری از تغییرات آن تاثیر می‌پذیرد که این نشان از مشکل در OCP در این بخش است.

Event Logging: در این الگو، همان‌طور که در شکل ساختار آن مشخص است، Device ها به EventRemembered ها دید مستقیم دارند و در زمان اجرا، اشیایی از EventRemembered نمونه‌گیری می‌کنند. بنابراین جفت‌شدگی میان این دو موجودیت بالا است. از واسط، انتزاع و در کل روش‌هایی برای کاهش جفت‌شدگی استفاده نشده است. بنابرین OCP رعایت نمی‌شود و تغییرات منتشر می‌شوند. اما تا حدی توسعه‌پذیری در EventRemembered موجود است و می‌توان ویژگی‌ها و عملیات به‌مرور زمان به آن اضافه کرد ولی امکان modification نیز وجود دارد.

مقایسه: بجز الگوی سوم، بقیه الگوها از روش‌هایی برای رعایت این اصل استفاده کرده‌اند که شرح داده شده است.

LSP ۲-۱۸-۲

Builder: در این الگو، زیرکلاس‌های سازنده، رابطه is-a با پدر دارند و هرکدام از زیرکلاس‌ها می‌تواند به عنوان سازنده به کارگردان معرفی شده و استفاده شود. بنابراین اصل LSP به خوبی رعایت شده است.

Command: در این الگو، زیرکلاس‌های کامند، رابطه is-a با پدر دارند و هرکدام از زیرکلاس‌ها می‌تواند به عنوان کامند به Invoker معرفی شود. بنابراین اصل LSP برقرار است.

Event Logging: در این الگو از وراثت استفاده نشده است.

مقایسه: بجز الگوی سوم که از وراثت استفاده نمی‌کند، بقیه الگوها این اصل را رعایت می‌کنند.

ISP ۳-۱۸-۲

Builder: در این الگو، فرایند کلی ساخت در کلاس کارگردان و عملیات، ساختار داخلی و جزئیات ساخت در سازنده تعریف شده است که تخصص آفرینش دارد. بنابراین واسط‌های موجود انسجام خوبی داشته، عملیات نامرتبط انجام نداده و اصل ISP به خوبی محقق شده است.

Command: در این الگو، واسط موجود که Invoker با آن ارتباط دارد، عملیات بی‌ربط تعریف نکرده و واسط از انسجام خوبی برخوردار است و عملیات مربوط به کامند مانند متد execute را ارائه می‌کند. بنابراین این اصل رعایت شده است.

Event Logging: در این الگو، وظیفه پایش دستگاه بیرونی و نمونه‌گیری از اشیای رویداد به منظور آغاز زنجیره تعاملی بر گردن Device است و اشیای رویداد نیز عملیات خود را انجام می‌دهند. لذا جداسازی دغدغه‌ها اینجا دیده شده است. اما به‌مرور زمان ممکن است ویژگی‌ها و عملیات متعدد نظیر گزارش‌گیری و غیره به اشیای رویداد اضافه شوند و این به‌مرور می‌تواند باعث پیچیدگی آن‌ها شود پس این اصل به خطر می‌افتد.

مقایسه: الگوها از روش‌هایی که شرح داده شده است استفاده کرده‌اند تا به تفکیک واسط‌ها بپردازند.

DIP ۴-۱۸-۲

Builder: کارگردان با واسط سازنده ارتباط دارد که این ارتباط در سطح انتزاع بوده و DIP برقرار است. هر گونه تغییر در زیرکلاس‌های سازنده به کارگردان منعکس نمی‌شود. اما مشتری که نقش پیکربند را ایفا می‌کند، به دلیل این که دید مستقیم به زیرکلاس‌های سازنده دارد، در مورد او DIP نقض شده است.

Command: در این الگو، Invoker با واسط کامند ارتباط دارد و از تغییرات زیرکلاس‌های کامند منتزع است. بنابراین اصل DIP در این قسمت رعایت شده است. اما از سمت مشتری که پیکربند نیز هست، دید مستقیم با زیرکلاس‌های کامند دارد. همچنین زیرکلاس‌های کامند نیز دید مستقیم به گیرنده دارند که در این قسمت‌ها اصل DIP نقض می‌شود.

Event Logging: در این الگو، Device به منظور نمونه‌گیری و آغاز زنجیره تعاملی، دید مستقیم به EventRemembered دارد و اینجا از مکانیزم‌های کنترل جفت‌شدگی مانند انتزاع و واسط و غیره استفاده نشده است و DIP برقرار نیست. تغییرات سمت EventRemembered می‌تواند روی دیوایس تاثیر بگذارد.

مقایسه: بجز الگوی سوم، بقیه در قسمت‌هایی از روش‌هایی که شرح داده شد استفاده کرده‌اند تا این اصل

را رعایت کنند.

CRP ۵-۱۸-۲

Builder: این الگو در زمان اجرا به کمک واسپاری[۲۱] محقق می‌شود. مشتری در زمان اجرا یک کارگردان را با استفاده از یک سازنده پیکربندی می‌کند و کارگردان از سازنده تخصیص یافته برای واسپاری عملیات در الگوریتم کلی ساخت استفاده می‌کند. بنابراین اصل CRP رعایت شده است.

Command: این الگو در زمان اجرا محقق شده و ارتباط Invoker با کامند و همچنین زیرکلاس‌های کامند و گیرنده، از طریق delegation صورت می‌پذیرد. مشتری در زمان اجرا باید پیکربندی انجام دهد و ساختارهای واسپاری را بسازد تا الگو محقق شود.

Event Logging: در این الگو، دید از سمت Device به EventRemembered از یک شی به کلاس است و این دید در زمان کامپایل محقق می‌شود. به این گونه نیست که دیوایس یک شی EventRemembered در اختیار داشته و به آن واسپاری کند. بلکه یک شی از EventRemembered ساخته و زنجیره تعامل از آن آغاز می‌شود و آن شی حاوی رفتار مورد نیاز برای آن رویداد است. از وراثت نیز استفاده نشده است.

مقایسه: الگوها با استفاده از روش‌های شرح داده شده این اصل را رعایت می‌کنند البته در الگوی سوم کم‌رنگ‌تر است.

PLK ۶-۱۸-۲

Builder: در این الگو دید ترایا[۲۲] دیده نشده و زنجیره پیغام[۲۳] تشکیل نمی‌شود. محصول نهایی توسط مشتری با فراخوانی متد getResult روی زیرکلاس سازنده دریافت می‌شود که نقض PLK نیست. پس این اصل به خوبی رعایت شده است.

Command: در این الگو دید ترایا وجود نداشته و زنجیره پیغام تشکیل نمی‌شود. Invoker کامند را اجرا می‌کند و کامند عملیات Receiver خود را اجرا کرده و پاسخ را بازمیگرداند ولی خود شی را برنمی‌گرداند و دید ترایا ایجاد نمی‌شود. پس این اصل رعایت شده است.

Event Logging: در این الگو، دید ترایا وجود نداشته و زنجیره پیغام تشکیل نمی‌شود. دیوایس متد مربوط به پایش را اجرا کرده و در مواقع لازم، برای رویداد مربوطه یک شی از EventRemembered می‌سازد

[۲۱] delegation
[۲۲] transitive visibility
[۲۳] message chain

و زنجیره تعاملی آغاز می‌گردد.

مقایسه: الگوها این اصل را رعایت می‌کنند.

۱۹-۲ الگوهای GRASP

۱-۱۹-۲ Information Expert

Builder: در این الگو، الگوریتم کلی ساخت در کلاس کارگردان قرار دارد که با فراخوانی عملیات از واسط سازنده، محصول نهایی ساخته می‌شود. اگر طراحی به گونه‌ای باشد که کارگردان حاوی اطلاعات لازم برای ساخت محصول نهایی باشد و هنگام فراخوانی عملیات آن‌ها را ارسال کند، این الگو نقض می‌شود. ولی اگر کارگردان صرفا دستور دهنده و واسط باشد که فقط ترتیب عملیات را مشخص می‌کند، در آن صورت زیرکلاس‌های سازنده، خود حاوی اطلاعات و رفتار لازم برای تولید محصول نهایی هستند و در این صورت این الگو نقض نمی‌شود.

Command: در این الگو، عملیات درکنار داده خود قرار دارد. Invoker فرمان آغاز را به کامند می‌دهد، کامند می‌داند که Receiver خود کیست و عملیات لازم را فراخوانی می‌کند و Receiver نیز حاوی داده و رفتار مخصوص به خود است. پس این الگو محقق شده است.

Event Logging: در این الگو بنظر می‌رسد داده کنار رفتارش قرار گرفته است. دیوایس حاوی داده و رفتار لازم برای پایش و نمونه‌گیری از EventRemembered مربوطه است و EventRemembered ها نیز حاوی داده و رفتار مربوط به خود هستند.

مقایسه: الگوها در قسمت‌های بیان شده این الگو ر محقق کرده و در قسمتی که شرح داده شده است نقض می‌کنند.

۲-۱۹-۲ Creator

Builder: در این الگو، اگر به زیرکلاس‌های سازنده بنگریم، این زیرکلاس‌ها مسئول ساخت محصول نهایی هستند که در نهایت با متد getResult محصول را در اختیار مشتری قرار می‌دهند. این کلاس بیش‌ترین اطلاع لازم برای ساخت محصول را دارد بنابراین در این قسمت توصیه Creator رعایت شده است. اما نمونه‌گیری از کارگردان و سازنده توسط مشتری انجام می‌شود و مشتری یک نمونه سازنده به کارگردان ارسال می‌کند. در حالی که استفاده کننده اصلی سازنده، کارگردان است و طبق اولویت ۴ در توصیه Creator این وظیفه باید به

کارگردان محول می‌شد که این مورد نقض شده است.

Command: در این الگو، Invoker با کامند رابطه association داشته و از سرویس‌های آن استفاده می‌کند. همچنین کامند نیز با Receiver رابطه association داشته و عملیات آن را فراخوانی می‌کند. اما ساخت کامند و Receiver به ترتیب با Invoker و کامند نیست و این وظیفه بر عهده پیکربند ثالث که همان مشتری است سپرده شده است که اولویت پایین‌تری دارد.

Event Logging: در این الگو، وظیفه نمونه‌گیری از EventRemembered بر عهده Device است که داده‌های لازم برای مقداردهی اولیه را دارا می‌باشد. در ساختار الگو مشخص نشده است که وظیفه نمونه‌گیری Device با کدام عنصر است ولی به هر حال یک عنصر نیاز است تا آن را نمونه‌گیری کرده و متد پایش آن را فراخوانی کند. اولویت‌ها نقض نشده‌اند پس این الگو محقق شده است.

مقایسه: بجز الگوی سوم، در بقیه نقض شده است.

۳-۱۹-۲ Low Coupling

Builder: در این الگو، دید یک‌سویه از سمت کارگردان به واسط Builder وجود دارد. در نتیجه، اولا دید معکوس و جفت‌شدگی زائد از سمت سازنده به کارگردان وجود ندارد و دوما کارگردان از تغییرات زیرکلاس‌های Builder منتزع است و تغییرات آن‌ها به کارگردان منتشر نمی‌شود و در این مجموعه، جفت‌شدگی مطلوب است. همچنین کارگردان هیچ دیدی نسبت به محصولات ندارد. اما در سمت مشتری که پیکربند نیز هست، مشتری باید تمام زیرکلاس‌های Builder را بشناسد و از آن‌ها نمونه‌گیری می‌کند و این به معنی جفت‌شدگی بالا از سمت مشتری به مجموعه است. ولی در سمت کارگردان، تا زمانی که واسط Builder تغییر نکند، کارگردان نیز تغییر نمی‌کند. مشتری با فراخوانی getResult روی زیرکلاس Builder، محصول را دریافت می‌کند.

Command: در این الگو، رابطه بین Invoker و Command از طریق واسط است و DIP برقرار شده است. در نتیجه زیرکلاس‌های کامند می‌توانند تغییر کنند و Invoker از آن بی‌تاثیر است. همچنین در واقع کامند به عنوان یک موجودیت میانی بین Invoker و Receiver نشسته است و یک سطح indirection ایجاد می‌کند و Invoker به Receiver دید مستقیم ندارد. در نتیجه در این بخش‌ها جفت‌شدگی مطلوب است. البته مشتری که نقش پیکربند را نیز ایفا می‌کند، به Invoker، ConcreteCommand و Receiver دید دارد و باید آن‌ها را بشناسد هرچند که این دید نیازی به مانا بودن ندارد ولی به دلیل دید مستقیم، در این قسمت جفت‌شدگی مطلوب نیست. همچنین ConcreteCommand و Receiver نیز دید مستقیم دارند و انتزاع وجود ندارد.

Event Logging: در این الگو، همان‌طور که در شکل ساختار آن مشخص است، Device ها به EventRemembered ها دید مستقیم دارند و در زمان اجرا، اشیایی از EventRemembered نمونه‌گیری

می‌کنند. بنابراین جفت‌شدگی میان این دو موجودیت بالا است. از واسط، انتزاع و در کل روش‌هایی برای کاهش جفت‌شدگی استفاده نشده است.

مقایسه: بجز الگوی سوم، بقیه از روش‌هایی که شرح داده شده است برای تحقق این الگو استفاده کرده‌اند.

۴-۱۹-۲ High Cohesion

Builder: در این الگو، کارگردان و سازنده‌ها از انسجام خوبی برخوردار هستند و عملیات بی‌ربط در کلاس‌ها دیده نمی‌شود. مسولیت اعمال الگوریتم کلی ساخت در کارگردان تعریف شده است و پیاده‌سازی عملیاتی که در واسط سازنده تعریف شده‌اند، بر عهده زیرکلاس‌های سازنده است که جزئیات ساخت هر محصول را پیاده‌سازی می‌کنند. درواقع این‌ها اشیا متخصص آفرینش هستند که این ساختار به افزایش انسجام کمک می‌کند. مشتری نیز از تعریف الگوریتم ساخت و همچنین جزئیات ساخت هر محصول منتزع شده و این عملیات در کلاس‌های خود تعریف شده‌اند که نشانه انسجام مناسب در این الگو است.

Command: در این الگو، Command یک شی متخصص است که به درخواست یا پیغام، هویت داده است و همچنین در سایر کلاس‌ها نیز عملیات نامرتبط به هم انجام نمی‌پذیرد و موجودیت‌ها انسجام مطلوب دارند.

Event Logging: در این الگو، کلاس‌هایی به نام Device تعریف می‌شوند که وظیفه‌شان پایش یک دستگاه بیرونی است. این‌ها باید تشخیص دهند که یک رویداد خارجی رخ داده و سپس زنجیره‌ای از تعاملات را برای پاسخ‌دهی آغاز کنند. برای هر رویداد، یک شی از کلاسی به نام EventRemembered ساخته می‌شود که نماینده‌ی آن رویداد است. این شی شامل اتریبیوت‌هایی است که اطلاعات مربوط به رویداد در آن‌ها ذخیره می‌شود. این اشیا نه تنها داده‌های مربوط به رویداد را در خود نگه می‌دارند، بلکه ممکن است خودشان بتوانند رفتارهایی از خود نشان دهند. به‌عنوان مثال، می‌توانند گزارش تولید کنند، یا حتی خودشان عملیات مربوط به آن رویداد را آغاز نمایند. یعنی بسته به محتوای اتریبیوت‌هایشان، می‌توانند تشخیص دهند که چه پاسخی باید داده شود و اجرای بخشی از این پاسخ را نیز بر عهده بگیرند. در نتیجه، به این شکل انسجام افزایش یافته است و جداسازی دغدغه‌ها دیده می‌شود. اما باید توجه کرد که اشیای EventRemembered ممکن است به مرور زمان بزرگ‌تر و پیچیده‌تر شوند و باید مواظب از دست رفتن انسجام آن با اضافه شدن داده و رفتار نامرتبط و حرکت به سمت God Class جلوگیری کرد. همینطور برای Device نیز به همین شکل باید مواظب انسجام آن بود.

مقایسه: الگوها از روش‌هایی که شرح داده شده است استفاده کرده‌اند برای افزایش انسجام.

Controller ۵-۱۹-۲

Builder: در این الگو، کارگردان صرفا مسئول فراخوانی عملیات واسط سازنده و اجرای الگوریتم کلی ساخت است که در نهایت محصول نهایی توسط متد getResult دریافت می‌شود. بنابراین کارچرخانی و آغاز زنجیره تعاملی به منظور ارائه سرویس به رویداد بیرونی دیده نمی‌شود بنابراین کنترلر وجود ندارد. البته اگر رویداد بیرونی و درخواست سرویس همان ساخت محصول نهایی باشد، بنابراین کارگردان با دریافت درخواست از مشتری، شروع به اجرای فرایند ساخت و فراخوانی عملیات سازنده می‌کند و شاید بتوان این را همانند کنترلر مورد–کاربرد دانست ولی نه دقیق.

Command: در این الگو، آغاز کننده زنجیره تعاملی وجود ندارد پس از این الگو استفاده نشده است.

Event Logging: در این الگو، اساسا یکی از هدف‌ها پایش یک دستگاه بیرونی، دریافت رویدادها، ثبت و آغاز زنجیره‌های تعاملی است که قبلا مفصل‌تر شرح داده شد. پس این الگو محقق شده است.

مقایسه: در الگوی سوم محقق می‌شود ولی در الگوی اول و دوم نه که توضیحات مفصل‌تر ارائه شده است.

Polymorphism ۶-۱۹-۲

Builder: در این الگو، با استفاده از توارث در سازنده‌ها، امکان تعریف انواع سازنده که در نهایت محصولات مختلفی تولید می‌کنند وجود دارد. ارتباط کارگردان با سازنده نیز از طریق واسط است و تغییرات زیرکلاس‌ها به کارگردان منعکس نمی‌شود و حتی در زمان اجرا می‌توان سازنده جدیدی به کارگردان تخصیص کرد. پس این الگو رعایت شده است.

Command: در این الگو، Invoker با واسط کامند ارتباط دارد و ساختار توارثی کامند این اجازه را می‌دهد که کامندهای مختلف با تعریف زیرکلاس‌های جدید تعریف کرد که بسته به نوع در نهایت عملیات متفاوتی اجرا می‌کنند. پس این الگو محقق شده است.

Event Logging: در این الگو مکانیزمی برای برقرار چندریختی استفاده نشده است و این الگو محقق نمی‌شود.

مقایسه: بجز الگوی سوم، بقیه از روش‌هایی که شرح داده شده است استفاده کرده‌اند تا این الگو محقق شود.

Indirection ۷-۱۹-۲

Builder: در حالت قبل از اعمال الگو، یعنی قبل از تعریف سازنده، مشتری یا کارگردان خود مسئولیت ساخت شی را برعهده داشتند. اما با تعریف سازنده، مشتری دستور آغاز را به کارگردان می‌دهد و کارگردان نیز به کمک سازنده‌ای که به آن تخصیص داده شده است، فرایند ساخت را اجرا می‌کند و در نهایت مشتری محصول نهایی را با فراخوانی getResult دریافت می‌کند. بنابراین این الگو به منظور افزایش انعطاف‌پذیری دیده می‌شود.

Command: در این الگو، کامند یک سطح indirection بین Invoker و Receiver ایجاد کرده است و آن‌ها مستقیما به هم دید ندارند. بنابراین این الگو استفاده شده است. البته دید بین کامند و گیرنده مستقیم است.

Event Logging: در این الگو، شی Device رویدادهای بیرونی را پایش کرده و در صورت نیاز، از EventRemembered نمونه‌گیری کرده و زنجیره تعاملی آغاز می‌شود. درست است که دیوایس بین دو شی موجود قرار نگرفته است ولی با دریافت رویداد، پردازش را خودش انجام نداده و یک شی رویداد ساخته و زنجیره آغاز می‌شود پس بین دستگاه بیرونی و شی رویداد که خودش ساخته قرار می‌گیرد. پس می‌توان گفت تا حدی این الگو محقق شده است.

مقایسه: در دو الگوی اول و تا حدی الگوی سوم دیده می‌شود.

Pure Fabrication ۸-۱۹-۲

Builder: در این الگو، کارگردان و همچنین سازنده‌ها ناشی از تحلیل قلمرو مسئله نیستند و اشیا جعلی محسوب می‌شوند که برای افزایش انعطاف‌پذیری و انسجام تعریف شده‌اند. بنابراین این الگو دیده می‌شود.

Command: در این الگو، شی Command ناشی از تحلیل قلمرو مسئله نبوده و قبلا که پیام به صورت آنی انتقال می‌یافت اکنون در کامند هویت مستقل یافته است تا اهدف این الگو محقق شود و این شی جعلی است. پس این الگو استفاده شده است.

Event Logging: در این الگو، EventRemembered ناشی از تحلیل قلمرو مسئله نیست و بنابراین شی جعلی محسوب می‌شود. پس این الگو محقق می‌شود.

مقایسه: هر سه محقق می‌کنند.

Protected Variations ۹-۱۹-۲

Builder: کارگردان با واسط سازنده رابطه دارد، بنابرین در این بخش DIP رعایت شده است و این باعث می‌شود تغییرات زیرکلاس‌های سازنده به کارگردان منتشر نشوند و کارگردان صرفا به واسط وابسته است و تا زمانی که واسط تغییر نکند، کارگردان نیز تغییر نمی‌کند. اما مشتری به مجموعه دید دارد و با تغییر کارگردان یا زیرکلاس‌های سازنده، مشتری نیز بالقوه تغییر می‌کند. همچنین تغییر واسط سازنده مانند اضافه کردن یک متد جدید، باعث می‌شود تمام زیرکلاس‌هایش تغییر کنند و آن متد را پیاده‌سازی کنند که این مصداق تغییر منتشر شونده است.

Command: ارتباط بین Invoker و Receiver به واسطه کامند غیر مستقیم شده است و تغییرات آن‌ها به یکدیگر منتشر نمی‌شود. همچنین ارتباط Invoker نیز با واسط کامند است و تغییرات زیرکلاس‌های کامند تاثیری بر Invoker نمی‌گذارد که در این بخش‌ها این الگو محقق شده است. اما دید بین زیرکلاس‌های کامند و گیرنده به صورت مستقیم است و تغییرات گیرنده به کامند منتشر می‌شود. همچنین مشتری نیز که نقش پیکربند را ایفا می‌کند، به زیرکلاس‌های کامند دید مستقیم دارد.

Event Logging: در این الگو، Device به EventRemembered دید دارد به منظور نمونه‌گیری و آغاز زنجیره تعاملی که این دید در زمان کامپایل محقق می‌شود که صلب است. از مکانیزم‌های کنترل جفت‌شدگی و انتزاع استفاده نشده و این باعث می‌شود تغییرات EventRemembered به Device منتشر شود.

مقایسه: بجز الگوی سوم، بقیه از روش‌هایی که شرح داده شده است برای تحقق این الگو استفاده می‌کنند.

فصل ۳

ارزیابی الگوهای Bridge، Iterator و State

در این فصل، سه الگوی Bridge، Iterator و State مبتنی بر معیارهایی که در فصل یک معرفی شدند، تحلیل، ارزیابی و مقایسه می‌شوند.

۳-۱ دسته

Bridge: این الگو در دسته الگوهای طراحی ساختاری[۱] قرار دارد. این دسته از الگوها بیش‌تر به ساختاردهی کلاس‌ها و سیستم‌ها توجه دارند با هدف کم کردن جفت‌شدگی[۲]. این کار نوعا به شکل جدا کردن انتزاع یا واسط از پیاده‌سازی انجام می‌شود. یعنی وجه انتزاعی و specification از بخش پیاده‌سازی کننده جدا می‌شود که به این شکل امکان رشد مستقل این دو فراهم می‌شود. در این الگوها، ساختار وجه اساسی است.

Iterator: این الگو در دسته الگوهای رفتاری[۳] قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام[۴] و انعطاف‌پذیری.

State: این الگو در دسته الگوهای رفتاری قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی

[۱] structural
[۲] coupling
[۳] behavioral
[۴] cohesion

رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام و انعطاف‌پذیری.

مقایسه: الگوی اول در دسته ساختاری و دو الگوی دیگر در دسته رفتاری قرار دارند که دغدغه هر کدام شرح داده شد.

۳-۲ هدف

Bridge: این یک الگوی پایه است و در آن انتزاع[۵] از پیاده‌سازی جدا می‌شود. یعنی تعریف کلی یک کلاس، از پیاده‌سازی جدا می‌شود با این هدف که بشود آن‌ها را به صورت مستقل تغییر داد، رشد داد و extend کرد. همچنین در زمان اجرا نیز بشود پیاده‌سازی یک کلاس را جایگزین کرد.

Iterator: این الگو، راهکاری ارائه می‌دهد برای پیمایش عناصر یک شی مرکب (مانند لیست، آرایه، درخت و غیره) بدون آنکه ساختار داخلی آن شی فاش شود. هدف اصلی این الگو جداسازی مسئولیت پیمایش از خود شی مرکب است، تا هم کپسوله‌سازی حفظ شود و هم بتوان چندین پیمایش همزمان انجام داد. اگر خود شی مرکب مسئول پیمایش باشد، لازم است وضعیت پیمایش (مثل موقعیت فعلی) را در خود نگه دارد و در نتیجه تنها یک پیمایش همزمان ممکن خواهد بود. این کار، شی را سنگین می‌کند و گسترش‌پذیری را محدود می‌سازد. از طرف دیگر، اگر پیمایش را مستقیماً به مشتری واگذار کنیم، برای انجام آن باید به ساختار داخلی شی دسترسی داشته باشد و این نقض آشکار اصل کپسوله‌سازی خواهد بود. راه‌حل این الگو این است که شی مرکب خودش یک شی پیمایش‌گر[۶] تولید کند و آن را به مشتری بدهد. این پیمایش‌گر است که مسئول پیمایش می‌شود و از طریق متدهایی مثل first، next، isDone، currentItem مشتری را قادر به کنترل فرایند پیمایش می‌کند. مزیت مهم این الگو آن است که پیمایش همزمان چندگانه ممکن می‌شود و شی مرکب نیز سبک باقی می‌ماند. اما هزینه‌ای که پرداخت می‌شود این است که پیمایش‌گر باید به ساختار داخلی شی مرکب دسترسی داشته باشد و این باعث شکل‌گیری جفت‌شدگی بالا بین این دو می‌شود و کپسوله‌سازی نیز نقض می‌شود. این جفت‌شدگی نسبت به نقض کپسوله‌سازی از سمت مشتری قابل قبول‌تر تلقی می‌شود و هزینه‌ای است که می‌پردازیم.

State: در این الگو هدف آن است که یک شی بتواند بسته به حالت درونی‌اش رفتار متفاوتی از خود نشان دهد و هرگاه این حالت تغییر کند، رفتار شی نیز به‌صورت خودکار تغییر یابد. گویی شی کلاس خود را تغییر داده است. به عبارت دیگر، با تغییر وضعیت داخلی، رفتار آن کاملا دگرگون می‌شود، انگار که نمونه‌ای از یک کلاس

[۵] abstraction
[۶] iterator

متفاوت شده باشد. اما در واقع، آنچه رخ می‌دهد این است که شی اصلی دارای اشیا داخلی است که تمامی رفتارهای وابسته به حالت را در بر دارند، و با جایگزینی این اشیا داخلی، رفتار نیز تغییر می‌کند. برای تحقق این هدف، لازم است شی اصلی، نمونه‌هایی از اشیا وضعیت‌های مختلف را در خود نگه دارد و هنگام تغییر وضعیت، شی مربوطه را جایگزین کند. رفتارهایی که به وضعیت وابسته‌اند، توسط شی اصلی به این اشیا داخلی واگذار[۷] می‌شوند. از این رو، با هر بار تغییر وضعیت، رفتار متناظر نیز به صورت خودکار تغییر خواهد یافت. بدین ترتیب، کلیه‌ی رفتارهای مبتنی بر وضعیت، به این اشیای درونی محول می‌گردند. گرچه برخی وضعیت‌ها ممکن است پس از تعیین شدن، به‌ندرت تغییر کنند، اما این الگو عمدتا زمانی به‌کار می‌رود که وضعیت پویا باشد و انتظار رود هر بار که شی داخلی تغییر می‌یابد، رفتار نیز متناسب با آن دگرگون شود. از این طریق، اطمینان حاصل می‌شود که همواره رفتاری متناسب با وضعیت فعلی اجرا خواهد شد.

مقایسه: اهداف هر الگو به صورت مفصل در بخش‌های مرتبط شرح داده شده و ارتباط آن‌ها نیز در قسمت الگوهای مرتبط بیان شده است.

۳-۳ حوزه

Bridge: این الگو در حوزه شی[۸] قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

Iterator: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

State: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

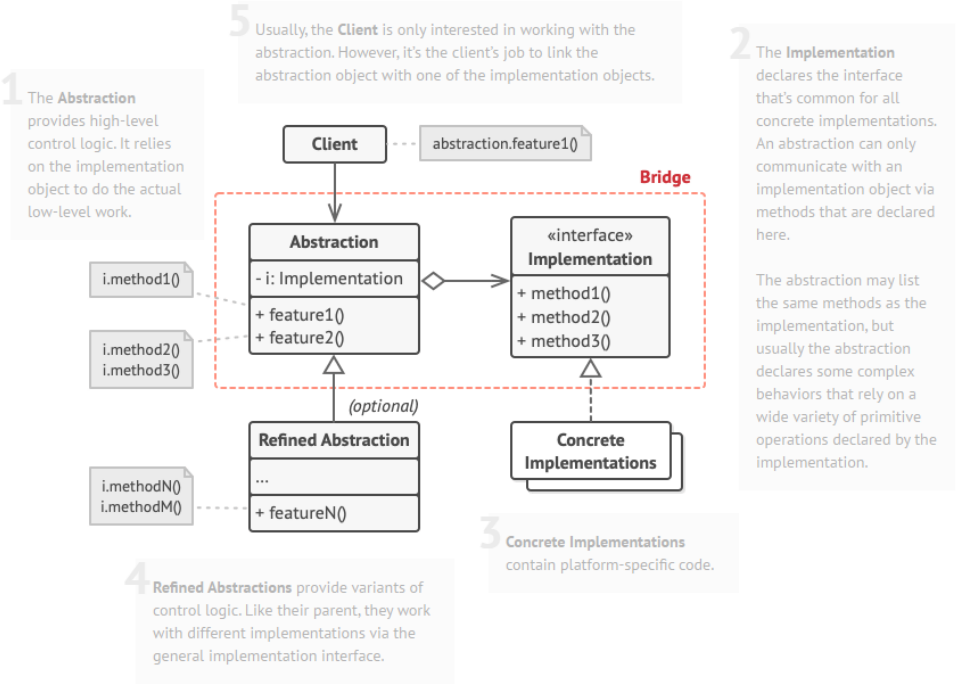
مقایسه: هر سه در حوزه شی قرار دارند.

---

[۷] delegate
[۸] object scope

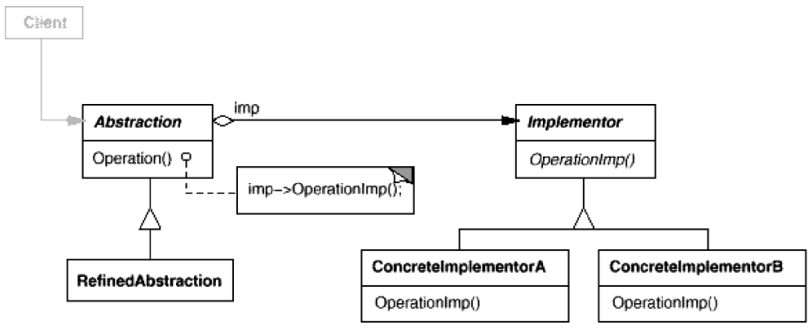
۳-۴ ساختار و رفتار

Bridge: ساختار این الگو در شکل ۳-۱ و ۳-۲ قابل مشاهده است. در زمان اجرا یک نمونه از Abstraction در اختیار مشتری قرار می‌گیرد و به آن نمونه از Abstraction هم باید یک نمونه از Implementor منتسب شود. در این قسمت نیازی به رابطه aggregation نیست و رابطه association ساده است. در زمان اجرا سه شی پشت سر هم قرار می‌گیرند: نمونه‌های مشتری، Abstraction و Implementor و بدین شکل الگو محقق می‌شود. یعنی در عمل باید ارتباط بین Abstraction و Implementor برقرار شود به عبارتی پل زده شود تا این الگو محقق شود. در این ساختار خود مشتری پیکربندی را انجام نمی‌دهد و یک پیکربند ثالث وجود دارد که یک نمونه از Implementor را در اختیار Abstraction قرار می‌دهد. این پیکربند تمام زیرکلاس‌ها را می‌شناسد و پیوندهای delegation را برقرار می‌کند. مشتری نوعا فقط به Abstraction دید دارد. متدهای تعریف شده در Implementor نوعا نسبت به Abstraction ریزدانه‌تر می‌شوند تا بتوان تنوع بیش‌تری برای فراخوانی در سمت Abstraction داشت.



