بسمه تعالی

محمدرضا سابقی ۹۹۲۰۱۴۲۱

الگوها در مهندسی نرم افزار

استاد: دکتر رامسین

تمرین اول

بهار ۱۴۰۰

فهرست مطالب

۱ مقایسه Command-Iterator-Prototype

۱.۱ دسته

الگوی Command در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
الگوی Iterator در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
الگوی Prototype در دسته ی الگوهای آفرینشی یا Creational در GoF قرار دارد.
مقایسه: دو الگوی Command و Iterator در دسته ی مشابه به همی قرار دارند که هر دو از دسته ی رفتاری می باشند. در واقع تمرکز اصلی در این دو الگو بر تعاملات بین کلاس ها و اشیاء می باشد در حالی که الگوی Prototype با نمونه ساختن و پیکربندی کلاس ها و اشیاء بیشتر سروکار دارد و بدین جهت، از دسته ی آفرینشی است.

۲.۱ حوزه

الگوی Command در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی Iterator در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی Prototype در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
مقایسه: هیچ کدام از این ۳ الگو در زمان کامپایل و از طریق inheritance محقق نمی شوند. برای تحقق همه ی آن ها باید اجرا صورت بگیرد و به کمک delegation تحقق آن ها را ثابت کرد.

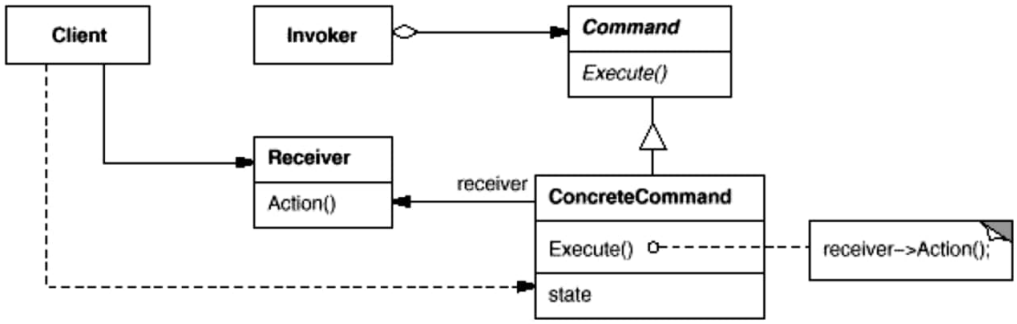
۳.۱ هدف

الگوی Command یک درخواست را به عنوان یک شیء کپسوله سازی می کند؛ که به این طریق می توان کلاینت ها را با انواع مختلف درخواست ها پارامتربندی کرد، درخواست ها را صف بندی یا log کرد و عملیات هایی که امکان برگشت(undoable) دارند را پشتیبانی کرد. به طور کلی پس هدف آن است که درخواست های مبادله شده در یک سیستم که آبجکت ها نسبت به هم خواهند داشت، به جای ازبین رفتن، مانا باشد.

الگوی Iterator امکانی را فراهم می آورد تا بدون افشای بازنمایی اجزای درونی شیءای مرکب، به صورت متوالی بتوان روی المان های آن پیمایش کرد و به آن ها دسترسی داشت.

الگوی Prototype در صدد این هست تا نوعی از اشیاء را مشخص کند که با استفاده از تعدادی نمونه اولیه ساخته شود و در واقع اشیاء جدید را با استفاده از کپی کردن این نمونه اولیه می سازد.

مقایسه: تقریبا اهدافی که این سه الگو در جهت برآورده شدن آن هستند از یکدیگر کاملا متفاوت است. الگوی Prototype برای نمونه گیری از اشیاء به کمک کپی کردن از نمونه های اولیه می باشد، در حالی که Iterator تمرکز خود را روی حفظ محرمانگی اجزای داخلی شیء مرکب در عین پیمایش روی المان های آن گذاشته است. الگوی Command نیز کلا برای دیدن درخواست به شکل یک شیء است تا مانایی آن را برقرار سازد و باعث شود که درخواست ها در سیستم به سرعت از بین نروند و بتوان روی آن ها عملیاتی داشت که به بهبود برنامه کمک می کند.



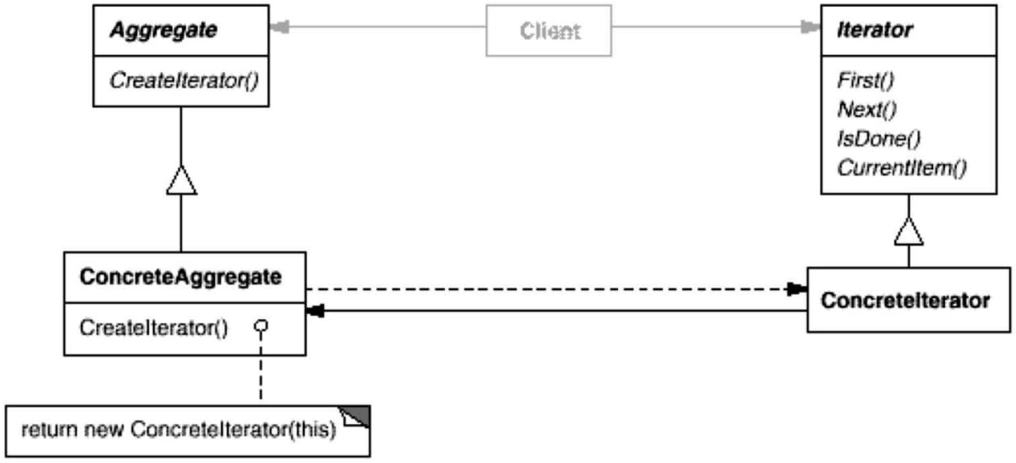
شکل ۱: ساختار الگوی Command

۴.۱ ساختار

ساختار الگوی Command را در شکل ۱ داریم: در ساختار کلاسی این الگو کاملا مشخص است که به هنگام اجرا و delegation بین اشیاء محقق می شود. در این ساختار مشخص است که کلاینت نسب به کلاس های سطح پایین Command و همچنین کلاس Receiver دید دارد. کلایت که پیکربند مجموعه نیز نامیده می شود، ابتدا concreteCommand خود را می سازد و receiver آن را نیز پاس می دهد تا این کلاس سطح پایین Command بتواند سرویس دهنده ی خود را بشناسد. سپس برای Invoker مشخص می کند که شیء command مربوط به آن کدام است؛ چرا که هر receiver با یک کلاس command در ارتباط است. رفتار الگو زمانی محقق می شود که شیء invoker دستور execute را روی شیء command فرا می خواند. حال دستوری در سیستم داریم که در آن مانا است و او می داند که به کدام receiver مراجعه کند و بدین ترتیب متد action را روی شیء receiver مرتبط به خود، فراخوانی می کند.

کما اینکه از منظر Invoker متوجه می شویم اصل DIP در آن برقرار است؛ چرا که این کلاس باکلاس انتزاعی Command در ارتباط است و از لایه ی زیرین کلاس های Command منتزع می باشد. از طرفی اما، شیء ثالث رفتار که کلاینت باشد اینگونه نیست.

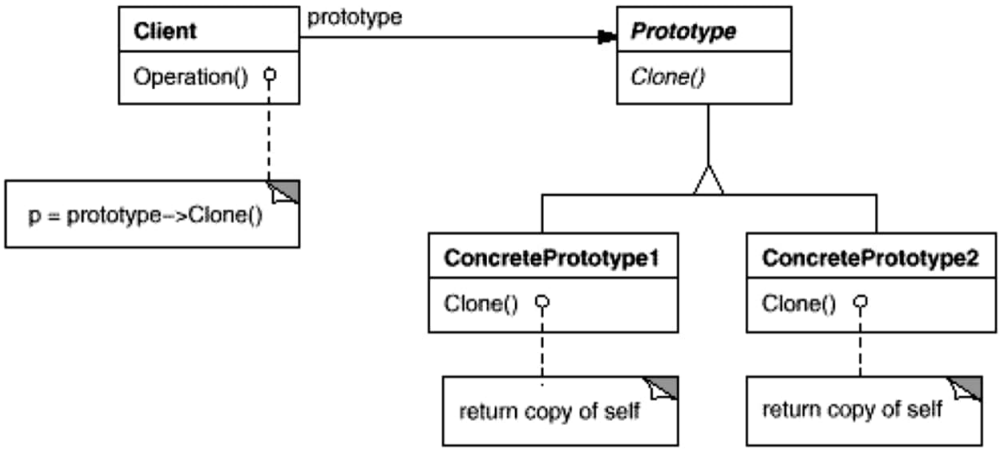
الگوی Iterator ساختاری به مانند شکل ۲ دارد. دقت داریم که در سطح



شکل ۲: ساختار الگوی Iterator

بالای کلاس ها، کلاینت به هر دو کلاس دید دارد که بدین معناست که به صورت مستقل، یک مشتری بیرونی داخل سیستم می تواند با کلاس های مرکب یا کلاس های پیمایشگر کار کند. اما در بحث رابطه ی بین کلاس های مرکب و کلاس های پیشمایشگر، به سطح غیرانتزاعی می آییم. کلاس های concrete با یکدیگر رابطه دارند و این رابطه به گونه ای است که شیء کلاس مرکب، می داند و می خواهد که پیمایشگری بسازد تا پیمایش را انجام دهد. پس به کمک operation ی به نام CreateIterator این کار را انجام می دهد و در این راستا خود را نیز پاس می دهد تا پیمایشگر ساخته شده، آن را بگیرد و با دید مانایی بداند که روی کدام شیء مرکب قرار هست تا پیمایش انجام شود و عملیات خود را به کار گیرد.

پس نکته اصلی در آن است که در ساختار توارثی دو طرف، اصل DIP دیده نمی شود و اشیاء ConcreteAggregator مسئول ساخت پیمایشگر و پاس دادن خود هستند. بدین ترتیب اما در نهایت توانسته ایم به کمک شیء پیمایشگر مخصوص به یک شیء مرکب، بدون افشای اجزای داخلی آن، روی المان های آن حرکت کنیم و آن ها را مورد پایش قراردهیم. این الگو نیز کما اینکه از ساختار و رفتار آن بر می آید، الگویی که است که بالقوه در زمان کامپایل هنوز محقق نشده است.



شکل ۳: ساختار الگوی Prototype

الگوی Prototype نیز ساختاری مشابه با شکل ۳ دارد. ساختاری را در Prototype می بینیم که ایده ی ساده ای در خود دارد. به هر کلاس مشتری که نام آن را کلاینت می گذاریم و می گوییم متقاضی و مشتری کلاس دیگری است تا به نحوی به آن دید داشته باشد، یک کلاس Prototype انتساب داده می شود که به کمک آن می تواند یک کپی از آن شیء کلاس ConcretePrototype بگیرد. طبیعی است که در نسبت دید کلاینت، اصل DIP برقرار است؛ چرا که از کلاس های سطح پایین Prototype منتزع است و کلاس سطح بالای آن را می شناسد.

در هر کلاسی که پیکربند به عنوان Prototype به مشتری نسبت می دهد، متد Clone گذاشته می شود که وظیفه ی آن تنها برگرداندن یک کپی از خود شیء کلاس است. این کپی در اختیار کلاینت این درخواست قرار می گیرد تا نسبت به آن کلاس ConcretePrototype دید پیدا کند. با این کار جلوی ساختارهای موازی توارثی گونه گرفته می شود که بعدا به آن خواهیم پرداخت. این الگو نیز همانطور که از توضیحات پیدا می باشد، در زمان اجرا محقق خواهد شد.

مقایسه: هر سه تای این الگوها به هنگام اجرا، رفتار خود را بروز می دهند و از ساختار پیدا می باشد که در زمان کامپایل، ثابت نیستند. الگوی Command

با نیاز به کلاینت جهت برقراری روابط به عنوان شخص ثالث، ساختار خود را نشان می دهد ولی الگوی Iterator کاری به آبجکت های ثالث بیرونی ندارد. الگوی Prototype نیز پیکربندی نیاز دارد تا prototype ی را به کلاینت نسبت دهد. اصل DIP به نسبت در آن ها رعایت شده است اما Iterator در بین این ها مستثنی است. در آخر آن که در هر سه الگو، ساختار توارثی دیده می شود که این ساختار کمک شایانی به Iterator به دلیل پتانسیل رشد درخت گونه ی موازی توارث، نمی تواند کمک شایانی کند.

۵.۱ پیکربندی

الگوی Command در نحوه ی پیکربندی خود متکی بر کلاینت هست. بدین معنا که در ابتدای کار به عنوان آبجکت ثالث که نام آن را مشتری می گذاریم نیاز داریم تا پیکربندی انجام شود تا زمان بعدی که نیاز به پیکربندی داشتیم. پیکربندی توسط کلاینت می تواند در زمان اجرا شده و شیء کلاینت کاری که می کند آن است که شیءای از کلاس Command می سازد و شیء receiver مربوط به آن را پاس می دهد تا بدین ترتیب پیکربندی بین یک درخواست با سرویس دهنده ش شکل بگیرد. سپس شیءای از invoker می سازد و درخواست را به آن پاس می دهد تا بین درخواست کننده با درخواست نیز پیکربندی تشکیل شود. همه ی این ها در حالی است که کلاینت می تواند در هر لحظه از اجرا، آبجکت سرویس دهنده را برای command و خود آبجکت command را برای invoker تغییر دهد.

الگوی Iterator در نحوه ی پیکربندی به شکلی عمل می کند که کلاینت با فراخوانی متد CreateIterator در آبجکت مرکب سعی دارد تا پیمایشگر مخصوص را بسازد و دید آن را نیز به این آبجکت فراهم کند. البته ناگفته نماند که این کار میسر هست؛ چرا که کلاینت نسبت به هر دو دید دارد پس می تواند این فراخوانی را روی یک شیء aggregate انجام دهد. دید مانای روی شیء مرکب از طریق پاس دادن خود شیء مرکب به سازنده ی Iterator متخصص انجام می شود. حال که همه چیز مهیا می باشد، کلاینت که نیز از

قبل پیمایشگرها را می شناسد می تواند با آن ها کار کند.

الگوی Prototype در چگونگی پیکربندی خود نیز از یک آبجکت ثالث که وظیفه ی پیکربندی را برعهده دارد استفاده می کند. بدین صورت که پیکربند، شیء کلاینت را با یک شیء دیگر از کلاس های concrete از Prototype ها منتسب می کند و در هر لحظه از زمان اجرا می تواند نوع شیء انتسابی را تغییر دهد.

مقایسه: الگوها هر سه به پیکربند نیاز دارند. در Command این وظیفه به شرحی که گذشت به آبجکت کلاینت سپرده می شود که از این نظر تشابهی با Iterator دارد. اما Prototype پیکربند خود را الزاما کلاینت نمی داند و وابسته به آبجکتی ثالث است. در این مقایسه، الگوی Iterator به مراتب از نظر پیکربندی نسبت به Command ساده تر است. چرا که در Command مشتری باید هم آبجکت receiver را بشناسد و آن را به آبجکت دستور ایجاد شده بشناساند و هم باید آبجکت دستور را برای شیء invoker معرفی کند.

۶.۱ انعطاف پذیری

الگوی Command همانطور که پیشتر هم بیان شد، میان دو کلاس In-voker و Command اصل DIP را برقرار کرده است. بدین گونه که Invoker تنها کلاس انتزاعی Command را می بیند و از کلاس های زیرین آن که فرزند آن به حساب می آیند، اطلاعی ندارند. این دید منتزع شده که باعث می شود برای Invoker بتوان هر زمان نوع Command را عوض کرد باعث انعطاف پذیری در الگو می شود. این درحالی است چنین دیدی بین Command و Re-ceiver وجود ندارد. کما اینکه بیان شد، آبجکت سومی که آن را کلاینت نام گذاری کرده ایم شیء command را با پاس دادن receiver ی که به آن دید دارد پیکربندی می کند. در اینجا کلاس های غیرانتزاعی درگیر هستند که نمی تواند کمک آن به انعطاف پذیری در این حد هست که بتوان سرویس دهنده به command را در حین اجرا بدون انتشار تغییر عوض کرد. در نهایت آن که در کنار این بازپیکربندی ها، کپسوله شدن یک درخواست در یک کلاس و

ارتباط آن با فراخواننده از طرفی و سرویس دهنده از طرف دیگر کمک شایانی به انعطاف پذیری می کند؛ بدین گونه که می توان روی درخواست، عملیات تعریف کرد و با مانا کردن آن، عملیات هایی مانند صف بندی، برگشت عمل و log کردن عمل را اجرایی کرد.

الگوی Iterator انعطاف پذیری نسبتا بالایی دارد؛ بدین گونه که کلاینت کافی است اراده کند تا پیمایشگری جدید ساخته شود و آن پیمایشگر دید مانایی را به شیء مرکب پیدا کند. ارتباطی که کلاینت با شیء مرکب و شیء پیمایشگر دارد در سطح بالایی است و کلاینت از کلاس های فرزند هیچکدام مطلع نیست و از آن ها منتزع می باشد. در نتیجه اگر کلاسی در زیرکلاس های پیمایشگر تغییر کند یا اضافه شود سمت کلاینت، خطری برای تغییر وجود ندارد. همچنین است برای کلاس های مرکب و کلاینت. این بازپیکربندی در این سطح می تواند مطلوب باشد که موجب افزایش انعطاف پذیری می شود، اما از طرفی دقت داریم که رابطه ی بین کلاس های مرکب و کلاس های پیمایشگر در سطح غیرانتزاعی قرار دارد؛ بدین صورت که هر کلاس concrete مرکب باید خود بداند که دقیقا از کدام کلاس concrete پیمایشگر می خواهد شیء بسازد و باید آن ها را در سطح پایین بشناسد که باعث افت در انعطاف پذیری و نقض اصل DIP می شود. پس متوجه هستیم که این الگو هر چند قابلیت گسترش و استفاده ی مجدد را تا حدی از طریق وابستگی های سطح بالا و برقراری توارث ایجاد می کند اما وابستگی های سطح پایین زیرکلاس ها به یکدیگر، نمی تواند انعطاف پذیری خیلی بالایی را به آن بدهد.

الگوی Prototype در منعطف بودن سطح بالایی از خود نشان می دهد؛ زیرا کافی است تا هر کلاس زیرکلاس Prototype که قطعا متد clone را پیاده کرده و در آن کپی خودش را برمی گرداند، توسط پیکربند به کلاینت متقاضی Prototype منتسب شود. لذا دید بسیار خوبی در سطح بالا برقرار است که این دید از کلاینت به کلاس انتزاعی Prototype در سطح بالا تعریف شده است. پس متوجه آن هستیم که با هر گونه تغییر در اندازه ی زیرکلاس ها و یا خود زیرکلاس ها، تغییری متوجه کلاینت نخواهد بود و کلاینت منتزع از اتفاقات زیرین کلاس Prototype می باشد که باعث رعایت اصل DIP می شود و کمک

شایانی به بازپیکربندی شیء concrete از نوع Prototype داده شده به کلاینت در حین اجرا و بدون مشکل می کند.

مقایسه: هنگامی که به مقام مقایسه بین این سه الگو برمی آییم متوجه آن هستیم که الگوی Prototype حد زیادی از انعطاف پذیری را رعایت کرده است و این کار را به سهولت و با قرار دیدن دید کلاس ها به هم در سطح بالا فراهم کرده است. در مرتبه ی بعدی می تواند الگوی Command باشد که چنین امکانی را در بخشی از الگو آورده است اما از طرفی، آبجکت سومی که پیکربند مجموعه است، دید خود را روی کلاس های concrete گذاشته است. در آخر آن که الگوی Iterator نیز از نظر انعطاف پذیری، نسبتا خوب عمل کرده و وابستگی سطح بالایی بین کلاینت خارجی مجموعه و با اشیاء مجموعه وجود دارد اما ارتباطات خود اشیاء مجموعه به کلاس های concrete محدود می شود.

۷.۱ کارآیی

الگوی Command تاثیر خود روی کارآیی را زمانی نشان می دهد که بدانیم تا پیش از این، یک درخواست فقط قابلیت این را داشت که پردازش شود و سپس از بین برود. اما الگوی Command می تواند به آن به چشم یک شیء نگاه کند و برای آن داده_رفتارهایی تعیین کند. دقت داریم که این کار به طور کلی می تواند در کنار قابلیت هایی که به همراه دارد، تاثیر منفی ای را روی میزان حجم مصرفی داشته باشد؛ چرا که در برنامه، تعداد اشیایی موسوم به شیء command افزایش پیدا کرده اند. پس این واسطی که بین فراخواننده و سرویس دهنده بوجود آمده است، سرباری را متوجه سیستم می کند. وجود اشیاء زیاد درخواست به همراه ذخیره ی وضعیت های قبل، تاثیر بر روی حافظه و log کردن یا پیکربندی ها تاثیری روی زمان خواهند داشت که تاثیری غیرمثبت می باشد.

الگوی Iterator تمرکزش را روی پیمایش عناصر می گذارد بدون آن که اجزای داخلی شیء در معرض دید قرار بگیرند. لذا کلاسی را به لزوم تعریف می

کند که خارج از کلاس مرکب است و اشیاء آن متخصص چنین نیتی هستند. در این راه پیمایشگرها دید کاملی را روی شیء مرکب باید برقرار کنند تا در واقع نائب کلاینت باشد. پس نیاز به ارسال و دریافت زیادی پیغام بین آبجکت ها خواهد بود که تاثیر منفی را روی زمان حالت اجرا خواهد بود. جدای از آن، با ایجاد ساختار توارث گونه ی کلاس های پیمایش کننده، اشیاء زیادی بالقوه می توانند ایجاد شود که تاثیر منفی را روی حافظه نشان می دهد.

الگوی Prototype راه حلی است برای عیوب الگویی مانند Fac- Method tory که می خواهد بدین وسیله آبجکت های جدید را با کپی نمونه های اولیه بسازد. به نظر می رسد که تنها معضلی که بتوان برای آن در حوزه ی کارآیی دانست(که اغلب الگوها از آن رنج می برند) واگذاری های حین اجرا می باشد که در این جا تاثیر منفی را روی عملکرد از نظر زمانی می گذارد؛ چرا که کلاینت در Operation کار را به آبجکت Prototype ی که می شناسد واسپاری می کند و این delegation ها به طور کلی اثر منفی را بر کارآیی زمانی دارند. البته در هر صورت عوامل دیگری مانند خود زبان برنامه نویسی می توانند در این فاکتور موثر باشند و لازم به ذکر است که نمونه گیری اشیاءی اولیه و جمع شدن در pool نیز زمان خاص خود را می طلبد.

مقایسه: کارآیی در هر سه به صورت کلی نزولی است. بدین منظور الگوی Command را می توان شدیدتر از بقیه دانست و کاهش محسوس تری برای آن قائل شد. الگوی Iterator را می توان در مرتبه ی بعدی دانست؛ جایی که شاید کارآیی روی حافظه و زمان باشد اما به شدتی که در Command هر درخواست به شیء تبدیل شده است، نمی باشد. تنها تعدادی زیرکلاس و تبادل تعدادی پیغام، کارآیی را تحت تاثیر قرار می دهد. در آخر نیز Prototype که الگویی با راه حل متداولی است که سربار کمتری را با خود به همراه می آورد.

۸.۱ وابستگی

الگوی Command وابستگی مستقیم بین Invoker و Receiver را شکسته است. می دانیم که قبلا رابطه این دو به صورت مستقیم بوده است ولی با راه حل این الگو، این وابستگی، غیرمستقیم می شود و وابستگی تا حدودی از بین می رود. همانطور که قبلا نیز بیان کردیم، اصل DIP برای Invoker و Command برقرار هست؛ چون وابستگی بین آن ها در سطح انتزاع هست و Invoker با واسط یک دستور در ارتباط است که خود این امر وابستگی میان آن ها را به حداقل می رساند و از انتشار تغییرات جلوگیری می کند و همچنین امکان گسترش مستقل از هم را می دهد. نکته ی منفی ای که در الگو به چشم می خورد این است که کلاینت با کلاس های مجرد Command به صورت مستقیم ارتباط دارد و آن ها را می شناسد. کلاینت که پیکربند مجموعه می باشد برای Receiver و Invoker نیز اینگونه عمل می کند. پس باید از تمام تغییرات در آن ها مطلع باشد که این وابستگی نسبتا شدیدی را نشان می دهد. البته از آنجایی که وظیفه ای در حد پیکربندی مجموعه دارد، این وابستگی دائمی نیست و تا پیکربندی بعدی مورد استفاده قرار نخواهد نگرفت که باعث می شود در کل برای این الگو، وابستگی را مثبت ارزیابی کنیم.

الگوی Iterator زمانی که به کلاینت نگاه می کند، وابستگی شدید او به شیء مرکب را شکننده می بیند؛ چرا که کلاینت دیگر مجبور نیست زیرکلاس های شیء مرکب را بشناسد و بداند که چگونه با همه ی آن ها کار کند و پیمایش کند. رابطه ی او با شیء مرکب به سطح انتزاع رفته است و همین وابستگی را در رابطه با Iterator نیز تجربه می کند که منتزع از تغییر در زیرکلاس های آن است(وجود اصل DIP در این سطح از الگو). از طریق واسط می داند که چگونه باید با آن ها کار کند. اما اگر نگاهی به رابطه ی بین اشیاء مرکب و پیمایشگرها بیاندازیم متوجه می شویم که وابستگی در سطح تجرید و بسیار قوی است. این می تواند وابستگی را برای این الگو نامطلوب جلوه دهد.

الگوی Prototype توانسته است که ارتباط بین کلاس های فرزند Proto-type را با کلاینت، بوسیله ی واسط Prototype ازبین ببرد و به DIP پایبند

باشد. اصل DIP که نیز برای مجموعه برقرار باشد گویا وابستگی خوبی در مجموعه اشیاء فراهم شده است. پیکربند خارجی اما باید همه ی کلاس های concrete را بشناسد و از کم و زیاد شدن آنان آگاه باشد. با این کار، ارتباط کلاینت به نمونه های اولیه کاهش یافته و پیکربند این انتساب را عهده دار می شود تا به کمک واسط ،Prototype کلاس مجردی برای کلاینت در نظر بگیرد.

مقایسه: اصل DIP در هر کدام در سطحی به کار رفته است که خب کاهش وابستگی را نتیجه می دهد. در قسمت هایی از Command و Prototype این پیکربند هست که وابستگی های مجرد را باید داشته باشد که باعث می شود تا وابستگی تحت تاثیر قرار بگیرد. اما در Iterator این رابطه ی سطح پایین کلاس های پیمایشگر و شیء مرکب هستند که وابستگی را تحت الشعاع قرار می دهند ولی پیکربند آن را اگر کلاینت در نظر بگیریم، سطح انتزاع خوب آن سبب کاهش وابستگی می شود.

۹.۱ همبستگی

الگوی Command با کپسوله سازی درخواست و گرفتن آن به عنوان کلاس و ایجاد اشیاء از آن نشان می دهد که انسجام خوبی را دارد. دقت داریم که با این کار، کلاس Command نوعی آبجکتی به شمار می آید که یک purpose را دنبال می کند و همین هدف برای Invoker و Receiver به خوبی محقق می شود و آن ها نیاز شدیدی به رابطه و دانستن یکدیگر نداشته و بیشتر روی کار خود تمرکز دارند. هر گونه ارتباطی از طریق متد execute فراهم خواهد بود که این سبب افزایش همبستگی در مجموعه ی الگو شده است.

الگوی Iterator اینگونه همبستگی را فراهم می آورد که یک متخصص برای پیمایش بدون افشای محتوای شیء پیشمایش شونده در نظر می گیرد و سعی می کند تک‌ـکار را در شیءای کپسوله کند. زمانی که این الگو به کار گرفته می شود تمامی عملیات مربوط به پیمایش در آبجکت پیمایشگر خواهد بود و اینگونه انسجام بسیار بالایی فراهم است. ضمن آن که آبجکت مرکب نیازی به در معرض دید قرار دادن اجزای خود ندارد.

الگوی Prototype تنها می تواند از این جنبه، همبستگی را بروز دهد که هر کلاسی که زیرکلاس Prototype می باشد، متد Clone را در خود دارد که در آن یک نمونه کپی از خودش برمی گرداند و خیلی کار خاصی را انجام نمی دهد. یعنی آن که اگر کلاسی زیرکلاس Prototype شود، آنگاه فقط کافی است که این پیاده سازی را انجام دهد و سپس پیکربند به آن آگاه است و می تواند آن را به یک کلاینت مشتری، منتسب کند تا کلاینت از Clone آن استفاده کند.

مقایسه: کار مهمی در Prototype برای همبستگی صورت نگرفته است؛ اما Iterator می تواند با ارائه ی توضیحات بالا، سطح خوبی از همبستگی را با قرار دادن آبجکت های متخصص پیمایش که عملیات پیمایش را کپسوله می کنند فراهم کند. همچنین برای الگوی Command گفته شد که انسجام بالای آن به دلیل کپسوله سازی داده و رفتار یک آبجکت و شکستن ارتباط مستقیم Invoker و Receiver می باشد که باعث می شود آبجکت ها روی کار(هدف) خود تمرکز کنند و وابسته نباشند.

۱۰.۱ اثرات جانبی

الگوی Command کما اینکه پیشتر نیز بیان شد مشکل اضافه کردن تعداد اشیاء ـ به دلیل تبدیل یک دستور ساده به یک آبجکت ـ را با خود به همراه دارد. این رشد تعداد آبجکت ها به همراه افزایش اطلاعات برای نگهداری وضعیت های قبلی، از اثرات جانبی این الگو می باشد. در کنار آن وجود امکانی مانند log کردن دستورهای پردازش شده و همچنین پیکربندی هر ـ از ـ چندگاه می تواند سربار زمانی را به همراه الگو داشته باشد. در ادامه نیز خواهیم گفت که عدم سادگی در پیاده سازی سبب آن شده است که این الگو برای پیاده شدن، سخت به نظر بیاید؛ چرا که درک و فهم کد را می تواند پیچیده کند و اثر بدی را از خود به جای بگذارد.

