الگوها در مهندسی نرم افزار

دکتر رامان رامسین

تمرین اول

مهدی حاجی‌محمدعلی - 402211615

بهار ۱۴۰۳

فهرست مطالب

مقایسهٔ الگوهای proxy، iterator و state over collection

دسته - نوع

Proxy

این الگو از الگوهای ساختاری می‌باشد. این دسته از الگوها بیشتر به ساختاردهی کلاس‌ها با هدف کاهش coupling و انعطاف‌پذیری بیشتر توجه دارند. این کار را معمولا با جداسازی abstraction از implementation و گسترش ساختار انجام می‌دهند. باید توجه کرد که proxy یکی از wrapperها است که همگی جزئی از الگوهای ساختاری هستند. هر چند ممکن است این الگوها وجوه رفتاری نیز داشته باشند (که در proxy نیز وجه رفتاری جایگزینی مشخص است) اما وجوه ساختاری در آن پررنگ‌تر است.

Iterator

این الگو از الگوهای رفتاری می‌باشد. این الگوها به رفتار اشیا در زمان اجرا توجه دارند و نحوه‌ی تعامل اشیا با یکدیگر تمرکز دارند. هر چند ممکن است وجوه ساختاری نیز داشته باشند اما این وجه در آن‌ها پررنگ‌تر است.
این الگوها، دغدغه‌ی تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار شی، اداره‌ی درخواست‌های به یک شی و همچنین مدیریت بهتر تعامل میان اشیا در زمان اجرا با هدف افزایش cohesion و کاهش coupling را دارند.

State/Behavior over collection

این الگو از الگوهای coad است. از نظر وجوه ساختاری، رفتاری و آفرینشی، می‌توان گفت الگوی state over collection الگوی بسیار ریزدانه و پیش پا افتاده‌ای است و تنها یک وجه ساختاری بسیار کوچک را مورد توجه قرار می‌دهد اما الگوی behavior over collection وجه رفتاری دارد و به تخصیص مسئولیت رفتار کل یک collection به کل آن به جای یک جز می‌پردازد و در نتیجه وجه رفتاری آن پررنگ است.

مقایسه

دو الگوی iterator و proxy از نوع الگوهای GOF هستند که iterator در دسته‌ی الگوهای رفتاری و proxy در دسته‌ی الگوهای ساختاری قرار می‌گیرد. اما الگوی state/behavior over collection از الگوهای ابتدایی coad است ولی می‌توان گفت که الگوی behavior over collection وجه رفتاری و الگوی state over collection وجه ساختاری پررنگ‌تری دارد.

هدف

Proxy

این الگو یکی از wrapperها است و با این ایده ایجاد شده تا یک شی، جایگزین یک شی دیگر شود و کلاینت هیچ تغییری میان آن دو احساس نکند. شی proxy می‌تواند فعالیت‌هایی را به جای شی اصلی انجام دهد. برای مثال می‌تواند رفتاری را پیش یا پس از یک درخواست از شی اصلی به آن اضافه کند و در صورت لزوم، شی اصلی را برای رفتارها به کار گیرد. در واقع یک سطح indirection اضافه می‌کند تا وظایف شی اصلی را در مواقع مورد نیاز برعهده بگیرد. هدف از این کار می‌تواند صرفه‌جویی در حافظه و ... باشد که بسته به نوع proxy به کار گرفته شده متفاوت است.

Iterator

هدف آن این است که یک عنصر مرکب را پیمایش کنیم بدون آن که از جزئیات ساختار داخلی آن با خبر باشیم و کپسوله‌سازی را رعایت کنیم. اگر پیمایش برعهده‌ی شی اصلی باشد، علاوه بر آنکه احتمال سنگین شدن شی و تبدیل آن به large class وجود دارد، امکان چند پیمایش بر روی شی را نخواهیم داشت. همچنین اگر وظیفه را به کلاینت بسپاریم، encapsulation نقض می‌شود. در نتیجه مطابق با هدف آن، با افزودن یک شی پیمایش‌گر که توسط کلاینت تخصیص داده می‌شود، می‌توانیم هدف را محقق کنیم.

State/Behavior over collection

هدف آن، جداسازی رفتار - حالت یک کل از رفتار - حالت اجزای آن است. در واقع در یک رابطه‌ی whole part که یک کل و تعدادی جز وجود دارد، ممکن است یک رفتار یا حالت منتسب به کل اجزا باشد اما این رفتار و حالت را به اشتباه در اجزا قرار داده باشیم. هدف این الگو، انتساب این رفتار - حالت به کل و جداسازی آن از اجزا است. البته در این شرایط باید رفتار - حالتی که مخصوص اجزا است حتما در آن‌ها باقی بماند و به کل اختصاص نیابد که خطر ایجاد god class را ممکن می‌سازد.

مقایسه

میان اهداف این سه الگو، شباهت چندانی وجود ندارد. هدف از پراکسی، تعیین یک نایب برای شی اصلی است تا بتواند به شی اصلی رفتار افزوده و یا دسترسی به شی اصلی را کنترل نماید. هدف iterator، پیمایش یک شی مرکب است بدون نقض encapsulation نسبت به کلاینت (هر چند در آن یک شی زائد مانند پراکسی ایجاد می‌شود) و همچنین هدف state/behavior over collection، جداسازی رفتار - حالت یک کل از اجزای آن می‌باشد.

حوزه

Proxy

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

Iterator

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

State/Behavior over collection

این الگو را می‌تواند در حوزه‌ی شی حساب کرد چرا که یک شی کل افزوده شده و رفتار - حالت به آن منتسب می‌شود و تحقق آن را می‌توان در زمان اجرا در نظر گرفت.

مقایسه

هر سه الگو در حوزه‌ی object هستند و در زمان اجرا محقق می‌شوند.

کاهش coupling

Proxy

Wrapperها به صورت کلی به دلیل آن که یک interface روی شی اصلی تعریف می‌کنند، از تغییرات منتشر شونده شی اصلی به کلاینت جلوگیری می‌کنند. هر چند proxy خود coupling بسیار بالایی با شی اصلی خواهد داشت و در نتیجه تغییرات بر روی شی اصلی می‌تواند بر پراکسی نیز تاثیرگذار باشد. اما نسبت به دید کلاینت، این coupling تقریبا تغییری نمی‌کند و یا می‌توان گفت کمی کمتر می‌شود. پیکربند نیز باید نسبت به شی proxy و شی اصلی دید داشته باشد و در نتیجه coupling آن نیز تغییری نمی‌کند و زیاد است. هر چند در الگوها برای ما coupling میان کلاینت و شی اصلی مهم است.

Iterator

در این الگو وابستگی میان شی اصلی و کلاینت کمتر می‌شود. چرا که در صورت نبود شی iterator، یا باید کلاینت نسبت به شی اصلی یک دید کامل داشته باشد و در نتیجه coupling بسیار بالا خواهد بود و یا شی باید بتواند امکان پیمایش را فراهم کند که شی را بیش از حد سنگین می‌کند.
اما نکته‌ای که باید در نظر گرفته شود، وابستگی شدید میان شی iterator و شی اصلی است. چرا که iterator وظیفه‌ای را برعهده می‌گیرد که در اصل بر عهده‌ی شی اصلی بوده و در نتیجه لازم است تا نسبت به آن دید کاملی داشته باشد و در نتیجه coupling میان iterator و شی اصلی بسیار زیاد خواهد بود. اما میان کلاینت و شی اصلی، coupling کاهش می‌یابد که این خود علتی برای پذیرفتن هزینه‌ی coupling میان پیمایش‌گر و شی اصلی می‌دهیم.

State/Behavior over collection

می‌توان گفت تا حدی coupling میان اشیا جز را کمتر می‌کند. در حالت پیشین امکان duplicate شدن در طراحی اجزا وجود داشت چرا که رفتارهای کل باید توسط آن‌ها محقق می‌شد و همگی لازم بود اطلاعات یکسانی را برای کل مجموعه نگهداری کنند. همچنین ممکن بود برای پاسخگویی نیازمند همکاری با دیگر اجزا شوند. اما پس از اعمال الگو، شی کل یک interface روی کل مجموعه محسوب می‌شود (البته منظر interface مفهومی است نه برنامه‌نویسی) و در نتیجه coupling را کاهش می‌دهد.

مقایسه

الگوی proxy تمرکز چندانی بر روی coupling ندارد هر چند به صورت غیر مستقیم، وابستگی را تا حدی کاهش می‌دهد. Iterator برای کاهش وابستگی میان کلاینت و شی مورد پیمایش بسیار واجب و لازم است و کمک زیادی به کاهش coupling میان کلاینت و شی اصلی می‌کند. الگوی آخر هم coupling را کاهش می‌دهد.

افزایش cohesion

Proxy

در برخی از انواع proxy می‌توان گفت که cohesion افزایش می‌یابد. برای مثال در remote proxy، وظیفه‌ی ارتباط با یک شی خارجی را به کمک یک نایب براورده می‌کنیم در نتیجه مسئولیت کارهای آن به یک شی نایب سپرده می‌شود و لازم نیست که کلاینت مستقیما وظیفه‌ی ارتباط با شی خارجی را برعهده بگیرد. یا برای مثال در protection proxy، وظیفه‌ی کنترل دسترسی به شی را به صورت اختصاصی به شی proxy محول می‌کنیم در نتیجه cohesion افزایش می‌یابد.

Iterator

یک شی متخصص پیمایش در این الگو ساخته می‌شود. اساسا می‌توان گفت که این الگو به افزایش cohesion با تخصیص رفتار به یک شی متخصص، کمک می‌کند.

State/Behavior over collection

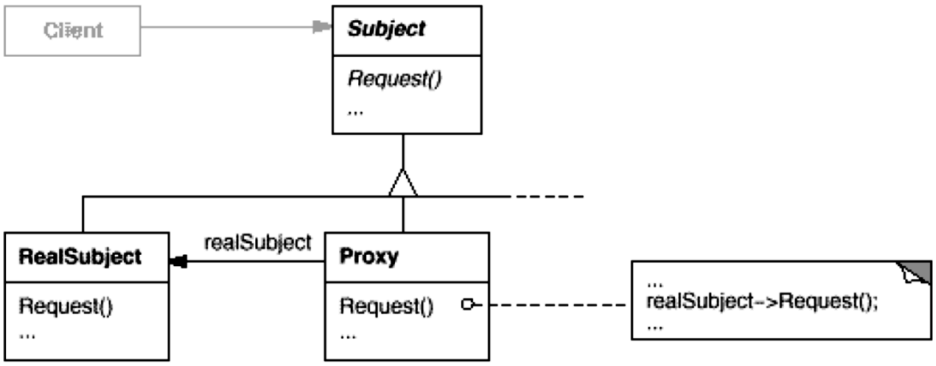
این الگو نیز به افزایش cohesion کمک می‌کند. چرا که وظیفه یا حالتی که اساسا مخصوص به یک کل است و به اجزا مرتبط نیست را از روی دوش اجزا برداشته و به کل محول می‌کند. در نتیجه از یک حالت با cohesion کمتر که در آن اجزا وظایف مربوط به کل را بر عهده داشتند به یک وضعیت جدید که یک کل وظایف مربوط به خود را برعهده دارد می‌رسیم و در نتیجه cohesion افزایش می‌یابد.

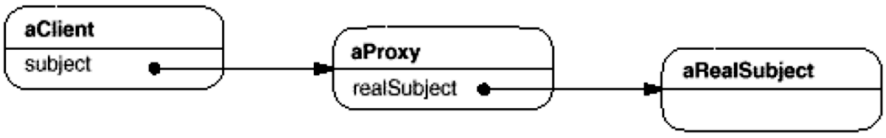
مقایسه

به صورت کلی می‌توان دید که cohesion در هر سه الگو افزایش می‌یابد با تخصیص وظایف به یک شی که متخصص آن کار است و جداسازی وظیفه‌مندی از شیئی دیگر تا آن را سبک‌تر نماید.

ساختار و رفتار

Proxy



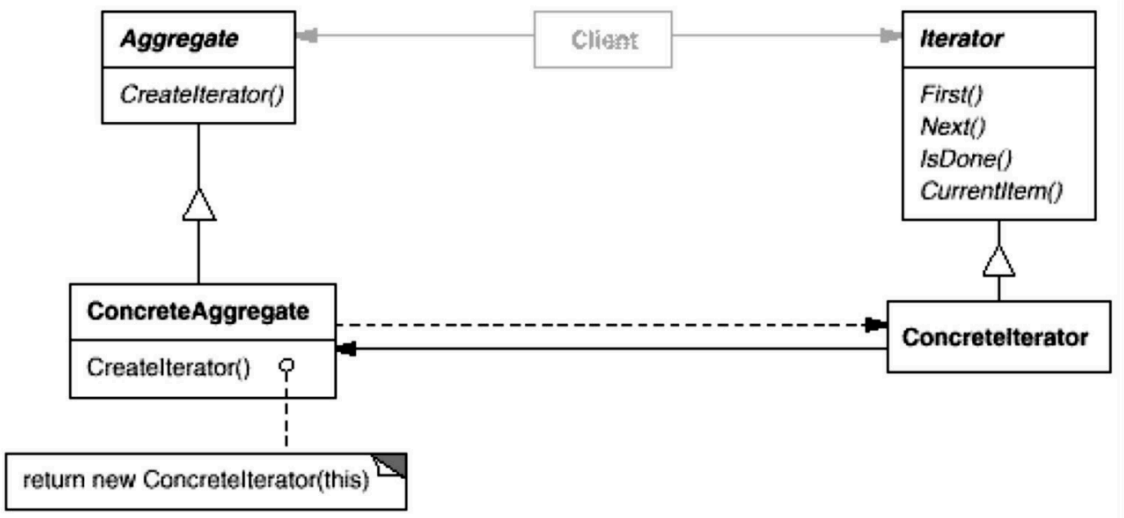


یک پراکسی همان interfaceای که شی اصلی (RealSubject) دارد (در اینجا Subject) را پیاده‌سازی می‌کند. در هنگام پیاده‌سازی یه سری کار اضافی نیز ممکن است انجام شود و پس از آن کار را ممکن است به شی اصلی واسپاری کند (مطابق شکل). یک رابطه‌ی association میان RealSubject و شی proxy به همین منظور برقرار است.

این RealSubject لزوما توسط خود پراکسی ساخته نمی‌شود و می‌تواند توسط یک پیکربند تخصیص یابد اما ممکن است ساخته شدن آن نیز به عهده خود شی باشد. برای مثال در virtual proxyها به دلیل مصرف کمتر منابع سیستمی، در آخرین زمان ممکن شی اصلی ساخته می‌شود. باید دقت کرد که برای نشان دادن نمودار virtual Proxy، باید در کنار رابطه‌ی association میان proxy و کلاس RealSubject، یک رابطه‌ی خط چین instantiation کشید چرا که توسط خود شی ساخته می‌شود. دلیل association هم مانا بودن دید میان پراکسی و RealSubject است.

در حالت کلی یک ثالث که پیکربند است که این ساختار اشیا را می‌سازد و از آن جایی که کلاینت تنها به Subject دید دارد و زیرکلاس‌های آن را مستقلا تشخیص نمی‌دهد، نمی‌تواند خود یک پیکربند باشد. این الگو همانطور که مشخص است در زمان اجرا محقق می‌شود و به اشیا وابسته است.

Iterator

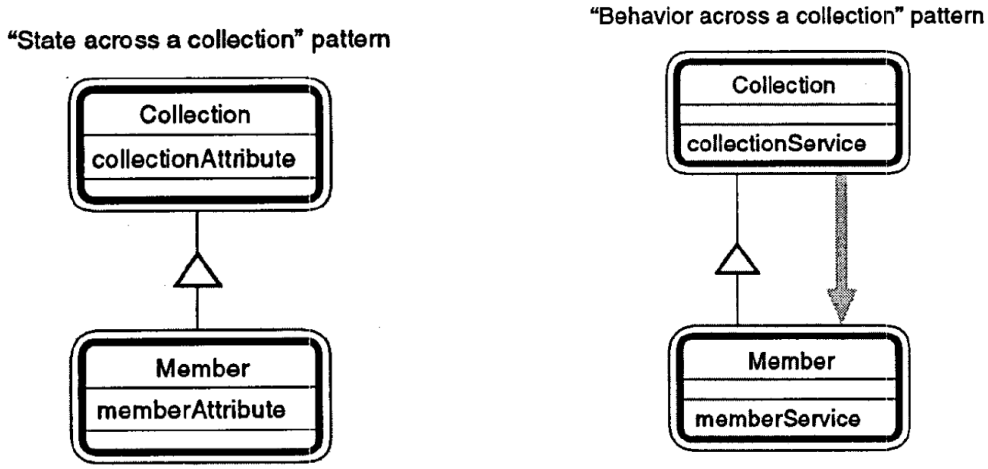


کلاینت یک دید سطح بالا به iterator و aggregate که شی مرکب است دارد. در واقع تنها interface آن‌ها را می‌بیند. به همین دلیل dip میان کلاینت و شی‌های iterator و aggregator کاملا برقرار است و coupling در وضعیت خوبی قرار دارد.

اما از آنجایی که لازم است شی مرکب خود پیمایش‌گر خود را در اختیار کلاینت قرار دهد، یک الگوی factory method در دل این الگو به کار می‌رود تا وظیفه‌ی آفرینش یک شی iterator به زیر کلاس‌های aggregate تخصص یابد. همانطور که در الگوی factory method، در ساختار توارثی میان دو شی، یک coupling بالا وجود دارد و dip نقض می‌شود، در اینجا نیز dip در ساختار توارثی iterator و aggregate نقض می‌شود.

وظیفه‌ی آفرینش ConcreteIterator با ConcreteAggregate است. به همین دلیل، یک رابطه‌ی association به همراه خط چین از ConcreteAggregate به ConcreteIterator وجود دارد که نشان‌دهنده‌ی این آفرینش است. در کنار آن، خود ConcreteAggregate به ConcreteIterator نیز پاس داده می‌شود تا رابطه‌ی association از ConcreteIterator به ConcreteAggregate با دید مانا برقرار شود. در نتیجه در این حالت، کلاینت از طریق Iterator می‌تواند پیمایش روی شی مرکب داشته باشد بدون آن که دید به اجزای داخلی آن‌ها داشته باشد و encapsulation نقض شود. همانطور که مشخص است این الگو در زمان اجرا محقق می‌شود.

State/Behavior over collection



همانطور که در شکل می‌بینید، یک رابطه‌ی aggregation که در اینجا Whole Part نامیده می‌شود میان Collection و Member وجود دارد. هر چند اینجا از notation مربوط به Coad استفاده شده است.
وظایف و attributeهای مربوط به کل، همانطور که در ساختار دیده می‌شود، نباید در خود اجزا قرار گیرد بلکه باید به کل نسبت داده شده و از اجزا جدا شود.

مقایسه

در proxy، به کمک یک شی که عینا interface شی اصلی را پیاده‌سازی می‌کند، شی اصلی را جعل می‌کنیم تا به عنوان یک نایب، جایگزین آن شده و رفتارهای بیشتری را به آن بیفزاییم. در iterator به کمک یک ساختار توارثی موازی با شی اصلی، یک پیمایش‌گر متخصص تعریف کرده ایم تا کلاینت دید مستقیم به جزئیات شی مرکب نداشته باشد. در این دو الگو، می‌توان وابستگی شدید میان شی جدید ایجاد شده (پراکسی و iterator) با شی اصلی پیدا کرد که coupling بالایی دارند. همچنین می‌توان دید که کلاینت دید مستقیم به concreteها نداشته و از طریق interface با شی اصلی و اشیای جدید ارتباط برقرار می‌کند و encapsulation نسبت به کلاینت کاملا رعایت می‌شود.
در state/behavior over collection، رفتار و حالت به شی کل نسبت داده شده و از شی اصلی گرفته می‌شود. گاهی ممکن است در وضعیت پیش از اعمال الگو، شی کل وجود نداشته باشد و به صورت static در شی جز پیاده‌سازی انجام شده باشد و در نتیجه‌ی اعمال این الگو، شی کل را یک شی جدید در نظر گرفت که رفتارهای شی جز را جداسازی کرده است و از این جهت شباهتی به الگوهای پیشین دارد.

پیکربندی

Proxy

همانطور که در بخش قبل توضیح داده شد، پراکسی‌ها به صورت معمول توسط یک پیکربند پیکربندی شده و هر سه شی کلاینت، شی اصلی و پراکسی و روابط بین آن‌ها را پیکربندی می‌کنند. اما در virtual proxy، شی اصلی در دیرترین زمان ممکن و توسط خود شی پراکسی instantiate می‌شود.

Iterator

این الگو نیاز به پیکربندی ندارد. چرا که کلاینت دید به interfaceهای شی مرکب و شی پیمایش‌گر دارد و از طریق interface شی مرکب و الگوی factory method موجود در آن، یک شی iterator مختص شی concrete را بسازد و بدون دید مستقیم به زیرکلاس‌ها، پیمایش را انجام دهد.

State/Behavior over collection

پیکربندی به صورت جدی وجود ندارد و تنها می‌توان حالتی را متصور شد که شی جز به شی کل منتسب شود.

مقایسه

می‌توان گفت تنها سه نوع از انواع proxyها نیاز به پیکربندی به کمک یک شی ثالث دارد و دو الگوی دیگر و virtual proxy نیازی به پیکربند ندارند.

انعطاف‌پذیری

Proxy

این الگو انعطاف پذیری مناسبی دارد. می‌توان با افزودن زیرکلاس‌های جدید به interface شی اصلی، پراکسی‌های متنوعی تعریف کرد که هر کدام به کمک یک شی ثالث، در شرایط مورد نیاز پیکربندی می‌شوند. همچنین یک سطح indirection در ارتباط میان کلاینت و شی اصلی ایجاد می‌شود که تغییرات در شی اصلی را بیش از پیش منتزع می‌کند (هر چند پیش‌تر نیز کلاینت از طریق interface با شی اصلی در ارتباط بوده است). از طرفی با توجه به هدف این الگو می‌توان گفت بدون تغییر در interface می‌توانیم رفتارهای جدیدی را به شی اصلی اضافه کنیم بدون آن که آن را تغییر دهیم که خود انعطاف بسیار مناسبی به سیستم می‌دهد.

Iterator

پس از اعمال این الگو انعطاف‌پذیری افزایش می‌یابد. در حالت پیشین، ممکن است کلاینت دید مستقیم به شی مرکب داشته باشد که در نتیجه encapsulation نقض شده و تغییرات در شی مرکب، ممکن است منتشر شود و در نتیجه تغییر به راحتی ممکن نیست و انعطاف‌پذیری کم است. همچنین ممکن است در حالت پیشین، وظایف مربوط به iteration برعهده‌ی خود شی مرکب باشد که این مورد موجب می‌شود که تنوانیم به راحتی پیمایش‌های جدید تعریف کنیم و انعطاف پذیری کم است. علاوه بر آن، تنها یک پیمایش در آن واحد می‌توان متصور بود و در نتیجه ساختار پیش از الگو، ساختاری با امکان تغییر کم و انعطاف ناپذیر است.
پس از اعمال الگو، کلاینت تنها از طریق interface به شی مرکب و همچنین شی پیمایش‌گر دید دارد و در نتیجه، از تغییرات داخلی شی مرکب یا پیمایش‌گر منتزع می‌شود و تغییرات منتقل نمی‌شوند. همچنین امکان افزودن پیمایش‌گرهای جدید، امکان به دست آوردن چند پیمایش‌گر و ... در این ساختار وجود دارد و به راحتی قابل تغییر و انعطاف‌پذیر است.
تنها اشکال جدی این الگو، وابستگی یا coupling بالا میان ساختار توارثی iterator و aggregate است که به علت نقض dip و وابستگی میان concreteها است. این مورد منجر به انتشار تغییرات در سطح زیرکلاس‌های aggregate و iterator می‌شود و انعطاف‌پذیری را در این سطح تا حدی کم می‌کند. اما انعطاف‌پذیری نسب به دید کلاینت بسیار بالا و مناسب است.

State/Behavior over collection

انعطاف‌پذیری در این الگو چندان مطرح نیست. چرا که سطوح انتزاع جدیدی نسبت به حالت پیشین با هدف عدم انتشار تغییرات دیده نمی‌شود. به خصوص که این الگو ساختار بسیار ساده‌ای دارد و انعطاف‌پذیری مورد بحث این الگو نیست.

مقایسه

دو الگوی proxy و iterator انعطاف‌پذیری را افزایش می‌دهند. Proxy با افزودن رفتار و همچنین امکان افزودن proxyهای مختلف، به خوبی امکان بازپیکربندی و همچنین بروز رفتارهای مختلف بدون تغییر در کلاینت را به ما می‌دهد و iterator نیز کلاینت را از دید مستقیم به شی مرکب باز داشته و امکان تعریف پیمایش‌های مختلف روی شی مرکب را فراهم می‌کند. اما در الگوی state/behavior over collection، این مورد مد نظر نیست.

کارایی

Proxy

در این الگو، کارایی کاهش می‌یابد چرا که ارتباط میان کلاینت و شی اصلی توسط proxy مدیریت شده است. در نتیجه برخی از رفتارهایی که بایستی توسط شی اصلی انجام شود (و تنها توسط proxy انجام نمی‌گیرد)، با یک سطح مستقیم و انتقال پیام زائد توسط شی اصلی انجام می‌شود. در نتیجه تا حدی کارایی فدای تحقق اهداف شده است.
البته شایان ذکر است که در virtual proxy، هدف تعریف proxy این است که اشیایی که فشار زیادی به سیستم می‌آورند را در آخرین لحظه‌ی ممکن دسترسی به آن ایجاد کنیم.

Iterator

در این الگو نیز کلاینت برای پیمایش به جای استفاده از رفتار aggregate به صورت مستقیم، از طریق iterator و به طور غیر مستقیم پیمایش را انجام می‌دهد و در نتیجه به صورت غیر مستقیم به آن دسترسی پیدا می‌کند. به همین منظور اطلاعات مورد نیاز iterator نیز در دسترسی مستقیم آن نیست و لازم به دسترسی به aggregate است در نتیجه از نظر performance عملکرد را نسبت به حالت پیشین بدتر می‌کند.

State/Behavior over collection

بستگی به نحوه‌ی طراحی آن دارد. اگر شی کل برای بروز دادن رفتار به تک تک اعضای مجموعه نیاز داشته باشد، در نتیجه یک سطح دسترسی غیر مستقیم ایجاد شده و performance کاهش می‌یابد. اما اگر اطلاعات را در خود نگاه دارد، در آن صورت به طور مستقیم پاسخگوی کلاینت خواهد بود. در نتیجه بستگی به پیاده‌سازی دارد.

مقایسه

هر دو الگوی iterator و proxy یک سطح دسترسی غیر مستقیم ایجاد کرده و کارایی را کاهش می‌دهند. در virtual proxy البته شاید عملکرد آن بهتر شود. در State/Behavior over collection می‌توان گفت بسته به طراحی دارد.

عواقب و تبعات

Proxy

اثرات مثبت

یک سطح indirection در ارتباط میان کلاینت و شی اصلی ایجاد می‌کنیم که در نتیجه‌ی آن، پراکسی می‌تواند کارهای مختلفی کند و رفتارهای بیشتری را اضافه کند. در ادامه
مثلا در یک remoteProxy، این که شی اصلی در جای دیگری قرار دارد را از کلاینت‌ها پنهان می‌کند و کارهای مربوط به ارتباط با شی اصلی را بر عهده می‌گیرد.
یک vitualproxy، یک شی را بر حسب لزوم در دیرترین زمان لازم می‌سازد و در نتیجه مصرف حافظه و لود سیستم را مدیریت می‌کند.
Protection و smart reference نیز کارهایی مانند کنترل دسترسی، لود شدن شی و همچنین حضور شی در حافظه و کارهای اینچنینی مربوط به کنترل شی را انجام می‌دهند.

اثرات منفی

Indirection ایجاد شده منجر به کندی و پایین آمدن performance می‌شود. هرچند که چشم‌گیر نیست و در ازای رفتارهای جدیدی که به دست می‌آوریم، قابل چشم‌پوشی است.

Iterator

اثرات مثبت

می‌شود انواع پیمایش برای یک شی مرکب تعریف کنیم. مثل مثال درخت که می‌توان روی آن پیمایش ،preorder postorder و inorder را تعریف کرد.
شی مرکب ساده می‌شود چرا که پیاده‌سازی پیمایش داخل interface خود شی قرار نمی‌گیرد و سبک می‌شوند.
بیش از یک پیمایش فعال می‌شود روی یک شی مرکب تعریف کرد.

اثرات منفی

بین iterator کی coupling شدید وجود دارد و encapsulation نقض می‌شود. دقیقا مانند الگوی factoryMethod اشکال نقض dip در concrete به concrete نیز وجود دارد.

State/Behavior over collection

اثرات مثبت

این الگو منجر می‌شود که رفتار و حالتی که بر عهده‌ی جز نیست، از آن مجزا شود و به درستی تخصیص وظایف صورت گیرد.

اثرات منفی

اثرات منفی چندانی دیده نمی‌شود خصوصا که این الگوها نسبتا الگوهای ریزدانه‌ای هستند.

مقایسه

الگوی state/behavior over collection اثرات منفی چندانی ندارد و بسیار الگوی مناسب و رایجی است. الگوی proxy نیز اثر منفی performanceی قابل چشم پوشی دارد که تاثیر منفی چندانی ندارد. الگوی iterator در صورتی که ساختارهای توارثی بزرگ شوند، به دلیل به وجود آمدن زیرکلاس‌های متعدد و افزایش سطوح ساختار توارث و ارتباط تنگاتنگ میان دو ساختار، می‌تواند مشکلاتی را بوجود بیاورد. اما در صورتی که چنین مشکلی وجود نداشته باشد، نقض dip میان دو ساختار را به بهای به دست آوردن مزایای ذکر شده می‌توان بخشید.

Encapsulation

Proxy

دید کلاینت به شی اصلی و همچنین proxy از طریق یک interface که شی اصلی و proxyها آن را پیاده‌سازی می‌کنند، محقق می‌شود. در نتیجه encapsulation رعایت شده است. هر چند این الگو در حالت پس از اعمال خود، چیز جدیدی برای تحقق encapsulation ارائه نمی‌کند اما آن را نقض نیز نمی‌کند و به خوبی رعایت می‌شود.