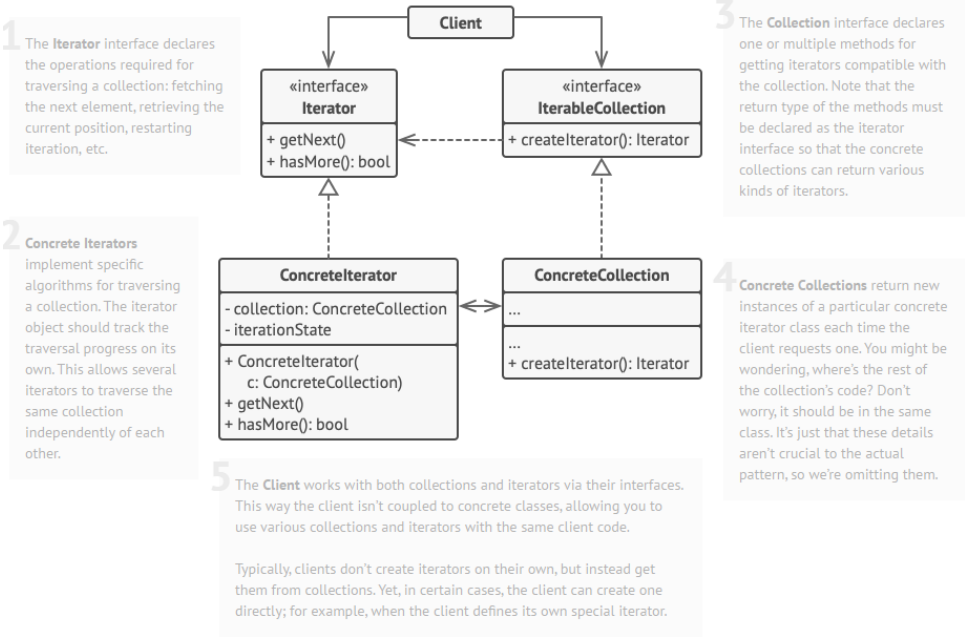
شکل ۳-۱: ساختار الگوی Bridge [۱]

Iterator: ساختار این الگو در شکل ۳-۳ و ۳-۴ قابل مشاهده است. در نمودار کلاس این الگو، مشتری واسط Aggregate و Iterator را می‌شناسد. واسط Aggregate شامل عملیاتی نظیر CreateIterator است که وظیفه‌ی تولید یک نمونه از Iterator را بر عهده دارد. این سازوکار دقیقا با الگوی Factory Method منطبق است، چرا که کلاس‌های ConcreteAggregate مسئول تعیین نوع دقیق Iterator متناظر خود هستند. هنگامی که یک نمونه از ConcreteIterator ساخته می‌شود، شی ConcreteAggregate خود را (از طریق



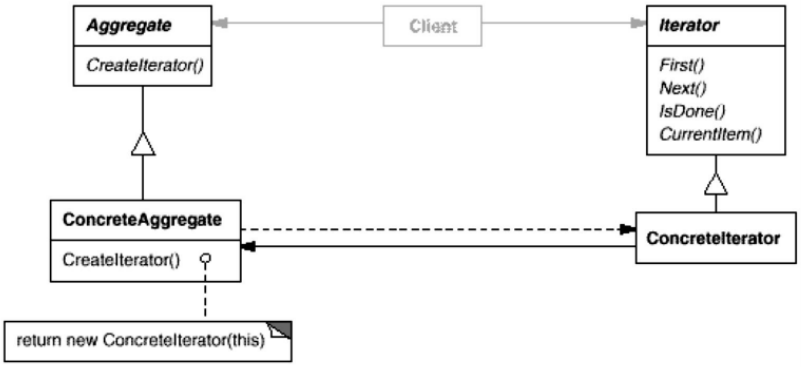
شکل ۳-۲: ساختار الگوی Bridge در کتاب GoF [۲]

this) به آن پاس می‌دهد. این کار موجب می‌شود که پیمایش‌گر به ساختار داخلی aggregate مربوط به خودش دسترسی داشته باشد، آن را مانا کند و بتواند عملیات پیمایش را انجام دهد.



شکل ۳-۳: ساختار الگوی Iterator [۱]

State: ساختار این الگو در شکل ۳-۵ و ۳-۶ قابل مشاهده است. در این الگو، به شی اصلی اصطلاحا Context گفته می‌شود و درون آن نمونه‌هایی از اشیا وضعیت‌ها قرار می‌گیرند. در حالت کلی ممکن است وضعیت‌های گوناگونی مطرح باشد که در آن صورت، برای هرکدام از آن‌ها ساختار توارثی تعریف می‌گردد. در زمان اجرا، برای هر وضعیت، یک نمونه از زیرکلاس مرتبط با وضعیت درون Context قرار می‌گیرد. شایان ذکر است که این «درون قرار گرفتن» لزوما به معنای وجود رابطه‌ی کل به جز (composition یا aggregation) نیست. برخلاف آنچه در نمودارها به‌صورت aggregation ترسیم می‌شود، در اینجا ضرورتی به استفاده از این نوع رابطه وجود ندارد. آنچه واقعا مدنظر است، رابطه‌ای از نوع association است. در واقع، الزامی نیست که وضعیت‌ها به‌صورت اختصاصی متعلق به یک Context باشند. امروزه می‌دانیم که در صورتی که اشیای

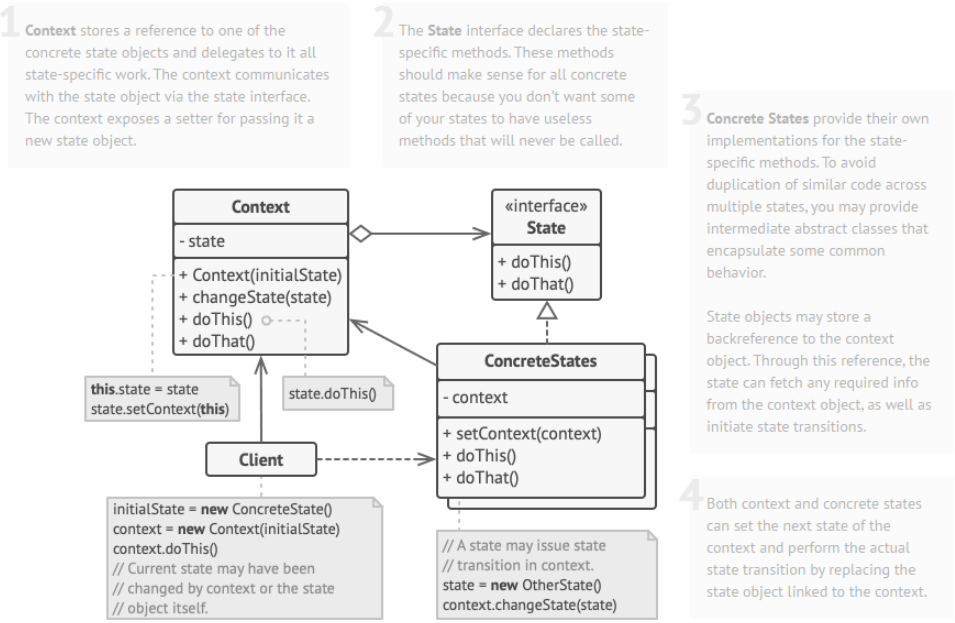


شکل ۳-۴: ساختار الگوی Iterator در کتاب GoF [۲]

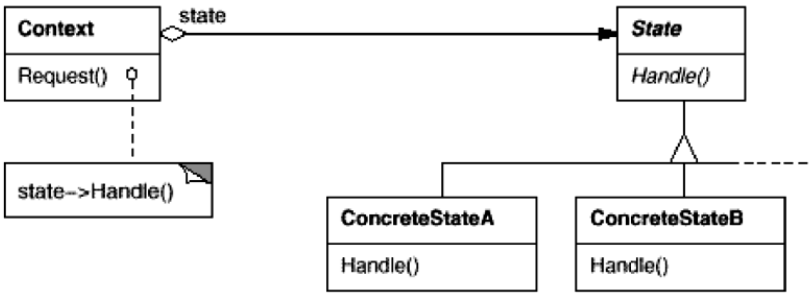
وضعیت به صورت stateless باشند و صرفا به‌عنوان ارائه‌دهنده‌ی سرویس عمل کنند، می‌توانند به‌صورت مشترک بین چندین Context مورد استفاده قرار گیرند. در این حالت، دیگر نیازی نیست این اشیا درون Context قرار بگیرند. در زمان اجرا، یکی از زیرکلاس‌های وضعیت به Context داده می‌شود و Context با یک وضعیت اولیه مقداردهی می‌شود. این مقداردهی اولیه معمولاً توسط مشتری (یعنی یک شی بیرونی) انجام می‌گیرد که وظیفه‌ی پیکربندی اولیه را برعهده دارد. پرسشی اساسی در این الگو آن است که تغییر وضعیت [۹] چگونه و توسط چه عنصری باید تشخیص و اعمال شود. به عبارت دیگر، چه موجودیتی مسئول اجرای state transition diagram خواهد بود؟ ممکن است در نگاه نخست چنین تصور شود که خود Context می‌تواند مسئول این تغییر باشد. بدین معنا که Context وقوع یک تغییر وضعیت را تشخیص داده، سپس وضعیت داخلی خود را با نمونه‌ای از وضعیت جدید جایگزین نماید. اما اگر این مسئولیت به Context واگذار شود، پیامد آن افزایش جفت‌شدگی است. زیرا Context برای اجرای چنین تغییری باید اطلاعات دقیقی درباره‌ی زیرکلاس‌های وضعیت داشته باشد. یعنی باید بداند که در وضعیت فعلی و با وقوع رویدادی خاص، گذار به کدام وضعیت باید صورت گیرد و همچنین باید از کلاس مقصد آگاه باشد تا بتواند نمونه‌ای از آن را ایجاد و جایگزین نماید. این وابستگی، Context را از یک سرویس‌گیرنده رفتارهای وابسته به وضعیت، به یک کنترل‌کننده تبدیل می‌کند که تمام جزئیات پیاده‌سازی و ساختار وضعیت‌ها را می‌شناسد. این امر، اصل DIP را نقض می‌کند. مطابق با DIP، Context باید به واسط وضعیت وابسته باشد و از جزییات زیرکلاس‌های خاص هیچ اطلاعی نداشته باشد. Context صرفاً رفتار وابسته به وضعیت را به شی وضعیت واگذار می‌کند و هیچ نقشی در منطق انتقال وضعیت ندارد. راهکار مناسب آن است که خود اشیای عینی وضعیت، منطق انتقال را در اختیار داشته باشند و هنگام لزوم، Context را وادار به جایگزینی وضعیت نمایند. به این ترتیب، وابستگی‌ها کنترل‌شده باقی می‌مانند و انسجام طراحی حفظ می‌شود. راه‌حل مناسب برای حفظ اصول طراحی و جلوگیری از افزایش جفت‌شدگی آن است که الگوریتم تغییر وضعیت به‌جای آن‌که در Context متمرکز باشد، میان زیرکلاس‌های

[۹] state transition

وضعیت توزیع گردد. بدین معنا که هر وضعیت خود منطق گذار به وضعیت بعدی را درون خود نگه می‌دارد. یعنی تشخیص می‌دهد که آیا باید وضعیت تغییر کند یا نه، و اگر تغییر لازم باشد، مسئولیت جایگزینی وضعیت جدید را نیز برعهده می‌گیرد. در واقع، وضعیت فعلی با ایجاد نمونه‌ای از وضعیت بعدی، خود را با آن جایگزین می‌کند و به این ترتیب، Context را بازپیکربندی می‌نماید. برای آن‌که زیرکلاس‌های وضعیت بتوانند چنین تغییری در Context اعمال کنند، نیاز است که به Context دید داشته باشند. این دید معکوس را می‌توان هنگام اجرای عملیات فراهم کرد، بدین صورت که Context در هنگام فراخوانی متد handle، خود را از طریق this به‌عنوان پارامتر به State ارسال می‌کند. این روش که بسیار رایج است، دسترسی موقتی و کنترل‌شده‌ای از State به Context ایجاد می‌نماید. این دید معکوس تنها در محدوده‌ی اجرای عملیات جاری معتبر است و وابستگی دائمی ایجاد نمی‌کند، بنابراین مشکلی از نظر اصول طراحی به‌وجود نخواهد آمد. برای این منظور، باید یک واسط در Context تعریف شود که از طریق آن، وضعیت‌ها بتوانند وضعیت داخلی Context را تغییر دهند. بدون آن‌که به جزییات پیاده‌سازی Context وابسته باشند.



شکل ۳-۵: ساختار الگوی State [۱]



شکل ۳-۶: ساختار الگوی State در کتاب GoF [۲]

۳-۵ جفت‌شدگی

Bridge: در این الگو، مشتری با کلاس انتزاعی Abstraction ارتباط دارد و دید مستقیم به پیاده‌سازی کننده‌ها و همچنین زیرکلاس‌های Abstraction ندارد. Abstraction نیز با واسط پیاده‌سازی کننده در ارتباط است و DIP برقرار بوده و تغییرات زیرکلاس‌های پیاده‌سازی کننده به آن منتشر نمی‌شود. پیاده‌سازی کننده نیز دید به Abstraction یا مشتری ندارد و ارتباط بین Abstraction و پیاده‌سازی کننده یک‌سویه است. زیرکلاس‌های پیاده‌سازی کننده و انتزاع می‌توانند به صورت مستقل رشد کنند. در نتیجه در این مجموعه جفت‌شدگی مطلوب است. ولی پیکربند ثالث باید به زیرکلاس‌ها دید مستقیم داشته باشد تا پیکربندی را انجام دهد و این به معنی جفت‌شدگی بالا است. این ساختار مشکلات حالت قبل از اعمال الگو نظیر مشکلات ساختار توارثی و زیرکلاس‌های متعدد و ترکیبی و غیره را ندارد.

Iterator: در این الگو، مشتری از طریق واسط با Aggregate و پیمایش‌گر ارتباط دارد. بنابراین DIP برقرار است و تغییرات زیرکلاس‌های Aggregate و پیمایش‌گر تاثیری روی مشتری نمی‌گذارد. همچنین مشتری دید به جزئیات Aggregate ندارد و خود عمل پیمایش را انجام نمی‌دهد بلکه صرفا با واسط پیمایش‌گر کار می‌کند. پس در این قسمت جفت‌شدگی مطلوب است. اما به این دلیل که زیرکلاس‌های Aggregate باید خودشان پیمایش‌گر خود را مشخص کنند، دید مستقیم بین زیرکلاس‌هایشان وجود داشته، DIP نقض شده است و جفت‌شدگی در این قسمت وجود دارد. پیمایش‌گرها نیز دید مانا به Aggregate مربوط به خود دارند تا عملیات پیمایش را انجام دهند و این به معنی جفت‌شدگی بالا در این قسمت است.

State: در این الگو، کانتکست ارتباط و دید مستقیم به زیرکلاس‌های وضعیت ندارد و این ارتباط از طریق واسط است. بنابراین در این قسمت اصل DIP رعایت شده است و کانتکست از تغییرات زیرکلاس‌های وضعیت منتزع شده است. الگوریتم تغییر وضعیت نیز در دوش کانتکست نبوده و این الگوریتم به اشیا وضعیت توزیع شده است. بدین جهت یک دید از سمت زیرکلاس‌های وضعیت به کانتکست به منظور انتقال وضعیت، وجود دارد اما این دید لزوما مانا نیست و کنترل شده است. مشتری نیز پیکربند اولیه بوده و یک شی وضعیت به عنوان وضعیت اولیه نمونه‌گیری کرده و به عنوان پارامتر هنگام نمونه‌گیری از کانتکست به آن تخصیص می‌دهد. مشتری دید مستقیم به کانتکست دارد و دید مشتری به زیرکلاس‌های وضعیت از نوع وابستگی و غیرمانا است ولی به آن‌ها دید مستقیم دارد. در کل جفت‌شدگی در این الگو در قسمت کانتکست مطلوب است.

مقایسه: هر سه الگو از روش‌هایی در قسمت‌هایی که شرح داده شده است استفاده کرده‌اند تا جفت‌شدگی را کنترل کنند.

۳-۶ انسجام

Bridge: در این الگو، وجه انتزاعی از وجه پیاده‌سازی جدا شده است و مسئولیت پیاده‌سازی به متخصص پیاده‌سازی سپرده شده است که به این شکل به جداسازی دغدغه‌ها توجه شده و انسجام موجودیت‌ها افزایش یافته است. در موجودیت‌ها عملیات نامرتبط وجود ندارد و انسجام در سطح مطلوبی است.

Iterator: در این الگو، شی متخصص پیمایش تعریف شده است و عملیات مربوط به پیمایش از دوش مشتری یا Aggregate برداشته شده و اکنون پیمایش‌گر این نقش را ایفا می‌کند و این کمک به جداسازی دغدغه‌ها و افزایش انسجام کرده است. مشتری صرفا با واسط‌ها کار کرده، Aggregate عملیات مربوط به شی مرکب را انجام داده و پیمایش‌گر نیز وظیفه پیمایش را انجام می‌دهد.

State: در این الگو، رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود و آن شی است که پیاده‌سازی کننده رفتار بوده و همچنین انتقال حالت را انجام می‌دهد. این بار از دوش کانتکست برداشته شده است و کانتکست از پیاده‌سازی‌های حالت‌ها، switch statement های پیچیده و انتقال حالت رها شده است و چندکارگی از بین رفته است. در نتیجه اعمال این الگو، انسجام افزایش یافته و مطلوب شده است.

مقایسه: هر سه برای کنترل اسنجام با روش‌ها و در قسمت‌های شرح داده شده استفاده کرده‌اند.

۳-۷ پیکربندی

Bridge: در این الگو، یک پیکربند ثالث غیر از مشتری، وظیفه نمونه‌گیری از زیرکلاس پیاده‌سازی کننده و انتزاع را دارد و سپس پیوندهای delegation را با منتسب نمودن نمونه پیاده‌ساز به انتزاع و نمونه انتزاع به مشتری برقرار می‌کند و الگو بدین شکل در زمان اجرا محقق می‌شود. این پیکربند ثالث درگیری مشتری با پیکربندی را رفع نموده و مشتری فقط کلاس انتزاع را می‌بیند.

Iterator: در این الگو، مشتری به واسط شی مرکب و پیمایش‌گر دید داشته و می‌تواند از طریق متد CreateIterator، یک پیمایش‌گر مخصوص آن شی دریافت کند (در این قسمت الگوی Factory Method دیده می‌شود). شی مرکب خودش پیمایش‌گر مربوطه‌اش را می‌شناسد و از آن نمونه‌گیری کرده و آن را با پاس دادن خودش با this پیکربندی کرده و پیمایش‌گر را تحویل مشتری می‌دهد. الگو بدین شکل در زمان اجرا محقق شده و مشتری می‌تواند از طریق واسط با پیمایش‌گر و شی مرکب کار کند.

State: در این الگو، مشتری در هنگام نمونه‌سازی از کانتکست، یک نمونه از شی حالت به عنوان حالت اولیه ساخته و به عنوان پارامتر به کانتکست ارسال می‌کند و بدین شکل پیکربندی اولیه صورت می‌پذیرد. در

ادامه و در روند اجرا، الگوریتم انتقال وضعیت بین اشیای وضعیت توزیع شده است و هرکدام از آن‌ها می‌دانند که وضعیت‌های بعدی کدام هستند و در شرایط مناسب وضعیت کانتکست را تغییر می‌دهند که این دید از سمت زیرکلاس‌های وضعیت به کانتکست از طریق ارسال this هنگام فراخوانی handle در کانتکست صورت می‌پذیرد.

مقایسه: هر سه نیاز به پیکربندی دارند که برای هرکدام مفصل شرح داده شده است.

۳-۸ انعطاف‌پذیری

Bridge: در نتیجه اعمال این الگو، وجه انتزاع از پیاده‌سازی کننده جدا می‌شود و هرکدام می‌توانند به صورت مستقل رشد کنند. ارتباط بین کلاس انتزاع و پیاده‌سازی کننده از طریق واسط بوده و DIP برقرار است. در نتیجه هیچ تغییری در زیرکلاس‌های پیاده‌سازی کننده به انتزاع منتشر نمی‌شود. پیاده‌سازی کننده نیز به انتزاع دید ندارد و زیرکلاس‌های انتزاع نیز می‌توانند رشد کنند. مشتری نیز صرفا کلاس انتزاع را می‌بیند و از هیچ تغییری در داخل مجموعه تاثیر نمی‌پذیرند. در زمان اجرا می‌توان بدون مشکل پیکربندی و بازپیکربندی کرد. همچنین نوعا عملیات واسط پیاده‌سازی کننده‌ها ریزدانه تعریف می‌شود که تنوع عملیات در انتزاع را افزایش می‌دهد که همه این موارد نشان از انعطاف‌پذیری هستند.

Iterator: پس از اعمال این الگو، مشتری از طریق واسط به شی مرکب و پیمایش‌گر ارتباط دارد در نتیجه از تغییرات زیر آن‌ها منتزع شده است. همچنین دیگر نیازی به داشتن دید به داخل شی مرکب برای انجام عملیات پیمایش ندارد. شی مرکب نیز خود عملیات پیمایش را انجام نمی‌دهد و این کار توسط پیمایش‌گر متخصص انجام می‌شود. این ساختار باعث شده است که بشود پیمایش‌گرهای جدید اضافه کرد یا تغییراتی داد. همچنین امکان انجام چندین پیمایش نیز فراهم شده است. در نتیجه در این قسمت تغییرپذیری و انعطاف‌پذیری افزایش یافته است. البته در این ساختار، زیرکلاس‌های شی مرکب و پیمایش‌گر دید مستقیم به یکدیگر داشته و نقض DIP صورت گرفته است و تغییرات آن‌ها به یکدیگر منتشر می‌شود.

State: انعطاف‌پذیری در اثر استفاده از این الگو بالا رفته است زیرا کانتکست از طریق واسط با وضعیت ارتباط دارد و رفتار وابسته به حالت را به شی حالت خود واسپاری می‌کند و از تغییرات زیرکلاس‌های وضعیت منتزع شده است. بدین شکل امکان تعریف وضعیت‌های جدید وجود دارد و در هنگام اجرا نیز الگوریتم انتقال وضعیت بین اشیای وضعیت توزیع شده است و آن‌ها در شرایط مناسب وضعیت داخلی کانتکست را تغییر می‌دهند. این ساختار، انعطاف‌پذیری بسیار بیش‌تری نسبت به ساختار توارثی دارد. البته مشتری که پیکربند اولیه است، به زیرکلاس‌های وضعیت دید مستقیم دارد که برای حالت اولیه نیاز است.

مقایسه: هر سه الگو از روش‌ها در قسمت‌های بیان شده برای تحقق انعطاف‌پذیری استفاده کرده‌اند.

۳-۹ کارایی

Bridge: پس از اعمال این الگو، وجه انتزاع از پیاده‌سازی کننده جدا شده است و رفتاری که قبلا در یک کلاس تعریف شده بود، اکنون نیازمند تعامل انتزاع با پیاده‌سازی کننده از طریق delegation است و این مورد تعداد اشیا در حافظه و تعداد پیام‌های رد و بدل شده را اندکی افزایش می‌دهد ولی بنظر در مقابل منافع این الگو، این موارد قابل چشم‌پوشی هستند.

Iterator: در نتیجه اعمال این الگو، وظیفه پیمایش شی مرکب به شی جدید پیمایش‌گر محول می‌شود. در نتیجه تعداد اشیا اندکی افزایش می‌یابد. همچنین اطلاعات مورد نیاز پیمایش‌گر برای پیمایش شی مرکب نزد خود نیست و نیاز به ارسال پیام بیش‌تر بوجود می‌آید. در نتیجه کارایی اندکی کاهش یافته است.

State: در این الگو، ممکن است در زمان اجرا تعداد زیادی نمونه از وضعیت‌ها در حافظه نگه‌داری شوند. از آن‌جا که برای هر نمونه از کانتکست، معمولا یک نمونه‌ی وضعیت داخلی نیز وجود دارد، چنانچه تعداد زیادی کانتکست در سیستم فعال باشند، این امر می‌تواند به افزایش چشمگیر تعداد اشیای وضعیت منجر شود. راهکار مؤثر برای مقابله با این مسئله، استفاده از اشتراک‌گذاری است. اگر بتوان اشیای وضعیت را به گونه‌ای طراحی کرد که بی‌حالت[۱۰] باشند، آنگاه می‌توان آن‌ها را میان کانتکست‌های مختلف به اشتراک گذاشت. در این صورت، به‌جای ایجاد چندین نمونه از یک وضعیت مشخص، تنها یک نمونه کافی خواهد بود که به عنوان سرویس‌دهنده‌ی اشتراکی عمل کند. این رویکرد منجر به صرفه‌جویی قابل توجهی در منابع سیستم و بهینه‌سازی حافظه می‌شود. با این حال، الگوی State دارای یک سطح indirection است. بدین معنا که عملیات‌هایی که ممکن بود مستقیما توسط کانتکست انجام شوند، اکنون از طریق واسپاری به شی وضعیت انجام می‌گیرند. این انتقال پیام‌ها در عمل موجب افزایش اندکی در سربار اجرایی سیستم می‌شود و می‌تواند بر کارایی تأثیر منفی بگذارد. با این‌حال، این کاهش عملکرد معمولا در برابر مزایایی که الگوی State فراهم می‌آورد پذیرفته می‌شود. مهم‌ترین این مزایا عبارت‌اند از maintainability بالا به واسطه‌ی جداسازی مسئولیت‌ها و کاهش جفت‌شدگی، حذف منطق‌های پیچیده‌ی شرطی مانند switch-case یا if-else های تو در تو و جلوگیری از ساختارهای توارثی پیچیده برای پوشش انواع حالات رفتاری.

مقایسه: در اثر استفاده از این الگوها کارایی کاهش می‌یابد که توضیحات مفصل ارائه شده است.

---

[۱۰] stateless

۳-۱۰ شی جعلی

Bridge: در اثر اعمال این الگو، پیاده‌سازی کننده‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو بوجود آمده‌اند. کلاس‌های انتزاع نیز اگر ناشی از تحلیل قلمرو مسئله نباشند به همین شکل اشیا جعلی محسوب می‌شوند.

Iterator: در اثر اعمال این الگو، پیمایش‌گرها تعریف می‌شوند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو بوجود آمده‌اند. پس شی جعلی محسوب می‌شوند.

State: در این الگو، کلاس‌های نمایانگر وضعیت‌های مختلف می‌توانند ناشی از تحلیل قلمرو مسئله باشند. مانند وضعیت‌های مختلف دانشجو، مثل وضعیت مشروطی، که در این صورت این اشیا جعلی نیستند. اگر آن اشیا ناشی از تحلیل قلمرو مسئله نباشند و برای تحقق اهداف الگو تعریف شوند، در این صورت جعلی محسوب می‌شوند.

مقایسه: هر سه اشیای جعلی تعریف می‌کنند.

۳-۱۱ انتشار تغییرات

Bridge: در این الگو، مشتری صرفا به کلاس انتزاع دید دارد و از هیچ تغییری در داخل مجموعه تاثیر نمی‌پذیرد و صرفا زمانی که واسط انتزاع تغییر کند، مشتری تاثیر می‌پذیرد. کلاس انتزاع نیز با پیاده‌سازی کننده‌ها از طریق واسط ارتباط دارد و این ارتباط یک‌سویه نیز هست. در نتیجه تغییرات زیرکلاس‌های پیاده‌سازی کننده به انتزاع منتشر نمی‌شود و تغییرات انتزاع نیز در پیاده‌سازی کننده‌ها تاثیری ندارد. فقط در صورت تغییر واسط پیاده‌سازی کننده، کلاس انتزاع نیز مشمول تغییر می‌شود. البته در صورت تغییر واسط در پیاده‌سازی کننده یا انتزاع، زیرکلاس‌ها دچار تغییر می‌شوند و باید آن عملیات را پیاده‌سازی کنند. پیکربند نیز به تمام زیرکلاس‌ها دید مستقیم دارد و تغییرات به آن منتشر می‌شوند.

Iterator: در این الگو، مشتری از طریق واسط با شی مرکب و پیمایش‌گر ارتباط دارد و در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار می‌شود که باعث می‌شود تغییرات زیرکلاس‌های شی مرکب و پیمایش‌گر به مشتری منتشر نشوند. مشتری تنها در صورتی تغییر می‌کند که واسط‌ها تغییر کنند. اما به این دلیل که زیرکلاس‌های شی مرکب و پیمایش‌گر به هم دید مستقیم داشته و ساختارهای توارثی موازی شکل گرفته است، جفت‌شدگی بالا بوده و تغییرات آن‌ها به یکدیگر منتشر می‌شوند.

State: در این الگو، رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود ولی کانتکست از طریق

واسط با وضعیت در ارتباط است و اصل DIP برقرار بوده در نتیجه OCP برقرار شده و تغییرات زیرکلاس‌های وضعیت به کانتکست منتشر نمی‌شود و می‌توان در زمان اجرا وضعیت‌های داخل کانتکست را تغییر داد که این منجر به تغییر رفتار وابسته به حالت می‌شود. البته مشتری که پیکربند اولیه است، به زیرکلاس‌های وضعیت به منظور مقداردهی اولیه به کانتکست، دید مستقیم دارد. زیرکلاس‌های حالت نیز به منظور انجام انتقال حالت، به کانتکست دید موقت پیدا می‌کنند و از واسط مخصوص تغییر حالت درونی کانتکست استفاده می‌کنند و تا زمانی که واسط‌ها تغییر نکنند، تغییر منتشر نمی‌شود.

مقایسه: هر سه الگو از روش‌هایی در قسمت‌های شرح داده شده استفاده می‌کنند برای کنترل انتشار تغییرات.

۱۲-۳ سهولت پیاده‌سازی

Bridge: با استفاده از یک زبان شی‌گرا می‌توان این الگو را بدون مشکل پیاده‌سازی کرد. نیاز به تعریف واسط‌ها یا کلاس‌های انتزاعی داشته و کلاس‌های عینی برای تحقق آن‌ها باید تعریف شوند. در کل پیاده‌سازی مشکل خاصی ندارد.

Iterator: پیاده‌سازی این الگو مشکل خاصی در زبان‌های شی‌گرا نداشته و به آسانی قابل پیاده‌سازی است. در برخی زبان‌ها مانند جاوا نیز واسط پیمایش‌گر تعریف شده است و باید آن را پیاده‌سازی کرد.

State: پیاده‌سازی این الگو در زبان‌های شی‌گرا که واسط یا کلاس‌های انتزاعی را پشتیبانی می‌کنند بسیار آسان بوده و مشکل خاصی وجود ندارد و می‌توان موجودیت‌های این الگو مانند وضعیت‌ها، کانتکست و غیره را تعریف و پیاده‌سازی کرد.

۱۳-۳ موارد کاربرد

Bridge:

- وقتی نمی‌خواهیم یک انقیاد[۱۱] دائمی بین انتزاع و پیاده‌سازی‌اش وجود داشته باشد و هدف داریم پیاده‌سازی در زمان اجرا انتخاب یا تغییر یابد، از این الگو استفاده می‌کنیم.
- وقتی می‌خواهیم انتزاع و پیاده‌سازی به‌صورت مستقل و جداگانه قابل گسترش باشند، از این الگو استفاده می‌کنیم. منظور از قابل گسترش بودن هم این است که بتوان به‌سادگی برای هر کدام زیرکلاس تعریف کرد، بدون اینکه تغییری در بخش‌های دیگر سیستم ایجاد شود.

[۱۱] binding

- می‌خواهیم پیاده‌سازی یک انتزاع هیچ‌گونه تأثیری روی مشتری نداشته باشد. در این حالت، مشتری فقط به واسط تعریف شده در انتزاع وابسته است و هیچ وابستگی مستقیمی به پیاده‌سازی ندارد. پیاده‌ساز می‌تواند یک کامپوننت یا حتی یک گره کاملاً مستقل باشد، بدون ایجاد هیچ‌گونه وابستگی در سطح کد یا کامپایل. این ساختار استقلال بین انتزاع و پیاده‌سازی را فراهم می‌کند.
- در زبان C++ یک الگوی ریزدانه‌ای وجود دارد به نام Cheshire Cat یا PIMPL[۱۲] که هدف آن کاهش وابستگی‌های کامپایلی بین مشتری و پیاده‌سازی است. در حالت عادی، اگر یک هدر فایل شامل تعریف کلاس باشد و این کلاس به یک فایل پیاده‌سازی (cpp.) مرتبط باشد، هر تغییری در آن فایل باعث می‌شود تمام مشتری‌هایی که از آن هدر استفاده می‌کنند، نیاز به کامپایل دوباره داشته باشند. این مشکل به این خاطر است که در C++ نمایش داخلی یک کلاس در واسط آن قابل مشاهده است، حتی اگر عملاً تغییری در واسط رخ نداده باشد. برای حل این مشکل، از یک تکنیک میانجی استفاده می‌شود. بدین شکل که یک هدر فایل خصوصی بین هدر فایل اصلی و فایل پیاده‌سازی قرار می‌گیرد. ساختار داخلی کلاس در این هدر خصوصی تعریف می‌شود و در فایل cpp. پیاده‌سازی می‌شود. کلاس اصلی فقط یک اشاره‌گر به این ساختار داخلی دارد. این لایه‌ی غیرمستقیم باعث می‌شود تا وابستگی‌های کامپایلی حذف شده و تغییری در پیاده‌سازی نیاز به کامپایل دوباره مشتری‌ها نداشته باشد. این تکنیک شبیه الگوی Bridge عمل می‌کند و در واقع نمونه‌ای خاص، ساده شده و سبک از آن محسوب می‌شود.
- وقتی می‌خواهیم پیاده‌سازی را به اشتراک بگذاریم. یعنی اشیا مختلفی در زمان اجرا بتوانند از یک پیاده‌سازی کننده استفاده کنند و مشتری از آن ناآگاه باشد.

:Iterator

- زمانی که می‌خواهیم به محتویات داخلی یک شی مرکب دسترسی پیدا کنیم و روی آن پیمایش انجام دهیم ولی نمی‌خواهیم ساختار داخلی به بیرون نشان داده شود. یعنی نمی‌خواهیم کپسوله‌سازی برای مشتری نقض شود.
- می‌خواهیم امکان این را داشته باشیم که همزمان چند پیمایش روی شی مرکب انجام دهیم.
- می‌خواهیم ساختارهای aggregate متفاوت را با یک واسط یکنواخت یعنی یک واسط واحد مورد پیمایش قرار دهیم. این واسط که برای Iterator تعریف شده است شامل first، next، isDone و غیره است که استاندارد شده است و چون در فوق‌کلاس واسط تعریف می‌شود، تمام پیمایش‌گرها همان واسط را پیاده‌سازی می‌کنند و استفاده از این الگو باعث استاندارد سازی این واسط شده است.

:State

[۱۲] Pointer to IMPLementation

- وقتی رفتار یک شی وابسته به وضعیت آن باشد و باید بتوان در زمان اجرا رفتار را به صورت خودکار و با کمترین تبعات، کمترین انتشار تغییرات و کد سبک، بر اساس وضعیتش تغییر داد.
- استفاده از سوییچ‌های پیچیده نشانه‌ی طراحی ضعیف است و باید به آن شک کرد. برای تشخیص تغییر حالت، معمولاً وضعیت اتریبیوت‌ها بررسی می‌شود، اما این کار می‌تواند پیچیده و غیرقابل‌ردیابی باشد. راه‌حل مناسب، استفاده از الگوی State است که در آن تغییر حالت به صورت صریح با جایگزینی شی داخلی وضعیت انجام می‌شود. این روش زمان دقیق تغییر حالت را مشخص می‌کند و ردیابی اجرا و اشکال‌زدایی را ساده‌تر می‌سازد.

مقایسه: موارد کاربرد الگوها به صورت مفصل شرح داده شد و ارتباط آن‌ها در قسمت الگوهای مرتبط بیان شده است.

۱۴-۳ کپسوله‌سازی

Bridge: در این الگو، مشتری صرفا با کلاس انتزاع در ارتباط بوده و از جزئیات داخل مجموعه بی‌خبراست. همچنین کلاس انتزاع نیز صرف با واسط پیاده‌سازی کننده ارتباط داشته و دید به داخل ندارد. کپسوله‌سازی به خوبی رعایت شده است.