الگوی Iterator پیاده سازی به نسبت ساده تری دارد و پیچیدگی عجیب غریبی را به مجموعه تحمیل نمی کند. با اعمال این الگو باید مراقب رشد موازی ساختار های توارثی بود تا از وجود تعداد کلاس های زیاد در امان بود.

وجود ارتباط های تجرید در بین کلاس ها و اشیاء پیمایشگر_مرکب سطح زیاد وابستگی را برای سیستم خواهد شد که شاید افت کیفیت اجرایی از نظر زمان را به همراه داشته باشد. این الگو با این حال که اجازه ی فعال بودن پیمایش چندین تایی را روی یک شیء مرکب در هر لحظه فراهم می کند و دید کلاینت را از داخل شیء مرکب مخفی می کند اما باید از نظر سربار حافظه ای نیز مراقب باشد تا به مشکل بالقوه ی الگوی Command برنخورد.

الگوی Prototype اثرات جانبی خاصی را به همراه ندارد. بدین معنا که نه می تواند از نظر پیچیده سازی کد کاری انجام دهد و نه سربار حافظه ی خاصی را می تواند داشته باشد. تنها شایع است که به دلیل اجازه ی پیکربند در هر لحظه برای تغییر شیء prototype منتسب به کلاینت، سربار زمانی در حین اجرا داشته باشد. امکان کم و زیاد کردن نمونه های اولیه در حین اجرا و کاهش وابستگی کلاینت به زیرکلاس های Prototype از فواید الگو است که اثر جانبی خاصی را متوجه مجموعه نمی کند.

مقایسه: الگوی Prototype جز delegation و پیکربندی که زمانی را صرف می کند، اثر جانبی خاصی را از خود بروز نمی دهد. الگوی Iterator و Com- mand در معرض خطر های بیشتری هستند. مخصوصا الگوی Command که علاوه بر ایجاد مشغولی های حافظه و زمان اجرا، به پیچیده تر شدن ساختار پیاده سازی، ممکن است بیانجامد.

۱۱.۱ کپسوله سازی

الگوی Command همانطور که قبلا نیز بیان شد، کپسوله سازی را به شکل خوبی در میان کلاس های Command و Invoker فراهم کرده است. کپسوله سازی از این جهت است که Invoker فقط متد execute را اجرا می کند و سپس بدون دانستن از اینکه کدامین شیء دستور می خواهد اجرا شود، به کار خود ادامه می دهد؛ سپس شیء دستور مرتبط، به سرویس دهنده ای کار را واگذاری می کند و خود دخالتی در پردازش ندارد. بدین سبب، کپسوله سازی بسیار خوبی در الگو جاری شده است.

الگوی Iterator هم به دلیل مخفی نگه داشتن اطلاعات داخلی شیء مرکب از دید کلاینت و انحصار دید به پیمایشگری که خود به صورت تخصصی در یک کلاس جمع شده است و کپسوله سازی به صورت خوبی فراهم است، کپسوله سازی را به شکل مطلوبی پوشش داده است.

الگوی Prototype تقریبا کار خاصی را در بحث کپسوله سازی انجام نمی دهد جز آن که تمامی کلاس هایی که زیرکلاس Prototype هستند متد مشترک Clone را در خود پیاده سازی کرده اند.

مقایسه: در بحث مقایسه ی بین این سه الگو کاملا مشخص است که الگوی Command بسیار خوب توانسته است تا کپسوله سازی را فراهم کند و وابستگی ها را کاهش و همبستگی را افزایش دهد؛ الگوی Iterator نیز با آوردن شیء متخصص و پوشاندن اجزای محرمانه توانسته به این ویژگی دست یابد و در نهایت Prototype که کاری را در جهت نقض کپسوله سازی انجام نداده است.

۱۲.۱ انتشار تغییرات

الگوی Command تغییرات منتشر شونده را اینگونه مهار می کند که با آمدن Command در وسط فراخواننده و سرویس دهنده، ارتباط این دو از مستقیم به غیرمستقیم شکسته می شود و بدین ترتیب تغییر در یکی سبب انتشار تغییر به دیگری نمی شود. منتزع بودن دید سرویس دهنده از نوع آبجکت مجرد Command کمکی بر این موضوع است. فقط می توان گفت تغییرات در جایی منشتر می شوند که تغییری در زیرکلاس های Command بوجود بیاید که کلاینت به دلیل وابستگی باید از آن مطلع شود و به نسبت هم دچار تغییر می شود؛ و یا این موضوع صادق است برای تمام کلاس هایی که پیکربند کلاینت به آن ها دید دارد.

الگوی Iterator تغییرات را از دید کاربر برای پیمایشگرها و آبجکت های مرکب مخفی نگه داشته است که این می تواند نقطه ی قوتی باشد تا در صورت تغییر در زیرکلاس ها، کلاینت از آن بی خبر خواهد بود و لزومی به آگاهی ندارد. نکته ی منفی اما فقط در وابستگی سطح پایین کلاس های concrete پیمایشگر و آبجکت مرکب هستند. در این جا، لازم است تا تغییر در هر کلاس مجرد Aggregate به کلاس مرتبط مجرد Iterator منتشر شود تا از آن آگاه باشد. تغییرات به صورت بالعکس نیز نیاز است. پس درست است که کلاینت می تواند با انواع مختلف از هر دو کار کند اما خود آن ها در رابطه ای که ندارد نمی توانند اینگونه باشند و علاوه بر خود آن ها، پیکربند نیز از تغییرات آگاه می شود.

الگوی Prototype شرایطی را فراهم آورده تا کلاینت بتواند با هر proto-type ی که به او توسط پیکربند نسبت داده می شود کار کند و از این نظر هیچ آگاهی نسبت به زیرکلاس ها ندارد و فقط با واسط Prototype در ارتباط است. کم و زیاد شدن و تغییر در زیر کلاس ها، تاثیری برای کلاینت ندارد؛ پس تغییر منتشر شونده ای در آن وجود ندارد و رابطه ی کلاینت و Prototype در سطح بالا شکل گرفته است. به مانند قبل فقط نیاز هست تا پیکربند از تغییرات زیرکلاس های Prototype آگاه باشد.

مقایسه: الگوی Prototype فقط خبر تغییر در کم و زیاد شدن زیرکلاس های Prototype را به پیکربند منتشر می کند. اما الگوی Iterator نمی تواند انتشار تغییر را فقط منوط به این بداند و تغییر در کلاس های سطح پایین هر یک از پیمایشگرها و یا آبجکت های مرکب، تغییر در دیگری را هم به همراه خواهد داشت. الگوی Command اما می تواند بهتر عمل کند و فقط تغییرات منتشرشونده را در کلاینت که پیکربند مجموعه است محدود کند و به علاوه اینکه این انتشار خیلی دائمی نخواهد بود و موقتی است.

۱۳.۱ میزان استفاده از منابع سیستمی

الگوی Command بنا بر دلایلی که برای آن برشمردیم در میزان استفاده از منابع سیستمی پتانسیل بالایی دارد تا فشار زیادی را وارد کند. همه ی این ها به خاطر این هست که تا پیش از این الگو، یک دستور فقط به صورت یک درخواستی بود که در یک متد می توانست با صدا زدن آن انجام شود؛ اما این الگو مبحث شیء دستور را مطرح کرد که بالقوه می تواند باعث زیاد شدن آبجکت های سیستمی و افزایش مصارف حافظه ای بدلیل ثبت و ذخیره ی وضعیت های قبلی است. به علاوه آنکه با عملیات هایی نظر logging و یا پیکربندی نسبتا طولانی تر هر_از_چندگاه می تواند منابع سیستم را برای پردازش، درگیر خود کند.

الگوی Iterator نیز به دلیل مطرح کردن آبجکت هایی متخصص برای کار پیمایش بدون درز اطلاعات به کلاینت، توانایی فشار آوردن به منابع حافظه ای سیستم را دارد. وجود ساختارتوارثی آن و همچنین ساختار توارثی کلاس های Aggregate بالقوه می تواند معضلی برای تشدید مصرف حافظه و سرعت باشد؛ چرا که نگرانی از وجود تعداد زیاد اشیاء وجود خواهد داشت. پس وجود لحظه ای اشیاء مرکب به همراه هر تعداد از پیمایشگر نظیرشان، نشان دهنده ی مصرف منابع سیستمی در آن لحظه خواهد بود. ارسال و تبادل پیام بین کلاس های غیرانتزاعی که پیشتر صحبت آن را کردیم نیز می تواند عاملی برای درگیری منابع سیستمی باشد، چرا که نیاز به پردازش دارند.

الگوی Prototype در بحث شدت استفاده از حافظه تاکید بیشتری از خود نشان می دهد. بدین صورت که اگر در ابتدای اجرای برنامه، اشیاء نمونه های اولیه را بسازیم(یعنی از هر کدام اشیاء خود را بسازیم) و به پیکربند این اجازه را بدهیم تا از میان این استخر که در حال نگه داری از اشیاء است بتواند پیکربندی را با کلاینت انجام دهد، آنگاه مصرف قوی حافظه بسیار محتمل می نماید. با این کار می توان با هر Clone درواقع کپی شیء را برگرداند اما این رشد اولیه تعداد اشیاء در برنامه، باعث استفاده ی زیادتر از منابع سیستم می شود.

مقایسه: هر کدام از الگوها به نوعی مصرف خاص خود را از منابع سیستم دارند. الگوی Iterator همانطور که توضیح دادیم، عیان تر این موضوع را در استفاده از حافظه از خود نشان می دهد. اما الگوی Command به شرحی که گذشت و با قیاس با حالت before آن کاملا این موضوع روشن است. الگوی Iterator نیز با جلب کردن توجه ها به ساختار موازی گونه توراثی خود، احتمال فشار بر روی حافظه و سرعت را زیاد می کند.

۱۴.۱ استفاده از شیء جعلی

الگوی Command دو کلاس Invoker و Receiver را در خود مساله می بیند اما با افزودن کلاس Command که ارتباط این دو را غیرمستقیم می کند و دستور ها را تثبیت می کند، از تحلیل یک قلمرو مساله بدست نیامده است؛ که نشان می دهد کلاسی جعلی است و درنتیجه ی اعمال الگو پدید آمده است.

الگوی Iterator واسط Iterator و کلاس های فرزند آن را به عنوان کلاس جعلی به مجموعه اضافه می کند. بدین صورت که در فضای مساله، چنین موجودیتی مورد تحلیل قرار نمی گیرد و خواستگاه پیدایش آن، نتیجه ی اعمال الگو می باشد؛ زیرا این کلاس ما_به_ازای خارجی ندارد.

الگوی Prototype نیز بسیار ساده، واسطی تعریف کرده است که تخصص کاری آن، کپی گیری از نمونه های اولیه است. این کلاس متناظری در مساله ندارد و موجودیتی جعلی است.

مقایسه: در هر سه الگو می توان موجودیت های جعلی ای را پیدا کرد که

از تحلیل قلمرو مساله نشات نمی گیرند.

۱۵.۱ سادگی پیاده سازی

الگوی Command را که بخواهیم پیاده سازی کنیم، کار زیاد سختی وجود ندارد و در اکثر زبان های برنامه نویسی به راحتی قابل استفاده است. فقط ذکر این نکته کافی است که جهت اضافه نشدن پیچیدگی به ساختارکد برنامه و توانایی خوانایی و درک روابط و همچنین دانش سریع از به کار برده شدن الگو، بهتر است تا به نام خود(یا حداقل به عنوان پسوند) ،Command کلاس جعلی به کار رود.

الگوی Iterator در اغلب زبان های برنامه نویسی مانند جاوا پیاده سازی شده است. هرچند که اگر چنین نبود هم برای پیاده سازی این الگو، دردسر زیادی به برنامه نویسان وارد نخواهد شد. مثلا در زبان جاوا، روی یکی از انواع لیست می توان iterator آن را گرفت و آبجکت آن را دردسترس داشت و سپس روی آن عملیات مربوط به پیمایش را انجام داد.

الگوی Prototype نیز امروزه در اکثر زبان های برنامه نویسی تسهیلات آن فراهم شده است و با متدهای clone و یا self و از این دست، کپی گرفتن یک آبجکت و برگرداندن آن را محقق می کنند. بنابراین برنامه نویسان با دشواری پیاده سازی کپی کردن روبرو نخواهند شد.

مقایسه: الگوی Command معمولا از نظر پیاده سازی آن به مجموعه کار بیشتری را نسبت به دو الگوی دیگر می طلبد و حتی ممکن است کد را از خوانایی و سادگی بیاندازد که باید در آن دقت فراوانی داشت. دو الگوی دیگر اما با چنین مشکلاتی دست و پنجه نرم نمی کنند و پیاده سازی آن ها راحتتر می باشد؛ هرچند که در اکثر محیط ها به صورت built-in پیاده سازی شده اند.

۱۶.۱ موارد کاربرد

الگوی Command موارد کاربرد زیر را دارد:

- زمانی که می خواهیم اشیاء توسط عملیاتی که می خواهد انجام شود(ماهیت عمل ثابت نیست) پارامتربندی بشوند.
- زمانی که می خواهیم بتوانیم امکان مشخص سازی، صف بندی و اجرای درخواست ها را در زمان های مختلف داشته باشیم.
- زمانی که بخواهیم امکان برگشت یک عمل انجام شده را داشته باشیم. عملیات Execute در Command می تواند state را ذخیره کند تا اثرش را در خود دستور بتواند برگرداند. بدین ترتیب باید Command یک operation ی مثل Unexecute داشته باشد تا اثر دستور را به حالت قبلی برگرداند.
- زمانی که می خواهیم امکان تغییرات logging داشته باشیم به صورتی که بتوانند به هنگام crash کردن سیستم دوباره اعمال شوند. این کار نیاز دارد تا واسط Command عملیات های load و store را تعریف کرده باشد.
- زمانی که می خواهیم یک سیستم را حول عملیات های سطح بالا که در عملیات های اولیه بنا شده اند، ساختاربندی کنیم(اشاره به Command و Macro-Command که بعضی سیستم ها فراهم می کنند).

الگوی Iterator موارد کاربرد زیر را دارد:

- زمانی که می خواهیم به محتوای یک شیء مرکب دسترسی داشته باشم بدون آن که بخواهیم بازنمایی داخلی آن را در معرض دید قرار دهیم.
- زمانی که می خواهیم امکان پیمایش های آبجکت های مرکب را فراهم کنیم.

- زمانی که می خواهیم تا یک واسط یکپارچه برای پیمایش انواع مختلف ساختارهای مرکب تامین کنیم

الگوی Prototype موارد کاربرد زیر را دارد:

- زمانی که کلاس هایی که برای نمونه گرفتن از آن ها موجود هستند، بخواهیم در زمان اجرا مشخص شوند مانند لود پویا.
- زمانی که می خواهیم از ساختار توارثی کلاس factory ها که معمولا با ساختار کلاسی محصولات، موازی است دوری کنیم. مانند آنچه که در الگوی Factory Method دیده می شود. که به صورت موازی
- زمانی که نمونه های کلاس می توانند یکی از تنها تعداد کم ترکیب حالت های متفاوت داشته باشند. یعنی ممکن است یا تعدادی اشیاء مشخص و یا تعدادی حالت محدود برای یک کلاس موجود باشد که استفاده از این الگو توصیه می شود؛ چرا که بسیار راحت است اگر تعدادی نمونه اولیه ایجاد کنیم و آن ها را clone کنیم به جای اینکه از کلاس به صورت دستی نمونه گیری کنیم.

مقایسه: با بررسی کاربردهای هر یک از الگوها، شباهت یا تفاوت خاصی بین این سه پیدا نشد؛ زیرا کاربردهای هر یک مخصوص به خود و دنبال انگیزه های متفاوتی است.

۱۷.۱ الگوهای مرتبط با الگو

الگوی Command از آنجایی که می تواند وضعیتی را ذخیره کند شاید بتواند از Memento بهره بگیرد. اگر بخواهیم از دستورهای ساده ای که دستورهای بزرگتر را می سازند بهره مند شویم، احتمالا الگوی Composite بتواند کمک خوبی باشد. درنهایت آن که الگوی Prototype می تواند کمکی باشد برای زمانی که از Command می خواهیم در زمان اجرا نمونه بگیریم. با این کار دستورات را بدون ساختن آن و فقط با برگرداندن کپی شیء، حفظ می کنیم.

الگوی Iterator می توان به دلیل استفاده ی موازی از ساختار توارثی از الگوی Factory Method استفاده کرد تا یکی از زیرکلاس های مناسب ایجاد و منتسب شود. بلافاصله شاید بتوان گفت که می توان از الگوی Prototype بهره برد.

الگوی Prototype بسیار ساده هست و بیشتر از آن که الگویی به آن کمک کند یا در آن استفاده شده باشد، در الگوهای دیگر نظیر همین الگوهای بالا که نام برده شد، کمک کننده می باشد.

مقایسه: الگوی Prototype می تواند کمک کننده ی خوبی در دو الگوی دیگر باشد و الگوی Command می تواند از الگوهای Memento و Composite استفاده کند در حالی که Iterator از Factory Method می تواند استفاده کند.

OCP ۱۸.۱

الگوی Command به نظر در رعایت اصل OCP موفق عمل کرده است. چرا که Invoker با کاهش وابستگی، حال به کلاس انتزاعی Command مرتبط شده است. لذا هر تغییری در زیرکلاس ها پیش بیاد، Invoker از آن منتزع هست و به علاوه برای او فرقی نمی کند به کدام کلاس مجرد قرار است کار را واگذار کند. پس تا زمان ثابت ماندن واسط Command هیچ تغییری برای Invoker نمی توان قائل شد. در سمت دیگر نیز Receiver که می تواند بالقوه پدری برای همه ی انواع سرویس دهنده باشد می تواند چنین حالتی را با Command تجربه کند. پس این DIP به OCP کمک می کند. در سویی اما کلاینت که پیکربند مجموعه است، هر از چندگاهی وارد عمل شده و مجموعه را بازپیکربندی می کند که در این حالت مجبور است با کلاس های concrete آشنا باشد و رابطه ی سطح پایینی با انواع آبجکت ها دارد؛ به طوری که تغییر در آن ها باید به اطلاع کلاینت برسد.

الگوی Iterator در رابطه ی کلاینت با واسط های سطح بالا اصل OCP برقرار است که در واقع وابستگی شکسته شده و در صورت تغییر نکردن واسط پیمایشگر یا پیمایش شونده، کلاینت از تغییر آسوده است. اما در رابطه ی بین خود این دو به صراحت DIP از بین می رود و نقض آن، OCP را دچار خدشه می کند. پس تغییر در یکی سبب لزوم آگاهی دیگری از تغییر می شود.

الگوی Prototype به خوبی اصل OCP را دارد. زیرا رابطه ی کلاینت با کلاس انتزاعی Prototype بوده و در سطح بالایی است و تغییرات زیر Proto-type سبب انتشار تغییر به کلاینت نمی شود.

مقایسه: الگوی Prototype به خوبی OCP را پوشش داده است در حالی که در Iterator و Command نقض می شود. الگوی Command به دلیل ارتباط سطح تجرید پیکربند با کلاس ها و Iterator به دلیل ارتباط آبجکت های مرکب با پیمایشگرها در سطح پایین، در واقع OCP را می شکنند.

LSP ۱۹.۱

الگوی Command توانسته است LSP را رعایت کند. چون زیرکلاس های Command می توانند جایگزین کلاس پدر که خود Command شود؛ فارغ از این که Invoker تحت تاثیر قرار بگیرد.

الگوی Iterator نیز در LSP موفق عملکرده است و تمامی زیرکلاس های پیمایشگر می توانند جای فوق کلاس پیمایشگر قرار بگیرند.

الگوی Prototype نیز اصل LSP را به درستی رعایت کند. زیرا هر Con- cretePrototype یک is a از خود Prototype می باشد و بدرستی می تواند جایگیزین کلاس پدر شود.

مقایسه: هر سه الگو به LSP پایبند می باشند و آن را رعایت کردند و زیرکلاس ها می توانند جایگزین فوق کلاس شوند.

DIP ۲۰.۱

الگوی Command بین Invoker و Command توانسته است DIP را برقرار کند؛ چرا که ارتباط بین این دو در سطح بالا تعریف شده است. به علاوه آنکه اگر Receiver های مختلفی داشته باشیم می توان گفت که کلاس مجرد Command با ارتباط با واسط Receiver ها از DIP استفاده کرده است. اما به مانند توضیحات قسمت قبل در قسمت پیکربند این اصول نقض می شود و رابطه ی سطح تجرید را با کلاس هاس Command یا Receiver برقرار می کند.

الگوی Iterator کما اینکه پیشتر نیز بیان شد توانسته است در رابطه ی کلاینت با کلاس های انتزاعی پیمایشگرها و آبجکت های مرکب، DIP را برقرار نماید. اما از طرفی ارتباط بین خود آبجکت مرکب با پیمایشگر، نقض کامل DIP است. زیرا وابستگی شدید در سطح پایین را به همراه دارد که سبب انتشار تغییرات می شود.

الگوی Prototype بسیار ساده بین کلاینت و Prototype توانسته است

DIP را برقرار کند؛ زیرا ارتباط کلاینت با واسط می باشد نه با کلاس های concrete که زیرکلاس Prototype هستند.

مقایسه: هر سه الگو از DIP بهره برده اند اما تفاوت در آن جا می باشد که الگوی Command نقض DIP را به هنگام نشان دادن کلاینت که پیکربند مجموعه است از خود بروز می دهد و Iterator در هنگامی که دو کلاس پیمایشگر و پیمایش شونده به صورت غیرانتزاعی به یکدیگر وابسته هستند.

۲۱.۱ ISP

الگوی Command نشان می دهد که ISP را به خوبی آدرس دهی کرده است. دقت داریم که این الگو هر دستور را در یک کلاس، کپسوله کرده است و آن کلاس، داده و رفتار یک دستور را کنار هم جمع آورده است. پس همبستگی خوبی را از نظر کار روی Command فراهم کرده است و هر شیء از نوع Command متد Execute را در خود پیاده کرده و وظیفه ی خود را در قالب آن انجام می دهد.

الگوی Iterator نیز آبجکتی را معرفی که در پیمایش کردن ماهر است و به کلاینت دسترسی مستقیم داده نمی شود تا به درون آبجکت مرکب را دید داشته باشد. ایجاد این آبجکت متخصص که پیشتر گفتیم جعلی است، باعث شده تا جداسازی اتفاق بیافتد و دغدغه ی کلاس های Aggregate در آبجکت های جدیدی به نام Iterator جمع شود و طهور یابد که نشان دهنده ی اصل ISP است.

الگوی Prototype فقط Clone را برای برگرداندن کپی شیء از خود به کلاس های زیرکلاس Prototype مقید می کند که نشان دهنده ی خاص بودن تک‌ـکاری است که واسط Prototype اجبار می کند.

مقایسه: هر ۳ الگو در حد مطلوبی به ISP پرداخته اند و آن را رعایت کرده اند.

CRP ۲۲.۱

الگوی Command این اصل را به خوبی رعایت کرده است. چرا که اصلا بعد از اجرا این الگو محقق می شود و در زمان کامپایل هنوز الگو محقق نیست. الگو با delegation های متعدد از Invoker به Command و از آن به Receiver نشان از رعایت CRP دارد. ساختار توارثی آن نقش خاصی در تحقق الگو ندارد و فقط برای استفاده مجدد و بالابردن اصول دیگر به کار گرفته شده است.

الگوی Iterator نیز به طور قطع از CRP بهره برده است. ارتباطات بین کلاینت با هم پیمایشگر و هم شیء مرکب به همراه دید مانای پیمایشگر روی آبجکت مرکب حکایت از تحقق الگو در زمان اجرا و با واسپاری ها دارد که خود نشان دهنده ی CRP است. از ساختار توارثی گونه ای که هر آبجکت مرکب خود می توانست دارار زیرکلاس هایی با انواع مختلف پیمایشگر باشد پرهیز شده است.

الگوی Prototype نیز CRP را دارد؛ برای این که بسیار ساده کلاینت از delegation استفاده می کند و ساختار توارثی ترجیح داده نشده است.

مقایسه: همه ی الگوهای بالا اصل CRP را رعایت کرده اند و از ساختار توارثی صلب اجتناب شده است.

PLK ۲۳.۱

الگوی Command توانسته است از شکل گیری دید ترایا دوری کند. در این الگو Invoker متد Execute را اجرا می کند و سپس کلاس Command منتسب به Invoker وظیفه دارد تا متد معین Receiver را فراخوانی کند و در تمام این مدت هیچ زمانی پیش نمی آید که کل دید اشیاء برای همدیگر باز باشد. پس PLK ثابت است.

الگوی Iterator برای پیمایشگر ابتدا دید کلی مانا را روی آبجکت مرکب ایجاد می کند تا بتواند پیمایش را روی اعضای داخلی ایجاد کند و به کلاینت به مقدار مورد نیاز اطلاعات دهد و نه کل شیء؛ که اگر این چنین بود کاملا

تعریف الگو نقض می شد. پس PLK در این الگو نیز دیده می شود.
الگوی Prototype اصل PLK را رعایت کرده است. زیرا در جواب Clone تنها یک کپی از اشیاء نمونه اولیه برگردانده می شود و این برای ساختن و نمونه گیری از شیء است و نه برگشت دادن شیء در حالی که اطلاعاتی را می توانستیم مخفی کنیم. پس می توان برای آن PLK را قائل شد.
مقایسه: در همه ی الگوهای بالا اصل PLK دیده می شود و نقضی صورت نمی پذیرد.

۲۴.۱ الگوهای GRASP

۱.۲۴.۱ Information Expert

الگوی Command با تبدیل کردن یک دستور ساده ای که در سیستم می توانست به صورت یک فراخوانی متد و یک واسپاری باشد را به کلاس تبدیل کرده و تمام عملیات مربوط به دستور را در آن قرار داده و با این کار کپسوله سازی داده و رفتار را به صورت خوبی مهیا کرده است. چرا که دقت داریم اطلاعات مربوط به Invoker یا Receiver همچنان در کنار خودشان هست و اطلاعات مربوط به Command در کلاس متخصصی قرار گرفته که می داند پذیرنده ی دستور کدام آبجکت است تا به آن واسپاری کند. در نتیجه نمی توان گفت که رفتار از داده ی خود دور افتاده است. پس تحقق این معیار در الگوی Command مشخص است.
الگوی Iterator تقریبا در برآورده کردن این معیار نقص دارد؛ چرا که توجه داریم کلاس پیمایشگر در هر نوعی، برای پیمایش نیاز به داده ی خود دارد تا بتواند روی آن پیمایش را انجام دهد اما این داده از رفتار پیمایش دور افتاده و در کلاس Aggregate محصور شده است. پس رفتار در شیء متخصص آمده ولی از داده دور مانده است که نتیجتا باعث وابستگی بالای بین کلاس ها نیز شده است.
الگوی Prototype در تحقق این معیار به خوبی عمل کرده است. زیرا تنها

رفتار مشخصی که در آن مشاهده می شود برگرداندن یک کپی از آبجکت مورد انتظار است که این رفتار در هر کلاس از نوع Prototype نزدیک به داده ی موردنیاز برای اجرای خود قرار گرفته است. نتیجه ی آن نیز کاهش وابستگی و افزایش انسجام بوده است که در بخش های قبل آن را بررسی کردیم.

۲.۲۴.۱ Creator

الگوی Command نمی تواند این معیار را پوشش دهد. دقت داریم که برای ایجاد یک شیء از نوع Command باید اجازه دهیم تا پیکربند این کار را انجام دهد که در اولویت ۵ قرار دارد، اما Invoker با داشتن رابطه ای از نوع حداقل association که به اشتباه aggregation معرفی شده در اولویت ۴ می باشد که اولویت بالاتری است و نتوانسته creator باشد. پس نقض Creator به وضوح مشخص است.

الگوی Iterator تنها وجود ساختن انواع مختلف اشیاء پیمایشگر به وسیله ی آبجکت شیء مرکب دیده می شود که باعث می شود دقت کنیم دارای اطلاعاتی برای ساختن آن است و به همین دلیل می تواند سازنده ی آن باشد و هیچ الولویت بالاتری نیز وجود نداشته است؛ پس این معیار رعایت شده است.

الگوی Prototype کلاینت را دارد که اینجا دیگر پیکربند نیست و فقط متد Clone را می تواند فراخوانی کند. پس سازنده نبوده و واحد دیگری از نمونه های اولیه، آبجکت ها را ساخته است. این نمونه های اولیه در Manager Prototype نگهداری می شوند و کلاس دیگری نیز با اولویت بالاتر دیده نمی شود تا وظیفه ی ساختن نمونه به او ملحق شود. پس Creator در این الگو نیز تحقق یافته است.

۳.۲۴.۱ Low Coupling

الگوی Command با معرفی کلاس Command وابستگی مستقیم بین In-voker و Receiver را شکسته است. وابستگی مستقیم آن دو تبدیل به رابطه ی غیرمستقیم و تا حدودی از بین می رود. وابستگی بین Invoker و Command در سطح انتزاع هست و Invoker با واسط یک دستور در ارتباط است که خود این امر وابستگی میان آن ها را به حداقل می رساند. در واقع Invoker نمی داند کدام دستور قرار است تا سرویس را از Receiver بگیرد که نشان دهنده ی coupling پایین است. اما کلاینت با کلاس های غیزانتزاعی Command به صورت مستقیم ارتباط دارد و برای Receiver و Invoker نیز اینگونه آن ها را می شناسد که این وابستگی نسبتا شدیدی را نشان می دهد. در نهایت می توان گفت الگوی Command در قسمتهایی وابستگی را حداقل کرده اما در قسمت های دیگر وابستگی شدیدی را بروز می دهد.