Iterator

یکی از اهداف این الگو، تحقق encapsulation است. یکی از حالت‌های ممکن پیش از اعمال الگو، محول کردن وظیفه‌ی پیمایش به کلاینت است که در این صورت دید مستقیم میان کلاینت و شی مرکب را لازم می‌دارد.
پس از اعمال الگو، دید کلاینت تنها از طریق interfaceهای aggregate و iterator به پیمایش‌گر و شی مرکب وجود دارد و در نتیجه encapsulation نه تنها رعایت می‌شود که یکی از اهداف این الگو می‌باشد و در حالت پسین نسبت به حالت پیش از اعمال الگو، امکان بهبود encapsulation دیده می‌شود.
هر چند دید میان ConcreteIteratorها و ConcreteAggregateها به صورت مستقیم است و encapsulation نقض می‌شود.

State/Behavior over collection

در این الگو نیز رفتار یک کل از اجزا جداسازی می‌شود و encapsulation براورده می‌شود و کلاینت برای رفتار - حالت مورد نظر، از طریق کل درخواست را ارائه می‌دهد.

مقایسه

هر سه الگو encapsulation را رعایت می‌کنند. الگوی iterator به صورت خاص، خود می‌تواند منجر به بهبود encapsulation شود و وضعیت پیشین نامناسبی برای آن می‌توان متصور بود که در نتیجه‌ی اعمال الگو، encapsulation در بالاترین حال خود محقق شود. هر چند encapsulation از دید اشیای ساختارهای توارثی برقرار نیست اما نسبت به کلاینت کاملا برقرار است.

انتشار تغییرات

Proxy

کلاینت از طریق interface از شی اصلی و proxy منتزع شده است و در نتیجه، تغییرات در پراکسی و شی اصلی به کلاینت منتقل نمی‌شود.
هر چند به دلیل وابستگی و دید میان proxy و شی اصلی، تغییرات در شی اصلی می‌تواند به proxyها منتشر شود.

Iterator

در این الگو، کلاینت از طریق interfaceهای aggregate و iterator با اشیا ارتباط برقرار می‌کند و در نتیجه نسبت به تغییرات در اشیا concrete پیمایش‌گر و شی مرکب، مصون است و تغییرات در پیمایش‌گر و شی مرکب به کلاینت تسری پیدا نمی‌کند.
اما ساختارهای توارثی iterator و aggregator با یکدیگر coupling شدید دارند و concreteها یکدیگر را می‌بینند و در نتیجه تغییرات میان آن‌ها منتشر می‌شود.

State/Behavior over collection

فرض کنید یک کل داریم که از اجزای متنوعی تشکیل شده است. در این صورت تغییر در رفتار کل‌نگر هر یک از اجزا، در صورتی که ویژگی و رفتار به اجزا تخصیص داده شده بود، به دیگر اشیا منتشر می‌شود. اما پس از اعمال الگو، تغییرات در وجه کل‌نگر منتشر نمی‌شود چرا که در شی collection اعمال شده است و به اجزا مرتبط نمی‌شود.

مقایسه

الگوهای iterator و proxy نسبت به کلاینت، تغییرات را منتشر نمی‌کنند چرا که دید کلاینت به اشیا، از طریق interface محقق می‌شود. اما در این الگوها میان اشیای محقق کننده‌ی الگو مانند ساختارهای توارثی و یا proxyها با اشیای اصلی و شی مرکب، coupling وجود دارد و تغییرات منتشر می‌شود. از این منظر وضعیت iterator در صورت پیچیده شدن ساختارهای توارثی بسیار بد خواهد شد.
در الگوی state/behavior over collection اما وضعیت بسیار مناسب است و تغییرات منتشر نمی‌شوند.

میزان استفاده از منابع سیستمی

Proxy

در این الگو یک شی اضافی (proxy) ایجاد شده است که در حالت پیشین ندارد. همچنین ممکن است رفتارهای اضافی نسبت به حالت پیشین اضافه شده باشد و در نتیجه منابع بیشتری نسبت به حالت قبل مصرف می‌کند. اما در virtual proxy، هدف از استفاده از الگو کاهش استفاده از منابع سیستمی و صرفه جویی آن است. در واقع شی سنگین در دیرترین حالت ممکن load می‌شود و در نتیجه منابع بهتر از قبل مصرف می‌شود. در نتیجه در virtual proxy منابع سیستمی بهینه شده است و در باقی، این گونه نیست.

Iterator

در این الگو، رفتار پیمایش به iterator محول شده است. در نتیجه یک ساختار توارثی موازی با aggregate ایجاد می‌شود که ممکن است تعداد اشیا را بسیار زیاد کند. در واقع به ازای هر aggregate یک یا بیشتر شی iterator ایجاد می‌شود در نتیجه منابع سیستمی را بهبود می‌یابد. در رابطه با ارسال پیغام به صورت غیر مستقیم نیز پیش‌تر در بخش کارایی نیز صحبت کردیم. در نتیجه در مجموع حافظه‌ی بیشتری نسبت به حالت پیشین استفاده می‌کند.

State/Behavior over collection

در حالت پیشین، ممکن بود مجبور شویم اطلاعات و رفتارهای یکسان را در اشیا جز مجموعه نگهداری کنیم. اما در حالت پسین، این مجموعه رفتار و اطلاعات در شی کل ذخیره می‌شود و در نتیجه احتمالا وضعیت پسین بهتر از وضعیت پیشین در زمینه‌ی مصرف حافظه است.

مقایسه

در دو الگوی iterator و proxy، شی جدید تعریف شده است که سربار حافظه و cpu دارد. البته این مورد درباره virtual proxy همانطور که توضیح داده شد برقرار نیست. در الگوی آخر نیز بسته به طراحی، به نظر می‌رسد که منابع سیستمی بهتر مصرف می‌شود.

استفاده از شی جعلی

Proxy

Wrapperها به صورت کلی و این الگو به عنوان یکی از wrapperها، همگی به شی جعلی برای تحقق الگو نیاز دارند. این الگو، با افزودن یک یا چند شی نایب یا جانشین، شی اصلی را جعل می‌کند و برای این کار، interface شی اصلی را پیاده‌سازی می‌کنند تا کلاینت تفاوت آن‌ها را درک نکند. در نتیجه، شی جعلی جایگزین شی اصلی می‌شود تا رفتارهایی به آن اضافه کند و یا کنترل و مدیریت دسترسی انجام دهد و یا به اهداف دیگری دست یابد. در نتیجه، اساس این الگو ساختن یک شی جعلی است که در دامنه‌ی problem وجود ندارد.

Iterator

این الگو نیز با افزودن interface پیمایش‌گر و زیرکلاس‌های آن، اشیا جعلی را به سیستم اضافه می‌کند. علت جعلی بودن آن این است که این اشیا در دامنه‌ی مسئله (problem domain) وجود نداشتند و برای افزودن encapsulation و در مرحله‌ی طراحی، افزوده شده اند. وظیفه‌ی پیمایش یا باید برعهده‌ی شی مرکب باشد و یا می‌توان آن را به کلاینت تخصیص داد اما برای جلوگیری از large class و یا نقض encapsulation، شی جعلی پیمایش‌گر ایجاد می‌شود.

State/Behavior over collection

در این الگو هیچ شی جعلی وجود ندارد. چرا که یک رابطه‌ی aggregation وجود دارد که یک کل از اجزا تشکیل شده است و این رابطه یک رابطه‌ی موجود در دامنه‌ی مسئله است و در نتیجه، هیچ شیئی جعل نمی‌شود. در واقع رابطه‌ی aggregation در صورت تحقق یکی از سه شرط assembly، containment و یا membership پدید می‌آید که در هر یک از این سه نوع، کل و جز هر دو در دامنه‌ی مسئله قابل دریافت هستند.

مقایسه

الگوی state/behavior over collection شی جعلی ندارد اما دو الگوی iterator و proxy، هر یک با افزودن شی یا اشیا جعلی، الگو را محقق می‌کنند تا به هدف الگو برسند.

سادگی پیاده‌سازی

Proxy

این الگو پیاده‌سازی‌های مختلفی دارد. در واقع بسته به نوع پراکسی و کارکردی که از شی می‌خواهیم، می‌توان این الگو را به شکل متفاوتی پیاده‌سازی کرد.
از منظر سادگی، پراکسی نیز پیچیدگی ندارد و نکته‌ی مهم، پیاده‌سازی اینترفیس مشترک با شی اصلی می‌باشد تا بتوان رفتار آن را جعل کرد.

Iterator

پیاده‌سازی آن بسیار ساده است و در زبان‌های شی گرا کاملا در دسترس است. در جاوا یک اینترفیس Iterator تعریف شده است که می‌توان آن را پیاده‌سازی کرد. پیاده‌سازی‌های مختلف collection نیز برای خود iterator را پیاده‌سازی کرده اند.
از منظر شی گرایی، پیاده‌سازی آن پیچیده نیست و تنها مشکل ممکن آن، ساختارهای توارثی است که ممکن است ایجاد شود.

State/Behavior over collection

این الگوها بسیار ساده هستند. هر یک کافی است که method یا attibuteهای کلاس‌های جز به کلاس‌های کل منتقل شوند و در نتیجه پیچیدگی ندارد.

مقایسه

هر سه الگو پیاده‌سازی نسبتا ساده‌ای دارند. ساده‌ترین الگو از این منظر، State/Behavior over collection است که پیاده‌سازی بسیار ساده‌ای دارد. در این الگو شی جدیدی ساخته نمی‌شود. در میان iterator و proxy، می‌توان گفت که در هر دو پیاده‌سازی، کلاس و اشیا جدیدی ساخته می‌شود. هر چند پیاده‌سازی iterator می‌تواند کمی پیچیده‌تر از proxy باشد چرا که یک ساختار توارثی موازی ایجاد می‌شود و از یک شی باید به شی اصلی دید داشته باشیم و رفتارهای مربوطه را بروز دهیم.

موارد کاربرد

Proxy

بستگی به نوع پیاده‌سازی دارد. اما به طور کلی زمانی به کار می‌رود که می‌خواهیم یک شی جعلی به جای شی اصلی بگذاریم تا رفتار را تغییر دهیم. حال به تفکیک نوع هر کدام را توضیح می‌دهیم:

- **Remote proxy**: وقتی که شی اصلی ما یک شی ریموت است (local نیست) و در فضای local یک نایب برای آن در نظر می‌گیریم و کلاینت‌ها فکر می‌کنند یک شی اصلی است. ملاحظات مربوط به ارتباط برقرار کردن با شی خارجی با proxy خواهد بود.
- **Virtual proxy**: اشیایی که فشار می‌آورند به منابع سیستمی و منابع زیادی مصرف می‌کنند را در آخرین لحظه‌ی ممکن می‌سازیم. (lazy intialization)
- **Protection proxy**: وقتی که می‌خواهیم به کمک یک شی جعلی، دسترسی را به یک شی اصلی کنترل کنیم. (access contorl)
- **Smart reference:** یک reference به شی که از یک پوینتر ساده کارهای بیشتری انجام می‌دهد. برای مثال reference counting می‌کند و در صورتی که شی اصلی رفرنس نداشته باشد آن را حذف می‌کند. lock می‌کند. وقتی که یک ابجکتی که نیاز به lock دارد، این فعالیت را انجام می‌دهد.

Iterator

- محتویات داخلی یک شی مرکب دسترسی و پیمایش داشته باشیم ولی نمی‌خواهیم encapsulation برای کلاینت نقض شود.
- برای اینکه بتوانیم چند پیمایش بر روی یک شی مرکب انجام دهیم.
- برای اینکه ساختارهای aggregate متفاوت رو با یک interface واحد مورد پیمایش قرار دهیم. استفاده از این الگو باعث استاندارد شدن این interface نیز شده است.

State/Behavior over collection

هنگامی که رابطه‌ای از نوع aggregation و به صورت کل به جز وجود داشته باشد و بتوانیم رفتار/حالت یا ویژگی‌ای را به کل اجزا نسبت دهیم. در این صورت رفتار/حالت را از اشیا جز خارج کرده و به کل منتقل می‌کنیم.

مقایسه

کاربردهای این سه الگو شباهتی به یکدیگر ندارند و هر یک کاربرد خاصی را دنبال می‌کنند.

الگوهای مرتبط

Proxy

برخی الگوهای GOV به صورت مستقیم به proxy مرتبط می‌شوند. در نتیجه در خارج از الگوهای gof نیز کاربردی است.
همچنین با الگوهای adapter decorator facade به علت wrapper بودن در ارتباط است.
در proxy می‌خواهیم یک شی جعلی را به جای شی اصلی قرار دهیم تا رفتاری را به آن بیافزاییم.
در facade، هدف پنهان کردن پیچیدگی‌های رفتار یک زیر سیستم، یک اینترفیس روی آن تعریف می‌کنیم و زیر سیستم را به کمک یک شی جعلی از کلاینت جدا می‌کنیم. از این جهت متفاوتند که proxy، واسط یکسانی با شی هدف دارد.
در decorator، هدف ایجاد یک ساختار بسیار منعطف و قابل پیکربندی برای افزودن رفتار به یک شی اصلی است. مانند proxy شی جعلی یک interface یکسان با شی اصلی را پیاده‌سازی می‌کند.
در adapter، دغدغه‌ی اصلی چیز متفاوتی است و هدف تبدیل میان interfaceها است. اما کلاینت از تغییر منتزع می‌شود و ارتباط میان آن و واسط هدف توسط adapter انجام می‌شود.

Iterator

یک الگوی مرتبط، factory method است که می‌تواند داخل aggregate به کار رود تا iterator مناسب را در اختیار ما بگذارد.
یک الگوی مرتبط دیگر، memento است. به کمک آن می‌توان وضعیت iteration را نگهداری کرد و پس از اعمال آن، بخشی از پیمایش را undo کرد.
همچنین visitor نیز شباهت جدی دارد. Visitor روی یک مجموعه‌ی اشیا و پیمایش آن، عملیاتی را تعریف می‌کند. در نتیجه در کنار پیمایش‌گر استفاده شود تا عملیات روی اشیا انجام شود.

State/Behavior over collection

این الگو بسیار ریزدانه است و می‌توان گفت تا حدی شباهتی به باقی الگوها ندارد.

مقایسه

میان این سه الگو شباهت چندانی دیده نمی‌شود. همچنین برای هر یک، الگوهای مرتبط را گفتیم که البته میان آن‌ها نیز اشتراک وجود ندارد.

OCP

Proxy

می‌توان گفت ocp نسبت به کلاینت برقرار است چرا که dip از کلاینت به ساختار شی اصلی و proxy برقرار است. در واقع کلاینت تنها یک interface می‌بیند که proxy و شی اصلی هر دو آن را پیاده‌سازی می‌کنند. در نتیجه‌ی آن، تغییرات در proxy و شی اصلی، به کلاینت منتشر نمی‌شود و در نتیجه امکان گسترش ساختار کلاینت و ساختار proxy و شی اصلی وجود دارد.
اما نکته‌ی قابل توجه، وابستگی شدید proxy به شی اصلی است. همانطور که قبلا هم اشاره کردیم، coupling میان این دو بسیار بالا است و در نتیجه به علت آنکه دید concrete از proxy به شی اصلی وجود دارد، dip برقرار نیست و تغییرات و extension شی اصلی، بر روی ساختار proxy اثر می‌گذارد.
اما شی اصلی می‌تواند بدون توجه به تغییرات کلاینت و proxy، رشد کند.

Iterator

در این الگو، ocp برای کلاینت نسبت به ساختار iterator و aggregator برقرار است. چرا که dip میان کلاینت و این دو ساختار برقرار است و در نتیجه تغییرات در ConcreteIteratorها و ConcreteAggregateها، تاثیری بر روی کلاینت نمی‌گذارند و در نتیجه گسترش ساختار توارثی آن‌ها منجر به تغییر در کلاینت نمی‌شود.
اما میان دو ساختار توارثی aggregator و iterator، وابستگی شدید است به گونه‌ای که دید concrete به concrete وجود دارد. در نتیجه گسترش و تغییر در هر یک از زیرکلاس‌های یک ساختار، منجر به تغییر در ساختار دیگر می‌شود و در نتیجه ocp میان این دو ساختار برقرار نیست.

State/Behavior over collection

به گونه‌ای می‌توان گفت ocp برقرار نیست. افزودن یک کلاس جز به مجموعه یا تغییر در یکی از اجزا، ممکن است منجر به بررسی و تغییر در کد کلاس کل شود و در نتیجه تغییرات ممکن است منتشر شود. یا ممکن است افزودن یک رفتار منجر به نیاز به افزودن رفتار به کلاس کل شود و در نتیجه به طور کلی می‌توان گفت این الگو در حدی ocp را نقض می‌کند.

مقایسه

Iterator و proxy از این منظر به یکدیگر بسیار شباهت دارند. هر دو نسبت به کلاینت، ocp را نقض نمی‌کنند ولی در ساختارهای داخلی خود و تحقق الگوها به دلیل نقض dip، گسترش منجر به تغییر می‌شود و در نتیجه ocp می‌تواند نقض شود. اما کلاینت‌ها از تغییرات مادامی که interfaceها تغییر نکنند در امان هستند.
ساختار state/behavior over collection متفاوت است اما آن نیز ocp را نقض می‌کند.

LSP

Proxy

در این الگو این قاعده نقض می‌شود. چرا که proxy واقعا با پدر خود از یک جنس نیستند و proxy یک شی جعلی است که رفتار subject را جعل می‌کند تا بتوان به جای realSubject بنشیند وگرنه رابطه‌ی is/a بین proxy و subject برقرار نیست.
از طرفی شاید بتوان گفت که proxy در هر صورت باید بتواند جایگزین subject شود چرا که در غیر این صورت کلاینت نمی‌تواند به چشم یک subject به آن نگاه کند. این یکی از

Iterator

در این الگو این قاعده برقرار است. در واقع بستگی به کسی دارد که پیمایشگر را طراحی می‌کند اما باید ConcreteIteratorها و ConcreteAggregatorها بتوانند جایگزین کلاس پدر شوند.

State/Behavior over collection

در این الگو با توجه به ساختار و مفهوم آن، این قاعده چندان قابل طرح نیست اما به طور کلی می‌توان گفت که برقرار است.

مقایسه

الگوی proxy را از جهتی می‌توان گفت که این قاعده را در رابطه‌ی proxy و subject از نظر مفهومی نقض می‌کند. در iterator برقرار است و در الگوی سوم بلا موضوع است.

DIP

Proxy

در این الگو dip بین کلاینت و subject که interface مربوط به proxy و realSubject است برقرار است. پیش‌تر هم توضیح داده شد علت آن این است که کلاینت دید مستقیم به concreteها ندارد و از طریق واسط با concreteها در ارتباط است.
اما در رابطه‌ی میان Proxy و RealSubjectها، این قاعده برقرار نیست چرا که proxy لازم است تا دید مستقیم به RealSubject داشته باشد تا بتواند رفتار آن را در مواقع لزوم به شی اصلی محول کند. در نتیجه در رابطه‌ی proxy و شی اصلی این قاعده نقض می‌شود.

Iterator

در این الگو، dip میان کلاینت و پیمایش‌گر و همچنین میان کلاینت و شی مرکب کاملا برقرار است چرا که از طریق واسط آن‌ها را می‌بیند.
اما در رابطه‌ی میان ساختارهای توارثی Iterator و Aggregate، این قاعده برقرار نیست چرا که دید concrete به concrete میان این دو ساختار وجود دارد و از طریق واسط با یکدیگر ارتباط ندارند و در نتیجه، این قاعده نقض می‌شود.

State/Behavior over collection

در این الگو نیز این قاعده تقریبا بلا موضوع است اما می‌توان گفت برقرار است.

مقایسه

مانند ocp، می‌توان دید که نسبت به کلاینت این قاعده برقرار است ولی در داخل ساختار توارثی، رابطه‌ی میان concreteها وجود دارد.

ISP

Proxy

در این الگو، interface شی اصلی subject است که به اندازه‌ی کافی جداسازی وظایف را برعهده دارد و proxy نیز لازم است تا این واسط را پیاده‌سازی کند تا رفتار را جعل کند. این ریزدانه‌ترین حالتی است که می‌توان این واسط را تعریف کرد.

Iterator

در این الگو نیز این قاعده برقرار است. واسط iterator اساسا از aggregate جداسازی شده است تا تک وظیفه بودن شی مرکب رعایت شود و از طرفی اشیا متخصص پیمایش‌گر تنها کافی است واسط ساده‌ی iterator را پیاده‌سازی کنند و در نتیجه به خوبی این قاعده رعایت شده است.

State/Behavior over collection

در این الگو نیز می‌توان گفت این قاعده رعایت شده است. تعریف یک شی که نماینده کل مجموعه باشد و رفتارها و داده‌های مجموعه را در خود نگهداری می‌کند، موجب می‌شود تا رفتارهایی که نباید در اجزا باشد، از آن‌ها جدا شده و به واسط کل منتقل شود. در نتیجه به خوبی این قاعده رعایت شده است.

مقایسه

هر سه الگو این قاعده را رعایت می‌کنند. بخصوص دو الگوی آخر که هر دو با هدف بهبود انعطاف و وابستگی، رفتارها را از یک شی به متخصص خود واگذار می‌کنند و با تعریف واسط می‌توانند ریزدانگی کارهای انجام شده را بالاتر برده و cohesion را بهبود دهند.

CRP

Proxy

در این الگو این قاعده رعایت می‌شود. هر چند ساختارهای توارثی در این الگو دیده نمی‌شود و تنها proxy و RealSubject هستند که واسط Subject را پیاده‌سازی می‌کنند، اما می‌توان گفت که در این الگو به کمک رابطه‌ی association میان proxy و RealSubject و تحقق الگو در زمان اجرا، رابطه‌ی delegation بر inheritance ترجیح داده شده است.

Iterator

این الگو اساسا وظیفه‌ای که می‌توانست در Aggregate باشد را از آن جدا کرده و در یک interface مجزا به آن پرداخته است و از طریق رابطه‌ی association میان iterator و aggregate، الگو تحقق می‌یابد. در نتیجه delegation در این الگو بر inheritance ترجیح داده شده است که در ساختار الگو کاملا واضح است. در نتیجه این الگو نیز این قاعده را رعایت می‌کند. در زمان اجرا می‌توان iteratorهای مختلف روی شی مرکب تعریف کرد و این، یکی از نتایج رعایت این قاعده است.

State/Behavior over collection

این الگو رفتار و یا اطلاعات را از اجزا به کل منتقل می‌کند و رابطه‌ی association میان اجزا و کل می‌تواند برقرار باشد (به طراحی بستگی دارد). چرا که کل لزوما به معنی پدر نیست بلکه یک مجموعه‌ای از اجزا است. در نتیجه این الگو نیز (بسته به طراحی) می‌تواند این قاعده را رعایت کند. اما چندان مورد توجه هدف اصلی الگو نبوده است.

مقایسه

در هر سه الگو این قاعده رعایت شده است و delegation بر inheritance اولویت دارد. هر چند در الگوی آخر، شاید بتوان گفت که محل بحث نیست چرا که delegation و یا ساختار توارثی می‌تواند وجود نداشته باشد.

PLK

Proxy

در این الگو این قاعده رعایت شده است. در واقع هیچ جایی دید ترایا دیده نمی‌شود. میان کلاینت و اشیا و همچنین میان proxy و realSubject نیز دید ترایا دیده نمی‌شود و در نتیجه قاعده‌ی demeter رعایت شده است.

Iterator

در این الگو نیز این قاعده رعایت شده است. کلاینت نمی‌تواند از طریق فراخوانی، یک دید ترایا به aggregate پیدا کند و از طرفی، میان iterator و aggregate نیز این قاعده برقرار است.
هر چند امکان وقوع این حادثه در این شی وجود داشت. به گونه‌ای که کلاینت می‌توانست از طریق یک operation تعریف شده در واسط، به شی aggregate دسترسی پیدا کند. اما اساسا هدف این الگو جلوگیری از وابستگی شدید میان کلاینت و aggregate بوده و به همین دلیل واسط iterator به صورتی تعریف نمی‌شود که aggregate را در اختیار کلاینت قرار دهد. در نتیجه این قاعده برقرار است.

State/Behavior over collection

در این الگو نیز این قاعده برقرار است (هر چند چندان مورد بحث این الگو نیست)

مقایسه

در هر سه الگو این قاعده برقرار است و دید ترایا میان سه شی دیده نمی‌شود. هر چند iterator پتانسیل این اشکال را دارد اما هر سه رعایت می‌کنند.

مقایسهٔ الگوهای proxy، iterator و state over collection بر اساس grasp

Information Expert

Proxy

در این الگو می‌توانیم بگوییم نقض نشده است. رفتارهای اضافی بر شی، داده‌های مورد نیاز خود را یا در داخل شی proxy دارد و یا در شی RealSubject است که proxy در کپسول داده رفتار خود به آن دید مانا دارد و می‌تواند از آن استفاده کند و در نتیجه، رفتارها در کنار دادگان خود قرار گرفته اند.

Iterator

در این الگو این قاعده نقض می‌شود. علت آن، جلوگیری از coupling بالای کلاینت با aggregate بوده است. به همین منظور و با مصلحت بالاتر آن، تصمیم گرفته‌ایم تا رفتار پیمایش را از داده‌ی مربوط به آن که در Aggregate قرار دارد جدا کنیم با هدف کاهش coupling. در نتیجه این قاعده نقض می‌شود. یکی از علل coupling شدید در ساختارهای توارثی این الگو، همین جدایی داده از رفتار است.

State/Behavior over collection

هدف هر دو الگو، اساسا قرار دادن اطلاعات و رفتار در جایگاهی است که به آن مربوط می‌شود و در نتیجه، رفتار و داده مربوط به کل را در شی کل قرار می‌دهد و آن را از اجزا جدا می‌کند تا اطلاعات و رفتار در کنار یکدیگر قرار بگیرند و در نتیجه این الگو از این قاعده پیروی می‌کند.

مقایسه

الگوهای proxy و state/behavior over collection این الگو را در خود دارند اما iterator این الگو را نقض می‌کند.

Creator

Proxy

در شی proxy یک دید مانا به realSubject برقرار است. اما در حالت کلی الگو، وظیفه‌ی ساخت اشیا proxy و realsubject و پیکربندی کلاینت با این اشیا بر عهده‌ی یک پیکربند ثالث است در نتیجه اولویت سوم نقض شده و proxy باید سازنده‌ی realSubject باشد و این قاعده در proxy نقض شده است.
در virtual proxy اما این قاعده برقرار است. چرا که در این کاربرد از proxy، وظیفه‌ی ساخت شی اصلی با proxy است و در نتیجه این الگو نقض نمی‌شود. اما در حالت کلی این الگو، creator را نقض می‌کند.

Iterator

در این الگو، این قاعده رعایت می‌شود. شی مرکب می‌تواند توسط کلاینت و یا یک پیکربند ساخته شود که اطلاعات مورد نیاز آن را در دسترس دارد و با توجه به اولویت پنج و نبود اولویت بالاتر، این قاعده در این مورد رعایت می‌شود.
همچنین شی پیمایشگر توسط شی مرکب ساخته می‌شود که اطلاعات مورد نیاز برای ساخت آن را دارد (خود شی) و در نتیجه در اینجا نیز اولویت بالاتری وجود ندارد و قاعده رعایت شده است.

State/Behavior over collection

در این الگو، آفرینش شی چندان مورد توجه نیست و بلا موضوع است.

مقایسه

الگوی پیمایش‌گر این الگو را رعایت می‌کند، proxy در بیشتر حالات ان را نقض می‌کند و برای الگوی سوم نیز این الگو معنی چندانی ندارد.

Low Coupling

Proxy

همانطور که بالاتر نیز اشاره شده است، proxy یک wrapper است که یک سطح دسترسی غیر مستقیم از کلاینت به شی اصلی را ایجاد می‌کند و در نتیجه تا حدی از coupling این دو به صورت مستقیم جلوگیری می‌کند.
از طرفی ارتباط بین کلاینت و proxy نیز از طریق واسط است و در نتیجه coupling تا حد خوبی پایین است.
اما از طرفی یک دید مستقیم بین proxy و realSubject وجود دارد که منجر به وابستگی شدید میان این دو شی می‌شود. هر چند برای تحقق الگو این مورد لازم است.
به طور کلی، می‌توان گفت proxy از سطحی از coupling برخوردار است اما وضعیت آن چندان بد نیست.

Iterator

همانطور که در بخش اهداف و کاربردها اشاره شده است، ایجاد شی جعلی و متخصص iterator منجر به جدایی و عدم دسترسی مستقیم کلاینت به اطلاعات داخلی شی مرکب می‌شود که خود منجر به کاهش coupling شدید میان کلاینت و aggregate است و در نتیجه در تخصیص وظایف، کاهش coupling میان کلاینت و شی مرکب مد نظر بوده است.
اما میان ساختارهای توارثی iterator و aggregate، وابستگی نسبتا شدید است هر چند مطابق هدف اصلی ما، رابطه‌ی aggregate و کلاینت مورد توجه بوده است.

State/Behavior over collection

در این الگو نیز وظایف از اجزا به کل منتقل می‌شود و در نتیجه تقسیم وظایف به گونه‌ای انجام می‌شود که برای فراخوانی یک رفتار یا به دست اوردن داده‌ای مربوط به کل مجموعه، به تک تک اجزا وابسته نباشیم بلکه یک شی کل اطلاعات و رفتار را در خود نگهداری کند و در نتیجه این الگو نیز به کاهش coupling توجه دارد.

مقایسه

الگوهای proxy و iterator، وابستگی میان کلاینت و ساختار پیشین را کاهش می‌دهند اما یک ساختار جدید با coupling ایجاد می‌کنند که هر چند به علت مصلحت بزرگتری مانند کاهش وابستگی به کلاینت، می‌توان نادیده گرفت. الگو سوم اما کاهش وابستگی را به همراه دارد.

High Cohesion

Proxy

در این الگو یک شی متخصص تعریف می‌شود که رفتارهایی را به شی اصلی می‌افزاید و یا وظایفی را برعهده می‌گیرد که برای مصالح مختلفی به کار می‌روند و همچنین این وظایف لزوما مرتبط با شی اصلی نبوده و در نتیجه، cohesion شی اصلی را بالا نگه می‌دارند. در نتیجه در مجموع شی متخصصی تعریف می‌شود که cohesive است و مجموعه را cohesive نگه می‌دارد و در نتیجه این الگو برقرار است.

Iterator

همانطور که در افزایش cohesion در مقایسه‌ی الگوها آورده شد، این الگو یک ساختار متخصص پیمایش تعریف می‌کند که کارهای مربوط به پیمایش را از دوش شی مرکب برداشته و خود بر عهده می‌گیرد و در نتیجه، این الگو منجر به افزایش cohesion می‌شود.