Iterator: در این الگو، مشتری از طریق واسط با شی مرکب و پیمایش‌گر ارتباط دارد و ساختار درونی شی مرکب برای مشتری آشکار نشده است. عملیات پیمایش نیز توسط شی متخصص پیمایش‌گر انجام می‌شود و مشتری صرفا از طریق واسط با آن کار می‌کند. بنابراین در این قسمت کپسوله‌سازی رعایت شده است. اما به دلیل این که زیرکلاس‌های پیمایش‌گر برای انجام عملیات پیمایش به شی مرکب دید مستقیم دارند، کپسوله‌سازی در این قسمت نقض شده است.

State: در این الگو، کانتکست از طریق واسط با وضعیت ارتباط دارد، DIP برقرار است و کانتکست از تغییرات زیرکلاس‌های وضعیت منتزع شده است و دیدی به جزئیات آن‌ها ندارد. همچنین الگوریتم انتقال وضعیت نیز در دوش کانتکست نیست. در این قسمت کپسوله‌سازی رعایت شده است. اما برای تحقق انتقال وضعیت به صورت توزیع شده توسط زیرکلاس‌های وضعیت، نیاز است که این زیرکلاس‌ها به خود کانتکست دید داشته باشند که این دید با استفاده از ارسال خود کانتکست به عنوان پارامتر با استفاده از this محقق می‌شود و این دید موقتی کپسوله‌سازی را موقتا نقض می‌کند. البته این زیرکلاس‌ها عمدتا فقط با واسط مربوط به تغییر حالت داخلی کانتکست کار می‌کنند.

مقایسه: در الگوی اول رعایت شده و در بقیه در قسمت‌های شرح داده شده نقض می‌شود.

۳-۱۵ جداسازی دغدغه‌ها

Bridge: در این الگو، جداسازی دغدغه‌ها به خوبی دیده می‌شود و وجه انتزاع از وجه پیاده‌سازی کننده جدا شده است و انسجام به خوبی دیده می‌شود.

Iterator: در این الگو، وظیفه پیمایش شی مرکب نه توسط مشتری و نه توسط خود شی انجام می‌شود بلکه شی متخصص پیمایش‌گر تعریف شده است که این عملیات را انجام می‌دهد. به این شکل به خوبی جداسازی دغدغه‌ها رعایت شده است.

State: در این الگو، دغدغه‌های مربوط به رفتار وابسته به حالت به شی حالت مربوطه واسپاری شده و از کانتکست جدا شده است. همچنین الگوریتم مربوط به انتقال وضعیت نیز در دوش کانتکست نبوده و خود زیرکلاس‌های وضعیت می‌دانند که وضعیت‌های بعدی مناسب کدام هستند و در چه صورت انتقال به آن وضعیت‌ها نیاز است. بنابراین در این الگو جداسازی دغدغه‌ها به خوبی دیده می‌شود.

مقایسه: هر سه از روش‌هایی که شرح داده شد برای جداسازی دغدغه‌ها استفاده کرده‌اند.

۳-۱۶ مزایا و معایب (تبعات)

Bridge:

مزایا:

- در این الگو، انتزاع و پیاده‌سازی آن به صورت مستقل از یکدیگر تعریف می‌شوند و میان آن‌ها پیوندی دائمی برقرار نیست. این تمایز موجب می‌شود که پیوند میان این دو در زمان اجرا تعیین شود و بدین ترتیب انعطاف‌پذیری بالایی فراهم می‌کند. به واسطه این ساختار، می‌توان در زمان اجرا یک پیاده‌سازی مشخص را به یک انتزاع تخصیص داد یا حتی پیاده‌سازی مورد نظر را بدون نیاز به تغییر در انتزاع یا کد مشتری، جایگزین نمود.
- در این الگو، از آنجا که ساختارهای توارثی انتزاع و پیاده‌سازی کننده از یکدیگر تفکیک شده‌اند، هر یک می‌توانند به‌طور مستقل گسترش یابند و گسترش‌پذیری سیستم به‌طور قابل توجهی افزایش می‌یابد. به‌واسطه رعایت اصل DIP در سراسر این الگو، تغییرات یا افزودن زیرکلاس‌های جدید در یکی از این ساختارها، تأثیری بر سایر بخش‌های سیستم نخواهد داشت.
- جزئیات پیاده‌سازی کاملا از دید مشتری پنهان است و مشتری هیچ وابستگی به آن‌ها ندارد. تغییر در پیاده‌سازها در مشتری تاثیر نمی‌گذارد.

Iterator:

مزایا:

- انواع پیمایش‌ها را روی یک شی مرکب می‌توان در قالب پیمایش‌گرها تعریف کرد. یعنی پیمایش‌گرهای مختلف برای مثال pre-order، post-order و غیره برای درخت.
- باعث ساده شدن شی مرکب می‌شود زیرا واسط مربوط به پیمایش دیگر روی آن تعریف نشده و در پیمایش‌گر تعریف شده است.
- برای یک شی مرکب می‌توان به صورت همزمان بیش از یک پیمایش فعال داشت.

عیب: بین پیمایش‌گر و شی مرکب جفت‌شدگی شدید وجود دارد و کپسوله‌سازی نیز نقض شده است. مانند الگوی Factory Method نقض DIP و ارتباط مستقیم زیرکلاس‌های عینی وجود دارد.

State:

مزایا:

- در این الگو، رفتارهای مرتبط با هر حالت در اشیای وضعیت مربوطه قرار می‌گیرند. چون اصل DIP رعایت می‌شود، کانتکست به وضعیت‌های خاص وابسته نیست و آن‌ها را نمی‌شناسد. بنابراین می‌توان وضعیت‌ها و انتقال‌های جدید را انعطاف‌پذیر و بدون تغییر در کانتکست یا بخش‌های دیگر سیستم اضافه کرد. در مقابل، در طراحی با سوییچ‌های متعدد یعنی حالت قبل از اعمال الگو، افزودن یک وضعیت جدید نیازمند تغییر در همه سوییچ‌هاست که منجر به انتشار تغییر و کد شکننده می‌شود.
- یکی از مزایای دیگر این الگو این است که انتقال حالت‌ها به صورت صریح انجام می‌شود. هنگام دیباگ یا ردیابی اجرای برنامه، دقیقا مشخص است که در چه لحظه‌ای یک تغییر حالت مهم رخ داده، چون این تغییر با جایگزینی یک شی وضعیت با شی دیگر همراه است. برخلاف روش‌هایی که صرفا با تغییر مقادیر اتریبیوت‌ها وارد حالت جدید می‌شوند، در اینجا تغییر حالت کاملا شفاف و قابل تشخیص است.
- اشیای وضعیت می‌توانند به اشتراک گذاشته شوند و این می‌تواند تعداد اشیای وضعیت که در زمان اجرا ایجاد میشوند را کم کند.

معایب:

- در زمان اجرا، ممکن است تعداد زیادی شی وضعیت ایجاد شود چون هر شی کانتکست یک وضعیت داخلی دارد. این افزایش تعداد اشیا می‌تواند به مصرف بالای منابع سیستم منجر شود. برای کاهش این فشار، اگر اشیای وضعیت stateless طراحی شوند، می‌توان آن‌ها را بین کانتکست‌های مختلف به اشتراک گذاشت و از ایجاد مکرر اشیا جلوگیری کرد. در نتیجه، مصرف منابع کاهش می‌یابد و کارایی

بهبود می‌یابد.

- در این الگو، یک سطح indirection اضافه می‌شود. به این معنا که بجای اجرای مستقیم رفتار توسط کانتکست، پیام‌ها به اشیای وضعیت فرستاده می‌شوند که می‌تواند کمی عملکرد را کاهش دهد. با این حال، این هزینه پذیرفته می‌شود چون در عوض، نگهداری سیستم آسان‌تر می‌شود، کاهش جفت‌شدگی داریم، تکرار حذف می‌شود و دیگر نیازی به استفاده از سوییچ‌ها یا ساختارهای توارثی پیچیده برای پوشش حالت‌های مختلف نیست. این مزایا باعث می‌شوند الگوی State انتخابی مطلوب باشد.

۱۷-۳ الگوهای مرتبط

Bridge: این الگو معمولا از ابتدا و در مرحله‌ی طراحی به کار گرفته می‌شود تا اجزای مختلف یک برنامه بتوانند به‌صورت مستقل از هم گسترش یابند. در مقابل، Adapter اغلب در شرایطی به کار می‌رود که برنامه‌ای از پیش موجود است و نیاز داریم کلاس‌هایی ناسازگار را به شکلی سازگار با یکدیگر وادار به همکاری کنیم. هرچند الگوهای Bridge، State، Strategy (و تا حدی Adapter) ساختارهای مشابهی دارند، اما هرکدام برای حل مسئله‌ای خاص طراحی شده‌اند. همه‌ی این الگوها بر پایه‌ی واسپاری استوارند، بطوری که مسئولیت انجام کار را به سایر اشیا واگذار می‌کنند. الگوی Bridge را می‌توان با Abstract Factory ترکیب کرد. این ترکیب زمانی مفید است که برخی انتزاع‌های تعریف شده در Bridge فقط با پیاده‌سازی‌های خاصی کار می‌کنند. در این صورت، Abstract Factory می‌تواند این وابستگی‌ها را کپسوله کند و پیچیدگی را از دید کد مشتری پنهان سازد. همچنین می‌توان Bridge را با Builder ترکیب کرد. در این ترکیب، کلاس Director نقش Abstraction را ایفا می‌کند و سازنده‌های مختلف، نقش پیاده‌سازی کننده را برعهده دارند.

Iterator: می‌توان از الگوی Iterator برای پیمایش ساختارهای درختی در الگوی Composite استفاده کرد. الگوی Factory Method را می‌توان همراه با Iterator به کار برد تا زیرکلاس‌های شی مرکب بتوانند انواع مختلفی از پیمایش‌گرهایی را برگردانند که با ساختار داخلی خودشان سازگار باشند (این موضوع در ساختار این الگو قابل مشاهده است). با ترکیب Iterator و Memento می‌توان وضعیت فعلی پیمایش را ذخیره کرد و در صورت نیاز به آن بازگشت داشت. همچنین، ترکیب Visitor و Iterator این امکان را می‌دهد که یک ساختار داده‌ی پیچیده را پیمایش کنیم و عملیات خاصی را بر روی عناصر آن حتی اگر متعلق به کلاس‌های مختلفی باشند اعمال کنیم.

State: الگوهای Bridge، State، Strategy و تا حدی Adapter ساختاری مشابه دارند، چون همگی بر پایه‌ی واگذاری کار به اشیای دیگر طراحی شده‌اند. با این حال، هر یک مسئله‌ای متفاوت را حل می‌کنند. الگوی State را می‌توان گسترشی از Strategy در نظر گرفت. هر دو رفتار کانتکست را از طریق واگذاری

به اشیای کمکی تغییر می‌دهند. در Strategy، این اشیا کاملا مستقل و از هم بی‌خبر هستند. اما در State، وضعیت‌های مشخص می‌توانند به یکدیگر وابسته باشند و حتی وضعیت کانتکست را به دلخواه تغییر دهند.

۳-۱۸ اصول شی‌گرایی

۳-۱۸-۱ OCP

Bridge: در این الگو، مشتری صرفا با کلاس انتزاع ارتباط دارد و از تغییرات داخل مجموعه بی‌تاثیر است و مجموعه می‌تواند به هر شکل رشد کند. همچنین رابطه کلاس انتزاع با پیاده‌سازی کننده از طریق واسط است و DIP رعایت شده در نتیجه OCP وجود دارد و زیرکلاس‌های پیاده‌سازی کننده می‌توانند رشد کنند و انتزاع هیچ تاثیری از آن نمی‌پذیرد. تغییرات انتزاع نیز هیچ تاثیری روی پیاده‌سازی کننده‌ها ندارد و این اصل به خوبی رعایت می‌شود. فقط پیکربند است که به زیرکلاس‌ها دید دارد و تاثیر می‌پذیرد.

Iterator: در این الگو، مشتری از طریق واسط با شی مرکب و پیمایش‌گر ارتباط داشته و DIP رعایت شده است و در نتیجه آن OCP در این قسمت دیده می‌شود که باعث شده است تغییرات زیرکلاس‌های پیمایش‌گر و شی مرکب تاثیری روی مشتری نداشته باشند و به آن منتشر نشوند. اما به این دلیل که زیرکلاس‌های پیمایش‌گر و شی مرکب ارتباط مستقیم با یکدیگر دارند و DIP در این قسمت نقض شده است، تغییرات آن‌ها بر یکدیگر منتشر شده و OCP در این قسمت نقض می‌شود.

State: در این الگو، کانتکست از طریق واسط با وضعیت در ارتباط است و رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود. در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار می‌شود و کانتکست از تغییرات زیرکلاس‌های وضعیت منتزع شده است. می‌توان وضعیت‌های جدید تعریف کرد و حتی در زمان اجرا وضعیت داخل کانتکست تغییر کند و کانتکست از آن منتزع است. پس در این قسمت‌ها OCP رعایت می‌شود. اما مشتری به عنوان پیکربند اولیه، به زیرکلاس‌های وضعیت دید مستقیم و موقت دارد تا بتواند حالت اولیه کانتکست را مقداردهی کند که در این قسمت OCP نقض می‌شود.

مقایسه: هر سه الگو از روش‌هایی در قسمت‌های شرح داده شده استفاده کرده‌اند تا این اصل محقق شود.

۳-۱۸-۲ LSP

Bridge: در این الگو، زیرکلاس‌های پیاده‌سازی کننده با پدر خود و زیرکلاس‌های انتزاع با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

Iterator: در این الگو، زیرکلاس‌های پیمایش‌گر با پدر خود و زیرکلاس‌های شی مرکب با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

State: در این الگو، زیرکلاس‌های وضعیت با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

مقایسه: هر سه این اصل را رعایت می‌کنند.

ISP ۳-۱۸-۳

Bridge: هدف این الگو جداسازی وجه انتزاع و پیاده‌سازی کننده بوده و نسبت به قبل از اعمال الگو که این دو با یک واسط ارائه می‌شدند، اکنون مسئولیت‌ها جدا شده است و دو واسط ارائه می‌شود که نوعا عملیات تعریف شده در پیاده‌سازی کننده‌ها نیز ریزدانه‌تر از انتزاع است. در نتیجه تفکیک کارها و واسط‌ها به خوبی شکل گرفته است.

Iterator: در این الگو، واسط مربوط به پیمایش از خود شی مرکب جدا شده است و شی متخصص پیمایش تعریف شده است. به این شکل جداسازی دغدغه‌ها و واسط‌ها به خوبی رعایت شده است.

State: در این الگو، دغدغه‌های مربوط به رفتار وابسته به حالت به شی حالت مربوطه واسپاری شده است و نوعا برای حالت‌های مختلف ساختار توارثی مربوطه برای تعریف حالت‌های آن تشکیل می‌شود و الگوریتم انتقال حالت نیز بر عهده خود زیرکلاس‌های وضعیت است. بدین شکل جداسازی دغدغه‌ها و واسط‌ها به خوبی رعایت شده است.

مقایسه: هر سه الگو این اصل را محقق می‌کنند.

DIP ۴-۱۸-۳

Bridge: در این الگو، مشتری صرفا با کلاس انتزاع ارتباط داشته و هیچ دیدی نسبت به داخل مجموعه ندارد. همچنین کلاس انتزاع رابطه یک‌سویه از طریق واسط با پیاده‌سازی کننده دارد و تغییرات زیرکلاس‌های پیاده‌سازی کننده به انتزاع منتشر نمی‌شود. فقط پیکربند است که به زیرکلاس‌های پیاده‌سازی کننده و انتزاع دید دارد و در این قسمت DIP نقض می‌شود.

Iterator: در این الگو، مشتری از طریق واسط با پیمایش‌گر و شی مرکب در ارتباط است و DIP در این قسمت رعایت شده است که در نتیجه آن، مشتری دید به داخل آن‌ها نداشته و همچنین تغییرات زیرکلاس‌های آن‌ها به مشتری منتشر نمی‌شود. اما به دلیل این که زیرکلاس‌های پیمایش‌گر و شی مرکب دید مستقیم به یکدیگر

دارند، DIP در این قسمت نقض شده است و تغییرات آن‌ها به یکدیگر منتشر می‌شود.

State: در این الگو، کانتکست از طریق واسط با وضعیت در ارتباط است و رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود که در این قسمت DIP رعایت شده و کانتکست از تغییرات زیرکلاس‌های وضعیت منتزع شده است. البته مشتری به عنوان پیکربند اولیه، به زیرکلاس‌های وضعیت دید مستقیم دارد به منظور مقداردهی اولیه به کانتکست که در اینجا DIP نقض می‌شود. زیرکلاس‌های وضعیت نیز به منظور تحقق الگوریتم انتقال وضعیت، به کانتکست دید موقت دارند البته نوعا فقط با واسط مربوط به جایگزینی حالت داخلی آن کار می‌کنند.

مقایسه: هر سه الگو از روش‌هایی در قسمت‌های شرح داده شده استفاده کرده‌اند تا این اصل محقق شود.

CRP ۵-۱۸-۳

Bridge: این الگو در زمان اجرا و پس از تشکیل پیوندهای delegation توسط پیکربند محقق می‌شود. مشتری دستور را به انتزاعی که با آن در ارتباط است ارسال می‌کند و انتزاع نیز عملیات مربوط به پیاده‌سازی کننده‌ای که با آن رابطه دارد را فراخوانی می‌کند. پس این اصل به خوبی رعایت شده است و از ساختارهای توارثی ترکیبی متعدد جلوگیری کرده است.

Iterator: در این الگو، واسط مربوط به پیمایش از شی مرکب جدا شده است و در زمان اجرا هرکدام از اشیاء مرکب می‌دانند که پیماش‌گر مخصوص به خودشان کدام است و از آن نمونه‌گیری کرده و خود را از طریق this به آن پاس می‌دهند. پیمایش‌گر نیز دید به شی مرکب را مانا می‌کند و از طریق delegation با آن کارکرده و عملیات مربوط به پیمایش را پیاده‌سازی می‌کند.

State: در این الگو، کانتکست از طریق واسط با وضعیت در ارتباط است و رفتار وابسته به حالت را به شی حالت مربوطه واسپاری می‌کند و این رفتارها از طریق واسپاری محقق می‌شوند. در واقع شی اصلی حاوی اشیای داخلی برای حالت‌های مختلف است و رفتار مربوطه را به آن‌ها واسپاری می‌کند. پس این اصل به خوبی در این الگو رعایت شده است.

مقایسه: هر سه این اصل را رعایت می‌کنند.

PLK ۶-۱۸-۳

Bridge: در این الگو دید ترایا[۱۳] دیده نشده و زنجیره پیغام[۱۴] تشکیل نمی‌شود. پیاده‌سازی کننده از طریق انتزاع در اختیار مشتری قرار نمی‌گیرد و این اصل رعایت شده است.

Iterator: در پیاده‌سازی درست این الگو، دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. مشتری از طریق واسط پیمایش‌گر عملیات مربوط به پیمایش روی شی مرکب را فراخوانی می‌کند اما شی پیمایش‌گر خود شی مرکب را در اختیار مشتری قرار نمی‌دهد و این الگو نقض نمی‌شود.

State: در این الگو دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. رفتار وابسته به حالت از کانتکست به اشیای حالت مربوطه واسپاری می‌شوند و آن‌ها در جواب خود شی را برنمی‌گردانند و دید ترایا ایجاد نمی‌شود.

مقایسه: هر سه این اصل را رعایت می‌کنند.

۱۹-۳ الگوهای GRASP

Information Expert ۱-۱۹-۳

Bridge: در این الگو، اگر زیرکلاس‌های پیاده‌سازی کننده حاوی تمام اطلاعات لازم برای انجام عملیات خود باشند، این الگو رعایت می‌شود. ولی ممکن است انتزاع حاوی اطلاعاتی باشد که زمان فراخوانی عملیات پیاده‌سازی کننده‌ها آن‌ها را ارسال کند که در این صورت این الگو نقض می‌شود.

Iterator: در این الگو، عملیات مربوط به پیمایش در کنار داده مربوطه قرار نگرفته است زیرا داده در اختیار شی مرکب است ولی عملیات پیمایش در شی متخصص پیمایش تعریف شده است که باعث می‌شود این الگو نقض شود. البته این کار با هدف جلوگیری از نقض کپسوله‌سازی و کاهش جفت‌شدگی بین مشتری و شی مرکب و همچنین ساده‌تر سازی شی مرکب انجام شده است.

State: در این الگو، رفتار وابسته به حالت به شی مربوط به آن حالت واسپاری می‌شود و آن شی حاوی منطق لازم برای پیاده‌سازی آن حالت است و اگر نیاز به داده‌ای نباشد که در نزد کانتکست است، این الگو رعایت می‌شود و اشیای وضعیت به صورت مستقل منطق مربوطه را اجرا کرده و بعد الگوریتم انتقال وضعیت را اجرا می‌کنند. اما اگر کانتکست حاوی اطلاعاتی باشد که برای اشیای وضعیت مورد نیاز است، در این صورت این

[۱۳] transitive visibility
[۱۴] message chain

الگو نقض می‌شود.

مقایسه: در هر سه در قسمت‌های شرح داده شده این الگو نقض می‌شود.

Creator ۲-۱۹-۳

Bridge: در این الگو، پیکربند ثالث وظیفه ساخت اشیا را دارد. در حالی که کلاس انتزاع رابطه association با پیاده‌سازی کننده داشته و با آن کار می‌کند. همچنین مشتری نیز با انتزاع رابطه داشته و با آن کار می‌کند که نشان دهنده اولویت بالاتر برای ساخت شی است ولی پیکربند که اولویت ۵ برای ساخت را دارد این نقش را ایفا می‌کند پس این الگو محقق نشده است.

Iterator: در این الگو، پیکربند یا مشتری می‌تواند از شی مرکب نمونه بگیرد که اولویت بالاتر وجود ندارد و این الگو رعایت می‌شود. در این ساختار، شی مرکب که اطلاعات اولیه پیماش‌گر را دارد، از آن نمونه‌گیری کرده و در اختیار مشتری می‌گذارد تا از آن استفاده کند. استفاده کننده اصلی پیمایش‌گر مشتری است که این اولویت ۴ بوده و بالاتر از ساختار فعلی است. بنابراین در این قسمت این الگو نقض شده است.

State: در این الگو، پیکربندی اولیه را مشتری انجام می‌دهد و کانتکست را با حالت اولیه مقداردهی می‌کند در حالی که کانتکست با اشیای وضعیت ارتباط داشته و با آن‌ها کار می‌کند پس اولویت بالاتری برای نمونه‌گیری داشت و همچنین الگوریتم انتقال وضعیت را خود کانتکست اجرا نکرده و اشیای وضعیت اجرا می‌کنند و کانتکست را با اشیای وضعیت جدید نمونه‌گیری و مقداردهی می‌کنند. پس این الگو نقض می‌شود.

مقایسه: در هر سه بنابر توضیحات ارائه شده این الگو نقض می‌شود.

Low Coupling ۳-۱۹-۳

Bridge: در این الگو، مشتری با کلاس انتزاعی Abstraction ارتباط دارد و دید مستقیم به پیاده‌سازی کننده‌ها و همچنین زیرکلاس‌های Abstraction ندارد. Abstraction نیز با واسط پیاده‌سازی کننده در ارتباط است و DIP برقرار بوده و تغییرات زیرکلاس‌های پیاده‌سازی کننده به آن منتشر نمی‌شود. پیاده‌سازی کننده نیز دید به Abstraction یا مشتری ندارد و ارتباط بین Abstraction و پیاده‌سازی کننده یک‌سویه است. زیرکلاس‌های پیاده‌سازی کننده و انتزاع می‌توانند به صورت مستقل رشد کنند. در نتیجه در این مجموعه جفت‌شدگی مطلوب است. ولی پیکربند ثالث باید به زیرکلاس‌ها دید مستقیم داشته باشد تا پیکربندی را انجام دهد و این به معنی جفت‌شدگی بالا است. این ساختار مشکلات حالت قبل از اعمال الگو نظیر مشکلات ساختار توارثی و زیرکلاس‌های متعدد و ترکیبی و غیره را ندارد.

Iterator: در این الگو، مشتری از طریق واسط با Aggregate و پیمایش‌گر ارتباط دارد. بنابراین DIP برقرار است و تغییرات زیرکلاس‌های Aggregate و پیمایش‌گر تاثیری روی مشتری نمی‌گذارد. همچنین مشتری دید به جزئیات Aggregate ندارد و خود عمل پیمایش را انجام نمی‌دهد بلکه صرفا با واسط پیمایش‌گر کار می‌کند. پس در این قسمت جفت‌شدگی مطلوب است. اما به این دلیل که زیرکلاس‌های Aggregate باید خودشان پیمایش‌گر خود را مشخص کنند، دید مستقیم بین زیرکلاس‌هایشان وجود داشته، DIP نقض شده است و جفت‌شدگی در این قسمت وجود دارد. پیمایش‌گرها نیز دید مانا به Aggregate مربوط به خود دارند تا عملیات پیمایش را انجام دهند و این به معنی جفت‌شدگی بالا در این قسمت است.

State: در این الگو، کانتکست ارتباط و دید مستقیم به زیرکلاس‌های وضعیت ندارد و این ارتباط از طریق واسط است. بنابراین در این قسمت اصل DIP رعایت شده است و کانتکست از تغییرات زیرکلاس‌های وضعیت منتزع شده است. الگوریتم تغییر وضعیت نیز در دوش کانتکست نبوده و این الگوریتم به اشیا وضعیت توزیع شده است. بدین جهت یک دید از سمت زیرکلاس‌های وضعیت به کانتکست به منظور انتقال وضعیت، وجود دارد اما این دید لزوما مانا نیست و کنترل شده است. مشتری نیز پیکربند اولیه بوده و یک شی وضعیت به عنوان وضعیت اولیه نمونه‌گیری کرده و به عنوان پارامتر هنگام نمونه‌گیری از کانتکست به آن تخصیص می‌دهد. مشتری دید مستقیم به کانتکست دارد و دید مشتری به زیرکلاس‌های وضعیت از نوع وابستگی و غیرمانا است ولی به آن‌ها دید مستقیم دارد. در کل جفت‌شدگی در این الگو در قسمت کانتکست مطلوب است.

مقایسه: با توجه به توضیحات مفصل هر بخش، هر سه الگو در قسمت‌هایی این الگو را محقق می‌کنند.

High Cohesion ۴-۱۹-۳

Bridge: در این الگو، وجه انتزاعی از وجه پیاده‌سازی جدا شده است و مسئولیت پیاده‌سازی به متخصص پیاده‌سازی سپرده شده است که به این شکل به جداسازی دغدغه‌ها توجه شده و انسجام موجودیت‌ها افزایش یافته است. در موجودیت‌ها عملیات نامرتبط وجود ندارد و انسجام در سطح مطلوبی است.

Iterator: در این الگو، شی متخصص پیمایش تعریف شده است و عملیات مربوط به پیمایش از دوش مشتری یا Aggregate برداشته شده و اکنون پیمایش‌گر این نقش را ایفا می‌کند و این کمک به جداسازی دغدغه‌ها و افزایش انسجام کرده است. مشتری صرفا با واسط‌ها کار کرده، Aggregate عملیات مربوط به شی مرکب را انجام داده و پیمایش‌گر نیز وظیفه پیمایش را انجام می‌دهد.

State: در این الگو، رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود و آن شی است که پیاده‌سازی کننده رفتار بوده و همچنین انتقال حالت را انجام می‌دهد. این بار از دوش کانتکست برداشته شده است و کانتکست از پیاده‌سازی‌های حالت‌ها، switch statement های پیچیده و انتقال حالت رها شده است

و چندکارگی از بین رفته است. در نتیجه اعمال این الگو، انسجام افزایش یافته و مطلوب شده است.

مقایسه: در هر سه الگو در قسمت‌های شرح داده شده رعایت شده است.

۳-۱۹-۵ Controller

Bridge: در این الگو، به آن صورت وظیفه کارچرخانی و آغاز زنجیره‌های تعاملی دیده نمی‌شود و از کنترلر استفاده نشده است.

Iterator: در این الگو، کارچرخانی و آغاز زنجیره‌های تعاملی دیده نمی‌شود و از کنترلر استفاده نشده است.

State: در این الگو، کارچرخانی و آغاز زنجیره‌های تعاملی دیده نمی‌شود و از کنترلر استفاده نشده است.

مقایسه: هیچ‌یک از الگوها محقق نمی‌کنند.

۳-۱۹-۶ Polymorphism

Bridge: در این الگو، از توارث برای انتزاع و پیاده‌سازی کننده استفاده شده است و می‌توان این دو موجودیت را به صورت مستقل رشد داد و انواع پیاده‌سازی کننده و انتزاع اضافه کرد. پس این الگو محقق شده است.

Iterator: در این الگو، مشتری از طریق واسط با شی مرکب و پیمایش‌گر در ارتباط است و با استفاده از توارث، می‌توان با یک واسط واحد، انواع مختلف شی مرکب و همچنین پیمایش‌گر داشت و با آن‌ها کار کرد. در نتیجه این الگو به خوبی رعایت شده است.

State: در این الگو، کانتکست از طریق واسط با وضعیت در ارتباط بوده و رفتار وابسته به حالت را به شی مربوطه واسپاری می‌کند و بدین ترتیب تغییرات زیرکلاس‌های وضعیت به کانتکست منتشر نمی‌شود و از طریق ساختار توارثی تعریف شده برای حالت، می‌توان اشیا از وضعیت‌های مختلف را به کانتکست تخصیص کرده و کانتکست رفتار وابسته به حالت را به آن واسپاری می‌کند و رفتارهای گوناگون اجرا می‌شوند و بدین شکل این الگو محقق شده است.

مقایسه: هر سه الگو در قسمت‌های شرح داده شده این الگو را محقق می‌کنند.

Indirection ۷-۱۹-۳

Bridge: در این الگو، هدف مشتری اجرای دقیق عملیات تعریف شده در پیاده‌سازی کننده‌ها نیست زیرا آن عملیات نوعا ریزدانه تعریف می‌شوند و این انتزاع است که عملیات اصلی که مورد نظر مشتری است را تعریف می‌کند. بنابراین نمی‌توان گفت که کلاس انتزاع یک سطح indirection بین مشتری و پیاده‌سازی کننده ایجاد کرده است. کلاس انتزاع صرفا سرویس‌های ریزدانه خود را از طریق واسپاری به پیاده‌سازی کننده‌ای که با آن در ارتباط است دریافت می‌کند. اگر عملیات پیاده‌سازی کننده‌ها ریزدانه نبوده و دقیقا مانند انتزاع باشد که صرفا آن‌ها را پیاده‌سازی می‌کنند، در این صورت انتزاع یک indirection محسوب می‌شود.

Iterator: پس از اعمال این الگو، مشتری خود دید به داخل شی مرکب نداشته و عملیات پیمایش را خودش انجام نمی‌دهد. همچنین عملیات پیمایش از دوش شی مرکب نیز برداشته شده است. اکنون مشتری برای ارتباط با شی مرکب به منظور پیمایش، از واسط پیمایش‌گر استفاده می‌کند و این یک سطح indirection ایجاد کرده است. پس این الگو رعایت می‌شود.

State: در این الگو یک سطح indirection اضافه شده است و همه کارها بجای آن که توسط خود شی کانتکست انجام شود، به شی دیگری سپرده می‌شود. در واقع کانتکست رفتار وابسته به حالت را به شی مربوط به آن حالت که با آن در ارتباط است واسپاری می‌کند و به این شکل این الگو محقق شده است.

مقایسه: در هر سه الگو بنابر توضیحات ارائه شده رعایت می‌شود.

Pure Fabrication ۸-۱۹-۳

Bridge: در اثر اعمال این الگو، پیاده‌سازی کننده‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو بوجود آمده‌اند. کلاس‌های انتزاع نیز اگر ناشی از تحلیل قلمرو مسئله نباشند به همین شکل اشیا جعلی محسوب می‌شوند. پس این الگو محقق شده است.

Iterator: در اثر اعمال این الگو، پیمایش‌گرها تعریف می‌شوند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو بوجود آمده‌اند. پس شی جعلی محسوب می‌شوند.

State: در این الگو، کلاس‌های نمایانگر وضعیت‌های مختلف می‌توانند ناشی از تحلیل قلمرو مسئله باشند. مانند وضعیت‌های مختلف دانشجو، مثل وضعیت مشروطی، که در این صورت این اشیا جعلی نیستند. اگر آن اشیا ناشی از تحلیل قلمرو مسئله نباشند و برای تحقق اهداف الگو تعریف شوند، در این صورت جعلی محسوب می‌شوند.

مقایسه: هر سه الگو محقق می‌کنند.

Protected Variations ۹-۱۹-۳

Bridge: در این الگو، مشتری صرفا با کلاس انتزاع در ارتباط است و تغییرات داخل مجموعه به مشتری منتشر نمی‌شوند. همچنین کلاس انتزاع نیز از طریق واسط با پیاده‌سازی کننده‌ها ارتباط داشته و تغییرات زیرکلاس‌های پیاده‌سازی کننده به آن منتشر نمی‌شود. پیاده‌سازی کننده‌ها نیز دید نسبت به انتزاع ندارند. در نتیجه برای این قسمت‌ها این الگو محقق شده است. ولی پیکربند که به مجموعه دید دارد، از تغییرات آن‌ها تاثیر می‌پذیرد. همچنین تغییر واسط در پیاده‌سازی کننده و انتزاع باعث بروز تغییر در زیرکلاس‌ها می‌شود.

Iterator: در این الگو، مشتری از طریق واسط با شی مرکب و پیمایش‌گر در ارتباط است و در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار است و تغییرات زیر پیمایش‌گر و شی مرکب به مشتری منتشر نمی‌شوند. البته به این دلیل که زیرکلاس‌های عینی پیمایش‌گر و شی مرکب ارتباط مستقیم به هم داشته و DIP نقض شده است، تغییرات آن‌ها به یکدیگر منتشر می‌شود. البته تغییر در واسط‌ها منجر به انتشار تغییرات به مشتری و زیرکلاس‌ها می‌شود اما واسط پیمایش‌گر تا حدی استاندارد بوده و واسط شی مرکب نیز تا حد امکان پایدار تعریف می‌شود.

State: در این الگو، رفتار وابسته به حالت به شی حالت مربوطه واسپاری می‌شود ولی کانتکست از طریق واسط با وضعیت در ارتباط است و اصل DIP برقرار بوده در نتیجه OCP برقرار شده و تغییرات زیرکلاس‌های وضعیت به کانتکست منتشر نمی‌شود و می‌توان در زمان اجرا وضعیت‌های داخل کانتکست را تغییر داد که این منجر به تغییر رفتار وابسته به حالت می‌شود. البته مشتری که پیکربند اولیه است، به زیرکلاس‌های وضعیت به منظور مقداردهی اولیه به کانتکست، دید مستقیم دارد. زیرکلاس‌های حالت نیز به منظور انجام انتقال حالت، به کانتکست دید موقت پیدا می‌کنند و از واسط مخصوص تغییر حالت درونی کانتکست استفاده می‌کنند و تا زمانی که واسط‌ها تغییر نکنند، تغییر منتشر نمی‌شود.