الگوی Iterator توانسته است کلاینت را با شیءهای مرکب با وابستگی کم تصویر کند؛ چرا که کلاینت رابطه ای در انتزاع با واسط Aggregate دارد. همین وابستگی را در رابطه با Iterator نیز تجربه می کند که منتزع از تغییر در زیرکلاس های آن است. از طریق واسط می داند که چگونه باید با آن ها کار کند. اما رابطه ی بین اشیاء مرکب و پیمایشگرها نشان می دهد که وابستگی در سطح کلاس های concrete و بسیار قوی است. این می تواند معیار وابستگی کم را برای این الگو نامطلوب جلوه دهد. در نهایت نیز می توان گفت الگوی Iterator در قسمتهایی وابستگی را حداقل کرده اما در قسمت های دیگر وابستگی شدیدی را بروز می دهد.

الگوی Prototype ارتباط بین کلاس های فرزند واسط Prototype را با کلاینت ازبین ببرد و به DIP پایبند باشد. در عین حال پیکربند باید همه ی کلاس های concrete را بشناسد و از کم و زیاد شدن آنان آگاه باشد. با این کار، ارتباط کلاینت به نمونه های اولیه کاهش یافته و پیکربند دچار وابستگی به مجموعه می شود.

High Cohesion ۴.۲۴.۱

الگوی Command با تبدیل کردن یک دستور ساده به یک کلاس درخواست نشان می دهد که انسجام خوبی را دارد و مجموعه ی کلاس های سیستم را تک‌ـمنظوره می کند که باعث افزایش تمرکز کاری می شود. با دید کمتری که کلاس ها از یکدیگر پیدا کرده اند، هر گونه ارتباطی از طریق متد execute فراهم خواهد بود که این سبب افزایش همبستگی در مجموعه ی الگو شده است.

الگوی Iterator همبستگی بالایی از خود نشان می دهد؛ زیرا متخصص پیمایش را تعریف می کند و سعی می کند تک‌ـکار را در اشیاء کپسوله کند. بدین منظور دید کم، باعث تمرکز بر هدف کلاس می شود، مانند اینکه تمامی عملیات مربوط به پیمایش در آبجکت پیمایشگر آمده اند و اینگونه انسجام بسیار بالایی فراهم شده است و در عین حال آبجکت مرکب محتوای خود را برای کلاینت افشا نمی کند.

الگوی Prototype تا حدودی به این معیار پایبند است. هر کلاسی از نوع Prototype تنها متد Clone را در خود پیاده سازی می کند که در آن یک نمونه کپی از خودش برمی گرداند. یعنی آن که اگر کلاسی زیرکلاس Prototype شود. سپس پیکربند به آن آگاه است و می تواند آن را به یک کلاینت مشتری، منتسب کند تا کلاینت از Clone آن استفاده کند. در کل انسجام در آن خیلی تحت تاثیر قرار نمی گیرد.

Controller ۵.۲۴.۱

الگوی Command درواقع این معیار را پوشش نمی دهد و در آن شاخصه ی Controller دیده نمی شود. چرا که صحبتی از آن به میان نیامده است.
الگوی Iterator درواقع این معیار را پوشش نمی دهد و در آن شاخصه ی Controller دیده نمی شود. چرا که صحبتی از آن به میان نیامده است.
الگوی Prototype درواقع این معیار را پوشش نمی دهد و در آن شاخصه ی Controller دیده نمی شود. چرا که صحبتی از آن به میان نیامده است.

Polymorphism ۶.۲۴.۱

الگوی Command با مطرح کردن فوق کلاس Command برای Invoker این امکان را فراهم می کند تا روی آبجکت دستور منتسب به خودش متد Execute را اجرا کند و این متد که رفتاری برای کلاس های concrete دستور هست دارای عملکرد وابسته به type می باشد. پس از چندریختی استفاده شده است تا با تغییر type کلاس هر آبجکت منتسب به Invoker فقط رفتار در سطح پایین تغییر کند بدون آنکه Invoker اطلاعی داشته باشد. پس این الگو از Polymorphism بهره برده است.
الگوی Iterator به خوبی توانسته است از چندریختی بهره ببرد. برای همه ی کلاس های مرکب و همه ی کلاس های پیمایشگر واسطی را برای هر یک تعریف کرده تا رفتار متغیر وابسته به type مهار شود. بدین ترتیب کلاینت، واسط های سطح بالا رو می شناسد و بدون اطلاع از زیرکلاس ها می تواند عملکرد خود را حفظ کند و فقط براساس نوع هر زیرکلاس، رفتار متفاوت اجرا خواهد شد.
الگوی Prototype نیز تنها با تعریف واسط Prototype توانسته است تا چندریختی را فراهم کند. زیرکلاس های آن همگی رفتار Clone را باید از خود بروز دهند و هر یک وابسته به type خود چنین می کند.

Indirection ۷.۲۴.۱

الگوی Command در واقع مسئولیت میانجیگری را بر عهده ی کلاس Command گذاشته و با واسطه شدن خود، ارتباط مستقیم فراخواننده و سرویس دهنده را به غیرمستقیم می شکند. ارتباط کلاینت پیکربند با مجموعه در سطح پایین و بدون کلاس میانی می باشد که شاید آسیبی برای این معیار به نظر برسد.

الگوی Iterator میانجی ساختن یک پیمایشگر توسط کلاینت را کلاس مرکب می داند. در واقع خود کلاینت فقط می تواند آن را در اختیار بگیرد و عملیات های آن را اجرا کند اما در این سطح نمی تواند بدون ارتباط غیرمستقیم، پیمایشگری داشته باشد. خود کلاس های concrete نیز با کلاس انتزاعی از دید و ارتباط مستقیم کلاینت مخفی شده اند. از طرفی اما در سطح پایین، ارتباط مستقیم و بدون میانجی را بین کلاس های پیمایشگر و پیمایش شونده می بینیم.

الگوی Prototype معیار Indirection را با فرستادن پیغام کلاینت به Pro-totype Manager جهت کپی گرفتن یک آبجکت از خود و برگرداندن آن پشتیبانی می کند.

Pure Fabrication ۸.۲۴.۱

الگوی Command فقط بعد از اعمال می تواند یک دستور ساده غیرمانا را در سیستم به چشم یک آبجکت نگاه کند و آن را مانا کند. برای همین کلاس تصنعی Command را تعریف می کند که منشاءیی از قلمروی مساله ندارد

الگوی Iterator کلاس انتزاعی Iterator به همراه زیرکلاس های آن را ناشی از تحلیل قلمرو نمی داند، کاربرد تعریف چنین کلاسی که ما ـ به ـ ازای خارجی ندارد، قابلیت استفاده مجدد و افزایش همبستگی و همبستگی می باشد.

الگوی Prototype نیز کلاس جعلی دارد که متناظر خارجی ندارد. کلاس انتزاعی Prototype برای کاهش وابستگی و قابلیت استفاده مجدد به مجموعه

اضافه می شود که رفتار Clone را برای زیرکلاس ها اجبار کند.

۹.۲۴.۱ Protected Variations

الگوی Command در واسط آوردن کلاس انتزاعی Command که تغییرات احتمالی انواع مختلف Command را مهار می کند، سعی کرده است تا از تغییرات محتمل و غیرپایداری ها جلوگیری کند. بدین صورت که Invoker از همه ی تغییرات و کم و زیاد شدن های محتمل، منتزع بوده و از Receiver نیز جدا می شود. تغییرات در Receiver مادامی که در خود واسط Command تغییری به بار نیاورد، به Invoker نخواهد رسید و بدین ترتیب وابستگی کم شده است و variation ها محافظت شده اند.

الگوی Iterator دو کار مهم را برای حفاظت از متغیرها و انتشار تغییرات محتمل ناشی از آن ها کرده است. یکی آن که با تعریف واسط برای پیمایشگرها و دیگری تعریف واسط برای کلاس های مرکب است. بنابراین تغییرات ممکن روی هر کدام از این کلاس ها، بدین شکل از دید کلاینت دور مانده و حفاظت می شوند و ارتباط غیرشدیدی را نیز بین آن ها رقم می زند.

الگوی Prototype بسیار ساده با تعریف یک واسط اصلی به نام Prototype می تواند از بخش متغیر که زیرکلاس های آن هستند محافظت کند و وابستگی را به سطح بالا ببرد. بدین ترتیب کلاینت از تغییرات، منتزع خواهد شد.

۲ مقایسه Chain-Mediator- of Responsibility State/Behaviour Over A Collection

۱.۲ دسته

الگوی Chain of Responsibility: در دسته ی الگوهای رفتاری یا Be-haviour در GoF قرار دارد.
الگوی Mediator: در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
الگوی State/Behaviour Over A Collection: از مجموعه الگوهای Coad می باشد.
مقایسه: دو الگوی اول در دسته ی رفتاری از سری الگوهای GoF قرار می گیرند که بدین معنی است که به واسطه ی ارتباطات آبجکت ها حین اجرا محقق می شود. اما الگوی State/Behaviour Over A Collection اصلا از مجموعه الگوهای GoF نمی باشد. در واقع دو الگو هستند که از مجموعه ۷ الگوی معرفی شده Coad می باشند.

۲.۲ حوزه

الگوی Chain of Responsibility: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی Mediator: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی State/Behaviour Over A Collection: می توان با توجه به ارتباطی که معمولا از جنس aggregation در آن یافت می شود، این الگو را نیز در حوزه ی شیء تصور کرد.
مقایسه: هیچ کدام از این ۳ الگو در زمان کامپایل و فقط از طریق inheri-tance محقق نمی شوند. برای تحقق همه ی آن ها باید اجرا صورت بگیرد و به

کمک delegation یا همان واسپاری وظیفه حین اجرا بوسیله ی رابطه ها تحقق آن ها را ثابت کرد.

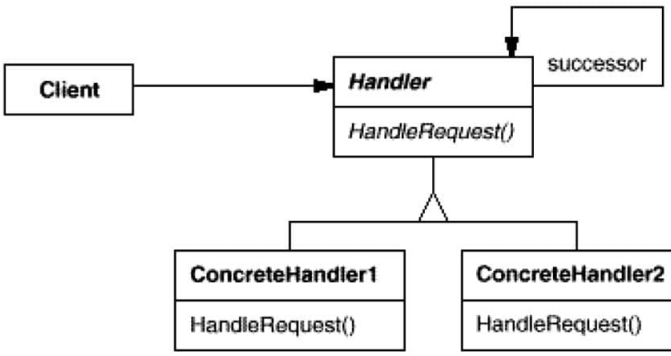
۳.۲ هدف

الگوی Chain of Responsibility: هدف، پرهیز از وابستگی ارسال کننده ی درخواست به گیرنده ی خود توسط دادن شانس به تعداد بیشتر از یک آبجکت جهت بررسی درخواست می باشد. اشیاء دریافت کننده زنجیر می شوند و درخواست در میان زنجیر پاس داده می شوند تا یک آبجکت بتواند آن را مدیریت کند.

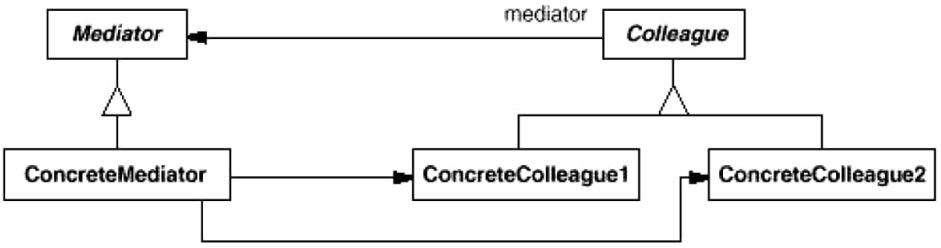
الگوی Mediator: هدف این الگو تعریف یک شیء که چگونگی تعامل مجموعه ای از آبجکت ها را کپسوله بکند می باشد. این شیء یک میانجی بین همه است که نتیجه ی وجود آن، کم شدن وابستگی از طریق دوری از ارجاعات مستقیم آبجکت ها به یکدیگر می باشد. همچنین اجازه ی آن به تغییر دادن تعاملات بین آن ها به صورت کاملا مستقل، از هدف دیگر الگو می باشد.

الگوی State/Behaviour Over A Collection: در این الگو، رابطه ای وجود دارد که عمدتا جزء به کل می باشد. هدف این است که آن ویژگی یا عملیاتی که به جزء تعلق ندارد، روی کل تعریف شود. یعنی اگر بدانیم که برای عملیاتی یا ویژگی ای روی هر جزء، مشکل داریم، باید به فکر بردن عملیات یا ویژگی به قسمت کل باشیم.

مقایسه: تقریبا اهدافی که این سه الگو در جهت برآورده شدن آن هستند از یکدیگر کاملا متفاوت است. الگوی Chain of Responsibility برای کاهش وابستگی یک درخواست کننده ی ارسال به گیرنده ی خود می باشد در حالی که Mediator در جهت میانجی گری بین رفتارهای وابسته بهم چندین شیء می باشد. در نهایت نیز الگوی State/Behaviour Over A Collection جهت تثبیت عملکرد یا ویژگی اجزاء روی قسمت کل می باشد.



شکل ۴: ساختار الگوی Chain of Responsibility

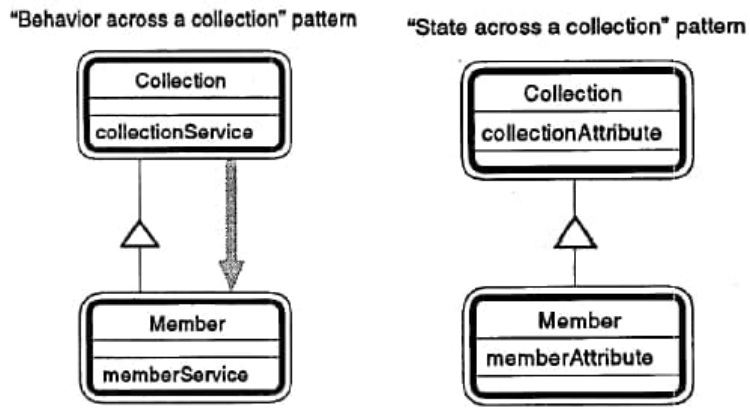


شکل ۵: ساختار الگوی Mediator

۴.۲ ساختار

ساختار الگوی Chain of Responsibility را در شکل ۴ داریم: یک رابطه ی انعکاسی از Handler به خودش اولین چیزی است که در ساختار این الگو یافت می شود. بدین معنا که هر concrete از نوع Handler دیدی را می تواند به آبجکتی از نوع خودش داشته باشد تا در صورت نداشتن جواب به درخواست، بتواند آن را به آبجکت Handler بعدی که می شناسد واسپاری کند. متد HandlerRequest می تواند در فوق کلاس فقط یک hook باشد اما کلاس های سطح پایین Handler آن را به گونه ای که می خواهند پیاده سازی می کنند تا به صورت تخصصی به درخواست پاسخ داده و آن را handle کنند. سلسه زنجیر مسئولیت به خوبی در این ساختار مفهومی، بیان شده است.

الگوی Mediator ساختاری به مانند شکل ۵ دارد. در ساختار این الگو مشاهده می شود که یک رابطه ی سطح بالا از کلاس های همکار به کلاس



شکل ۶: ساختار الگوی State/Behaviour Over A Collection

میانجیگر وجود دارد؛ پس همه ی اشیاءیی که با یکدیگر همکاری خواهند داشت و state هرکدام روی دیگری تاثیری خواهد گذاشت، شیء میانجی را می بینند و به صورت مانا به آن دید دارند. رابطه ی از طرف میانجی به اشیاء همکار به صورت سطح پایین و غیرانتزاعی است که بدان معنی است که هر شیء میانجی متخص، هر شیء همکاری را که خودش بخواهد می تواند به آن دید داشته باشد.

الگوی State/Behaviour Over A Collection نیز ساختاری مشابه با شکل ۶ دارد. در ساختار کلاسی این الگو مشخص است که رابطه ای از نوع aggregation بین جزء و کل وجود دارد که طبیعی است؛ هر collection ی مجموعه ی کلی از اعضاء یا member می باشد. حال این الگو با ساختاری که ارائه می دهد نشان می دهد که مقادیر و عملکرد های صادق برای مجموعه اجزاء، باید در مجموعه ی کل گذاشته شود.، یعنی آن که مثلا Collection–Attribute ویژگی ای صادق برای همه member ها می باشد. پس تصریح می شود که نباید در اعضا به کار رود. اعضا خود می توانند دارای ویژگی های خود باشند که در MemberAttribute می آید.

مقایسه: مقایسه ی این سه الگو در ساختار این گونه است که الگوی Re–Chain of sponsibility از رابطه ی association انعکاسی روی خودش بهره می برد تا بتواند زنجیره مسئولیت ها را بسازد و به کمک successor مسئولیتی را که نمی تواند یک آبجکت handle کند، به دیگری رد کند. کلاینت نیز

فقط واسط Handler ها را می بیند و از تغییرات زیرکلاس ها منتزع می باشد. اما Mediator توانسته است تا فراهم کردن دو واسط، یکی برای میانجیگرها و دیگری برای اشیاء همکار، ارتباطی سطح بالا را از آبجکت های همکار به آبجکت میانجیگر برقرار کند و بدین وسیله، هر شیء همکار، یک میانجی بشناسد، اما از آن طرف میانجی در سطح پایین، با هر آبجکت همکاری که پیکربندی شود در ارتباط خواهد بود که این مفهوم میانجی را می رساند. سپس الگوی Collection State/Behaviour Over A با تعریف رابطه ی کل به جزء و برقراری aggre-gation بین آن ها، عملکردهای موجود برای یک مجموعه و همچنین مقادیر آن را از روی اجزاء برداشته و تاکید می کند تا روی آبجکت Collection تعریف شود.

۵.۲ پیکربندی

الگوی Chain of Responsibility: مشتری مجموعه دیدی روی واسط Handler دارد که می تواند هر زیرکلاسی از این نوع، برای مشتری منتسب شده باشد. به علاوه آنکه خود زنجیره ی Handler ی که قرار است بین آبجکت های از این نوع وجود داشته باشد و تشکیل شود، به صورت پویا می تواند تحت تاثیر قرار بگیرد. لذا ممکن است زنجیره به هر شکلی دستخوش تغییر بشود و کلاس concrete دیگری برای یک کلاس concrete از نوع Handler به او منتسب شود. همه ی این کارها از وظایف پیکربند ثالث هست.
الگوی Mediator: در این الگو پیکربندی توسط یک ثالث انجام می پذیرد(مشتری فقط به میانجی دید دارد)؛ بدین صورت که پیربکند به عنوان آبجکت ثالث، مشخص شود اشیاء همکار، کدامین میانجیگر مجموعه را می شناشند که قرار است تا بین آن ها میانجیگری کند. این پیکربندی می تواند در زمان اجرا به شیء میانجی گر دیگری منتسب شود که در هر صورت باید شیء میانجیگر جدید، در مرحله ی طراحی و کامپایل، نسبت به مجموعه ی اشیاء همکار در سطح پایین که می خواهد بین آن ها وساطت کند دید داشته باشد.
الگوی State/Behaviour Over A Collection: این الگو پیکربندی خاصی

ندارد، فقط آنکه قسمت کل می داند مجموعه اعضا آن کدامند و این را با رابطه ی Aggregation ی که با آن ها برقرار می کند، تحقق می بخشد. تنها کار پیکربند، در صورت وجود، می تواند آن باشد که انتساب یک جزء را به یک کل انجام بدهد.

مقایسه: در مقام مقایسه می توان گفت که آبجکت ثالث به عنوان پیکربند نقش خود را در هر ۳ الگو ایفا می کند و به کمک می آید تا پیکربندی به شکلی انجام شود که دید اشیاء نسبت به یکدیگر بوجود بیاید تا الگو محقق باشد. فقط در الگوی State/Behaviour Over A Collection ممکن است تا کار خاصی برای پیکربندی نیاز باشد و خود Collection رابطه ی مشخص خود را با اعضاء خود، پیکربندی کرده باشد.

۶.۲ انعطاف پذیری

الگوی Chain of Responsibility: این الگو می تواند انعطاف پذیری خوبی به همراه داشته باشد؛ از آنجا که رابطه ی کلاینت با Handler در سطح بالا تعریف شده و زیرکلاس های concrete ی که صرفا درخواست را مدیریت می کنند، از دید کلاینت مخفی هستند، پس امکان بازپیکربندی مجموعه به خوبی فراهم هستند. در ضمن این بازپیکربندی هم در زمان اجرا و زمانی که زنجیره را شکل دادیم، به خوبی خود را نشان می دهد؛ آنجا که پیکربند می تواند –suc cessor یک شیء را به شیء دیگری از همان نوع ارجاع دهد و زنجیره به همین سادگی تغییر کند. در کل انعطاف پذیری در این الگو بالا می باشد.

الگوی Mediator: این الگو با اضافه کردن میانجیگر بین اشیاء باعث می شود تا آبجکت های همکار به جای شناخت همدیگر و شلوغ شدن روابط و پیچیدگی در کار، همان آبجکت میانجیگری را بشناسند که منطق روابط بین آن ها را می داند. با این کار، در صورت کم یا زیاد شدن یک شیء همکار، فقط میانجیگر از آن متاثر می شود و اشیاء همکار تاثیری نمی پذیرند. با این کاهش وابستگی، انعطاف پذیری خوبی مهیا می شود. اما از طرفی باید مراقب بود تا Mediator تمرکزش بر روی گردانندگی امور بین آبجکت ها برحسب state

ورودی هر یک باشد؛ تا بتوان از انعطاف پذیری بالایی سخن گفت و اگر اجازه دهیم که God Class از آن ساخته شود، انعطاف پذیری بدی را تجربه خواهیم کرد که قابل نگهداری نیست. با همه ی این تفاسیر این امکان که بازپیکربندی مهیا باشد، به شیوه ای که انتساب پویای یک میانجیگر به مجموعه آبجکت های همکار داشته باشیم، مهیا بوده و یک میانجیگر تا زمانی که واسط آبجکت همکار تغییر نکند، تنها از کم و زیاد شدن آنها متاثر خواهد شد که باید منطق خود را به روز کند.

الگوی State/Behaviour Over A Collection: خیلی نمی توان برای این الگو صحبت از انعطاف پذیری کرد؛ چرا که قدرت بازپیکربندی یا انعطاف پذیری، بستگی به همان پیکربندی ای دارد که در تیتر قبل صحبت آن شد. این الگو ساختار ساده ای دارد که در نهایت بتوان در زمان اجرا کلاس Member را به یک Collection منتسب کرد، که البته امری بعید به نظر می رسد و معمولا باید انواع Member را تعریف کرده باشیم که پیکربند یکی را به Collection در هر لحظه انتساب کند. یعنی باید واسط ها را تعریف کنیم و روابط را در سطح انتزاعی داشته باشیم که خارج از بحث این الگو می باشد.

مقایسه: به دلیل ارتباط در سطح واسط و انتزاعی که از کلاینت به Han-dler و از خودش روی خودش موجود است، بنا بر استدلال بالا، انعطاف پذیری خیلی خوبی به خاطر بازپیکربندی آسان در الگوی of Responsibility Chain موجود است. الگوی Mediator البته به شرط دوری از خطرات بالقوه، پتانسیل خوبی از نظر بازپیکربندی دارد و در نهایت الگوی Over A Collection State/Behaviour می باشد که در آن روابط ساده هستند و در سطح واسط تعریف نشده اند و خیلی به آن پرداخته نشده اند و نمی توان جز گفته های بالا از انعطاف پذیری آن به صورت خاص صحبت کرد .

۷.۲ کارآیی

الگوی Chain of Responsibility: این الگو کارآیی خود را با ایجاد زنجیره های پیغام که در بدترین حالت باید تا بالاترین سطح رفت و ارجاع داد، کم می بیند. بدین معنا که سربار زمانی به دلیل رابطه های زنجیرواری که وظیفه را به یکدیگر واسپاری می کنند مشخصا بر زمان اجرا و تاخیر کاری اثر می گذارد. این الگو حتی پتانسیل ساخت رابطه های پیچیده و تودرتو نیز دارد که ممکن است به ساختار برنامه و کد آسیب برساند و آن را از خوانایی و سهولت در درک و فهم بیاندازد.

الگوی Mediator: زمانی که این الگو به مجموعه کلاسی اعمال می شود، روابط تودرتوی بین اشیاء همکار، گسسته می شود و جای خود را به رابطه ی یک_به_چند با Mediator می دهند. با این حساب، به شرط پرهیز از فعالیت های خطرناکی مانند ایجاد God Class و یا ایجاد یک نقطه برای شکست کل سیستم، می توان کارآیی بالای آن را در جهت ساختاردهی مناسب به مجموعه و شکستن پیچیدگی بین آبجکت ها و همچنین کاهش وابستگی و ایجاد انسجام دید. با این توضیح، تبادل پیغام های زیاد چند_به_چند شکسته شده وسربار ارتباطی شدید کاهش می یابد.

الگوی State/Behaviour Over A Collection: زمانی که یک عملکرد یا یک ویژگی برای مجموعه ی از اعضا صدق می کند، مانند همه ی اجزای هواپیما وقتی در ارتفاع مشخصی حرکت می کند همه ی اجزای آن در آن ارتفاع حرکت می کنند، آنگاه درنظر گرفتن آن عملکرد و ویژگی برای همه ی اجزا، صرفا اشتباهی است که سبب تمرکزگرایی و عدم انسجام می شود و به پیچیدگی کد و ساختار کمک می کند. پس از اعمال این الگو اما شاهد Collection ی هستیم که باعث زدوده شدن تمرکزگرایی و بالارفتن انسجام و خوانایی ساختار و کد، به همراه درک و فهم بیشتر آن می شود. از طرفی نکته ی منفی آن، اضافه شدن رابطه ای از کل به جزء است که قسمت کل بتواند از آن طریق با اجزای خود در ارتباط بوده و پیغام ها را به آن ها بسپارد. این باعث سربار پردازشی می شود و رابطه ای را به مجموعه اضافه می کند که نکته ای منفی را برای کارآیی

به همراه می آورد.

مقایسه: الگوی Chain of Responsibility نمی تواند کارآیی بالایی را از خود نشان دهد در حالی که این اوضاع در Mediator بهتر است و به دلیل کاهش وابستگی ها به نسبت قبل، می تواند سریعتر نیز عمل بکند. الگوی State/Behaviour Over A Collection نیز می تواند اثرات جانبی خود را روی کارآیی بگذارد، هر چند که در کارآیی سرعت در درک و فهم ساختار کمک می کند.

۸.۲ وابستگی

الگوی Chain of Responsibility: این الگو در واقع تنها در زمان اجرا، پتانسیل آن را دارد که وابستگی شدیدی را بین مجموعه زنجیر خود ایجاد کند، اما در روابط کلاسی خود الگو، میزان رابطه ی زیادی به چشم نمی خورد. می دانیم که کلاینت رابطه ای در سطح انتزاع با Handler دارد که می تواند به کاهش وابستگی کمک کند، اما وجود یک رابطه ی انعکاسی روی واسط Handler فقط به این کمک می کند که زیرکلاس ها از یکدیگر مستقل شوند و دید غیرمستقیمی به یکدیگر(هر شیء فقط به یکی) داشته باشند که می تواند به بحث وابستگی پایین کمک کند. این دید که در سطح بالا برده شده است اما در نهایت می تواند با افزایش طول زنجیره، تبدیل به وابستگی های طولانی شود که البته تنها در زمان اجرا می تواند خطر رابطه ها را نشان دهد اما این الگو نحوه ی ارتباط بین کلاس ها را در انتزاع آورده است و نوع آن را از نوع Association گذاشته است که وابستگی کمی را برای این الگو نشان می دهد.. حالت قبل از اعمال الگو که رابطه ها به هر شکلی می توانست بین آبجکت ها باشد نیز دلیلی بر این مدعاست(پیکربندی نیز وابستگی خاص خود را دارد).

الگوی Mediator: این الگو به شکل واضحی در کاهش وابستگی نقش دارد. به طور واضح مشخص است که با آورده شدن یک کلاس که بتواند متخصص منطق روابط اشیاء همکار باشد، تا چه حد می تواند روابط مستقیم آن اشیاء را ازبین ببرد و روابط را غیرمستقیم کند. این کاهش وابستگی منجر به رابطه ی

همه ی همکاران با تک آبجکت میانجیگر می شود اما از آن طرف وابستگی یک شیء concrete میانجی، با تمام اشیاء همکار در سطح پایین پیکربندی شده است که در واقع می توان گفت صرفا رابطه ی متقابل را نیز رقم می زند، اما با همه ی این احوال، وابستگی پیچیده ی حالت قبل از الگو دیگر ملاحظه نمی شود. وابستگی جزئی پیکربند نیز که بخش متداول و رایجی است.

الگوی State/Behaviour Over A Collection: وابستگی برای این الگو را در حد ساختاری که برای آن تمثیل شده است در نظر می گیریم که صرفا رابطه ای ساده از جنس Aggregation از سمت قسمت کل به سمت اعضای آن هستند. رابطه، ساده اما مستقیم است و جای پیچیدگی را به همراه نمی آورد و باعث نمی شود که وابستگی را خیلی زیاد و غیرمعقول بدانیم. دلیل این صحبت آن است که اگر زیرکلاس هایی برای Member قائل شویم، ارتباط هر یک با کل خود به صورت غیرمستقیم است. به مانند قبل برای پیکربندی بسیار جزئی نیز می توان درصدی از وابستگی قائل شد.