State/Behavior over collection

بی راه نیست اگر بگوییم این الگو هدف اصلی خود را بر افزایش cohesion گذاشته است. چرا که وظیفه یا داده های نامربوط به اشیا جز را از آن‌ها جدا کرده و به شی کل منتسب می‌کند که باعث افزایش cohesion می‌شود و رفتارهای نامربوط را از اشیا نامربوط جدا می‌کند.

مقایسه

هر سه الگو، این قاعده را رعایت می‌کنند.

Controller

Proxy

می‌توان این الگو را به نوعی کارچرخان در نظر گرفت که درخواست‌ها را از کلاینت دریافت کرده و سپس به شی اصلی منتقل می‌کند. هر چند proxyها تنها کارچرخانی نمی‌کنند بلکه اغلب یک رفتار و وظیفه‌مندی به رفتار اصلی می‌افزایند که در نتیجه کار آن‌ها تنها کارچرخانی نیست. در نتیجه شاید controller را نتوانیم به صورت کامل در این الگو ببینیم.

Iterator

در این الگو مشخصا از controller استفاده نشده است و هیچ یک از سه شی اصلی این وظیفه را برعهده ندارند.

State/Behavior over collection

این الگو نیز controller معنی ندارد.

مقایسه

شاید بتوان به proxy وظیفه‌ی کنترلر را نسبت داد اما در دو الگوی دیگر این الگو دیده نمی‌شود.

Polymorphism

Proxy

چند ریختی اساسا مورد توجه این الگو است. چرا که پیاده‌سازی operation تعریف شده در واسط، در proxy و realsubject متفاوت است و اساسا هدف اصلی این الگو نیز تعریف مضاعف رفتار بر شی اصلی است و در نتیجه polymorphism در این الگو کاملا دیده می‌شود و از اهداف الگو پشتیبانی می‌کند.

Iterator

در این الگو، polymorphism دیده می‌شود. واسط iterator این امکان را می‌دهد تا ConcreteIteratorهای متنوعی را با پیاده‌سازی‌های مختلف پیمایش داشته باشیم. برای مثال، می‌توانیم پیمایش‌گرهای ،preorder postorder و inorder را به کمک چندریختی پیاده‌سازی کنیم و هر یک را به یک شی مرکب graph نسبت دهیم. همچنین اشیا مرکب نیز امکان چندریختی دارند و در نتیجه در این الگو نیز دیده می‌شود.

State/Behavior over collection

در این الگو، polymorphism دیده نمی‌شود.

مقایسه

در دو الگوی proxy و iterator به خوبی از polymorphism برای تحقق الگو استفاده می‌شود اما در الگوی سوم این مورد وجود ندارد.

Indirection

Proxy

اساس این الگو بر ایجاد یک شی میانجی است تا رفتارهایی به شی اصلی اضافه کند. در نتیجه این الگو کاملا از indirection بهره می‌برد.

Iterator

این الگو نیز از indirection بهره می‌برد. در حالت پیشین، کلاینت به صورت مستقیم بایستی به شی مرکب دید داشته و عملیات را به گونه‌ای انجام می‌داد اما پس از اعمال الگو، وظیفه‌ی پیمایش برعهده‌ی پیمایش‌گر قرار گرفته و در نتیجه این شی، یک سطح indirection اضافه می‌کند.

State/Behavior over collection

در این الگو نمی‌توان indirection دید و کلاینت مستقیما اطلاعات و رفتار مورد نیاز خود را از شی مربوطه دریافت خواهند کرد.

مقایسه

در دو الگوی proxy و iterator می‌توان این الگو را مشاهده کرد اما در الگوی سوم این الگو وجود ندارد.

Pure Fabrication

Proxy

همانطور که در بخش شی جعلی اشاره شد، proxy در قلمرو مسئله و در مرحله‌ی تحلیل قابل برداشت نیست بلکه یک شی جعلی است که جایگزین شی اصلی می شود و کلاینت تفاوت این دو شی را متوجه نمی‌شود. هدف از آن نیز افزودن قابلیت‌ها و رفتارهایی به شی اصلی است بدون تاثیر گرفتن کلاینت از تغییرات. در نتیجه این الگو از یک شی جعلی برای تحقق خود استفاده می‌کند.

Iterator

شی iterator یک شی جعلی است که در قلمرو مسئله دیده نمی‌شود و برای کاهش coupling میان کلاینت و شی مرکب و برای تحقق کاربردهای الگو، تعریف می‌شود. این شی یک شی متخصص است که وظیفه‌ی پیمایش روی شی مرکب را برعهده می‌گیرد. در نتیجه این الگو از pure fabrication بهره می‌برد.

State/Behavior over collection

این الگو از شی جعلی برای تحقق هدف خود بهره نمی‌برد.

مقایسه

دو الگوی proxy و iterator برخلاف State/Behavior over collection از pure fabrication استفاده می‌کنند.

Protected Variations

Proxy

در این الگو از protected variation استفاده می‌شود. هر چند که هدف از proxy، جلوگیری از تغییرات شی کلاینت نیست بلکه افزودن رفتار و وظیفه‌مندی یا بهینه‌سازی مصرف منابع است، اما ارتباط کلاینت با اشیا proxy و realSubject از طریق واسط صورت می‌گیرد و تغییرات ساختاری آن‌ها به بیرون منتشر نمی‌شود. البته باید توجه داشت که هدف از الگو چیز دیگری بوده است و اساسا رابط برای ارتباط بین کلاینت و realSubject پیش از اعمال الگو نیز می‌تواند وجود داشته باشد پس تحقق آن لزوما به الگو مرتبط نبوده است.
شایان ذکر است که هدف remote proxyها را می‌توان به گونه‌ای protected variation در نظر گرفت. در واقع برای انکه وظایف مربوط به ارتباطات شبکه و خارج از برنامه را از کلاینت منتزع کنیم، رفتارهای مربوطه را به یک remote proxy محول می‌کنیم تا ارتباط با شی خارجی را برعهده بگیرد و به همین شکل، جلوی انتشار تغییرات مداوم در اثر تغییرات مورد نیاز در پروتوکل و ... در کلاینت گرفته می‌شود.

Iterator

در این الگو با تعریف یک واسط پیمایش‌گر، کلاینت و شی مرکب را از یکدیگر برای پیمایش جدا کرده‌ایم و در نتیجه هر گونه تغییرات در اشیا مرکب و یا تغییر در روش‌های پیمایشی که کاملا ممکن هستند، در کلاینت منتشر نمی‌شوند و در نتیجه اینگونه با تعریف واسط، یک protected variation ایجاد کرده‌ایم. هر چند تغییرات شی مرکب به راحتی به iterator می‌تواند منتقل شود چرا که dip وجود ندارد.

State/Behavior over collection

در این الگو، protected variations دیده نمی‌شود و ارتباطات از طریق واسط مد نظر نبوده است.

مقایسه

الگوی proxy تا حدی از این الگو پیروی می‌کند هر چند خود منجر به استفاده از این الگو در ساختار نشده است. البته remote proxyها مصداق عینی protected variation هستند.
الگوی iterator در رابطه‌ی بین کلاینت و شی مرکب از این الگو بهره برده و با تعریف واسط، پیمایش را از میان این دو برداشته است. الگوی آخر اما از این الگو بهره نمی‌برد.

مقایسهٔ الگوهای Command، Bridge و Strategy

دسته - نوع

Bridge

این الگو از الگوهای ساختاری می‌باشد. این دسته از الگوها بیشتر به ساختاردهی کلاس‌ها با هدف کاهش coupling و انعطاف‌پذیری بیشتر توجه دارند. این کار را معمولا با جداسازی abstraction از implementation و گسترش ساختار انجام می‌دهند. هر چند ممکن است این الگوها وجوه رفتاری نیز داشته باشند (که در proxy نیز وجه رفتاری جایگزینی مشخص است) اما وجوه ساختاری در آن پررنگ‌تر است.
این الگو برخلاف اکثریت الگوهای ساختاری، در دسته‌ی wrapperها محسوب نمی‌شود.

Command

این الگو از الگوهای رفتاری می‌باشد. این الگوها به رفتار اشیا در زمان اجرا توجه دارند و نحوه‌ی تعامل اشیا با یکدیگر تمرکز دارند. هر چند ممکن است وجوه ساختاری نیز داشته باشند اما این وجه در آن‌ها پررنگ‌تر است.
این الگوها، دغدغه‌ی تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار شی، اداره‌ی درخواست‌های به یک شی و همچنین مدیریت بهتر تعامل میان اشیا در زمان اجرا با هدف افزایش cohesion و کاهش coupling را دارند.

Strategy

این الگو از الگوهای رفتاری می‌باشد. توضیحات مربوطه در بخش command داده شده است.

مقایسه

دو الگوی command و strategy در یک دسته از الگوها قرار دارند و وجه رفتاری آن‌ها پررنگ‌تر از وجوه ساختاری آن است. از طرفی bridge از الگوهای ساختاری است که وجوه ساختاری آن اهمیت بیشتری دارد.

هدف

Bridge

هدف اصلی این الگو، جداسازی پیاده‌سازی یا implementation از abstraction است به گونه‌ای که بتوان هر دو ساختار را گسترش داد. در واقع معمولا هدف با گسترش پیاده‌سازی است و abstraction معمولا ساختار غیر قابل گسترشی دارد. اما در اینجا به کمک جدا کردن پیاده‌سازی از abstraction، دو ساختار موازی که dip در آن‌ها برقرار است و coupling در سطح concrete وجود ندارد ساخته می‌شود تا پیاده‌سازی‌های مختلف از abstractionهای مختلف جدا شده و بتوان هر دو را موازی و مستقل از یکدیگر گسترش داد.

Command

معمولا انتقال پیام میان دو شی از طریق فراخوانی و به صورت آنی انجام می‌شود. اما هدف این الگو، تبدیل پیغام میان دو شی به یک شی دیگر است تا بتوان پیام منتقل شده را نه به صورت آنی بلکه در طول زمان نگهداری کرد. هدف از آن می‌تواند برای logging، امکان undo، امکان persist کردن پیغام، صف کردن پیغام‌ها و همچنین تعریف رفتارهای مختلف برای یک پیغام استفاده کرد. در واقع هدف اصلی، مانا کردن انتقال پیغام میان دو شی است.

Strategy

فرض کنید خانواده‌ای از الگوریتم‌ها داریم که می‌خواهیم بسته به شرایط، یکی از آن‌ها را انتخاب کرده و آن را اجرا کنیم. شرایط می‌تواند توسط کلاینت، توسط سیستم و یا اکتور خارجی و ... تعیین شود.
برای مثال الگوریتم‌هایی از یک خانواده داریم که یکی time optimized و دیگری space optimized هستند و بر اساس آنکه در حال حاضر حافظه‌ی سیستم و یا پردازنده چقدر درگیر هستند، الگوریتم را انتخاب می‌کنیم.
در نتیجه هدف اصلی جایگزینی الگوریتم در زمان اجرا است بدون coupling بالا و سیستم بتواند بر اساس نیاز پیکربندی شود. خود خانواده الگوریتم نیز بدون منتشر شدن تغییرات و تاثیر بر روی کلاینت‌ها قابل گسترش باشد.

مقایسه

میان bridge و strategy می‌توان شباهت‌هایی یافت. در bridge می‌خواهیم هم پیاده‌سازی و هم abstraction را گسترش دهیم بدون اینکه dip بالا باشد. در اینجا نیز می‌توانیم گسترش الگوریتم‌ها را مانند گسترش رفتار و یا همان پیاده‌سازی در نظر بگیریم و از این جهت شباهت دارند. هر چند این گسترش‌پذیری رفتار جزو اهداف اصلی strategy نیست بلکه قابل پیکربندی بودن آن هدف اصلی است که این را نیز می‌توان به

صورت یک مورد جانبی در bridge نیز پیدا کرد. در واقع در bridge نیز امکان پیکربندی وجود دارد هر چند برخلاف strategy، هدف اصلی پیکربندی نیست بلکه هدف قابل گسترش کردن دو ساختار abstraction و implementation است.

الگوی command هدف کاملا متفاوتی از هر دو دارد.

حوزه

Bridge

این الگو در سطح object و به کمک delegation و در زمان اجرا محقق می‌شود.

Command

این الگو در سطح object و به کمک delegation و در زمان اجرا محقق می‌شود.

Strategy

این الگو در سطح object و به کمک delegation و در زمان اجرا محقق می‌شود.

مقایسه

هر سه در زمان اجرا و در حوزه‌ی object هستند.

کاهش coupling

Bridge

در این الگو به کمک جداسازی implementation از abstraction این امکان را به وجود می‌آوریم که این دو ساختار مستقل از یکدیگر و بدون وابستگی بتوانند رشد کنند.
میان این دو ساختار dip کاملا برقرار است یعنی تنها از طریق interfaceها، abstraction به implementation دید دارد و به کلاس‌های concrete دید ندارد. همچنین کلاینت نیز تنها به abstraction دید دارد و در نتیجه کاملا منتزع از جزئیات پیاده‌سازی است و dip کاملا برقرار است. تنها مشکل، پیکربند است که باید abstraction را با implementation مناسب پیکربندی کند که برای این کار لازم است تا تمام زیرکلاس‌های implementation را بشناسد.
همچنین باید توجه کرد که در حالت پیشین، ممکن بود مجبور شویم تا زیرکلاس‌های متعدد و ترکیبی ایجاد کنیم. اما حالا با جداسازی پیاده‌سازی از abstraction، کلاس‌های متعدد بوجود نمی‌آید. اما در کل از نظر coupling، این الگو در حالت پسین بسیار خوب است.

Command

این الگو چندان بر روی coupling تمرکز ندارد. اما یک سطح indirection میان receiver و invoker ایجاد می‌کند. در واقع در حالت پیشین الگو، invoker مستقیما به receiver دید داشته است اما حالا این دید از بین رفته و به جای آن command را می‌بیند. ولی در حالت پسین نیز command باید مستقیما به receiver دید داشته باشد. در نتیجه دید میان invoker و receiver به دید میان command و هر دوی آن‌ها تبدیل شده است. اما dip میان command و invoker و همچنین command و receiver برقرار است.
نکته‌ی حائز اهمیت این که میان client و command، با توجه به instantiation و پیکربندی command برای invoker توسط کلاینت، dip برقرار نیست. در نتیجه coupling میان کلاینت و command شدید است. در حالت پیشین client تنها با invoker در ارتباط بود و dip نیز می‌توانست برقرار باشد (بسته به تعریف interface مربوط به invoker) اما در این حالت، کلاینت به command وابستگی شدید خواهد داشت. اما این وابستگی در پیکربندی‌ها دیده می‌شود و می‌توان نادیده گرفت.
در مجموع می‌توان گفت تاثیر command روی وابستگی چشم‌گیر نیست.

Strategy

در این الگو، میان context و strategy به صورت کامل dip برقرار است و در نتیجه coupling در وضعیت بسیار خوبی است. اما پیکربند که می‌تواند همان کلاینت، اکتور خارجی و یا یک پیکربند سیستمی و ... باشد، لازم است

تا نسبت به کلاس‌های ConcreteStrategy دید داشته باشد و در نتیجه dip نقض می‌شود. اما در مجموع dip پیکربند را می‌توانیم نادیده بگیریم و در نتیجه وضعیت این الگو از نظر coupling مناسب است.

مقایسه

این الگوها از نظر coupling بسیار شبیه یکدیگر هستند. می‌توان گفت وضعیت پسین آن‌ها از نظر coupling بسیار مناسب است و dip تا حد خوبی برقرار می‌شود اما این الگوها تاکید زیادی روی coupling ندارند و در واقع حالت پیشین خود را بهبود چندانی نمی‌دهند. هر چند می‌توان گفت bridge تا حدی وضعیت پیشین خود را بهبود می‌دهد اما بیشتر تمرکز آن روی cohesion است تا coupling. در نتیجه این الگوها حالت پیشین و پسین مناسبی از نظر coupling دارند و تنها مشکل همگی آن‌ها وابستگی شدید پیکربندی است که در اکثر الگوها این موضوع دیده می‌شود.

افزایش cohesion

Bridge

این الگو cohesion را بهبود می‌دهد. چرا که پیاده‌سازی را به کلاس‌های متخصص پیاده‌سازی می‌سپارد و در نتیجه مسئولیت‌های اضافی را از abstraction جدا کرده و به پیاده‌سازی می‌دهد و دو ساختار پیاده‌سازی و انتزاع را از یکدیگر جدا می‌کند. می‌توان گفت اساس این الگو برای بالا بردن cohesion و seperation of concerns بوده است و در نتیجه، از نظر افزایش cohesion این الگو بسیار مناسب است.

Command

این الگو نیز انسجام را بهبود می‌دهد. Commandها به جای آنکه به صورت مستقیم و از طریق فراخوانی میان دو کلاس صورت پذیرد، به کمک جداسازی و سپردن انتقال پیام به یک کلاس، یک هویت برای انتقال پیام در نظر می‌گیرد و در نتیجه می‌توانیم بگوییم cohesion تا حد خوبی افزایش می‌یابد.

Strategy

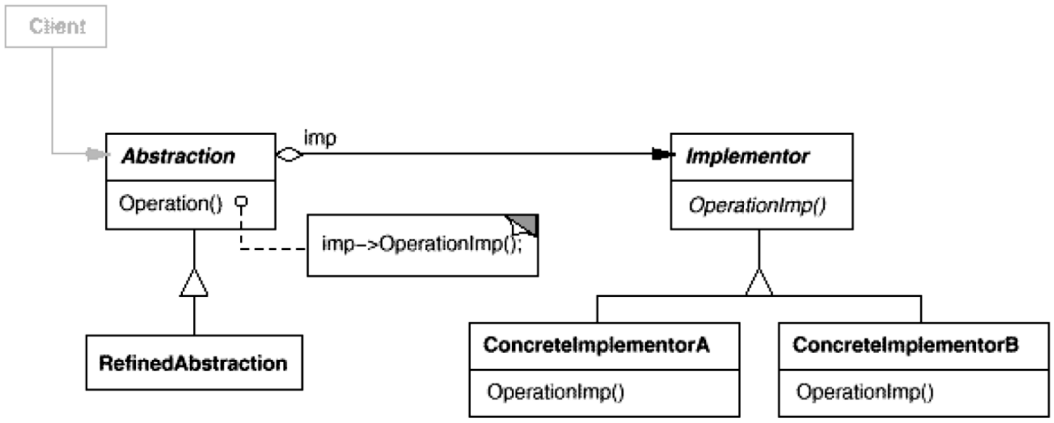
هدف اصلی این الگو پیاده‌سازی‌های رفتارهای مختلف و الگوریتم‌های مختلف از context است و این رفتارها یا الگوریتم‌ها را به یک ساختار مجزا محول می‌کند و در نتیجه seperartion of concerns را بهبود داده و به افزایش cohesion کمک شایانی می‌کند.

مقایسه

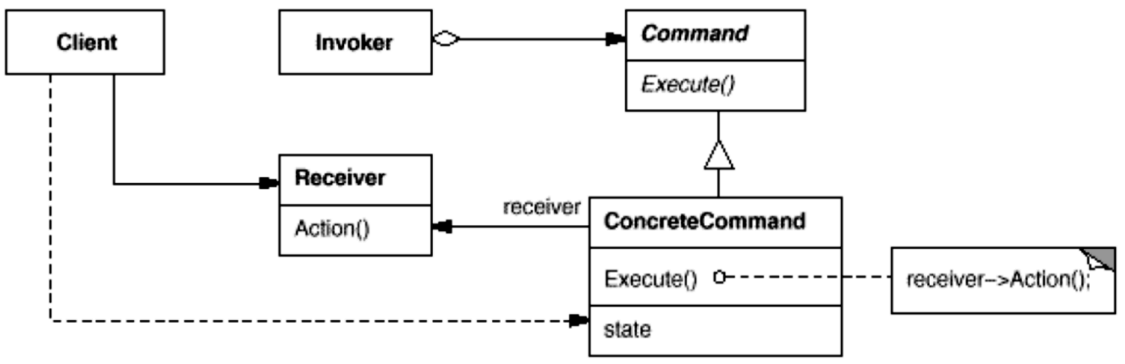
هر سه الگو به افزایش cohesion کمک زیادی می‌کنند و از این منظر شباهت بسیاری دارند. در واقع جداسازی و ایجاد اشیا و ساختارهای متخصص و جداکردن مسئولیت‌ها یکی از کارهای مهم هر سه الگو است.

ساختار و رفتار

Bridge

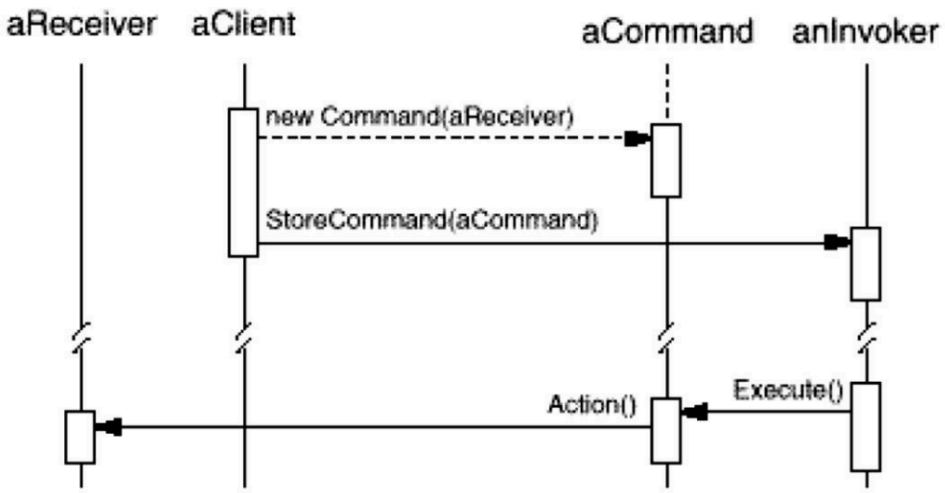


نمودار کلاس این الگو را در شکل می‌بینید. کلاینت تنها به abstraction دید دارد. Abstraction نیز لازم است تا یک رابطه‌ی association با implementor داشته باشد تا بتواند در انتزاع‌های مختلف آن را فراخوانی کند.
لازم به ذکر است که رابطه‌ی aggregation میان abstraction و implementor بی معنی است و لزومی ندارد بلکه یک association کافی است.
پیکربند در زمان اجرا، یک instance از implementor را در اختیار abstraction و یک instance از abstraction را در اختیار کلاینت قرار می‌دهد تا الگو محقق شود.

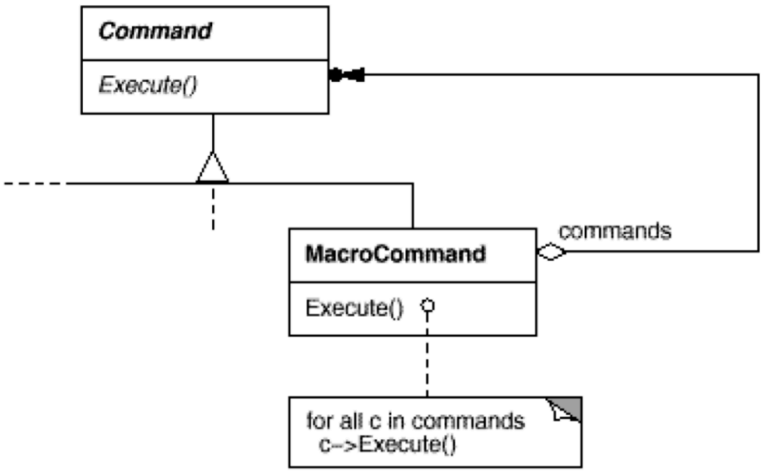


در شکل بالا، یک نمودار کلاسی البته نادرست از command را که در کتاب اشاره شده است می‌بینیم. این نمودار نقص‌هایی دارد که ابتدا به آن‌ها اشاره کرده و سپس به توضیحات آن و سپس به نمودار رفتاری می‌پردازیم.
یکی از نقص‌های نمودار، رابطه‌ی مانا میان client و receiver است. دید میان کلاینت و receiver نیز لازم است اما لزوما مانا نیست. این دید برای این است که یک شی از receiver را بتواند در اختیار command قرار دهد. از طرفی یک دید میان client و invoker باید باشد تا کلاینت کامند را در اختیار آن قرار دهد تا اجرا کند.
رابطه‌ی میان command و invoker نباید از نوع aggregation بلکه یک association کافی است.
در نتیجه مطابق چیزی که گفته شد و بخش‌های درست نمودار، برای تحقق الگو، لازم است تا کلاینت یک ConcreteCommand را بسازد (یا از پیش در اختیار داشته باشد) و آن را با یک receiver مناسب پیکربندی کند. سپس این command را در اختیار یک invoker قرار می‌دهد. Invoker در زمان مناسب، command را اجرا می‌کند و پیام به دست receiver می‌رسد.
برای فهم بهتر این رفتار، به نمودار sequence diagram زیر می‌توانیم نگاه کنیم.

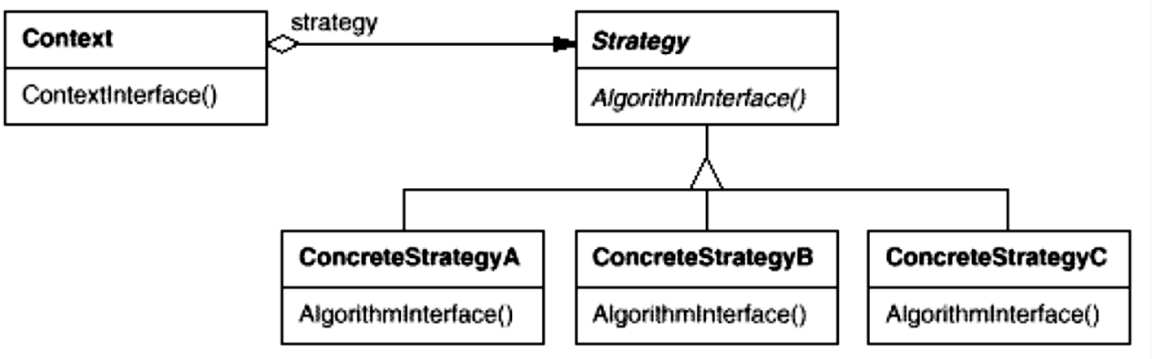
مطابق توضیحات بالا، کلاینت درخواست ساخت command را با یک receiver مناسب انجام می‌دهد و سپس command را در یک invoker ذخیره کرده و سپس اجرا می‌کند تا پیام به receiver برسد.



در ادامه‌ی این الگو، می‌توانیم به الگوی macro command یا command مرکب نیز اشاره کنیم. Command مرکب، خود از چندین command تشکیل شده و با ترتیب و اجرای مناسب، آن‌ها را اجرا می‌کند. نمودار کلاسی آن به شکل زیر است:



Strategy



ساختار کلاس این الگو در شکل آمده است. Context یک دید سطح بالا به stategy دارد که بسته به شرایط و موارد نیاز کلاینت یا شرایطی که منجر به پیکربندی می‌شود، پیکربندی context با strategy مناسب انجام می‌شود.

شایان ذکر است که کلاینت یک رابطه‌ی سطح بالا با context خواهد داشت و کلاینت (که در اینجا پیکربند نیز هست)، به کلاس‌های ConcreteStrategy دید دارد تا بتواند پیکربندی انجام دهد. در نتیجه الگو در زمان اجرا و از طریق delegation انجام می‌شود. همچنین در برخی شرایط، در صورتی که پارامترهای ورودی الگوریتم‌ها یکی نباشد و نخواهیم آن را بزرگ کنیم، ممکن است یک دید غیر مانا از strategy به context داشته باشیم و context به عنوان ورودی خود را به الگوریتم‌ها بدهد تا اطلاعات و داده‌ی مورد نیاز را از آن استخراج کنند.

مقایسه

میان ساختارهای این سه الگو شباهت‌هایی دیده می‌شود. در هر یک، ساختارهای توارثی جدیدی ساخته می‌شود که می‌توانند به راحتی گسترش یابند (ساختار implementor، command و Stratagy) بدون آنکه dip نقض شود (البته مجزا از پیکربند که همواره نیاز به دید به concreteها دارد). این کلاس‌ها نتیجه‌ی جداسازی مسئولیت‌ها هستند و از هر یک، یک شی instantiate شده و توسط پیکربند در اختیار شی استفاده کننده قرار می‌گیرد.

از طرفی بر خلاف Bridge، هم Strategy و هم Command توسط کلاینت پیکربندی می‌شوند. در واقع کلاینت به command و strategy دید دارد و invoker با command مناسب و context با strategy مناسب پیکربندی می‌شوند. در bridge اما پیکربندی مجزا از کلاینت است و کلاینت دید مستقیم به ساختار سطح پایین ساختارهای توارثی جدید ندارد.

پیکربندی

Bridge

همانطور که در بخش قبل کمی توضیح داده شد، یک پیکربند خارجی لازم است تا implementor مناسب را برای abstraction پیکربندی کند. در ساختار نهایی، یک کلاینت داریم که با abstraction در ارتباط است و از طرفی یک پیکربند، implementor را برای abstraction پیکربندی می‌کند.
این الگو به خوبی قابل پیکربندی است. هدف اصلی این است که بتوانیم abstractionها را با implementorهای مختلف بتوانیم پیکربندی کنیم.

Command

پیکربندی این الگو توسط کلاینت صورت می‌گیرد. Command باید با receiver مناسب پیکربندی شود که کلاینت هنگام ایجاد یک شی جدید از command، آن را با receiver مناسب پیکربندی می‌کند. از طرفی خود کلاینت نیز برای انتقال پیام، یک invoker را باید با command پیکربندی کند تا الگو محقق شود. در نتیجه کلاینت وظیفه‌ی پیکربندی کند و در نهایت الگو محقق می‌شود. در نتیجه client به راحتی می‌تواند پیام‌های متفاوتی را پیکربندی کند و به خوبی قابل پیکربندی می‌باشد.

Strategy

در این الگو، context توسط کلاینت با یک strategy مناسب شرایط، پیکربندی می‌شود. در واقع کلاینت باید به concreteStrategyها دید داشته باشد تا context را پیکربندی کند. همچنین در صورتی که شرایط تغییر کرد و لازم به استفاده از strategy دیگری بود، به راحتی قابل تغییر و پیکربندی مجدد می‌شود.
البته این پیکربندی توسط سیستم، ورودی اکتور خارجی و یا موارد دیگر نیز می‌تواند انجام شود.

مقایسه

هر سه الگو قابل پیکربندی هستند. در الگوهای command و استراتژی، کلاینت وظیفه‌ی پیکربندی را بر عهده دارد و در bridge، پیکربند یک شی ثالث است که به implementorها دید خواهد داشت.