مقایسه: هر سه الگو بنابر توضیحات ارائه شده در قسمت‌های بیان شده الگو را محقق می‌کنند.

فصل ۴

ارزیابی الگوهای Decorator، Proxy و Strategy

در این فصل، سه الگوی Decorator، Proxy و Strategy مبتنی بر معیارهایی که در فصل یک معرفی شدند، تحلیل، ارزیابی و مقایسه می‌شوند.

۴-۱ دسته

Decorator: این الگو در دسته الگوهای طراحی ساختاری[۱] قرار دارد. این دسته از الگوها بیش‌تر به ساختاردهی کلاس‌ها و سیستم‌ها توجه دارند با هدف کم کردن جفت‌شدگی[۲]. این کار نوعا به شکل جدا کردن انتزاع یا واسط از پیاده‌سازی انجام می‌شود. یعنی وجه انتزاعی و specification از بخش پیاده‌سازی کننده جدا می‌شود که به این شکل امکان رشد مستقل این دو فراهم می‌شود. در این الگوها، ساختار وجه اساسی است.

Proxy: این الگو در دسته الگوهای طراحی ساختاری قرار دارد. این دسته از الگوها بیش‌تر به ساختاردهی کلاس‌ها و سیستم‌ها توجه دارند با هدف کم کردن جفت‌شدگی. این کار نوعا به شکل جدا کردن انتزاع یا واسط از پیاده‌سازی انجام می‌شود. یعنی وجه انتزاعی و specification از بخش پیاده‌سازی کننده جدا می‌شود که به این شکل امکان رشد مستقل این دو فراهم می‌شود. در این الگوها، ساختار وجه اساسی است.

---

[۱] structural

[۲] coupling

Strategy: این الگو در دسته الگوهای رفتاری[۳] قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام[۴] و انعطاف‌پذیری.

مقایسه: دو الگوی اول در دسته ساختاری و الگوی سوم در دسته رفتاری است.

۲-۴ هدف

Decorator: این الگو روشی برای افزودن یا حذف رفتار (مسئولیت) به یک شی در زمان اجرا به صورت دینامیک و انعطاف‌پذیر است. این الگو با واگذاری[۵] کار می‌کند و همچنین یک wrapper است. یک راه‌حل جایگزین آن، استفاده از وراثت است. یعنی تعریف زیرکلاس‌هایی با ترکیب‌های مختلف قابلیت‌ها. اما این روش منجر به انفجار ترکیبی و ساختار توارثی عمیق و صلب می‌شود و حذف یا افزودن قابلیت‌ها به اشیا دشوار و پرهزینه است. در مقابل، Decorator با اجتناب از وراثت پیچیده، امکان ترکیب و تغییر رفتار را بدون تغییر ساختار اصلی یا بازسازی اشیا فراهم می‌کند. در این الگو، می‌توان در زمان اجرا به صورت دینامیک رفتارهای جدید را به زنجیره اضافه یا حذف کرد، بدون نیاز به پیش‌بینی و تعریف همه ترکیب‌های ممکن به‌عنوان زیرکلاس. یک راه‌حل جایگزین دیگر ولی نامطلوب این است که یک کلاس بزرگ با تمام قابلیت‌ها طراحی شود که فیچرهای آن قابل روشن و خاموش شدن (on/off) باشند (feature-laden class). اما این نوع کلاس‌ها معمولا پیچیده بوده و خطر تبدیل به God Class وجود دارد که نگهداری را دشوار می‌کند.

Proxy: این الگو یک wrapper است و در نقش نماینده‌ای برای یک شی اصلی ظاهر می‌شود. به‌گونه‌ای که مشتری متوجه تفاوت آن با شی اصلی نمی‌شود. این الگو وظایفی را به جای شی اصلی انجام می‌دهد تا به اهدافی مانند کنترل دسترسی یا صرفه‌جویی در حافظه برسد. یک سطح indirection ایجاد می‌کند و می‌تواند عملیاتی قبل یا بعد از ارسال درخواست به شی اصلی انجام دهد. انواع مختلفی از پراکسی وجود دارد که بسته به نوع رفتاری که نشان می‌دهند، دسته‌بندی می‌شوند. اما همه‌ی آن‌ها در اصل کاری اضافی برای شی اصلی انجام می‌دهند. برای مثال، نوعی از الگوی پراکسی به نام virtual proxy وجود دارد. وظیفه‌ی آن جلوگیری از بارگیری زودهنگام اشیای پرهزینه، مانند اشیایی با مصرف بالای حافظه است. برای نمونه، یک تصویر بزرگ در یک سند را در نظر بگیرید که بارگذاری مستقیم آن در حافظه هزینه‌بر است. به‌جای این کار، از lazy loading استفاده می‌شود. یعنی تصویر تنها زمانی که واقعا به محتوای آن نیاز داریم، بارگذاری می‌شود. وقتی نیاز به

[۳] behavioral
[۴] cohesion
[۵] delegation

نمایش محتوای واقعی تصویر به کاربر پیش می‌آید، پراکسی تصویر اصلی را بارگذاری می‌کند و از آن لحظه به بعد، درخواست‌ها را مستقیما به شی اصلی واسپاری می‌کند. مشتری تا پیش از بارگذاری، این پراکسی را به‌عنوان تصویر اصلی می‌شناسد. بنابراین، پراکسی تا زمانی فعال باقی می‌ماند که شی اصلی نیاز نشده و پس از ایجاد آن، به یک واسطه‌ی ساده برای واسپاری درخواست‌ها تبدیل می‌شود. سایر انواع پراکسی در قسمت موارد کاربرد شرح داده شده‌اند.

Strategy: این الگو شباهت ساختاری با الگوی State دارد، اما هدفشان کاملا متفاوت است. در Strategy، ما با یک خانواده از الگوریتم‌ها سر و کار داریم که همگی یک وظیفه را انجام می‌دهند، ولی هرکدام به شکلی متفاوت، برای مثال یکی از نظر زمان بهینه است و دیگری از نظر فضا. ما می‌خواهیم در زمان اجرا، بر اساس شرایط سیستم یا ترجیحات مشتری، بتوانیم یکی از این الگوریتم‌ها را انتخاب و استفاده کنیم. مثلا اگر فشار روی پردازنده زیاد باشد، الگوریتمی که از نظر زمان بهینه است انتخاب می‌شود. اگر فضای ذخیره‌سازی محدود باشد، الگوریتمی که حافظه‌ی کمتری مصرف می‌کند انتخاب می‌شود. هدف Strategy این است که انتخاب و تعویض الگوریتم‌ها در زمان اجرا بدون نیاز به تغییر در مشتری‌ها انجام شود. کاهش جفت‌شدگی بین الگوریتم‌ها و مصرف‌کنندگان آن‌ها صورت گیرد. امکان گسترش راحت خانواده‌ی الگوریتم‌ها فراهم باشد بدون اینکه به سایر قسمت‌های سیستم آسیبی برسد.

مقایسه: اهداف الگوها به صورت مفصل شرح داده شدند و ارتباط آن‌ها در قسمت الگوهای مرتبط بیان شده است.

۳-۴ حوزه

Decorator: این الگو در حوزه شی[۶] قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

Proxy: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

Strategy: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

---

[۶] object scope

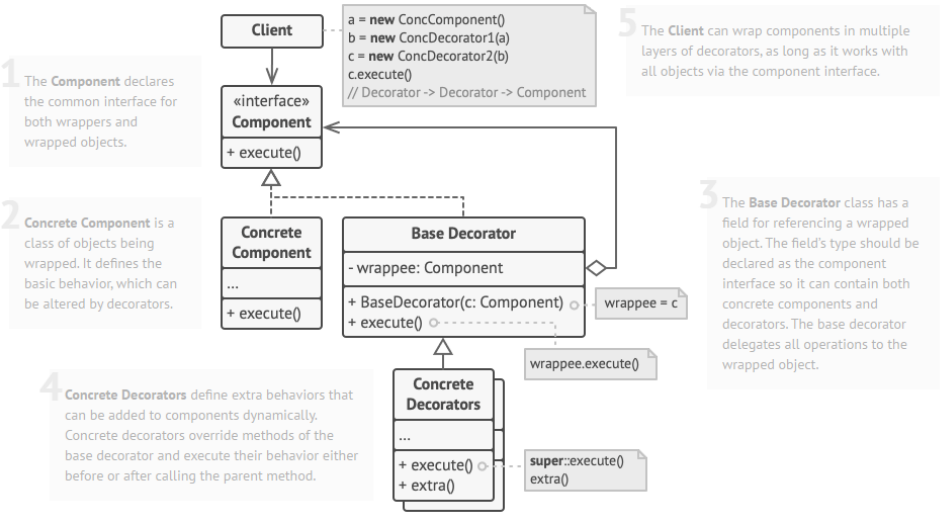
مقایسه: هر سه الگو در حوزه شی قرار دارند.

۴-۴ ساختار و رفتار

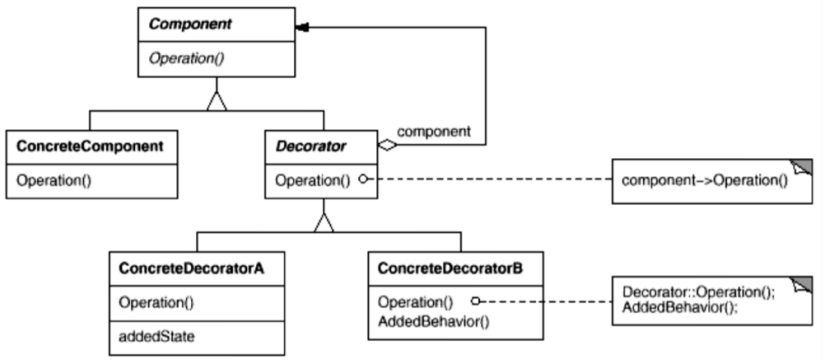
Decorator: ساختار این الگو در شکل ۴-۱ و ۴-۲ قابل مشاهده است. در این ساختار، مشتری فقط با واسط Component کار می‌کند و از جزئیات داخلی بی‌خبر است. در انتهای زنجیره، ConcreteComponent قرار دارد و روی آن با استفاده از Decoratorهای سلسله‌وار رفتارهای جدید افزوده می‌شود. خود Decorator یک کلاس انتزاعی است که زیرکلاس‌های کانکریت آن، رفتارهای خاص را پیاده‌سازی می‌کنند. رفتارها می‌توانند به‌صورت پیش‌پردازش[۷] یا پس‌پردازش[۸] اضافه شوند. این ساختار یکی از قوی‌ترین انواع wrapper هاست، زیرا امکان افزودن یا حذف دینامیک رفتارها در زمان اجرا را می‌دهد، آن هم بدون اینکه مشتری از این تغییرات آگاه باشد. یک پیکربند ثالث وجود دارد که آن می‌تواند این زنجیره را پیکربندی کرده و عنصر اول زنجیره را در اختیار مشتری قرار دهد. می‌شود از وسط زنجیره دیدی را برای یک مشتری برقرار کرد ولی نوعا این خیلی رایج نیست. معمولا به این شکل است که زنجیره در تمامیتش در اختیار مشتری‌ها قرار میگیرد. همچنین در این ساختار، الگوی Composite مشاهده می‌شود. توجه شود که روی فلش ارتباط Decorator با Component کلمه مفرد component ذکر شده است یعنی یک کامپوننت و این درواقع یک زنجیره را تشکیل می‌دهد. در شکل ۴-۱ که از منبع [۱] استخراج شده است، مشتری خودش پیکربندی اولیه را انجام می‌دهد و بدین شکل به تمام زیرکلاس‌های دکوریتور دید دارد اما این یکی از مزیت‌های مد نظر را خراب می‌کند یعنی منتزع کردن مشتری از تغییرات رفتارهای جدید، بنابراین بهتر است یک پیکربند ثالث این عمل را انجام داده و سر زنجیره را در اختیار مشتری قرار دهد.

Proxy: ساختار این الگو در شکل ۴-۳ و ۴-۴ و یک نمودار شی[۹] در شکل ۴-۵ قابل مشاهده است. پراکسی همان واسطی را پیاده‌سازی می‌کند که مشتری از شی اصلی انتظار دارد. این واسط در Subject تعریف شده و پراکسی نیز زیرکلاسی از آن است تا بتواند رفتاری مشابه شی اصلی داشته باشد و مشتری تفاوتی احساس نکند. هنگام دریافت درخواست از مشتری، پراکسی آن را به شی اصلی ارجاع می‌دهد، اما ممکن است قبل از آن عملیات اضافی مانند کنترل دسترسی یا به تعویق انداختن بارگذاری را انجام دهد. پس پراکسی صرفا یک واسطه‌ی ساده نیست و تا زمانی که شی اصلی بارگذاری نشده، نقش فعالی دارد. اما پس از آن صرفا درخواست‌ها را منتقل می‌کند. در برخی انواع پراکسی مانند virtual proxy، خود پراکسی وظیفه‌ی ساخت شی اصلی را هم بر عهده دارد. اما در سایر انواع، ممکن است شی اصلی از قبل وجود داشته باشد و نیازی به

[۷] pre-processing
[۸] post-processing
[۹] object diagram



شکل ۴-۱: ساختار الگوی Decorator [۱]

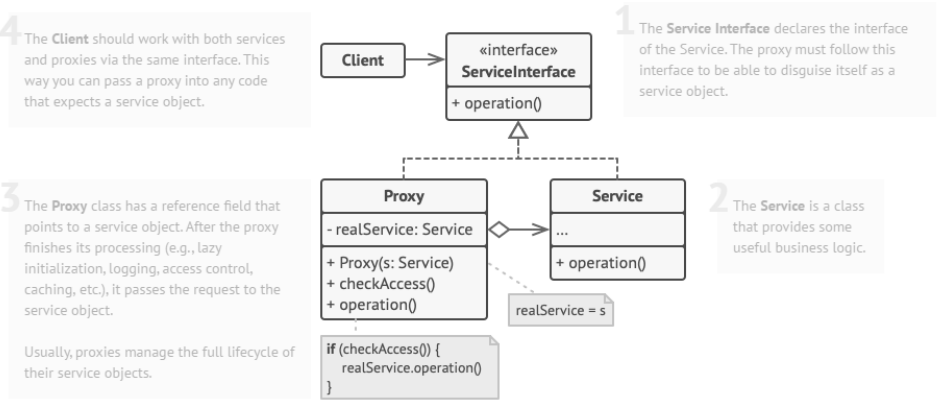


شکل ۴-۲: ساختار الگوی Decorator در کتاب GoF [۲]

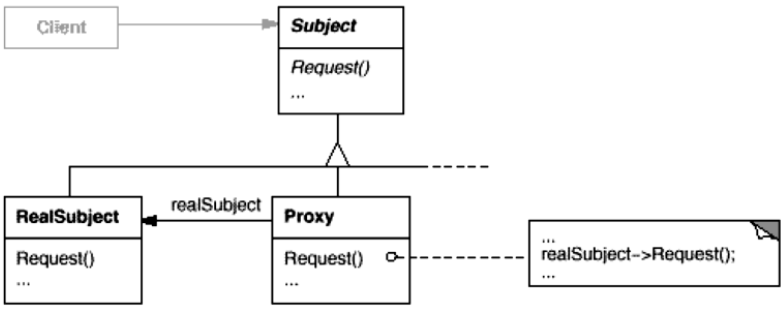
ساخت آن توسط پراکسی نباشد. بنابراین، همیشه یک کمان خط‌چین برای نشان دادن رابطه‌ی نمونه‌گیری[۱۰] میان پراکسی و شی اصلی در نمودار وجود ندارد، ولی در virtual proxy چنین رابطه‌ای باید نمایش داده شود. اما در همه‌ی موارد، پراکسی باید پس از ساخت، یک ارجاع مانا به شی اصلی نگه دارد، که با رابطه association در نمودار نمایش داده می‌شود. در نهایت، خود مشتری هیچگاه از وجود زیرکلاس‌ها آگاه نیست و نمی‌تواند نمونه‌ای از آن‌ها بسازد. وظیفه‌ی ساخت و پیکربندی این ساختار شی بر عهده‌ی یک پیکربند ثالث در سیستم است، که ممکن است هر بخش از سیستم باشد، به جز خود مشتری.

Strategy: ساختار این الگو در شکل ۴-۶ و ۴-۷ قابل مشاهده است. ساختار الگوی استراتژی مشابه الگوی State است. در این الگو نیز یک Context وجود دارد که یک شی از نوع Strategy به آن منتسب می‌شود. پیاده‌سازی‌های مختلف الگوریتم در قالب زیرکلاس‌های ConcreteStrategy ارائه می‌شوند،

[۱۰] instantiation



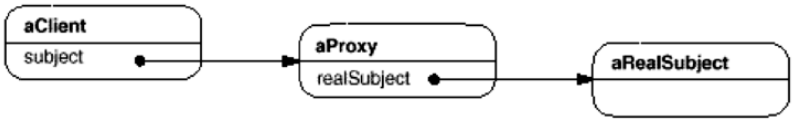
شکل ۴-۳: ساختار الگوی Proxy [۱]



شکل ۴-۴: ساختار الگوی Proxy در کتاب GoF [۲]

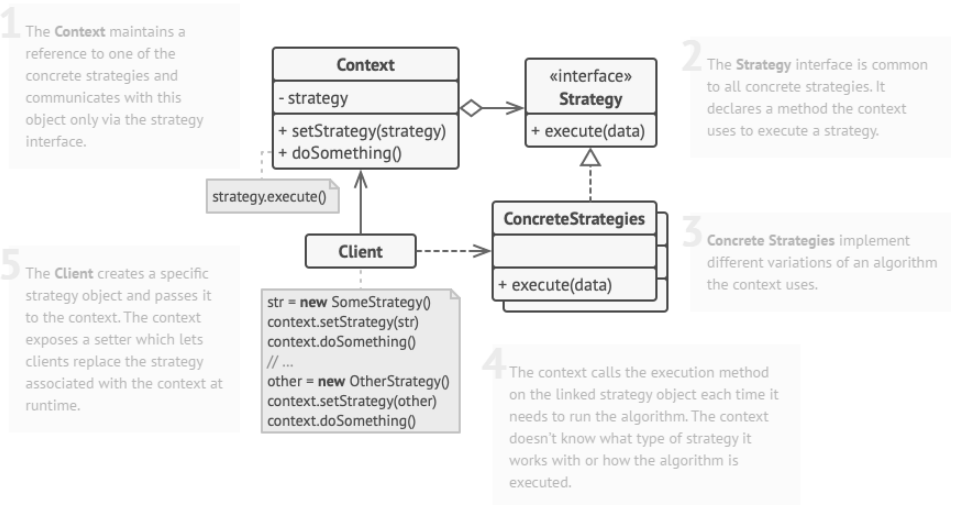
و در نهایت یکی از آن‌ها باید در اختیار Context قرار گیرد. زمان تخصیص یا تغییر استراتژی به عوامل مختلفی بستگی دارد. مثل شرایط سیستم، ترجیحات کاربر یا منطق مشتری داخلی و هیچ محدودیتی از این لحاظ وجود ندارد. در واقع، آنچه در Strategy تغییر می‌کند، الگوریتم است. یک چالش مهم در این الگو، طراحی واسط Strategy است. این واسط باید امضای عملیاتی را در بر گیرد که برای پیچیده‌ترین استراتژی لازم‌اند. اگرچه بسیاری از استراتژی‌ها ساده‌اند و به چنین اطلاعاتی نیازی ندارند، اما ناچارند همه پارامترها را دریافت کنند. چرا؟ چون واسط پایه باید تمام نیازهای ممکن در زیرکلاس‌ها را پوشش دهد. گاهی مجبوریم اجتماع[۱۱] واسط‌های زیرکلاس‌ها را در کلاس پایه لحاظ کنیم. این موضوع باعث ایجاد سربار می‌شود، زیرا برخی استراتژی‌ها پارامترهایی دریافت می‌کنند که هرگز استفاده نمی‌کنند. برای کاهش این سربار، گاهی مانند الگوی State عمل می‌شود یعنی به‌جای ارسال پارامترهای متعدد، خود Context به استراتژی پاس داده می‌شود تا استراتژی بتواند مستقیما از وضعیت و داده‌های Context استفاده کند. این راه‌حل نیز بسیار رایج است. نکته‌ای که باید به آن توجه داشت این است که پیکربند یعنی مشتری، باید تمام این زیرکلاس‌ها را بشناسد و از نقش آن‌ها آگاه باشد. این وابستگی یکی از نقاط ضعف این الگو محسوب می‌شود، ولی معمولا اجتناب‌ناپذیر

[۱۱] union

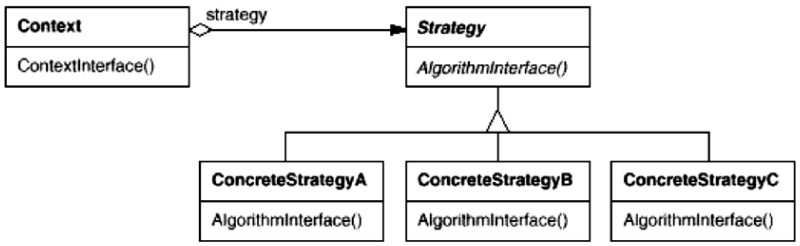


شکل ۴-۵: نمودار شی الگوی Proxy [۲]

است. زیرا مشتری ناچار است برای انتخاب و نمونه‌گیری صحیح، اطلاعات لازم را داشته باشد. گاهی این انتخاب بر اساس ورودی کاربر یا ترجیح یک عامل بیرونی انجام می‌شود. همچنین، علاوه بر پیکربندی اولیه، امکان بازپیکربندی الگوریتم در زمان اجرا نیز وجود دارد. نکته‌ی مهم دیگر آن است که تمام ویژگی‌ها و عملیات فرعی مورد نیاز هر الگوریتم، داخل همان شی Strategy مربوطه قرار می‌گیرد. بنابراین وظیفه‌ی نگهداری و مدیریت این موارد از دوش Context برداشته می‌شود.



شکل ۴-۶: ساختار الگوی Strategy [۱]



شکل ۴-۷: ساختار الگوی Strategy در کتاب GoF [۲]

۴-۵ جفت‌شدگی

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط است و DIP برقرار بوده و مشتری از جزئیات داخل بی‌خبر است. تفاوتی میان شی اصلی و دکوریتور قائل نیست و نمی‌شناسد و از تغییرات داخل مجموعه منتزع شده است. در نتیجه در این قسمت جفت‌شدگی بسیار مطلوب است. ولی پیکربند ثالث برای پیکربندی و ساخت این زنجیره، نیاز است که زیرکلاس‌ها را بشناسد تا بتواند زنجیره را تشکیل داده و سر زنجیره را در اختیار مشتری قرار دهد. بنابراین در این قسمت جفت‌شدگی بالا است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته و جفت‌شدگی در این قسمت نیز بالا است. در برخی انواع پراکسی مانند Remote Proxy این الگو کمک می‌کند تا مشتری از جزئیات پیاده‌سازی ارتباط از راه دور مانند پرتکل‌ها و غیره نیز بی‌خبر شده و جفت‌شدگی به آن‌ها نیز کاهش می‌یابد.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی

به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود.

مقایسه: هر سه در قسمت‌های بیان شده با توجه به توضیحات شرح داده شده سعی در کنترل جفت‌شدگی دارند.

۶-۴ انسجام

Decorator: در این الگو، از feature-laden class که خطر تبدیل به God Class داشته و قابلیت‌ها را یک‌جا جمع کرده بود و همچنین از ساختارهای توارثی ترکیبی پرهیز شده است و رفتارها در اشیای مخصوص تمرکز پیدا کرده و این رفتار چه به صورت پیش‌پردازش و چه به صورت پس‌پردازش در زنجیره تشکیل شده اجرا می‌شوند و این رفتار داخل خود شی اصلی نیست. مشتری نیز از آن‌ها منتزع شده است. بنابراین انسجام افزایش یافته و مطلوب است.

Proxy: در Remote Proxy وظیفه پیاده‌سازی مکانیزم‌های ارتباط از راه دور از دوش مشتری برداشته شده و در پراکسی پیاده‌سازی شده‌اند. همچنین در Protection Proxy وظیفه کنترل دسترسی به شی پراکسی سپرده شده است. در کل مشتری از انجام مسئولیت‌های زائد رها شده و در اشیای مخصوص برخی مسئولیت‌ها متمرکز شده‌اند و این به افزایش انسجام کمک می‌کند.

Strategy: در این الگو، رفتار مربوط به الگوریتم‌ها در استراتژی‌ها متمرکز شده و از کانتکست خارج است. در نتیجه کانتکست از عملیات اضافی رها شده است. هر استراتژی نیز رفتار مختص خود را دارد. بنابراین انسجام در این الگو مطلوب است.

مقایسه: هر سه الگو بنابر توضیحات داده شده سعی در کنترل انسجام دارند.

۷-۴ پیکربندی

Decorator: یک پیکربند ثالث وجود دارد که آن می‌تواند این زنجیره را پیکربندی کرده و عنصر اول زنجیره را در اختیار مشتری قرار دهد تا الگو محقق شود. می‌شود از وسط زنجیره دیدی را برای یک مشتری برقرار کرد ولی نوعا این خیلی رایج نیست. معمولا به این شکل است که زنجیره در تمامیتش در اختیار مشتری‌ها قرار میگیرد. این پیکربند ثالث به مجموعه دید داشته و می‌تواند پیکربندی و بازپیکربندی در زمان اجرا انجام دهد.

Proxy: به صورت کلی یک پیکربند ثالث وظیفه نمونه‌گیری از شی اصلی (در صورتی که از قبل موجود

نباشد) و پراکسی و تشکیل ساختار delegation در زمان اجرا را داشته و مشتری از پیکربندی و شناخت زیرکلاس‌ها منتزع شده است. در Virtual Proxy، شی اصلی در دیرترین زمان ممکن توسط خود شی پراکسی نمونه‌گیری می‌شود ولی خود پراکسی توسط پیکربند نمونه‌گیری شده و مشتری با آن پیکربندی می‌شود.

Strategy: در این الگو، یک پیکربند ثالث که می‌تواند مشتری نیز باشد، آگاهی کامل به زیرکلاس‌های استراتژی داشته و نمونه‌گیری‌های مناسب را انجام و به کانتکست تخصیص می‌دهد و بدین شکل در زمان اجرا ساختار delegation ایجاد شده و الگو محقق می‌شود.

مقایسه: هر سه الگو به پیکربند نیاز دارند.

۸-۴ انعطاف‌پذیری

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط است و DIP برقرار بوده و مشتری از جزئیات داخل مجموعه منتزع شده و تغییرات آن‌ها به مشتری منتشر نمی‌شود. مشتری صرفا عنصر اول زنجیره را دیده و پیغام Operation را به آن می‌فرستد. رفتارهای اضافه شده به صورت پیش‌پردازش یا پس‌پردازش توسط دکوریتورها در این زنجیره اجرا می‌شوند. این زنجیره کاملا انعطاف‌پذیر بوده و می‌توان دکوریتورهایی اضافه یا کم کرد که در نتیجه آن رفتار اضافه یا کم می‌شود بدون آن که مشتری تاثیر بپذیرد. نیاز به پیش‌بینی قابلیت‌های مختلف از قبل نیز وجود نداشته و کارها از طریق واسپاری انجام می‌شود. در نتیجه انعطاف‌پذیری افزایش یافته است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط است و از تغییرات داخل مجموعه منتزع شده است. در نتیجه می‌توان انواع پراکسی به عنوان زیرکلاس سابجکت اضافه کرده و رفتارهای مدنظر را به شکل‌هایی که قبلا توضیح داده شد اضافه کرد که مشتری از وجود آن‌ها آگاه نخواهد بود و تاثیر نمی‌پذیرد. وجود یک سطح indirection بین مشتری و شی اصلی این انعطاف را محیا کرده است. تا زمانی که واسط سابجکت تغییر نکند، مشتری تاثیر نمی‌پذیرد.

Strategy: در این الگو، کانتکست از طریق واسط با استراتژی ارتباط داشته و DIP برقرار بوده و در نتیجه OCP نیز برقرار است که می‌توان استراتژی‌های متعددی را اضافه کرد بدون این که کانتکست تاثیر بپذیرد. در زمان اجرا می‌توان استراتژی مناسب را به کانتکست تخصیص داد تا با واسپاری، عملیات مربوطه انجام شود. همچنین در زمان اجرا امکان بازپیکربندی با استراتژی دیگر نیز وجود دارد. در نتیجه انعطاف‌پذیری در این الگو مطلوب است.

مقایسه: الگوها بنابر توضیحات ارائه شده در هر بخش اقدام به کنترل انعطاف‌پذیری می‌کنند.

۹-۴ کارایی

Decorator: در سیستم‌های مبتنی بر الگوی Decorator، تعداد زیادی شی کوچک در زمان اجرا تولید می‌شود. هر شی اصلی ممکن است چندین دکوریتور داشته باشد، و این باعث می‌شود تعداد کل اشیا زیاد شود که به منابع سیستمی فشار وارد می‌کند. همچنین، وقتی یک عملیات روی شی انجام می‌شود، این عملیات باید بین چندین شی پیاپی (دکوریتورها) رد و بدل شود که باعث افزایش پیچیدگی و کاهش کارایی می‌شود. در عوض، این ساختار بسیار انعطاف‌پذیر است. در نتیجه ما در این الگو انعطاف‌پذیری را به دست می‌آوریم و بخشی از کارایی را از دست می‌دهیم. با این حال، در بسیاری از سیستم‌ها این هزینه قابل قبول است چون تعداد اشیا و دکوریتورها در عمل محدود هستند و مزایای انعطاف‌پذیری بیشتر از هزینه منابع می‌شود. این الگو یک wrapper معروف و پرکاربرد است، اما در سیستم‌های بزرگ و data-intensive که تعداد اشیای دکوریت شده زیاد است، استفاده زیاد از آن می‌تواند فشار زیادی به منابع سیستم وارد کند. در چنین شرایطی بهتر است استفاده از دکوریتور محدود شود به بخش‌هایی که ترکیب‌های زیاد رفتاری نیاز است و در باقی موارد از وراثت استفاده شود.

Proxy: در این الگو، پراکسی یک سطح indirection بین مشتری و شی اصلی ایجاد می‌کند و این به معنی افزایش تعداد اشیا در زمان اجرا و سربار حافظه و همچنین افزایش تعداد پیام‌ها و در کل کاهش اندکی کارایی است. اما باید توجه کرد که Virtual Proxy در دیرترین زمان ممکن شی اصلی را بارگذاری می‌کند و تا قبل از آن خود پاسخ مشتری را می‌دهد و این کمک بسیاری به کاهش سربار حافظه می‌کند و فقط در دیرترین زمان ممکن شی اصلی را بارگذاری کرده و از آن لحظه به بعد صرفا واسپاری انجام می‌دهد و پس از آن لحظه، باعث کاهش اندکی کارایی می‌شود. در Smart Reference نیز زمانی که تعداد ارجاعات به شی اصلی صفر شود، شی از حافظه پاک می‌شود و این به مراقبت از مصرف بیهوده حافظه کمک می‌کند.

Strategy: در الگوی استراتژی یک سربار ارتباطی وجود دارد که هم به دلیل وجود indirection و نیاز به واسپاری است (که موجب کاهش عملکرد نسبت به حالت قبلی می‌شود)، و هم به دلیل واسط سنگینی که همه استراتژی‌ها باید رعایت کنند. در این وضعیت، پارامترهایی به متدها ارسال می‌شوند که برخی از آن‌ها برای استراتژی جاری بی‌استفاده‌اند، چون Context نمی‌داند دقیقا کدام استراتژی اجرا خواهد شد. راه‌حل نسبی، برقراری دید از استراتژی به Context است، یعنی Strategy به‌جای گرفتن پارامترهای زیاد، به خود Context دسترسی داشته باشد و فقط اطلاعات مورد نیاز را از آن دریافت کند. البته این روش همیشه کافی نیست، چون ممکن است برخی اطلاعات اساسا در Context نباشند و باید از بیرون بیایند. در نتیجه، مشکل ممکن است کاهش یابد ولی به‌طور کامل حل نشود. همچنین تعداد زیادی شی در زمان اجرا ساخته می‌شود، چون ممکن است برای هر متد از کانتکست یک استراتژی جدا تعریف شود. این موضوع باعث افزایش مصرف

منابع سیستمی می‌شود. این مشکل مشابه چیزی است که در الگوی Decorator هم وجود دارد، ولی الگوی State معمولا این مسئله را ندارد چون لازم نیست برای هر متد یک شی جدا تعریف شود. همچنین، اشیای Strategy معمولا ریزدانه‌تر از اشیای State هستند.

مقایسه: بنابر توضیحات ارائه شده هر سه الگو کارایی را کاهش می‌دهند.

۴-۱۰ شی جعلی

Decorator: در اثر اعمال این الگو، دکوریتورها که wrapper هستند شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

Proxy: در اثر اعمال این الگو، پراکسی‌ها که wrapper هستند شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

Strategy: در اثر اعمال این الگو، استراتژی‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

مقایسه: هر سه اشیای جعلی تعریف می‌کنند.

۴-۱۱ انتشار تغییرات

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط بوده و DIP برقرار است در نتیجه OCP برقرار بوده و مشتری از جزئیات داخل مجموعه خبر ندارد که باعث می‌شود تغییرات زیرکلاس‌ها به مشتری منتشر نشود. پیکربند در زمان اجرا یک زنجیره از دکوریتورها و شی اصلی ساخته و سر زنجیره را در اختیار مشتری قرار می‌دهد تا با فراخوانی متد داخل واسط کامپوننت، این زنجیره طی شده و رفتارهای اضافه نیز همراه با رفتار اصلی اجرا شوند. اما مشتری از تغییرات داخل این زنجیره منتزع است. ولی پیکربند که باید این زنجیره را پیکربندی یا بازپیکربندی کند، به تمام زیرکلاس‌ها دید داشته و تغییرات آن‌ها به پیکربند منتشر می‌شود. البته از آن‌جایی که مشتری باید همیشه به ابتدای زنجیره دید داشته باشد، اگر عنصر ابتدای زنجیره تغییر کرد، این مورد باید در مشتری تاثیر گذاشته و مطلع شود تا به عنصر جدید سر زنجیره اشاره کند.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت

تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است و پراکسی از تغییرات شی اصلی تاثیر می‌پذیرد. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته، جفت‌شدگی در این قسمت نیز بالا رفته و از تغییرات آن‌ها تاثیر می‌پذیرد. همچنین تغییر در واسط سابجکت باعث انتشار تغییرات به زیرکلاس‌ها از جمله پراکسی و شی اصلی شده و آن‌ها تاثیر می‌پذیرند.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود. در صورت تغییر واسط استراتژی نیز این تغییر به کانتکست و زیرکلاس‌های استراتژی منتشر می‌شود.