مقایسه: جدای از پیکربندی که هرکدام به طریقی وابستگی را برای آن خواهند داشت، الگوی Chain of Responsibility می تواند در مجموعه کلاسی موثر عمل کند و با بردن روابط در سطح انتزاع، به آن شکل مشخص داده و آن را کمتر کند نسبت به حالتی که شاید هر آبجکتی به دیگری متصل بود. الگوی Mediator نیز سطح خوبی را از وابستگی فراهم کرده و برای الگوی State/Behaviour Over A Collection می توان گفت که رابطه ی ساده ای برای آن درنظرگرفته شده که لطمه ی خاصی به معیار وابستگی آن نمی خورد.

۹.۲ همبستگی

الگوی Chain of Responsibility: این الگو توانسته است تا همبستگی خوبی مهیا کند. بدین صورت که هر زیرکلاسی از نوع Handler مجبور است که متد HandleRequest را در خود پیاده سازی کند و می داند که چه پاسخی را باید بدهد و متخصص handle کردن چیست و اگر نتوانست می داند که شیء بعدی در زنجیره کدام است تا کار را به او واسپاری کند. همه ی این ها باعث افزایش انسجام کلاس های Handler می شود؛ زیرا میزان وظیفه ی مشخصی را به حد خوبی از خود نشان می دهند.

الگوی Mediator: این الگو، متخصصی را بوجود می آورد که منطق روابط بین مجموعه ی همکار را در خود جای می دهد. بدین ترتیب تمرکز کلاس های همکار روی کار غیرمرتبط کمتر شده و انسجام به دلیل تک‌ـکاره بودن افزایش می یابد. همبستگی خود به دلیل کاهش روابط، افزایش می یابد. اما همچنان که گفتیم، خطر God Class برای کلاس Mediator وجود دارد که پتانسیل بالایی را برای عدم چسبنده بودن به همراه دارد.

الگوی State/Behaviour Over A Collection: درنتیجه ی اعمال این الگو، همبستگی خوبی متوجه کلاس های مجموعه خواهد بود؛ چرا که ویژگی و عملکردهای متعلق به قسمت کل، در خود آن جای خواهند گرفت و آن دسته از ویژگی ها و عملکردهایی که متعلق به قسمت کل نیستند نیز درون اجزاء گذاشته می شوند. بدین ترتیب انسجام خوبی را بین کلاس هایی که تمرکز بیشتری روی منظور خود دارند شاهد هستیم.

مقایسه: هر سه الگو دارای انسجام بخشی خوبی به مجموعه کلاسی هستند که در نهایت باعث می شود تا کلاسهای تخصصی تری شکل بگیرد و از کلاس هایی چندمنظوره پرهیز شود.

۱۰.۲ اثرات جانبی

الگوی Chain of Responsibility: یکی از نکات نگران کننده ای که در این الگو می تواند مشکل جدی را رقم بزند، ایجاد و تشکیل به مرور زنجیره ی طولانی پاسخگویی است که می تواند به شدت در کارآیی تاثیرگذار باشد. همچنین دقت به زنجیر حلقوی و سپس رعایت شرط ها و نکات آن از دیگر عوارضی است که ممکن است در اثر استفاده از الگو نمایان شود. وجود سربار مکانی به دلیل پتانسیل وجود رشد تعداد زیاد آبجکت های Handler می تواند تاثیر منفی روی حافظه داشته باشد.

الگوی Mediator: مهمترین معضلاتی که ممکن است در اثر استفاده از این الگو به صورت جانبی نمایان شود، God Class به همراه ایجاد Single Point of Failure می باشد. البته گاهی نیز پتانسیل بروز Bottleneck محتمل هست. همه ی اینها به این دلیل هست که کل منطق روابط مجموعه و اثرگذاری اشیاء همکار در Mediator انجام می شود که باعث می شود پس از مدتی از نگهداری بیافتد. درنتیجه باید refactor شود که از اثرات جانبی استفاده از این الگو می باشد.

الگوی State/Behaviour Over A Collection: از عوارض جانبی که می توان برای این الگو در نظر گرفت آن است که با آوردن رابطه ی Aggregation تاثیری منفی روی کارآیی بوجود خواهد آمد که سبب می شود تا پیغام بین قسمت کل و اجزا بوجود بیاید. الگو ساده بوده و عوارض منفی خاص دیگری که بدانیم در اثر استفاده از الگو پدید خواهد آمد، وجود ندارد.

مقایسه: در الگوی Mediator می توان گفت که یکی از خطرناک ترین الگوها می باشد؛ به دلیل ۳ تا مشکلی که ممکن است نگهداری سیستم را غیرقابل پیش بینی کند. دو الگوی دیگر بیشتر روی زمان و سربار اجرایی تمرکز دارند، با این تفاوت که State/Behaviour Over A Collection توانسته است تا از مشکل جانبی حافظه رنج نبرد اما Chain of Responsibility ممکن است به دلیل امکان وجود اشیاء زیاد در زنجیره، از مشکل حافظه ای نیز رنج ببرد.

۱۱.۲ کپسوله سازی

الگوی Chain of Responsibility: این الگو کپسوله سازی را با وجود واسطی به نام Handler مهیا می کند که سبب می شود تا هر زیرکلاسی از آن که بنا دارد تا درخواستی از مشتری را پاسخ دهد(در ارتباط سطح بالا و بدون دید از اتفاقات زیرکلاس ها)، از طریق پیاده سازی متد HandleRequest این تخصص را برای خود بوجود بیاورد. این کپسوله سازی در هر زیرکلاس Handler این کمک را به کلاینت می کند تا بدون دانستن از مجموعه ی کلاس های زیرین و حتی با تغییر پویای شیء Handler منتسب به او، درخواست خود را بفرستد و این درخواست در زنجیره طی شود تا پاسخ را یکی از آبجکت های گیرنده برگرداند. اینکه هر کدام از اشیاء زنجیره، توانایی تشخیص برای امکان پاسخ یا عدم پاسخ را دارند، نشان دهنده ی کپسوله سازی داده است که داده در کنار رفتار می باشد.

الگوی Mediator: شکل خوب کپسوله سازی در میانجیگر دیده می شود؛ به طوری که کلاسی متخصص برای منطق روابط و گردانندگی اثرات متقابل اشیاء همکار هست و همچنین ارتباط با او از طریق واسط سطح بالا فراهم شده است. این درست است که اشیاء همکار از درگیری به اینکه چگونه بر یکدیگر اثر کنند فارغ می شوند اما میانجیگر قادر است تا با ارتباط مستقیمی که با آن ها دارد، به حالت درونی آن ها نیز دید برقرار کرده و آن را بشناشد؛ این خود پتانسیل نقض کپسوله سازی را دارد. پس کپسوله سازی در این الگو ثابت است، به شرطی که Mediator دید خود را محفوظ از حالت درونی نگه دارد.

الگوی State/Behaviour Over A Collection: کپسوله سازی آنجایی در این الگو یافت می شود که ویژگی ها و عملکردهای موجود برای همه اعضاء، متعلق به قسمت کل می شود. یعنی آن که هر کدام از آبجکت ها، چه از نوع عضو باشد چه از نوع کل، دقیقا می داند که چه عملکرد یا ویژگی هایی دارد و این همان محصورسازی داده در کنار رفتار می باشد. پس کپسوله سازی در این الگو صادق است و قسمت کل می تواند برای عملکردی که شامل همه ی اجزاء می باشد، از عملکردهای تک تک آن ها استفاده کند، که نشان می دهد

در این کار تخصص وجود دارد.

مقایسه: هر کدام از این الگوهای بالا کپسوله سازی را پشتیبانی می کنند.

۱۲.۲ انتشار تغییرات

الگوی Chain of Responsibility: منتشرشدن تغییرات در این الگو فقط زمانی است که واسط Handler دچار تغییرات شود و سپس تغییر به کلاینت نیز منتشر می گردد. در غیر اینصورت هر تغییری در زیرکلاس های آن و کم و زیاد شدن آن ها، مادامی که خود تغییری نکرده است به جایی منتشر نمی شود.

الگوی Mediator: دقت داریم که در این الگو، هر گونه تغییری بخواهد در سمت کلاس های همکار که میانجی آن ها را در سطح پایین می شناسد، صورت بگیرد، میانجی از آن تغییر بی بهره نخواهد بود. اما تا زمانی که خود واسط Mediator دستخوش تغییر نشود، کلاس های همکار نیازی به تغییر نخواهند داشت. دلیل تغییر منتشرشونده ی اولی نیز همین بود که روابط ایجاد شده از سمت میانجیگر به اشیاء همکار، در سطح غیرانتزاعی و بدون واسطه می باشد.

الگوی State/Behaviour Over A Collection: اگر یک ویژگی یا عملکرد که بین همه ی اعضا مشترک است، در خود آن ها تعبیه شده بود و خواستار تغییر در آن بودیم، باید این تغییر در کل همه ی اعضاء اعمال می شد، اما با این الگو چنین تغییر منتشرشونده ای نداریم و با تغییر یک ویژگی یا عملکر مرتبط با همه ی اجزاء، فقط آن را در قسمت کل دچار تغییر می کنیم. این درست است که تغییر در Member می تواند تغییر را در Collection داشته باشد، آن هم به دلیل رابطه ی مستقیم این دو می باشد، اما انتشار تغییر آن ناچیز هست.

مقایسه: مقدار اندکی از تغییرات منتشر شونده در State/Behaviour Over A Collection وجود دارد اما این تغییر بالقوه در Mediator خیلی بیشتر است؛ چرا که روابط میانجی با هر شیء همکار از طریق واسط نیست و به صورت مستقیم است. الگوی Chain of Responsibility هم به طریقی که گذشت می تواند تغییرات منتشرشونده را مهار کند.

۱۳.۲ میزان استفاده از منابع سیستمی

الگوی Chain of Responsibility: از نظر میزان مصرف حافظه باید این نکته را در نظر گرفت که این الگو بالقوه می تواند با ایجاد زنجیره های پاسخگویی طولانی، حافظه ی زیادی را برای نگهداری اشیاء و ارجاعات آن ها مصرف کند. همچنین واسپاری ها می تواند پردازش زیادی را به خود اختصاص دهد که مطلوب نیست و سبب تاخیر در کار و برگردادندن پاسخ به کلاینت می شود.

الگوی Mediator: با به کاربردن الگوی Mediator می توان دید که از نظر تعدد اشیاء در مجموعه خیلی فرق خاصی با حالت قبل از میانجی به وجود نیامده است. لذا از نظر مصرف سیستمی منابع حافظه ای خیلی تاثیری نخواهد گذاشت. از طرفی هم می توان گفت با بهبود روابط و ایجاد تمرکزگرایی در میانجی شاید بتوان گفت که از نظر پردازشی، سرعت بهتری به دلیل منطق مشخص تر و ارجاعات کمتر بوجود خواهد آمد که سبب می شود استفاده از منابع پردازشی کمی بهبود یابد.

الگوی State/Behaviour Over A Collection: تجمیع رفتارها و ویژگی موجود در همه ی اعضاء حال به قسمت کل منتسب می شود که این خود نشان دهنده ی صرفه جویی خوبی در میزان مصرف حافظه است؛ چرا که دیگر مجبور به نگهداری مثلا داده های یکسان برای همه اعضاء نیستیم.

مقایسه: الگوی Mediator خیلی تاثیری در میزان مصرف از منابع سیستمی نمی گذارد. فقط میانجی را اضافه می کند که آن هم نمونه های زیادی در سیستم ندارد. اما الگوی State/Behaviour Over A Collection توانسته است تا افزونگی داده را حذف کند و به میزان مصرف حافظه کمک کند اما در مقابل، الگوی Chain of Responsibility محتمل است تا مقدار حافظه ی زیادی را به خود اختصاص داده تا بتواند آبجکت های موجود در زنجیره را نگه دارد. این کار از نظر مصرف پردازش هم می تواند نامطلوب باشد.

۱۴.۲ استفاده از شیء جعلی

الگوی Chain of Responsibility: این الگو برای پاسخگویی به درخواست های مشتری به صورت زنجیره وار، واسطی را به عنوان Handler تعریف کرده که بدیهی است، منشاء خارجی در مساله نداشته و صرفا در نتیجه ی اعمال الگو پدید آمده است.

الگوی Mediator: بسیار واضح هست که کلاس میانجی اصلا متناظر خارجی ندارد و برگرفته از قلمرو مساله نمی باشد و صددرصد جعلی است. یعنی آنکه بدون آن و قبل از اعمال الگو، اشیاء همکار به صورت مستقیم با یکدیگر ارتباط داشته و وجودی برآمده از تحلیل قلمرو مساله داشته اند. اما میانجی این چنین نبوده و ما ـ به ـ ازای این چنینی ندارد.

الگوی State/Behaviour Over A Collection: در این الگو می دانیم که اجزاء از قبل بوده اند و نشات گرفته از فضای مساله هستند؛ اما اگر بدانیم که کلاسی که قسمت کل را نمایندگی می کند، در اثر انتقال داده ها و رفتارهای همه ی اعضاء به قسمت کل بوجود آمده و قسمت کل در قلمروی مساله جایی نداشته، در آن صورت حتی Collection جعلی نیست. درغیراینصورت نیز که حتما کلاس جعلی در این الگو وجود ندارد. مانند آنکه اگر بخواهیم اعضای داخلی بدن را تحلیل کنیم و در تحلیل خود بدن جایی نداشته باشد و در اثر اعمال الگو، زاییده ی آن باشد، نمی توان گفت کلاس بدن، موجودیتی جعلی است، چون در هر صورت نظیر خارجی دارد. اما اگر بخواهیم هواپیمایی را مدل کنیم و آن را از همان اول بعنوان کلاس Collection نظیر کرده باشیم که قطعا پس جعلی نیست.

مقایسه: در الگوی اول، موجودیت جعلی وجود دارد(در اولی برای ایجاد متخصص پاسخگویی و ارجاعات زنجیری و در دومی برای کارچرخانی بین مجموعه اشیاء همکار) اما در الگوی State/Behaviour Over A Collection نمی توان گفت که کلاسی جعلی در الگو دیده می شود. چون اگر اعضایی وجود داشته باشند، آن اعضا، عضو یک کل بزرگتر هستند. حال آن قسمت کل می خواهد در تحلیل باشد و یا می خواهد با الگو بوجود بیاید.

۱۵.۲ سادگی پیاده سازی

الگوی Chain of Responsibility: پیاده سازی این الگو کار زیاد سختی را به همراه نخواهد داشت. وجود رابطه ی انعکاسی باید دقت شود و اینکه باید به طور مشخص برای هر زیرکلاس Handler الگوریتم پاسخگویی را توصیف کرد.

الگوی Mediator: این الگو در پیاده سازی ساده می نماید. پرهیز از ایجاد God Class اولین چیزی است که باید در پیاده سازی از آن بر حذر بود. در صورت نامگذاری کلاس میانجیگر به نام خود الگو یا وجود آن در پسوند نام کلاس، می تواند کمک کننده به این موضوع باشد. دقت در اینکه کلاس میانجی به کدامین کلاس های همکار دسترسی دارد و با آن ها در ارتباط است، از موارد پیاده سازی است که باید رعایت شود.

الگوی State/Behaviour Over A Collection: الگو بسیار ساده و قابل پیاده سازی است و فقط باید دقت داشت که Collection به اعضا دید داشته باشد و حاوی آن دسته از ویژگی ها و عملکردهای صادق روی همه ی اعضاء می باشد.

مقایسه: در مقام مقایسه می توان گفت که هر ۳ الگو در پیاده سازی، ساده هستند و پیچیدگی خاصی در آن ها دیده نمی شود و با اکثر زبان های برنامه نویسی قابل نوشتن هستند.

۱۶.۲ موارد کاربرد

الگوی Chain of Responsibility:

- زمانی که بیشتر از یک آبجکت بتواند یک درخواست را پاسخ بدهد و در عین حال handler هیچ مقدمی نمی شناسد و به صورت خودکار معین می شود.
- زمانی که می خواهید درخواستی را بدون مشخص کردن گیرنده به صورت واضح، به یکی از چندین آبجکت مطرح کنید.

- زمانی که مجموعه ای از آبجکت های handle کننده ی درخواست، باید به صورت پویا و زمان اجرا مشخص شود.

الگوی Mediator:

- زمانی که مجموعه ای از آبجکت ها به شکل دقیق مشخص شده اما پیچیده ای با یکدیگر ارتباط برقرار می کنند. نتیجه ی این وابستگی های بینابینی، به صورت فاقد ساختار و سخت برای درک و فهم هستند.
- استفاده ی مجدد از آبجکت، دشوار می نماید؛ چرا که به بسیاری از آبجکت های دیگر ارجاعات دارد و با آن ها ارتباط برقرار می کند.
- زمانی که یک رفتار توزیع شده بین چند کلاس، بهتر باشد تا بدون تعداد زیادی subclass سازی، شخصی سازی شود.

الگوی State/Behaviour Over A Collection:

- زمانی که در دامنه ی مساله، رابطه ای از نوع کل به جزء وجود دارد که یک قسمت مربوط به کل و قسمت دیگر مربوط به اعضاء می باشد و یک یا چند ویژگی به قسمت کل یا همان Collection مربوط است.
- زمانی که در دامنه ی مساله، رابطه ای از نوع کل به جزء وجود دارد که یک قسمت مربوط به کل و قسمت دیگر مربوط به اعضاء می باشد و یک یا چند رفتار یا سرویس به قسمت کل یا همان Collection اعمال می شود.

مقایسه: با بررسی کاربردهای هر یک از الگوها، شباهت یا تفاوت خاصی بین این سه پیدا نشد؛ زیرا کاربردهای هر یک مخصوص به خود و دنبال انگیزه های متفاوتی است.

۱۷.۲ الگوهای مرتبط با الگو

الگوی Chain of Responsibility: برای این الگو شباهت یا ارتباط خاصی با الگوی دیگری یافت نشد.

الگوی Mediator: از لحاظ شباهت گفته می شود که این الگو شبیه به الگوی Facade می باشد. چرا که آن الگو نیز به معرفی آبجکت Facade می پردازد که کار کارچرخانی را در داخل مجموعه برعهده می گیرد و پیچیدگی را از دید کلاینت می کاهد. در Mediator همه ی اشیاء همکار باید میانجیگر را بشناسند اما در Facade لزومی به اینکه همه ی آبجکت های درون سیستمی باید Facade را بشناسند وجود ندارد. الگوی Mediator البته می تواند توسط الگوی دیگری مورد شباهت قرار بگیرد که آن را در الگوی بعدی بررسی می کنیم.

الگوی State/Behaviour Over A Collection: نکته ای که برای این الگو می توان گفت آن است که شاید بتوان Behaviour Over A Collection شباهتی به Mediator داشته باشد؛ چرا که رفتار پخش شده در تعدادی آبجکت عضو را در آبجکت دیگری گردآوری می کند.

مقایسه: الگوی Mediator با State/Behaviour Over A Collection از نظر مفهومی شباهاتی با یکدیگر دارند. در همین مفهوم نیز با Facade اتحاد هدف دارند.

۱۸.۲ OCP

الگوی Chain of Responsibility: این الگو توانسته است تا OCP را در حد بالایی برقرار کند. دقت داریم که کلاینت، کلاس پاسخ دهنده ای را به صورت مستقیم نمی شناسد و تنها با واسط آن ها در سطح انتزاع وابسته است و تا زمانی که آن تغییر نکند، هر توسعه ای در زیر آن، تغییرات را بسته نگه خواهد داشت. از طرفی نیز رابطه ی بین خود گیرنده ها به شکلی است که رابطه ی انعکاسی روی واسط تعریف شده و دوباره همان تعریف OCP کاملا صادق

است.

الگوی Mediator: این اصل در این الگو تا حد مطلوبی برقرار است؛ چرا که اشیاء همکار تنها با واسط Mediator در ارتباط هستند. بدان معنی که هم نیاز نیست مستقیما با هم در ارتباط باشند و از تغییرات همدیگر اثر بپذیرند و هم اینکه مادامی که واسط میانجیگر عوض نشود، تغییری متوجه آنها نخواهد بود. اما از طرفی به دلیل عدم رابطه در سطح انتزاع از میانجی به کلاس های همکار، اگر هر تغییری در یکی از کلاس های همکار رخ بدهد، کلاس Mediator باید از آن آگاه شده و خود نیز تغییر کند که این نمی تواند مطابق با اصل OCP باشد. چرا که نمی توان در عین باز بودن برای توسعه، بسته بودن نسبت به تغییر داشت.

الگوی State/Behaviour Over A Collection: در این الگو می توان گفت که جهت تغییر در اعضا، شاید لازم است تا لزوما کلاس Collection نیز از آن آگاه باشد و در صورت توسعه ی آن، حتما باید این کلاس نیز دستخوش تغییر بشود، چرا که صحبتی از ارتباط سطح بالا نیست. پس نمی توان گفت که OCP در این الگو دقیقا وجود دارد. مثلا ممکن است یکی از رفتارهای قسمت اعضاء توسعه یابد که خب این ما را ملزم می سازد تا انتشار این توسعه را در Collection نیز بررسی کنیم اگر عملکردی در Collection از آن رفتارهای اعضاء استفاده می کند، آن عملکرد را نیز تغییر دهیم. از آنجایی هم که پیشتر گفتیم نمی توان تصور کرد که ممکن است کلاس Member در ساختار، فوق کلاسی برای انواع Member باشد.

مقایسه: الگوی Chain of Responsibility به خوبی توانسته است تا از اصل OCP پشتیبانی کند اما الگوی State/Behaviour Over A Collection به ظاهر آن را نقض می کند. الگوی Mediator نیز از طرف همکاران به میانجیگر این الگو را نقض نمی کند اما در رابطه ی برعکس آن، نمی تواند OCP را رعایت کند.

LSP ۱۹.۲

الگوی Chain of Responsibility: در این الگو مشاهده می کنیم که هر ConcreteHandler یک Handler می باشد؛ چرا که درخواست را به شکلی سفارشی پیاده سازی شده، پاسخ می دهد. پس این معیار در این الگو ثابت است.

الگوی Mediator: این اصل در این الگو رعایت شده است؛ بدین صورت که هر زیرکلاسی از کلاس های همکار می توانند جایگزین فوق کلاس خود شوند و دقیقا is-a برقرار است. همچنین هر میانجیگری می تواند جایگزین کلاس بالاتر خود شود و نقضی از این بابت وجود ندارد.

الگوی State/Behaviour Over A Collection: بررسی این معیار برای این الگو بلاموضوع است.

مقایسه: در دو الگوی اول معیار LSP به خوبی برقرار است اما در الگوی سوم، نمی توان از آن سخن گفت، به دلیل آنکه ساختار توارثی ای در آن دیده نمی شود.

DIP ۲۰.۲

الگوی Chain of Responsibility: این الگو DIP را خیلی عالی لحاظ کرده است. هم دررابطه ی کلاینت با گیرنده ها که در سطح انتزاعی است و هم ارتباط انعکاسی از خود گیرنده ها روی خودشان که آن هم در سطح بالا می باشد. پس DIP برقرار است.

الگوی Mediator: کما اینکه پیشتر بحث شد، این الگو زمانی که ارتباط از اشیاء همکار به میانجیگر می باشد، در سطح interface و انتزاعی می باشد که DIP را به همراه دارد. اما وابستگی از میانجی به همکاران در سطع غیرانتزاعی و کلاس های concrete هست که DIP را نقض می کند.

الگوی State/Behaviour Over A Collection: خیلی نمی توان گفت که این اصل در این الگو بیان شده یا خیر. چون اصلا برای ارتباط، سطح interface

و abstract یا سطح concrete بحثی نمی شود. در کل اگر اینجوری بگوییم که رابطه در سطح دو کلاس غیرواسط هست، پس DIP در آن برقرار نیست.
مقایسه: الگوی Chain of Responsibility می تواند DIP را برقرار کند اما State/Behaviour Over A Collection خیر. به علاوه آنکه Mediator در ارتباط از میانجی به همکاران آن را نقض می کند اما در رابطه ی برعکس آن را رعایت می کند.

۲۱.۲ ISP

الگوی Chain of Responsibility: این الگو توانسته است تا ISP را در حد خوبی برقرار نماید. بدین صورت که با گذاشتن واسطی برای گیرنده، امکان تعریف انواع گیرنده فراهم شده است و کلاینت به صورت مستقیم با یک گیرنده در ارتباط نیست و دیدی به آن ندارد. کلاینت فقط واسط Handler را می بیند که در نتیجه ی آن، عدم نقض ISP است.
الگوی Mediator: این معیار در Mediator تا حدی محقق شده است؛ چرا که با آوردن کلاسی برای میانجیگری و طراحی واسط آن، تخصص به مجموعه اضافه شده است و همچنین همه ی اشیاء همکار که خود واسطی را دارند، از وظیفه ی بررسی اثرات متقابل بر یکدیگر بی نیاز شدند و همه برای انجام شدن کارچرخانی به کلاس میانجیگر متصل شده اند. این خود نشان دهنده ی ISP می باشد؛ اما دقت داریم که اگر الگوریتم پخش شده ی قبل از الگو در اشیاء همکار به طور مجتمع و سنگین در یک واسط گذاشته شود و به نوعی از توزیع شدگی استفاده نکنیم، ممکن است ISP نیز زیر سوال برود.
الگوی State/Behaviour Over A Collection: بررسی این معیار برای این الگو بلاموضوع است.
مقایسه: در دو الگوی اول معیار ISP برقرار است ولی باید دقت داشت که State/Behaviour Over A Collection با قرار دادن یک میانجیگر مشخص بدون interface و سنگین، پتانسیلی برای نقض ISP دارد. اما در الگوی سوم، نمی توان از این معیار سخن گفت، به دلیل آنکه interface ی در ساختار در

آن دیده نمی شود.

CRP ۲۲.۲

الگوی Chain of Responsibility: ترجیح واسپاری به توارث در این الگو در حد بالایی رعایت شده است. این الگو با واسپاری پویای وظیفه به گیرنده های در زنجیره، از ایجاد ساختار توارثی جلوگیری می کند.
الگوی Mediator: این الگو نیز با طراحی واسط های همکار و واسطی برای میانجیگران، از انتساب یک میانجی به همکاران و برقراری دید از میانجی به هر آبجکت همکار، می تواند delegation را بر توارث ترجیح بدهد. یعنی کلا الگو با پیاده سازی محقق می شود. این الگو جایگزین خوبی برای توارث است؛ به گونه ای که به جای ایجاد زیرکلاس های توارثی و ساختن کلاس های فراوان، از واسپاری با رابطه ی یک_به_چند استفاده می کنیم.
الگوی State/Behaviour Over A Collection: این الگو با رابطه ی Ag-gregation ی که فراهم می کند، قسمت کل را قادر می سازد تا با واسپاری قسمتی یا همه ی عملکرد به اعضا، عملکرد خود را به کلاینت بیرونی بروز دهد که در نتیجه توانسته است این معیار را محقق کند.
مقایسه: در هر سه الگو این اصل رعایت شده است.

PLK ۲۳.۲

الگوی Chain of Responsibility: این الگو از خود زنجیره ی دید ترایا را بروز نمی دهد. کلاینت درخواست خود را ارسال می کند و این درخواست به اولین گیرنده فرستاده می شود و سپس در زنجیره ی گیرنده ها می چرخد تا آبجکت Handler ی بتواند پاسخ مطلوب بدهد. سپس این پاسخ فقط برگشت داده می شود. در نتیجه در همه جا فقط این تعامل بین آبجکت ها می باشد که وجود دارد و شیء ای در پاسخ یک درخواست ارسال نمی شود.

الگوی Mediator: هیچ جایی از این الگو یافت نمی شود که آبجکت در ازای یک درخواست فرستاده شود و در نتیجه، دید ترایا در الگو وجود ندارد. لازم است تا ذکر کنیم که اصلا نباید شیء در پاسخ به یک درخواست شیء همکار فرستاده شود؛ چرا که اصلا در این صورت امضای الگو در GoF حفظ نشده است.

الگوی State/Behaviour Over A Collection: دید ترایا در این الگو پیدا نمی شود؛ چرا که به سادگی می توان مطمین شد که فقط تعامل بین اشیاء موجود است و اشیاء منتقل نمی شوند.

مقایسه: در هر سه الگو این اصل رعایت شده است.

۲۴.۲ الگوهای GRASP

۱.۲۴.۲ Information Expert

الگوی Chain of Responsibility: کل کاری که در این الگو انجام شده است قرار دادن واسطی برای کلاینت بوده است تا درخواست خود را در زنجیره ای از گیرنده ها رها کند و هر کدام که متخصص پاسخگویی به یک درخواست خاص هستند، بتوانند آن را پاسخ دهند. دقت داریم که قرار دادن خبره ی اطلاعات مشهود است و از صدق کردن Information Expert در Chain of Responsibility مطمئین هستیم.

الگوی Mediator: این الگو با ایجاد شیء متخصص برای اعمال اثرات متقابل اشیاء همکار، با بیرون کشیدن منطق پراکنده از دل هر کدام و جمع آوری متمرکز آن، آبجکتی را متخصص برای داده و رفتار بوجود آوره است. در نتیجه کلاس همکار از داشتن وظیفه ی این چنینی فارغ می شود و ارتباطی غیر مسقیم با دیگر همکاران دارد. پس Information Expert در این الگو صادق است.