انعطاف‌پذیری

Bridge

این الگو یک ساختار بسیار صلب را به دو ساختار توارثی منعطف تبدیل می‌کند. در واقع با توجه به هدف اصلی این الگو، با جداسازی abstraction از implementorها، این دو ساختار می‌توانند مستقل از یکدیگر رشد کنند بدون آن که به یکدیگر وابستگی زیاد داشته باشند. در نتیجه‌ی آن، می‌توان implementorهای جدید را تعریف کرد و همچنین به خوبی می‌توان آن را پیکربندی مجدد کرد و در ساختار جدید، تعریف پیاده‌سازی‌های جدید بسیار راحت‌تر از حالت پیشین می‌شود. در حالت پیشین، برای تعریف پیاده‌سازی‌های جدید، مجبوریم تا زیرکلاس‌های ترکیبی تعریف کنیم که منجر به افزایش زیاد و زائد کلاس‌ها می‌شود. در نتیجه حرکت از یک ساختار کلاسی که شامل پیاده‌سازی و abstraction به صورت همزمان می‌شود و رفتن به delegation وظیفه‌ی پیاده‌سازی به ساختار پیاده‌سازی، ساختار جدید بسیار منعطفی ایجاد می‌شود.

Command

در حالت پیشین این الگو، کلاینت به صورت مستقیم پیغام را در لحظه ارسال می‌کرد. پس از اعمال الگو، پیام در قالب یک شی مستقل در آمده و امکان پیکربندی، صف، لاگ، persist و ... را به ما می‌دهد. همچنین می‌توان با پیغام‌های مختلف، پیکربندی انجام داد و در نتیجه اعمال این الگو، یک ساختار تقریبا غیر منعطف را به یک ساختار تغییر پذیر و منعطف تبدیل می‌کند. این کارها همگی با برقراری dip و همچنین در نتیجه‌ی منتشر نشدن تغییرات داخلی command به invoker و client انجام می‌شود.

Strategy

این الگو نیز در حالت پیشین، ساختار منعطفی ندارد و مشکلاتی مانند کلاس‌های ترکیبی در اثر inheritance خواهد داشت. در واقع در حالت پیشین برای تعریف الگوریتم‌های جدید، ممکن است مجبور شویم برای همه‌ی زیرکلاس‌های context، یک زیرکلاس جدید تعریف کنیم. پس از اعمال الگو، می‌توان به راحتی استراتژی‌های جدید تعریف کرد بدون آنکه نیاز به تغییر در کلاس‌های context و client باشد. پیکربندی آن نیز قابل تغییر است و در نتیجه در زمان اجرا می‌توان با انتساب اشیا جدید، یک ساختار منعطف را ایجاد کرد. گسترش کلاس‌های استراتژی نیز کاملا امکان پذیر است و در مجموع، یک ساختار منعطف ایجاد می‌شود.

مقایسه

همه‌ی الگوها، به خوبی با ایجاد یک ساختار جدید که dip برقرار می‌شود و تغییرات را منتشر نمی‌کند، امکان تغییر در زمان اجرا و همچنین گسترش‌پذیری را به ما می‌دهند و در نتیجه، هر سه منعطف هستند. در میان این

سه الگو، bridge و strategy، انعطاف‌پذیری را بیشتر از command تحت الشعاع قرار می‌دهند. به خصوص که کلاس کلاینت در command، وابستگی به زیر کلاس‌های command دارد و همچنین دید به receiver و invoker و وظیفه‌ی پیکربندی آن سبب می‌شود تا انعطاف‌پذیری آن کمتر از دو الگوی دیگر باشد. اما در دو الگوی دیگر، به خوبی ساختارها از یکدیگر منتزع هستند و گسترش و پیکربندی آن به خوبی انجام می‌پذیرد.

کارایی

Bridge

در این الگو با تعریف یک ساختار توارثی مجزا برای پیاده‌سازی و استفاده از delegation، رفتاری که در حالت پیشین به کمک inheritance و داخل abstraction پیاده‌سازی شده بود، به یک شی دیگری واسپاری شده است. در نتیجه performance فدای انعطاف‌پذیری شده است و نسبت به حالت پیشین پیغام بیشتری رد و بدل می‌شود. اما این مورد چشم گیر نیست چرا که تنها یک سطح انتقال پیام اضافی انجام می‌شود.

Command

در این الگو به جای انتقال پیام مستقیم و با فراخوانی، از یک شی برای انتقال پیام و انجام عملیات بهره برده شده است و در واقع یک سطح indirection به مجموعه اضافه می‌شود. در نتیجه performance نسبت به قبل کاهش یافته است. همچنین احتمال به وجود امدن اشیا متعدد وجود دارد چرا که به جای هر انتقال پیام مستقیم بایستی یک شی جدید ساخته و از طریق آن پیام منتقل شود که سربار اضافی دارد.

Strategy

در این الگو، پیاده‌سازی‌هایی که داخل context انجام می‌شده، به شی بیرونی محول می‌شود. در عین حال اطلاعات لازم برای پیاده‌سازی الگوریتم ممکن است با انتقال خود context همراه شود. در نتیجه یک سطح انتقال پیام میان اشیا اضافه شده و همچنین فراخوانی به دلیل جدا بودن داده از رفتار ممکن است هزینه‌ی زیادی داشته باشد. در نتیجه این الگو نیز کارایی را کاهش می‌دهد که امری طبیعی در الگوهای gof است چرا که در ازای آن انعطاف و کاهش coupling به دست می‌آید و این کاهش کارایی چندان چشم‌گیر نیست.

مقایسه

در هر سه الگو یک سطح انتقال پیام غیر مستقیم افزوده شده و در command امکان تعدد تعریف اشیا نیز بیش از بقیه وجود دارد و در نتیجه، کارایی در هر سه کاهش می‌یابد اما این کاهش کارایی با دستاوردهایی همراه است که آن را قابل چشم‌پوشی می‌کند. مهمترین دستاورد آن افزایش انعطاف پذیری یا بهبود ساختار است.

عواقب و تبعات

Bridge

اثرات مثبت

- interface و implementation از یکدیگر جدا می‌شوند و binding میان آن‌ها در زمان اجرا انجام می‌شود و انعطاف پذیر است. یعنی در زمان اجرا یک abstraction را به یک impl خاص پیکربندی کنیم.
- گسترش‌پذیری بهبود پیدا می‌کند چرا که هر کدام به صورت مستقل به دلیل وجود dip در سطوح مختلف، می‌توانند رشد پیدا کنند و تغییر منتشر نمی‌شود.
- پیاده‌سازی و جزئیات آن از کلاینت جدا است و پنهان است و تاثیری بر روی کلاینت ندارد.
+ (کاربرد خاص) در سیستم‌های تجاری ممکن است ساختارهای توارثی بسیار عمیق شده باشند بر اساس ترکیب میان چند ساختار. با tease apart inheritance می‌توان این ساختارها را جدا کرد. این الگو که یکی از الگوهای refactoring است، شباهت بسیاری به bridge دارد.

Command

اثرات مثبت

- شی فراخواننده از شی دریافت‌کننده decouple می‌شود. در زمانی که command مشخص می‌شود نوع رفتار مشخص می‌شود.
- command ها خودشان object هستند و در یک دوره‌ای در سیستم هستند و می‌توان آن‌ها را extend کرد و یا با یکدیگر جمع کرد.
- commandها را می‌شود تجمیع و composite کرد و به صورت macro command از آن استفاده کرد.
- commandهای جدید می‌توان اضافه کرد و تغییر به جای دیگری منتشر نمی‌شود. Invoker تنها به interface وابسته است و dip برقرار است البته پیکربندها تاثیر می‌پذیرند (اینجا کلاینت پیکربند است) اما invoker یا فراخواننده جدا می‌شود.

اثرات منفی

- اثر منفی مهم آن، اضافه شدن تعداد اشیا و به وجود آمدن یک سطح indirection است که البته به علت مزایای این الگو، این معایب را می‌پذیریم.

Strategy

اثرات مثبت

- اجازه‌ی ساختن خانواده‌های مختلفی از الگوریتم‌های مرتبط بسازیم و به راحتی گسترش می‌پذیرد.
- یک جایگزین برای subclass ساختن از context است که برای هر operation که خود می‌تواند الگوریتم‌های مختلفی را استفاده کند، مجبور شویم زیرکلاس بسازیم و یا حتی زیرکلاس‌های ترکیبی از الگوریتم‌های مختلف بسازیم و در نتیجه انفجار ترکیبی رخ دهد. این راه حل delegation به جای subclass به مراتب بهتر است. تغییر در اینجا بسیار راحت‌تر از راه حل subclass است.
- به جای استفاده از switch و شرط‌های متعدد در متد برای فراخوانی الگوریتم و بررسی شرایط، از این الگو استفاده می‌کنیم که کد را بهتر، ساده و منعطف می‌کند.
- انتخاب‌هایی از implementationهای مختلف داریم. پیاده‌سازی‌ای که در شرایط فعلی به نفع ما است را می‌توانیم استفاده کنیم. در واقع رفتار و نتیجه‌ی یکسان را نیاز داریم اما بسته به شرایط باید انتخاب کنیم که چه optimizationی انجام دهیم.

اثرات منفی

- درست است که dip بین context و strategy داریم، اما کلاینت یا configurer ما باید strategy و زیر کلاس‌ها را بشناسد و در نتیجه coupling بالا است.
- یک communication overhead بین strategy و context به دلیل indirection به وجود آمده است. در حالت before یک کلاس داشتیم که الگوریتم را اجرا می‌کرد اما پس از اعمال الگو، یک message passing اضافی نیز داریم
- overhead نیز در تعریف interface ممکن است داشته باشیم به علت در نظر گرفتن بیشینه‌ی پارامترهای اضافی در تعریف متدها. البته در اکثریت مواقع می‌توان با کمک دید غیر مانا از strategy به context و پاس دادن آن مشکل را حل کرد. هر چند ممکن است مشکل کامل حل نشود و برخی پارامترها در context نباشد و آن‌ها را نیز باید پاس دهیم.
- تعداد اشیا زیاد می‌شود. در حالت before یک شی وظیفه‌ی اجرا را داشتیم اما حالا به کمک delegation انجام می‌دهیم. همچنین ممکن است یک context متدهای مختلفی داشته باشد که در هر کدام strategyهای مختلفی استفاده شده باشد.

مقایسه

مشکل وابستگی پیکربند به کلاس‌های concrete تقریبا در همه‌ی الگوها دیده می‌شود. همچنین اضافه شدن یک سطح indirection در هر سه الگو دیده می‌شود. باقی موارد نیز به خوبی در توضیحات بالا آورده شده است.

Encapsulation

Bridge

این الگو کاملا encapsulation را برقرار می‌کند. باید توجه کرد که در این الگو کلاینت تنها در سطح بالا به abstraction دید دارد و abstraction نیز در سطح بالا به implementor دید دارد در نتیجه دید به کلاس‌های concrete نیست در نتیجه encapsulation نقض نمی‌شود.

Command

در این الگو invoker و receiver از یکدیگر خبر ندارند. همچنین command نیز یک نگاه سطح بالا به receiver دارد و invoker نیز interface مربوط به command را می‌بیند. تنها موردی که می‌تواند encapsulation را نقض کند، دید مستقیم کلاینت به ConcreteCommandها را دارد. اما نسبت به حالت پیشین، این الگو encapsulation را به مراتب بهینه کرده است.

Strategy

در این الگو نیز encapsulation نقض نمی‌شود چرا که نگاه context به استراتژی یک نگاه سطح بالا و از طریق interface است و در نتیجه، با فراخوانی متدهای strategy، رفتار انجام می‌شود و داده‌های آن بروز بیرونی پیدا نمی‌کنند. در نتیجه به خوبی encapsulation رعایت می‌شود.

مقایسه

هر سه الگو به خوبی encapsulation را رعایت می‌کنند و هیچ کدام آن را نقض نمی‌کنند.

انتشار تغییرات

Bridge

این الگو به خوبی از انتشار تغییرات میان ساختارهای توارثی و کلاینت جلوگیری می‌کند. کلاینت از طریق interface به abstraction دید دارد. همچنین، میان دو ساختار توارثی به خوبی dip برقرار است و در نتیجه، کلاینت از تغییرات abstraction و implementorها منتزع است و همچنین abstraction نیز نسبت به تغییرات implementor مصون می‌ماند.

Command

در این الگو تغییرات receiver و invoker در یکدیگر منتشر نمی‌شود چرا که به وسیله ی command از یکدیگر منتزع شده‌اند. به علت دید سطح بالای command به invoker و receiver نیز، تغییرات میان آن‌ها نیز منتشر نمی‌شود. مشکل اصلی این الگو در انتشار تغییرات command به کلاینت است به علت آن‌که کلاینت باید دید کاملی به command داشته باشد تا از آن instantiate کرده و پیکربندی الگو را انجام دهد.

Strategy

Context دید سطح بالا به strategyها دارد و در نتیجه تغییرات ConcreteStrategyها به context به دلیل برقرار بودن dip منتشر نمی‌شود. مشکل اصلی پیکربندی (در اینجا کلاینت) است که به تمام ConcreteStrategyها دید دارند و در نتیجه به علت coupling بالا میان آن‌ها و زیرکلاس‌های strategy، تغییرات در آن‌ها منتشر می‌شود.

مقایسه

در الگوهای Strategy و Bridge، به خوبی میان ساختارهای توارثی، dip برقرار است و در نتیجه تغییرات در میان آن‌ها منتشر نمی‌شود. در command نیز این تغییرات میان کلاس‌های invoker و receiver و command نیز منتشر نمی‌شود اما تغییر command روی client تاثیر می‌گذارد.

میزان استفاده از منابع سیستمی

Bridge

در این الگو، با جداسازی اشیا implementation از abstraction، مصرف حافظه نسبت به قبل افزایش می‌یابد چرا که به ازای هر abstraction باید یک implementation حداقل داشته باشیم تا الگو محقق شود و در نتیجه منابع حافظه‌ی بیشتری مصرف می‌کند.

Command

در این الگو، امکان افزایش شدید مصرف منابع وجود دارد. چرا که به ازای هر فراخوانی در حالت پیشین الگو، در حالت پسین باید حداقل یک شی جدید تعریف کنیم. در نتیجه ممکن است اشیا متعددی برای کاربردهای مختلف تعریف و ساخته شود و مشکلات حافظه به وجود آید. همچنین باید توجه داشت که برای عملیاتی چون log یا undo، باید اطلاعات اضافه‌تری را داخل ساختار ثبت کرد و در نتیجه منابع سیستمی به طرز چشم‌گیری می‌تواند افزایش یابد.

Strategy

در این الگو نسبت به حالت پیشین، لازم است تا با delegation، شی پیاده‌سازی کننده‌ی الگوریتم به context داده شود. در نتیجه بسته به تعداد الگوریتم‌ها و همچنین تعداد فرایندهای الگوریتم اشیا مختلفی می‌تواند ایجاد شود که مصرف حافظه را نسبت به گذشته افزایش می‌دهد.

مقایسه

هر سه الگو از منظر استفاده از منابع سیستمی و به خصوص حافظه، وضعیت را بدتر می‌کنند. بخصوص command که پتانسیل افزایش بیش از حد اشیا سیستمی را دارد. در دو الگوی دیگر می‌توان گفت این افزایش اشیا کنترل می‌شود.

استفاده از شی جعلی

Bridge

در این الگو، اشیا implementor پیش از اعمال الگو وجود نداشتند و همچنین در دامنه‌ی مسئله نیز نمی‌توان آن‌ها را پیدا کرد. این اشیا جعلی ایجاد می‌شوند تا cohesion بالا برود و انعطاف ساختار بیشتر شود. در نتیجه در این الگو نیز ساختار implementor یک ساختار جعلی است.

Command

شی command، در قلمرو مسئله تعریف نشده است. در حالت پیشین نیز به کمک فراخوانی در زمان اجرا، انجام می‌شود. اما پس از اعمال الگو این شی جعل می‌شود تا پیام را نمایندگی کند و کارهای مربوط به اجرای یک دستور را انجام دهد.

Strategy

در این الگو نیز ساختار strategy و اشیا مربوط به الگوریتم، در دامنه‌ی مسئله نیستند و در دل context قرار داشته‌اند. در نتیجه برای بهبود انعطاف و همچنین cohesion و امکان گسترش‌پذیری، این اشیا جعل می‌شوند.

مقایسه

هر سه الگو برای رسیدن به هدف خود، از شی جعلی که در قلمرو مسئله دیده نمی‌شود بهره می‌برند.

سادگی پیاده‌سازی

Bridge

این الگو از منظر پیاده‌سازی ساده است. کافی است مفاهیم implementation و abstraction را در دامنه‌ی راه حل پیدا کنیم و در نتیجه با تعریف interface یا abstract کلاس‌های هر کدام، بسته به نیاز خود، آن‌ها را گسترش دهیم. زبان‌های شی گرا همگی امکان پیاده‌سازی آن را به ما می‌دهند و پیچیدگی ندارد.

Command

این الگو بسیار رایج است و تقریبا در اکثر سیستم‌های امروز به کار می‌رود. پیاده‌سازی آن نیز بسیار ساده است. کافی است تا پیام را در یک شی encapsulate کرده و با ایجاد دید مانا به یک receiver و تعریف command processor، این الگو محقق می‌شود.

Strategy

برای پیاده‌سازی کافی است تا موقعیت آن را در سیستم پیدا کنیم. برای مثال الگوریتم‌هایی که یک وظیفه‌مندی را به روش‌های مختلف اجرا می‌کنند و در سیستم مورد نیاز هستند را شناسایی می‌کنیم. سپس با تعریف context و strategy به عنوان interface یا abstract class به راحتی می‌توانیم آن‌ها را گسترش دهیم. یک دید مانا از context به الگوریتم باید برقرار کنیم و در صورت لزوم می‌توانیم در تعریف strategy از context به عنوان ورودی استفاده کنیم. پیاده‌سازی آن در زبان‌های شی گرا به سادگی انجام پذیر است.

مقایسه

هر سه الگو به سادگی قابل پیاده‌سازی هستند.

موارد کاربرد

Bridge

- می‌خواهیم یک انقیاد (binding) دائمی میان abstraction و implementation وجود نداشته باشد و برای مثال بخواهیم در زمان اجرا implementation را عوض کنیم. در این حالت باید implementation را از دل abstraction خارج و وارد یک فضای توارثی دیگر کرد.
- وقتی می‌خواهیم abstraction و implementation مستقلا گسترش پذیر باشند abstraction و implementationهای مختلفی داشته باشیم. در اینجا اگر subclassing انجام دهیم، وقتی ساختارها جدا می‌شود به راحتی انجام می‌شود.
- می‌خواهیم پیاده‌سازی abstraction تاثیری روی کلاینت نگذارد. کلاینت تنها به اینترفیس abstraction وابسته است و implementation جدا است. کاملا استقلال وجود دارد.
- یک کاربرد مخصوص c++ است. یک idiom است. الگوی سطح زبان cheshire cat. این یک گربه‌ای است که implementation آن از بین میرود ولی interface آن باقی می‌ماند. در cpp به آن Pimpl هم گفته می‌شود. کاربرد ان در cpp این است که وقتی یک header file به یک cpp اشاره می‌شود، اگر cpp عوض شود، کلاینت‌های header file نیز باید مجددا کامپایل شوند. برای حل این مشکل، یک header file پرایوت میان header file اصلی و cpp قرار می‌دهند و در نتیجه با یک سطح indirection این وابستگی را حذف می‌کنند. شباهت زیادی به bridge دارد اما عینا یکی نیست.
- وقتی می‌خواهیم یک implementation را به اشتراک بگذاریم و کلاینت متوجه نشود.

Command

- وقتی که می‌خواهیم اشیا را با اکشن‌ها پارامتریزه کنیم و برای آن‌ها درجه‌ی آزادی تعریف کنیم که بتوانیم اکشن آن را عوض کنیم.
- در زمان اجرا می‌شود صف درست کرد، در زمان اجرا آن‌ها را مشخص کرد کرد و ترتیب مورد نظر را روی آن‌ها اعمال کرد.
- می‌شود از undo پشتیبانی کرد. به این صورت که حالت پیش از اجرا را ثبت می‌کنیم (برای مثال با یک پشته) و بعد کامند رو وارد پشته کرده و آن را پاپ می‌کنیم و undo می‌کنیم.
- می‌شود تغییرات و پیام‌ها را log کرد. در این صورت persist شده و در نتیجه می‌شود پس از آن‌که crash رخ داد از جایی که کامندها اجرا شده، restore یا reaply کرد (به کمک load و store در خود کامند)
- میشه سیستم را حول مجموعه‌ای از operationهای درشتدانه که از operationهای ریزدانه تشکیل شده اند تشکیل داد (در واقع macro command). بسیاری از سیستم‌های بانکی را می‌شود به همین

شکل درست کرد. سطوح مختلفی از ترکیب command را می‌شود ایجاد کرد و تمام ترکیب‌ها هم لزومی به پیش‌بینی ندارند بلکه با پویایی بالا و در زمان اجرا درست می‌شوند و ساختار مجموعه commandها را تشکیل می‌دهند.
- کامند یک سطح indirection میان فرستنده و گیرنده که coupled بودند ایجاد می‌کند و در نتیجه coupling میان کلاینت و سرور کم می‌شود.

Strategy

- تعداد زیادی کلاس داریم که در رفتار با یکدیگر متفاوتند. استراتژی این امکان را می‌دهد که بشود یک کلاس داشت و بتوان در زمان اجرا رفتار را configure کرد. بجای اینکه کلاس‌های مختلفی داشته باشیم که به ازای الگوریتم‌های مختلف ساخته شده اند، یک کلاس داریم و آن را با رفتار مورد نظر configure می‌کنیم.
- کاربرد اصلی آن این است که انواع مختلفی از الگوریتم را نیاز داریم مثلا الگوریتم‌هایی که زمان یا فضا را بهینه‌سازی می‌کنیم و می‌خواهیم آن را در زمان اجرا تغییر دهیم.
- یک الگوریتم از داده‌هایی استفاده می‌کند که کلاینت نباید از آن‌ها بداند. در این حالت قرار دادن آن‌ها در context درست نیست. وقتی که آن را در strategy قرار می‌دهیم، دادگان از دید کلاینت محفوظ می‌ماند. باید توجه کرد که کلاس strategy بسیار کوچک است و رفتار و داده‌های زیادی ندارد. اگر همه‌ی رفتار و attributeها را در context قرار دهیم، context بزرگ می‌شود اما اگر رفتارهای تکمیلی و attributeهای مربوط به یک الگوریتم را در strategy قرار می‌دهیم و به این صورت context را سبک می‌کنیم و جزئیات الگوریتم را از دید کلاینت محفوظ می‌کنیم.
- یک کلاس که تعداد زیادی رفتار را از خود بروز می‌دهد که خود را به صورت یک switch یا تعدادی conditional statement نشان می‌دهند. مثلا یک operation را نگاه می‌کنیم که شرایط را می‌سنجد و رفتار مربوطه را بروز می‌دهد. این کار مطلوبی نیست. باید توجه داشت که strategy در سطح یک متد می‌تواند باشد. در سطح یک operation باشد. رفتار اما می‌تواند چند رفتار را در بر بگیرد. در نتیجه در حالت before به صورت switchهای سنگین بوده‌اند اما در حال حاضر بر اساس state شی یا ... تغییر رفتار دهد.

مقایسه

کاربردهای این سه الگو تقریبا با یکدیگر ارتباطی ندارند و هر یک در شرایط مورد نیاز می‌توانند به کار روند.

الگوهای مرتبط

Bridge

از جهت به وجود آمدن دو ساختار توارثی موازی، می‌توان builder را به آن مشابه دانست چرا که در آن‌جا director و builder می‌توانند مستقل از یکدیگر رشد کنند و در عین حال director برای آفرینش از builder استفاده کند. در اینجا نیز implementorها که کارهای ریزدانه‌تر را نسبت به کارهای درشت‌دانه و سطح بالای abstraction انجام می‌دهند، از Abstraction جدا شده‌اند.
از طرفی برای ساخت implementorها توسط پیکربند، می‌توان از یک ساختار موازی برای آفرینش مطابق الگوی abstract factory استفاده کرد.

Command

از الگوی composite برای ساخت commandهای ترکیبی یا macro command می‌توان استفاده کرد. از الگوی memento برای نگهداری الگوی command می‌توان استفاده کرد.
همانطور که پیش‌تر گفته شد، یکی از کاربردهای این الگو undo کردن فرمان است. این کار می‌تواند به کمک memento انجام شود.
اگر بخواهیم یک command را به صورت مکرر ایجاد کنیم، می‌توانیم از prototype استفاده کنیم.
شباهت زیادی به visitor دارد. چرا که هر المان خود را در اختیار visitor قرار می‌دهد تا یک عملیات انجام شود. Command نیز receiver را دریافت می‌کند تا یک عملیات روی آن انجام دهند.

Strategy

با الگوی state در ارتباط است چرا که ساختار آن‌ها عینا شبیه یکدیگر است با این تفاوت که هدف آن‌ها با یکدیگر متفاوت است و استراتژی به رفتار متفاوت و state به تغییر حالت مرتبط می‌شوند.
با الگوی bridge شباهت دارد. از منظر ساختار توارثی می‌توان شباهت را دید. چرا که در هر دو کلاینت از یک ساختار کاملا منتزع است و یک ساختار را از طریق واسط می‌بیند. همچنین بخشی از عملیات به ساختار توارثی جدید داده شده است.

مقایسه

این سه الگو از منظر انتساب رفتار خود به یک شی دیگر، شباهت جدی به یکدیگر دارند. از طرفی الگوهای دیگری نیز در این موارد به آن‌ها اشاره شد که مورد شباهت این الگوها نبودند.

OCP

Bridge

در این الگو ocp به بهترین شکل ممکن برقرار است. اساسا پس از این الگو، می‌توان abstraction و implementation را مستقل از یکدیگر و به صورت مجزا گسترش داد بدون آن‌که ساختار دیگر نیاز به تغییر داشته باشد. کلاینت نیز از طریق interface و با abstraction در ارتباط است و به علت برقراری dip، با گسترش آن‌ها، تغییری در کلاینت نیاز نخواهد شد مادامی که interface تغییر کند. در نتیجه ocp برقرار است.

Command

در این الگو، می‌توان گفت تا حدی ocp برقرار است. هدف این الگو جداسازی invoker از receiver با ایجاد یک شی میانی بوده است. در حالت پیشین invoker مستقیما از receiver تاثیر می‌پذیرفته است. اما حالا فراخوانی توسط یک شی میانی که واسط command را پیاده‌سازی می‌کند انجام می‌شود. در نتیجه تغییرات در command به invoker منتشر نمی‌شوند.
همچنین command از طریق واسط می‌تواند receiver خود را بشناسد و در نتیجه، dip بین command و invoker و command و receiver برقرار است و گسترش زیرکلاس‌ها، کلاس دیگر را تغییر نمی‌دهد و تنها تغییر در interface روی آن‌ها اثرگذار است.
باید توجه داشت که کلاینت در این الگو نقش یک پیکربندی را ایفا می‌کند و به همین دلیل به زیرکلاس‌های command و invoker و receiver می‌تواند دید داشته باشد و در نتیجه ocp نسبت به آن برقرار نیست.

Strategy

Dip میان context و strategy برقرار است چرا که دید context به strategyها از طریق واسط تحقق می‌گردد تا بتوان در زمان اجرا، بسته به شرایط، استراتژی‌های مختلف را برای context پیکربندی کرد و در نتیجه، ocp نیز برقرار است. گسترش context و strategy موجب تغییر در دیگری نمی‌شود مادامی که واسط strategy تغییر نکرده باشد.
اما کلاینت (که در این الگو همان پیکربند است)، بایستی به ConcreteStrategyها دید داشته باشد و از آن‌ها instantiate کند و در نتیجه در رابطه با اون این قاعده برقرار نیست که طبیعی نیز هست.

مقایسه

در هر سه الگو، ocp برقرار است. در مورد strategy و command به علت یکی بودن پیکربندی و کلاینت، پیکربند پس از گسترش دیگر کلاس‌ها ممکن است تغییر کند.

LSP

Bridge

در این الگو، زیرکلاس‌های implementation و abstraction همگی با پدرشان قابل جایگزین هستند (و باید باشند) و در نتیجه این الگو از این قاعده پیروی می‌کند. در نتیجه رابطه‌ی is a میان زیرکلاس‌های این دو ساختار توارثی و پدرشان برقرار است.

Command

در این الگو نیز تمامی ConcreteCommandها با command بایستی قابل جایگزین باشد و رابطه‌ی is a میان ConcreteCommand و Command برقرار است و این قاعده رعایت می‌شود.

Strategy

در این الگو نیز زیرکلاس‌های strategy همگی رابطه‌ی is a با کلاس پدر دارند و در نتیجه این الگو نیز این قاعده را رعایت می‌کند.
هر چند باید دقت کرد که برخی اوقات ممکن است operationها به گونه‌ای تعریف شوند که ورودی‌های مورد نیاز برای برخی ConcreteStrategyها با یکدیگر متفاوت باشد و در نتیجه مجبور می‌شویم یا اجتماع ورودی‌ها را در نظر بگیریم و یا context را ورودی بدهیم. اما این مورد در برقراری رابطه‌ی is a میان ConcreteStrategyها برقرار است.

مقایسه

در هر سه الگو این قاعده رعایت می‌شود.

DIP

Bridge

در این الگو، قاعده‌ی dip به بهترین نحو رعایت می‌شود و کلاینت، abstraction و implementationها از طریق واسط با یکدیگر سر و کار دارند و در نتیجه دید Concrete به Concrete در این الگو وجود ندارد. هر چند پیکربند لازم است تا کلاس‌های Concrete را ببیند اما به طور کلی این الگو dip را رعایت می‌کند.

Command

میان invoker ،command و receiver، این قاعده برقرار است چرا که رابطه‌ی میان آن‌ها از طریق واسط برقرار می‌شود. اما کلاینت که نقش پیکربند را دارد و لازم است زیرکلاس‌های آن‌ها را دیده و instantiate کند و در نتیجه میان پیکربند و ساختارها dip برقرار نیست. اما به طور کلی این الگو نیز dip را رعایت می‌کند.

Strategy

در این الگو نیز دید میان context به strategyها از طریق واسط صورت می‌گیرد و در نتیجه dip برقرار است و دید Concrete به Concrete وجود ندارد. اما باید توجه داشت که مانند command، دید پیکربند به strategyها یک دید concrete است تا بتواند در زمان اجرا پیکربندی مناسب را تغییر دهد. اما در حالت کلی این الگو نیز dip را رعایت می‌کند.