مقایسه: هر سه الگو در قسمت‌های شرح داده شده با استفاده از روش‌های بیان شده اقدام به کنترل انتشار تغییرات می‌کنند.

۱۲-۴ سهولت پیاده‌سازی

Decorator: با استفاده از یک زبان شی‌گرا می‌توان این الگو را بدون مشکل پیاده‌سازی کرد. نیاز به تعریف واسط‌ها یا کلاس‌های انتزاعی داشته و کلاس‌های عینی برای تحقق آن‌ها باید تعریف شوند. همچنین رابطه composition نیز پیاده‌سازی شود. در کل پیاده‌سازی مشکل خاصی ندارد.

Proxy: این الگو بسته به نوع پراکسی، پیاده‌سازی‌های مختلفی دارد ولی در کل پیاده‌سازی آن در یک زبان شی‌گرا پیچیده نیست. نیاز به تعریف واسط سابجکت و برقراری روابط مطرح شده در شکل ساختار این الگو وجود دارد که به آسانی قابل انجام هستند. البته برخی انواع پراکسی مانند Remote Proxy که نیاز به پیاده‌سازی مکانیزم‌های ارتباط از راه دور هستند، نیاز به پیاده‌سازی ملاحظات مربوطه مانند شبکه در آن وجود دارد.

Strategy: با استفاده از یک زبان شی‌گرا می‌توان این الگو را بدون مشکل پیاده‌سازی کرد. نیاز به تعریف واسط‌ها یا کلاس‌های انتزاعی داشته و کلاس‌های عینی برای تحقق آن‌ها باید تعریف شوند مطابق ساختاری که قبلا شرح داده شد.

۱۳-۴ موارد کاربرد

Decorator:

- در این الگو، می‌توان مسئولیت‌ها را به‌صورت پویا در زمان اجرا اضافه یا کم کرد و این کار به صورت شفاف[۱۲] برای مشتری انجام می‌شود. یعنی مشتری متوجه تغییرات زنجیره نمی‌شود. تنها نکته مهم این است که مشتری همیشه باید به ابتدای زنجیره اشاره کند. اگر آدرس عنصر ابتدایی تغییر کند، باید این تغییر به مشتری منتقل شود. در غیر این صورت، تغییرات داخلی زنجیره تاثیری بر بیرون ندارد و سیستم به صورت انعطاف‌پذیر کار می‌کند.
- امکان حذف مسئولیت وجود داشته باشد.
- این الگو مخصوصا وقتی تعداد زیادی ترکیب رفتاری مختلف داریم، بسیار مؤثر است و جایگزین بهتری از وراثت محسوب می‌شود. چون در وراثت، با اضافه شدن هر ویژگی جدید، باید تعداد زیادی زیرکلاس جدید بسازیم. اما در Decorator فقط کافی است یک دکوریتور جدید تعریف کنیم و نیازی به تولید ترکیبات مختلف به شکل زیرکلاس نیست.

Proxy:

- وقتی نیاز به یک جایگزین یا نایب برای یک شی وجود دارد، بسته به دلیل این نیاز، انواع مختلفی از الگوی پراکسی به‌وجود می‌آید. در ادامه، چهار نوع پراکسی توضیح داده می‌شوند:
  - Remote Proxy: زمانی استفاده می‌شود که شی اصلی در یک مکان دور (مثلا روی گره دیگر در شبکه) قرار دارد. در این حالت، یک پراکسی محلی به‌عنوان نماینده‌ی آن در اختیار مشتری قرار

[۱۲] transparent

می‌گیرد، به‌طوری که مشتری تصور می‌کند با شی اصلی در حال تعامل است. این پراکسی وظیفه‌ی مدیریت ارتباط شبکه‌ای با شی اصلی و انتقال درخواست‌ها و پاسخ‌ها را بر عهده دارد و دغدغه‌ی این ارتباط را از دوش مشتری برمی‌دارد.

- Virtual Proxy: برای مدیریت منابع سیستم، به‌ویژه حافظه، از این نوع پراکسی استفاده می‌شود. اشیایی که ساخت آن‌ها هزینه‌بر است، تنها در صورت نیاز واقعی و در آخرین لحظه ممکن (lazy loading) ساخته می‌شوند. تا آن زمان، پراکسی به‌جای شی اصلی عمل کرده و برخی اطلاعات یا پاسخ‌ها را خودش ارائه می‌دهد.
- Protection Proxy: این نوع پراکسی برای کنترل دسترسی طراحی شده است. با بررسی سطح دسترسی مشتری‌ها، مشخص می‌کند که آیا اجازه‌ی تعامل با شی اصلی را دارند یا خیر و بدین ترتیب از دسترسی غیرمجاز جلوگیری می‌کند.
- Smart Reference: این نوع پراکسی مانند یک اشاره‌گر باهوش‌تر است که در هنگام دسترسی به شی، عملیات اضافی انجام می‌دهد. این عملیات می‌تواند شامل reference counting (برای آزادسازی خودکار منابع زمانی که دیگر نیازی به شی نیست)، lazy loading (اما بدون نقش پاسخ‌دهی مستقیم مانند virtual proxy)، یا locking (برای مدیریت دسترسی هم‌زمان در محیط‌های چندنخی) باشد. برخلاف virtual proxy، smart reference نقش پاسخ‌دهی ندارد و ادای شی اصلی را درنمی‌آورد.

:Strategy

- در بسیاری از موارد، تعداد زیادی کلاس وجود دارند که تنها در رفتار با یکدیگر تفاوت دارند، در حالی که واسط آن‌ها یکسان است. الگوی استراتژی این امکان را فراهم می‌کند که به‌جای تعریف کلاس‌های متعدد، تنها یک کلاس داشته باشیم و رفتار آن را در زمان اجرا پیکربندی کنیم. به این صورت که برای هر نمونه از کلاس، مجموعه‌ای از اشیای Strategy تخصیص داده می‌شود و هر متد، برای اجرای وظیفه‌اش به شی Strategy مربوطه پیام می‌فرستد. در نتیجه، به‌جای داشتن n کلاس با رفتارهای مختلف، یک کلاس عمومی با واسط مشترک تعریف می‌شود که در لحظه‌ی مقداردهی اولیه با رفتار دلخواه پیکربندی می‌گردد. این کار با نسبت دادن اشیای Strategy به آن انجام می‌شود. در ظاهر، انگار از کلاس‌های مختلف قبلی نمونه گرفته‌ایم، اما در واقع همه‌چیز در قالب یک کلاس واحد و قابل پیکربندی خلاصه شده است. البته باید توجه داشت که این استفاده، کاربرد اصلی الگوی استراتژی نیست، اما مزیتی مهم در ساده‌سازی ساختار کلاس‌ها به شمار می‌رود.
- زمانی که ما به انواع مختلفی از الگوریتم نیاز داریم مثلا الگوریتم‌هایی که از نظر بهینه بودن فضا یا زمان با هم متفاوت هستن و ما آن‌ها را تعریف میکنیم برای اینکه هر زمان که لازم بود بین آن‌ها انتخاب کرده

و در زمان اجرا تعویضشان کنیم.

- گاهی یک الگوریتم از داده‌هایی استفاده می‌کند که نباید برای مشتری‌ها قابل مشاهده باشند. اگر این داده‌ها درون Context قرار گیرند، Context سنگین می‌شود. اما با انتقال این داده‌ها به درون اشیای Strategy، هم پیچیدگی پنهان می‌ماند و هم بار حافظه‌ای صرفا محدود به خود Strategy می‌گردد. در حالی که Context ممکن است به‌تنهایی یک کلاس بزرگ باشد، اشیای Strategy معمولا کلاس‌هایی کوچک‌اند که تنها رفتار و داده‌های مربوط به همان الگوریتم را در خود نگه می‌دارند. اگر قرار بود همه ویژگی‌ها و عملیات خاص تمام الگوریتم‌ها را داخل Context بگنجانیم، این منجر به پیچیدگی و بار اضافی می‌شد. ولی با تفکیک الگوریتم‌ها به درون Strategyها، هم ساختار Context سبک‌تر می‌شود، و هم جزئیات داخلی الگوریتم‌ها پنهان و محلی باقی می‌مانند.
- در برخی کلاس‌ها، مجموعه‌ای از رفتارهای مختلف وجود دارد که به‌صورت دستورات شرطی پیچیده سوئیچ در عملیات ظاهر می‌شوند. در این حالت، یک متد سنگین تمامی الگوریتم‌های ممکن را در قالب شاخه‌های مختلف در خود جای داده است، که نه‌تنها خوانایی را کاهش می‌دهد، بلکه نگهداری را نیز دشوار می‌سازد. با به کارگیری الگوی استراتژی، این رفتارها به صورت پارتیشن شده به اشیای Strategy و زیرکلاس‌های مربوطه منتقل می‌شوند و در نتیجه، پیچیدگی متدها به‌طور چشمگیری کاهش می‌یابد. افزون بر این، در زمان اجرا نیز می‌توان به سادگی رفتارها را بدون استفاده از سوئیچ تغییر داد.

الگوی استراتژی معمولا در سطح یک متد به کار می‌رود، به طوری که یک عملیات مشخص با الگوریتم‌های مختلف پیاده‌سازی می‌شود. اما گاهی یک مجموعه عملیات در قالب یک رفتار مشترک قرار می‌گیرند و در این صورت، چندین متد در Context و چندین پیاده‌سازی متناظر در Strategy خواهیم داشت. در این حالت، به جای استفاده از ساختارهای شرطی پیچیده، رفتار به شکل مستقل درون اشیای Strategy منتقل می‌شود. گرچه از نظر ساختار، الگوی Strategy شباهت زیادی به State دارد، اما از نظر هدف با آن متفاوت است. در الگوی State، انتخاب رفتار به وضعیت داخلی خود شی Context وابسته است، ولی در Strategy، انتخاب الگوریتم ممکن است به عوامل خارجی مانند شرایط سیستم یا ترجیح کاربر وابسته باشد. بنابراین تغییر رفتار در Strategy لزوما به وضعیت داخلی شی بستگی ندارد.

مقایسه: توضیحات مفصل درباره موارد کاربرد هر الگو بیان شد و ارتباط آن‌ها در قسمت الگوهای مرتبط شرح داده شده است.

۱۴-۴ کپسوله‌سازی

Decorator: در این الگو، ارتباط بین مشتری و بقیه مجموعه از طریق واسط کامپوننت است و به داخل مجموعه دید ندارد. دکوریتورها نیز با واسط کامپوننت رابطه دارند در نتیجه دید در سطح بالا بوده و representation و مشخصه‌های داخلی عناصر به بیرون نشر پیدا نمی‌کند پس کپسوله‌سازی رعایت شده است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط است و از جزئیات داخل مجموعه شامل پراکسی‌ها و شی اصلی بی‌خبر است و نمایش داخلی آن‌ها به مشتری درز نمی‌شود. اما پراکسی رابطه مستقیم و دید مانا با شی اصلی دارد و عملیات را به آن واسپاری می‌کند. در صورت پنهان نکردن جزئیات داخلی، پراکسی می‌تواند به این موارد در شی اصلی دسترسی داشته باشد و کپسوله‌سازی نقض شود. اما عمدتا پراکسی صرفا عملیاتی که در واسط سابجکت تعریف شده است را به شی اصلی واسپاری می‌کند و می‌توان ساختار داخلی آن را پنهان کرد.

Strategy: در این الگو، کانتکست از طریق واسط با استراتژی در ارتباط بوده و از جزئیات داخل و نمایش داخلی ناآگاه است و در زمان اجرا عملیات را به استراتژی که با آن منتسب شده است واسپاری می‌کند. پس در این قسمت کپسوله‌سازی رعایت شده است. اما اگر کانتکست هنگام واسپاری رفتار به استراتژی، خود را نیز ارسال کرده و استراتژی به کانتکست دید پیدا کند (بنابر دلیلی که قبلا توضیح داده شد)، در این صورت کپسوله‌سازی نقض می‌شود چون استراتژی داده‌های مورد نیاز خود را مستقیما از کانتکست می‌گیرد.

مقایسه: در الگوی اول رعایت شده و در دو الگوی دیگر در قسمت‌های شرح داده شده نقض می‌شود.

۱۵-۴ جداسازی دغدغه‌ها

Decorator: در این الگو، دکوریتورها مسئول پیاده‌سازی رفتار اضافه‌ای هستند که به شی اصلی اضافه می‌شود. یعنی یک پیکربند یک زنجیره‌ای از دکوریتورها و شی اصلی تشکیل می‌دهد و رفتار اضافه توسط دکوریتورها به صورت پیش‌پردازش یا پس‌پردازش اجرا می‌شود که این اجرا از طریق واسپاری است. در نتیجه رفتارهای متعدد داخل خود شی اصلی تعریف نشده‌اند و از ساختارهای وراثتی ترکیبی و feature-laden class پرهیز شده است و اکنون جداسازی دغدغه‌ها به خوبی دیده می‌شود.

Proxy: در Remote Proxy وظیفه پیاده‌سازی مکانیزم‌های ارتباط از راه دور شبکه و غیره از دوش مشتری برداشته شده و در پراکسی پیاده‌سازی شده‌اند. همچنین در Protection Proxy وظیفه کنترل دسترسی به شی پراکسی سپرده شده است. Smart Reference مانند یک اشاره‌گر باهوش‌تر است که در هنگام دسترسی به شی، عملیات اضافی انجام می‌دهد. درکل مشتری از انجام مسئولیت‌های زائد رها شده و در اشیای مخصوص

برخی مسئولیت‌ها متمرکز شده‌اند و این به جداسازی دغدغه‌ها کمک می‌کند.

Strategy: در این الگو، الگوریتم‌های مختلف در استراتژی‌های مخصوص متمرکز شده و کانتکست دغدغه پیاده‌سازی آن‌ها را ندارد. کانتکست رفتار را به استراتژی مربوط به خود واسپاری می‌کند و آن استراتژی پیاده‌سازی الگوریتم مخصوص خود را شامل می‌شود. پس جداسازی دغدغه‌ها به خوبی رعایت شده است.

مقایسه: هر سه الگو از روش‌هایی که شرح داده شده است برای جداسازی دغدغه‌ها استفاده می‌کنند.

۱۶-۴ مزایا و معایب (تبعات)

Decorator:

مزایا:

- از وراثت ایستا خیلی بهتر است زیرا همه ترکیبات ممکن را لازم نیست پیشبینی کرده و برایشان زیرکلاس تعریف کنیم و این که قابلیتی به یک شی اضافه یا حذف شود به معنی این نخواهد بود که باید کلاس آن را عوض کرد. یعنی شی را کشت و یک شی از کلاس جدید آورد.
- یکی از جایگزین‌ها feature-laden class است که قبلا توضیح داده شد. یعنی یک کلاس بسیار بزرگ که تمام رفتارهای ممکن را در خودش دارد و از طریق toggle کردن اتریبیوت‌ها، می‌توان رفتارها را فعال یا غیرفعال کرد. اما این روش باعث می‌شود کلاس بسیار پیچیده و حجیم شده و نگهداری آن بسیار سخت شود. به همین دلیل نامطلوب تلقی می‌شود. در مقابل، Decorator نه تنها نیاز به ساختار توارثی ترکیبی را از بین می‌برد، بلکه دیگر نیازی به استفاده از feature-laden class هم نیست. در نتیجه، یک جایگزین مطلوب و منعطف است.

معایب:

- در این الگو، شی دکوریتور با شی اصلی یکسان نیستند و هویت[۱۳] متفاوتی دارند. اگر برنامه‌نویس برای بررسی تغییر اشیا از object identity استفاده کند (مثلا برای فهمیدن اینکه آیا یک شی که در اختیار داشته تغییر کرده یا نه)، این روش با استفاده از دکوریتور جواب نمی‌دهد، چون با اضافه شدن دکوریتور، ابجکت جدیدی با هویت جدید در سر زنجیره قرار می‌گیرد. پس در سیستم‌هایی که از دکوریتور استفاده می‌کنند، اتکا بر object identity برای تشخیص تغییر، نادرست است. گرچه این نکته مهم است، اما ضعف بزرگی به حساب نمی‌آید چون امروزه استفاده از object identity برای چنین بررسی‌هایی دیگر رایج نیست.

[۱۳] identity

- در سیستم‌های مبتنی بر الگوی Decorator، تعداد زیادی شی کوچک در زمان اجرا تولید می‌شود. هر شی اصلی ممکن است چندین دکوریتور داشته باشد، و این باعث می‌شود تعداد کل اشیا زیاد شود که به منابع سیستمی فشار وارد می‌کند. همچنین، وقتی یک عملیات روی شی انجام می‌شود، این عملیات باید بین چندین شی پیاپی (دکوریتورها) رد و بدل شود که باعث افزایش پیچیدگی و کاهش کارایی می‌شود. در عوض، این ساختار بسیار انعطاف‌پذیر است. در نتیجه ما در این الگو انعطاف‌پذیری را به دست می‌آوریم و بخشی از کارایی را از دست می‌دهیم. با این حال، در بسیاری از سیستم‌ها این هزینه قابل قبول است چون تعداد اشیا و دکوریتورها در عمل محدود هستند و مزایای انعطاف‌پذیری بیشتر از هزینه منابع می‌شود.

الگوی Decorator یک wrapper معروف و پرکاربرد است، اما در سیستم‌های بزرگ و data-intensive که تعداد اشیای دکوریت شده زیاد است، استفاده زیاد از آن می‌تواند فشار زیادی به منابع سیستم وارد کند. در چنین شرایطی بهتر است استفاده از دکوریتور محدود شود به بخش‌هایی که ترکیب‌های زیاد رفتاری نیاز است و در باقی موارد از وراثت استفاده شود. به هیچ وجه نباید به سراغ feature-laden class رفت، چون این نوع کلاس‌ها ممکن است در ابتدا کوچک باشند ولی در طول زمان بزرگ و پیچیده شده و تبدیل به God Class می‌شوند. نتیجه نهایی این مسیر، یک سیستم مرده و غیرقابل نگهداری خواهد بود.

Proxy:

مزایا:

- در این الگو، یک سطح indirection بین مشتری و شی اصلی ایجاد می‌شود که امکان انجام عملیات مختلف را فراهم می‌کند. این واسطه بسته به نوع پراکسی می‌تواند نقش‌های متفاوتی داشته باشد:
  - در Remote Proxy، واقعیتِ دور بودن شی اصلی (مثلا قرار داشتن روی گره دیگر در شبکه) از مشتری پنهان می‌شود و مشتری تصور می‌کند با یک شی محلی کار می‌کند.
  - در Virtual Proxy، ایجاد شی پرهزینه تا لحظه‌ی نیاز واقعی به تعویق می‌افتد.
  - در Protection Proxy و Smart Reference، پراکسی به اجرای پردازش‌های اضافی مانند کنترل دسترسی، locking یا مدیریت حافظه کمک می‌کند.

عیب: با این حال، الگوی پراکسی همیشه بدون هزینه نیست. وجود indirection می‌تواند منجر به کاهش کارایی شود، زیرا باعث افزایش تبادل پیام بین اجزا می‌شود. این افت عملکرد معمولا اندک و قابل چشم‌پوشی است، اما در سیستم‌هایی با نیاز بالا به کارایی باید در نظر گرفته شود.

Strategy:

مزایا:

- اجازه می‌دهد که خانواده‌هایی از الگوریتم‌های مختلف را بسازیم.
- الگوی استراتژی یک جایگزین مناسب برای استفاده از توارث در Context است. اگر قرار بود برای پیاده‌سازی الگوریتم‌های مختلف از زیرکلاس‌گیری از Context استفاده کنیم، با افزایش تعداد الگوریتم‌ها و ترکیب‌های ممکن بین آن‌ها، با مشکل انفجار ترکیبی کلاس‌ها مواجه می‌شدیم. چرا که برای هر ترکیب جدید از رفتارها باید یک زیرکلاس تازه تعریف می‌شد. در مقابل، Strategy با استفاده از واسپاری، رفتار را به شی جداگانه‌ای می‌سپارد و از این پیچیدگی جلوگیری می‌کند. به این ترتیب، اضافه‌کردن الگوریتم جدید به مراتب ساده‌تر و نگهداری سیستم منطقی‌تر خواهد بود.
- استراتژی، conditional statementها را هم از بین می‌برد. اگر از سوئیچ استفاده کرده باشیم، سوئیچ‌ها از بین رفته و منطق ساده‌تر می‌شود.
- از پیاده‌سازی‌های مختلف انتخاب‌هایی داریم. ممکن است پیاده‌سازی‌های مختلف از یک رفتار را داشته باشیم ولی با نوع‌های مختلف از نظر بهینه بودن. مثلا یکی برای فضا بهینه باشه دیگری برای زمان. آن پیاده‌سازی که در حال حاضر به نفع است را می‌شود انتخاب کرد. این برای وضعیتی است که رفتار یکسان و نتیجه نهایی یکسان است و فقط ما می‌خواهیم بهینه‌سازی‌های متفاوت داشته باشیم.

معایب:

- اگرچه در الگوی استراتژی با رعایت اصل DIP بین Context و Strategy، افزودن استراتژی‌های جدید بدون تغییر در Context ممکن است، اما کلاسی که وظیفه پیکربندی (پیکربند ثالث یا مشتری) را دارد، باید زیرکلاس‌های Strategy را بشناسد و از مزایا و کاربرد هرکدام آگاه باشد. این موضوع باعث ایجاد جفت‌شدگی شدید بین Client و زیرکلاس‌های Strategy می‌شود. با اینکه این وابستگی چندان مطلوب نیست، اما در عمل اجتناب‌ناپذیر است و در بسیاری از سیستم‌ها، پیکربندها به‌طور طبیعی با اجزای مختلف سیستم جفت‌شدگی دارند.
- در الگوی استراتژی یک سربار ارتباطی وجود دارد که هم به دلیل وجود indirection و نیاز به واسپاری است (که موجب کاهش عملکرد نسبت به حالت قبلی می‌شود)، و هم به دلیل واسط سنگینی که همه استراتژی‌ها باید رعایت کنند. در این وضعیت، پارامترهایی به متدها ارسال می‌شوند که برخی از آن‌ها برای استراتژی جاری بی‌استفاده‌اند، چون Context نمی‌داند دقیقا کدام استراتژی اجرا خواهد شد. راه‌حل نسبی، برقراری دید از استراتژی به Context است، یعنی Strategy به‌جای گرفتن پارامترهای زیاد، به خود Context دسترسی داشته باشد و فقط اطلاعات مورد نیاز را از آن دریافت کند. البته این روش همیشه کافی نیست، چون ممکن است برخی اطلاعات اساسا در Context نباشند و باید از بیرون بیایند. در نتیجه، مشکل ممکن است کاهش یابد ولی به‌طور کامل حل نشود.
- در الگوی استراتژی، تعداد زیادی شی در زمان اجرا ساخته می‌شود، چون ممکن است برای هر متد از

کانتکست یک استراتژی جدا تعریف شود. این موضوع باعث افزایش مصرف منابع سیستمی می‌شود. این مشکل مشابه چیزی است که در الگوی Decorator هم وجود دارد، ولی الگوی State معمولا این مسئله را ندارد چون لازم نیست برای هر متد یک شی جدا تعریف شود. همچنین، اشیای Strategy معمولا ریزدانه‌تر از اشیای State هستند.

۱۷-۴ الگوهای مرتبط

Decorator: الگوی Adapter برای زمانی مناسب است که بخواهیم به یک شی موجود از طریق یک واسط کاملا متفاوت دسترسی داشته باشیم. برخلاف آن، الگوی Decorator واسط را تغییر نمی‌دهد بلکه یا همان رابط را حفظ می‌کند یا آن را گسترش می‌دهد. همچنین، Decorator از ترکیب بازگشتی استفاده می‌کند، ویژگی‌ای که در Adapter وجود ندارد. در الگوی Decorator، می‌توان رفتار جدیدی به شی اضافه کرد، بدون اینکه تغییری در ساختار اصلی آن رخ دهد یا مشتری از این تغییرات آگاه شود. این افزودن رفتار می‌تواند به صورت پیش‌پردازش یا پس‌پردازش باشد. در این الگو، درخواست همیشه تا انتهای زنجیره منتقل می‌شود و Decorator نباید جریان پردازش را بشکند. در مقابل، الگوی Chain of Responsibility نیز ساختاری شبیه به Decorator دارد، زیرا هر دو از ترکیب بازگشتی استفاده می‌کنند و مسئولیت‌ها از یک شی به دیگری منتقل می‌شود. با این حال، در CoR هر هندلر می‌تواند مستقل از بقیه عمل کند، تصمیم بگیرد که پاسخ دهد یا درخواست را متوقف کند. چیزی که در Decorator مجاز نیست. الگوی Composite نیز از ترکیب بازگشتی بهره می‌برد و ساختار مشابهی با Decorator دارد. با این تفاوت که Composite چندین فرزند دارد و نتیجه آن‌ها را با هم ترکیب می‌کند، در حالی که Decorator فقط یک فرزند دارد و مسئولیت‌های جدیدی به آن اضافه می‌کند. این دو الگو می‌توانند با یکدیگر همکاری کنند. برای مثال می‌توان از Decorator برای افزودن رفتار خاص به یکی از اجزای درخت Composite استفاده کرد. در طراحی‌هایی که از Composite و Decorator به صورت گسترده استفاده می‌شود، به‌کارگیری الگوی Prototype می‌تواند مفید باشد. با کمک این الگو می‌توان ساختارهای پیچیده را به جای بازسازی مجدد، به راحتی کلون کرد و در نتیجه هزینه زمان و حافظه کاهش می‌یابد. از طرفی، الگوی Strategy به ما اجازه می‌دهد تا منطق داخلی شی را تغییر دهیم، در حالی که Decorator فقط ظاهر بیرونی آن را تغییر می‌دهد. در نهایت، الگوهای Proxy و Decorator از نظر ساختار بسیار شبیه هستند، چرا که هر دو مبتنی بر ترکیب هستند و مسئولیت‌هایی را به شی دیگری واگذار می‌کنند. اما تفاوت اساسی در هدف آن‌هاست. Proxy معمولا چرخه عمر شی سرویس‌دهنده را خودش مدیریت می‌کند، در حالی که در Decorator این ترکیب و چینش توسط پیکربند کنترل می‌شود.

Proxy: با Adapter، به یک شی موجود از طریق واسطی متفاوت دسترسی پیدا می‌کنیم. اما در Proxy،

واسط تغییر نمی‌کند و همان واسط شی اصلی حفظ می‌شود. در Decorator، به شی از طریق واسط تقویت‌شده دسترسی داریم. Facade از نظر ساختاری شبیه به Proxy است چون هر دو برای پنهان‌سازی یک موجودیت پیچیده و مقداردهی اولیه خودکار آن به‌کار می‌روند. اما تفاوت اصلی این است که Facade واسط متفاوتی دارد، در حالی که Proxy دقیقا همان واسط شی سرویس‌دهنده را حفظ می‌کند و بنابراین می‌تواند جایگزین مستقیم آن باشد. Decorator و Proxy از نظر ساختار به هم شباهت دارند، چون هر دو بر پایه‌ی واسپاری ساخته شده‌اند. یعنی یک شی برخی وظایف را به شی دیگری واگذار می‌کند. تفاوت آن‌ها در هدف است.

Strategy: الگوهای Bridge، State، Strategy و تا حدی Adapter ساختارهای مشابهی دارند چون همگی بر اساس واسپاری هستند و کارها را به اشیای دیگر می‌سپارند، اما مسائل متفاوتی را حل می‌کنند. الگوها فقط ساختار نیستند، بلکه بیانگر نیت و مسأله‌ای هستند که حل می‌کنند. الگوهای Command و Strategy ممکن است شبیه به نظر برسند چون هر دو اجازه می‌دهند یک شی را با یک عملیات پارامتریزه کنیم، اما نیت آن‌ها متفاوت است. Command هر عملیاتی را به یک شی تبدیل می‌کند. این تبدیل امکان تأخیر در اجرا، صف‌بندی، ذخیره تاریخچه، ارسال به سرویس‌های دور و غیره را فراهم می‌کند. Strategy راه‌های مختلف انجام یک کار را مشخص می‌کند و اجازه می‌دهد الگوریتم‌ها را درون یک کلاس ثابت Context تعویض کنیم. الگوی Decorator ظاهر شی را تغییر می‌دهد، در حالی که Strategy درونیات آن را تغییر می‌دهد. الگوی State را می‌توان گسترشی از Strategy دانست. هر دو رفتار Context را با واسپاری بخشی از کار به اشیای کمکی تغییر می‌دهند. تفاوت اصلی این است که در Strategy، استراتژی‌ها از هم مستقل‌اند، ولی در State، وضعیت‌ها می‌توانند با هم تعامل داشته باشند و وضعیت Context را تغییر دهند.

۱۸-۴ اصول شی‌گرایی

OCP ۱-۱۸-۴

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط بوده و DIP برقرار است در نتیجه OCP برقرار بوده و مشتری از جزئیات داخل مجموعه خبر ندارد که باعث می‌شود تغییرات زیرکلاس‌ها به مشتری منتشر نشود. پیکربند در زمان اجرا یک زنجیره از دکوریتورها و شی اصلی ساخته و سر زنجیره را در اختیار مشتری قرار می‌دهد تا با فراخوانی متد داخل واسط کامپوینت، این زنجیره طی شده و رفتارهای اضافه نیز همراه با رفتار اصلی اجرا شوند. اما مشتری از تغییرات داخل این زنجیره منتزع است. ولی پیکربند که باید این زنجیره را پیکربندی یا بازپیکربندی کند، به تمام زیرکلاس‌ها دید داشته و تغییرات آن‌ها به پیکربند منتشر می‌شود. البته از آن‌جایی که مشتری باید همیشه به ابتدای زنجیره دید داشته باشد، اگر عنصر ابتدای زنجیره تغییر

کرد، این مورد باید در مشتری تاثیر گذاشته و مطلع شود تا به عنصر جدید سر زنجیره اشاره کند. در کل OCP برای مشتری به خوبی رعایت می‌شود. همچنین دکوریتورها می‌توانند به راحتی گسترش یافته و این موضوع روی سایر کامپوننت‌ها تاثیری ندارد. در نتیجه OCP در بخش‌های ذکر شده نمایان است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است و پراکسی از تغییرات شی اصلی تاثیر می‌پذیرد. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته، جفت‌شدگی در این قسمت نیز بالا رفته و از تغییرات آن‌ها تاثیر می‌پذیرد. همچنین تغییر در واسط سابجکت باعث انتشار تغییرات به زیرکلاس‌ها از جمله پراکسی و شی اصلی شده و آن‌ها تاثیر می‌پذیرند.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود. در صورت تغییر واسط استراتژی نیز این تغییر به کانتکست و زیرکلاس‌های استراتژی منتشر می‌شود.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این اصل را رعایت می‌کنند.

LSP ۲-۱۸-۴

Decorator: در این الگو، زیرکلاس‌های کامپوننت با پدر خود و زیرکلاس‌های دکوریتور با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است. دکوریتورها متد Operation فرزند خود را فراخوانی می‌کنند (که از جنس کامپوننت است و اینجا الگوی Composite مشاهده می‌شود) تا درخواست در طول زنجیره حرکت کرده و به شی اصلی برسد و پاسخ برگردد و عملیات پیش‌پردازش و پس‌پردازش در این حین انجام می‌شود.

Proxy: در این الگو، زیرکلاس‌های سابجکت یعنی پراکسی‌ها و شی اصلی می‌توانند با پدر خود جایگزین شوند و با آن رابطه is-a دارند. این ویژگی باعث می‌شود که مشتری بجای شی اصلی، از طریق واسط یکسان با پراکسی در ارتباط باشد و متوجه تفاوت نشود. البته پراکسی برا تحقق رفتار مورد نیاز، لازم است که با شی اصلی پیکربندی شود توسط پیکربند ثالث. پس این اصل رعایت می‌شود.

Strategy: در این الگو، زیرکلاس‌های استراتژی با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

مقایسه: در هر سه رعایت شده است.

ISP ۳-۱۸-۴

Decorator: در این الگو، دکوریتورها هرکدام عملیات مخصوصی را پیاده‌سازی می‌کنند و آن رفتار را به نوعی به شی اصلی در زمان اجرا اضافه می‌کنند. بنابراین انسجام خوبی داشته و جداسازی واسط‌ها در این قسمت دیده می‌شود. مشتری نیز صرفا با واسط کامپوننت در ارتباط بوده و فقط عملیات آن را مشاهده می‌کند و جزئیات مربوط به دکوریتورها در این واسط تعریف نشده و عملیات نامرتبط در واسط‌ها دیده نشده‌اند. بنابراین جداسازی به خوبی شکل گرفته و ISP رعایت می‌شود.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت ارتباط دارد و شی اصلی و پراکسی هردو واسط سابجکت را پیاده‌سازی می‌کنند که درواقع عملیات شی اصلیست. واسطی با عملیات نامرتبط دیده نشده و این اصل رعایت می‌شود.

Strategy: در این الگو، الگوریتم‌های مختلف در استراتژی‌های مخصوص به خود پیاده‌سازی شده و از دوش کانتکست برداشته شده‌اند و این باعث سبک‌تر شدن کانتکست می‌شود. یک چالش مهم در این الگو، طراحی واسط Strategy است. این واسط باید امضای عملیاتی را در بر گیرد که برای پیچیده‌ترین استراتژی لازم‌اند. اگرچه بسیاری از استراتژی‌ها ساده‌اند و به چنین اطلاعاتی نیازی ندارند، اما ناچارند همه پارامترها را

دریافت کنند. چرا؟ چون واسط پایه باید تمام نیازهای ممکن در زیرکلاس‌ها را پوشش دهد. گاهی مجبوریم اجتماع واسط‌های زیرکلاس‌ها را در کلاس پایه لحاظ کنیم. این موضوع باعث ایجاد سربار می‌شود، زیرا برخی استراتژی‌ها پارامترهایی دریافت می‌کنند که هرگز استفاده نمی‌کنند و این اصل در این بخش تا حدی ضربه می‌خورد.

مقایسه: بجز الگوی سوم که تا حدی ضربه می‌خورد و شرح داده شده است، در بقیه رعایت می‌شود.