الگوی State/Behaviour Over A Collection: وجود رفتاری در کل که بر روی اعضاء قابل اعمال هست، نشان از کلاس متخصص دارد که داده و رفتار

را کنار هم گرد آورده و با این کپسوله سازی، می تواند عملکردهای ریزدانه ای که به اعضا سپرده شده است را بگیرد تا عملکرد کلی خود را نشان دهد. یا حتی می تواند عملکرد صادق برای همه ی اعضاء را به صورت یکجا نشان دهد. مثلا همه ی اعضای هواپیمای در حال پرواز و جابجایی با سرعتی معین، در حال پرواز و جابجایی با آن سرعت هستند. یا مثلا توان مصرفی کل هواپیما برابر مجموع توان مصرفی تک تک تجهیزات برقی آن است. لذا در سطح خوبی این معیار برای State/Behaviour Over A Collection ثابت است.

Creator ۲.۲۴.۲

الگوی Chain of Responsibility: ساخت اشیاء و منتسب سازی آن ها به صورت پویا در دستان پیکربند است که در ۵مین اولویت قرار دارد. این درحالی است که رابطه های Association کلاینت به گیرنده و از گیرنده به گیرنده ی بعدی، در اولویت بالاتری هستند. پس نمی توان پیکربند را که صرفا اطلاعات لازمه را برای نمونه گیری از اشیاء دارد مناسب دانست و این معیار در این الگو رعایت نشده است.

الگوی Mediator: به نظر می رسد که Creator در این جا نقض می شود. اگر دقت داشته باشیم اشیاء همکار توسط پیکربند در زمان برنامه ایجاد می شوند، این درحالی است که رابطه ای از نوع Association از میانجیگر به همکاران وجود دارد که در نتیجه اولویت بالاتری(۴) را به نسبت پیکربند که صرفا اطلاعات لازم را برای ساخت دارد(۵)، دارا می باشد. پس شاهد نقض آن هستیم.

الگوی State/Behaviour Over A Collection: رابطه ی تقریبا قوی از نوع Aggregation از قسمت کل به قسمت اعضا می باشد. پس عیان است که باید برای رعایت این معیار، آبجکت Collection بتواند تا Member ها را بسازد. اگر چنین باشد. اگر همانطور که در قسمت ۵ گفتیم، پیکربند دارد اعضاء را برای کل منتسب می کند پس شاهد نقض Creator هستیم.

۳.۲۴.۲ Low Coupling

الگوی Chain of Responsibility: کم بودن وابستگی در این الگو بدین ترتیب تعریف می شود که کلاینت تنها با واسط Handler مواجه هست و اصلا نیاز ندارد تا همه ی انواع آن را بشناسد. ارتباط خود گیرنده ها نیز در سطح بالا تعریف شده و دید مستقیم سطح پایینی بر روی یکدیگر ندارند. هر شیء گیرنده توسط پیکربند، دیگر گیرنده را می شناسد و به او وابسته است و درنهایت ارتباط را رقم می زند. این ارتباط ساده است و فقط هم یک‌ـبه‌ـیک هست. لذا می توان وابستگی کم را برای آن قائل شد.

الگوی Mediator: وابستگی پس از اعمال الگو و آوردن میانجی به شدت کاسته می شود و از پیچیدگی ارتباطات دوری می شود. کلاس های همکار دیگر دید مستقیم خود را به یکدیگر ندارند و به علاوه آن که از طریق واسط به میانجی متصل شده و نوعی از آن را می شناسند. همه ی اینها در کاهش وابستگی موثر بوده است. از طرفی اما ارتباط غیرانتزاعی و در سطح کلاسهای concrete از سمت میانجی به کلاسهای همکار، روی کاهش وابستگی، تاثیر منفی خود را می گذارد.

الگوی State/Behaviour Over A Collection: وابستگی در این الگو دیده می شود و حتی از حالت قبل آن که کلاسی تحت عنوان Collection شاید در مجموعه یافت نمی شد، وابستگی ایجاد کرده باشیم اما خب این وابستگی سطحی و ساده است. تنها بین دو کلاس و درنهایت تعدادی آبجکت خواهد بود که رابطه ای از کل به همه ی اجزا را رقم خواهد زد. نمی توان وابستگی را برای آن پایین دانست اما می توان آن را ساده معرفی کرد.

۴.۲۴.۲ High Cohesion

الگوی Chain of Responsibility: همبستگی خوبی در این الگو یافت می شود، بدین صورت که زنجیره ای در برنامه بوجود می آید که هر یک متخصص پاسخگویی به درخواستی از کلاینت می باشد. لذا درخواست در بین زنجیره ی متخصص، گردانده می شود تا اولین آبجکت گیرنده ای که توانایی دارد، درخواست را به کاربر صادر کند. کم بودن بالای وابستگی ها این اثر را داشته است تا انسجام آبجکت ها برای تمرکز روی تخصص خود واقع شود.

الگوی Mediator: متمرکز کردن اثرات متقابل تغییر اشیاء همکار از داخل آن ها که پراکنده بوده اند، آبجکت متخصصی را بوجود می آورد که اجزای آن همبسته هستند و کلاسهای دیگر را در این امر بهبود می دهند. بدین منظور که دیگر یک شیء همکار نگرانی از اثرات متقابل state ش بر بقیه را ندارد و بر روی کار خود تمرکز دارد. همه ی اینها مهر تاییدی برای High Cohesion می باشد.

الگوی State/Behaviour Over A Collection: این الگو انسجام خوبی را به همراه می آورد. زیرا با بردن آن دسته از عملکردها و ویژگی های مربوط به همه اعضاء به قسمت کل، می تواند همبستگی را برای کلاس ها به همراه داشته باشد.

۵.۲۴.۲ Controller

الگوی Chain of Responsibility: این الگو درواقع این معیار را پوشش نمی دهد و در آن شاخصه ی Controller دیده نمی شود. چرا که صحبتی از آن به میان نیامده است.

الگوی Mediator: خود میانجیگر یا Mediator در واقع Controller می باشد که زنجیره ی لازم گرداندن وظیفه را در قبال درخواست وارد شده شروع می کند.

الگوی State/Behaviour Over A Collection: البته صحبت دقیقی از Controller به میان نیامده است اما شاید بتوان گفت که در بعضی مواقع،

هنگامی که سناریویی برای کل اعضاء قابل اعمال است، درواقع Collection بتواند نقش Controller را داشته باشد و عملکردهایی را با واسپاری از اجزایش مدیریت کند.

Polymorphism ۶.۲۴.۲

الگوی Chain of Responsibility: این الگو توانسته است تا این معیار را از طریق قرار دادن واسط Handler برای کلاینت و خود گیرنده ها فراهم کند. بدین منظور کلاینت می تواند با هر نوع Handler منتسب شود و به آن درخواست دهد و همچنین خود گیرنده ها به واسطه ی چندریختی می توانند هر کدام از دیگری را به عنوان بعدی در زنجیره بشناسد.
الگوی Mediator: در این الگو چندریختی را می توان در توسعه دادن انواع میانجیگر یافت کرد. بدین سبب پس هر شیء همکاری می تواند هر نوع از Mediator را بشناسد و بپذیرد.
الگوی State/Behaviour Over A Collection: در این الگو نمی توان از Polymorphism خیلی صحبتی داشت و به نظر می رسد که در ساختار پایه ی آن، چندریختی وجود ندارد.

Indirection ۷.۲۴.۲

الگوی Chain of Responsibility: تشکیل شدن زنجیره ای برای پاسخگویی به درخواست کلاینت از ارتباط کلاینت با تمام اشیاء Handler جلوگیری می کند که سطح خوبی را از Indirection فراهم می کند. بدین ترتیب، دید کلاینت به کل مجموعه یا حتی دید هر شیء گیرنده ی مجموعه به شیء دیگر از این نوع منتفی می شود و ارتباطات برای رسیدن به پاسخ، تا حدی غیر مستقیم است.
الگوی Mediator: ایجاد دید غیر مستقیم و وجود ذاتی خود کلاس میانجیگر، دید آبجکت های همکار را به یکدیگر غیرمستقیم کرده به گونه ای که همه فقط و فقط با میانجیگر مرتبط هستند. این ارتباط یک‌ـبه‌ـچند سبب ایجاد

Indirection می شود.

الگوی State/Behaviour Over A Collection: ارتباط در خود اعضای مجموعه به شکل مستقیم در ساختار پایه ترسیم شده و قسمت کل با اعضای خود به شکل رابطه ی مستقیم Aggregation در ارتباط هست که سبب می شود نتوان از Indirection بهره برد.

Pure Fabrication ۸.۲۴.۲

الگوی Chain of Responsibility: وجود Handler به عنوان واسطه ای برای پیگری درخواست کاملا جعلی بوده و برآمده از مساله نیست. صرفا اضافه کردن تخصصی به مجموعه برای گیرندگی درخواست و بررسی آن جهت پاسخ یا واسپاری، متناظر خارجی ندارد. پس Pure Fabrication در این الگو یافت می شود.

الگوی Mediator: موجودیت جعلی Mediator به هیچ وجه برگرفته از نظیر خارجی نیست که بیانگر جعلی بودن آن است؛ در نتیجه می توان گفت که تخصص در وساطت در ارتباط همکاران را فقط دارد، پس این معیار برای این الگو صادق است.

الگوی State/Behaviour Over A Collection: این که قسمت کل وجود یا ماهیتی جعلی دارد را در قسمت های قبل رد کردیم. پس موافق هستیم که Collection حتما ما_به_ازای مفهوم یا وجود خارجی در قلمروی مساله دارد. لذا می توان گفت که این معیار در State/Behaviour Over A Collection محقق نشده است.

Protected Variations ۹.۲۴.۲

الگوی Chain of Responsibility: با قراردادن واسط Handler می توان خوشبین بود که تغییرات محافظت شده اند. بدین ترتیب که کلاینت تا زمانی که این واسط دستخوش تغییر نشود، از هر تغییر و توسعه ای در زیرکلاس های

آن، منتزع است. همچنین است برای کلاس های concrete از نوع Handler که تا زمانی که هر کدام برای خود تغییر کنند و واسط، دست نخورده باشد، این تغییر به هیچ جایی از مجموعه منتشر نمی شود.

الگوی Mediator: ارتباط انتزاعی از سمت واسط همکار به واسط میانجی سبب آن شده تا وابستگی به شدت کم شده و از تغییرات منتشرشونده پرهیز شود. پس می توان گفت مادامی که واسط میانجی تغییری نداشته است، هر تغییر و توسعه ای در زیرکلاس های آن، به میانجیگر تغییری را منتشر نخواهد کرد. نکته ی منفی اما آنجایی است که ارتباط از سمت میانجیگر به کلاس های همکار، به صورت سطح پایین و از طریق کلاس های concrete هست که مسبب آن می شود تا تغییر یا توسعه ای در سطح کلاس های همکار، حتی با عدم تغییر واسط همکار، انتشار تغییر را نیز به میانجی داشته باشیم.

الگوی State/Behaviour Over A Collection: ارتباطی با واسطه در این الگو به چشم نمی خورد که الزاما سبب کاهش وابستگی شده باشد و تغییرات را از انتشار نگه دارد. بدیهی است که تغییر یا توسعه ای برای Member تغییر متناظر را برای Collection محتمل می سازد.

۳ مقایسه Builder-Facade-State

۱.۳ دسته

الگوی Builder: در دسته ی الگوهای آفرینشی یا Creational در GoF قرار دارد.
الگوی Facade: در دسته ی الگوهای ساختاری یا Structural در GoF قرار دارد.
الگوی State: در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
مقایسه: این سه الگو کاملا از این نظر با هم متفاوت هستند. الگوی Builder با نمونه گیری از اشیاء و پیکربندی کلاس ها دست و پنجه نرم می کند در حالی که الگوی Facade روی کاهش وابستگی بین واسط کلاس ها و پیاده سازی آن ها تمرکز دارد و در نهایت State با تعاملات پویای بین اشیاء سروکار بیشتری دارد.

۲.۳ حوزه

الگوی Builder: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی Facade: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی State: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
مقایسه: هیچ کدام از این ۳ الگو در زمان کامپایل و از طریق inheritance محقق نمی شوند. برای تحقق همه ی آن ها باید اجرا صورت بگیرد و به کمک delegation تحقق آن ها را ثابت کرد.

۳.۳ هدف

الگوی Builder: این الگو به دنبال هدفی است که بخواهیم ساختن یک شیء پیچیده را از بازنمایی آن جدا کنیم؛ بدین صورت که یک فرآیند ساخت یکسان بتواند بازنمایی های مختلفی را تولید کند.

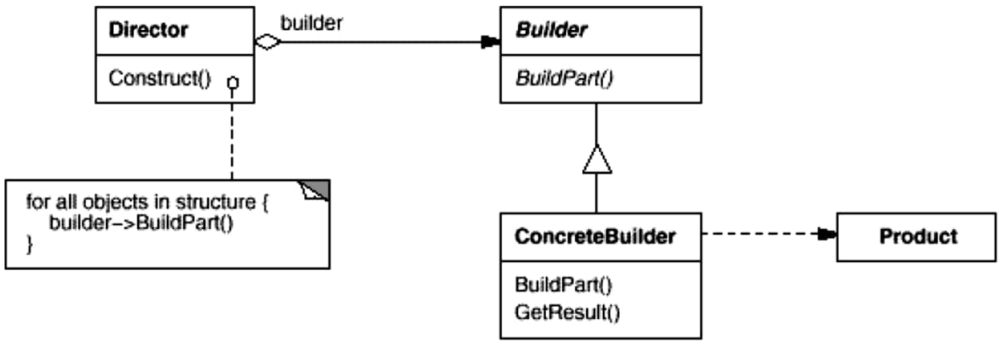
الگوی Facade: هدف این الگو آن است که یک واسط مشخص یکپارچه را برای مجموعه ای از واسط های موجود در subsystem فراهم بیاورد. Facade در واقع هدف دارد تا یک واسط سطح بالاتر از بقیه ی واسط های موجود تعریف می کند که کار استفاده از زیرسیستم را ساده تر کند.

الگوی State: این الگو بسیار ساده می خواهد تا یک آبجکت در سیستم بتواند رفتارش را تغییر بدهد زمانی که state داخلی اش یا به قولی مقدارش تغییر کرد. درواقع الگوی State قصد دارد تا کلاسی از آبجکت را تغییر دهد که رفتار متفاوت در آنجا قرار گرفته است.

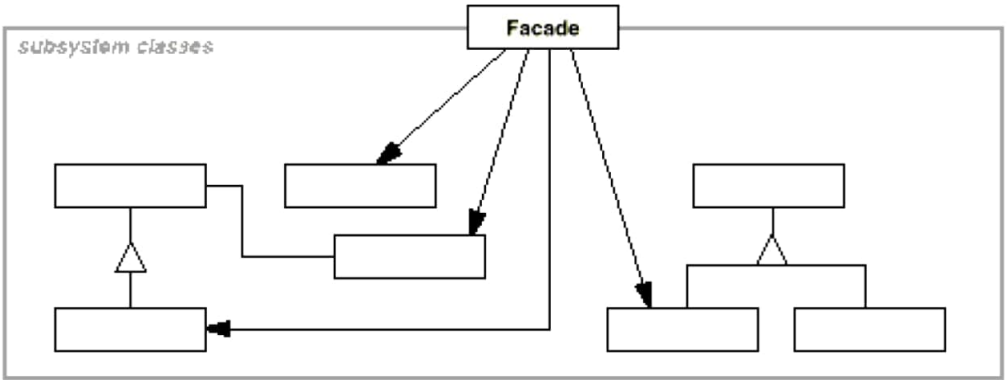
مقایسه: هر کدام از الگو تقریبا اهداف متفاوتی را دنبال می کنند. الگوی Builder برای مهیا کردن یک کارخانه ی ساخت شیء های متفاوت با فرآیندی مشابه و الگوی Facade برای قرار دادن واسط سطح بالا برای کارچرخانی در زیرسیستم هدفمند شده اند. اما الگوی State هدفش در راستای تغییر رفتار با تغییر حالت درونی شیء است.

۴.۳ ساختار

ساختار الگوی Builder را در شکل ۷ داریم: در ساختار این الگو مشخص است که کلاس Director یا کارگردان با سپردن ایجاد هر قسمت از کل کار به انواع مختلف کلاس های سازنده یا Builder ها سعی در تحویل کل کار دارد. متد BuildPart به منظور همین واسپاری در هر زیرکلاس Builder اجبار شده است. براساس می توان سفارشی را از کارگردان گرفت و طبق تخصص، آن را انجام داد. اما طبق همین ساختار پایه اما، متد GetResult اختیاری است که در واقع یک کلاس سازنده می تواند آن را داشته باشد یا خیر. دلیل آن هم این



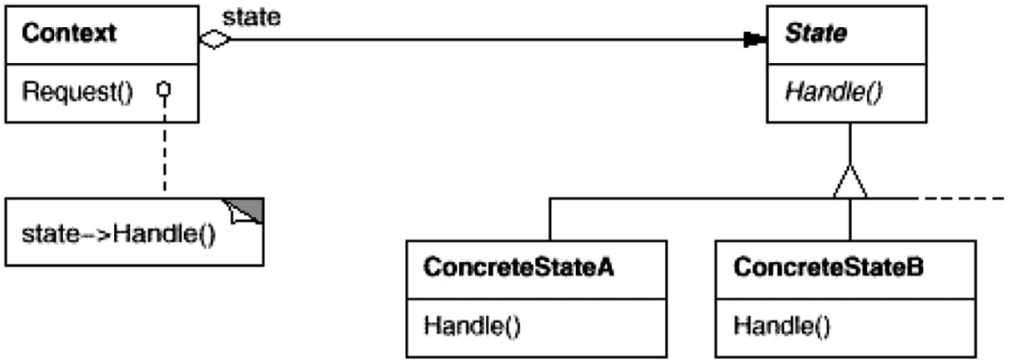
شکل ۷: ساختار الگوی Builder



شکل ۸: ساختار الگوی Facade

است که مهم نیست آیا کارگردان می خواهد اطلاعی از چگونگی قسمت ساخته شده داشته باشد یا خیر. او صرفا کار را محول می کند و تحویل را می بیند و نه چگونگی شکل کار تحویل داده شده.

الگوی Facade ساختاری به مانند شکل ۸ دارد. از تحلیل ساختار این الگو چنین برمیاید که واسطی تحت عنوان Facade به معنای منظره، روی زیرسیستم تعبیه شده است که وظیفه ی مستقیم کارچرخانی در زیر سیستم را برعهده دارد. خود او وظیفه ی پردازشی ندارد و فقط وظایف را به کلاس های زیرین می سپارد. هر تعداد از این واسط سطح بالا بخواهیم می توانیم روی زیرسیستم تعریف کنیم و محدودیتی از این بابت وجود ندارد؛ چرا که با این کار، از طریق مختلف می توان دید را به زیرسیستم مهیا کرد و به معیارهای



شکل ۹: ساختار الگوی State

GRASP کمک نمود.

الگوی State نیز ساختاری مشابه با شکل ۹ دارد. در ساختار این الگو واسط State تعریف شده است که پدر همه ی حالات موجود در سیستم تلقی می شود. هر حالتی که در سیستم موجود است توسط این فوق کلاس Capture می شود و اجبارا Handle را به نحوی که خود می داند پیاده سازی می کند. در واقع رفتار را از خود، در آن جا بروز می دهد. نکته آن است که هر گاه از Context درخواستی شد، آن شیء این درخواست را برحسب حالت درونی خود یا همان state خود بررسی می کند که برای تحقق این منظور، درخواست را به state ی که می شناسد و دارد واسپاری می کند.

مقایسه: هر سه تای این الگوها به هنگام اجرا، رفتار خود را بروز می دهند و از ساختار پیدا می باشد که در زمان کامپایل، ثابت نیستند. الگوهای Builder و State شبیه تر بهم هستند و الگوی Facade از ساختار آن ها دور است. در واقع Facade واسطی را برای زیرسیستم تعریف می کند تا بدین وسیله بتواند کارچرخانی کند در حالی که دو الگوی دیگر واسطی را تعریف می کنند تا کار تخصصی به آن ها سپرده شود. الگوی State می خواهد کل کار را در رفتار خود بروز دهد اما کارگران الگوی Builder قسمت های مختلف را با رجوع به Builder هایی که می شناسد می سازد.

۵.۳ پیکربندی

الگوی Builder: اگر نگاهی به نمودار توالی این الگو بیاندازیم متوجه آن هستیم که کلاینت در واقع پیکربند مجموعه است. کلاینت با دیدی که روی مجموعه دارد، کارگردان و سازندگان را می سازد و به کارگران می گوید که سازنده ی تو چه کسی است. سپس در ادامه کارگردان درخواست های ساخت را به سازنده فرستاده و در یک دنباله ی متوالی، وظیفه ی ساخت را واسپاری می کند تا در نهایت خود کلاینت آن را از سازنده ای که به کارگردان پاس داده است، دریافت کند.

الگوی Facade: دراین الگو خود Facade از قبل می داند که با کدام کلاس های زیرسیستم تعامل دارد و از طرفی هیچ خبری از نحوه ی ارتباط آنان ندارد.پس پیکربندی معینی در الگو یافت نمی شود. در این جا پس اگر کلاینتی بخواهد از زیر سیستم استفاده کند، ارتباط خود را با واسط سطح بالای Facade برقرار کرده و سپس درخواست خود را مطرح می کند تا Facade با مجموعه ی اشیاءیی که از قبل می شناسد تعامل کند و کار را به آنها بسپارد.

الگوی State: برای این الگو حتما نیازی به پیکربند ثالث وجود دارد تا حالت را برای Context مشخص کرده و آن را چه در اول ساخت Context چه به هنگام اجرا تعویض کند. یعنی تعویض حالت درونی Context از وظایف پیکربند است که state را برای او تغییر داده و برابر با یکی از انواع State موجود می کند.

مقایسه: الگوی Facade پیکربند مشخصی را ندارد اما دو الگوی دیگر از پیکربند برخوردار هستند. الگوی Builder را کلاینت، پیکربندی می کند و الگوی State نیز به پیکربند ثالثی برای مشخص شدن حالت درونی Context نیاز دارد.

۶.۳ انعطاف پذیری

الگوی Builder: امکان بازپیکربندی در این الگو وجود دارد؛ بدین صورت که کلاینت می تواند در زمان اجرا، سازنده یا شیء Builder دیگری را به کارگردان منتسب کند. پس انعطاف پذیری در این الگو محقق است.

الگوی Facade: این الگو البته خیلی پیکربندی خاصی را برای شروع کار نیاز نداشت و در نتیجه ی آن، بازپیکربندی برای آن بی معنا است؛ پس بررسی انعطاف پذیری برای این الگو بلاموضوع است.

الگوی State: این الگو دارای انعطاف پذیری می باشد. بدین صورت که در هر زمان از زمان اجرا، پیکربندی ثالث ممکن است برحسب حالت درونی Context که عوض شد، شیءای از انواع مخلف State ساخته و آن را برابر با state موجود در Context بکند. تا بدین ترتیب، حالت جدید و تغییریافته ی خود را بشناسد و از این به بعد واسپاری ها به این state انجام شود.

مقایسه: دو الگویی که در مورد قبلی دارای پیکربند بودند، در زمان اجرا نیز می توانند پیکربندی مجدد را از خود بروز دهند. انعطاف پذیری برای Facade موضوعیتی ندارد؛ زیرا پیکربند مشخصی برای آن وجود ندارد.

۷.۳ کارآیی

الگوی Builder: کارآیی در این الگو کاهش پیدا می کند. زیرا دقت داریم که در صورت نبود کارگردان، خود کلاینت به سازنده ی دلخواه مراجعه می کرد و قسمت های محصول را به مرور می ساخت. پس با وجود کارگردان، ارتباط مستقیم شده است و کارگردان باید واسطه ی بین کلاینت و سازنده باشد تا درخواست مشتری را به سازنده داده تا محصول آماده شود. درنتیجه شاهد پیغام های ردوبدل شده از کارگردان به سازنده هستیم که موجب کاهش کارآیی می شود؛ در حالی که مشتری می توانست با یک پیغام، محصول را برای ساخته شدن تحویل دهد. البته لازم به ذکر است که کاهش وابستگی می تواند در جای خود مفید باشد.

الگوی Facade: الگوی Facade آمده است تا ارتباط مستقیمی که می توانست پیش از این از بیرون زیرسیستم روی هر کلاسی داخل آن وجود داشته باشد را ازبین ببرد. بدین حساب، کاهش وابستگی های زیاد محتمل کلاینت ها به داخل مجموعه از کارآیی های اولیه این سیستم هست. این کاهش وابستگی می تواند در جای خود مناسب باشد اما وجود واسط و مداخله با هدف غیرمستقیم کردن روابط می تواند تاخیری برای پاسخگویی داشته باشد که تاثیر منفی خود را روی کارآیی می گذارد. بدین ترتیب هر کلاینت بیرونی برای دریافت سرویس به جای مراجعه ی مستقیم به کلاس های زیرسیستم، ابتدا به Facade موجود روی زیرسیستم مراجعه کرده که خود سربار زمانی بوجود می آورد.

الگوی State: الگوی State شاید اندکی تاثیر منفی روی کارآیی داشته باشد؛ چرا که متغیر حالت را از Context بیرون کشیده و آن را مجبور می کند برای رفتار خود که زین پس مبتنی بر حالت است، کار را به شیء حالت delegate کند که این خود سرباری را بوجود می آورد تا زمینه ساز کارآیی پایین باشد.

مقایسه: در هر سه الگو کارآیی افت خواهد داشت که در ازای به دست آوردن محول کردن وظیفه، این سربار پردازشی و تاخیر در گرفتن جواب بوجود آمده است.

۸.۳ وابستگی

الگوی Builder: وابستگی در این الگو بین کلاس Director و واسط Builder برقرار است. این ارتباط در سطح بالا و انتزاعی است که واسط Builder دلیل همین کمبودن وابستگی دو طرف می باشد. اما کلاینت به عنوان پیکربند مجموعه باید دید خود را تک تک کلاس ها حفظ کند که البته کار آن شاید هر_از_چندگاه باشد اما وابستگی شدیدی را رقم می زند.

الگوی Facade: این الگو از خود وابستگی کمی را به نمایش می گذارد. بدین صورت که همه ی ارتباطات مستقیم قبلی از هر کلاینتی که با اعضای زیرسیستم کار داشت شکسته شده و واسط Facade آن را به وابستگی غیرمستقیم تبدیل

کرده است. حال از این به بعد، همه ی ارتباطات از بیرون به داخل زیرسیستم، مشخص بوده و از طریق واسط های موجود روی آن انجام می شود که این خود بسیار به وابستگی پایین کمک می کند.

الگوی State: ارتباط از نوع Indirect در این الگو به چشم می خورد. متوجه هستیم که Context نیازی نیست تا همه ی انواع State را بشناسد و با آن ارتباط داشته باشد؛ این واسط State هست که تعبیه شده تا Context بی نیاز از رابطه ی سطح پایین اینچنینی شود و ارتباط خود را در سطح بالا نگه دارد تا منتزع از تغییرات زیرکلاس های حالت شود. از طرفی اما پیکربند نیاز دارد تا همه ی کلاس های موجود در سیستم را بداند و با آن ها ارتباط داشته باشد.

مقایسه: هر سه الگو در ارائه ی وابستگی پایین نسبت به حالت قبل از اعمال الگو موفق بوده اند. دقت داریم اما که پیکربند برای الگوی State و Builder نتوانسته است تا وابستگی را در حد مطلوب، پایین حفظ کند. مخصوصا برای الگوی Builder که کلاینت، پیکربند قطعی آن است.

۹.۳ همبستگی

الگوی Builder: با آوردن کارگردانی که به خوبی می داند روال ساخت محصول چیست، دغدغه های پراکنده ی جانبی از کلاینت و خود سازنده ها برداشته شده و همه در کلاس کارگردان متخصص ، متمرکز می شوند که این خود دلیلی بر همبستگی بالای این الگو می باشد. با این کار هم خود کارگردان، انسجام دارد و هم بقیه ی کلاس های مجموعه به انسجام رسیده و از شلختگی دور می مانند.

الگوی Facade: وجود میانجیگری مانند Facade به انسجام مجموعه خیلی کمک می کند. عدم تمرکز داشتن بی مورد کلاس های زیرسیستم برای آگاهی از منطق روال پردازش درخواست بیرونی، کمک می کند تا تمرکز کاری هر کلاس بیشتر شده و همبستگی بهتری مشاهده می خورد. دقت داریم که خود Facade نباید درگیر کارهای دیگری جز گرداندن یک درخواست خارجی باشد تا بتوان منسجم بودن را برای آن تصور کرد.

الگوی State: این الگو توانسته است تا با معرفی واسط State انواعی از مقادیر متفاوت حالت را این بار در کلاس هایی کپسوله کند تا رفتار را در آنجا بروز دهد. این کنار هم قرار گرفتن رفتار در کنار داده ی خود به شکل مطلوبی انسجام است. به علاوه آنکه هر حالتی، متخصصی برای بروز یک رفتار است که سبب تمرکز و تک‌ـکاره بودن هر یک می شود و از بهم ریختگی رفتار در Context جلوگیری می شود که این خود همبستگی است.

مقایسه: هر یک از ۳ الگو می توانند همبستگی بالایی از خود نشان دهند که سبب انسجام مجموعه می شود.

۱۰.۳ اثرات جانبی

الگوی Builder: در کنار کاهش کارآیی و سرعتی که در قسمت قبل بدان پرداخته شد، این الگو باید کلاینت را از همه ی کلاسها آگاه بگذارد که این خود از اثرات جانبی منفی به شمار می آید. برای مساله حافظه نمی توان آنچنان تاثیری برای اضافه شدن کارگردان و زیرکلاس های سازنده در نظر گرفت. فقط می توان گفت تاثیر آن برای مصرف حافظه مقداری بیشتر شده است.