مقایسه

در هر سه الگو این قاعده به خوبی رعایت می‌شوند هر چند در هر سه الگو، پیکربند دید concrete به دیگر ساختارها را دارد اما در حالت کلی این قاعده را رعایت می‌کنند.

ISP

Bridge

اساسا هدف این الگو، جداسازی implementation از abstraction بوده است. در حالت پیشین این دو توسط یک interface محقق می‌شدند اما پس از اعمال الگو برای جداسازی مسئولیت‌ها و وظایف، implementation از abstraction جدا می‌شود و وظایف مربوط به پیاده‌سازی به واسط آن محول می‌شود. در نتیجه، به خوبی کارها شکسته شده و جداسازی interfaceها صورت می‌گیرد و کارهای پیاده‌سازی در ذیل واسط abstraction تعریف نمی‌شود که نشان از رعایت این قاعده در این الگو دارد.

Command

در این الگو نیز این قاعده رعایت شده است. پیش از این وظیفه‌ی انتقال پیام از طریق یک فراخوانی مستقیم انجام می‌شد. اما با تعریف یک interface فرمان (command)، فراخوانی از طریق ایجاد یک شی میانجی انجام می‌شود. در نتیجه وظایف مربوط به undo کردن و ... که می‌تواند پس از پیاده‌سازی command انجام شود همگی از روی دوش سرویس گیرنده برداشته می‌شود و به command محول می‌شود و در نتیجه می‌توان تا حدی نمود این قاعده را در این الگو دید.

Strategy

در این الگو نیز در حالت پیشین لازم بود با تعریف methodهای مختلف در context، این کلاس را سنگین کرده و سپس با switch caseهای بزرگ، استراتژی‌های مختلف را اعمال کرد. اما پس از اعمال الگو، وظیفه‌ی پیاده‌سازی الگوریتم از context جدا شده و به strategy محول می‌شود و در نتیجه isp به خوبی در این الگو دیده می‌شود.

مقایسه

در دو الگوی bridge و strategy، به صورت واضحی این قاعده نمود دارد. در الگوی command نیز می‌توان گفت هر چند تمرکز اصلی الگو بر سبک کردن وظایف واسط نیست اما این قاعده را تا حدی رعایت می‌کند.

CRP

Bridge

در این الگو، crp در بهترین شکل خود نمود دارد. در حالت پیشین، inheritance تنها راه اعمال پیاده‌سازی‌های مختلف برای abstractionهای مختلف لازم بود و در نتیجه برای پیاده‌سازی‌های مختلف، زیرکلاس‌های مختلف آن با abstraction ترکیب شود. در نتیجه زوج‌های abstraction implementation ذیل یک ساختار توارثی ترکیب می‌شدند و به طور متعدد بوجود می‌آمدند که منجر به انفجار ترکیبی می‌توانست شود.
پس از اعمال الگو، از طریق delegate کردن یک شی implementation به abstraction، می‌توان ترکیبات متنوعی از abstraction و implementation داشت بدون آن که لازم به تعریف زیرکلاس‌های جدید کرد و در نتیجه هر دو ساختار مستقل از یکدیگر رشد می‌کنند. در نتیجه این الگو یکی از بهترین مصادیق این قاعده است.

Command

در این الگو نمی‌توان گفت delegation بر ساختار توارثی ترجیح پیدا کرده است چرا که ساختار توارثی منجر به ایجاد رابطه‌ی delegation میان command و invoker نشده است بلکه اهداف الگو مانند undo و ایجاد indirection و ... منجر به delegate شدن command به invoker شده است. در نتیجه crp در این الگو رعایت شده است چرا که الگو از طریق delegation و در زمان اجرا محقق می‌شود اما ساختار توارثی خاصی پیش از آن نبوده است. اما delegation در این الگو نمود دارد.

Strategy

این الگو نیز مصداق مناسبی برای این قاعده است. در حالت پیشین می‌توانستیم برای context زیرکلاس‌ها یا operationهای مختلف تعریف کنیم که کار ما را برای پیکربندی در زمان اجرا سخت می‌کرد. اما پس از اعمال الگو، delegate کردن یک استراتژی به context می‌تواند الگوریتم را تغییر دهد و پیکربندی شود و در نتیجه، هدف الگو محقق می‌شود. پس این الگو اساسا از این قاعده به خوبی بهره برده است.

مقایسه

دو الگوی strategy و bridge به خوبی از این قاعده بهره‌مند شده اند و الگوی command نیز در زمان اجرا و به کمک delegation محقق می‌شود و در نتیجه هر سه الگو این قاعده را رعایت می‌کنند.

PLK

Bridge

در این الگو دید ترایایی میان کلاینت و هیچ یک از دو شی دیگر صورت نمی‌گیرد. برای مثال implementation از طریق abstraction در اختیار کلاینت قرار نمی‌گیرد در نتیجه این قاعده به خوبی رعایت شده چرا که message chain در آن دیده نمی‌شود.

Command

در این الگو نیز این قاعده به بهترین شکل رعایت شده است. Invoker، فرمان را اجرا می‌کند. Command، در داخل خود receiver را نگهداری می‌کند اما آن را در اختیار invoker قرار نمی‌دهد بلکه خود انتقال پیام را انجام می‌دهد. همانطور که می‌بینید در این الگو امکان خطا و message chain وجود داشت اما به خوبی این الگو از آن محافظت کرده و دید ترایا دیده نمی‌شود و این قاعده رعایت می‌شود.

Strategy

در این الگو نیز دید ترایا دیده نمی‌شود و message chain وجود ندارد. در نتیجه این الگو نیز این قاعده را رعایت می‌کند.

مقایسه

هر سه الگو از این قاعده پیروی می‌کنند.

مقایسهٔ الگوهای Bridge، Command و Strategy بر اساس grasp

Information Expert

Bridge

در این الگو می‌توان گفت این قاعده رعایت شده است اما ممکن است این قاعده نقض شود. با توجه به اینکه implementation را از abstraction جدا کرده‌ایم، اگر داده‌هایی که implementation نیاز دارد تا رفتار مناسب را بروز دهد، در اختیار abstraction باشد، در این حالت رفتار از داده جدا شده است.
اما به نظر می‌رسد در شکل کلی این الگو، این مسئله دیده نمی‌شود و در نتیجه، فرض می‌کنیم که داده و رفتار مورد نیاز در هر یک از اشیا وجود دارند.

Command

در این الگو داده و رفتار در کنار یکدیگر قرار داده شده است. Command داده مورد نیاز خود برای انتقال فرمان به recevier را دارد. باقی اشیا نیز رفتار و داده را در کنار یکدیگر قرار می‌دهند و در نتیجه این الگو قاعده‌ی information expert را رعایت می‌کند.

Strategy

در این الگو، رفتار از داده جدا می‌شود و این قاعده نقض می‌شود. البته این اتفاق بنا به مصلحتی انجام شده است که اولویت بالاتری داشته است (یعنی جدایی این رفتار منجر به منطعف شدن و پیکربندی استفاده از الگوریتم‌های مختلف شده است). به همین دلیل، می‌بینیم که داده‌های مورد نیاز strategey در اختیار context قرار دارد و حتی به علت این دور بودن داده از رفتار، مجبور می‌شویم متدهایی با ورودی‌های زیاد یا با ورودی context تعریف کنیم که از جهاتی مناسب نیست. در نتیجه این الگو از این قاعده پیروی نمی‌کند.

مقایسه

الگوی strategy این الگو را نقض می‌کند. الگوی bridge امکان نقض کردن این الگو را دارد اما command از این الگو استفاده می‌برد.

Creator

Bridge

در این الگو وظیفه‌ی ساخت اشیا بر عهده‌ی یک پیکربند ثالث است که abstraction و implementation مناسب را در اختیار کلاینت قرار می‌دهد. در نتیجه این الگو از این قاعده به درستی پیروی نمی‌کند. برای مثال abstraction یک دید مانا به implementation دارد در نتیجه اولویت بالاتری نسبت به پیکربند برای ساخت آن دارد اما از آنجایی که نوع پیاده‌سازی نمی‌تواند توسط implementation تعریف شود، این وظیفه برعهده پیکربند قرار گرفته است که اطلاعات initialization را دارد و در اولویت پنجم است. در نتیجه این الگو نقض می‌شود.

Command

در این الگو، وظیفه‌ی ساخت command بر عهده‌ی کلاینت است که در اولویت پنجم قرار دارد اما invoker یک association به command دارد و در نتیجه اولویت بالاتری نسبت به کلاینت برای ساخت آن را دارد. در نتیجه این الگو در command استفاده نشده است.

Strategy

در این الگو وظیفه‌ی ساخت استراتژی نیز بر عهده‌ی کلاینت است که یک پیکربند محسوب می‌شود اما اولویت بالاتر بر با context است که association به strategy دارد. در نتیجه creator در این الگو نقض می شود و در این الگو استفاده نشده است.

مقایسه

هر سه الگو به کمک یک پیکربند افرینش را انجام می‌دهند در حالی که اولویت بالاتری (رابطه‌ی association) وجود دارد در نتیجه این الگو را نقض می‌کنند.

Low Coupling

Bridge

همانطور که بالاتر در بخش مقایسه گفته شد، در این الگو به کمک جداسازی implementation از abstraction این امکان را به وجود می‌آوریم که این دو ساختار مستقل از یکدیگر و بدون وابستگی بتوانند رشد کنند.

میان این دو ساختار dip کاملا برقرار است یعنی تنها از طریق interfaceها، abstraction به implementation دید دارد و به کلاس‌های concrete دید ندارد. همچنین کلاینت نیز تنها به abstraction دید دارد و در نتیجه کاملا منتزع از جزئیات پیاده‌سازی است و dip کاملا برقرار است. تنها مشکل، پیکربند است که باید abstraction را با implementation مناسب پیکربندی کند که برای این کار لازم است تا تمام زیرکلاس‌های implementation را بشناسد. در مجموع اما باید گفت که این الگو، coupling را کاهش می‌دهد و از وابستگی کمی میان اشیا و ساختارها برخوردار است.

Command

این الگو چندان بر روی coupling تمرکز ندارد. اما یک سطح indirection میان receiver و invoker ایجاد می‌کند. در واقع در حالت پیشین الگو، invoker مستقیما به receiver دید داشته است اما حالا این دید از بین رفته و به جای آن command را می‌بیند. ولی در حالت پسین نیز command باید مستقیما به receiver دید داشته باشد. در نتیجه دید میان invoker و receiver به دید میان command و هر دوی آن‌ها تبدیل شده است. اما dip میان command و invoker و همچنین command و receiver برقرار است.
نکته‌ی حائز اهمیت این که میان client و command، با توجه به instantiation و پیکربندی command برای invoker توسط کلاینت، dip برقرار نیست. در نتیجه coupling میان کلاینت و command شدید است. در حالت پیشین client تنها با invoker در ارتباط بود و dip نیز می‌توانست برقرار باشد (بسته به تعریف interface مربوط به invoker) اما در این حالت، کلاینت به command وابستگی شدید خواهد داشت. اما این وابستگی در پیکربندی‌ها دیده می‌شود و می‌توان نادیده گرفت.
در مجموع می‌توان گفت command نسبتا از coupling پایینی برخوردار است.

Strategy

در این الگو، میان context و strategy به صورت کامل dip برقرار است و در نتیجه coupling در وضعیت بسیار خوبی است. اما پیکربند که می‌تواند همان کلاینت، اکتور خارجی و یا یک پیکربند سیستمی و ... باشد، لازم است

تا نسبت به کلاس‌های ConcreteStrategy دید داشته باشد و در نتیجه dip نقض می‌شود. اما در مجموع dip پیکربند را می‌توانیم نادیده بگیریم و در نتیجه وضعیت این الگو از نظر coupling مناسب است.

مقایسه

این الگوها از نظر coupling بسیار شبیه یکدیگر هستند. می‌توان گفت وضعیت پسین آن‌ها از نظر coupling بسیار مناسب است و dip تا حد خوبی برقرار می‌شود و در نتیجه از coupling پایینی برخوردارند.

High Cohesion

Bridge

این الگو cohesion را بهبود می‌دهد. چرا که پیاده‌سازی را به کلاس‌های متخصص پیاده‌سازی می‌سپارد و در نتیجه مسئولیت‌های اضافی را از abstraction جدا کرده و به پیاده‌سازی می‌دهد و دو ساختار پیاده‌سازی و انتزاع را از یکدیگر جدا می‌کند. می‌توان گفت اساس این الگو برای بالا بردن cohesion است و در نتیجه، از نظر افزایش cohesion این الگو بسیار مناسب است.

Command

این الگو نیز از cohesion مناسبی برخوردار است. Commandها به جای آنکه به صورت مستقیم و از طریق فراخوانی میان دو کلاس صورت پذیرد، به کمک جداسازی و سپردن انتقال پیام به یک کلاس، یک هویت برای انتقال پیام در نظر می‌گیرد و در نتیجه می‌توانیم بگوییم cohesion برقرار است و یک کلاس متخصص انتقال فرمان ایجاد شده است.

Strategy

هدف اصلی این الگو پیاده‌سازی‌های رفتارهای مختلف و الگوریتم‌های مختلف از context است و این رفتارها یا الگوریتم‌ها را به یک ساختار مجزا محول می‌کند. در واقع وظیفه‌ی پیاده‌سازی الگوریتم را از context جدا کرده و به strategy محول می‌کند و در نتیجه seperartion of concerns را بهبود داده و به افزایش cohesion کمک شایانی می‌کند.

مقایسه

هر سه الگو به افزایش cohesion کمک زیادی می‌کنند و از این منظر شباهت بسیاری دارند. در واقع جداسازی و ایجاد اشیا و ساختارهای متخصص و جداکردن مسئولیت‌ها یکی از کارهای مهم هر سه الگو است.

Controller

Bridge

نمی‌توان میانجی و کارچرخان را در این الگو دید. در واقع وظیفه‌ی کارچرخانی چندان در این الگو دیده نمی‌شود و در نتیجه controller در این الگو به کار نرفته است.

Command

Command نیز وظیفه‌ی کارچرخانی را نمی‌توان دید و در نتیجه در رابطه با این الگو، از کنترلر استفاده نشده است.

Strategy

در این الگو نیز controller محل بحث نیست و کارچرخانی دیده نمی‌شود.

مقایسه

هر سه الگو از controller درون خود استفاده نمی‌کنند.

Polymorphism

Bridge

به خوبی در این الگو از polymorphism استفاده می‌شود. در واقع هدف اصلی الگو، گسترش مجزای پیاده‌سازی و انتزاع است. این دو ساختار را از یکدیگر جدا کردیم در نتیجه با extend کردن آن‌ها و پیاده‌سازی متدها به صورت چندریختی در فرزندان، الگو تحقق یافته و می‌توان abstraction و implementationهای متنوع تعریف کرد.

Command

در این الگو نیز از polymorphism استفاده می‌شود. ConcreteCommandهای مختلف می‌توانند ذیل Command تعریف شوند و فرمان‌های مختلفی invoke شوند در نتیجه به خوبی از این الگو استفاده می‌شود.

Strategy

در این الگو نیز از polymorphism به خوبی استفاده می‌شود. الگوریتم‌های مختلف که از جهت کارایی و ... با یکدیگر متفاوت هستند، به کمک polymorphism در زیرکلاس‌های strategy و در ذیل methodها تعریف می‌شوند و در نتیجه اساسا این الگو به کمک polymorphism تحقق می‌یابد و می‌توان الگوریتم‌ها را تعریف کرد.

مقایسه

در هر سه الگو از polymorphism بهره برده می‌شود.

Indirection

Bridge

در این الگو شی میانجی که مسئولیت واسطه گری را برعهده داشته باشد دیده نمی‌شود. هر چند شاید در نگاه اول به نظر برسد که abstraction چنین وظیفه‌ای برعهده دارد، اما پیش از اعمال الگو نیز ارتباط کلاینت با abstraction بوده و پس از آن صرفا بخشی از رفتارها به implementation محول شده و انتقال پیام غیر مستقیم نیز دیده نمی‌شود. پس از این الگو استفاده نشده است.

Command

در این الگو، command یک شیئی است که میان invoker و receiver یک میانجی محسوب می‌شود و موجب می‌شود تا فراخوانی و انتقال پیام به صورت مستقیم صورت نگیرد بلکه به صورت غیر مستقیم و از طریق command پیام منتقل شود. در نتیجه این الگو از indirection استفاده می‌کند.

Strategy

این الگو نیز از یک شی میانجی که واسطه‌گری انتقال پیام داشته باشد را ندارد. بلکه وظایف context و پیاده‌سازی الگوریتم‌ها است که از context جدا شده در نتیجه انتقال پیام غیر مستقیم دیده نمی‌شود و می‌توان گفت در این الگو نیز indirection وجود ندارد. شاید در نگاه اول اینگونه به نظر برسد که context واسطه‌ی انتقال پیام از کلاینت به strategy است، اما اساسا مطابق اهداف الگو و همچنین از نظر رفتاری می‌دانیم که چنین چیزی صحیح نیست و صرفا بخشی از رفتاری که برعهده‌ی context بوده و توسط context انجام می‌شده، به strategy محول شده و مسئله جداسازی کلاینت از strategy اساسا از مورد بحث نیست.

مقایسه

تنها الگوی command از این الگو بهره می‌برد.

Pure Fabrication

Bridge

در این الگو نیز همانطور که قبلا نیز اشاره شد، اشیا implementor اساسا در قلمرو مسئله وجود نداشته اند و برای گسترش مستقل از هم ساختارهای implementor و abstraction و همچنین بالا بردن انعطاف پذیری به وجود آمده اند. در نتیجه الگوی bridge نیز از pure fabrication بهره می‌برد.

Command

در این الگو اشیا command در قلمرو مسئله دیده نمی‌شوند. بلکه در مرحله‌ی طراحی و بنا به ضرورت قرار داده می‌شوند تا انتقال پیام به صورت غیر مستقیم صورت گرفته و هدف الگو محقق شود. در نتیجه این الگو از pure fabrcation بهره می‌برد.

Strategy

در این الگو اشیا strategy اشیا جعلی هستند که صرفا برای تحقق الگوریتم‌های مختلف ایجاد می‌شوند و در قلمرو مسئله دیده نمی‌شوند. بلکه با هدف بالا بردن انعطاف‌پذیری ایجاد شده اند تا بتوان در زمان اجرا الگوریتم‌های مورد نیاز را استفاده کرد. در نتیجه از این الگو بهره برده می‌شود.

مقایسه

هر سه الگو از pure fabrication بهره می‌برند.

Protected Variations

Bridge

در این الگو به خوبی رعایت شده است. کلاینت با واسط abstraction، از abstraction و implementationها منتزع شده است و تغییرات آن‌ها در کلاینت تاثیری نمی‌گذارد. همچنین implementation نیز از abstraction جدا شده و از طریق واسط، abstraction به آن دید دارد و در نتیجه تغییرات implementation نیز به abstraction منتشر نمی‌شود و از تغییرات مصون است.

Command

ارتباط invoker و کلاینت با receiver با تعریف command از یکدیگر جدا شده است. Commandها نیز خود ذیل واسط command تعریف شده اند و در نتیجه تغییرات در پیام به invoker منتشر نمی‌شود. هر چند کلاینت لازم است به concreteها دید داشته باشد اما invoker و receiver نسبت به تغییرات command مصون هستند.

Strategy

در این الگو، واسط strategy از context جدا شده و تعریف strategyهای جدید و هر گونه تغییر در آن، به context منتقل نمی‌شود. هر چند کلاینت یا پیکربند بایستی به concreteها دید داشته باشند و در نتیجه از تغییرات آن‌ها منتزع نیستند.

مقایسه

در هر سه الگو با تعریف واسط(ها) مناسب، ناپایداری و تغییرات زیاد به دیگر اشیا منتقل نمی‌شود، هر چند پیکربندی در این الگوها، منجر به این می‌شود که پیکربند دید مستقیم به concreteها داشته باشد اما در مجموع هر سه الگو از protected variations استفاده می‌کنند.

مقایسهٔ الگوهای Adapter، Builder و Mediator

دسته - نوع

Mediator

این الگو از الگوهای رفتاری می‌باشد. این الگوها به رفتار اشیا در زمان اجرا توجه دارند و نحوه‌ی تعامل اشیا با یکدیگر تمرکز دارند. هر چند ممکن است وجوه ساختاری نیز داشته باشند اما این وجه در آن‌ها پررنگ‌تر است.
این الگوها، دغدغه‌ی تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار شی، اداره‌ی درخواست‌های به یک شی و همچنین مدیریت بهتر تعامل میان اشیا در زمان اجرا با هدف افزایش cohesion و کاهش coupling را دارند.

Builder

این الگو از دسته‌ی الگوهای آفرینشی است.

Adapter

این الگو از دسته‌ی الگوهای ساختاری است و جز wrapperها به شمار می‌رود.
این دسته از الگوها بیشتر به ساختاردهی کلاس‌ها با هدف کاهش coupling و انعطاف‌پذیری بیشتر توجه دارند. این کار را معمولا با جداسازی abstraction از implementation و گسترش ساختار انجام می‌دهند. هر چند ممکن است این الگوها وجوه رفتاری نیز داشته باشند (که در proxy نیز وجه رفتاری جایگزینی مشخص است) اما وجوه ساختاری در آن پررنگ‌تر است.

مقایسه

هر سه الگو در دسته‌های مختلفی هستند.

هدف

Mediator

هدف این الگو، کاهش وابستگی میان مجموعه‌ای از اشیا به هم مرتبط است که با یکدیگر برای انجام وظیفه‌مندی‌های مختلف تعامل می‌کنند. در واقع در حالت پیشین، تعداد زیادی شی داریم که برای انجام وظایف، روابط پیچیده‌ای با یکدیگر دارند و پروتوکل ارتباطی آن‌ها با یکدیگر پیچیدگی دارد. در نتیجه در نگاه اول نمی‌توان رفتار آن‌ها را متوجه شد. هدف این الگو این است که با تعریف یک شی میانجی که وظیفه‌ی کنترل ارتباطات میان این اشیا به هم مرتبط را دارد، این پیچیدگی ارتباطی را کاهش دهد و در نتیجه‌ی آن، همه‌ی اشیا مرتبط که به آن‌ها همکار گفته می‌شود، به جای یک ارتباط توزیع شده، با شی میانجی در ارتباط خواهند بود و در نتیجه coupling میان همکاران با یکدیگر کاهش و میان میانجی با همکاران بالا خواهد شد.

Builder

هدف این الگو این است که با یک فرایند آفرینش یکسان بتوانیم اشیا متفاوتی تولید کنیم بدون آنکه ساختار داخلی شی را تغییر دهیم. برای این منظور، این الگو فرایند آفرینش را از ساختار شی متمایز کرده و در یک Builder قرار می‌دهد. در نتیجه در این الگو یک الگوریتم کلی ساخت داریم که استفاده می‌کنیم تا انعطاف پذیری داشته باشیم و بشود آبجکت‌های متفاوت ساخت.
همچنین می‌توان گفت این الگو، فرایند آفرینش را به گام‌های کوتاه تبدیل می‌کند تا بتوان با تغییر پارامترهای هر گام، شی متفاوتی تولید کرد.

Adapter

این الگو از wrapperها است. به صورت کلی در wrapperها، یک آبجکت روی یک آبجکت دیگری می‌نشیند و آن را در بر میگیرد تا یک interface روی آن تعریف کند. آبجکت‌های دیگر وقتی با wrapper کار می‌کنند فکر می‌کنند با آبجکت اصلی کار می‌کنند اما در واقع با آن کار نمی‌کنند.
در adapter، هدف الگو، رفع ناسازگاری میان interfaceهای client و سرور است. در حالت پیشین، یک کلاس داریم که interface آن با interface مد نظر کلاینت متفاوت است و کلاینت با آن کار نمی‌کند. **هدف** wrapper ما، تبدیل interfaceها است. به جای تغییر در کلاس مربوطه که بعضا ممکن نیست (مثلا از library جداگانه‌ای آمده است)، یک adapter قرار داده می‌شود تا از interface سرور به interface کلاینت تبدیل را انجام می‌دهد.

مقایسه

هدف این سه الگو را نمی‌توان چندان با یکدیگر مقایسه کرد. تمرکز mediator در حل مشکل پیچیدگی روابط است. در buildr، هدف ساختن یک الگوریتم یکسان آفرینش با خروجی‌های متفاوت می‌باشد و در adapter، هدف از بین بردن ناسازگاری میان interfaceها می‌باشد.

حوزه

Mediator

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

Builder

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

Adapter

این الگو هم در حوزه‌ی object قابل پیاده‌سازی و به کمک delegation در زمان اجرا قابل اعمال است و هم در حوزه‌ی class و با رابطه‌ی gen/spec و در زمان کامپایل قابل پیاده‌سازی است.

مقایسه

هر سه الگو در دسته‌ی الگوهای object هستند و در زمان اجرا به کمک delegation محقق می‌شوند.

کاهش coupling

Mediator

این الگو با هدف کاهش coupling میان مجموعه‌ی اشیا مرتبط که با یکدیگر ارتباطات در هم تنیده‌ای دارند و پیچیدگی اساسی میان آن‌ها در نحوه‌ی تعامل آن‌ها با یکدیگر است به کار می‌رود. در واقع شی میانجی ایجاد می‌شود تا از coupling شدید میان اشیا همکار جلوگیری کند.
اما نکته‌ای که در این الگو باید به آن توجه کرد، افزایش coupling میان همکاران با میانجی است. در واقع در حالت پیشین، تعداد زیادی شی با یکدیگر وابستگی داشتند اما پس از اعمال الگو، همه‌ی اشیا به واسطه‌ی دسترسی غیر مستقیم و از طریق میانجی، به آن وابستگی خواهند داشت. coupling میان آن‌ها loose می‌شود ولی به صورت کامل از بین نمی‌رود بلکه به وابستگی به میانجی تبدیل می‌شود. چرا که میانجی دید سطح concrete به اشیا همکار دارد. نکته‌ی مهمی که در این الگو وجود دارد، نبود dip در رابطه‌ی میانجی به همکاران است.
اما در مجموع coupling در وضعیت بهتری نسبت به حالت پیشین قرار می‌گیرد هر چند کامل از بین نمی‌رود.

Builder

در این الگو، وابستگی میان director و builder بسیار کم است چرا که دید میان آن‌ها از director به builder و در سطح بالا است که تنها با interface آن سر و کار دارد. در نتیجه میان این دو کلاس dip برقرار است و تغییرات منتشر نمی‌شود. اما میان client که در این الگو یک پیکربند است با builderها، وابستگی شدید دارد و باید تمامی زیرکلاس های آن را بشناسد چرا که builder مناسب را در اختیار director قرار دهد و به وسیله‌ی delegation، الگو را محقق کند. در نتیجه در مجموع می‌توان گفت این الگو از نظر coupling در وضعیت نسبتا مناسبی قرار می‌گیرد.

Adapter

در حالت class scope، این الگو، هر دو کلاس adaptee و target را implement می‌کند. در نتیجه وابستگی شدیدی میان adapter و دو کلاس دیگر خواهد بود و coupling بسیار شدید است.
در حالت object scope، این الگو، تنها target را implement و ارثبری می‌کند چرا که کلاینت نباید تفاوت آن را با شی target تشخیص دهد. اما در اینجا رابطه‌ی میان adaptee و adapter به شکل یک رابطه‌ی association در می‌آید که یک دید سطح بالا به یک adaptee توسط adapter ایجاد می‌شود. در نتیجه میان adapter و adaptee می‌توان گفت dip برقرار است.
شایان ذکر است که یک پیکربند ثالث نیز لازم است تا نسبت به همه‌ی کلاس‌های concrete دید داشته باشد در نتیجه coupling شدیدی با کلاس‌ها دارد. اما این در تمامی الگوها صدق می‌کند.

در نتیجه شکل class scope این الگو coupling شدیدی دارد که در حالت object scope به کمک delegation بهتر می‌شود.

مقایسه

الگوهای mediator و adapter از منظر coupling در وضعیت مناسبی قرار ندارند. البته mediator حالت پیشین را بهبود می‌بخشد اما همچنان coupling میانجی با کلاس‌های همکار بسیار بالا است. از طرفی وضعیت adapter بسیار بدتر است و بخصوص class scope آن اصلا از منظر وابستگی در وضعیت مناسبی قرار ندارد.
الگوی builder اما بهتر از دو الگوی دیگر coupling را مدیریت می‌کند.

افزایش cohesion

Mediator

در این الگو cohesion به طرز چشم‌گیری افزایش می‌یابد. چرا که یک کلاس متخصص کار چرخانی تعریف کرده‌ایم که تنها وظیفه‌ی آن تبادل اطلاعات و تبادلات میان اشیا است. در حالت پیشین هر یک از اشیای همکار، خود وظیفه‌ی ارتباط با دیگران و تبادل را برعهده داشتند اما در حالت پسین، میانجی وظیفه‌ی تبادل را برعهده گرفته و در نتیجه اشیا همکار سبک می‌شوند و seperation of conecerns صورت می‌پذیرد.

Builder

در این الگو نیز وظیفه‌ی افرینش به یک شی تخصصی افرینش محول می‌شود و این وظیفه از خود شی گرفته می‌شود و در نتیجه cohesion افزایش می‌یابد. در حالت پیشین وظیفه‌ی آفرینش بر عهده‌ی director بود اما پس از آن، این وظیفه از آن جدا شده و به یک شی متخصص داده می‌شود.

Adapter

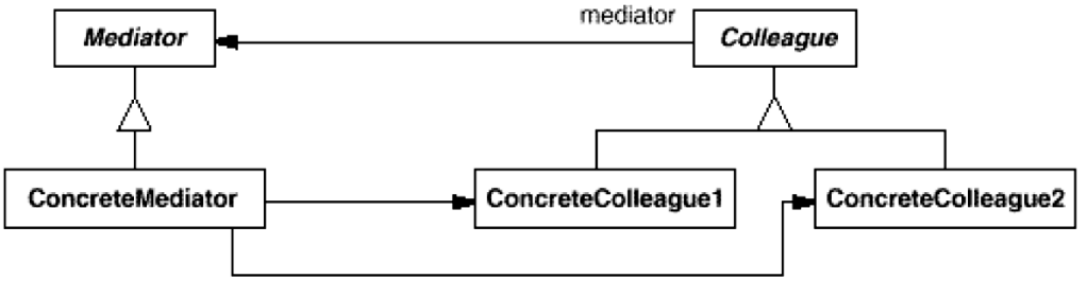
در این الگو نیز برای وظیفه‌مندی تبدیل دو interface، یک شی جدید جعل شده است تا این کار را بر عهده گرفته و رابط میان کلاینت و interfaceای که نمی‌شناسد شده است و در نتیجه یک وظیفه را برعهده گرفته است که پیش از این به صورت اضافی برعهده‌ی بخش دیگری از سیستم بوده است و در نتیجه cohesion افزایش می‌یابد.