DIP ۴-۱۸-۴

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط بوده و DIP برقرار است و مشتری از جزئیات داخل مجموعه خبر ندارد که باعث می‌شود تغییرات زیرکلاس‌ها به مشتری منتشر نشود. پیکربند در زمان اجرا یک زنجیره از دکوریتورها و شی اصلی ساخته و سر زنجیره را در اختیار مشتری قرار می‌دهد تا با فراخوانی متد داخل واسط کامپوننت، این زنجیره طی شده و رفتارهای اضافه نیز همراه با رفتار اصلی اجرا شوند. اما مشتری از تغییرات داخل این زنجیره منتزع است. ولی پیکربند که باید این زنجیره را پیکربندی یا بازپیکربندی کند، به تمام زیرکلاس‌ها دید داشته و تغییرات آن‌ها به پیکربند منتشر می‌شود و در این قسمت DIP نقض می‌شود. البته از آن‌جایی که مشتری باید همیشه به ابتدای زنجیره دید داشته باشد، اگر عنصر ابتدای زنجیره تغییر کرد، این مورد باید در مشتری تاثیر گذاشته و مطلع شود تا به عنصر جدید سر زنجیره اشاره کند.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است و پراکسی از تغییرات شی اصلی تاثیر می‌پذیرد. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته، جفت‌شدگی در این قسمت نیز بالا رفته و از تغییرات آن‌ها تاثیر می‌پذیرد. همچنین تغییر در واسط سابجکت باعث انتشار تغییرات به زیرکلاس‌ها از جمله پراکسی و شی اصلی شده و آن‌ها تاثیر می‌پذیرند. در کل DIP در قسمت توضیح داده شده برقرار است.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود. در صورت تغییر واسط استراتژی نیز این تغییر به کانتکست و زیرکلاس‌های استراتژی منتشر می‌شود.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این اصل را رعایت می‌کنند.

CRP ۵-۱۸-۴

Decorator: این الگو در زمان اجرا و پس از تشکیل ساختارهای delegation محقق می شود که در واقع زنجیری از دکوریتورها هست که به شی اصلی ختم شده و در طی این زنجیر عملیات اضافی توسط دکوریتورها به صورت پیش‌پردازش یا پس‌پردازش اجرا می‌شوند و برای حرکت در زنجیر دکوریتورها عملیات کامپوننت مربوط به خود را اجرا کرده و کار را به آن واسپاری می‌کنند.

Proxy: در این الگو، پیکربند در زمان اجرا ساختار delegation را تشکیل می‌دهد و پراکسی رابطه association با شی اصلی داشته و درخواست‌ها را در زمان نیاز به شی اصلی واسپاری می‌کند و پاسخ مشتری را از این طریق می‌دهد. مشتری متوجه تفاوت میان پراکسی و شی اصلی نمی‌شود. پس این اصل رعایت شده است.

Strategy: در این الگو، پیکربند در زمان اجرا ساختار delegation را تشکیل می‌دهد و استراتژی مناسب را به کانتکست منتسب می‌کند. کانتکست نیز رفتار را به آن شی استراتژی واسپاری می‌کند تا الگوریتم مناسب اجرا شده و پاسخ مشتری را بدهد. پس این اصل به خوبی رعایت شده است.

مقایسه: هر سه این اصل را رعایت می‌کنند.

PLK ۶-۱۸-۴

Decorator: در پیاده‌سازی صحیح این الگو دید ترایا[۱۴] دیده نشده و زنجیره پیغام[۱۵] تشکیل نمی‌شود. در زنجیر تشکیل شده از دکوریتورها که به شی اصلی ختم می‌شود، هر بار دکوریتور با فراخوانی متد حلقه بعدی در زنجیره، باعث حرکت پیغام می‌شود تا به شی انتهایی رسیده و عملیات اصلی اجرا شود و بازگردد و در این حین عملیات اضافی توسط دکوریتورها به صورت پیش‌پردازش یا پس‌پردازش اجرا می‌گردند اما هیچ‌کدام شی بعد از خود را در دید شی قبلی قرار نمی‌دهند و دید ترایا تشکیل نمی‌شود پس این اصل رعایت شده است.

Proxy: در پیاده‌سازی صحیح این الگو دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. مشتری از طریق واسط سابجکت درخواست را مطح کرده و پراکسی از طریق واسپاری به شی اصلی جواب را دریافت کرده و در اختیار مشتری قرار می‌دهد ولی دید به شی اصلی را برقرار نمی‌کند. پس این اصل رعایت شده است.

Strategy: در این الگو دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. مشتری درخواست را به کانتکست ارسال کرده و کانتکست که از طریق واسط با استراتژی در ارتباط است، عملیات را واسپاری می‌کند. پس این الگو رعایت شده است.

مقایسه: در هر سه الگو رعایت شده است.

۱۹-۴ الگوهای GRASP

Information Expert ۱-۱۹-۴

Decorator: در این الگو، دکوریتورها هرکدام حاوی اطلاعات لازم برای انجام عملیات خود هستند که به صورت پیش‌پردازش یا پس‌پردازش انجام شده و به نوعی به شی اصلی در زمان اجرا رفتار اضافه می‌شود. شی اصلی و پایه نیز عملیات خود را اجرا می‌کند و حاوی اطلاعات خود است و وابستگی به دکوریتورها ندارد. در نتیجه این الگو محقق شده است.

Proxy: در این الگو، شی اصلی مستقل از پراکسی‌ها عملیات مربوط به خود را اجرا می‌کند و داده و رفتارش نزدیک هم هستند. پراکسی‌ها نیز عملیات بخصوص خود را انجام داده و داده مخصوص عملیات خود را در کنار خود دارند ولی عملیات اصلی را صرفا به شی اصلی واسپاری می‌کنند و خودشان پیاده‌سازی آن را شامل نمی‌شوند. پس این الگو محقق می‌شود.

---

[۱۴] transitive visibility
[۱۵] message chain

Strategy: در این الگو، به منظور تحقق اهداف، استراتژی‌ها تعریف می شوند که هرکدام متخصص اجرای الگوریتم مخصوص به خود هستند و این موارد از کانتکست که حاوی داده مورد نیازشان است، جدا شده‌اند. کانتکست هنگام واسپاری عملیات به استراتژی باید داده مورد نیاز آن‌ها را نیز ارسال کند. یک چالش مهم در این الگو، طراحی واسط Strategy است. این واسط باید امضای عملیاتی را در بر گیرد که برای پیچیده‌ترین استراتژی لازم‌اند. اگرچه بسیاری از استراتژی‌ها ساده‌اند و به چنین اطلاعاتی نیازی ندارند، اما ناچارند همه پارامترها را دریافت کنند. چرا؟ چون واسط پایه باید تمام نیازهای ممکن در زیرکلاس‌ها را پوشش دهد. گاهی مجبوریم اجتماع واسط‌های زیرکلاس‌ها را در کلاس پایه لحاظ کنیم. این موضوع باعث ایجاد سربار می‌شود، زیرا برخی استراتژی‌ها پارامترهایی دریافت می‌کنند که هرگز استفاده نمی‌کنند. در کل این الگو محقق نشده است.

مقایسه: در دو الگوی اول محقق شده و در الگوی آخر بنابر توضیحات ارائه شده نقض می‌شود.

Creator ۲-۱۹-۴

Decorator: در این الگو، وظیفه پیکربندی و ساخت زنجیره در زمان اجرا بر عهده پیکربند ثالث است. اما دکوریتورها با کامپوننت (شامل دکوریتور و شی اصلی) رابطه aggregation یا composition دارند یعنی اولویت ۱ یا ۲ وجود دارد. مشتری نیز که از کامپوننت سر زنجیره استفاده می‌کند، اولویت بالاتری دارد اما پیکربند ثالث این نمونه‌گیری را انجام می‌دهد. پس این الگو محقق نشده است.

Proxy: در حالت کلی این الگو، یک پیکربند ثالث وظیفه نمونه‌گیری از شی اصلی (اگر وجود نداشته باشد) و پراکسی و پیکربندی پراکسی و مشتری را دارد. اما پراکسی که رابطه association با شی اصلی دارد، اولویت بالاتری نسبت به پیکربند ثالث برای نمونه‌گیری دارد که این مورد در حالت کلی الگو نقض می‌شود. همچنین مشتری نیز درخواست خود را به پراکسی ارسال می‌کند اما وظیفه نمونه‌گیری از پراکسی و پیکربندی آن با پیکربند ثالث است. البته در Virtual Proxy، کار نمونه‌گیری از شی اصلی را خود پراکسی انجام می‌دهد و در این مورد و در این بخش، الگو محقق شده است.

Strategy: در این الگو، وظیفه پیکربندی و تشکیل ساختار delegation با پیکربند ثالث است که می‌تواند مشتری نیز باشد. در نتیجه اوست که از استراتژی نمونه‌گیری می‌کند. اما کانتکست که رابطه association با استراتژی دارد، اولویت بالاتری برای نمونه‌گیری از آن دارد که این نقض شده است. پس این الگو محقق نمی‌شود.

مقایسه: بنابر توضیحات مفصل ارائه شده هر سه الگو نقض می‌کنند. در الگوی دوم در بخشی رعایت می‌شود.

Low Coupling ۳-۱۹-۴

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط است و DIP برقرار بوده و مشتری از جزئیات داخل بی‌خبر است. تفاوتی میان شی اصلی و دکوریتور قائل نیست و نمی‌شناسد و از تغییرات داخل مجموعه منتزع شده است. در نتیجه در این قسمت جفت‌شدگی بسیار مطلوب است. ولی پیکربند ثالث برای پیکربندی و ساخت این زنجیره، نیاز است که زیرکلاس‌ها را بشناسد تا بتواند زنجیره را تشکیل داده و سر زنجیره را در اختیار مشتری قرار دهد. بنابراین در این قسمت جفت‌شدگی بالا است. به صورت کلی این الگو در قسمت ذکر شده محقق شده است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته و جفت‌شدگی در این قسمت نیز بالا است. در برخی انواع پراکسی مانند Remote Proxy این الگو کمک می‌کند تا مشتری از جزئیات پیاده‌سازی ارتباط از راه دور مانند پرتکل‌ها و غیره نیز بی‌خبر شده و جفت‌شدگی به آن‌ها نیز کاهش می‌یابد.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی

به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

High Cohesion ۴-۱۹-۴

Decorator: در این الگو، از feature-laden class که خطر تبدیل به God Class داشته و قابلیت‌ها را یک‌جا جمع کرده بود و همچنین از ساختارهای توارثی ترکیبی پرهیز شده است و رفتارها در اشیای مخصوص تمرکز پیدا کرده و این رفتار چه به صورت پیش‌پردازش و چه به صورت پس‌پردازش در زنجیره تشکیل شده اجرا می‌شوند و این رفتار داخل خود شی اصلی نیست. مشتری نیز از آن‌ها منتزع شده است. بنابراین انسجام افزایش یافته و مطلوب است. پس این الگو محقق شده است.

Proxy: در Remote Proxy وظیفه پیاده‌سازی مکانیزم‌های ارتباط از راه دور از دوش مشتری برداشته شده و در پراکسی پیاده‌سازی شده‌اند. همچنین در Protection Proxy وظیفه کنترل دسترسی به شی پراکسی سپرده شده است. در کل مشتری از انجام مسئولیت‌های زائد رها شده و در اشیای مخصوص برخی مسئولیت‌ها متمرکز شده‌اند و این به افزایش انسجام کمک می‌کند. پس این الگو محقق شده است.

Strategy: در این الگو، رفتار مربوط به الگوریتم‌ها در استراتژی‌ها متمرکز شده و از کانتکست خارج است. در نتیجه کانتکست از عملیات اضافی رها شده است. هر استراتژی نیز رفتار مختص خود را دارد. بنابراین انسجام در این الگو مطلوب است. پس این الگو محقق شده است.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

Controller ۵-۱۹-۴

Decorator: در این الگو، وظیفه کارچرخانی و آغاز زنجیره‌های تعاملی برای رویدادهای وارد شده به سیستم از سمت بیرون، دیده نمی‌شود و از کنترلر استفاده نشده است.

Proxy: به صورت کلی این الگو وظیفه دریافت رویداد از UI، آغاز زنجیره‌های تعاملی و کارچرخانی را نداشته و این نقش را به طور معمول ایفا نمی‌کند. بنابراین از کنترلر استفاده نشده است. اما اگر پراکسی به نوعی طراحی شود که اولین دریافت کننده رویداد بیرونی سیستم بوده، رفتاری را بروز داده و سپس واسپاری کند، تا حدی رفتار مشابه کنترلر را می‌شود دید. اما مانند Facade نیست که یک زیرسیستم پیچیده را نمایندگی

کرده و کارچرخانی کند. پس هدف آن نبوده است.

Strategy: در این الگو، وظیفه کارچرخانی و آغاز زنجیره‌های تعاملی برای رویدادهای وارد شده به سیستم از سمت بیرون، دیده نمی‌شود و از کنترلر استفاده نشده است.

مقایسه: هر سه الگو محقق نمی‌کنند. در الگوی دوم توضیحات مفصل‌تر بیان شده است.

Polymorphism ۶-۱۹-۴

Decorator: در این الگو، از توارث برای کامپوننت و دکوریتور استفاده شده است و می‌توان این دو موجودیت را به صورت مستقل رشد داد و انواع کامپوننت و دکوریتور اضافه کرد. هرکدام از دکوریتورها رفتاری را به صورت پیش‌پردازش یا پس‌پردازش به شی اصلی اضافه می‌کنند در زمان اجرا. پس این الگو محقق شده است.

Proxy: در این الگو، از توارث در سابجکت استفاده شده است تا بتوان پراکسی‌های متعدد در کنار شی اصلی تعریف کرد که همگی یک واسط را پیاده‌سازی می‌کنند و مشتری صرفا با این واسط ارتباط داشته و از این رو متوجه پراکسی یا شی اصلی نبوده و می‌توان از پراکسی‌های متعدد استفاده کرد که رفتار متفاوتی بروز می‌دهند. پس این الگو محقق شده است.

Strategy: در این الگو، کانتکست از طریق واسط با استراتژی در ارتباط است و الگوریتم‌های مختلف، توسط زیرکلاس‌های استراتژی پیاده‌سازی می‌شوند و کانتکست در زمان اجرا عملیات را به استراتژی مربوطه که به آن تخصیص داده شد است، واسپاری می‌کند و بدین شکل این الگو محقق شده است.

مقایسه: هر سه الگو در قسمت‌های بیان شده محقق می‌کنند.

Indirection ۷-۱۹-۴

Decorator: در این الگو، یک زنجیره از دکوریتورها و شی اصلی تشکیل می‌شود و هر گره در این زنجیره پیغام را به بعدی منتقل می‌کند تا به شی اصلی انتهای زنجیر رسیده و عملیات شی اصلی اجرا شود. در نتیجه چندین سطح indirection بین مشتری و شی اصلی ایجاد می‌شود که هدف آن اضافه کردن رفتار اضافی به شی اصلی در زمان اجرا به صورت پیش‌پردازش یا پس‌پردازش بوده است که با تعریف دکوریتورها انجام می‌پذیرد. پس این الگو محقق می‌شود.

Proxy: در این الگو، یک سطح indirection بین مشتری و شی اصلی ایجاد می‌شود (پراکسی) تا رفتاری را بسته به نوع پراکسی، در زمان اجرا پیاده‌سازی کند. پس این الگو محقق شده است.

Strategy: در این الگو، کانتکست از طریق واسط با استراتژی ارتباط داشته و عملیات را به استراتژی که در زمان اجرا به آن منتسب شده است واسپاری می‌کند. مشتری درخواست را به کانتکست ارسال کرده و او نیز عملیات را به استراتژی مربوط به خود واسپاری می‌کند و یک سطح indirection تشکیل می‌شود. در نتیجه این الگو محقق شده است.

مقایسه: هر سه محقق می‌کنند.

Pure Fabrication ۸-۱۹-۴

Decorator: در اثر اعمال این الگو، دکوریتورها که wrapper هستند شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس از این الگو استفاده شده است.

Proxy: در اثر اعمال این الگو، پراکسی‌ها که wrapper هستند شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس از این الگو استفاده شده است.

Strategy: در اثر اعمال این الگو، استراتژی‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس از این الگو استفاده شده است.

مقایسه: هر سه محقق می‌کنند.

Protected Variations ۹-۱۹-۴

Decorator: در این الگو، مشتری صرفا با واسط کامپوننت در ارتباط بوده و DIP برقرار است در نتیجه OCP برقرار بوده و مشتری از جزئیات داخل مجموعه خبر ندارد که باعث می‌شود تغییرات زیرکلاس‌ها به مشتری منتشر نشود. پیکربند در زمان اجرا یک زنجیره از دکوریتورها و شی اصلی ساخته و سر زنجیره را در اختیار مشتری قرار می‌دهد تا با فراخوانی متد داخل واسط کامپوینت، این زنجیره طی شده و رفتارهای اضافه نیز همراه با رفتار اصلی اجرا شوند. اما مشتری از تغییرات داخل این زنجیره منتزع است. ولی پیکربند که باید این زنجیره را پیکربندی یا بازپیکربندی کند، به تمام زیرکلاس‌ها دید داشته و تغییرات آن‌ها به پیکربند منتشر می‌شود. البته از آن‌جایی که مشتری باید همیشه به ابتدای زنجیره دید داشته باشد، اگر عنصر ابتدای زنجیره تغییر کرد، این مورد باید در مشتری تاثیر گذاشته و مطلع شود تا به عنصر جدید سر زنجیره اشاره کند. شی اصلی نیز وابستگی به دکوریتورها نداشته و از تغییرات آن‌ها تاثیر نمی‌پذیرد. پس این الگو محقق شده است.

Proxy: در این الگو، مشتری صرفا با واسط سابجکت در ارتباط بوده و DIP در این قسمت برقرار است

که در نتیجه آن OCP برقرار شده و مشتری از تغییرات زیر سابجکت منتزع شده است. به این شکل می‌توان پراکسی‌های متعدد به عنوان زیرکلاس سابجکت تعریف کرد. تمام پراکسی‌ها از واسط تعریف شده در سابجکت تبعیت می‌کنند و این باعث می‌شود که مشتری تفاوتی بین پراکسی و شی اصلی قائل نشده و متوجه تفاوت نشود. در زمان اجرا یک شی پراکسی به عنوان یک سطح indirection بین مشتری و شی اصلی قرار می‌گیرد و رفتار پراکسی که قبلا توضیح داده شد را محقق می‌کند. در نتیجه مشتری صرفا به واسط وابسته بوده و تا زمانی که واسط تغییر نکند از تغییرات پراکسی‌ها یا شی اصلی منتزع شده و جفت‌شدگی در این قسمت مطلوب است. اما پراکسی‌ها دید مستقیم به شی اصلی داشته و رابطه association با آن دارند تا بتوانند واسپاری انجام دهند. این ارتباط مستقیم از پراکسی به شی اصلی در زیر سابجکت باعث نقض DIP شده و جفت‌شدگی میان این عناصر زیاد است و پراکسی از تغییرات شی اصلی تاثیر می‌پذیرد. پیکربند ثالث نیز به دلیل داشتن وظیفه پیکربندی ساختار شی در زمان اجرا، به زیرکلاس‌ها دید مستقیم داشته، جفت‌شدگی در این قسمت نیز بالا رفته و از تغییرات آن‌ها تاثیر می‌پذیرد. همچنین تغییر در واسط سابجکت باعث انتشار تغییرات به زیرکلاس‌ها از جمله پراکسی و شی اصلی شده و آن‌ها تاثیر می‌پذیرند.

Strategy: در این الگو، مشتری با کانتکست در ارتباط است و کانتکست نیز از طریق واسط با استراتژی در ارتباط بوده و DIP در این قسمت برقرار است. در نتیجه OCP در این بخش برقرار بوده و کانتکست از تغییرات زیر استراتژی منتزع شده و تاثیر نمی‌پذیرد. در زمان اجرا می‌توان استراتژی به کانتکست منتسب کرد و کانتکست رفتار مربوطه را واسپاری می‌کند. پس جفت‌شدگی در این قسمت مطلوب است. اما پیکربند که می‌تواند مشتری نیز باشد، باید از زیرکلاس‌های استراتژی آگاهی کامل داشته باشد تا بتواند پیکربندی مناسبی در زمان اجرا انجام دهد و ساختار delegation را تشکیل دهد تا الگو محقق شود. پس در این قسمت جفت‌شدگی بالا است. همچنین مشتری نیز با کانتکست ارتباط مستقیم دارد و تغییرات آن روی مشتری تاثیر می‌گذارد. اگر کانتکست در هنگام واسپاری به استراتژی مربوطه، خود را نیز ارسال کرده و دید به کانتکست از سمت استراتژی ایجاد شود (بخاطر دلیلی که در قسمت تبعات توضیح داده شده است)، جفت‌شدگی از سمت استراتژی به کانتکست تشکیل و تغییرات کانتکست به استراتژی منتشر می‌شود. در صورت تغییر واسط استراتژی نیز این تغییر به کانتکست و زیرکلاس‌های استراتژی منتشر می‌شود. پس در کل این الگو در قسمت‌های بیان شده محقق شده است.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

فصل ۵

ارزیابی الگوهای Abstract Factory، Mediator و Visitor

در این فصل، سه الگوی Abstract Factory، Mediator و Visitor مبتنی بر معیارهایی که در فصل یک معرفی شدند، تحلیل، ارزیابی و مقایسه می‌شوند.

۵-۱ دسته

Abstract Factory: این الگو در دسته الگوهای طراحی آفرینشی[۱] قرار دارد. این دسته از الگوها کار نمونه‌گیری[۲] و پیکربندی کلاس‌ها و اشیا را بر عهده دارند. به این سوال می‌پردازند که چگونه نمونه‌گیری انجام دهیم که انعطاف‌پذیری بالا بوده و جفت‌شدگی[۳] در حد معمول پایین باشد. به دو دغدغه اصلی توجه می‌کنند که چگونه اشیا ساخته شوند و همچنین دانش در مورد اینکه سیستم از چه کلاس‌های concrete استفاده می‌کند را پنهان سازند.

Mediator: این الگو در دسته الگوهای رفتاری[۴] قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام[۵] و انعطاف‌پذیری.

---

۱ creational
۲ instantiation
۳ coupling
۴ behavioral
۵ cohesion

Visitor:این الگو در دسته الگوهای رفتاری قرار می‌گیرد. در این دسته از الگوها بیش‌تر با اشیایی سر و کار داریم که با هم تعامل دارند و وجه رفتاری پررنگی دارند. دغدغه تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار در شی، اداره درخواست‌ها و مدیریت بهتر تعامل اشیا در زمان اجرا دیده می‌شود با هدف کاهش جفت‌شدگی و افزایش انسجام و انعطاف‌پذیری.

مقایسه: الگوی اول در دسته آفرینشی و دو الگوی بعدی در دسته رفتاری هستند.

۵-۲ هدف

Abstract Factory: این الگو زمانی به کار می‌رود که هدف، فراهم کردن یک واسط برای مشتری‌ها به‌منظور ایجاد خانواده‌ای از اشیای مرتبط و سازگار با یکدیگر باشد. به گونه‌ای که این اشیا الزاما از یک خانواده مشخص ساخته شوند. در این الگو، رعایت اصل DIP اهمیت دارد. بنابراین، مشتری‌ها تنها با واسط محصولات سروکار دارند و نباید به زیرکلاس‌های کانکریت دسترسی داشته باشند. چون مشتری‌ها کلاس‌های کانکریت را نمی‌شناسند، قادر به نمونه‌گیری مستقیم نیستند. از این‌رو، ایجاد اشیا باید به اشیای متخصص در نمونه‌گیری سپرده شود. فلسفه این الگو دقیقا در همین نکته نهفته است که ایجاد یک ساختار که در آن مشتری‌ها بدون شناخت از پیاده‌سازی‌های مشخص، بتوانند با محصولات کار کنند و مسئولیت نمونه‌گیری را به اشیای متخصص نمونه‌گیری واسپاری کند تا هم سازگاری محصولات حفظ شود و هم وابستگی به کلاس‌های کانکریت از بین برود.

Mediator: در وضعیت قبل از اعمال الگو، تعداد زیادی شی به‌صورت درهم تنیده با یکدیگر تعامل داشتند که این امر موجب افزایش شدید جفت‌شدگی میان آن‌ها شده بود. راه حل این الگو، معرفی یک شی میانی به نام میانجی[۶] است که نقش واسطه را ایفا می‌کند و ارتباط مستقیم میان اشیا را قطع کرده و تعاملات را به صورت غیرمستقیم از طریق خود مدیریت می‌نماید. در این ساختار جدید، الگوریتم تعامل در میانجی متمرکز می‌شود. اشیا دیگر مستقیما با یکدیگر در ارتباط نیستند و از وجود و نحوه کار یکدیگر اطلاعی ندارند. بلکه هرگونه درخواست یا پیغام را به میانجی ارسال می‌کنند و او وظیفه دارد پیام‌ها را به اشیا مناسب منتقل کند و پاسخ‌ها را نیز مدیریت نماید. مزیت اصلی این الگو، کاهش چشمگیر وابستگی مستقیم میان اشیا و ساده‌سازی روابط میان آن‌هاست. با این حال، تمامی اشیا اکنون به میانجی وابسته‌اند و بنابراین نوعی وابستگی به این عنصر مرکزی ایجاد می‌شود. میانجی را می‌توان به مدیر یک سازمان تشبیه کرد که سایر اعضای سازمان (اشیا همکار[۷]) تعاملات خود را از طریق او انجام می‌دهند. از نگاه مشتری خارجی، تنها میانجی دیده می‌شود و نه اشیا داخلی، و از این منظر، میانجی رفتاری شبیه Facade دارد. هرچند هدف این دو الگو متفاوت است. هدف

[۶] Mediator
[۷] colleague

Facade، ساده‌سازی و پنهان‌سازی پیچیدگی‌های یک زیرسیستم برای مشتری است، در حالی که میانجی با تمرکز بر کاهش جفت‌شدگی شدید میان اشیا داخلی عمل می‌کند، نه الزاما برای تسهیل تعامل با مشتری بیرونی. در نتیجه، اگرچه جفت‌شدگی مستقیم میان اشیا داخلی کاهش می‌یابد، وابستگی به میانجی باقی می‌ماند و به‌نوعی جفت‌شدگی به شکل متمرکز منتقل می‌شود. مشکلی دیگر این است که میانجی می‌تواند تبدیل به گلوگاه شود چون سرش خیلی شلوغ می‌شود. همه اشیا از آن انتظار داشته و این کارچرخانی می‌تواند کار سنیگینی باشد. در نتیجه اینجا Single Point of Failure داریم. این میانجی می‌تواند به راحتی کل این مجموعه را با مشکل روبرو کند. یکی از مهم‌ترین ایرادهای این الگو این است که میانجی به راحتی ممکن است به یک God Class تبدیل شود. به این معنا که هنگام افزودن رفتار جدید به مجموعه، به‌جای توزیع مسئولیت میان اشیا همکار، ممکن است آن رفتار مستقیما در میانجی پیاده‌سازی می‌شود. در نتیجه، میانجی داده‌ها را از اشیا می‌گیرد و خودش منطق را اجرا می‌کند. این روند در بلندمدت می‌تواند میانجی را بسیار متورم و پیچیده کند. وضعیتی مشابه آنچه ممکن است در الگوی Facade نیز رخ دهد. با وجود این ایرادها، که در عمل می‌توانند دردسرساز باشند، نمی‌توان از کارایی و مزیت اصلی این الگو چشم‌پوشی کرد. میانجی با کاهش محسوس جفت‌شدگی میان اشیا در یک سیستم درهم‌تنیده، الگویی مناسب و مفید به شمار می‌رود. به شرط آن که طراح سیستم از مخاطرات احتمالی آن آگاه باشد و تدابیر لازم برای کنترل پیچیدگی میانجی را بیندیشد. قبل از اعمال الگو، الگوریتم تعامل میان اشیا به‌صورت توزیع شده پیاده‌سازی می‌شد. به این معنا که هیچ مکان مرکزی برای تعریف رفتار تعاملی وجود نداشت و هر شی می‌دانست چگونه با سایر اشیا همکاری کند. در نتیجه، فهمیدن یا تغییر این رفتار تعاملی دشوار بود، چراکه منطق تعامل در میان چندین کلاس پراکنده بود. با به‌کارگیری الگوی میانجی و متمرکز کردن الگوریتم تعامل در یک میانجی واحد، این منطق رفتاری در یک مکان مشخص قرار می‌گیرد. این تمرکز موجب می‌شود که تغییر الگوریتم تعامل ساده‌تر شود، چون تمام منطق در یک کلاس متمرکز است. همچنین درک ساختار تعاملات میان اشیا آسان‌تر گردد و گسترش رفتار تعاملی تسهیل شود. زیرا کافی است از میانجی زیرکلاس گرفته شود و رفتار مورد نظر در آن تعریف شود. در مقابل، در حالت توزیع شده‌ی قبلی، گسترش الگوریتم تعامل نیازمند ایجاد زیرکلاس برای هر شی همکار درگیر بود و همچنان هر شی باید از تعامل با دیگر اشیا آگاه می‌بود. در حالی که اکنون، تنها با گسترش یک کلاس مرکزی (میانجی) می‌توان کل الگوریتم تعامل را توسعه داد.

Visitor: ما در اینجا با مسئله‌ای مواجه‌ایم که در آن می‌خواهیم عملیاتی را روی اجزای مختلف یک ساختار آبجکتی مانند یک درخت نحوی انتزاعی اجرا کنیم. در وضعیت قبل از استفاده از الگو، اگر بخواهیم عملیاتی جدید به مجموعه عملیات قبلی اضافه کنیم، باید این متد جدید را در تمام کلاس‌های زیرمجموعه پیاده‌سازی کنیم. این فرایند باعث بروز تغییرات گسترده و منتشرشونده می‌شود، چرا که هم کلاس پایه و هم تمامی زیرکلاس‌ها باید تغییر کنند. این الگو این مشکل را با جدا کردن عملیات از ساختار داده حل می‌کند. در این الگو، به جای اینکه عملیات در خود نودها تعریف شود، مجموعه‌ای از ویزیتورها تعریف می‌شود که هرکدام مسئول انجام عملیاتی

خاص هستند (مثلا یک Visitor برای Type Check و یکی دیگر برای Code Generation). نودها صرفا یک متد به نام accept دارند که یک Visitor را می‌پذیرد و متد متناظر با نوع خود را روی آن فراخوانی می‌کند. به این ترتیب، نودها رفتار سبکی پیدا می‌کنند و دیگر مسئول اجرای عملیات نیستند. مزیت مهم این الگو در این است که برای اضافه کردن عملیات جدید نیازی به تغییر در ساختار نودها نیست. کافی است یک Visitor جدید تعریف شود. این مسئله موجب کاهش جفت‌شدگی و افزایش انعطاف‌پذیری سیستم می‌شود. در مجموع، الگوی Visitor راهکاری مؤثر برای جداسازی عملیات از ساختار داده است که مزایای قابل‌توجهی در توسعه‌پذیری و نگه‌داری سیستم دارد. قبل از استفاده از این الگو، عملیات مختلفی مستقیما در کلاس‌های نود پیاده‌سازی شده‌اند، با اینکه این عملیات از نظر ماهیت ارتباطی با یکدیگر ندارند. این باعث کاهش انسجام در نودها شده و اضافه‌کردن عملیات جدید نیازمند تغییر در واسط اصلی و پیاده‌سازی در تمام زیرکلاس‌های آن است. یعنی مصداقی از تغییر منتشرشونده. الگوی Visitor این مسئله را حل می‌کند. با استفاده از این الگو، عملیات از درون نودها خارج شده و به اشیایی مجزا به نام Visitor سپرده می‌شود که هرکدام متخصص اجرای یک عملیات خاص هستند. در نتیجه، نودها ساده‌تر و سبک‌تر می‌شوند و تنها مسئول اجرای رفتارهای بسیار محدود و اختصاصی خود هستند. عملیات پیچیده و غیرمرتبط دیگر به جای آنکه در کلاس نود باشند، در کلاس‌های Visitor مربوطه پیاده‌سازی می‌شوند. ویژگی مهم این الگو آن است که اضافه‌کردن عملیات جدید بسیار آسان می‌شود. تنها کافی است Visitor جدیدی تعریف شود، بدون نیاز به تغییر در نودها. این طراحی به اصل DIP نیز وفادار است. چرا که نودها تنها به واسط Visitor وابسته‌اند و از پیاده‌سازی‌ها و زیرکلاس‌های آن اطلاعی ندارند. در این الگو ترجیح بر این است که انواع نودها پایدار و ثابت بمانند، در حالی که افزودن عملیات جدید آزادانه و آسان انجام گیرد. همچنین، در این الگو برای اجرای عملیات، هر نود با دریافت یک Visitor و اجرای متد accept، کنترل را به Visitor واگذار می‌کند که متد خاص مربوط به آن نود را اجرا می‌کند.

مقایسه: توضیحات مفصل برای هر کدام از الگوها شرح داده شده و ارتباط الگوها نیز در قسمت الگوهای مرتبط بیان شده است.

۳-۵ حوزه

Abstract Factory: این الگو در حوزه شی[۸] قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

[۸] object scope

Mediator: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

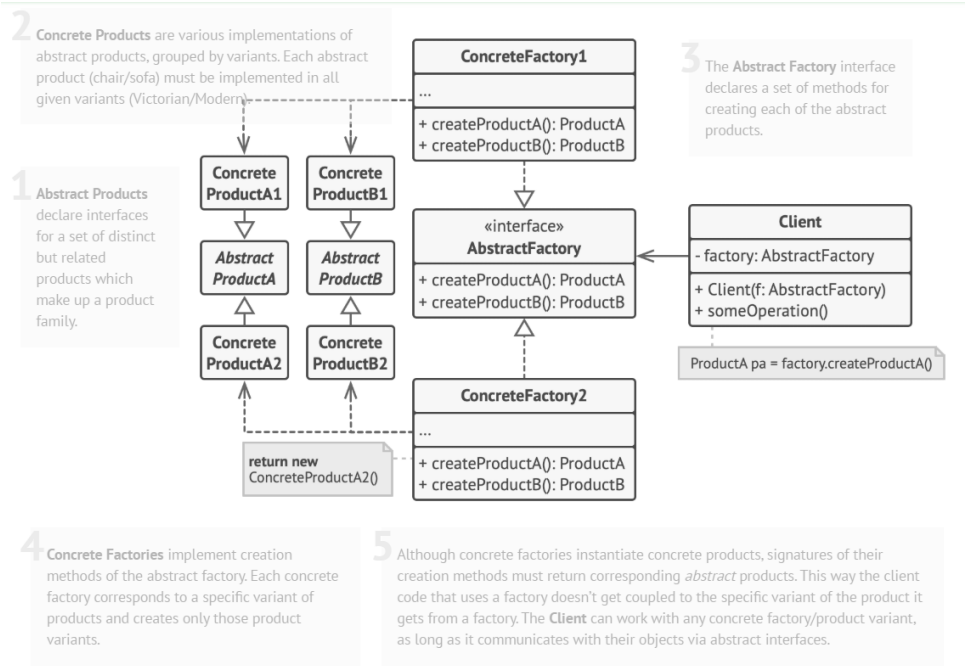
Visitor: این الگو در حوزه شی قرار دارد. این دسته از الگوها از راه‌حل منعطف مبتنی بر delegation استفاده می‌کنند و در زمان کامپایل محقق نمی‌شوند بلکه در زمان اجرا باید ساختارهای delegation تعریف شده و روابط برقرار شوند که امکان واسپاری وجود داشته و الگو محقق شود.