الگوی Facade: دقت داریم که این الگو شیء آنچنان زیادی را به سیستم تحمیل نمی کند که بخواهد آن را دچار مشکل حافظه جدی بکند. اما مشکل حافظه و کارآیی آن که پیشتر نیز گذشت، برای آن وجود دارد. در سویی دیگر، وجود مشکل بالقوه ای برای واسط های Facade آن است که در صورت افزودن کارهای بی ارتباط با وظیفه ی اصلی آن، می تواند تشکیل God Class and Bottleneck بدهد. به علاوه آنکه برای دوری از عواقب منفی در آینده بهتر آن است تا چندین واسط Facade را به جای یک Facade سنگین قرار دهیم تا از اثرات جانبی ای مانند شلوغی و ناخوانی کد آن مصون بمانیم.

الگوی State: این الگو مشکل حافظه ای را به مانند Builder دارد. تعریف و ایجاد کلاس هایی از انواع State سبب می شود تا اشیاءیی در هنگام اجرا در حافظه نگهداری شوند که پیش از این الگو اصلا وجود نداشته اند. مشکل کارآیی و کم کردن سرعت پاسخگویی از دیگر عواقب منفی این الگو می تواند

تلقی شود.

مقایسه: الگوی Facade مشکل حافظه ای را مانند دو الگوی دیگر خیلی حس نمی کند اما در عوض می تواند باعث ایجاد Bottleneck یا حتی بروز God Class بشود. در عوض اما هر سه مشکل کاهش کارآیی را از اثرات جانبی خود برمی شمارند.

۱۱.۳ کپسوله سازی

الگوی Builder: این الگو با قراردادن میانجی ای به نام Director سعی در گردآوری داده در کنار رفتارش در یک جا می کند. بوجود آوردن آبجکت متخصص در کنار ارتباط با واسط Builder ها به رعایت خوب کپسوله سازی کمک کرده است.

الگوی Facade: این الگو، کاری به ارتباط بین کلاس های درون زیرسیستم نداشته و صرفا تمرکزش روی تعریف و گذاشتن واسط هایی روی زیرسیستم جهت دسترسی کلاینت های بیرونی به درون زیرسیستم است. هیچ دسترسی ای از جانب واسط های Facade روی حالت های درونی کلاس های زیرسیستم و تعاملات آن ها وجود ندارد و اگر چنین باشد در اصل اشتباه هست و کپسوله سازی برای آن رعایت نشده است. مشتری بیرونی نیز از احتمال دید داشتن روی اشیاء و داده های زیرسیستم دور می شود. پس کپسوله سازی میان اجزای این الگو برقرار است.

الگوی State: این الگو با تعریف کپسول های داده_رفتار(مشخصا منظور کلاس های ConcreteState می باشد) که اصلا هدف الگو بوده ، کپسوله سازی را به شکل خوبی فراهم کرده است. رفتار بروزدهنده از یک آبجکت که در کلاس حالت آن قرار گرفته است، در نزدیکی داده ی خود که قرار است این رفتار روی آن داده انجام شود، قرار گرفته است. یعنی درواقع الگوی State آمده است تا رفتار متغیر برای یک داده ی شیء را encapsulate کند و آن را در کنارش قرار داده است.

مقایسه: هر سه الگو در کپسوله سازی موفق عمل می کنند. فقط باید متوجه

خطراتی که Facade با آن روبرو هست باشیم. بدین معنا که از cohesive بودن نیافتد و دیدهای غیرضرور به داخل زیرسیستم نداشته باشد. دو الگوی دیگر در ساختن شیء متخصص اطلاعات که بعدا نیز به آن خواهیم پرداخت، کپسوله سازی را به خوبی پشتیبانی می کنند.

۱۲.۳ انتشار تغییرات

الگوی Builder: وجود ارتباط سطح بالا از Director به واسط انتزاعی Builder سبب شده است تا تغییر و توسعه در زیرکلاس های Builder بی دغدغه نسبت به انتشار تغییرات صورت پذیرد. یعنی تا زمانی که خود Interafce Builder دستخوش تغییر نشود، کارگردان تغییری را احساس نخواهد کرد. نکته منفی اما کلاینتی است که پیکربند مجموعه است و باید از هر تغییری آگاه بوده و خود را با تغییر، تطبیق دهد. این تغییر منتشرشونده برای همه ی پیکربندها می باشد.

الگوی Facade: انتشار تغییرات اگر در درون کلاس های زیرسیستمی باشد که میانجیگر Facade آن ها را می شناسد و در ارتباط است، آنگاه به دلیل دانش حداقلی Facade از خود کلاس ها و همچنین عدم نیاز برای کار و تعامل با آن ها باعث می شود تا Facade از تغییر حفظ شود. کلاینت بیرونی نیز به دلیل ارتباط مستقیم با Facade درنتیجه بی تغییر خواهد ماند و مادامی که Facade تغییر نکند، نیاز به آگاهی از تغییر نخواهد داشت. پس این الگو انتشار تغییرات را به حداقل رسانده است.

الگوی State: به کارگیری از یک interface سطح بالا به نام State موجب شده است تا تغییرات منتشرشونده از کلاس های concrete از نوع State به Context وجود نداشته باشد اما همچنان برای پیکربند ثالث این قضیه صدق نمی کند و تغییرات در هر صورت به او انتشار می یابند.

مقایسه: برای دو الگوی Builder و State مشکل انتشار تغییرات در پیکربند احساس می شود در حالی که از آن طرف هر سه الگو در بخش هایی قابلیت تغییرپذیری را بدون منتشرشدن تغییرات فراهم کرده اند.

۱۳.۳ میزان استفاده از منابع سیستمی

الگوی Builder: استفاده از منابع سیستمی در این الگو به دلیلی که پیشتر بیان شد، می تواند به خاطر افزایش یافتن احتمالی اشیاء(کارگردان و سازنده ها) موجود در برنامه تحت تاثیر قرار بگیرد و حافظه را مقداری بیشتر به خود مشغول کند. از طرفی نیز به دلیل سربارهای پردازشی ناشی از واسپاری کارها و عدم پاسخگویی و ارتباطات مستقیم می تواند درگیری را برای توان پردازشی سیستم به همراه داشته باشد.

الگوی Facade: مشکل حافظه ی خیلی خاصی نمی تواند برای این الگو دردسرساز باشد. بدین منظور که با اعمال این الگو تعدادی محدود از آبجکت های Facade به سیستم اضافه خواهند شد که هرچند حافظه را به خود مشغول می کند، اما خیلی مصرف سیستمی را به همراه ندارند.

الگوی State: ساختن اشیاء از انواع کلاس های ممکن State سرباری را به وجود می آورد که حافظه ی سیستم مجبور است در زمان اجرا، آن ها را حفظ کند. این می تواند مشکلی اندک را برای حافظه داشته باشد. از نظر پردازشی نیز از آنجا که رفتار با واسط، واسپاری می شود می تواند مصرف پردازشی سیستم را به دنبال داشته باشد.

مقایسه: در مقام مقایسه باید گفت که هر کدام شاید مصرف بیشتری را منابع حافظه ای و پردازشی سیستم و مخصوصا برای حافظه به همراه داشته باشند اما Facade در این زمینه نسبت به دو الگوی دیگر بهتر عمل می کند. تعداد اشیاء اضافه شده در آن کم می باشد اما به مانند آن دو باید برای عملکرد، کار را به اشیاء محول کند که درگیری پردازشی دارد.

۱۴.۳ استفاده از شیء جعلی

الگوی Builder: کارگردانی که در این الگو وجود دارد ما_به_ازای خارجی ندارد و نمی توان گفت که نظیر آن در قلمرو مساله یافت می شود؛ چرا که تا پیش از این کلاینت، خود به صورت مستقیم با سازندگان در ارتباط بود و هر دو نیز از نیازمندی های قلمرو مساله برآمده بودند.

الگوی Facade: کلاسی که واسط بین کلاینت های بیرون از زیرسیستم و کلاس های درون آن قرار می گیرد از دنیای خارجی و در قلمرو مساله نشات نمی گیرد. این خود نشان می دهد که Facade موجودیتی جعلی است که اضافه شده است تا کارچرخانی را انجام دهد و دید به زیرسیستم را برای بیرون یکپارچه کند.

الگوی State: این الگو، کلاس های concrete از نوع State در آن، متناظری در دنیای مساله ندارند و فقط به واسطه ی بهره گیری از الگو پدید می آیند، پس جعلی محسوب می شوند.

مقایسه: هرکدام از الگوهای بالا از شیء جعلی استفاده می کنند. Director برای الگوی Builder و Facade برای الگوی Facade جعلی حساب می شوند. انواع زیرکلاس حالت نیز جعلی هستند.

۱۵.۳ سادگی پیاده سازی

الگوی Builder: تقریبا هیچ کدام از زبان های شیءگرا برای پیاده سازی این الگو مشکلی را ندارند. نکته ای که در پیاده سازی آن می تواند اهمیت یابد این است که متدهای موجود در انواع Build باید در حد درشت دانگی مناسبی نوشته شوند. به گونه ای که مثلا یک سازنده ی خاص اگر بخواهد آن را برای خود پیاده سازی کند، سازنده ی دیگری با بزرگی یا کوچکی زیاد آن مشکلی نداشته باشد. به عنوان مثال سازنده ی پیتزا به شکل هاوایی، یک متد اضافه کردن گوشت را در تک_لایه ی وسط نیاز دارد اما سازنده ی پیتزای ایتالیایی از دو متد برای پر کردن دو لایه ی وسط خود استفاده می کند. دقت به این

نکات در پیاده سازی موجب وحدت معنایی می شود و به درک و فهم و همچنین خوانایی کد کمک می کند.

الگوی Facade: پیاده سازی ساده ای برای این الگو قابل تصور است که در همه ی زبان های برنامه نویسی قابل اعمال است. تعریف واسط های Facade برای کلاس های زیرسیستم، از ساده ترین کارهایی است که زبان های سطح بالای برنامه نویسی شیءگرا آن را پشتیبانی می کنند. بهتر آن است که از پسوند Facade برای این واسط ها استفاده شوند تا مراقب افزودن عملکرد به آن باشیم و آن را از انسجام نیاندازیم.

الگوی State: برای پیاده سازی این الگو مشکل خاصی وجود ندارد و در هر زبان شیءگرایی می تواند اعمال شود.

مقایسه: مشکل خاصی برای پیاده سازی الگوها در زبان های مبتنی بر اصول شیءگرایی وجود ندارد و فقط بهتر آن است که در الگوی Builder اتحاد معنایی در قسمت متدهای BuildPart وجود داشته باشد تا سهولت در پیاده سازی راحتتر باشد.

۱۶.۳ موارد کاربرد

الگوی Builder:

- زمانی که الگوریتم برای ساختن یک آبجکت پیچیده باید مستقل از قسمت هایی باشد که آبجکت را می سازند و همچنین قطعات سرهمبندی می شوند.
- زمانی که فرآیند ساختن اشیاء باید اجازه ی نمایش های مختلف را برای آبجکتی که ساخته می شود بدهد. نمایش از مرحله های اولیه تا مراحل پیچیده تر را شامل می شود.

الگوی Facade:

- زمانی که می خواهیم یک واسط ساده را برای زیرسیستمی پیچیده تامین کنیم.

- زمانی که تعداد وابستگی های زیادی بین کلاینت و کلاس های سطح پایین یک انتزاع وجود دارد.
- زمانی که می خواهیم تا زیرسیستم را لایه بندی کنیم. بدین منظور از Facade استفاده می کنیم تا هر نقطه ورودی را به هر سطح از زیرسیستم تعریف کنیم.

الگوی State:

- زمانی که رفتار یک آبجکت بستگی به حالت آن داشته باشد و در زمان اجرا برحسب آن حالت باید رفتارش را تغییر دهد.
- زمانی که عملیات، تعداد زیادی گزاره ی شرطی چندقسمته دارند که مبتنی بر حالت آبجکت می باشد.

مقایسه: با بررسی کاربردهای هر یک از الگوها، شباهت یا تفاوت خاصی بین این سه پیدا نشد؛ زیرا کاربردهای هر یک مخصوص به خود و دنبال انگیزه های متفاوتی است.

۱۷.۳ الگوهای مرتبط با الگو

الگوی Builder: از نظر ساختاری می توان گفت که این الگو بی شباهت به ساختار الگوهای State and Strategy نیست. غالبا از آنجا که شیءای را باید سازندگان بسازند و آن شیء پیچیده است، پس می توان ارتباط خوبی بین این الگو و الگوی Composite برقرار کرد؛ به گونه که واسط سازنده با واسط یک شیء مرکب رابطه دارد.
الگوی Facade: نکته ی مثبتی که برای کلاس Facade این هست که از آن شیء، یکی در سیستم در هر لحظه وجود داشته باشد که این ما را به ارتباط با Singleton هدایت می کند. شباهت دیگر آن با Mediator است؛ جایی که میانجی ای را برای ارتباط قرار داده ایم اما تفاوت در آنجاست که این میانجی

دخالتی در درخواست ندارد و فقط کارها را از بیرون به داخل منعکس می کند، نه اینکه روی خود کلاینت های بیرونی اثری داشته باشد.

الگوی State: شباهت ساختاری با الگوی Strategy برای این الگو قطعی است(شباهت آن با Builder که نیز پیشتر آمد). تنها می توان تفاوت را در معنای آن ها دانست. به علاوه آنکه خود این الگو در اکثر الگوهای دیگر، کاندید خوبی برای اعمال است. مثلا در الگوی Observer که بحث State وجود دارد و از بخش های کلیدی آن است می تواند از این الگو بهره ببرد.

مقایسه: الگوی State کاربرد زیادی دارد و در اکثر مواقع به کار برده می شود و درنتیجه با الگوهای بیشتری مربتط می شود. از نظر ساختار نیز این الگو با Builder نزدیک هستند و الگوی Facade می تواند به صورت موثر از Singleton استفاده کند و کمی هم از نظر میانجیگری به Mediator شباهت دارد.

۱۸.۳ OCP

الگوی Builder: در این الگو به دلیل ارتباط سطح بالا از کارگردان به in-terface سازندگان، هر توسعه ای برای زیرکلاس های سازنده باز است و با این توسعه هر تغییری برای کارگردان بسته است.

الگوی Facade: همه چیز به کلاس واسط Facade بستگی دارد؛ یعنی آنکه مادامی که تغییر ـ توسعه ای برای این کلاس اعمال نشود، تغییری متوجه کلاینت بیرونی نخواهد بود. اما با هر توسعه ای در آن، کلاینت ها مجبور هستند نسبت به آن آگاه باشند. تغییرات توسعه ای زیرسیستم نیز باعث می شود Facade تحت تاثیر قرار بگیرد اما کلاینت از دانستن آن بی نیاز هست و اجباری برای تغییر ندارد. پس جایی که OCP در این الگو زیرسوال می رود آنجایی است که چه تغییر ناشی از کم و زیاد شدن زیرسیستم که دانستن آن برای Facade ضروری است موجب تغییر Facade شود و چه خودش دستخوش تغییر شود، کلاینت های بیرونی نسبت به آن بی تفاوت نبوده و دچار تغییر خواهند شد.

الگوی State: این الگو با تعریف رابطه ی سطح بالای انتزاعی بین Context

و State توانسته است تا این امکان را برای زیرکلاس های حالت فراهم کند تا نسبت به توسعه و کم و زیاد شدن باز باشند و در عین حال تغییرات برای جایی خارج از مجموعه حالات بسته باشد، مادامی که واسط آن ها تغییر نکرده است.

مقایسه: دو الگوی Builder و State اصل OCP را بخوبی رعایت می کنند اما الگوی Facade به دلیل ارتباطات سطح پایین و بدون interface مشخص نمی تواند در این معیار بی نقص بماند.

LSP ۱۹.۳

الگوی Builder: در این الگو، هر زیرکلاسی می توانند نوعی از فوق کلاس باشد و می تواند به جای کلاس پدر بنشیند. یعنی آنکه می توان از آن رفتاری به مانند عملکرد فوق کلاس خود انتظار داشت(با رعایت پیش_شرط و پس_شرط و عدم قوی کردن پیش_شرط و عدم تضعیف پس شرط) و آن را به خوبی جایگزینی کند. به همین دلیل ارتباط کارگردان با فوق کلاس برقرار شده و برایش فرقی ندارد که کدام کلاس سطح پیاده سازی به او منتسب خواهد شد. پس LSP در این الگو رعایت شد.

الگوی Facade: ساختار توارثی خاصی در این الگو دیده نمی شود پس بحث پیرامون LSP برای آن موضوعیت خاصی ندارد.

الگوی State: در این الگو، تمام کلاس هایی که از State ارثبری می کنند را نوعی از همان State می توان دانست. چرا که اگر جای آن بنشینند عملکردی مطابق رفتار پدر از خود نشان خواهند داد و چیزی نیست که Context بخواهد از آن آگاه باشد و رعایت کند. پس LSP در این الگو صادق است.

مقایسه: الگوی Facade رابطه ی پدر_فرزندی را در خود نمی بیند که بخواهیم برای آن در مورد LSP صحبت کنیم. اما دو الگوی دیگر این معیار را دارند.

DIP ۲۰.۳

الگوی Builder: وجود وابستگی در سطح انتزاع سبب شده است تا این الگو از DIP بهره ببرد. ارتباطی که Director به واسط Builder دارد وابستگی مستقیم به کلاس های concrete از نوع Builder را ازبین می برد که به معنای گسترش پذیری آزادانه کلاس های سازنده خواهد بود و اینکه Director فقط به واسط انتزاعی مرتبط است و از تغییرات زیرکلاس ها مادامی که خود واسط دست نخورده است منتزع می شود. اینکه کلاینت پیکربند با همه ی کلاس ها و آبجکت های موجود وابسته است، بحثی است که DIP را به خطر می اندازد؛ اما می دانیم که ارتباطی از نوع موقتی است.

الگوی Facade: هرچند ازبین رفتن ارتباط مستقیم کلاینت با کلاس های داخلی زیرسیستم شکسته شده است و حالا ارتباط غیر مستقیمی را تجربه می کنند اما DIP جایی به خطر می افتد که رابطه ی خود کلاینت ها با Facade به صورت مستقیم در ساختار پایه آمده است. این رابطه ی مستقیم به معنای گسترش پذیری آزادانه Facade نخواهد بود که موجب می شود تا کلاینت ها نسبت به توسعه ی Facade های سیستم آگاه باشند.

الگوی State: این الگو می تواند DIP را پشتیبانی کند. بدین صورت که با وجود رابطه ی سطح بالای انتزاعی بین Context و واسط State نگرانی از رشد و گسترش زیرکلاس های حالت وجود ندارد و Context منتزع از آن تغییرات آن خواهد بود. اما در هر حال پیکربند ثالث، می تواند تاثیر منفی خود را روی آن بگذارد؛ چرا که باید به همه ی آبجکت ها دید داشته باشد.

مقایسه: الگوی Facade نمی تواند در ساختار پایه ی خود تضمینی برای DIP بدهد اما دو الگوی دیگر جدای از بحث پیکربندی که آثار سوء آن همیشه وجود دارد از DIP پشتیبانی می کنند.

ISP ۲۱.۳

الگوی Builder: وجود Interface Builder در مجموعه این معنی را می دهد که تخصص کاری کپسوله شده است و این باعث بالاتر رفتن انسجام می شود. ایجاد واسط برای همه ی سازندگان می تواند دلیلی برای وجود ISP باشد.

الگوی Facade: تعریف های متعدد از Facade بر روی یک زیرسیستم اغلب اتفاق خوبی است که می افتد و این کلاس واسط را کمک می کند تا Cohesive بودن خود را حفظ کند. البته نبود واسط متخصص برای Capture کردن همه ی انواع Facade و اینکه ممکن است خود تک کلاس Facade آنقدر بزرگ شود که دور از انتظار ساختار پایه نیست، پتانسیل نقض ISP را محتمل می سازد.

الگوی State: گذاشتن رفتار یک برنامه در نزدیکی داده ی خود به کمک واسط مشخصی که تخصص را اجبار می کند، از ویژگی های مثبت الگوی State هست که به Cohesion از طریق وجود واسط State و انواع زیرکلاس های تخصصی آن کمک می کند.

مقایسه: الگوی Facade به راحتی می تواند خطر این را داشته باشد تا بوسیله ی چندـکاره شدن میانجی Facade و هم چنین نداشتن فوق کلاسی برای همه ی انواع آن، در واقع بتواند ISP را نقض کند. اما Builder و State به خوبی از ISP بهره مند هستند.

CRP ۲۲.۳

الگوی Builder: رابطه ی کارگردان با انواع سازنده ، دلیل بر وجود و ترجیح واسپاری به ایجاد ساختار توارثی است. در واقع به جای اینکه حالتی را داشته باشیم که برای هر نوع سازنده زیرکلاسی را بگیریم و کلاینت را مستقیما به آن ها متصل کنیم، کلاس واسط Director درخواست ها را یکبار دریافت کرده و سپس میانجیگری ها را انجام می دهد، بدون آنکه زیرحالت هایی پیچیده بخواهد انواع سازنده را تهدید بکند. پس CRP در این الگو برقرار است.

الگوی Facade: در اینجا هرچند صحبتی از توارث نشده است که بخواهد جایگزین کردن واسپاری برای آن مطلوب باشد، اما در هر صورت درخواست های بیرون از زیرسیستم به کلاس های واسط Facade سپرده می شود و سپس از طریق این آبجکت، کارچرخانی در اشیاء زیرسیستم شکل می گیرد و کارها محول می شوند. پس delegation نقش مهمی را در این الگو دارد و تا حدی CRP برای آن برقرار است.

الگوی State: بهترین نمونه ی CRP در این الگو یافت می شود. به جای وجود انواع حالات مختلف و ساختن ساختار توارثی بلندبالا با اضافه شدن هر حالت، آن چیزی هست که توسط State جلویش گرفته می شود و واسپاری جای آن ها را می گیرد. پس در آن CRP به خوبی وجود دارد.

مقایسه: هر سه الگو توانسته اند که CRP را در حد مطلوبی برقرار نمایند.

PLK ۲۳.۳

الگوی Builder: زنجیره ی دید ترایا در این الگو یافت نمی شود؛ چرا که یک آبجکت در پاسخ به هیچ کدام از درخواست ها ارسال نمی شود. پس اصل PLK در آن دیده می شود.

الگوی Facade: اگر به رابطه ی کلاینتی که Facade را می بیند و سپس رابطه ای که Facade با کلاس های زیرسیستم دارد دقت کنیم، متوجه هستیم که این احتمال وجود دارد تا دیدی که از کلاینت درخواست کننده روی Facade بوجود می آید، ممکن است با طراحی ضعیف Facade موجب آن شود که روی state های درونی آن یعنی کلاس های زیرسیستم نیز دید زنجیری پیدا کند که این خود نشان می دهد با دقت زیادی روی طراحی Facade داشته باشیم تا خود شیءای از کلاس های زیرسیستم را برنگرداند تا زنجیره ی دید ترایا بوجود بیاید و PLK نقض شود.

الگوی State: این الگو هرگز باعث بوجود آمدن زنجیره ی دید ترایا نمی شود، چرا که هیچ شیء ای در ازای دریافت سرویس فرستاده نمی شود، پس در آن PLK ثابت است.

مقایسه: الگوی Facade علیرغم دو الگوی دیگر پتانسیل بالایی برای نقض PLK دارد، به گونه ای که می تواند زنجیره ی ترایایی از کلاینت بیرونی به داخل زیرسیستم بوجود بیاورد. این مورد اما در Builder و State یافت نمی شود.

۲۴.۳ الگوهای GRASP

۱.۲۴.۳ Information Expert

الگوی Builder: در این الگو وجود دو موجودیت کارگردان و سازنده این شبهه را دارد که بالاخره آبجکت خبره ی اطلاعاتی در کجا قرار دارد؟ بدین معنا که آیا اگر الگو کارگردان را معرفی می کند، این کارگردان نباید باشد که نسبت به فرآیند ساخت الگو از خود عملکرد نشان دهد؟ آیا اگر انواع سازنده داریم، به قدر کافی دانش و خبره در اطلاعات هستند که کل فرآیند ساخت به آن ها سپرده شود؟ همه ی این ها بدین معنی است که وجود خبره ی اطلاعاتی می تواند در آن نقض شود؛ چرا که از طرفی کارگردان فقط اطلاعات لازم را برای پیشبرد فرآیند در خود کپسوله می کند اما در طرف دیگر عملکرد فرآیند در سازنده ها رخ می دهد و سازنده ها صرفا سرهم کردن قطعات مختلف را کپسوله می کنند. دقت می کنیم که با آوردن عبارت "کپسوله سازی تاکید می کنیم که داده در نزدیکی رفتار خود حفظ شده است".

الگوی Facade: این معیار در الگوی Facade آن جایی یافت می شود که کلاس های واسط Facade روی زیرسیستم ها قرار می گیرند تا متخصص کارچرخانی درونی برای یک درخواست بیرونی باشند. خبرگی در کنار اطلاعات، چیزی است که در آن دیده می شود. پس Information Expert برای آن صدق می کند.

الگوی State: این الگو با مطرح کردن انواع حالات مختلف برای state درونی، توانسته است تا متخصص اطلاعات در حالت را بوجود بیاورد و به واسطه ی فوق کلاس انتزاعی State همه ی آن را تعریف کرده و در اختیار Context قرار دهد تا هر موقع حالت درونی آن تغییر کرد، رفتاری که آن

آبجکت می داند بروز یابد؛ که این به معنی Information Expert می باشد.

۲.۲۴.۳ Creator

الگوی Builder: این الگو در Creator نمی تواند نمره ی قبولی بگیرد؛ زیرا کلاینت به عنوان پیکربند و در اولویت پنجم، صرفا کسی که می داند اطلاعات لازم را برای ساختن نمونه ها دارد، اشیاء را می سازد، در حالی که دایرکتور در اولویت بالاتری نسبت به آن قرار دارد؛ زیرا رابطه ای مستقیم از نوع Asso-ciation با آن دارد.

الگوی Facade: این الگو کما اینکه پیشتر آمد نیاز به پیکربندی خاصی ندارد پس در آن وجود پیکربند منتفی است و رابطه ی Facade با کلاس های زیرسیستمی مشخص است و خود آن در صورت نیاز، با داشتن اطلاعات حداقلی از آن آبجکت ها را خواهد ساخت تا کارچرخانی شکل بگیرد. اگر آبجکت به هر دلیل در زیرکلاس ساخته شده بود، پس حتما اولویت بالاتر یا مساوی نسبت به Facade داشته که باز هم Creator برای آن صادق است.

الگوی State: در این الگو Creator نقض می شود؛ از آنجا اولویت برای ساختن شیء از انواع حالت با پیکربندی است که در رده ی ۵ قرار دارد اما وجود رابطه ی Association از Context به آن، اولویت Context را بالا می برد.

۳.۲۴.۳ Low Coupling

الگوی Builder: وجود ارتباط سطح بالای انتزاعی از Director به Builder سبب حداقلی شدن وابستگی است. ولی این وابستگی کم، با ارتباط قوی داشتن پیکربندی به نام کلاینت با تمام کلاس ها و آبجکت ها، لطمه ای برای این معیار در الگو بوده و به دلیل ارتباط سطح پایین در حد کلاس های concrete سازنده، وابستگی به شدت افزایش می یابد. به علاوه آنکه می دانیم پیکربند این الگو اغلب بیشتر از بقیه ی پیکربندها، انتساب ها را باید تامین کند و دوره ی

موقتی بودن آن کم هست.

الگوی Facade: این الگو با غیرمستقیم کردن روابط موجود از کلاینت به کلاس های زیرسیستمی، وابستگی را در حد خوبی کاهش می دهد، به گونه ای که رابطه ها را به یک_به_چند تبدیل می کند. پس وابستگی کم در آن به چشم می خورد و باید در این مسیر سعی کرد تا آن را حداقلی نیز نگاه داشت.

الگوی State: در این الگو رابطه ای abstract بین Context و Inter– State face وجود دارد که کمک زیادی به کاهش وابستگی در مجموعه می کند. نکته در آنجایی است که متوجه هستیم دوباره، پیکربند کل مجموعه دید سطح پایین خود را روی همه ی آبجکت ها دارد و مدام با تغییر حالت ها باید پیکربندی را انجام دهد که این خود می تواند خدشه ی جدی ای مانند الگوی Builder برای Low Coupling داشته باشد.

High Cohesion ۴.۲۴.۳

الگوی Builder: این الگو انسجام بالایی را به دلیل ایجاد متخصص کارگردانی و سازندگی فراهم کرده است، به گونه ای که کارگردان تمرکزش را روی فرآیند مختلف ساخت، و سازنده تمرکزش را روی چگونگی ساخت هر قسمت می کند. پس Cohesion بالایی در این الگو مشاهده می شود.

الگوی Facade: ایجاد انسجام از طریق خارج شدن وظیفه ی کلاس های زیرسیستم از اینکه باید در جریان کارچرخانی باشند، میسر است. بدین ترتیب همبستگی زیادی برای Facade و کلاس ها بوجود می آید که Cohesion بالایی را به همراه دارد.

الگوی State: این الگو از طریق ایجاد آبجکت های متخصص برای دانایی از رفتار در کنار داده ای که قرار است رفتار روی آن اجرا شود، توانسته است انسجام خوبی را برای Context به همراه بیارد تا روی تک_کار خود تمرکز کند و درگیر رفتارهای متفاوت با داده های مختلف نشود. همچنین آنکه خود آبجکت های حالت همبستگی خوبی دارند، زیرا روی تمرکز خود برای تک_رفتار پایبند هستند.