مقایسه

هر سه الگو منجر به افزایش cohesion می‌شوند. در mediator یک شی متخصص میانجی، ارتباطات را از دید اشیا پنهان می‌کند، در builder، افرینش از دید director و کلاینت محفوظ می‌شود و در adapter، وظیفه‌ی تبدیل برعهده‌ی یک شی که هویت target را جعل می‌کند قرار گرفته است.

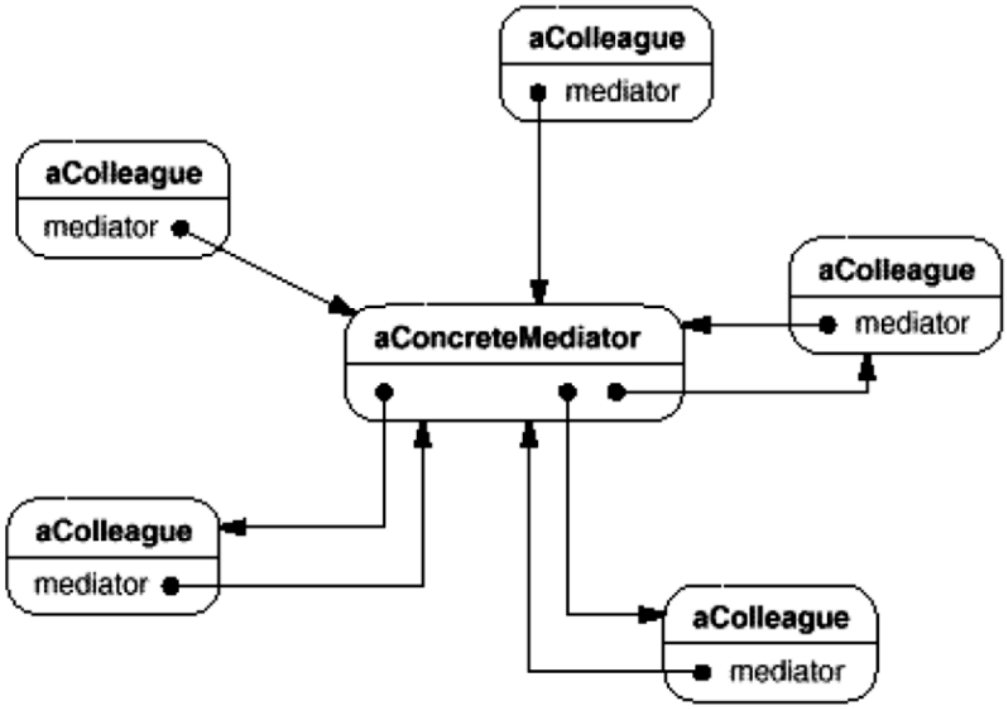
ساختار و رفتار

Mediator

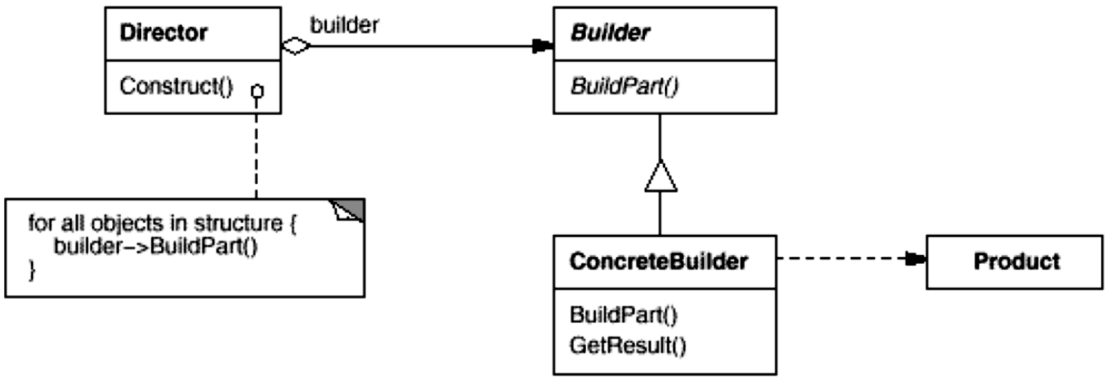


همانطور که می‌بینید، mediator یک interface است که همکاران به آن دید دارند. به عبارت دیگر اگر همکاری داشته باشیم که mediator را نمی‌بیند به معنی همکار نیست و جز آن محسوب نمی‌شود. به این‌گونه اشیا که mediator را نمی‌بینند اما در تعامل قرار می‌گیرند، اشیا سرویس‌دهنده می‌گوییم. این اشیا مشکل جدی بالا بردن coupling مجموعه را ایجاد نکرده اند بلکه اشیایی که دو طرفه به یکدیگر دید دارند یا شروع کننده‌ی زنجیره‌های تعاملی هستند مسئله‌ی اصلی این الگو هستند.
ارتباط برعکس میان mediator به colleagueها در سطح subclass تعریف می‌شود و لزومی ندارد که همه‌ی آن‌ها را ببیند. در این نمودار کلاس، متد خاصی به عنوان استاندارد در ConcreteMediator نمی‌بینیم و در نتیجه استاندارد متد برای آن نداریم و بسته به کارکرد مجموعه می‌تواند رفتار مختلف انجام دهد.

مطابق نمودار object زیر، ConcreteMediator لزومی ندارد که تمامی ConcretColleagureها را ببیند و در تعامل با آن‌ها باشد..

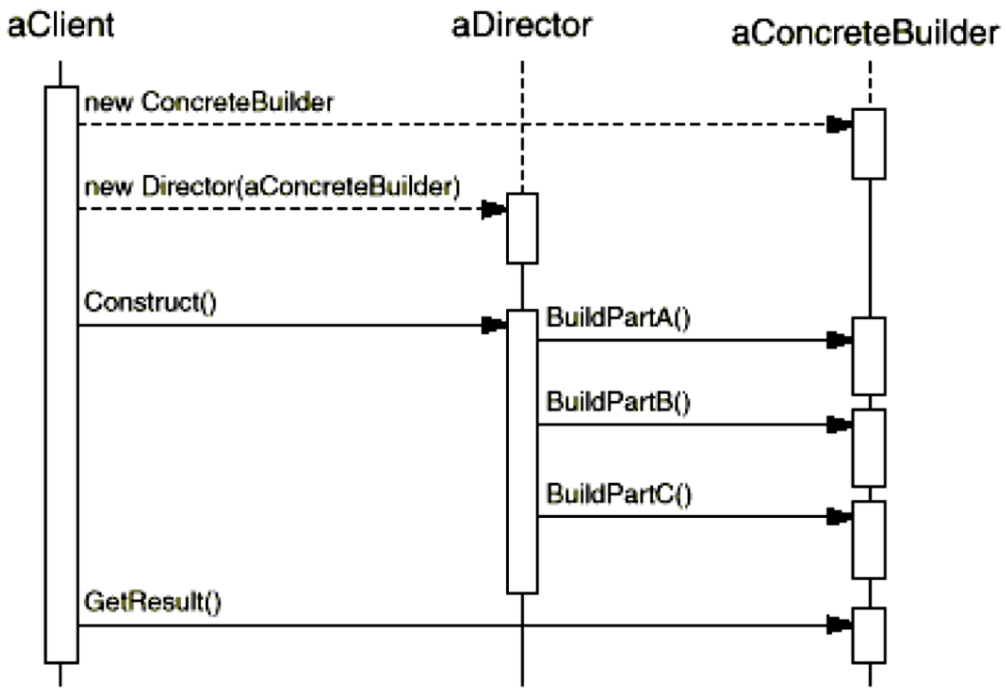


Builder



در این شکل، aggregation بین director و builder اشتباه است و ضرورت ندارد. برای مثال می‌شود builder مشترک میان چند شی داشته باشیم. یعنی لازم نیست که builder در دل director قرار بگیرد و به جای دیگری سرویس ندهد. رابطه‌ی میان این ها یک association ساده باید باشد.

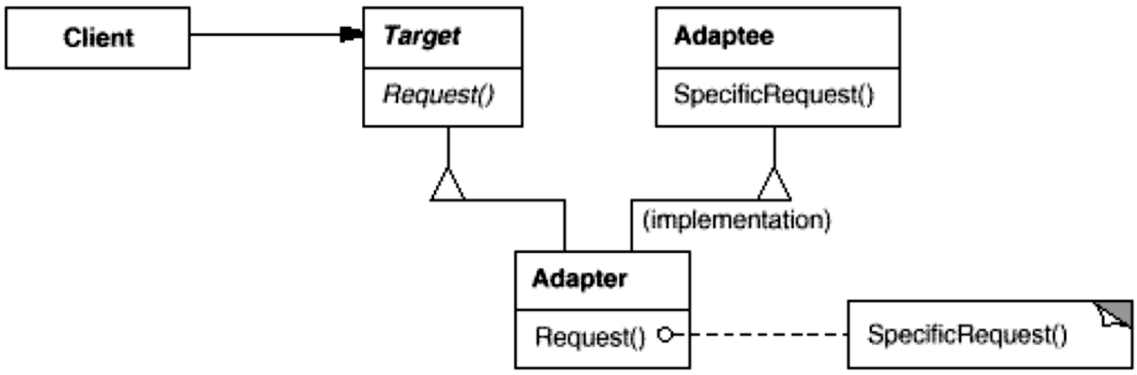
دید یک سویه است و director باید builder را بشناسد و برعکس آن صادق نیست. این یکسویه بودن دید در طراحی بسیار مناسب و مفید است.
معمولا operationهای سمت builder ریزدانه هستند برای اینکه بشود الگوریتم‌های ساخت متفاوتی ایجاد کرد.
باید دقت کرد که getresult در director مورد استفاده قرار نگرفته چرا که اساسا نیازی نیست به محصول نهایی دید داشته باشد چرا که کلاینت باید از آن استفاده کند. برای توضیح بهتر به نمودار توالی زیر دقت کنید.



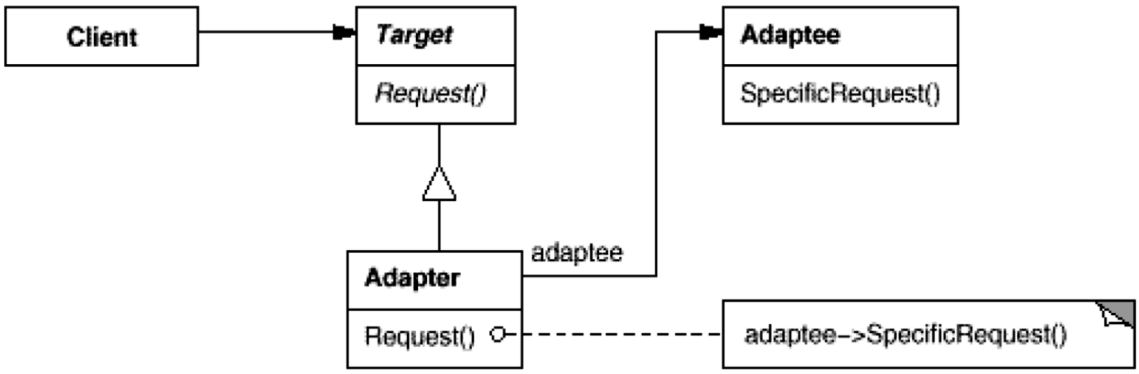
یک کلاینت، یک concrete builder را می‌سازد که در نتیجه coupling بالایی میان کلاینت و Builder وجود دارد چرا که تمام Builderها را باید بشناسد. اما ocp برقرار است چرا که تغییرات در بیلدر به director منتشر نمی‌شود ولی به کلاینت منتشر می‌شود. کلاینت‌ها عملا configurer نیز هستند. چرا که یک instance از concreteBuilder را به director می‌دهد. در نهایت کلاینت باید getResult کند. این متد در بخش abstraction قرار داده نشده است چرا که یک coupling زائد میان director و محصول نهایی بوجود می‌آید.
محصول نهایی توسط کلاینت استفاده می‌شود.

Adapter

Class Scope



همانطور که در اینجا می‌بینید، adapter از دو کلاس target که interface مد نظر کلاینت است و adaptee به ارث برده است. البته ارث بری adapter از adaptee به صورت رابطه‌ی is a نیست و فقط برای reuse به کار برده شده است (روی خط مربوطه implementation نوشته شده است). باید توجه داشت که adapter، رفتارهای زیرکلاس‌های adaptee را به ارث نمی‌برد و به همین دلیل، زیرکلاس‌ها جداگانه باید adapt شوند.
در این شرایط، در زمان اجرا فقط دو شی با یکدیگر کار می‌کنند. یک کلاینت و یک adapter. در نتیجه در زمان اجرا پیچیدگی ساختارهای objectی کمتر از حالت object scope است. یعنی adapter نیازی به instance از adaptee ندارد.
در اینجا نیز می‌شود رفتار به adaptee اضافه کرد. سهولت overriding در اینجا یک مزیت است. به دلیل ساختار صلب توارثی، سوییچ به راحتی انجام نمی‌شود و از adaptee هیچ instanceای گرفته نمی‌شود. کندی کار با چند object را نداریم چرا که فقط دو object با یکدیگر در ارتباط هستند. همچنین ضرورتی برای delegation نیست. در نتیجه سریع‌تر از حالت object scope است اما ساختار صلبی دارد. در نهایت باید اشاره کرد که این الگو با فراخوانی request توسط client و سپس اجرای SpecificRequest توسط Adpater در این پیاده‌سازی متد، محقق می‌شود.



در اینجا adapter به یک شی adaptee نیاز دارد و سه شی در زمان اجرا درگیر اجرای الگو هستند. همچنین سرویسی که کلاینت از adapter می‌گیرد به صورت غیر مستقیم از adaptee گرفته می‌شود در نتیجه کمی کندتر از adapter است. اینکه رفتاری را override کنیم معنی ندارد بلکه رفتار مجزا اضافه می‌کند. Override کردن رفتار adaptee تنها با ساختن زیرکلاس از adaptee انجام می‌شود.
نکتهٔ مهم اما این است که تمام زیرکلاس‌های adaptee نیز تطبیق می‌یابد. اما باید توجه کرد که یک پیکربندی لازم است تا adaptee در اختیار adapter قرار گیرد و الگو محقق شود.

مقایسه

در مقایسه‌ی این سه الگو می‌توان گفت که الگوهای Builder و Adapter در سطح زیر کلاس میان ساختارهای توارثی، دید وجود ندارد بلکه دید از پیکربند به آن‌ها است که منجر به coupling می‌شود. اما در Mediator، می‌توان گفت که Dip نقض شده و دید مستقیم میان ConcreteMediate و ConcreteColleagueها برقرار می‌شود.
همچنین باید توجه داشت که در Mediator، برخلاف دو الگوی دیگر، متد استانداردی در نظر گرفته نشده است چرا که این الگو بر روی ساختار به صورت عام و بدون نیاز به یک متد محقق می‌شود.

پیکربندی

Mediator

در این الگو از آنجایی که یک دید مستقیم از ConcreteMediator به ConcreteColleagueها و یک دید سطح بالا از Colleageها به Mediator داریم، لازم است تا یک شی ثالث به هر دو ساختار توارثی دید داشته و بتواند در زمان اجرا، ConcreteMediator را با ConcreteCollegeهای مناسب پیکربندی کند.

Builder

مطابق نمودار توالی این الگو که بالاتر آورده شد، کلاینت یک ConcreteBuilder ساخته و سپس Director را با این Builder پیکربندی می‌کند. در نتیجه وظیفه‌ی پیکربندی در این الگو بر عهده‌ی کلاینتی است که Director را می‌سازد. در این الگو کلاینت به ConcreteBuilderها و Director دید دارد چرا که پیکربند است اما دید میان Builder و Director سطح بالا است.

Adapter

Class scope

در این الگو، الگو در زمان کامپایل محقق می‌شود و نیاز به پیکربندی دیده نمی‌شود. در واقع تنها یک شی از کلاینت با یک شی از adapter در ارتباط هستند و در زمان اجرا کافی است تا یک شی adapter در اختیار کلاینت قرار گیرد.

Object scope

در حوزه‌ی شی اما این الگو نیاز به یک پیکربند دارد چرا که adapter لازم است تا به یک adaptee دید داشته باشد و در نتیجه یک پیکربند ثالث بایست یک شی از adaptee را در اختیار adapter قرار دهد. این adaptee می‌تواند در زمان اجرا به وسیله‌ی پیکربند تغییر کند چرا که دید adapter به آن یک دید سطح بالا است.

مقایسه

هر سه الگو تا حدی قابلیت پیکربندی دارند هر چند پیکربندی در adapter نسبت به دو الگوی دیگر کمرنگ‌تر است. در واقع در الگوی adapter تعداد اشیا درگیر الگو کمتر هستند و همچنین این الگو در class scope می‌تواند بدون پیکربندی محقق شود.
اما الگوی mediator به علت تعدد اشیا درگیر در الگو، نیاز به پیکربندی بسیار حیاتی است و حس می‌شود. در الگوی builder نیز نسبت به mediator این پیکربندی کمرنگ‌تر است اما همچنان توسط کلاینت انجام می‌شود.

انعطاف‌پذیری

Mediator

این الگو انعطاف‌پذیری را بهبود داده است. در حالت پیشین، به علت پیچیدگی در پروتوکل ارتباطی میان همکاران، افزودن همکار جدید و تغییر در ساختار پیچیده و سخت است و همچنین فهم عملکردی آن پیچیدگی دارد. در حالت پسین، با افزودن یک mediator، می‌توان به راحتی همکاران جدید اضافه و کم کرده و وظیفه‌مندی‌های جدید به آن افزود.
همچنین در صورتی که بخواهیم پروتوکل ارتباطی را تغییر دهیم کافی است تا mediator را بازپیکربندی کرده و یک mediator جدید جایگزین کنیم. در نتیجه انعطاف ساختار جدید بسیار بالا است.

Builder

این الگو، این امکان را به ما می‌دهد که با یک روش یکسان، شی‌های متفاوتی ساخت و در نتیجه، ساختار منعطفی دارد که امکان ایجاد شی‌های مختلف و مقداردهی مشخصه‌های شی را بدون پیاده‌سازی‌های متفاوت به ما می‌دهد. فرایند آفرینش شی نیز در این الگو به بخش‌های ریزدانه تقسیم شده است تا بتوان با انعطاف بیشتری شی را مقداردهی کرده و آن را ساخت بدون آنکه نیاز به تغییر فرایند آفرینش باشد.
همچنین امکان تغییر در builder نیز وجود دارد چرا که کلاینت می‌تواند director را با یک builder متفاوتی config کند و در نتیجه با افزودن builderهای جدید، می‌توان آن‌ها را به director منتسب کرده و به وسیله‌ی آن شی جدید را ایجاد کرد.

Adapter

Class scope

در این الگو ساختار صلب توارثی وجود دارد و در نتیجه در زمان اجرا انعطاف بالا نیست و با پیکربندی زمان کامپایل، دیگر نمی‌توان چیزی را تغییر داد و در نتیجه انعطاف پایین است.

Object scope

در این الگو به علت استفاده از رابطه‌ی association، به می‌توان ConcreteAdapteeهایی تعریف کرد که interface آن‌ها با adaptee یکی است و در نتیجه، adapter را می‌توان در زمان اجرا با adapteeهای متفاوت پیکربندی کرد و در نتیجه انعطاف الگو بالا است.

مقایسه

به جز الگوی class scope آداپتر، باقی الگوها از انعطاف‌پذیری بالایی برخوردار هستند. از طرفی می‌توان گفت انعطاف‌پذیری builder و adaptee شبیه به یکدیگر می‌باشد چرا که director و adapter هر کدام می‌توانند با پذیرفتن شیئی که interface مربوط به builder و adaptee را پیاده‌سازی می‌کنند، به خوبی بازپیکربندی شده و انعطاف بالایی داشته باشند. اما mediator انعطاف بسیار بالایی نسبت به دو الگوی دیگر دارد چرا که هم الگوریتم تعاملی و همچنین اشیا همکار به خوبی می‌توانند در زمان اجرا تغییر کنند و همچنین می‌توان با تغییرات کم، همکاران جدید تعریف کرد.

کارایی

Mediator

در این الگو، با تعریف یک شی میانجی، یک سطح دسترسی غیر مستقیم میان اشیا همکار به وجود می‌آید. در نتیجه با وجود بهتر شدن coupling میان آن‌ها، انتقال پیام بیشتری نسبت به حالت قبل نیاز داریم و در نتیجه کارایی تا حدی کمتر می‌شود. در واقع به ازای هر انتقال پیام میان همکاران در حالت پیش از الگو، یک انتقال پیام جدید در حالت پسا الگو خواهیم داشت.

Builder

در این الگو نیز فرایند آفرینش شی به بخش‌های ریزدانه‌تر و فراخوانی‌های متعدد تبدیل شده است و همچنین یک شی مجزا برای آفرینش در نظر گرفته می‌شود. در نتیجه انتقال پیام بیشتری لازم است تا الگو محقق شود و در نتیجه این الگو نیز کارایی را بدتر می‌کند.

Adapter

در این الگو در حوزه‌ی کلاس، هیچ واسپاری نداریم و الگو از طریق ارث‌بری محقق می‌شود در نتیجه انتقال پیام اضافی نخواهیم داشت و کارایی کاهش نمی‌یابد. اما در حوزه‌ی شی، یک شی adapter به adaptee انتقال پیام اضافی دارد در نتیجه کارایی کاهش می‌یابد.

مقایسه

به جز حوزه‌ی کلاسی adapter در باقی الگوها کارایی کاهش می‌یابد که علت آن مانند موارد قبلی به علت افزایش انتقال پیام است.

عواقب و تبعات

Mediator

اثرات مثبت

- subclassing را محدود می‌کند. چرا که mediator رفتار تعاملی توزیع شده را در خود مرکزیت می‌دهد و در نتیجه تغییر این رفتار با subclassing بسیار راحت می‌شود. اگر قبلا استفاده می‌کردیم تمام کلاس‌های درگیر extend می‌شدند.
- همکاران ارتباطشان غیر مستقیم می‌شود (با یکدیگر) که این هدف اصلی خواهد شد و در نتیجه coupling آن‌ها loose می‌شود و کامل از بین نمی‌رود اما نسبت به حالت پیشین بسیار کمتر می‌شود.
- کار میان اشیا و protocol ارتباطی آن‌ها و دانش آن‌ها نسبت به یکدیگر و سرویس گرفتن بسیار ساده می‌شود چرا که اشیا باید با mediator کار کنند و نیازی به دانستن آنکه چگونه از یکدیگر سرویس بگیرند را نخواهند داشت.
- مجموعه‌ی همکاری اشیا در یک mediator جمع می‌شود و این تعامل به جای توزیع در یک شی به عنوان دغدغه‌ی همان شی دیده می‌شود و وقتی به آن نگاه می‌کنیم صرفا interaction و وقتی به colleagueها نگاه می‌کنیم تنها کارهای مربوط به آن‌ها و سرویس آن انجام می‌شود و کارهای مربوط به ارتباط با دیگران در آن نیست. در نتیجه اشیایی داریم که cohesive شده اند و کارچرخانی برعهده‌ی mediator قرار گرفته است و متخصص کارچرخانی خواهد بود. در نتیجه تعامل از کارهای تخصصی منتزع شده و تعامل یک هویت پیدا کرده است. در نتیجه اگر خطایی رخ داد به راحتی می‌شود آن را یافت و تغییر را راحت می‌کند.

اثرات منفی

- این centralized شدن، ممکن است منجر شود که mediator بسیار پیچیده شود و حتی تبدیل به یک god class شود.
- درست است که coupling میان اشیا کم شده اما همه‌ی اشیا به mediator به گونه‌ای coupled شده اند.
- mediator می‌تواند single point of failure شود.
- mediator می‌تواند تبدیل به bottleneck شود.

Builder

اثرات مثبت

- اجازه می‌دهد که محصولاتی با ساختارهای داخلی متفاوت ساخته شود.
- کد ایجاد و کد ساخت نهایی از یکدیگر منتزع شده (ایجاد سمت director و ساخت نهایی سمت representation است)
- این الگو یک کنترل ریزدانه و دقیق به فرایند ساخت می‌دهد چرا که builder از operationهای ریزدانه تشکیل شده و فرایند ساخت را ریزدانه می‌کند.

Adapter

تبعات (class)

اثرات مثبت

- به خود adapter اجازه‌ی overriding می‌دهد.
- فقط یک object در اختیار client داریم و انتقال پیام غیر مستقیم نداریم.

اثرات منفی

- زیرکلاس‌های adaptee تطبیق نمی‌یابند و برای هر کدام از آن‌ها یک adapter جدا باید ساخت و پیاده سازی پیچیده‌تر می‌شود.

تبعات (object)

اثرات مثبت

- یک adapter با adaptee و تمام زیرکلاس‌هایش کار می‌کند و به صورت تجمیعی ساختار توارثی را تطبیق می‌دهد.
- انعطاف‌پذیری آن بیشتر از class است.

اثرات منفی

- رفتار adaptee را در adapter نمی‌شود override کرد و برای آن باید یک زیرکلاس جدید adaptee تعریف کرد و در زمان اجرا instance آن را در اختیار adapter قرار داد.

مقایسه

برای همه‌ی الگوهای گفته شده اثرات مثبت و منفی آورده شد. شباهت و تفاوت چندانی نمی‌توان برای آن‌ها اشاره کرد.

encapsulation

Mediator

در سطح ارتباط همکاران با یکدیگر و رابطه‌ی همکاران با mediator، ارتباط از سطح بالا و interface است در نتیجه دید نسبت به جزئیات کلاس mediator وجود ندارد و encapsulation برقرار است.
اما دید از mediator به concreteColleagueها برقرار است و در نتیجه ممکن است که encapsulation برای همکاران نقض شود.

Builder

دید از director به builder و از کلاینت به director در سطح interface است و در نتیجه encapsulation نقض نمی‌شود. البته کلاینت لازم است تا یک director را با یک ConcreteBuilder پیکربندی کند و در نتیجه نسبت به builder ممکن است encapsulation نقض شود.

Adapter

در هر دو حوزه‌ی شی و کلاس، encapsulation رعایت می‌شود چرا که دید از کلاینت به adapter و همچنین از adapter به adapteeها در سطح بالا است و در نتیجه state داخلی اشیا مورد دسترس قرار نمی‌گیرد.
البته در حالت کلاس باید توجه داشت که لازم است که attributeهایی که نیاز به encapsulation نسبت به adaptee هستند به صورت private تعریف شوند تا adapter و adaptee نسبت به یکدیگر encapsulation را رعایت کنند.

مقایسه

در adapter به صورت کامل encapsulation در کلاینت و اشیا دیگر رعایت شده است. اما در builder و mediater، در برخی شرایط که ذکر شد (دید کلاینت به ConcreteBuilder و دید mediator به ConcreteColleagueها) ممکن است که encapsulation نقض شود. اما به طور کلی از این منظر در وضعیت مناسبی قرار دارند.

انتشار تغییرات

Mediator

تغییرات در همکاران، به همکاران دیگر منتشر نمی‌شود چرا که از یکدیگر به صورت کامل منتزع هستند. در حالی که در حالت پیشین، تغییر در یکی از همکاران به سادگی می‌توانست منجر به تغییرات زیاد در دیگر همکاران شود. علت این اتفاق decoupled بودن همکاران نسبت به یکدیگر است.
اما همکاران نسبت به میانجی منتزع نیستند و اتفاقا وابستگی شدیدی دارند. در نتیجه تغییر در همکاران به راحتی می‌تواند منجر به تغییر در میانجی شود. از آنجایی که دید همکاران به میانجی از طریق interface است، تنها تغییر در interface آن می‌تواند منجر به تغییر در همکاران شود.

Builder

تغییر در ConcreteBuilder منجر به تغییر Director نمی‌شود چرا که دید آن در سطح interface است. اما کلاینت به صورت مستقیم به ConcreteBuilder دید دارد و در نتیجه تغییر در ConcreteBuilder به کلاینت منتقل می‌شود اما در director تاثیر نمی‌گذارد.
تغییرات در Director نیز می‌تواند منجر به تغییر در کلاینت شود چرا که کلاینت وظیفه‌ی پیکربندی را برعهده داشته و خود آن را instantiate می‌کند و در نتیجه تغییرات در آن می‌تواند منجر به تغییر در کلاینت شود.

Adapter

دید کلاینت به adapter سطح بالا است و در نتیجه تغییرات در هر کلاسی منجر به تغییر در کلاینت نمی‌شود مگر آنکه در interface تغییری ایجاد شود.. دید adapter به adaptee نیز سطح بالا است و مانند مورد گذشته می‌باشد.

مقایسه

در adapter تغییرات منتشر نمی‌شود. در Builder، کلاینت به راحتی می‌تواند از تغییرات Builder و Director تاثیر بپذیرد اما Builder و Director تغییرات خود را به یکدیگر منتشر نمی‌کند در نتیجه تا حدی انتشار تغییرات کنترل می‌شود و تاثیر پذیرفتن کلاینت به علت وظیفه‌ی پیکربندی آن طبیعی است. در Mediator اما میانجی نسبت به تغییرات مصون نیست و تغییرات در همکاران بسیار می‌تواند روی این کلاس تاثیر بگذارد و در نتیجه وضعیت این الگو نسبت به دو الگوی دیگر از این منظر نامناسب‌تر است.

میزان استفاده از منابع سیستمی

Mediator

در این الگو یک شی mediator جدید تعریف شده است که بایستی پروتوکل تعاملی را پیاده‌سازی کند. بیشتر داده‌های مورد نیاز برای این شی، referenceهایی به اشیا همکار است. در نتیجه می‌توان گفت این الگو نسبت به حالت پیشین، علاوه بر کارایی کمتر، کمی مصرف حافضه را افزایش می‌دهد. هر چند معمولا mediatorها اشیا متعددی نیستند و در نتیجه این مورد قابل چشم‌پوشی است.

Builder

در این الگو نیز با تعریف اشیا director و builder، اشیا جدیدی به سیستم اضافه می‌شود که برای مثال director نیازمند اطلاعاتی برای ساخت شی است. در نتیجه می‌تواند سربار روی حافظه داشته باشد. همچنین شی builder نیز اطلاعات مربوط به آفرینش شی اصلی را نیز بایستی در خود نگهداری کند در نتیجه الگو مصرف حافظه را افزایش می‌دهد.