مقایسه: هر سه در حوزه شی قرار دارند.

۵-۴ ساختار و رفتار

Abstract Factory: ساختار این الگو در شکل ۵-۱ و ۵-۲ قابل مشاهده است. در نمودار کلاس این الگو، مشتری با واسط‌ها یا کلاس‌های انتزاعی محصولات (AbstractProductX) در ارتباط است و هیچ آگاهی‌ای از زیرکلاس‌های کانکریت آن‌ها ندارد. یک AbstractFactory نیز تعریف می‌شود که مسئول ایجاد محصولات است، اما چون خودش یک واسط است، در زمان اجرا باید یک نمونه از زیرکلاس‌های کانکریت آن (ConcreteFactory) در اختیار مشتری قرار گیرد. از آنجا که مشتری تنها واسط AbstractFactory را می‌شناسد، نمی‌تواند مستقیما از زیرکلاس مناسب نمونه‌گیری کند. به همین دلیل، یک پیکربند ثالث لازم است که این مسئولیت را برعهده بگیرد. این پیکربند ممکن است بر اساس شرایط داخلی سیستم (مثلا محدودیت فضا یا زمان) یا دستور یک اکتور بیرونی، یکی از زیرکلاس‌های Factory را انتخاب کرده و نمونه‌ای از آن را به مشتری بدهد. پس از این مرحله، الگو محقق می‌شود.

Mediator: ساختار این الگو در شکل ۵-۳ و ۵-۴ قابل مشاهده است. در نمودار کلاس الگوی میانجی، یک Mediator به‌صورت یک واسط عمومی تعریف می‌شود. سپس واسط Colleague و پیاده‌سازی‌های مشخص آن یعنی ConcreteColleague ها و همچنین پیاده‌سازی مشخص میانجی یعنی ConcreteMediator را داریم. رابطه‌ی association از سمت Colleague به سمت Mediator است. یعنی هر Colleague باید به Mediator دسترسی داشته باشد. در واقع، اگر یک شی نتواند به Mediator دسترسی داشته باشد، دیگر جزو مجموعه‌ی Colleague ها محسوب نمی‌شود و میانجی نیز با آن کاری ندارد گرچه ممکن است صرفا از آن سرویس بگیرد، ولی نه در نقش یک Colleague. اشیایی که فقط نقش سرویس‌دهنده دارند و در این مجموعه‌های درهم‌تنیده قرار نمی‌گیرند، معمولا مشکل جدی در افزایش جفت‌شدگی ایجاد نمی‌کنند. مشکل اصلی زمانی به‌وجود می‌آید که اشیا هم‌زمان به یکدیگر دید داشته باشند یا آغازگر زنجیره‌ای از تعاملات باشند. چنین روابطی است که جفت‌شدگی شدید را رقم می‌زند. در نهایت، ارتباط بین ConcreteMediator و

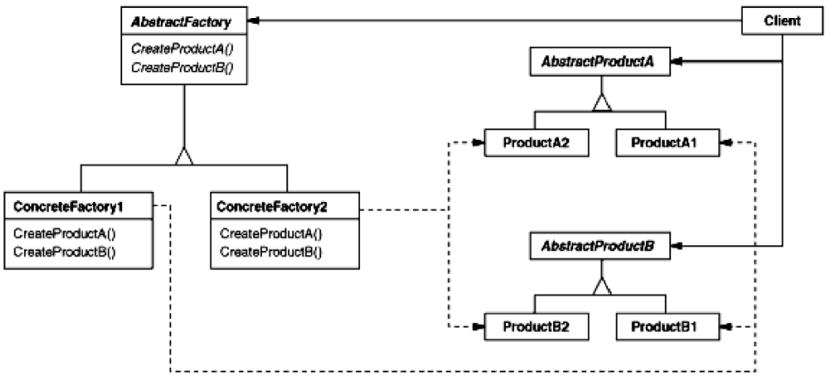


شکل ۵-۱: ساختار الگوی Abstract Factory [۱]

ConcreteColleague ها در سطح زیرکلاس‌ها تعریف می‌شود. به‌عبارت دیگر، یک میانجی ممکن است فقط برخی از Colleague ها را بشناسد و نه همه‌ی آن‌ها. در شکل ۵-۵ یک نمودار شی[۹] برای این الگو را مشاهده می‌کنید که در آن یک ConcreteMediator در مرکز قرار دارد و چندین Colleague پیرامون آن هستند. برخی از Colleague ها به Mediator دسترسی دارند، اما Mediator آن‌ها را نمی‌شناسد. یعنی رابطه‌ی دید یک‌سویه است. در مقابل، برخی دیگر از Colleague ها با Mediator رابطه‌ای دوسویه دارند. در این ساختار، متد یا اتریبیوت اجباری و استانداردی (مانند iterator) وجود ندارد و بسته به مسئله‌ی مورد نظر، می‌توان از متدها و اتریبیوت‌های متفاوتی استفاده کرد.

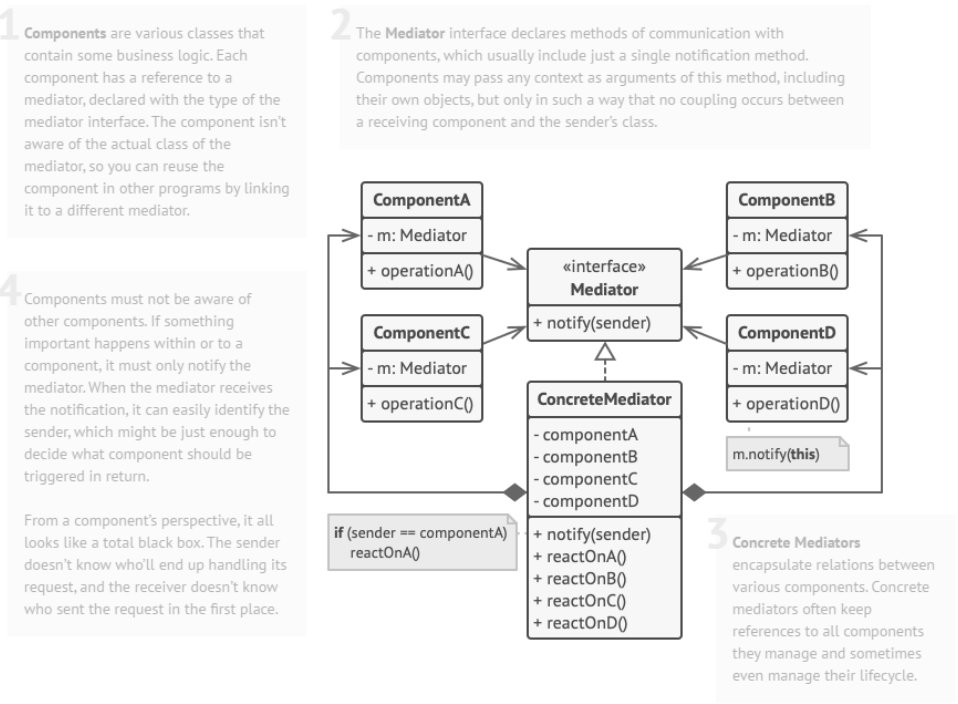
Visitor: ساختار این الگو در شکل ۵-۶ و ۵-۷ قابل مشاهده است. یک نمودار توالی نیز در شکل ۵-۸ قابل مشاهده است. مشتری به ObjectStructure دسترسی دارد و همچنین اینترفیس Visitor را نیز مشاهده می‌کند. در ظاهر، اصل DIP رعایت شده، اما در نمودار توالی مشاهده می‌شود که اینگونه نیست. زیرا مشتری معمولا نمونه‌ای از زیرکلاس‌های ConcreteVisitor ایجاد می‌کند و خود از واسط Visitor استفاده نمی‌کند. در واقع، مشتری باید انواع ConcreteVisitor را بشناسد، از نقش و وظیفه هرکدام آگاه باشد و سپس نمونه‌هایی از آن‌ها بسازد تا بتواند به المان‌ها منتقل کند. المان‌ها نیز این visitor را از طریق ObjectStructure دریافت کرده و با متد Accept، آن را می‌پذیرند. هر المان کانکریت در متد Accept خود، متد خاص مرتبط با نوعش را روی visitor فراخوانی کرده و خود را نیز به‌عنوان پارامتر با this به آن می‌دهد تا visitor به

[۹] object-diagram

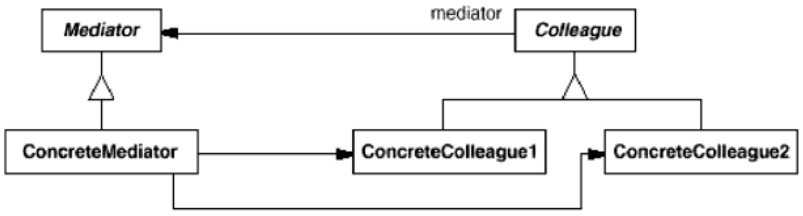


شکل ۵-۲: ساختار الگوی Abstract Factory در کتاب GoF [۲]

ویژگی‌های المان دسترسی داشته باشد. از سوی Element به visitor فقط واسط تعریف‌شده در Visitor دیده می‌شود، پس از این جهت اصل DIP برقرار است. اما از سوی زیرکلاس‌های Visitor به المان‌ها، هر visitor به‌طور مستقیم می‌داند که برای کدام نوع خاص عنصر باید کدام متد را فراخوانی کند (مثلا OperationA برای عنصر A). بنابراین، از دید ConcreteVisitor به المان‌ها، وابستگی در سطح زیرکلاس‌هاست و این وابستگی، اصل DIP را نقض می‌کند. این وابستگی هرچند از نوع dependency است، اما قوی تلقی می‌شود زیرا زیرکلاس‌ها دیده می‌شوند. به همین دلیل است که در صورت اضافه شدن نوع جدیدی از المان (زیرکلاس جدیدی از Element)، باید تغییرات لازم در Visitor نیز اعمال شود و این تغییر در Visitor منتشر می‌گردد. در نمودار توالی شکل ۵-۸ می‌بینیم که در این ساختار، عملیات پیمایش درون خود ObjectStructure تعریف شده و اجرای آن نیز به‌عهده خود اوست. ObjectStructure به‌هنگام پیمایش، به‌صورت مستقیم به هر عنصر پیام Accept(visitor) ارسال می‌کند. این visitor توسط مشتری ساخته شده و به ObjectStructure داده می‌شود تا پیمایش با آن انجام شود. بنابراین، visitorی که به عناصر منتقل می‌شود، در واقع نمونه‌ای است که مشتری آن را ساخته است. هنگامی که پیمایش به عنصر مشخصی مانند ConcreteElementA می‌رسد، این عنصر متد VisitConcreteElementA را روی visitor فراخوانی کرده و خود را نیز به‌عنوان پارامتر به آن می‌دهد. این کار موجب می‌شود visitor دید مستقیمی به عنصر پیدا کند. سپس visitor عملیات خاص خود را انجام می‌دهد و در جریان آن، متد OperationA مربوط به عنصر را نیز که حالا به آن دسترسی دارد، اجرا می‌کند. به‌عبارتی، visitor دقیقا از واسط المان اطلاع دارد و این موضوع باعث نقض Encapsulation و نیز اصل DIP می‌شود. با این حال، از سوی عناصر به visitor، تنها واسط Visitor دیده می‌شود، و از این جهت اصل DIP برقرار باقی می‌ماند. در نهایت، مشتری فقط زیرکلاس‌های visitor را می‌شناسد و نمونه‌ای از آن را می‌سازد و در اختیار ObjectStructure قرار می‌دهد. خود ObjectStructure وظیفه پیمایش را برعهده دارد و visitor را به تمامی عناصری که در طی پیمایش به آن‌ها می‌رسد، معرفی کرده و پیام Accept(visitor) را به آن‌ها می‌فرستد. نکته مهم این است که مشتری هیچ دیدی نسبت به زیرکلاس‌های عناصر ندارد و با آن‌ها



شکل ۵-۳: ساختار الگوی Mediator [۱]

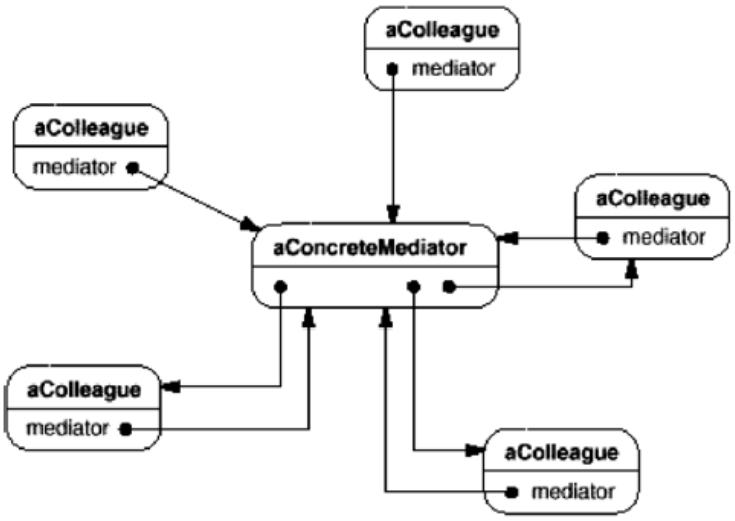


شکل ۵-۴: ساختار الگوی Mediator در کتاب GoF [۲]

مستقیما درگیر نیست. پس دید از سمت مشتری به واسط Visitor اشتباه بوده و این دید به زیرکلاس‌ها است.

۵-۵ جفت‌شدگی

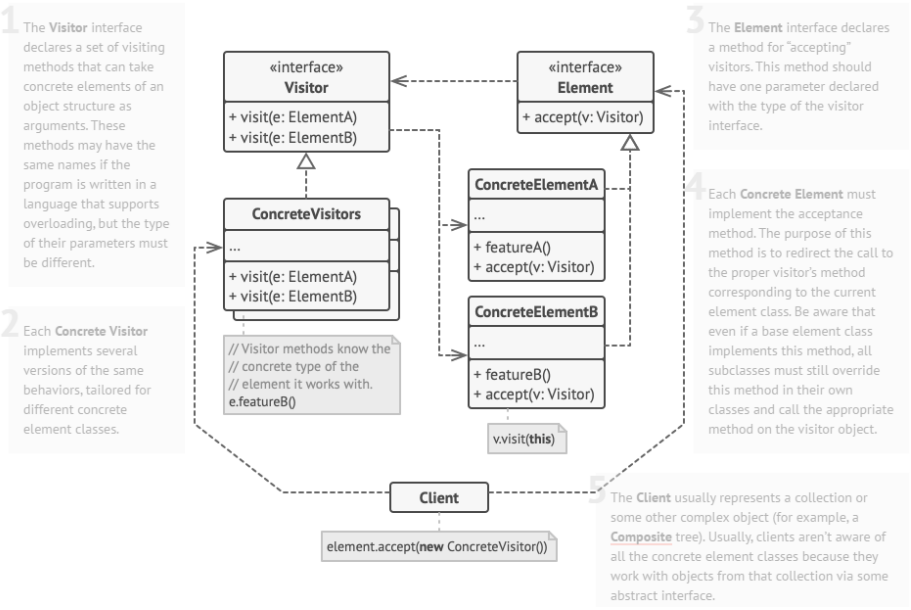
Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند



شکل ۵-۵: نمودار شی الگوی Mediator [۲]

و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین این ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود. همینطور افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد.

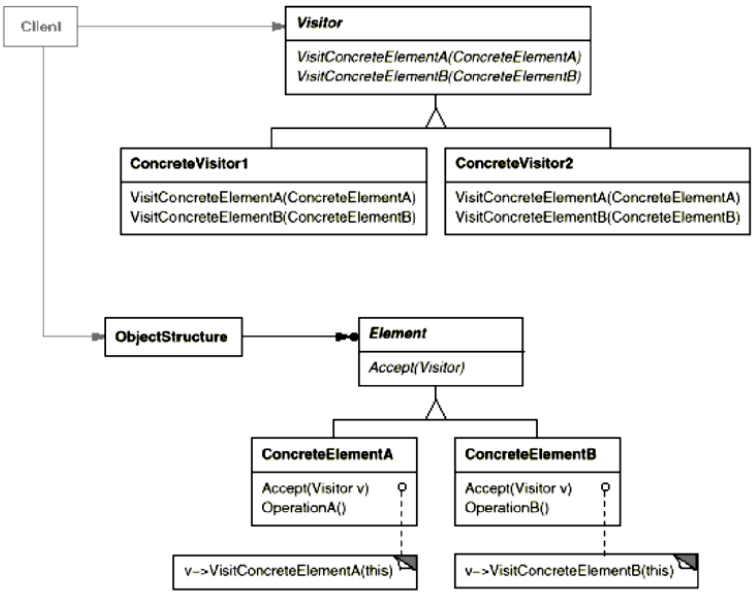
Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. در این حالت، هرچند جفت‌شدگی



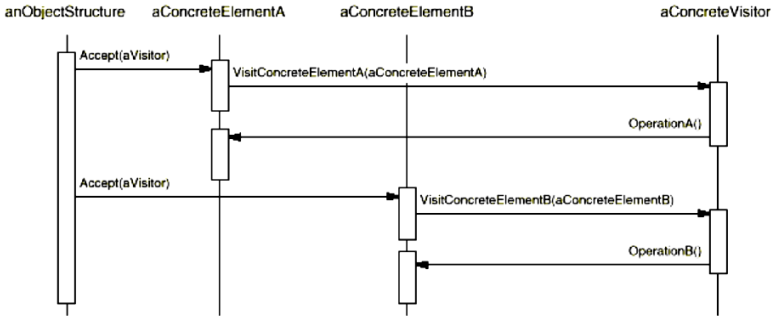
شکل ۵-۶: ساختار الگوی Visitor [۱]

میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.



شکل ۵-۷: ساختار الگوی Visitor در کتاب GoF [۲]



شکل ۵-۸: نمودار توالی الگوی Visitor [۲]

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در کنترل جفت‌شدگی می‌کنند.

۵-۶ انسجام

Abstract Factory: در این الگو، وظیفه ساخت از دوش مشتری برداشته شده و اشیای مخصوص فکتوری تعریف شده‌اند که هرکدام خانواده اشیای مرتبط به هم را نمونه‌گیری می‌کنند و به این شکل انسجام افزایش یافته است. در واقع آن بخشی از سیستم که سرویس‌گیری انجام می‌دهد، از ساخت محصولات، ترکیب و پیکربندی آن‌ها و ساختار داخلیشان مستقل شده است.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات در سیستم آسان‌تر شده، قابلیت اصلاح[۱۰] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود. یکی از مهم‌ترین خطرات این الگو این است که میانجی به راحتی ممکن است به یک God Class تبدیل شود. به این معنا که هنگام افزودن رفتار جدید به مجموعه، به‌جای توزیع مسئولیت میان اشیا همکار، ممکن است آن رفتار مستقیما در میانجی پیاده‌سازی می‌شود. در نتیجه، میانجی داده‌ها را از اشیا می‌گیرد و خودش منطق را اجرا می‌کند. این روند در بلندمدت می‌تواند میانجی را بسیار متورم و پیچیده کند و انسجام آن را به خطر بیندازد. وضعیتی مشابه آنچه ممکن است در الگوی Facade نیز رخ دهد.

Visitor: این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند. به این ترتیب، Visitor وظیفه تفکیک مسئولیت‌ها را برعهده می‌گیرد. یعنی عملیات وابسته به پیمایش که به ذات عنصر مربوط نیست، به visitorها منتقل می‌شود و در نتیجه انسجام سیستم افزایش می‌یابد. اصولا اضافه کردن نقش‌هایی مانند متخصص به مجموعه کلاس‌ها، موجب افزایش انسجام می‌شود. با این حال، این الگو موجب افزایش جفت‌شدگی نیز می‌شود. زیرا عملیاتی که پیش‌تر درون خود کلاس‌ها انجام می‌شد، اکنون نیازمند تعامل با visitor است. این موضوع همچنین منجر به نقض encapsulation می‌گردد. با وجود این، مزیت مهم آن است که می‌توان به‌راحتی visitorهای جدید (به‌عنوان متخصص‌های تازه) تعریف کرد، بدون اینکه نیاز به کاهش انسجام در کد باشد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در کنترل انسجام می‌کنند.

۵-۷ پیکربندی

Abstract Factory: در این الگو، به این دلیل که مشتری صرفا با واسط‌ها ارتباط داشته و زیرکلاس‌ها را نمی‌شناسد، خودش نمی‌تواند از آن‌ها نمونه‌گیری کند و در نتیجه یک پیکربند ثالث لازم است تا نمونه‌گیری کرده و ساختار delegation را برقرار کرده و فکتوری را در اختیار مشتری قرار دهد. بنابراین پیکربند باید تمام

[۱۰] modifiability

زیرکلاس‌های کانکریت فکتوری را بشناسد.

Mediator: در این الگو، مشتری صرفا به واسط میانجی دید داشته و یک پیکربند ثالث نیاز است تا کلاس‌های کانکریت همکار و میانجی را بشناسد و نمونه‌گیری‌های لازم را انجام داده و پیوندهای delegation را تشکیل دهد تا الگو محقق شده و میانجی را در اختیار مشتری قرار دهد.

Visitor: ObjectStructure به‌هنگام پیمایش، به‌صورت مستقیم به هر عنصر پیام Accept(visitor) ارسال می‌کند. این visitor توسط مشتری ساخته شده و به ObjectStructure داده می‌شود تا پیمایش با آن انجام شود یعنی مشتری آن را پیکربندی می‌کند. بنابراین، visitorی که به عناصر منتقل می‌شود، در واقع نمونه‌ای است که مشتری آن را ساخته است. هنگامی که پیمایش به عنصر مشخصی مانند ConcreteElementA می‌رسد، این عنصر متد VisitConcreteElementA را روی visitor فراخوانی کرده و خود را نیز به‌عنوان پارامتر به آن می‌دهد. این کار موجب می‌شود visitor دید مستقیمی به عنصر پیدا کند. سپس visitor عملیات خاص خود را انجام می‌دهد و در جریان آن، متد OperationA مربوط به عنصر را نیز که حالا به آن دسترسی دارد، اجرا می‌کند. در مجموع بدین شکل ساختار delegation در زمان اجرا تشکیل شده و الگو محقق می‌شود.

مقایسه: هر سه الگو نیاز به پیکربند دارند که جزییات آن‌ها شرح داده شده است.

۸-۵ انعطاف‌پذیری

Abstract Factory: در این الگو، مشتری صرفا با واسط‌ها در ارتباط بوده و از جزئیات داخل مجموعه خبر ندارد. بنابراین در این قسمت انعطاف‌پذیری از این جهت که می‌توان فکتوری‌های مختلف به مشتری در زمان اجرا تخصیص داد، بالاست. هر فکتوری نیز خانواده‌ای از اشیای سازگار را می‌تواند نمونه‌گیری کند. اما افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد. ساختار توارثی سازنده‌ها و محصولات به واسطه زیرکلاس‌ها به یکدیگر متصل می‌شوند. اما این شیوه اشکالاتی دارد. نخست آن که اصل DIP رعایت نمی‌شود، زیرا کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کدهای آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. دوم آن که چنین ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است که در این قسمت‌های ذکر شده انعطاف‌پذیری مشکل دارد.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات در سیستم آسان‌تر شده، قابلیت اصلاح[۱۱] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود. این که مشتری صرفا با واسط میانجی ارتباط دارد، باعث می‌شود از داخل مجموعه ناآگاه بوده و بتوان پیکربندی‌های مختلفی را انجام داده و در اختیار مشتری قرار داد. اشیای همکار نیز مستقیما با یکدیگر کار نمی‌کنند پس انعطاف‌پذیری در مورد آنان وجود دارد و می‌توان همکاران جدیدی اضافه کرد. همینطور می‌توان میانجی‌های جدید نیز اضافه کرد و گسترش داد.

Visitor: در این الگو، دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد و در این قسمت انعطاف‌پذیری پایین است. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد و در این قسمت انعطاف‌پذیری بالا است.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در کنترل انعطاف‌پذیری می‌کنند.

۹-۵ کارایی

Abstract Factory: پس از اعمال این الگو، ساختارهای توارثی محصول و فکتوری‌ها تعریف شده و اشیای متخصص ساخت بوجود آمده‌اند که پیکربند در زمان اجرا یک نمونه از فکتوری را در اختیار مشتری قرار می‌دهد تا مشتری وظیفه ساخت را به آن واسپاری کند. در نتیجه از لحاظ حافظه و زمان اندکی کارایی کاهش پیدا کرده است.

Mediator: مهم‌ترین خطر این الگو تبدیل شدن میانجی به یک God Class است. کلاسی که بیش از

[۱۱] modifiability

حد مسئولیت دارد و بیش از اندازه پیچیده می‌شود. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. این تمرکز می‌تواند میدیتور را به گلوگاه سیستم تبدیل کند، باعث افزایش بار کاری آن شود و حتی آن را به single point of failure بدل کند. در واقع، پیچیدگی تعاملات از Colleague ها حذف شده، اما درون میانجی انباشته شده است. که در این شرایط ارسال پیغام نیز افزایش یافته و در کل کارایی کاهش می‌یابد.

Visitor: این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند که به این شکل ویزیتورها تعریف شده و باعث افزایش تعداد اشیا در زمان اجرا و همچنین افزایش تعداد انتقال پیام‌ها بخاطر delegation می‌شوند که این کارایی را اندکی کاهش می‌دهد.

مقایسه: در اثر اعمال هر سه الگو کارایی کاهش می‌یابد.

۵-۱۰ شی جعلی

Abstract Factory: در اثر اعمال این الگو، فکتوری‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

Mediator: در اثر اعمال این الگو، میانجی‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

Visitor: در اثر اعمال این الگو، ویزیتورها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند.

مقایسه: هر سه الگو اشیای جعلی تعریف می‌کنند.

۵-۱۱ انتشار تغییرات

Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند

و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین این ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود. همینطور افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد.

Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند و تغییرات همکارها به یکدیگر منتشر نمی‌شود. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت

زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در کنترل انتشار تغییرات می‌کنند.

۱۲-۵ سهولت پیاده‌سازی

Abstract Factory: با استفاده از یک زبان شی‌گرا می‌توان این الگو را بدون مشکل پیاده‌سازی کرد. نیاز به تعریف واسط‌ها یا کلاس‌های انتزاعی داشته و کلاس‌های عینی برای تحقق آن‌ها باید تعریف شوند و ساختار delegation در زمان اجرا تشکیل شود.

Mediator: پیاده‌سازی این الگو با استفاده از یک زبان شی‌گرا قابل انجام است اما ملاحظاتی را باید در نظر گرفت. پیاده‌سازی الگوریتم تعامل در خود میانجی انجام می‌شود و این پیاده‌سازی می‌تواند پیچیده باشد. میانجی با خطراتی همراه است و باید با احتیاط و ملاحظه فراوان به‌کارگرفته شود. شی میانجی به‌راحتی می‌تواند به یک God Class تبدیل شود. می‌تواند به گلوگاه تبدیل شود چرا که نقش مرکزی در تعاملات را بر عهده دارد و تمام اجزا به آن وابسته‌اند. این امر ممکن است آن را به نقطه‌ی واحد خرابی نیز تبدیل کند. برای مدیریت این پیچیدگی، گاهی لازم است میانجی به چند میانجی کوچک‌تر تقسیم شود تا ساختار بهتری به آن داده شود. اما این راهکارها خود نیز پیچیدگی پیاده‌سازی را افزایش می‌دهند.

Visitor: پیاده‌سازی این الگو با استفاده از یک زبان شی‌گرا قابل انجام است. نیاز به تعریف واسط‌ها یا فوق کلاس‌های انتزاعی و برقرار ارتباطات مطابق ساختار الگو است.

۱۳-۵ موارد کاربرد

Abstract Factory:

- در طراحی سیستم، هدف آن است که بخش سرویس‌گیرنده (مشتری‌ها) از جزئیات ساخت، ترکیب و

ساختار داخلی محصولات مستقل باشد. این استقلال به معنای عدم وابستگی مشتری‌ها به نحوه ایجاد، پیکربندی و اجزای داخلی محصولات است. در ادبیات GoF، واژه representation به ساختار داخلی یک شی اشاره دارد، یعنی ترکیب درونی آن با دیگر اشیا. بنابراین، منظور از استقلال مشتری‌ها، بی‌نیازی آن‌ها از دانستن این ساختارهای داخلی است. برای تحقق این جدایی میان بخش ساخت و بخش استفاده در سیستم، لازم است اشیایی متخصص در ایجاد محصولات تعریف شوند و این همان جایی است که الگوی Abstract Factory وارد عمل می‌شود.

- سیستم ما باید با یکی از چند خانواده محصول پیکربندی بشود. یعنی ما خانواده‌های متعدد از محصولات داریم و یک خانواده باید تعیین شوند در هر لحظه و نمونه‌هایی که گرفته می‌شوند مربوط به این خانواده باشند. امکان جایگزینی خانواده نیز فراهم باشد.
- در برخی سیستم‌ها نیاز است که مجموعه‌ای از محصولات، به صورت یک خانواده سازگار با یکدیگر، مورد استفاده قرار گیرند. هدف، تحمیل و تضمین این سازگاری است. به طوری که بتوان خانواده مورد نظر را تعیین کرد و در صورت نیاز، آن را تغییر داد، در حالی که اطمینان حاصل شود تمامی نمونه‌های تولید شده متعلق به همان خانواده و با یکدیگر سازگار باشند. الگوی Abstract Factory این سازگاری و انعطاف‌پذیری را به‌صورت خودکار فراهم می‌سازد.
- در حالتی خاص، هدف آن است که مجموعه‌ای از کلاس‌های کتابخانه‌ای[۱۲] از محصولات ارائه شود که مشتری‌ها فقط با واسط آن‌ها سروکار داشته باشند، نه با پیاده‌سازی‌شان. مانند Java Collection، مشتری‌ها تنها باید واسط‌ها را ببینند و نباید به کلاس‌های کانکریت دسترسی داشته باشند. چون مشتری‌ها این کلاس‌ها را نمی‌شناسند، قادر به نمونه‌گیری مستقیم نیستند و تنها می‌توانند با واسط کار کنند. این ساختار که اصل DIP را رعایت می‌کند، در فریم‌ورک‌ها نیز دیده می‌شود، جایی که کلاس‌های پایه عمدتا انتزاعی هستند و باید در سیستم مقصد با تعریف زیرکلاس‌ها گسترش یابند. برخی از این کلاس‌های قابل استفاده مجدد، پیاده‌سازی‌های قابل override دارند و برخی متدهای کاملا انتزاعی. در چنین ساختارهایی، که مشتری‌ها تنها به واسط‌ها وابسته‌اند و نمی‌توانند به طور مستقیم نمونه‌گیری انجام دهند، استفاده از الگوی Abstract Factory ضروری است تا فرایند ایجاد اشیا به شی تخصصی مربوطه واگذار شود.

:Mediator

- در اینجا با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. الگوریتم تعامل میان آن‌ها مشخص است، اما به دلیل شدت جفت‌شدگی، درک رفتار کلی سیستم دشوار می‌شود. این وابستگی شدید همچنین باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر

[۱۲] class library

سیستم منتشر شود.

- وقتی میزان جفت‌شدگی در یک مجموعه از اشیا بالا باشد، قابلیت استفاده‌ی مجدد کاهش می‌یابد. دلیل آن این است که هر شی به سایر اشیای مجموعه وابسته است و به‌تنهایی قابل استفاده در جای دیگر نیست. زیرا برای عملکرد درست، نیازمند همراه داشتن اشیای وابسته‌اش است. در نتیجه، برای استفاده‌ی مجدد از یک جزء، باید چندین جزء دیگر را نیز همراه آن منتقل کرد. این وابستگی‌های گسترده باعث افزایش پیچیدگی و دشوار شدن استفاده‌ی مجدد از اجزای سیستم می‌شود.
- هدف ما این است که رفتاری را که به صورت توزیع شده میان مجموعه‌ای از اشیا قرار دارد، قابل سفارشی‌سازی و توسعه‌پذیر کنیم، بدون آن‌که ناچار به ایجاد تعداد زیادی زیرکلاس شویم. در حالت عادی، این رفتار تعاملی جمعی بین اشیا پخش شده است و برای گسترش آن، باید از هر شی جداگانه زیرکلاس بگیریم. اما با استفاده از الگوی Mediator، این رفتار جمعی در یک نقطه متمرکز می‌شود. در نتیجه، می‌توان تنها با گسترش میانجی، کل رفتار را به سادگی توسعه داد.

:Visitor

- ما یک ساختار آبجکتی داریم شامل اشیایی از کلاس‌های مختلف با واسط‌های متفاوت، اما چون می‌خواهیم عملیاتی روی این اشیا انجام دهیم که به نوع کلاس آن‌ها وابسته است، مجبور می‌شویم یک واسط کلی تعریف کنیم و هر عملیات را در همه زیرکلاس‌ها پیاده‌سازی کنیم. این باعث می‌شود کلاس‌هایی که هویت و وظایف خاص خودشان را دارند، پر شوند از عملیاتی که به آن‌ها ربطی ندارند، فقط به خاطر نیاز به پیمایش. نتیجه این کار یک ساختار سنگین و پیچیده با زیرکلاس‌های متعدد و پیاده‌سازی‌های پراکنده است، و افزودن عملیات جدید هم سخت و پرهزینه می‌شود. در چنین شرایطی، استفاده از الگوی Visitor راه‌حل مناسب‌تری است. یعنی به‌جای تعریف عملیات درون نودها، آن‌ها را به اشیایی به نام Visitor منتقل می‌کنیم که مسئول انجام این عملیات در حین پیمایش هستند.
- وقتی تعداد زیادی عملیات نامرتبط وجود دارد که باید روی اشیای یک ساختار آبجکتی اجرا شوند، ولی نمی‌خواهیم این عملیات را وارد کلاس‌های المان کنیم، چون این عملیات ذاتی المان‌ها نیستند و فقط پیاده‌سازی‌شان به نوع خاص هر المان وابسته است، در چنین حالتی استفاده از الگوی Visitor گزینه‌ای مناسب است.
- در ساختارهای آبجکتی، کلاس‌هایی که این ساختار را تعریف می‌کنند معمولا ثابت و پایدارند و به ندرت نوع جدیدی به آن‌ها اضافه می‌شود. در مقابل، آنچه معمولا تغییر می‌کند یا افزوده می‌شود، عملیاتی است که باید روی این مجموعه از کلاس‌ها اجرا شود.