Controller ۵.۲۴.۳

الگوی Builder: در این الگو Controller خاصی معرفی نشده است و این وظیفه برای آبجکت خاصی تصریح نشده است.
الگوی Facade: زنجیره ی تعاملی در اصل از خود آبجکت های موجود روی زیرسیستم تحت عنوان Facade شروع می شود و صددرصد این کلاس های واسط وظیفه ی کارچرخانی را داند که برای همین Controller هستند.
الگوی State: در این الگو Controller خاصی معرفی نشده است و این وظیفه برای آبجکت خاصی تصریح نشده است.

Polymorphism ۶.۲۴.۳

الگوی Builder: در انواع سازنده از چندریختی بهره گرفته شده است؛ به گونه ای که Director هر آبجکتی از نوع Builder را می پذیرد و با توسعه ی آن مشکلی ندارد.
الگوی Facade: دقیقا اینکه صحبتی از کلاس پدر برای این الگو وجود داشته باشد، در تعریف ساختار پایه وجود ندارد، پس بحث چندریختی برای آن بلاموضوع است.
الگوی State: این الگو از چندریختی در جایی استفاده کرده است که انواع حالت را زیرکلاس State تعریف کرده و اینگونه Context با هر ریخت از نوع حالت مشکلی ندارد.

Indirection ۷.۲۴.۳

الگوی Builder: این الگو در تحقق Indirection موفق عمل کرده است، زیرا با قرار دادن Director بین کلاینت و سازنده، ارتباط مستقیم، شکسته می شود.
الگوی Facade: یکی از انواع بهترین Indirection در این الگو می باشد، زیرا رابطه های چند به چند مستقیم که باعث پیچیدگی در ساختار و کد می شد، حال با اضافه شدن یک کلاس واسط جهت کارچرخانی ازبین رفته و روابطی

غیرمستقیم را شاهد هستیم که پیچیدگی را نیز کاهش می دهد. رابطه ی بین کلاینت بیرونی و کلاس های درون زیرسیتم Indirect شده است.

الگوی State: در این الگو Indirection اینگونه محقق شده است که کلاینت بیرونی که به Context دید دارد، حال هیچ ارتباط مستقیمی با انواع مختلف State ندارد و کلا هم دسترسی به آن بدون واسطه ای که حالت درونی مهمی دارد، بی معنی نیست.

Pure Fabrication ۸.۲۴.۳

الگوی Builder: وجود Director از قلمروی مساله نشات نمی گیرد و متناظری ندارد؛ که نشان می دهد موجودیتی جعلی است.

الگوی Facade: موجودیت Facade هیچ منشاءیی در دنیای خارج ندارد و نمی توان آن را انعکاسی از قلمرو مساله دانست. پس جعلی محسوب می شود.

الگوی State: موجودیت State و زیرکلاس های آن، آبجکت هایی هستند که در دنیای واقع، نظیر موجودیتی نمی شوند و فقط ساخته ی الگو هستند. پس جعلی هستند.

Protected Variations ۹.۲۴.۳

الگوی Builder: محافظت در برابر تغییر در این الگو را می توان در Capture کردن انواع مختلف سازنده در واسط Builder و فراهم آوردن متدهای لازم برای آن ها دانست که کارگردان بتواند با آن ها کار کند و از هر گونه توسعه یا کم و زیاد شدن نامطلع بماند. کلاس های concrete سازنده بدین طریق، به خوبی دربرابر variation حفظ شده اند.

الگوی Facade: اگر Facade را اینگونه تفسیر کنیم که کلاینت های بیرونی را از تغییرات و توسعه ی درون زیرسیستم، منتزع می سازد و آن ها را بی نیاز از دانستن کم و زیاد شدن ها می کند، پس می توان گفت این الگو در این معیار موفق ظاهر شده است.

الگوی State: وجود فوق کلاس انتزاعی State برای نگهداشتن همه ی حالات مختلف انواع state نویدگر آن است که برای Context شرایطی پیش آمده تا هر آنچه از تغییرات ممکنه در زیرکلاس های حالت را نبیند و بی نیاز از آگاهی باشد. الگوی State اینگونه دربرابر توسعه و تغییر از خود محافظت نشان می دهد.

۴ مقایسه Adapter-Observer-Strategy

۱.۴ دسته

الگوی Adapter: در دسته ی الگوهای ساختاری یا Structural در GoF قرار دارد.
الگوی Observer: در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
الگوی Strategy: در دسته ی الگوهای رفتاری یا Behavioural در GoF قرار دارد.
مقایسه: الگوی Adapter کاملا از این نظر با دو الگوی دیگر متفاوت هستند. الگوی Adapter روی کاهش وابستگی بین واسط کلاس ها و پیاده سازی آن ها تمرکز دارد و در نهایت Strategy and Observer با تعاملات پویای بین اشیاء سروکار بیشتری دارد.

۲.۴ حوزه

الگوی Adapter: این الگو هم در حوزه ی کلاس و هم شیء تعریف شده است. بدین معنا که تحقق الگو را هم می توان در زمان کامپایل و هم در زمان اجرا داشت.
الگوی Observer: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
الگوی Strategy: در حوزه ی شیء می باشد. بدین معنا که تحقق الگو در زمان اجرا می باشد و قبل از اجرا، هنوز الگو محقق نشده است.
مقایسه: تحقق هر سه الگو منوط به زمان اجرا می باشد اما Adapter می تواند در زمان کامپایل نیز از طریق وراثت، تحقق یابد.

۳.۴ هدف

الگوی Adapter: هدف از این الگو آن است که واسط یک کلاس را به واسط دیگری که کلاینت انتظار دارد، تبدیل کند. در واقع Adapter این امکان را می دهد تا کلاس ها با یکدیگر به گونه ای کار کنند که به دلیل ناسازگاری در Interface ها، این همکاری در غیر اینصورت ممکن نبود.

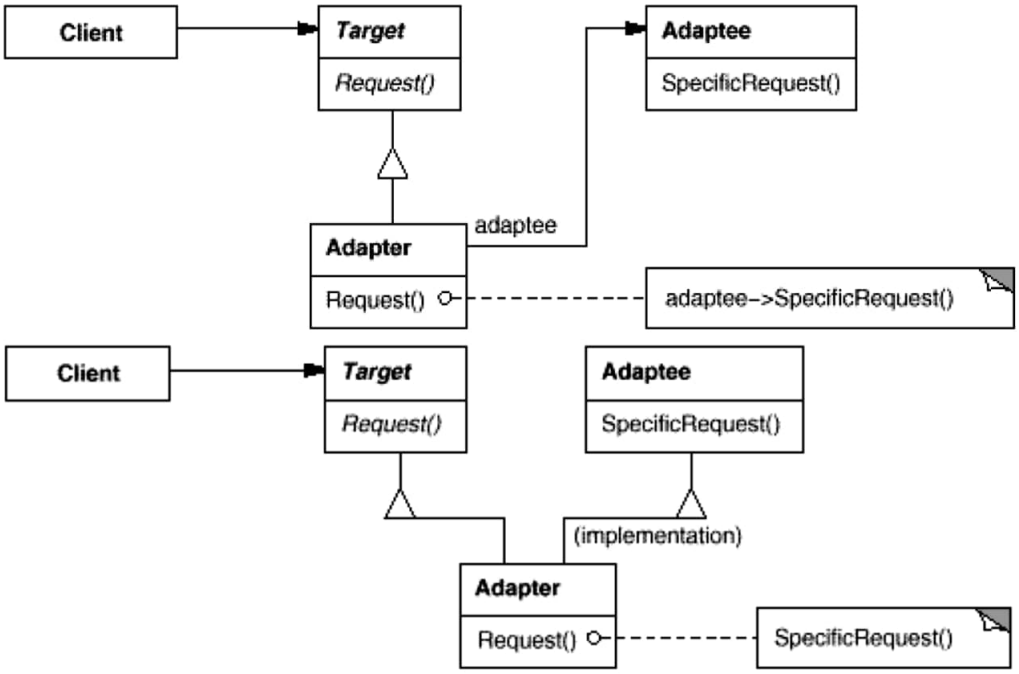
الگوی Observer: هدفی که این الگو دارد این است که یک وابستگی یک‌ـ به‌ـچند را بین آبجکت ها به گونه ای تعریف کند که وقتی یک حالت یک آبجکت تغییر کرد، همه ی وابسته های آن آبجکت خبردار شده و به صورت خودکار آپدیت می شوند.

الگوی Strategy: این الگو می خواهد تا یک مجموعه از الگوریتم را تعریف کند، هر کدام را کپسوله کند و همچنین کاری کند تا آن ها قابل تعویض باشند. این الگو اجازه می دهد که الگوریتم مستقلا از کلاینتی که از آن استفاده می کند، تغییر کند.

مقایسه: هر کدام از الگو تقریبا اهداف متفاوتی را دنبال می کنند. الگوی Adapter برای تبدیل کردن یک واسطی که نمی تواند امکان همکاری را بدهد به واسط دیگری که برای همکاری مطلوب است و الگوی Observer برای وابستگی یک‌ـ به‌ـچندی که با تغییر یکی، وابسته های متعدد آن سریعا خود را با حالت درونی آبجکت تغییریافته، به روز کند. اما الگوی Strategy هدفش در راستای تغییر مستقل الگوریتم با تغییر حالت درونی شیء است.

۴.۴ ساختار

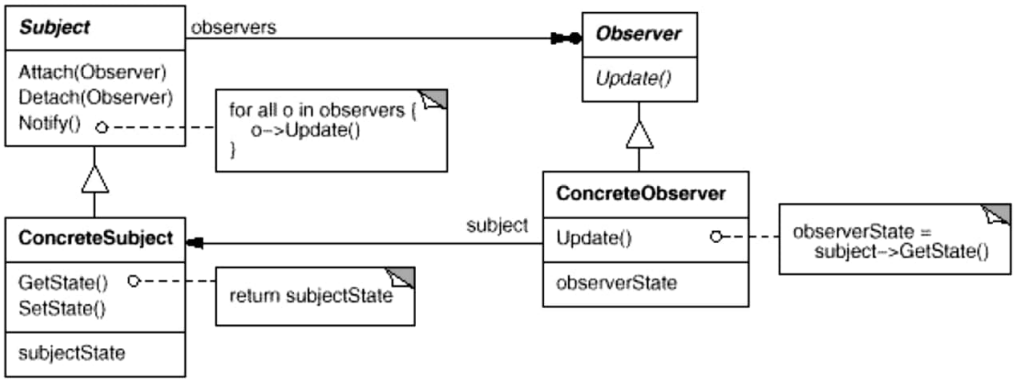
ساختار الگوی Adapter را در شکل ۱۰ داریم: برای این الگو دو نوع ساختار پایه وجود دارد. در ساختاری که در سطح آبجکت است، دقت داریم که کلاینت با واسط Target در ارتباط است و صرفا او را می شناسد. حال هر نوعی از درخواست را دارد باید Request را فراخوانی کند که روی آبجکت Adapter اجرا شده و خود این آبجکت کار را به آبجکت دیگری به نام Adaptee واسپاری



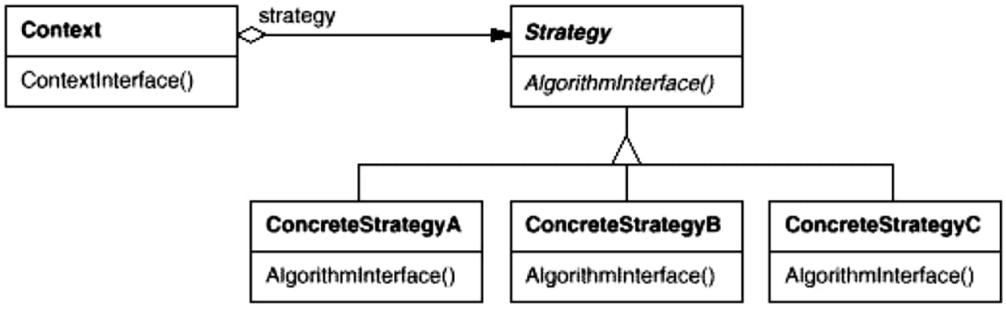
شکل ۱۰: ساختار الگوی Adapter

می کند. درواقع Adaptee همان آبجکتی است که Adapter فقط می داند چگونه با او کار کند تا تبدیل انجام شود و دراختیار کلاینت قرار بگیرد. در بررسی ساختار پایه ی آن در سطح کلاس به این نکته پی می بریم که Adapter با ارثبری از Target و پیاده سازی Interface ی به نام Adaptee می تواند متدهای Request و SpecificRequest را ببیند و با برقراری این دید، مجددا می توان با Adaptee کار کرد و درخواست مشخصی که کلاینت نمی تواند با او کار کند را به او سپرد.

الگوی Observer ساختاری به مانند شکل ۱۱ دارد. این الگو با تعریف دو کلاس واسط انتزاعی در سطح بالا هم برای Subject و هم برای Observer ها امکان تعریف انواع آن را داده اند. دقت داریم که همه ی Subject ها با یک یا بیشتر از Observer در ارتباط هست و آن ها را می بیند که این وابستگی در سطح انتزاع هست؛ اما ارتباط از سمت Observer ها به سمت Subject در حد کلاس های concrete است که بدین سبب هر آبجکت از Observer در



شکل ۱۱: ساختار الگوی Observer



شکل ۱۲: ساختار الگوی Strategy

زمان کامپایل مشخص است که متقاضی دید داشتن روی کدام Subject است. هر Subject ی با Notify برای همه ی Observer های خود متد Update را فراخوانی می کند که به فراخور آن، هر یک از Observer ها از Subject می خواهند تا حالت درونی جدید خود را با GetState ارسال کند.

الگوی Strategy نیز ساختاری مشابه با شکل ۱۲ دارد. وجود Context که دارای یکی از انواع Strategy است. واسط سطح بالای Strategy برای همه ی زیرکلاس های خود این امکان را فراهم می آورد تا الگوریتم سفارشی خود را در AlgorithmInterface بنویسند. حال Context کافی است تا آن متد را فراخوانی کند که برحسب نوع Strategy که به آن منتسب شده است، الگوریتم خاص آن نیز شروع به کار می کند.

مقایسه: هر یک از الگوها در سطح ساختار از دیگری تفاوت بسیار دارند.

دقتی که داریم این است که Adapter در سطح کلاس نیز می تواند وجود داشته باشد و در زمان کامپایل تحقق یابد. این الگو در ساختار خود روی تبدیل درخواست به کمک واسط تمرکز داشته در حالی که Observer با تعریف واسط های سطح بالا، انواع Subject و Observer را Capture می کند و رابطه ی سطح بالا را برای اطلاع رسانی همه ی Observer ها ایجاد می کند و وابستگی سطح پایین را برای گرفتن State آن Subject جدید تغییر کرده قرار می دهد.

۵.۴ پیکربندی

الگوی Adapter: در سطح کلاس، پیکربند فقط مسئول ساخت اشیاء می باشد؛ بدین معنا که بقیه ی موارد برای همکاری آبجکت ها در زمان کامپایل ثابت شده بودند و در زمان اجرا نیازی به تعامل وجود ندارد که پیکربند بخواهد شناسایی ای را انجام بدهد. نمونه گرفتن از Adapter کفایت است. در سطح شیء که هستیم نیز پیکربند اشیاء رامی سازد و ارتباط از Adapter به Adaptee قبل از زمان اجرا وجود دارد و فقط در زمان اجرا واسپاری به تحقق می رسید. در هر دو حالت اما، ممکن است تا این نیاز حس شود که نوع آداپتر برای کلاینت تغییر کند که این کار در کنار کارهای نمونه گیری، در زمان اجرا توسط پیکربند نیز انجام می شود.

الگوی Observer: پیکربند سومی نیاز هست تا از هر کدام از اشیاء نمونه گیری را انجام دهد. این پیکربند کار خاص دیگری را نمی کند زیرا Observer ها خود را با Detach and Attach جزء لیست یک Subject می گذارند یا برمی دارند. بدین شکل ارتباط از Subject به Observer برقرار است و پیکربند نیازی به شناساندن ندارد. ارتباط معکوس نیز که در سطح پایین برقرار است و از قبل کامپایل شده است که هر آبجکت از نوع Observer دقیقا به کدام Subject دید دارد. پس کلا پیکربند برای ایجاد اشیاء فقط به میدان می آید و در ادامه کار دیگری را ندارد و خود اشیاء با یکدیگر بدون حضور پیکربند تعامل می کنند.

الگوی Strategy: پیکربند در این الگو نوع شیء استراتژی قرارگرفته در

اختیار Context را می تواند در زمان اجرا عوض کند. یعنی برحسب شرایطی که رقم می خورد، پیکربند ثالث از کلاس concrete استراتژی موردنظر نمونه شیء را ساخته و آن را به Context منتسب می کند.

مقایسه: هر یک از الگوها به پیکربند ثالث نیاز داشته اما در الگوی Ob-server و Adapter در سطح کلاس، این پیکربند فقط برای شیء ساختن استفاده می شود و بقیه تعاملات و انتساب ها بدون حضور پیکربند انجام می شود. در الگوی Strategy و Adapter در سطح شیء، پیکربند شاید در زمان اجرا تصمیم بگیرد تا نوع شیء منتسب به Context یا کلاینت را تغییر دهد.

۶.۴ انعطاف پذیری

الگوی Adapter: پیکربندی مجدد در این الگو زمانی که در سطح کلاس هستیم، موضوعیت ندارد. چرا که قبل از اجرا الگو محقق شده است و به واسطه ی وراثت(از نوع دوگانه) می توان به این پی برد که انعطاف پذیری، صفر است و امکان بازپیکربندی اصلا در آن وجود ندارد. زمانی که در سطح آبجکت هستیم نیز بازپیکربندی خاصی نیاز نخواهد بود؛ اما در هر دو حالت، پیکربندی می تواند نوع Adapter انتساب داده شده به کلاینت را تعویض نماند و دوباره پیکربندی کند که خود نوعی انعطاف پذیری است.

الگوی Observer: انعطاف پذیری بالای این الگو از طریق وجود رابطه در سطح دو واسط در سطح بالا می باشد. بدون حضور پیکربند خارجی می توان الحاق و خارج شدن Observer ها از لیست Subject ها داشت. پس به دلیل همین بازپیکربندی مطلوب، انعطاف پذیری خیلی خوبی وجود دارد. تنها کافی است پیکربند خارجی، نمونه شیء را از Observer موردنظر بسازد و سپس آن شیء خود را به Attach پاس داده تا جزء لیست Subject قرار بگیرد و مجموعه بدین گونه مثلا پیکربندی شود.

الگوی Strategy: پیکربند ثالث درهر زمان از اجرا می تواند نوع شیء استراتژی منتسب شده به Context را تغییر بدهد که این به دلیل ارتباط سطح انتزاع Context به واسط Strategy است. همه ی این ها سبب انعطاف پذیری

خوب این الگو می شود.

مقایسه: هر یک از این ۳ الگو دارای انعطاف پذیری هستند اما Strategy با پیکربند ثالث و Observer بدون حضور آن، این را فراهم می آورد در حالی که در Adapter در هر دو حالت انعطاف پذیری شامل قسمتی از الگو است که می خواهیم Adapter را برای کلاینت تغییر دهیم.

۷.۴ کارآیی

الگوی Adapter: کارآیی در حالت کلاس، کاهش نخواهد داشت؛ چرا که فراخوانی بدون واسپاری است. اما در حالت شیء می توان گفت به دلیل ردوبدل شدن پیغام بین Adapter و Adaptee سربار زمانی ای شاهد هستیم که سبب کاهش کارآیی می شود.

الگوی Observer: اگر در نظر بگیریم که یکی از Subject ها دچار تغییر شود، پیغامی را برای تک تک Observer های خود باید بفرستد تا هر کدام به ترتیب از آن بخواهند که State درونی Subject را ببینند و به آن دسترسی داشته باشند. یعنی این خود Observer ها هستند که باید با Subject ارتباط بگیرند که این کار ممکن است شدیدا سربار زمانی را به همراه داشته باشد و کارآیی را کاهش دهد. دقت داریم که اگر از این الگو استفاده نکنیم با رابطه های مستقیم سطح پایین، تا چه حد میزان تبادل پیغام کم و کارآیی بالا بود. در آن چنین حالتی Subject خود می توانست به هر کدام State لازمه را مستقیما بفرستد.

الگوی Strategy: کارآیی در این الگو با کاهش روبرو هست؛ زیرا Context دیگر نمی تواند مستقیما به کلاینت بیرونی، نوع الگوریتم را بروز دهد و این کار را به شیء متخصص محول می کند که این خود سربار زمانی دارد. این تبادل پیغام از Context به ConcreteStrategy سبب کاهش کارآیی می شود.

مقایسه: کاهش کارآیی در هر یک از الگوها به سبب ارتباطات غیرمستقیم و تبادلات زیاد پیغام بین آبجکت ها، نسبت به حالت قبل خود، کارآیی کمتری دارند. کلا واسپاری امور سربار پردازشی دارد. در این بین اما Adapter در

سطح کلاس متوجه این کاهش کارآیی نخواهد شد؛ چون از ساختار صلب توارثی استفاده می کند.

۸.۴ وابستگی

الگوی Adapter: زمانی که در سطح کلاس هستیم، تنها وابستگی از نوع ارثبری است که این رابطه بسیار قوی می باشد و باعث وجود وابستگی قوی در الگو می شود اما زمانی که در سطح شیء هستیم، یک رابطه ی وراثت حذف شده و جای خود را به رابطه ی Association ای می دهد که Adapter را به Adaptee متصل می کند. این رابطه هر چند در سطح کلاس های انتزاعی است و همچنان وابستگی بالایی است اما به مانند شکل دیگر این الگو، قوی نیست و وابستگی ضعیف تری را نشان می دهد. در این جا باز هم تغییرات در هر یک سبب تغییر احتمالی در دیگری خواهد شد اما با ارثبری این تغییر احتمالی، حتمی است و با هر تغییر در Adaptee قطعا Adapter نیز تغییر کرده است.

الگوی Observer: در این الگو از طریق ارتباط سطح بالا برای دو Interface و وابستگی یک طرف در سطح بالا می توان ابراز کرد که در این قسمت، وابستگی کاهش یافته است. در ارتباط سطح پایین بین کلاس های غیرانتزاعی، وابستگی شدید هرچند موقتی دیده می شود. اما در کل دقت داریم که با ایجاد رابطه ی یک_به_چند تا چه حد از وابستگی مستقیم که قبلا چند_به_چند بوده است، بهبود وابستگی داریم. البته طبق روال همیشگی، پیکربند برای ساخت اشیاء باید به تمام کلاس ها دید داشته باشد که باعث افزایش وابستگی می شود.

الگوی Strategy: در این الگو وابستگی در حداقل هست؛ چرا که ارتباط Context با فوق کلاس Strategy در سطح انتزاع هست و سبب می شود تا نیاز نباشد که Context ارتباط مستقیم با همه ی انواع Strategy داشته باشد. طبق معمول، پیکربند دارای وابستگی زیاد با همه ی کلاس ها می باشد و باید همه آن ها را بشناسد.

مقایسه: در هر دو شکل الگوی Adapter ارتباط تقریبا قوی در الگو یافت می شود و البته که در نوع کلاس آن بسیار قوی تر نیز می باشد. در الگوی

Strategy و Observer اما این معیار کمرنگ تر شده است در حالی که به دلیل وجود پیکربند و لزوم ارتباط با همه ی کلاس ها، همچنان در این قسمت وابستگی زیاد مشهود است.

۹.۴ همبستگی

الگوی Adapter: چه در سطح کلاس و چه در سطح شیء، وجود موجودیت Adapter شیء متخصص را به مجموعه اضافه می کند که سبب عدم ارتباط مستقیم کلاینت و Adaptee می شود. این تخصص باعث می شود که Adaptee روی کار خود متمرکز باشد و سرویس دهی را به همان شکلی که خود می داند حفظ کند. این تک_کاره شدن ها باعث همبستگی می شود.

الگوی Observer: با تک_منظوره شدن کلاس ها به گونه ای که اشیاءیی فقط جهت رصد حالت اشیاء دیگر هستند، انسجام مجموعه بالا می رود. پس به دلیل پخش شدن وظیفه در کلاس های متخصص و ارتباط گرفتن در سطح انتزاع، وابستگی کم شده و متقابلا این همبستگی است که افزایش می یابد.

الگوی Strategy: با توزیع شدن انواع الگوریتم مبنی بر حالت در اشیاء مختلف، اشیاء متخصصی در مجموعه بوجود می آیند که منسجم هستند. با بیرون کشیده شدن انواع مختلف روش کار از داخل Context و سفارشی سازی آن ها در کلاس ها جداگانه، همبستگی مجموعه افزایش می یابد.

مقایسه: در هر یک از الگوها با ایجاد متخصص و حفظ تمرکز هر کلاس روی منظور خود، همبستگی زیاد می شود.

۱۰.۴ اثرات جانبی

الگوی Adapter: به همراه کاهش کارآیی که به آن پرداختیم، مشکل حافظه نیز متوجه این الگو می باشد، به گونه ای که با اضافه شدن موجودیت Adapter حافظه باید برای نگهداری آن جایی را درنظر بگیرد. از طرفی آنکه زمانی که از سطح کلاس آن استفاده می کنیم، باید به ملاحظات پیاده سازی دقت داشته

باشیم. مانند اینکه وراثت دوگانه چه گونه می شود و کلا باید یادآور شویم که ساختار خشک و صلب توارث یک اثر جانبی منفی می تواند تلقی شود.

الگوی Observer: جدای از مساله ی کارآیی که از اولین عواقب جانبی الگو می باشد، به دلیل تعریف اشیاء Observer که جدیدا اضافه شده اند، حافظه نیز درگیر این خواهد شد تا متوجه نگهداری آنان نیز باشد. سربار حافظه نیز در این الگو واضح است که سبب می شود دومین عواقب از عواقب جانبی الگو تصریح شود. ملاحظات پیاده سازی مثل اینکه در زمان ارسال پیام آپدیت اگر آپدیت دیگری متوجه Subject شد چه استراتژی خاصی پیش گرفته شود، از دیگر عوارض جانبی الگو است.

الگوی Strategy: در کنار کاهش کارآیی که از آن سخن گفتیم، احتمال رشد وجود اشیاء در سیستم از طریق نمونه گیری از انواع کلاس های استراتژی باعث الزام برای حافظه می شود که مکانی را برای نگهداری اشیاء جدید در نظر بگیرد و این از آثار سوءیی است که از این الگو سر می زند.

مقایسه: هم سربار پردازشی و هم حافظه از مهم ترین مشکلاتی است که بین هر ۳ الگو مشترک است. ساختار توارثی بدون انعطاف برای الگوی Adapter در سطح کلاس به همراه ملاحظات وراثت دوگانه در پیاده سازی، نکات منفی هستند که باید به آن دقت شود. ملاحظلات پیاده سازی برای انتخاب استراتژی آپدیت شدن به هنگام Notify آپدیت از معضلاتی است که می تواند برای الگوی Observer دقت شود.

۱۱.۴ کپسوله سازی

الگوی Adapter: نزدیک بودن داده به رفتار خودش که قرار هست روی انجام شود در سطح کلاس الگوی Adapter به خوبی دیده می شود. استفاده ی مستقیم از SpecificRequest از طریق فراخوانی Request شاهدی بر این مدعاست. پس کپسوله سازی خوبی در کلاس Adapter صورت می گیرد. در سطح آبجکت دقت داریم که رابطه ی مشخصی بدون هیچ اطلاعات اضافی از حالت درونی از نوع Association بین Adapter و Adaptee وجود دارد که

سبب می شود هر کدام کپسوله سازی خود را به طریق مطلوبی حفظ کنند. بدین منظور که کپسوله های داده رفتاری هستند که رفتار را در کنار داده حفظ می کنند و کپسوله سازی نقض نمی شود.

الگوی Observer: کپسوله سازی در این الگو به شکل خوبی از طریق ارتباط سطح بالای واسط ها دیده می شود که مسبب می شود تا دغدغه های هر کلاس جدا شود. در این بین اما فقط ارتباط مستقیم و سطح پایین که هر Observer به Subject خود دارد و نیاز دارد تا از حالت درونی آن باخبر شود سبب می شود تا داده در نزدیکی رفتار خود نباشد و Observer نیاز به دانستن آن جهت بروز رفتار داشته باشد که ولی از داده های Subject می باشد و این می تواند به معنای دور افتادن داده از عملکرد خود باشد و کپسوله سازی را زیرسوال ببرد.

الگوی Strategy: موردی وجود ندارد که دراین الگو که بخواهد کپسوله سازی را نقض کند؛ چرا که ارتباط از Context به یکی از استراتژی ها باعث می شود که با فراخوانی الگوریتم موردنظر، رفتار روی داده بروز پیدا کند و کپسوله های داده_رفتاری را در استراتژی ها ببینیم که داده با عملکرد خود در یک کلاس bundle شده است.

مقایسه: کپسوله سازی در الگوهای Adapter و Strategy بهتر از الگوی Observer می باشد. بدین صورت که الگوی Observer نمی تواند State را در کنار رفتار هر Observer داشته باشد و باید آن را جهت عملکرد خود از Subject طلب کند.

۱۲.۴ انتشار تغییرات

الگوی Adapter: کما اینکه پیشتر مطرح شد در سطح شیء الگو، به دلیل وابستگی مستقیم دو کلاس زیر از Adapter به Adaptee هرگونه تغییر در Adaptee منجر به تغییر بروز شونده در Adapter می شود. در سطح کلاس الگو نیز به دلیل وراثتی که موجود است، قطعا هر تغییری در فوق کلاس Adaptee کلاس Adapter را تحت تاثیر قرار می دهد و از این بابت خیلی باید محتاط عمل کرد. پیکربند نیز که همیشه وابستگی مستقیم آن سبب الزام مطلع بودن

نسبت به تغییرات می شود.