Adapter

در حوزه‌ی کلاس مصرف حافظه تغییری نمی‌کند. در حوزه‌ی شی اما یک شی adaptee تعریف شده است. هر چند این مورد قابل چشم پوشی است اما اگر در بخش‌های مختلف سیستم adapter و adapteeهای زیادی داشته باشیم می‌تواند مشکل ساز شود. اما در مجموع به نظر قابل چشم‌پوشی می‌آید.

مقایسه

هر سه الگو به جز حوزه‌ی کلاس adapter، مصرف حافظه را افزایش می‌دهند. اما builder مصرف بیشتری دارد. دو الگوی دیگر به علت کم بودن تعداد اشیا جدید (بخصوص mediator که بیشتر رفتار را در خود ذخیره می‌کند)، می‌توان از این افزایش مصرف صرف نظر کرد.

استفاده از شی جعلی

Mediator

میانجی، یک شی جعلی است که یک متخصص افزوده شده به حالت پیشین است و صرفا برای جلوگیری از پیچیدگی پروتوکل ارتباطی آن‌ها افزوده می‌شود و در قلمرو مسئله و در مرحله‌ی تحلیل، این شی وجود ندارد و در طراحی و برای بهبود وضعیت پیشین به ساختار افزوده می‌شود.

Builder

شی buildr نیز یک شی متخصص است که الگوریتم آفرینش را درون خود قرار داده و با ریزدانه کردن فرایند آفرینش یک شی، امکان ایجاد شی‌های مختلف با یک فرایند یکسان را به ما می‌دهد. این شی در دامنه‌ی مسئله در ابتدا وجود ندارد. صرف نیاز به یک فرایند یکسان برای شی‌های مختلف منجر به این می‌شود که در مرحله‌ی طراحی این ساختار تغییر کند و در نتیجه این الگو یک شی متخصص جعلی را به حالت پیشین می‌افزاید.
در این الگو می‌توانیم director را نیز یک شی جعلی بدانیم چرا که در قلمرو مسئله کلاینت وظیفه‌ی آفرینش را برعهده دارد.

Adapter

در حالت class، شی adapter هر دو interface را پیاده‌سازی می‌کند و به همین دلیل، تفاوتی با دیگر اشیا نمی‌کند. اما این شی در دامنه‌ی تحلیل وجود ندارد. بلکه دو interface اصلی شناخته می‌شوند و این الگو در فاز طراحی و برای سازگاری میان آن‌ها افزوده می‌شود.
در حال object نیز شی adapter یک شی جعلی است که هویت Target را جعل می‌کند تا بتواند سازگاری را به وجود بیاورد.

مقایسه

الگوی mediator و adpater یک شی جعلی برای تحقق الگو می‌افزایند و builder، دو شی جعلی را برای تحقق الگو نیاز دارد.

سادگی پیاده‌سازی

Mediator

این الگو در ظاهر پیچیدگی چندانی ندارد اما الگوی سختی است از این جهت که mediator یک کارچرخان است که با همکاران مختلفی باید در ارتباط باشد. در نتیجه پیاده‌سازی الگوریتم ارتباطی در آن قرار گرفته و این پیاده‌سازی می‌تواند پیچیدگی داشته باشد.
مهم‌تر از آن اینست که میانجی خطرات زیادی دارد و باید با ملاحظه‌ی زیادی این الگو را اعمال کرد. کلاس میانجی می‌تواند به God Class تبدیل شود و شی میانجی به راحتی پتانسیل SPF شدن را دارد. در نتیجه ممکن است مجبور شویم میانجی را خود به میانجی‌های دیگری تقسیم کنیم تا مدیریت پیچیدگی کلاس را انجام دهیم و با لایه‌بندی و تقسیم کارها، این پیچیدگی را مدیریت کنیم که خود پیچیدگی پیاده‌سازی را بیشتر می‌کند.

Builder

پیاده‌سازی این الگو در زبان‌های شی گرا به سادگی قابل انجام است. نکته‌ی مهم builder، ریزدانگی بخش‌های مختلف ساختن شی است و تعریف آن‌ها تا بتوان روش‌های مختلفی را در Director تعریف کرد.

Adapter

پیاده‌سازی class scope در زبان‌های شی گرا به کمک inheritance انجام می‌شود و به سادگی قابل انجام است. پیاده‌سازی object scope نیز ساده‌تر از حالت پیشین است و کافی است یک شی adaptee به adapter منتسب شود.

مقایسه

میانجی نسبتا الگوی پیچیده‌ای است و اعمال آن ساده نیست. اما دو الگوی دیگر بسیار رایج هستند و به راحتی با کمک ابزارهای شی گرایی قابل پیاده‌سازی هستند.

موارد کاربرد

Mediator

- وقتی که مجموعه‌ای از اشیا داریم که به شکل پیچیده‌ای با یکدیگر کار می‌کنند. الگوریتم تعاملی آن‌ها مشخص است اما بسیار پیچیده است و در هم تنیده هستند و فهمیدن آنکه چه می‌کنند به علت coupling بالا سخت است.
- چون coupling بالا است و ارتباطات شی بسیار زیاد است، reusablity پایین است و در نتیجه، نمی‌توان به سادگی از آن استفاده کرد چرا که همه‌ی اشیا باید با هم استفاده شوند.
- می‌خواهیم رفتاری را که میان مجموعه‌ای از اشیا توزیع شده است، گسترش‌پذیر و توزیع‌پذیر کنیم بدون آنکه در سطح بالا subclassing داشته باشیم. منظور از رفتار جمعی یک رفتار تعاملی جمعی است. اگر بخواهیم آن‌را extend کنیم باید تمامی آن‌ها را extend کنیم اما پس از اعمال تنها کافی است که mediator را extend کنیم.

Builder

- می‌خواهیم الگوریتم ایجاد یک شی پیچیده رو مستقل از قطعاتی که استفاده می‌شوند و چگونگی اتصال آن‌ها تعریف کنیم. در سمت director الگوریتم ساخت ساده است اما می‌تواند چیزهای متفاوتی را برای ما بسازد. مطابق دستور director کار می‌کند اما تصمیم چگونگی با خودش است.
- فرایند ساخت ما اجازه‌ی ساخت آبجکت‌های با ساختارهای داخل مختلف را به ما بدهد. اتفاقی که در خط تولید میفتد می‌تواند خروجی متفاوت باشد چرا که از جزئیات مختلفی ساخته شده اما فرایند یکی است.

Adapter

- می‌خواهیم از یک کلاس موجود استفاده کنیم اما interface آن با interfaceی که ما نیاز داریم همخوانی ندارد.
- می‌خواهیم از یک کلاس reusable استفاده کنیم تا با کلاس‌هایی که باهاشون قراره تعامل کنه اما پیش‌بینی نشده‌اند، بتونه تعامل کنه. در واقع همخوانی میان interfaceهاشون وجود ندارد. قدیم این شکلی بود که interface رو یه جوری طراحی می‌کردیم که همه چیز رو پیش‌بینی می‌کردیم با اسامی مختلف برای اینکه بتواند در سیستم‌های مختلف وارد شود. الان اما به کمک adapter این مسئله حل می‌شود.
- (برای مدل object scope) وقتی که نیاز داریم که از زیرکلاس‌های متعدد موجود استفاده کنیم اما نمی‌خواهیم تک تک آن‌ها را تبدیل کنیم بلکه فقط می‌خواهیم کلاس پدر را تبدیل کنیم. مثلا در مثال

بالا، TextView تعدادی زیرکلاس داشته باشه. در این حالت وقتی text view رو adapt می‌کنیم، همه‌ی زیرکلاس‌ها نیز adapt می‌شوند.

مقایسه

برای هر یک از الگوها موارد کاربرد را مطرح کردیم. شباهت زیادی به یکدیگر ندارند اما یک مورد مشترک میان Mediator و Builder دیده می‌شود. هر دو الگو برای کاهش پیچیدگی کاربرد دارند. Builder الگوریتم ساخت اشیا پیچیده را استاندارد می‌کند و mediator نیز به ما امکان مدیریت پیچدگی الگوریتم ارتباطی میان همکاران به ما می‌دهد.

الگوهای مرتبط

Mediator

با الگوی command در ارتباط است (البته بسیار ارتباط دوری دارد). چرا که در command، کسی که درخواست اجرای فرمان را دارد به جای آنکه به صورت مستقیم درخواست را مطرح کند، از طریق command پیام را به شی دیگر منتقل می‌کند. در mediator نیز پیام از فرستنده به صورت غیر مستقیم و از طریق میانجی منتقل می‌شود.

از طرفی این الگو به Facade شباهت بسیاری دارد. هر چند کارکرد آن‌ها متفاوت است. Facade، یک زیر سیستم را نسبت به بیرون آن به کمک ایجاد یک interface روی زیر سیستم منتزع می‌کند. در نتیجه اشیا خارجی برای سرویس گرفتن از این زیر سیستم لازم است از طریق facade با آن ارتباط برقرار کنند. اما اشیا زیر سیستم، خود از یکدیگر خبر دارند و بین آن‌ها ارتباط لزوما غیر مستقیم نمی‌شود. هر چند در حالت پیشین، کلاینت باید از اشیا مختلف زیر سیستم خبر داشته باشد.
اما در Mediator، هدف جداسازی اشیایی است که در یک component با یکدیگر همکاری می‌کنند. در نتیجه دید میان این اشیا را نسبت به یکدیگر منتزع می‌کنیم و ارتباط آن‌ها را در Mediator کپسوله می‌کنیم. در نتیجه هر کدام از اشیا شبیه کلاینت در facade می‌شوند که از کاری که برای ارتباط انجام می‌شود منتزع شده اند. اما هدف این دو الگو و کاری که می‌کنند تا حدی متفاوت است.

Builder

می‌توان الگوی Builder را شبیه به Bridge دانست. در واقع Director به عنوان abstraction و builder به عنوان implementation در نظر گرفته شده و می‌توان ساختار توارثی برای آن‌ها شکل داد.
همچنین builder را می‌توان به صورت singleton پیاده‌سازی کرد که از این جهت به آن شباهت دارد.

Adapter

این الگو یک wrapper است و از جهتی شباهت بسیاری به دیگر wrapperها مانند facade ،proxy و decorator دارد. چرا که یک شی جعلی ایجاد می‌شود تا از طریق آن، بخشی از سیستم نسبت به بیرون منتزع می‌شود هدف الگو محقق شود. شباهت خاص آن به proxy نیز در بخش proxy اشاره شده است.
Facade نیز یک interface جدید روی یک بخشی از سیستم تعریف می‌کند در حالی که adapter تلاش می‌کند interfaceهای موجود با یکدیگر قابل تعامل باشند.

مقایسه

Mediator و adapter از جهاتی که توضیح داده شد به facade شباهت دارند. در باقی موارد میان الگوها شباهت زیادی دیده نمی‌شود.

OCP

Mediator

در این الگو dip برقرار نیست. چرا که دید از ConcreteMediator به ConcreteColleagueها وجود دارد. از طرفی اما ConcreteColleagueها از طریق interface به mediatorها دید دارند. در نتیجه می‌توان گفت گسترش ساختارهای colleagueها، مستقیما منجر به تغییر در ConcreteMediator می‌شود اما گسترش Mediator روی Colleagueها تاثیری ندارد. در نتیجه می‌توان گفت این الگو ocp را نقض می‌کند.

Builder

با توجه به اینکه ارتباط میان director و builder از طریق interface برقرار است، گسترش builder بر روی director تاثیری نمی‌گذارد چرا که dip برقرار است و در نتیجه ocp در این رابطه برقرار است.
اما در رابطه‌ی کلاینت با director و builder باید گفت که چون کلاینت پیکربند است و دید مستقیم به زیرکلاس‌های builder دارد، گسترش builderها بر روی کلاینت نیز تاثیر دارد و منجر به تغییر می‌شود در نتیجه در این سطح ocp برقرار نیست.
اما از آن‌جایی که کلاینت نقش پیکربندی دارد، می‌توان گفت تا حدی ocp برقرار است.

Adapter

در حوزه‌ی کلاس، این الگو از طریق inheritance محقق می‌شود و adapter، هر دو interface کلاس هدف و کلاس adaptee را پیاده‌سازی می‌کند. به همین علت گسترش ساختارهای توارثی adaptee و target بر روی آن تاثیر ندارد. کلاینت نیز اساسا از طریق interface به target دید دارد و در نتیجه ocp برقرار است چرا که dip برقرار است.
در حوزه‌ی object، می‌توان دیدی که adapter توسط پیکربند با یک adaptee پیکربندی می‌شود اما dip میان ساختار adapter و adaptee برقرار است. کلاینت نیز اساسا با واسط target در ارتباط است و در نتیجه در این حوزه نیز ocp برقرار است. هر چند پیکربند بایستی دید مستقیم به Concreteها داشته باشد که یک ایراد رایج است.

مقایسه

در mediator، می‌توان دید که ocp نقض می‌شود، در builder، تا حدی ocp برقرار است و adapter، تا حد بالایی ocp را برقرار می‌کند با صرف نظر کردن پیکربندی.

LSP

Mediator

در این الگو، Colleagueها همگی رابطه‌ی is a با کلاس پدر خود بایستی داشته باشند و همچنین mediatorها نیز باید بتوانند به درستی کارچرخانی کرده و در نتیجه رابطه‌ی is a با پدر خود دارند. در نتیجه در این الگو lsp برقرار است.

Builder

در این الگو، زیرکلاس‌های builder رابطه‌ی is a با پدر خود دارند و در نتیجه، این الگو نیز lsp را رعایت می‌کند.

Adapter

در حوزه‌ی کلاس، adapter در واقع رابطه‌ی is a با adaptee ندارد بلکه تنها رفتار آن را جعل می‌کند تا بتواند هدف الگو را محقق کند. در نتیجه در واقع adapter را نمی‌توان لزوما با adaptee جایگزین کرد. ممکن است در این الگو refused bequest نیز اتفاق بیفتد چرا که یک operation در adaptee تعریف شود که آن رفتار در adapter نباشد. پس lsp در این الگو نقض می‌شود.

مقایسه

در mediator و builder، این قاعده رعایت شده است اما adapter آن را نقض می‌کند.

DIP

Mediator

در این الگو dip در رابطه‌ی میان Colleague با mediator، برقرار است چرا که colleague از طریق واسط با mediatorها ارتباط دارد. اما mediatorها رابطه‌ی concrete به concrete با کلاس‌های همکار دارند. در نتیجه، dip نقض می‌شود و در کل می‌توان گفت در این الگو dip نقض می‌شود چرا که mediator باید ConcreteColleagueهای مورد نیاز خود را بشناسد و با آن‌ها تعامل کند.

Builder

در این الگو، director رابطه‌ی سطح بالا با builder دارد. اما کلاینت‌ها به دلیل وظیفه‌ی پیکربندی، لازم است تا دید concrete به builderها داشته باشد و در نتیجه dip در رابطه‌ی کلاینت به concreteها نقض می‌شود.

Adapter

در حوزه‌ی کلاس، این الگو، در رابطه‌ی میان کلاینت با target، رابطه‌ی سطح بالا است و در نتیجه در این الگو dip برقرار است چرا که adaptee و target خود کلاس پدر adapter می‌باشند.
در حوزه‌ی شی، dip برقرار است. چرا که مانند حوزه‌ی کلاس، دید کلاینت به واسط target است و همچنین دید adapter به adaptee، یک دید سطح بالا است که توسط پیکربند محقق می‌شود. اما در این الگو مشکل پیکربند که بایستی دید به concrete adaptee ها داشته باشد وجود دارد و در نتیجه برای آن dip نقض می‌شود.

مقایسه

در mediator، این قاعده نقض می‌شود. در builder می‌توان گفت کلاینت dip را با دید مستقیم به ConcreteBuilderها نقض می‌کند هر چند وظیفه‌ی پیکربندی را دارد. ولی تا حدی در این الگو dip برقرار است.
در adapte نیز dip برقرار است به جز در رابطه‌ی پیکربندی حوزه‌ی شی این الگو مشکل dip در پیکربند یک پدیده‌ی طبیعی در اکثر الگوهای gof است.

ISP

Mediator

در این الگو، یک واسط mediator تعریف شده است تا وظیفه‌ی کارچرخانی را از روی دوش همکاران بردارد. در واقع پیچیدگی‌های پروتوکل ارتباطی به mediator منتقل شده است در نتیجه mediatorها cohesive هستند که وظیفه‌ی کارچرخانی را برعهده دارند. باقی interfaceها نیز وظایف مربوط به خود را انجام می‌دهند و کارهای ارتباطی را برعهده نمی‌گیرند در نتیجه در این الگو رعایت شده است.

Builder

در این الگو، فرایند آفرینش ریزدانه شده و در builder قرار گرفته است در حالی که در حالت پیشین بایستی بر عهده‌ی director یا کلاینت قرار می‌گرفت. در نتیجه با تعریف واسط builder، وظیفه‌ی آفرینش سیستم را از اشیا نامربوط جدا کرده و ساختار cohesive نسبت به حالت پیشین ایجاد شده است در نتیجه isp رعایت شده است.

Adapter

در هر دو حوزه برقرار است. چرا که وظایف مربوط به target با یک adapter جعل می‌شود و در کلاس این adapter خود adaptee را پیاده‌سازی کرده است و در حوزه‌ی شی به کمک delegation از یک adaptee بهره می‌برد. همه‌ی کلاس‌ها cohesive هستند و وظایف خود را انجام می‌دهند. هر چند می‌توان این ایراد را به حوزه‌ی کلاس وارد کرد که adapter، خود adaptee را نیز پیاده‌سازی می‌کند که منجر به پایین آمدن cohesion می‌شود اما در مجموع می‌توان گفت isp برقرار ست.

مقایسه

هر سه الگو، isp را رعایت می‌کنند.

CRP

Mediator

در این الگو، با در اختیار داشتن همکاران مناسب برای تحقق رفتار در mediator، الگو به کمک delegation تحقق می‌یابد. همچنین همکاران نیز یک دید مانا به mediator مناسب خود دارند و در نتیجه دید مانا میان mediator و همکاران باعث می‌شود تا این الگو از این قاعده پیروی کند. در زمان اجرا می‌توان mediatorها را پیکربندی کرد که از مزایای برخورداری از crp است و ساختار صلب توارثی دیده نمی‌شود.

Builder

این الگو نیز به کمک delegation تحقق می‌یابد. در زمان اجرا می‌توان builderهای مناسب را برای director پیکربندی کرد و در نتیجه این الگو به کمک delegation محقق می‌شود و ساختار صلب توارثی دیده نمی‌شود. در نتیجه crp برقرار است.

Adapter

در حوزه‌ی کلاس، این الگو crp را نقض می‌کند چرا که inheritance بر delegation ترجیح داده شده است و کلاس adapter هر دو واسط target و adaptee را پیاده‌سازی می‌کنند.
در حوزه‌ی شی اما الگو به کمک delegation و برقراری رابطه‌ی adapter با adaptee مناسب، محقق می‌شود و در نتیجه، crp در حوزه‌ی شی نقض نمی‌شود.

مقایسه

تنها حوزه‌ی کلاس الگوی adapter، این اصل را نقض می‌کند و در باقی موارد، این اصل رعایت شده است.

PLK

Mediator

در این الگو، دید ترایا دیده نمی‌شود. Mediator نقش یک کارچرخان را دارد که با فراخوانی توسط همکاران، پیام را میان همکاران مختلف پخش می‌کند و کار را می‌گرداند بدون آن‌که همکاران را در اختیار دیگری قرار دهد. در نتیجه دید ترایا دیده نمی‌شود، message chain نداریم و در نتیجه PLK برقرار است.

Builder

در این الگو، دید ترایا دیده نمی‌شود. Director درخواست کلاینت را با همکاری builder اجرا می‌کند ولی builder خود را در اختیار کلاینت قرار نمی‌دهد. در نتیجه message chain نداریم و plk برقرار است. شاید در نگاه اول اینطور به نظر برسد که خروجی builder که یک شیئی است که هدف ما آفرینش آن بوده، در نتیجه دید ترایا وجود دارد اما اساسا builder در زنجیره‌ی فراخوانی رفتارهای شی محسوب نمی‌شود و در نتیجه، هدف خود را محقق می‌کند اما پیام‌های پس از آن که برای شی هدف ارسال می‌شود، بدون دخالت builder اتفاق می‌افتد و خود زنجیره‌ی جداگانه‌ای دارد. در نتیجه در مجموع plk برقرار است.

Adapter

در این الگو نیز چه در حوزه‌ی کلاس و چه در حوزه‌ی شی message chain و دید ترایا دیده نمی‌شود. در واقع client با adapter از طریق واسط در ارتباط است و adapter نیز در حوزه‌ی کلاس اساسا خود جایگزین adaptee شده است و در حوزه‌ی شی نیز adaptee را در اختیار کلاینت قرار نمی‌دهد در نتیجه دید ترایا نداریم و plk برقرار است.

مقایسه

در هر سه الگو این قاعده برقرار است.

مقایسهٔ الگوهای Mediator، Builder و Adapter بر اساس grasp

Information Expert

Mediator

رفتار از داده در این الگو جدا نیفتاده است. در واقع اشیا کپسول‌های داده رفتار هستند. همکاران هر کدام وظایف خود را می‌دانند و به دادگان خود برای انجام رفتار دسترسی دارند. Mediator نیز اساسا تنها به همکاران خود نیاز دارد تا رفتار مطلوب را نشان دهد که به آن دسترسی دارد. در نتیجه همه‌ی اشیا کپسول‌های داده رفتار هستند و در نتیجه به خوبی در این الگو، information expert رعایت شده است.

Builder

در این الگو، نحوه‌ی افرینش در اختیار builder قرار دارد در حالی که برخی از داده‌های آفرینش می‌تواند در اختیار director باشد. در نتیجه Director با فراخوانی buildPart، بخش‌های مختلف را تکمیل می‌کند اما افرینش را خود انجام نمی‌دهد. به این صورت می‌توان گفت بسته به طراحی، می‌تواند این الگو توسط builder نقض شود چرا که داده در director و رفتار در builder قرار داده شده و الگو نقض شده است.

Adapter

در این الگو، شی adapter اطلاعات لازم برای تبدیل میان دو interface را در اختیار دارد و در نتیجه رفتار و داده‌های آن در اختیار خود قرار دارد. در حوزه‌ی کلاس، رفتار adaptee در خود adapter پیاده‌سازی شده است اما در حوزه‌ی شی، در اختیار یک شی متخصص adaptee قرار دارد که داده‌های مربوط به رفتار خود را ذخیره کرده است. در نتیجه در این الگو information expert به خوبی رعایت شده است.

مقایسه

در mediator و adapter این الگو اعمال شده است ولی در builder با در نظر گرفتن نوع طراحی می‌توان گفت می‌تواند این الگو نقض شود. به صورت معمول نیز این الگو در طراحی‌های رایج، information expert را نقض می‌کند.

Creator

Mediator

این الگو نقض می‌شود. چرا که mediator توسط پیکربند، پیکربندی می‌شود و در در حالی که یک دید مانا از همکاران به Mediator وجود دارد. البته باید توجه کرد که کلاس‌های همکار نیز توسط پیکربند ساخته می‌شوند در حالی که دید مستقیم از mediator به همکاران وجود دارد. در نتیجه این الگو نقض می‌شود چرا که اولویت ۳ یا ۴ وجود دارد اما با اولویت ۵، پیکربند وظیفه‌ی ساخت را برعهده می‌گیرد.

Builder

در builder، کلاینت که پیکربندی است builder را در اختیار director قرار می‌دهد. در حالی که رابطه‌ی association میان director و builder وجود دارد و در نتیجه اولویت بالاتری نسبت به پیکربند دارد. پس این الگو نقض می‌شود.

Adapter

در هر دو حوزه‌ی کلاس و شی، به دلیل دید از کلاینت به adapter، اولویت ساخت adapter با کلاینت است اما توسط پیکربند اشیا پیکربندی می شوند و در نتیجه creator رعایت نشده است.

مقایسه

در هر سه الگو، creator رعایت نشده است و اولویت association نادیده گرفته شده است.

Low Coupling

Mediator

در این الگو، در حالت پیشین، همکاران به یکدیگر دید مستقیم داشتند تا بتوانند با یکدیگر برای تحقق وظیفه‌مندی‌ها تبادل پیغام داشته باشند. اما پس از اعمال الگو، میان همکاران، coupling کم می‌شود چرا که آن‌ها فقط با mediator در ارتباط هستند و وظیفه‌ی کارچرخانی و ارتباط میان اشیا به mediator محول شده است.
اما از طرفی mediator، مشخصا coupling شدیدی با همکاران دارد. چرا که برای کارچرخانی لازم است تا زیرکلاس‌های همکاران را بشناسد. در نتیجه، می‌توان گفت coupling نسبت به حالت پیشین بهبود جدی پیدا کرده است اما همچنان coupling میان mediator و همکاران برقرار است.

Builder

در این الگو، coupling میان کلاینت و director و همچنین میان director و builder بسیار کم است چرا که دید میان آن‌ها سطح بالا است و کلاینت از طریق واسط با director و director نیز از طریق واسط با builder در ارتباط است. اما کلاینت بایستی زیرکلاس‌های builder را ببیند چرا که پیکربندی می‌کند در نتیجه coupling در میان این دو نسبتا بالا اما در باقی روابط پایین است.
به طور کلی می‌توان گفت از نظر coupling در وضعیت نسبتا خوبی برخوردار است هر چند کلاینت به builder وابستگی دارند.

Adapter

در حوزه‌ی کلاس، دید میان کلاینت و target سطح بالا است در نتیجه coupling آن‌ها کم خواهد بود. اما در رابطه‌ی میان adapter و adaptee و target به علت آنکه adapter این دو کلاس را ارث‌بری می‌کند، coupling بسیار شدید است.
در حوزه‌ی شی اما coupling در وضعیت بسیار مطلوبی قرار دارد چرا که adaptee از طریق delegation به adapter تعلق پیدا می‌کند و در نتیجه نسبت به حوزه‌ی کلاس، وابستگی میان adapter و adaptee دیده نمی‌شود و وضعیت آن از نظر coupling بسیار مناسب است.

مقایسه

در builder و adapter حوزه‌ی شی، وضعیت از نظر coupling مناسب است. هر چند پیکربند دید مستقیم به زیرکلاس‌ها دارد و coupling را افزایش می‌دهد اما در مجموع وضعیت مناسب است. در mediator نیز

coupling نسبت به حالت پیشین بسیار بهتر شده است و بهبود پیدا کرده است اما همچنان coupling میان mediator و همکاران وجود دارد. در adapter حوزه‌ی کلاس نیز coupling از نوع ارث‌بری وجود دارد که نسبتا بالا است.

High Cohesion

Mediator

در این الگو cohesion بسیار مناسب و بالا است. در حالت پیشین، کلاس‌های همکار علاوه بر وظایف خود، بایستی پروتوکل ارتباطی با دیگر همکاران را نیز پیاده‌سازی می‌کردند. اما در حالت پسین، یک شی cohesive کارچرخان که همان mediator است ایجاد می‌شود تا نحوه‌ی ارتباط میان همکاران را پیاده‌سازی کند. در نتیجه، این الگو از cohesion بسیار بالایی برخوردار است و کلاس‌ها وظایف بی ربط ندارند.

Builder

در این الگو، کلاس متخصص آفرینش تعریف می‌شود که فرایند آفرینش را در خود محفوظ می‌دارد. همچنین director نیز وظیفه‌ی اعمال فرایند برای به دست آمدن نتیجه را بر عهده می‌گیرد. در نتیجه اشیای متخصصی تعریف شده است که هر کدام cohesive هستند و مسئولیت بی ربط به کلاس‌ها تخصیص داده نشده است.

Adapter

در هر دو حوزه، کلاس adapter یک کلاس متخصص تبدیل interfaceها است که تعریف شده است تا وظیفه‌ی تبدیل آن ها را برعهده بگیرد. باقی اشیا نیز همگی cohesive هستند و مسئولیت بی ربط به اشیا تخصیص داده نشده است در نتیجه coehsion بالا است.

مقایسه

هر سه الگو از cohesion بالایی برخوردارند.

Controller

Mediator

Mediator یکی از معروف ترین کارچرخان‌ها است. اساسا وظیفه‌ی mediator این است که کار را میان همکاران بچرخاند و وظایف ارتباطی را از روی دوش آن‌ها بردارد. در نتیجه mediator خود یک controller است و این الگو استفاده می‌شود.

Builder

شاید به نوعی بتوان director را یک controller در نظر گرفت اما این گونه نیست چرا که وظیفه‌ی director کارچرخانی نیست بلکه ساخت شی را برعهده دارد و میان چند شی و زنجیره‌ی تعاملی دیده نمی‌شود که بتوان آن را کنترلر نامید. در نتیجه این الگو controller را ندارد.

Adapter

در این الگو، هیچ یک از اشیا وظیفه‌ی کارچرخانی ندارند. در نتیجه این الگو نیز از controller بهره نمی‌برد.

مقایسه

تنها mediator از الگوی کنترلر استفاده می‌کند.

Polymorphism

Mediator

در این الگو می‌توان میانجی‌های متفاوتی برای وظیفه‌های مختلف در نظر گرفت و این کار را با تعریف زیرکلاس‌های مختلف mediator که هر کدام operation را مجزا پیاده‌سازی می‌کنند، انجام داد. در نتیجه polymorphism نیز در این الگو استفاده می‌شود.

Builder

در این الگو نیز می‌توان builderهای مختلف تعریف کرد که فرایند آفرینش متفاوتی را در نظر می‌گیرند و این کار را با polymorphism در ساختار توارثی builder انجام می‌دهند. در نتیجه در این الگو نیز استفاده می‌شود.

Adapter

در حوزه‌ی کلاس و حوزه‌ی شی، adapter به خوبی از چند ریختی بهره می‌برد و رفتار target را پیاده‌سازی می‌کنند. Target نیز خود به طور مجزا می‌تواند زیرکلاس‌های دیگری نیز داشته باشد.

مقایسه

در هر سه الگو از polymorphism استفاده می‌شود یا می‌توان بهره برد.

Indirection

Mediator

در این الگو، mediator یک شی میانجی است که کارچرخانی کرده و یک سطح indirection در ارتباطات میان همکاران به وجود می‌آورد. در واقع در حالت پیشین، همکاران مستقیما با یکدیگر در ارتباط بودند اما پس از اعمال الگو، تنها از طریق mediator به یکدیگر پیغام ارسال می‌کنند و توالی و پروتکل ارتباطی آن‌ها در mediator قرار دارد. در نتیجه یک indirection توسط mediator ایجاد شده و این الگو استفاده می‌شود.