مقایسه: موارد کاربرد هر الگو به صورت مفصل شرح داده شد و ارتباط آن‌ها در قسمت الگوهای مرتبط

بیان شده است.

۵-۱۴ کپسوله‌سازی

Abstract Factory: در این الگو، مشتری صرفا با واسط‌ها در ارتباط بوده و به ساختار داخلی اشیا دید پیدا نمی‌کند و فکتوری‌ها نیز صرفا از محصولات نمونه‌گیری می‌کنند و در کل کپسوله‌سازی در این الگو رعایت شده است.

Mediator: در این الگو، همکاران از طریق واسط با میانجی در ارتباط بوده و دید نسبت به جزئیات میانجی نداشته و کپسوله‌سازی در این قسمت رعایت می‌شود. مشتری نیز صرفا با واسط میانجی ارتباط داشته و جزئیات داخل مجموعه به آن منتشر نمی‌شود و دید به داخل ندارد. اما میانجی بسته به نیاز می‌تواند با همکار ارتباط داشته باشد و این ارتباط مستقیم در سطح زیرکلاس‌ها است و در نتیجه امکان نقض کپسوله‌سازی محیا می‌شود در این قسمت.

Visitor: در این الگو، عملیاتی که پیش‌تر درون خود کلاس‌های المان انجام می‌شد، اکنون نیازمند تعامل با visitor است. این موضوع همچنین منجر به نقض encapsulation می‌گردد. رفتاری که از داخل این کلاس‌ها استخراج شده و به متخصص‌ها سپرده شده، قبل از استخراج از تمام امکانات داخلی این کلاس المان استفاده می‌کردند و اکنون که در متخصص قرار داده شده است، باید دسترسی داده شود که بتواند کماکان از آن رفتارهایی که به آن‌ها احتیاج دارد استفاده کند.

مقایسه: در الگوی اول رعایت می‌شود و در دو الگوی دیگر در قسمت‌های بیان شده نقض می‌شود.

۵-۱۵ جداسازی دغدغه‌ها

Abstract Factory: در این الگو، وظیفه نمونه‌گیری و ساخت به اشیای متخصص ساخت فکتوری محول می‌شود و مشتری درگیر آن نشده و به این شکل جداسازی دغدغه‌ها انجام شده است.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات

در سیستم آسان‌تر شده، قابلیت اصلاح[۱۳] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود.

Visitor: این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند. به این ترتیب، Visitor وظیفه تفکیک مسئولیت‌ها را برعهده می‌گیرد. یعنی عملیات وابسته به پیمایش که به ذات عنصر مربوط نیست، به visitorها منتقل می‌شود و در نتیجه انسجام سیستم افزایش می‌یابد. اصولا اضافه کردن نقش‌هایی مانند متخصص به مجموعه کلاس‌ها، موجب افزایش انسجام می‌شود. پس به این شکل جداسازی دغدغه‌ها انجام شده است.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در جداسازی دغدغه‌ها می‌کنند.

۵-۱۶ مزایا و معایب (تبعات)

Abstract Factory:

مزایا:

- کلاس‌های کانکریت در این ساختار از دید مشتری‌ها ایزوله شده‌اند. یعنی در پسِ واسط یا کلاس انتزاعی پنهان شده‌اند و مشتری‌ها تنها با واسط تعامل دارند، بدون آن که هیچ شناختی از کلاس‌های زیرین در کد آن‌ها وجود داشته باشد.
- در این الگو، تعویض خانواده محصولات بسیار آسان است. بدون آن که نیازی به تغییر در کد مشتری‌ها باشد.
- سازگاری بین محصولات را به این شکل ارتقا دادیم. این که خانواده محصولات بتوانند با هم نمونه‌گیری شده و با هم کار کنند، به صورت خودکار از طریق این الگو محقق شده است.

عیب: افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد. ساختار توارثی سازنده‌ها و محصولات به واسطه زیرکلاس‌ها به یکدیگر متصل می‌شوند. اما این شیوه اشکالاتی دارد. نخست آن که اصل DIP رعایت نمی‌شود، زیرا کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کدهای آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود.

[۱۳] modifiability

دوم آن که چنین ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است.

:Mediator

مزایا:

- این الگو نیاز به زیرکلاس گیری متعدد را کاهش می‌دهد. زیرا تمامی رفتارهای تعاملی توزیع شده میان اشیای درگیر را در خود متمرکز می‌کند. بنابراین، برای تغییر یا گسترش این رفتارها، کافی است تنها از میانجی زیرکلاس گرفته و متدهای مربوطه را override کنیم. در حالی که پیش از استفاده از این الگو، برای اعمال چنین تغییری، لازم بود از تمامی کلاس‌های درگیر زیرکلاس گرفته شود.
- ارتباط Colleague ها غیرمستقیم می‌شود که این هدف اصلی بوده است. جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است.
- وقتی از object protocol صحبت می‌کنیم، منظور نحوه‌ی تعامل و ارتباط میان اشیاست. یعنی این که چگونه از یکدیگر سرویس بگیرند و چه دانشی نسبت به هم داشته باشند. با به‌کارگیری الگوی میانجی، این تعامل‌ها بسیار ساده می‌شوند. زیرا اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. به این ترتیب، پیچیدگی تعاملات میان آن‌ها به‌طور قابل‌توجهی کاهش می‌یابد.
- در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات در سیستم آسان‌تر شده، قابلیت اصلاح[۱۴] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود.

عیب: مهم‌ترین خطر آن تبدیل شدن میانجی به یک God Class است. کلاسی که بیش از حد مسئولیت دارد و بیش از اندازه پیچیده می‌شود. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. این تمرکز می‌تواند میدیتور را به گلوگاه سیستم تبدیل کند، باعث افزایش بارکاری آن شود و حتی آن را به single point of failure بدل کند. در واقع، پیچیدگی تعاملات از Colleague ها حذف شده، اما درون میانجی انباشته شده است. اگر بتوان این هزینه‌ها

[۱۴] modifiability

را پذیرفت یا احتمال بروز آن‌ها پایین باشد، الگوی میانجی انتخاب مناسبی خواهد بود، زیرا جفت‌شدگی شدید میان اشیا اغلب مشکل‌سازتر از این معایب است.

Visitor:

مزایا:

- این امکان را می‌دهد که بتوانیم عملیات جدید اضافه کنیم. یعنی می‌شود ویزیتور جدید اضافه کرد به عنوان زیرکلاس به ویزیتور. فقط مشتری از این تغییر تاثیر پذیرفته و کد آن عوض می‌شود. بقیه قسمت‌ها هیچگونه تغییری نمی‌کنند.
- این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند. به این ترتیب، Visitor وظیفه تفکیک مسئولیت‌ها را برعهده می‌گیرد. یعنی عملیات وابسته به پیمایش که به ذات عنصر مربوط نیست، به visitorها منتقل می‌شود و در نتیجه انسجام سیستم افزایش می‌یابد. اصولا اضافه کردن نقش‌هایی مانند متخصص به مجموعه کلاس‌ها، موجب افزایش انسجام می‌شود. با این حال، این الگو موجب افزایش جفت‌شدگی نیز می‌شود. زیرا عملیاتی که پیش‌تر درون خود کلاس‌ها انجام می‌شد، اکنون نیازمند تعامل با visitor است. این موضوع همچنین منجر به نقض encapsulation می‌گردد. با وجود این، مزیت مهم آن است که می‌توان به‌راحتی visitorهای جدید (به‌عنوان متخصص‌های تازه) تعریف کرد، بدون اینکه نیاز به کاهش انسجام در کد باشد.

معایب:

- اگر یک ConcreteElement جدید اضافه کنیم، کل ساختار ویزیتورها دست می‌خورد. یعنی واسط مربوط به visitor دست خورده و برای این نوع جدید باید VisitElement جدید تعریف شود در تک تک زیرکلاس‌های ویزیتور و این تغییر منتشر شونده است.
- encapsulation نقض می‌شود. رفتاری که از داخل این کلاس‌ها استخراج شده و به متخصص‌ها سپرده شده، قبل از استخراج از تمام امکانات داخلی این کلاس المان استفاده می‌کردند و اکنون که در متخصص قرار داده شده است، باید دسترسی داده شود که بتواند کماکان از آن رفتارهایی که به آن‌ها احتیاج دارد استفاده کند.

۱۷-۵ الگوهای مرتبط

Abstract Factory: بسیاری از طراحی‌ها ابتدا با الگوی Factory Method آغاز می‌شوند چون ساده‌تر بوده و از طریق زیرکلاس‌ها قابل شخصی‌سازی است، اما با گذشت زمان به سمت الگوهای پیچیده‌تر و منعطف‌تری مانند Abstract Factory، Prototype یا Builder تکامل پیدا می‌کنند. Builder بر ساخت تدریجی و مرحله به مرحله اشیای پیچیده تمرکز دارد. Abstract Factory برای ساخت خانواده‌ای از اشیای مرتبط تخصص دارد. تفاوت آن‌ها این است که Abstract Factory بلافاصله محصول را برمی‌گرداند، ولی Builder اجازه می‌دهد مراحل ساخت بیشتری قبل از دریافت محصول انجام شود. کلاس‌های Abstract Factory معمولا بر پایه مجموعه‌ای از Factory Methodها ساخته می‌شوند، اما می‌توان از الگوی Prototype هم برای ترکیب این متدها استفاده کرد. الگوی Abstract Factory می‌تواند به‌عنوان جایگزینی برای Facade به کار رود، زمانی‌که فقط می‌خواهید فرایند ساخت اشیای زیرسیستم را از مشتری پنهان کنید. همچنین می‌توان از Abstract Factory همراه با Bridgeاستفاده کرد. این ترکیب زمانی مفید است که بعضی انتزاع‌ها فقط با پیاده‌سازی‌های خاصی کار می‌کنند. در این حالت، Abstract Factory می‌تواند این وابستگی‌ها را کپسوله کند و پیچیدگی را از مشتری پنهان سازد. در نهایت، الگوهای Abstract Factory، Builder و Prototype می‌توانند به‌صورت Singleton پیاده‌سازی شوند.

Mediator: الگوهای Chain of Responsibility، Command، Mediator و Observer روش‌های مختلفی برای ارتباط ارسال‌کنندگان و دریافت‌کنندگان درخواست‌ها ارائه می‌دهند. Chain of Responsibility درخواست را به‌طور ترتیبی در یک زنجیره‌ی دینامیک از دریافت‌کنندگان ممکن ارسال می‌کند تا زمانی که یکی از آن‌ها درخواست را پردازش کند. Command ارتباط یک‌طرفه‌ای بین ارسال‌کنندگان و دریافت‌کنندگان برقرار می‌کند. Mediator ارتباط مستقیم بین ارسال‌کنندگان و دریافت‌کنندگان را از بین می‌برد و آن‌ها را مجبور می‌کند که به‌طور غیرمستقیم از طریق یک شی میانجی با یکدیگر ارتباط برقرار کنند. Observer به دریافت‌کنندگان این امکان را می‌دهد که به‌صورت دینامیک در دریافت درخواست‌ها مشترک شوند یا از آن‌ها خارج شوند. الگوهای Facade و Mediator نقش‌های مشابهی دارند. آن‌ها سعی می‌کنند همکاری میان تعدادی کلاس‌های به‌هم‌وابسته را سازمان‌دهی کنند. Facade یک واسط ساده شده به زیرسیستم اشی تعریف می‌کند، اما هیچ قابلیت جدیدی اضافه نمی‌کند. خود زیرسیستم از وجود facade بی‌خبر است و اشیای درون زیرسیستم می‌توانند مستقیما با هم ارتباط برقرار کنند. Mediator ارتباطات میان اجزای سیستم را متمرکز می‌کند. اجزا تنها میانجی را می‌شناسند و به‌طور مستقیم با هم ارتباط برقرار نمی‌کنند. یک پیاده‌سازی محبوب از الگوی Mediator که به Observer وابسته است، در آن میانجی نقش ناشر[۱۵] را ایفا می‌کند و اجزا به‌عنوان مشترکین عمل کرده و به رویدادهای میانجی مشترک می‌شوند یا از آن‌ها خارج می‌شوند. زمانی که Mediator

[۱۵] publisher

به این شکل پیاده‌سازی می‌شود، ممکن است شبیه به Observer به نظر برسد.

Visitor: می‌توان الگوی Visitor را نسخه‌ای قدرتمندتر از الگوی Command در نظر گرفت، زیرا اشیای آن قادرند عملیاتی را روی انواع مختلفی از کلاس‌ها انجام دهند. این الگو برای اعمال یک عملیات روی کل ساختار درختی Composite نیز قابل استفاده است. همچنین می‌توان Visitor را همراه با Iterator به‌کار برد تا در ساختارهای داده پیچیده پیمایش کرده و روی عناصر مختلف حتی اگر کلاس‌های متفاوتی داشته باشند عملیاتی اجرا کند.

۱۸-۵ اصول شی‌گرایی

OCP ۱-۱۸-۵

Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین این ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود. همینطور افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد.

Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر

می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این اصل را رعایت می‌کنند.

LSP ۲-۱۸-۵

Abstract Factory: در این الگو، زیرکلاس‌های فکتوری با پدر خود و زیرکلاس‌های محصولات با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

Mediator: در این الگو، زیرکلاس‌های همکار با پدر خود و زیرکلاس‌های میانجی با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

Visitor: در این الگو، زیرکلاس‌های وزیتور با پدر خود و زیرکلاس‌های المان با پدر خود رابطه is-a داشته و می‌توانند با پدر جایگزین شوند پس این اصل رعایت شده است.

مقایسه: در هر سه رعایت می‌شود.

ISP ۳-۱۸-۵

Abstract Factory: در این الگو، برای هرکدام از محصولات واسط مخصوص تعریف شده و واسطی مخصوص فکتوری نیز تعریف شده است که عملیاتی برای ساخت محصولات تعریف می‌کند. در نتیجه جداسازی دغدغه‌ها و تفکیک واسط‌ها به خوبی دیده می‌شود.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات در سیستم آسان‌تر شده، قابلیت اصلاح[۱۶] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود. پس تعریف جداگانه این واسط‌ها به حفظ این اصل کمک کرده است البته باید توجه شود که همانند وضعیتی که برای Facade نیز صادق بود، باید از به مرور زمان سنگین‌تر شدن و پیچیده‌تر شدن میانجی جلوگیری کرد و نباید اجازه از دست رفتن انسجام واسط میانجی را داد تا این اصل حفظ شود.

Visitor: این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند. به این ترتیب، Visitor وظیفه تفکیک مسئولیت‌ها را برعهده می‌گیرد. یعنی عملیات وابسته به پیمایش که به ذات عنصر مربوط نیست، به visitorها منتقل می‌شود و در نتیجه انسجام سیستم افزایش می‌یابد. واسط مربوط به المان شامل متد پذیرش ویزیتور بوده و هر المان نیز عملیات خود را دارد. عملیات بی‌ربط به ذات المان‌ها در اشیای تخصصی ویزیتور قرار گرفته‌اند و به این شکل واسط‌ها تفکیک شده‌اند.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است سعی در رعایت این اصل می‌کنند.

---

[۱۶] modifiability

DIP ۴-۱۸-۵

Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود.

Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در

آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این اصل را رعایت می‌کنند.

CRP ۵-۱۸-۵

Abstract Factory: این الگو در زمان اجرا و پس از تشکیل ساختارهای delegation محقق می‌شود. درواقع در زمان اجرا یک پیکربند ثالث از فکتوری مربوطه نمونه‌گیری کرده و آن را در اختیار مشتری قرار می‌دهد. مشتری در زمان اجرا می‌تواند درخواست ساخت را به فکتوری واسپاری کند. فکتوری نیز از محصول مربوطه‌اش نمونه‌گیری کرده و در اختیار مشتری قرار می‌دهد. پس این اصل رعایت شده است.

Mediator: در این الگو، یک پیکربند ثالث در زمان اجرا ساختار delegation را تشکیل داده و در نتیجه آن مشتری دید به میانجی پیدا کرده، اشیای همکار دید به میانجی داشته و میانجی نیز به همکاران مورد نیاز دید پیدا می‌کند و بدین شکل تعامل‌ها از طریق واسپاری محقق می‌شوند. پس این اصل رعایت شده است.

Visitor: در این الگو، عملیات بی‌ربط به ذات المان‌ها که مربوط به پیمایش هستند از داخل المان خارج شده و در ویزیتورها قرار داده شده‌اند و از طریق واسپاری در زمان اجرا انجام می‌شوند. در نتیجه این اصل رعایت شده است.

مقایسه: هر سه رعایت می‌کنند.

PLK ۶-۱۸-۵

Abstract Factory: در این الگو دید ترایا[۱۷] دیده نشده و زنجیره پیغام[۱۸] تشکیل نمی‌شود. مشتری در زمان اجرا درخواست ساخت را به فکتوری مربوطه‌اش واسپاری کرده و فکتوری نیز از محصول مربوطه

[۱۷] transitive visibility
[۱۸] message chain

نمونه‌گیری کرده و در اختیار مشتری قرار می‌دهد که هدف الگو بوده است. مشتری نیز از طریق واسط محصول با آن کار می‌کند. پس این اصل رعایت شده است.

Mediator: در پیاده‌سازی صحیح این الگو دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. میانجی درخواست‌ها را دریافت کرده و الگوریتم تعامل را اجرا و کارچرخانی می‌کند اما همکاران را مستقیما در دید یکدیگر قرار نمی‌دهد. پس این اصل رعایت شده است.

Visitor: در این الگو دید ترایا دیده نشده و زنجیره پیغام تشکیل نمی‌شود. با این که المان‌ها خود را در معرض دید ویزیتور قرار می‌دهند و این نقض کپسوله‌سازی و افزایش دهنده جفت‌شدگی است، اما زنجیره تشکیل نمی‌شود. پس این اصل رعایت شده است.

مقایسه: هر سه رعایت می‌کنند.

۱۹-۵ الگوهای GRASP

۱-۱۹-۵ Information Expert

Abstract Factory: در این الگو، اشیای متخصص فکتوری حاوی داده لازم برای ساخت خانواده محصولات مربوط به خود هستند. مشتری درخواست ساخت را در زمان اجرا به فکتوری مربوطه‌اش واسپاری می‌کند و محصول ساخته شده را دریافت می‌کند. پس این الگو محقق شده است.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند. پس میانجی حاوی داده و رفتار مربوط به خود و همکارها نیز داده و رفتار مخصوص خود را در کنار خود دارند. پس این الگو محقق شده است.

Visitor: در این الگو، برای تحقق اهداف الگو که قبلا ذکر شد، اشیای متخصص ویزیتور تعریف شده‌اند و رفتارهایی از المان‌ها خارج شده و در ویزیتورها قرارگرفته است و ویزیتورها برای تحقق این عملیات‌ها نیاز به داده‌هایی دارند که در نزد خود نیستند پس رفتار در کنار داده خود قرار نگرفته است. پس این الگو محقق نمی‌شود.

مقایسه: دو الگوی اول محقق کرده و الگوی سوم نقض می‌کند.

Creator ۲-۱۹-۵

Abstract Factory: در این الگو، وظیفه نمونه‌گیری از محصولات با اشیای متخصص فکتوری است که داده لازم برای نمونه‌گیری را دارند. اما استفاده کننده نهایی محصول، مشتری است که با واسط محصولات رابطه نیز دارد. پس اولویت بالاتری برای نمونه‌گیری داشته است. همچنین استفاده کننده از فکتوری نیز مشتری بوده و با فکتوری رابطه نیز دارد اما نمونه‌گیری آن وظیفه پیکربند ثالث شده است. پس این الگو نقض می‌شود.

Mediator: در این الگو، یک پیکربند ثالث نمونه‌گیری‌های لازم را انجام داده و پیوندهای delegation را تشکیل می‌دهد. استفاده کننده از میانجی مشتری و همکاران هستند. همچنین همکاران از طریق واسط با میانجی رابطه دارند. اما سازنده میانجی پیکربند است که اولویت پایین‌تری داشت. همچنین میانجی به برخی همکارها دید مستقیم و رابطه دارد ولی سازنده آن‌ها نیز پیکربند است که اولویت پایین‌تری داشت. پس این الگو نقض شده است.

Visitor: در این الگو، مشتری تمام زیرکلاس‌های وزیتور را می‌شناسد و از وظایف آن‌ها آگاه است و وظیفه نمونه‌گیری از ویزیتور و ارسال به ObjectStructure با مشتری است. اما المان‌ها دید مانا از ویزیتور داشته و از آن‌ها استفاده می‌کنند در نتیجه اولویت آن‌ها برای نمونه‌گیری بالاتر بود. پس این الگو محقق نشده است.

مقایسه: هر سه محقق نکرده‌اند بنابر توضیحات ارائه شده.

Low Coupling ۳-۱۹-۵

Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین این ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود. همینطور افزودن نوع جدیدی از محصول در این

الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد.

Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

۵-۱۹-۴ High Cohesion

Abstract Factory: در این الگو، وظیفه ساخت از دوش مشتری برداشته شده و اشیای مخصوص فکتوری تعریف شده‌اند که هرکدام خانواده اشیای مرتبط به هم را نمونه‌گیری می‌کنند و به این شکل انسجام افزایش یافته است. در واقع آن بخشی از سیستم که سرویس‌گیری انجام می‌دهد، از ساخت محصولات، ترکیب و پیکربندی آن‌ها و ساختار داخلیشان مستقل شده است. پس این الگو محقق می‌شود.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی مسئول تعامل‌ها است و هر Colleague صرفا وظایف تخصصی خودش را انجام می‌دهد، بدون آن‌که درگیر نحوه‌ی همکاری با سایر اشیا باشد. بنابراین، هر شی به‌نوعی منسجم‌تر می‌شود. میانجی به عنوان متخصص تعامل عمل می‌کند و این تفکیکِ وظایف بین تعامل و منطق تخصصی، باعث می‌شود تغییرات در سیستم آسان‌تر شده، قابلیت اصلاح[۱۹] افزایش یابد و در نتیجه نگهداری سیستم نیز ساده‌تر شود. یکی از مهم‌ترین خطرات این الگو این است که میانجی به راحتی ممکن است به یک God Class تبدیل شود. به این معنا که هنگام افزودن رفتار جدید به مجموعه، به‌جای توزیع مسئولیت میان اشیا همکار، ممکن است آن رفتار مستقیما در میانجی پیاده‌سازی می‌شود. در نتیجه، میانجی داده‌ها را از اشیا می‌گیرد و خودش منطق را اجرا می‌کند. این روند در بلندمدت می‌تواند میانجی را بسیار متورم و پیچیده کند و انسجام آن را به خطر بیندازد. وضعیتی مشابه آنچه ممکن است در الگوی Facade نیز رخ دهد.

Visitor: این الگو باعث می‌شود که عملیات مرتبط در قالب visitorهای تخصصی متمرکز شوند، در حالی‌که کارهایی که به ماهیت اصلی عناصر مربوط هستند، در خود عناصر باقی می‌مانند. به این ترتیب، Visitor وظیفه تفکیک مسئولیت‌ها را برعهده می‌گیرد. یعنی عملیات وابسته به پیمایش که به ذات عنصر مربوط نیست، به visitorها منتقل می‌شود و در نتیجه انسجام سیستم افزایش می‌یابد. اصولا اضافه کردن نقش‌هایی مانند متخصص به مجموعه کلاس‌ها، موجب افزایش انسجام می‌شود. با این حال، این الگو موجب افزایش جفت‌شدگی نیز می‌شود. زیرا عملیاتی که پیش‌تر درون خود کلاس‌ها انجام می‌شد، اکنون نیازمند تعامل با visitor است. این موضوع همچنین منجر به نقض encapsulation می‌گردد. با وجود این، مزیت مهم آن است که می‌توان به‌راحتی visitorهای جدید (به‌عنوان متخصص‌های تازه) تعریف کرد، بدون اینکه نیاز به کاهش انسجام در کد باشد.

---

[۱۹] modifiability

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

۵-۱۹-۵ Controller

Abstract Factory: در این الگو، وظیفه کارچرخانی و آغاز زنجیره‌های تعاملی برای رویدادهای وارد شده به سیستم از سمت بیرون، دیده نمی‌شود و از کنترلر استفاده نشده است.

Mediator: در این الگو، تمام همکاری و تعامل میان اشیا درون یک نقطه متمرکز می‌شود. یعنی الگوریتم تعاملی به‌جای آن که میان اشیا توزیع شده باشد، درون یک شی واحد میانجی جمع می‌شود. در این ساختار، میانجی به عنوان متخصص تعامل و کارچرخانی عمل می‌کند مانند Facade. پس این الگو محقق شده است.

Visitor: در این الگو، وظیفه کارچرخانی و آغاز زنجیره‌های تعاملی برای رویدادهای وارد شده به سیستم از سمت بیرون دیده نمی‌شود و از کنترلر استفاده نشده است.

مقایسه: بجز الگوی دوم در بقیه محقق نشده است.

۶-۱۹-۵ Polymorphism

Abstract Factory: در این الگو، از توارث برای تعریف انواع محصول و فکتوری استفاده شده است و می‌توان این دو موجودیت را رشد داده و انواع محصول و فکتوری اضافه کرد (البته با تبعاتی که قبلا توضیح داده شده است). مشتری که با واسط‌ها ارتباط دارد می‌تواند از فکتوری‌ها و محصولات متعدد استفاده کند.

Mediator: در این الگو، از توارث برای تعریف انواع همکار و میانجی استفاده شده است و می‌توان انواع همکار و میانجی اضافه کرد. مشتری صرفا با واسط میانجی ارتباط دارد و از داخل مجموعه بی‌خبر است و می‌تواند با میانجی‌های مختلف پیکربندی شود. همکارها نیز از طریق واسط با میانجی در ارتباط هستند اما میانجی در سطح زیرکلاس به صورت مستقیم با بعضی همکارها در رابطه است. به صورت کلی این الگو محقق شده است.

Visitor: در این الگو، از توارث در تعریف ویزیتور و المان استفاده شده است و می‌توان ویزیتورها و المان‌های مختلف تعریف کرد تا ساختار و رفتاری که قبلا توضیح داده شده است محقق شود. پس این الگو محقق می‌شود.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

Indirection ۷-۱۹-۵

Abstract Factory: در این الگو، مشتری به صورت مستقیم از محصولات نمونه‌گیری نکرده و این رفتار را به فکتوری مربوطه‌اش واسپاری می‌کند تا فکتوری از عوض مشتری نمونه‌گیری را انجام داده و به مشتری ارائه دهد و در این قسمت و به این منظور indirection ایجاد می‌شود. پس این الگو دیده می‌شود. مشتری پس از دریافت محصول از فکتوری، با استفاده از واسط محصول با آن کار می‌کند.

Mediator: در حالت قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. یک سطح indirection ایجاد شده و اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند. پس این الگو محقق می‌شود.

Visitor: در این الگو، عملیاتی که باید روی اشیای مختلف انجام شوند، در کلاس‌های جداگانه Visitor قرار می‌گیرند. هر شی از طریق متد accept، عملیات را به شی Visitor واسپاری می‌کند. این ساختار باعث می‌شود که کلاس‌های اصلی بدون نیاز به آگاهی از جزئیات عملیات، بتوانند رفتارهای مختلفی را پشتیبانی کنند. با این تفکیک، وابستگی مستقیم بین کلاس‌های اصلی و عملیات کاهش می‌یابد. بنابراین، این الگو از یک مکانیزم واسطه یعنی متد accept بهره می‌برد، پس می‌توان گفت این الگو به نوعی محقق شده است.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

Pure Fabrication ۸-۱۹-۵

Abstract Factory: در اثر اعمال این الگو، فکتوری‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهدف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس این الگو محقق شده است.

Mediator: در اثر اعمال این الگو، میانجی‌ها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس این الگو محقق شده است.

Visitor: در اثر اعمال این الگو، ویزیتورها شکل می‌گیرند که ناشی از تحلیل قلمرو مسئله نبوده و برای تحقق اهداف این الگو که قبلا ذکر شد، بوجود آمده‌اند. پس این الگو محقق شده است.

مقایسه: هر سه محقق می‌کنند.

Protected Variations ۹-۱۹-۵

Abstract Factory: در این الگو، مشتری با واسط‌های AbstractFactory و AbstractProductX در ارتباط است بنابراین در این قسمت DIP برقرار بوده و در نتیجه OCP برقرار بوده و مشتری ارتباط مستقیم با فکتوری‌های کانکریت و محصولات کانکریت ندارد و تغییرات آن‌ها به مشتری منتشر نمی‌شود. در نتیجه جفت‌شدگی در این قسمت مطلوب است. اما زیرکلاس‌های فکتوری به صورت مستقیم با زیرکلاس‌های محصولات ارتباط داشته و DIP در آن قسمت نقض می‌شود. کلاس‌های کانکریت مستقیما به یکدیگر وابسته‌اند و در کد آن‌ها نام یکدیگر آمده است که منجر به جفت‌شدگی شدید می‌شود. همچنین این ساختارهای توارثی موازی معمولا نشانه Bad Smell هستند، چراکه با هم رشد می‌کنند و تغییر در یکی، تغییر در دیگری را نیز به‌دنبال دارد. این تغییر منتشرشونده و نشان‌دهنده جفت‌شدگی بالای سیستم است. همچنین به این علت که مشتری صرفا با واسط‌ها در ارتباط است و زیرکلاس‌ها را نمی‌شناسد، یک پیکربند ثالث نیاز است تا ساختار delegation را بسازد و این پیکربند نیاز است که تمام زیرکلاس‌های فکتوری را بشناسد و این باعث جفت‌شدگی بالا در این قسمت شده و تغییرات به پیکربند منتشر می‌شود. همینطور افزودن نوع جدیدی از محصول در این الگو با دشواری همراه است، زیرا مستلزم تغییر در کد مشتری‌ها است. مشتری باید با واسط محصول جدید آشنا شود و به آن وابسته گردد. همچنین، این تغییر باعث گسترش واسط AbstractFactory و افزودن متد ساخت مربوط به محصول جدید می‌شود، که خود موجب تأثیرپذیری و تغییر در کلاینت‌ها خواهد شد.

Mediator: در این الگو، همکاران از طریق واسط با میانجی رابطه یک‌سویه دارند. در نتیجه در این قسمت DIP برقرار است و در این قسمت OCP برقرار بوده و تغییرات زیر میانجی به همکاران منتشر نمی‌شود. اما رابطه میانجی با همکاران در صورت نیاز تعریف می‌شود که این رابطه بین زیرکلاس‌های کانکریت بوده و از سمت میانجی به همکار است و در این قسمت DIP نقض می‌شود و تغییرات همکاران به میانجی منتشر می‌شود. قبل از اعمال الگو، با مجموعه‌ای از اشیا مواجه هستیم که به صورت پیچیده و درهم‌تنیده با یکدیگر تعامل دارند. این جفت‌شدگی و وابستگی شدید باعث می‌شود که هرگونه تغییر، به‌صورت گسترده و شدید در سراسر سیستم منتشر شود. در اثر اعمال این الگو، ارتباط Colleague ها غیرمستقیم می‌شود و جفت‌شدگی ضعیف خواهیم داشت، در واقع indirect coupling اتفاق می‌افتد که بهترین نوع جفت‌شدگی است. اشیا دیگر مستقیما با یکدیگر کار نمی‌کنند، بلکه تنها با میانجی در ارتباط‌اند و تغییرات همکارها به یکدیگر منتشر نمی‌شود. در این حالت، هرچند جفت‌شدگی میان اشیا کاهش یافته، اما همگی به میانجی وابسته شده‌اند، که خود نوعی جفت‌شدگی متمرکز ایجاد می‌کند. پس جفت‌شدگی در قسمتی بهبود پیدا کرده ولی به قسمتی دیگر منتقل شده است. همچنین پیکربند که در زمان اجرا باید ساختار delegation را تشکیل دهد، باید به تمام زیرکلاس‌ها دید داشته باشد. بنابراین جفت‌شدگی پیکربند با مجموعه بالا است و تغییرات آن‌ها به او منتشر می‌شود. مشتری نیز نوعا صرفا با واسط میانجی در ارتباط است و برای او DIP برقرار بوده، OCP برقرار

می‌شود و مشتری از تغییرات درون مجموعه منتزع می‌شود.

Visitor: در این الگو، مشتری به زیرکلاس‌های ویزیتور دید مستقیم داشته و در این قسمت DIP برقرار نیست. OCP نیز برقرار نبوده و تغییرات زیر ویزیتور به مشتری منتشر می‌شود و از آن تاثیر می‌پذیرد. پس جفت‌شدگی در این قسمت بالا است. همچنین مشتری به ObjectStructure دید داشته اما دیدی به المان‌ها و زیرکلاس‌های المان ندارد و از تغییرات آن‌ها متاثر نمی‌شود. ObjectStructure به واسط المان دید داشته و در آن قسمت DIP برقرار است و از تغییرات زیر المان تاثیر نمی‌پذیرد و جفت‌شدگی با آن‌ها ندارد. دید از سمت زیرکلاس‌های المان به ویزیتور در سطح فوق کلاس انتزاعی بوده و مستقیم به زیرکلاس‌های ویزیتور دید ندارند پس DIP در این قسمت برقرار بوده و OCP برقرار شده و از تغییرات زیر ویزیتور تاثیر نمی‌پذیرند و با آن‌ها جفت‌شدگی مستقیم ندارند. اما دید از سمت زیرکلاس‌های ویزیتور به زیرکلاس‌های المان به صورت مستقیم در سطح زیرکلاس وجود دارد و در این قسمت DIP نقض شده، OCP برقرار نیست و ویزیتورها از تغییرات زیر المان تاثیر پذیرفته و تغییرات به آن‌ها منتشر می‌شود و جفت‌شدگی در این قسمت بالا است. پس نمی‌توان به آسانی المان جدید اضافه کرد. اما می‌توان ویزیتور جدید اضافه کرد و تنها مشتری از آن تاثیر می‌پذیرد.

مقایسه: هر سه الگو در قسمت‌های بیان شده با استفاده از روش‌هایی که شرح داده شده است این الگو را محقق می‌کنند.

Bibliography

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

[2] J. R. V. J. Gamma Erich, Helm Richard. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] P. Coad. Object-oriented patterns. 2001.

Abstract

In this study, twelve design patterns were analyzed and evaluated using a criteria-based approach. The evaluation employed criteria derived from object-oriented principles and GRASP. This analysis aids in clarifying the characteristics, strengths, and weaknesses of each pattern.

Keywords: patterns in software engineering, design patterns, criteria-based evaluation

Sharif University of Technology
Department of Computer Engineering

Patterns in Software Engineering

First Assignment

By:

Mehdi Eidi

Professor:

Dr. Raman Ramsin

Spring 2025