الگوی Observer: هرگونه تغییری تا زمانی که واسط Object تغییر نکرده است، هیچ تغییری انتشار نمی یابد. زیرا ارتباط یک Subject با Observer در سطح Interface برقرار شده است. اما از طرفی با توسعه یافتن کلاس های از نوع Subject لازم است تا این تغییر برای Observer های متقاضی رصد کردن آن، اعمال شود تا بدان دید پیدا کنند و اگر ثبت نام می کنند باید بتوانند State آن را نیز بگیرند که این کار باید در زمان کامپایل و از طریق دستکاری کد آن ها انجام شود. پیکربند نیز متوجه هر تغییری باید باشد که روال همیشگی است تا بتواند اشیاء را بسازد.

الگوی Strategy: تا زمانی که واسط سطح بالای Strategy دستخوش تغییر نشود، هیچ تغییری متوجه Context نخواهد بود و این آبجکت با توسعه زیرکلاس ها مشکلی ندارد. اما دوباره یادآور می شویم که پیکربند ثالث مجموعه نسبت به تغییر همه حساس بوده و روی آن اثر گذارند.

مقایسه: جدای از بحث تغییرمنتشرشونده ای که در پیکربند برای هر سه وجود دارد، می توان گفت بهترین عملکرد را Strategy دارد. سپس الگوی Adapter در سطح آبجکت و سپس کلاس خود و در نهایت نیز الگوی Ob-server به ترتیب تغییرات کمتر منتشرشونده ای دارند. الگوی Observer به دلیل ارتباط و وابستگی مستقیم سطح پایین خود از Observer ها به Subject ها، با کوچکترین تغییر در Subject این تغییر به Observer ها نیز انتشار می یابد.

۱۳.۴ میزان استفاده از منابع سیستمی

الگوی Adapter: تعداد اشیاء ملزم به نگهداری در زمان اجرا در هر دو حالت سطح کلاس و شیء الگو افزایش یافته اند که سبب مصرف حافظه ی بیشتر سیستم می شود، اما دوباره تکرار می کنیم که این افزایش خیلی زیاد نیست و محدودتر از الگوهایی مثل Command می باشد. واسپاری درخواست(در سطح شیء الگو) نیز که مسبب اصلی استفاده از منبع پردازشی

سیستم است که به وضوح مشخص است عامل اصلی افت کارآیی است.

الگوی Observer: وجود تعداد زیاد اشیاءیی که حاصل از وجود ناظران برای یک سوژه هستند، حجم بالایی را به حافظه ی سیستم متحمل خواهد کرد. می دانیم که معمولا تعداد آبجکت ها در این الگو معمولا کم نخواهد بود و باید انتظار مشغول بودن زیاد حافظه را داشت. از طرفی نیز به دلیل تبادلات گسترده بین آبجکت ها، سربار پردازشی بالایی نیز متوجه این الگو می باشد که در کل می توان گفت استفاده از منابع سیستمی به طور نسبتا بالایی صرف این الگو می شوند.

الگوی Strategy: این که الگوریتم ها در کلاس های مختلف پخش شده اند و هر کلاسی رفتار شخصی سازی خود را کپسوله کرده است، در نتیجه دیگر در دل Context وجود ندارد که بخواهد رفتار در همان آبجکت بروز یابد. این خود سربار پردازشی را متوجه سیستم می کند. علاوه بر آن، الزام نگهداری شیءای از نوع استراتژی می تواند حافظه را به خود مشغول کند که در اکثر موارد تقریبا سربار خیلی حجیمی نیست.

مقایسه: در کل هر سه الگو متوجه افزایش تعداد اشیاء در حافظه هستند و هر کدام به دلیل delegation هایی که دارند، کم و بیش در پردازش تاثیرگذارند. در این بین اما کلا الگوی Adapter می توان گفت که هم تعداد شیء محدودتری وارد مجموعه می کند و هم سطح کلاس آن تاثیر زیادی روی منبع پردازشی سیستم نخواهد داشت و استفاده ی زیادی از آن نخواهد کرد.

۱۴.۴ استفاده از شیء جعلی

الگوی Adapter: قرارگرفتن Adapter که در هر دو سطح کلاس و آبجکت الگو وجود دارد و همچنین کلاس انتزاعی Target برآمده از قلمروی مساله نیست و متناظری در دنیای خارجی ندارد. پس شیءای جعلی است.

الگوی Observer: به نظر نمی رسد که این الگو الزاما موجودیت جعلی داشته باشد؛ بدین معنا که ممکن است دنیای واقع چنین اقتضایی ایجاد کند. مثال معمول آن، نمودارهای مختلفی هستند که به عنوان Observer یک مجموعه

داده مثل بورس یا آمار به عنوان Subject را رصد می کنند. این الگو البته که شیء متخصص را تشریح می کند و می گوید که هر نقشی چه باید در خود گذاشته باشد اما موجودیت جدیدی را از پیش خود اضافه نمی کند. تمام کار آن، تصریح روابط روی نقش ها و تعریف واسط های سطح بالا و پیشنهاد یک ساختار پایه برای الگو می باشد.

الگوی Strategy: هر زیرکلاس استراتژی اصلا ما ـ به ـ ازایی در دنیای بیرون ندارد و به قوت الگو نشات می گیرد. بدین منظور که ابتدا همه چیز در Context جای داشته است که بواسطه ی الگو هر الگوریتم در کلاس مخصوص جدیدی قرار می گیرد. پس این موجودیت ها جعلی هستند.

مقایسه: فقط الگوی Observer هست که در مقام مقایسه، فارغ از موجودیت جعلی است و دو الگوی دیگر از شیء جعلی استفاده کرده اند.

۱۵.۴ سادگی پیاده سازی

الگوی Adapter: برای پیاده سازی نوع شیء آن، سختی خاصی در آن دیده نمی شود و بسیار ساده در همه ی زبان های شیءگرا پیاده سازی می شود. در بحث سطح کلاس باید به این نکته دقت داشت که بعضی زبان های برنامه نویسی شیءگرا امکان وراثت چندگانه را نمی دهند که باید دنبال راههای جایگزین مثل implements برای جاوا که Interface را پیاده سازی می کند، گشت. البته ممکن است کد از خوانایی و درک بیافتد.

الگوی Observer: پیاده سازی این الگو در زبان های برنامه نویسی شیءگرا ساده بوده و ملاحظه ی خاصی برای آن وجود ندارد.

الگوی Strategy: پیاده سازی این الگو در هیچ زبان برنامه نویسی شیءگرایی مشکل خاصی ندارد و به سادگی قابل پیاده سازی است. هرگز نکته ای پیچده ای را وارد کد نکرده و اتفاقا آن را بهتر و خواناتر می کند به گونه ای که روی کد، تمرکز و سبکی بوجود می آید.

مقایسه: الگوها برای پیاده سازی بسیار سرراست هستند اما برای الگوی Adapter در سطح کلاس، باید دقت داشت که بعضی زبان هایی که از وراثت

چندگانه استفاده نمی کنند، ممکن است دردسرساز باشند.

۱۶.۴ موارد کاربرد

الگوی Adapter:

- زمانی که می خواهیم از یک کلاس موجود استفاده کنیم و Interface آن مطابق نیاز ما نیست.
- زمانی که می خواهیم یک کلاس قابل استفاده مجدد بسازیم که با کلاس های غیرمرتبط یا غیرقابل پیش بینی کار کند. بدین معنا که کلاس ها لزوما یک Interface یکسان ندارند.
- زمانی که می خواهیم در سطح آبجکت الگو البته، از چندین زیرکلاس موجود استفاده کنیم اما غیرممکن است که Interface آن ها را توسط زیرکلاس گرفتن از هرکدام تطبیق دهیم. یک آبجکت از نوع آداپتر می تواند Interface کلاس پدرش را تطبیق دهد.

الگوی Observer:

- زمانی که یک انتزاع دو جنبه دارد که یکی به دیگری وابسته است. در این حالت کپسوله سازی این جنبه ها در آبجکت های جداگانه این امکان را فراهم می آورد که هرکدام را مستقلا تغییر داده و از آن استفاده مجدد کنیم.
- زمانی که یک تغییر برای یک آبجکت نیاز است تا دیگران را تغییر دهد و نمی دانیم که چه تعداد آبجکت نیاز به تغییر دارند.
- زمانی که یک آبجکت باید قادر باشد تا به بقیه ی آبجکت ها بدون اینکه فرضیه بسازد آن آبجکت ها کدام آبجکت ها هستند، خبر تغییر بدهد. به عبارت دیگر نمی خواهیم این آبجکت ها در دو جنبه نسبت به یکدیگر خیلی وابسته ی سختی باشند.

الگوی Strategy:

- زمانی که تعداد زیادی کلاس مرتبط فقط در رفتار خود متفاوت هستند. استراتژی ها، یک راهی را برای پیکربندی یک کلاس با یکی از چندین رفتار، تامین می کنند.
- زمانی که به تعدادی الگوریتم متفاوت متغیر نیاز هست. مثلا می خواهیم برای یک بازی فوتبال، الگوریتم های حمله ی متفاوتی را تعریف کنیم.
- زمانی که یک الگوریتم از داده ای استفاده می کند که کلاینت ها نباید درباره ی آن بدانند. برای همین از این الگو استفاده می کنیم تا ساختار داده ای پیچیده و مشخص در الگوریتم را در معرض دید قرار ندهیم.
- زمانی که یک کلاس تعدادی رفتار تعریف می کند و این ها در واقع چندین گزاره ی شرطی در داخل operation آن هستند.

مقایسه: با بررسی کاربردهای هر یک از الگوها، شباهت یا تفاوت خاصی بین این سه پیدا نشد؛ زیرا کاربردهای هر یک مخصوص به خود و دنبال انگیزه های متفاوتی است.

۱۷.۴ الگوهای مرتبط با الگو

الگوی Adapter: این الگو Wrapper نیز هست که در نتیجه از این نظر با بقیه ی الگوهای Wrapper شباهت دارد. به همین منظور با الگوی Proxy به دلیل قرار دادن یک نماینده برای آبجکت وحدت معنایی دارد. این الگو از جهاتی می توان آن را شبیه به Facade نیز دانست؛ چرا که به نوعی درخواست کلاینت بیرونی را به طریقی که Adapter می داند به کلاس دیگری برای کارچرخانی محول می شود. کلاینتی که خود دیگر نمی داند چگونه با آن کلاس کار کند. در نهایت نیز الگوی Decorator هست که به دلیل تفاهم هر دو الگو در عدم تغییر واسط آبجکت در عین افزودن قابلیت یا استفاده در زمان اجرا، این الگوها نیز شبیه هستند.

الگوی Observer: یکی از بهترین ارتباطی که می توان برای این الگو با الگوی دیگری دانست، استفاده از آن در Mediator هست. دقت داریم که اشیاء همکار می توانند Observer ی باشد برای میانجیگر که نقش Subject را دارد. به همین مناسبت، می توان حساسیت خاصی را برای Observer ها جهت رصد و تاثیرپذیری به مجموعه اضافه کرد.

الگوی Strategy: تشابه در ساختار پایه برای این الگو و الگوی State بسیار زیاد است به گونه ای که تطابق کاملی از این نظر با یکدیگر دارند اما از نظر وحدت معنایی بسیار دور هستند.

مقایسه: الگوی Observer و Mediator توانایی ترکیب شدن خوبی دارند و دو الگوی Strategy و State تشابه ساختاری بالایی دارند. الگوی Adapter نیز با مجموعه الگوهای Wrapper ارتباط مفهومی نزدیکی دارد.

۱۸.۴ OCP

الگوی Adapter: اینکه کلاینت با فوق کلاس انتزاعی ای به نام Target در ارتباط است بدین معنا می باشد که هر کم و زیاد شدنی برای انواع Adapter از دید کلاینت مخفی است و منتزع از آن است که این رعایت OCP است. اما اگر به پیاده سازی سطح کلاس برویم، متوجه هستیم که با هر subclassing از Adaptee باید برای آن کلاس جدید یک Adapter تعریف کرد که این خود نشان از انتشار تغییرات دارد و در مقابل اصل OCP است. زیرا با گسترش کلاس، تغییرات بسته نیستند. در سطح آبجکت الگو باید خاطرنشان کرد که در حالت پایه، ارتباط Association مستقیم بین Adapter و Adaptee می تواند لطمه ای برای OCP باشد مگر آنکه، کلاس Adaptee را به شکل انتزاعی تبدیل کرده و برای آن انواع زیرکلاس قائل شویم تا OCP به خطر نیافتد.

الگوی Observer: ارتباط از Subject به Observer ها در سطح فوق کلاس های انتزاعی برای هر دو برقرار شده است که این خود نشان می دهد. درنتیجه توسعه پذیری برای Observer ها درواقع بسته بودن تغییرات بعدی را برای Subject بدنبال دارد. اما این رعایت، به خاطر رابطه ی سطح پایین از Ob-

server به Subject خدشه دار می شود و با توسعه پذیری Subject تغییرپذیری برای Observer الزامی است. پس OCP در این الگو نقض می شود.

الگوی Strategy: گسترش پذیری در زیرکلاس های استراتژی باعث بسته ماندن تغییرات روی Context می شود که دلیل آن رابطه ی سطح بالای Con-text با فوق کلاس انتزاعی Strategy است. این توضیح ما را به رعایت OCP در الگو می رساند.

مقایسه: الگوی Strategy این اصل را به درستی رعایت می کند اما الگوی Observer در برآورده کردن آن ناتوان است. الگوی Adapter نیز در سطح الگو این اصل را نقض می کند اما در سطح آبجکت می تواند این معیار را با subclassing از Adaptee برقرار کند؛ درغیر اینصورت این معیار برایش هنوز زیرسوال است.

۱۹.۴ LSP

الگوی Adapter: در سطح آبجکت از الگو LSP برقرار است؛ چرا که دقیقا Adapter می تواند جایگزین Target شده و همان خدمات را به کلاینت ارائه دهد. اما در سطح کلاس الگو، نمی تواند گفت که Adapter نوعی از Adaptee است. چرا که فقط متد SpecificRequest برای Adapter قابل دیدن هست و پدر از آن مخفی ندارد. پس در اینجا LSP نقض می شود.

الگوی Observer: در این الگو LSP برقرار است؛ زیرا هر کلاس فرزند چه از نوع Observer و چه از نوع Subject می تواند جای فوق کلاس خود جای بگیرد و همان رفتار موردانتظار را از خود نشان دهد.

الگوی Strategy: بسیار ساده می توان گفت که ارثبری انواع کلاس Con-creteStrategy از واسط Strategy نشان می دهد که رابطه ی is-a برقرار است و هر آبجکت استراتژی نماینده ای برای فوق کلاس انتزاعی خود است و عملکردهای موردانتظار از آن را برآورده می کند. این توضیحات نشان دهنده ی LSP است.

مقایسه: در الگوی Strategy اصل LSP برقرار است؛ در حالی که در الگوی

Observer نیز چنین است اما الگوی Adapter فقط در سطح آبجکت آن این اصل رعایت می شود و سطح کلاس الگو، این معیار را زیرپا می گذارد.

DIP ۲۰.۴

الگوی Adapter: ارتباط سطح بالا از کلاینت به Target نشان می دهد که DIP در این قسمت برقرار است و این در هر دو نوع الگو ثابت است که کلاینت از تغییر زیرکلاس ها منتزع می باشد. اما در سطح آبجکت از الگو، ارتباطی از Adapter به Adaptee وجود دارد که می تواند به دلیل رابطه ی مستقیم این دو کلاس، اصل DIP را زیرسوال ببرد. اگر این رابطه به شکل انتزاعی باشد و Adapter فوق کلاس همه ی Adaptee ها را بشناسد و با آن وابستگی داشته باشد می توان از این خطر گذشت.

الگوی Observer: قسمتی که DIP در این الگو برقرار است ارتباط سطح بالای کلاس های انتزاعی است اما وابستگی مستقیم سطح پایین کلاس های concrete قطعا اثبات می کند که هر تغییری در انواع Subject سبب می شود تا Observer ها دستخوش بشوند و این می تواند باعث نقض DIP شود.

الگوی Strategy: دقت داریم که وابستگی در سطح بالای بین Context و کلاس انتزاعی Strategy سبب برقراری DIP می شود. بدین گونه که Context از دانستن همه ی انواع استراتژی و آگاهی از هر توسعه ای در آن منتزع است. پس در مجموعه ی الگو این معیار به خوبی پشتیبانی شده است.

مقایسه: در مقایسه ذکر می کنیم که الگوی Strategy اصل DIP را به درستی برقرار کرده است ولی الگوی Observer در تحقق کامل آن ناقص عمل می کند. الگوی Adapter زمانی که در سطح کلاس هست، می تواند DIP را به همراه داشته باشد اما در سطح آبجکت از آن ناتوان است.

ISP ۲۱.۴

الگوی Adapter: با قرار گرفتن واسط Target کلاینت به ساختاری دسترسی دارد که می داند با آن چگونه کار کند و نیاز به دید بیشتر ندارد. این واسطی که به خوبی می تواند نیاز کاربر را در خود تعریف کند، باعث افزایش Cohesion است. پس می توان گفت ISP در ساده ترین شکل خودش در این الگو صادق است.

الگوی Observer: در این الگو مشاهده می شود که برای هم Observer ها و هم Subject ها یک Interface خوش‌ـتعریف گذاشته شده است که انسجام مجموعه را به شدت بالا برده است. می دانیم که استفاده ی بیشتر از چندین واسط برای همبستگی بهتر، رعایت اصل ISP است.

الگوی Strategy: معیار ISP در این الگو به سادگی با انسجام شکل گرفته با فراهم آمدن اشیاء متخصص الگوریتم های سفارشی از دل Context توسط کلاس انتزاعی سطح بالا به نام Strategy برقرار شده است.

مقایسه: در هر سه الگو معیار ISP با واسط های خوب تعریف شده، مهیا شده است.

CRP ۲۲.۴

الگوی Adapter: نقض CRP به وضوح در پیاده سازی الگو در سطح کلاس دیده می شود. وجود ساختار توارثی صلب که انعطاف پذیری زیادی را در زمان اجرا می گیرد. وراثت شیوه ای است که بر delegation ترجیح داده می شود. پس درنتیجه، الگو در سطح آبجکت خود CRP را برقرار کرده است.

الگوی Observer: این الگو برپایه ی واسپاری بنا شده است و اصلا کار زیادی نیز به ساختارهای توارثی ندارد. دو جنبه از انتزاع وجود دارد که آبجکت ها با پیغام ارسال به یکدیگر به واسطه ی دیدی که نسبت بهم دارند، الگو را محقق می کنند. پس کلا با رابطه های از نوع Association برای آبجکت ها در الگو، این معیار برای آن برقرار است.

الگوی Strategy: اصل این الگو آن بوده است که انواع مختلف Context را برای هر الگوریتم مبتنی بر حالت داده نداشته باشیم و کار با واسپاری به آبجکت متخصص انجام شود تا جلوی ساختار توارثی صلب غیرمنعطف گرفته شود. پس CRP در آن مشهود است.

مقایسه: تنها جایی که این معیار نقض می شود در الگوی Adapter هست زمانی که از سطح کلاس آن استفاده کنیم. در سطح آبجکت آن و دو الگوی دیگر اصل CRP برقرار است.

PLK ۲۳.۴

الگوی Adapter: هیچ دید ترایایی در الگو برقرار نمی شود. هیچ بخشی در الگو در هر دو سطح پیاده سازی وجود ندارد که خود شیء در پاسخ به یک درخواست بیرونی فرستاده شود. پس واضح است که PLK ثابت است.

الگوی Observer: در ساختار پایه ی الگو متوجه هستیم که امکان بوجود آمدن زنجیره دید ترایا در الگو وجود ندارد؛ زیرا هیچ قسمتی در آن نیست که بخواهد در پاسخ یک درخواست، خود شیء را برگرداند. در GetState نیز تنها قسمتی از حالت درونی داده که موردنیاز است ارسال می شود.

الگوی Strategy: زنجیره ی دید ترایا در این الگو دیده نمی شود و خود شیء در پاسخ به یک درخواست فرستاده نمی شود. پس PLK برای آن برقرار است.

مقایسه: هر سه الگو توانسته اند تا اصل PLK را پشتیبانی کنند.

۲۴.۴ الگوهای GRASP

۱.۲۴.۴ Information Expert

الگوی Adapter: خود Adapter خبره ی اطلاعاتی است که هم می داند باید درخواست را از کجا بگیرد و هم رفتار درستی را مورد انتظار کلاینت از خود بروز می دهد. در هر دو نوع پیاده سازی این الگو، آبجکت آداپتر به گونه ای عمل می کند که اطلاعات داده ای آن در همان جا گذاشته شده است. پس این معیار برای آن محقق است.

الگوی Observer: در این الگو هم Observer و هم Subject خبره های اطلاعاتی الگو هستند؛ بدین صورت که هر کدام داده ی مورد نظر خود را در کنار عملکردی می داند و تعاملاتی که باید برقرار کند bundle می کند.

الگوی Strategy: انواع آبجکت های از نوع Strategy در واقع متخصص الگوریتم هستند که رفتاری بروز می دهند که از داده ی خود ینی آبجکت Con-text دور مانده اند. لذا می توان گفت که این الگو با رعایت کپسوله سازی اما از خبره ی اطلاعاتی محروم هست. بدین معنا که آبجکت ها همچنان کپسول های داده_رفتاری هستند که داده در نزدیکی رفتار خود می باشد اما اینکه آبجکت استراتژی روی داده ی خود تعریف شده باشد تا از نظر اطلاعاتی نیز خبره باشد وجود ندارد. پس تا مقداری نقض در این الگو را مشاهده می کنیم.

۲.۲۴.۴ Creator

الگوی Adapter: در هر دو نوع پیاده سازی از این الگو، کلاینت در اولویت بالاتری برای ساخت اشیاء می باشد اما پیکربند با اولویت ۵ که کمترین اولویت است، اقدام به نمونه گیری از اشیاء می کند که باعث نقض Creator می شود.

الگوی Observer: پیکربند دارای اطلاعات اولیه برای نمونه گیری از اشیاء می باشد که اولویت ۵ را در این زمینه دارد. اما از طرفی این ساختن شیء باید توسط اشیاء موجود در سیستم ساخته شود که به خاطر رابطه ی Association اولویت بالاتری دارند که چون این امر محقق نمی شود، پس Creator برای این

الگو نقض می شود.

الگوی Strategy: نقض این معیار در این الگو بدین دلیل هست که پیکربند ثالث اشیاء را می سازد و در واقع آبجکت استراتژی را به Context معرفی می کند، در حالی که اولویت بالاتری در اولویت ۲ وجود دارد که درواقع خود Context با رابطه ی Aggregation ی که دارد باید این کار را انجام دهد و شیء استراتژی دلخواه را بسازد.

۳.۲۴.۴ Low Coupling

الگوی Adapter: این الگو در هر دو سطح خود، از طرف کلاینت بیرونی، وابستگی کمی را بوجود می آورد؛ چرا که کلاینت به واسط Target دید دارد. اما در سطح کلاس خود یکی از قوی ترین ارتباطات مانند ارث بری تعریف شده است که وابستگی را به شدت زیادی افزایش داده است. اما در سطح آبجکت الگو، به دلیل ارتباط Association از Adapter به Adaptee وابستگی به شدت قبل نیست و فقط در حد ارتباط مستقیم دو کلاس concrete هست که می تواند با تعریف فوق کلاس برای Adaptee به این موضوع کمک کرد. در آخر نیز به پیکربند اشاره می کنیم که وابستگی بالایی را برای مجموعه به همراه می آورد.

الگوی Observer: وابستگی در کل به خاطر رابطه ی محدود شده در سطح انتزاع از Subject به Observer ها کاهش یافته اما به دو دلیل این کاهش کمتر حس می شود. یک آنکه Observer ها دید سطح پایین و مستقیم را به Subject دارند که به شدت وابستگی را بوجود می آورد. دوم آنکه پیکربند نیز باید نسبت به همه ی کلاس ها آگاه بوده و با آنها در ارتباط باشد. نکته ی امیدوارکننده اما همان مورد اول هست که رابطه ی چند_به_چند را به رابطه ی یک_به_چند تبدیل می کند و با همین کار وابستگی را پایین می آورد.

الگوی Strategy: سطح بالا بودن ارتباط Aggregation از Context به Strategy نشان از وابستگی کم دارد. در نتیجه تغییراتی که در زیرکلاس ها اتفاق می افتد مادامی که در واسط Strategy تغییر رخ ندهد، منتشر به Context

نخواهد شد. تنها نکته برای وابستگی، پیکربند هست که Coupling را افزایش می دهد.

۴.۲۴.۴ High Cohesion

الگوی Adapter: همبستگی در این الگو در هر دو سطح با ورود Adapter بوجود می آید که قابلیت تبدیل واسط را در اختیار می گذارد و آن را از کلاسهایی که وظیفه شان نیست، خارج می کند. هم خودش منسجم است هم به منسجم بودن کل مجموعه کمک می کند. این شیء متخصص باعث می شود تا پیچیدگی برای دردسرهای کلاینت نیز کم بشود. در کل Cohesion خوبی در اثر این الگو بوجود می آید.

الگوی Observer: تعریف اشیاء متخصص و واسط آن ها به همراه تصریح ارتباطات جهت تعامل پویا و در زمان اجرا، به خودی خود انسجام بالایی را برای آبجکت ها بالا می رود و دغدغه های بی دلیل را از آن ها حذف می کند و دوش کارهای زائد را از آنان بر می دارد. در واقع Subject و Observer دو مفهوم می شوند که هر آبجکتی یک نقشی را بازی می کند که به آن cohesion بالایی می دهد.

الگوی Strategy: کما اینکه پیشتر بیان کردیم، با فراهم آمدن متخصصی جهت بروز الگوریتم مناسب با حالت، انسجام کل مجموعه بیشتر می شود که نشان از Cohesion بالا برای این الگوی ساده دارد.

۵.۲۴.۴ Controller

الگوی Adapter: در این الگو کلاسی برای بازی کردن نقش واسط یک مجموعه(کنترلر) وجود ندارد.

الگوی Observer: در این الگو کلاسی برای بازی کردن نقش واسط یک مجموعه(کنترلر) وجود ندارد. باید ذکر شود که Subject تنها خبررسانی تغییر را برعهده می گیرد و وظیفه ی کنترلی خاصی ندارد.

الگوی Strategy: در این الگو کلاسی برای بازی کردن نقش واسط یک مجموعه(کنترلر) وجود ندارد.

Polymorphism ۶.۲۴.۴

الگوی Adapter: هر یک از زیرکلاس های Target می تواند به کلاینت منتسب شود. کلاینت نیز می تواند با هر نوع Adapter کار کند. پس چندریختی در این الگو تحقق یافته است.

الگوی Observer: چندریختی در نوع خوب خود برای توسعه ی انواع Sub-ject ها و انواع Observer ها وجود دارد. به گونه ای که یک Subject هر آنچه شیء از نوع Observer را در خود ثبت یا حذف می کند و به آن خبررسانی می کند.

الگوی Strategy: چندریختی در شکل خوب خود در این الگو به کار گرفته شده است، بدین گونه که هر Context می تواند انواع مختلف آبجکت استراتژی را بپذیرد.

Indirection ۷.۲۴.۴

الگوی Adapter: ارتباط مستقیم به واسطه ی Target و شیء Adapter شکسته شده که وابستگی مستقیم از کلاینت به Adaptee را ازبین می برد. پس Indirection در این الگو محقق است.

الگوی Observer: به نظر نمیاید مکانیزم خاصی برای Indirection در این الگو وجود داشته باشد.

الگوی Strategy: در اینجا ارتباط از کلاینت بیرونی به یک استراتژی به شکل غیرمستقیم ایجاد شده است. یعنی آنکه Indirection از طریق Context ی که در وسط قرار می گیرد، مراجعه به عملکرد موجودیت داده ای استراتژی با واسطه انجام می شود. پس این الگو از این معیار بخوردار است.

۸.۲۴.۴ Pure Fabrication

الگوی Adapter: شیء جعلی خود آبجکت Adapter هست. چرا که منشاءیی از دنیای پیرامون مساله ندارد و در اثر اعمال الگو بوجود می آید. پس Fabri- cation Pure در این الگو صادق است.

الگوی Observer: موجودیت جعلی در این الگو همانطور که قبلا بیان شد در این الگو نمی توان متصور شد. پس این الگو فارغ از Pure Fabrication می باشد.

الگوی Strategy: موجودیت و ممفهوم استراتژی زاییده ی الگو می باشد و متناظر خارجی ندارد. این جعلی بودنش نشان از Pure Fabrication برای این الگو دارد.

۹.۲۴.۴ Protected Variations

الگوی Adapter: در این الگو، این معیار وجود دارد؛ بدین گونه که با خدمات تبدیل واسط به کلاینت و انواع Adapter ی که می تواند داشته باشد، تغییرات و گسترش زیرکلاس های Target را محافظت می کند.

الگوی Observer: محافظت از مواردی که متغیر هستند، در انواع Subject و Observer دیده می شود. ارتباط سطح بالای بین واسط ها این امکان را می دهد که Subject از گسترش پذیری Observer ها منتزع باشد. البته این مورد در رابطه ی معکوس صدق نمی کند. دقت داریم که اگر Subject ی کم یا زیاد شود باید Observer از آن مطلع شود.

الگوی Strategy: این الگو با تعریف یک واسط خوش‌ـتعریف به نام Strat- egy تعریف شده است که بدین وسیله به کمک subclassing می توان از تغییرات محافظت کرد. هر توسعه و تغییری در زیرکلاس های آن، تغییرات را حفاظت شده می کند.