Builder

در این الگو پیش از وجود builder، کلاینت یا کارگردان خود وظیفه‌ی آفرینش شی را برعهده داشتند. اما builder یک سطح ارتباط غیر مستقیم میان director و شی در حال آفرینش ایجاد می‌شود تا انعطاف پذیری بالا رفته و فرایند آفرینش شی یک فرایند ثابتی بماند. در نتیجه در این الگو نیز indirection دیده می‌شود.

Adapter

در حوزه‌ی شی، adapter یک میانجی است که در هنگام فراخوانی، رفتار adaptee را بروز می‌دهد و در نتیجه یک سطح ارتباط غیر مستقیم برای تبدیل دو interface به یکدیگر ایجاد شده است در نتیجه در این حوزه از این الگو بهره برده می‌شود.
در حوزه‌ی کلاس اما رابطه‌ی غیر مستقیم دیده نمی‌شود و کلاینت مستقیم رفتار را از adapter دریافت می‌کند و adapter خود از طریق ارث‌بری جای adaptee می‌نشیند و رفتار هر دو ساختار را داشته باشد. در نتیجه نمیتوان گفت ارتباط کلاینت با adaptee به صورت مستقیم انجام شده چرا که adapter خود جای adaptee نشسته است.

مقایسه

بجز adapter حوزه‌ی کلاس، در باقی الگوها از یک سطح indirection برای تحقق الگو بهره برده می‌شود.

Pure Fabrication

Mediator

در این الگو شی میانجی، در قلمرو مسئله نیست و یک شی متخصص میانجی‌گری و کارچرخانی است که پروتکل ارتباطی را نگه می‌دارد و پیاده‌سازی می‌کند. در نتیجه این شی یک شی جعلی است و این الگو از pure fabrication بهره می‌برد.

Builder

شی builder یک شی متخصص است که الگوریتم آفرینش را درون خود قرار داده و با ریزدانه کردن فرایند آفرینش یک شی، امکان ایجاد شی‌های مختلف با یک فرایند یکسان را به ما می‌دهد. این شی در دامنه‌ی مسئله در ابتدا وجود ندارد. صرف نیاز به یک فرایند یکسان برای شی‌های مختلف منجر به این می‌شود که در مرحله‌ی طراحی این ساختار تغییر کند و در نتیجه این الگو یک شی متخصص جعلی را به حالت پیشین می‌افزاید.
در این الگو می‌توانیم director را نیز یک شی جعلی بدانیم چرا که در قلمرو مسئله کلاینت وظیفه‌ی آفرینش را برعهده دارد.

Adapter

در حالت class، شی adapter هر دو interface را پیاده‌سازی می‌کند و به همین دلیل، تفاوتی با دیگر اشیا نمی‌کند. اما این شی در دامنه‌ی تحلیل وجود ندارد. بلکه target و adaptee شناخته می‌شوند و این الگو در فاز طراحی و برای سازگاری میان آن‌ها افزوده می‌شود.
در حوزه‌ی object نیز شی adapter یک شی جعلی است که برای حل مشکل عدم تطابق target با adaptee این شی تعریف می‌شود.

مقایسه

هر سه الگو اشیا جعلی را برای پیش‌برد اهداف خود به کار می‌برند و pure fabrication در این سه الگو به کار می‌رود.

Protected Variations

Mediator

در این الگو، mediator یک interfaceای است که از دید همکاران و روی همکاران دیگر کشیده شده است تا پروتوکل ارتباطی توسط همکاران دیده نشود و توسط این interface پیاده‌سازی شود. در نتیجه تغییرات در این پروتکل‌ها به بیرون سرایت نمی‌کند و می‌توان به صورت منعطف، ترتیب‌های مختلف و میانجی‌های مختلف را تغریف کرد و تغییر در میانجی منجر به اتفاق بدی نمی‌شود. هر چند تغییر در کلاس‌های همکار می‌تواند مشکل افرین باشد چرا که coupling شدید میان میانجی و ConcreteColleague وجود دارد. اما می‌توان گفت این الگو از protected variation برای جداسازی همکاران از پروتوکل تعاملی بهره برده است.

Builder

در این الگو، builder ذیل interface تعریف می‌شود و director در سطح بالا دید دارد در نتیجه با گسترش و تغییر در builderها و یا فرایند آفرینش، director تغییری نمی‌کند. هر چند در رابطه‌ی کلاینت با builder، دید concrete وجود دارد اما به طور کلی تعریف builder و محول کردن فرایند آفرینش مصداق استفاده از protected variations است.

Adapter

در این الگو نیز adapter، زیرکلاس‌های target را از هر گونه تغییر مصون می‌دارد و وظیفه‌ی تبدیل را خود بر دوش می‌کشد. در نتیجه با تغییر در فرایند تبدیل، کلاینت و targetها و adaptee تاثیری نمی‌پذیرند. از آنجایی که در حوزه‌ی شی نیز دید adapter به adaptee در سطح بالا است، تغییرات adaptee می‌تواند به adapter منتقل نشود.

مقایسه

هر سه الگو از protected variations بهره می‌برند.

مقایسهٔ الگوهای Decorator، Flyweight و Visitor

دسته - نوع

Decorator

این الگو از دسته‌ی الگوهای ساختاری است و جز wrapperها به شمار می‌رود.
این دسته از الگوها بیشتر به ساختاردهی کلاس‌ها با هدف کاهش coupling و انعطاف‌پذیری بیشتر توجه دارند. این کار را معمولا با جداسازی abstraction از implementation و گسترش ساختار انجام می‌دهند. هر چند ممکن است این الگوها وجوه رفتاری نیز داشته باشند (که در proxy نیز وجه رفتاری جایگزینی مشخص است) اما وجوه ساختاری در آن پررنگ‌تر است.

Flyweight

این الگو از دسته‌ی الگوهای ساختاری است ولی جزو wrapperها به شمار نمی‌رود.
این دسته از الگوها بیشتر به ساختاردهی کلاس‌ها با هدف کاهش coupling و انعطاف‌پذیری بیشتر توجه دارند. این کار را معمولا با جداسازی abstraction از implementation و گسترش ساختار انجام می‌دهند. هر چند ممکن است این الگوها وجوه رفتاری نیز داشته باشند (که در proxy نیز وجه رفتاری جایگزینی مشخص است) اما وجوه ساختاری در آن پررنگ‌تر است.

Visitor

این الگو از الگوهای رفتاری می‌باشد. این الگوها به رفتار اشیا در زمان اجرا توجه دارند و نحوه‌ی تعامل اشیا با یکدیگر تمرکز دارند. هر چند ممکن است وجوه ساختاری نیز داشته باشند اما این وجه در آن‌ها پررنگ‌تر است.
این الگوها، دغدغه‌ی تخصیص مسئولیت‌ها به اشیا، کپسوله‌سازی رفتار شی، اداره‌ی درخواست‌های به یک شی و همچنین مدیریت بهتر تعامل میان اشیا در زمان اجرا با هدف افزایش cohesion و کاهش coupling را دارند.

مقایسه

دو الگوی Flyweight و Decorator ساختاری هستند هر چند flyweight جز wrapperها محسوب نمی‌شود.
الگو visitor اما رفتاری محسوب نمی‌شود.

هدف

Decorator

این الگو یک wrapper است. هدف اصلی این است که بتوانیم در زمان اجرا به یک شی مسئولیت‌های جدیدی اضافه و یا کم کنیم و در واقع بتوانیم کاملا منعطف، رفتارهای بیشتری را به یک شی اضافه کنیم.
در حالت عادی برای این کار یا باید یک ساختار توارثی ایجاد کرد که ترکیبات مختلفی از رفتارها را در قالب زیرکلاس به آن اضافه کنیم که مشکل زیرکلاس ترکیبی به وجود می‌آید و یا باید یک Feature leiden class داشته باشیم که بتوان featureهای مختلف را on و off کرد.
اما به کمک این الگو می‌توان به صورت کاملا منعطف، featureها را در زمان اجرا تغییر داد و featureهای جدیدی را به کمک delegation به یک کلاس اضافه و یا کم کرد. این کار به کمک ساخت زنجیره‌ای از رفتارها در قالب شی انجام شود.

Flyweight

در برخی سیستم‌ها ممکن است مجبور شویم برای عناصر یکسان، اشیا متفاوتی تولید کنیم. در این حالت تعداد زیادی شی در زمان اجرا ایجاد می‌شود که منجر به مصرف حافظه‌ی بالا شده و شاید ما را از پیاده‌سازی شی گرا منصرف کند.
در این حالت، یک دریاچه‌ای از اشیایی که عناصر یکسان ما را نمایندگی می‌کنند ایجاد می‌کنیم و موارد مشترک آن‌ها را در State داخلی این اشیا نگهداری می‌کنیم. در این صورت، به جای آنکه برای عناصر مشترک چندین شی ایجاد کنیم، یک شی به ازای هر کدام از آن‌ها خواهیم داشت و مشخصه‌هایی که در میان این اشیا مشترک نیست را در یک extrinsic State نگهداری می‌کنیم.

Visitor

می‌خواهیم operationهای متنوع و متعددی را روی یک سلسله شی در هنگام پیمایش آن‌ها اعمال کنیم. هر کدام از این اشیا نیز انواع متفاوتی می‌توانند داشته باشند. در حالت پیشین، یک interface کلی برای ساختار اشیا داریم که در interface، عملیات‌های مختلف در قالب متد تعریف شده و در زیرکلاس‌ها پیاده‌سازی شده‌اند. در این حالت، افزودن عملیات‌های جدید بسیار سخت است چرا که تمامی کلاس‌ها و اشیا آن‌ها بایستی تغییر کنند.
این الگو به کمک تعریف یک ساختار متخصص Visitor که به ازای هر نوع المان جدید در ساختار شی‌ها، یک متد در خود تعریف کرده است، رفتارها را به زیرکلاس های این ساختار منتسب می‌کند تا بتوان عملیات‌های جدید را با سهولت تعریف کرد و در هنگام پیمایش روی ساختار اعمال کرد.

مقایسه

هدف flyweight با دو الگوی دیگر کاملا متفاوت است و بیشتر بر روی بهینه کردن مصرف حافظه تمرکز دارد و بهبود استفاده از شی گرایی تمرکز دارد. اما الگوی visitor و decorator از جهاتی به یکدیگر شبیه هستند. هر دو الگو می‌خواهند رفتارهای متفاوتی را در زمان اجرا افزوده و یا کم کنند. هر چند در visitor این رفتارها روی اشیا مختلفی اعمال می‌شود و در نتیجه رفتار اعمالی به شیئی که روی آن اعمال می‌شود وابسته است ولی در Decorator، می‌خواهیم رفتارهای متفاوتی را به یک شی اضافه کنیم و اعمال آن روی تعدادی شی مسئله اصلی نیست. اما می‌توان شباهت میان این دو هدف را در افزودن رفتار دید.

حوزه

Decorator

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

Flyweight

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

Visitor

این الگو در حوزه‌ی object است و به کمک delegation و در زمان اجرا محقق می‌شود.

مقایسه

هر سه الگو در زمان اجرا و به کمک delegation محقق می‌شوند.

کاهش coupling

Decorator

در این الگو coupling بسیار کم است. چرا که دید از کلاینت به decoratorها از طریق interface مربوط به component ایجاد می‌شود و در نتیجه dip برقرار است چرا که زیرکلاس‌ها توسط آن دیده نمی‌شوند و حتی کلاینت تفاوتی میان component و decorator قائل نمی‌شود.
اما پیکربند که لازم است تا زنجیره‌ی decorator و component را ساخته و آن را در اختیار کلاینت قرار دهد، لازم است تا ConcreteDecoratorها را بشناسد و در نتیجه مانند اکثر الگوها dip میان پیکربند و decorator نقض می‌شود که قابل چشم پوشی است.

Flyweight

در این الگو coupling میان client و زیرکلاس‌های flyweight کم است. چرا که دید از کلاینت در سطح بالا به flyweightها است و کاربر interface آن‌ها محسوب می‌شود (البته در نمودار کلاسی کتاب این رابطه غلط نمایش داده شده است). البته یک نکته آن است که کلاینت باید انواع کلاس‌های flyweight را بداند تا بتواند کلید مناسب را به flyweightFactory درخواست دهد. اما نیازی به شناخت زیرکلاس‌ها به صورت مستقیم نیست.
اما در رابطه‌ی میان flywieghtFactory با زیرکلاس‌های flywieght، می‌توان گفت dip نقض می‌شود که طبیعی است. چرا که factory لازم است تا زیرکلاس‌ها را ببیند و از آن‌ها تنها یک instance ساخته و در قالب یک map ذخیره سازی کند و در اختیار کلاینت در زمان مورد نیاز قرار دهد. در نتیجه در اینجا یک coupling مشهود میان flyweight و زیرکلاس‌های آن دیده می‌شود.

Visitor

در این الگو، دید از المان‌های ساختار شیئی به Visitorها در سطح interface است اما از visitorها به المان‌ها، dip برقرار نیست و دید کاملی به ConcreteElements وجود دارد. در نتیجه، visitorها coupling شدیدی با Elementها دارند. در حالت پیشین الگو، رفتار اشیا در المان‌ها تعریف می‌شد و در نتیجه coupling کمتر از حالت پسین بود اما در حالت پیشین انعطاف کمتری داشته‌ایم و تعریف عملیات جدید بسیار سخت می‌شد.
در نتیجه‌ی این coupling، بایستی ساختار المان‌ها ثابت باشد و المان‌های جدید نمی‌توانند به راحتی تعریف شوند چرا که تغییرات آن‌ها به Visitor Interface منتقل می‌شود.

مقایسه

در الگوی decorator، می‌توان گفت coupling تا حد خوبی برقرار است. اما در visitor و flyweight، مشکل coupling وجود دارد. مشکل coupling در visitor که بسیار جدی است و منجر به این می‌شود که نتوانیم به راحتی نوع المان‌های جدیدی به سیستم اضافه کنیم و در نتیجه وابستگی بسیار بالا است. اما در flyweight میان factory و flyweight وابستگی دیده می‌شود که این وابستگی هر چند افزودن flyweightهای جدید را نیازمند تغییر در factory می‌کند، اما این کار هزینه‌ی زیادی برخلاف visitor ندارد.

افزایش cohesion

Decorator

این الگو cohesion بالایی برخوردار است. در حالت پیشین لازم بود تا یا از feature leiden class و یا تعداد زیادی زیر کلاس استفاده کنیم. در حالت feature leiden class، مجبور می‌شدیم رفتارهای مختلف مربوط و نامربوط را تجمیع کنیم که cohesion را از بین برده و منجر به god class می‌شد.
اما در این الگو هر decorator خود یک کلاس متخصص است که یک feature یا قابلیت جدید را نمایندگی می‌کند و در نتیجه cohesion برقرار است.

Flyweight

این الگو نیز تغییری در cohesion حالت پیشین خود ایجاد نمی‌کند ولی از cohesion مناسبی برخوردار است. در واقع کلاس‌های flyweight کاملا cohesive هستند ولی این الگو تمرکز خود را بر بهبود cohesion نگذاشته است.

Visitor

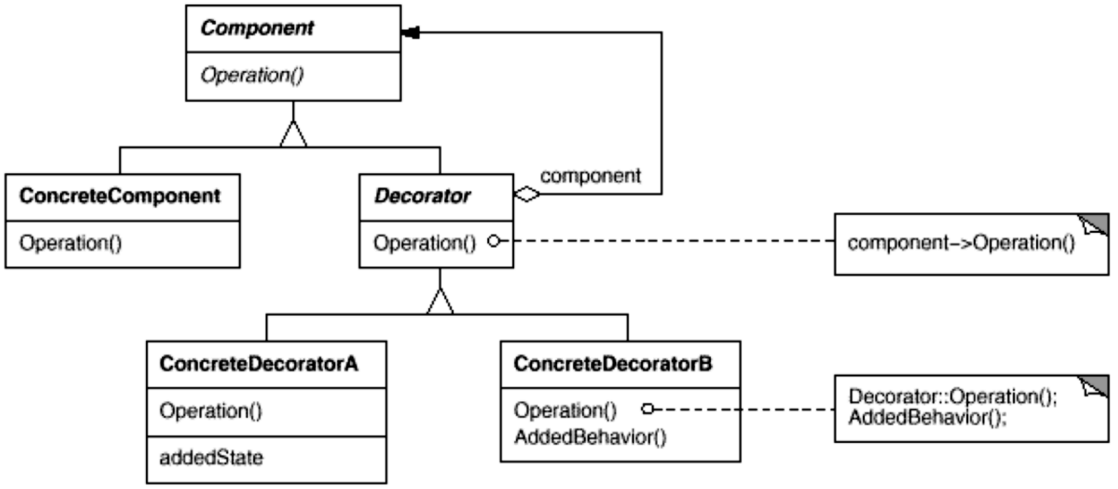
در این الگو با تعریف یک شی متخصص، cohesion به شدت افزایش می‌یابد چرا که Visitor وظیفه‌ی جداسازی operationهایی از زیرکلاس‌های المان را برعهده دارد که وظیفه‌ی ذاتی‌شان نیست و مختص پیمایش است. در نتیجه وظیفه‌مندی المان‌ها به ذات آن‌ها وابسته شده است و visitorها نیز کلاس‌های cohesive هستند که عملیات خاصی را پیاده‌سازی می‌کنند. این cohesion با فدا کردن coupling که در بخش قبل توضیح داده شد به دست آمده است.

مقایسه

هر سه الگو از cohesion مناسبی برخوردار هستند. Decorator و visitor هر کدام کلاس‌های متخصص رفتار را تعریف می‌کنند که خود منجر به بهبود cohesion نسبت به حالت پیشین می‌شود. در flyweight اما cohesion برقرار است هر چند تمرکز این الگو بر cohesive بودن ساختار نیست.

ساختار و رفتار

Decorator

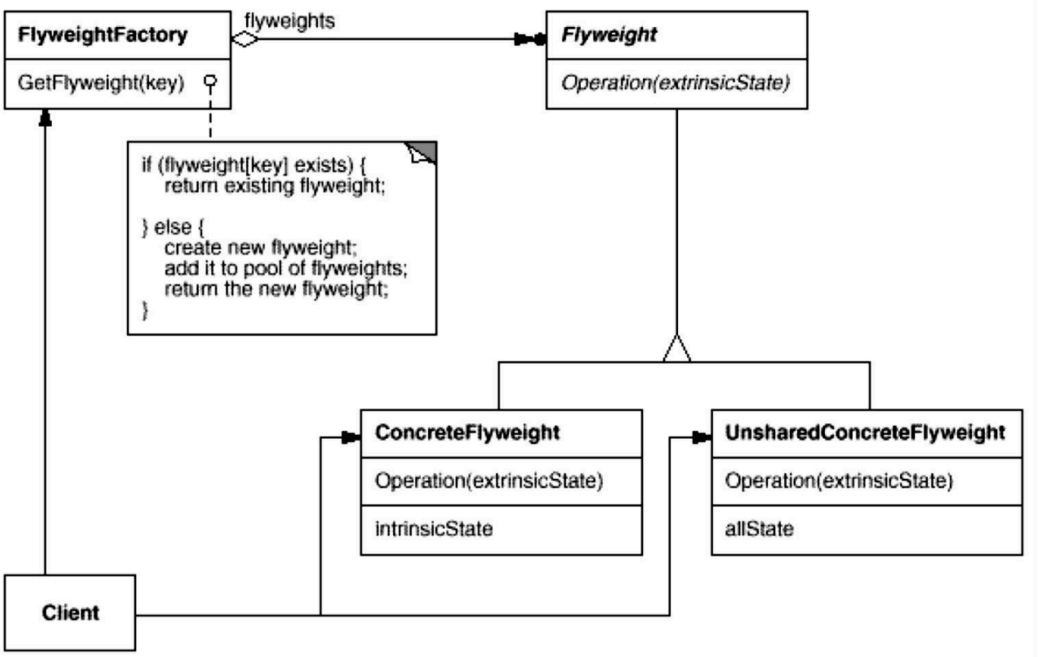


دید کلاینت در ساختار فقط به component یا interface کل مجموعه است. همچنین concrete component همان رفتار اصلی است که decoratorها رفتارهایی را اضافه می‌کنند. در اینجا می‌توان رفتار اضافی را هم به صورت پیش پردازش و هم پس پردازش اضافه کرد. همانطور که می‌بینید decorator از الگوی composite استفاده می‌کند.

Decorator یک رابطه‌ی composite با کلاس پدر دارد. در زیرکلاس‌های decorator، متد operation پیاده‌سازی شده است.

Decorator را هم به صورت پیش پردازش و هم به صورت پس پردازش پیاده‌سازی کرد. در واقع رفتارهای اضافی را پیش یا پس از component انجام دهند. در این نمودار کلاسی، ابتدا ادامه‌ی زنجیره مد نظر قرار گرفته (با فراخوانی Decorator::Operation) و هنگامی که به انتهای زنجیره که ConcreteComponent است رسیدیم و عملیات آن انجام شد، سپس در طول زنجیره باز می‌گردیم و هر decorator عملیات خود را انجام می‌شود.

در صورتی که به صورت پیش پردازش پیاده‌سازی می‌شد، بایستی پیش از فراخوانی عملیات پدر، عملیات خود decorator انجام می‌شد.



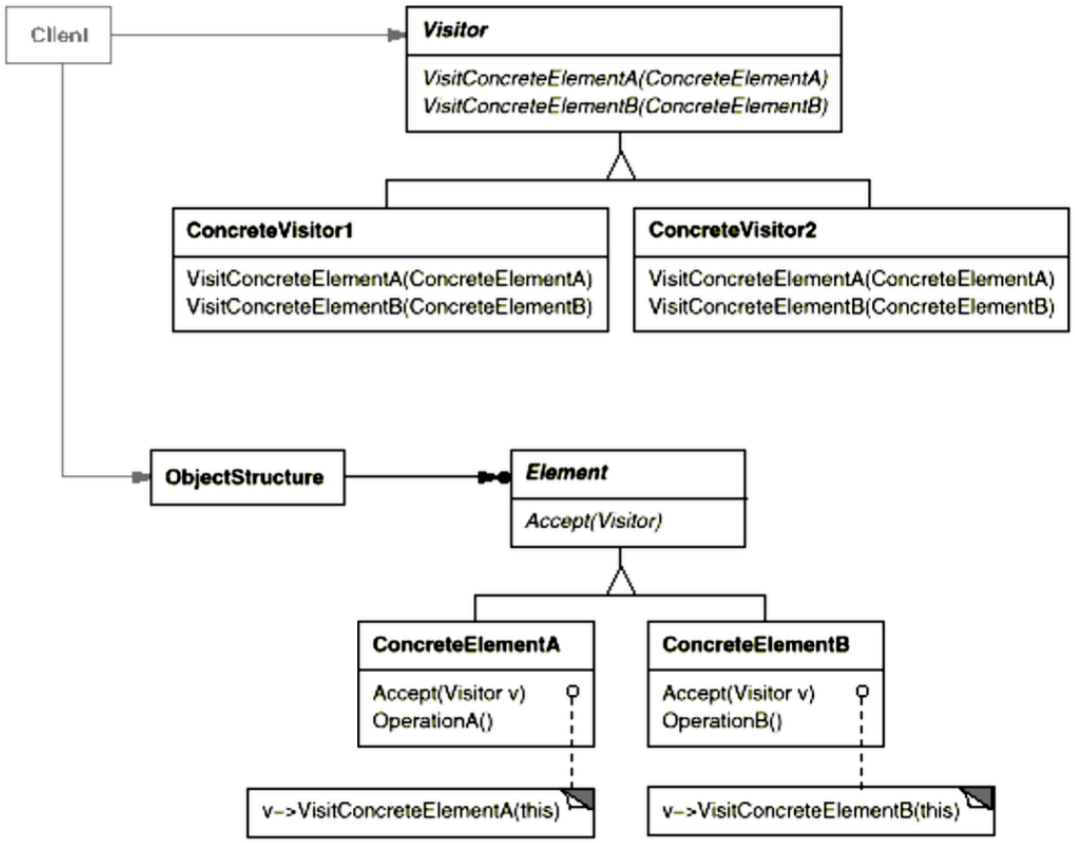
این ساختار که در کتاب آمده اشتباهات زیادی دارد.

یک flyweightFactory داریم که pool را کنترل می‌کند تا اشیا تکراری درست نشود و بر اساس کلید به ما شی flyweight می‌دهد. در این مثال ConcreteFlyweight همان اشیا flywieght هستند که در pool قرار می‌گیرند و UnsharedConcreteFlyweight اشیایی هستند که در کنار flyweight قرار می‌گیرند و برای تکمیل ساختار کلاسی و نگهداری اطلاعات context مربوط به flyweight لازم هستند. همچنین ممکن است که کامپوزیشنی به flyweight داشته باشد و شامل چند flyweight شوند.

داخل ConcreteFlyweight تنها intrinsicState نگهدای شده و در unshared تمام state را نگه می‌دارد.

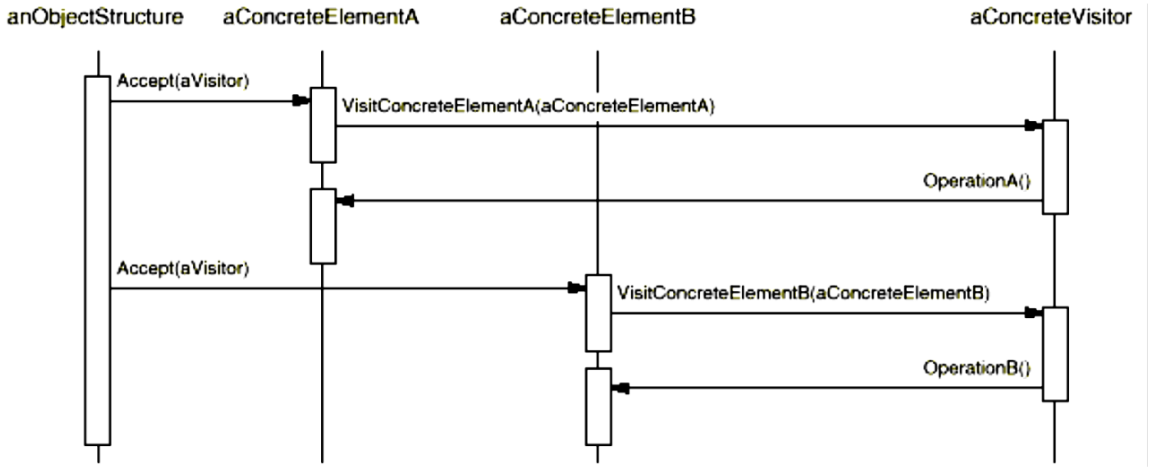
ایراد اصلی این ساختار این است که کلاینت ارتباط association با زیرکلاس‌های flyweight دارد و این مورد نقض dip است چرا که مستقیما با concreteها با خبر می‌شود. بلکه ارتباط کلاینت باید با flyweight باشد.

ایراد دیگر نیز به flyweightFactory مربوط است. این factory تنها دارای instanceهای concrete می‌باشد ولی اینجا نه از unshared اطلاع دارد و نه از concrete و در نتیجه نمی‌تواند از آن‌ها instance بگیرد. کلاس factory اساسا کاربر flyweight نیست بلکه ارتباط آن با concreteFlyweight باید برقرار باشد. وظیفه‌ی این کلاس اساسا ساختن unshared نیست.



کلاینت ساختار شی را می‌بیند تا پیمایش انجام دهد و interface مربوط به visitor را می‌بیند. n البته کلاینت باید زیرکلاس‌های Visitor را ببیند و تخصص آن‌ها را بداند تا instance آن‌ها را به المان‌ها ورودی دهد (accapet). در نتیجه هنگام پیمایش visitor رد می‌شود. در accept، متد operation مربوط به خود از visitor را فراخوانی می‌کنند و خود را نیز به اون نقض می‌کند. البته اینجا encapsulation نقض می‌شود. از سمت element به visitor به صورت کامل dip برقرار است.

اما از ConcreteVisitorها به ConcreteElementها دید کامل دارند و هم operationها و هم attributeهای ان‌ها را می‌بینند. در نتیجه رابطه‌ی میان ConcreteVisitor به ConcreteElementها dependency است و وابستگی شدید دارد.



مطابق نمودار توالی، این جا Objectstructure فرایند پیمایش را انجام می‌دهد. المان به المان accept را با یک visitor مناسب به یک ConcreteElement ارسال می کند. این المان نیز خود تابع operation مربوط به خود را فراخوانی می‌کند و در Visitor نیز این عمل را انجام می‌دهد و پاسخ را باز می‌گرداند.
در مرحله‌ی بعدی همین کار با bVisitor انجام می‌شود. همانطور که میبینید dip از objectStructure به بقیه و همچنینی از ConcreteVisitor نقض می‌شود. در کد Visitor دید کامل به آن‌ها وجود دارد.
از یه منظر دیگر باید بگوییم که کلاینت objectStructure را با یک ویزیتور مناسب پیکربندی می‌کند.

مقایسه

به نوعی می‌توان گفت شباهت زیادی میان الگوها در ساختار دیده نمی‌شود. در decorator از یک composite بهره برده شده است. در visitor دو ساختار توارثی مجزا دیده می‌شود که یکی به المان و دیگری به رفتار مرتبط می‌شود. در حالی که در decorator، رفتار خود زیرکلاسی از component محسوب می‌شود. Flyweight نیز ساختار کاملا متفاوتی دارد.

پیکربندی

Decorator

این الگو به خوبی قابل پیکربندی است. اساسا یکی از اهداف این الگو، ایجاد تغییر در زنجیره‌ی decoratorها برای تغییر در ترکیب رفتارها است. این الگو به خوبی قابل پیکربندی است به گونه‌ای که می‌توان ترتیب و ترکیب و تعداد رفتارها را در زمان اجرا با افزودن اشیا و تغییر آن‌ها در زنجیره‌ی decoratorها عوض کرد.

Flyweight

در این الگو چندان پیکربندی قابل طرح نیست. در واقع هر کلاینت باید بداند از چه flyweightهایی نیاز دارد و آن را استفاده کند.

Visitor

در این الگو، کلاینت لازم است تا ObjectStructure را با یک Visitor مناسب که هدف پیمایش است پیکربندی کند و از آن پس، ساختار شیئی پیمایش را با آن visitor انجام می‌دهد. در حین پیمایش نیز المان‌های مختلف به کمک تابع accept با visitor پیکربندی می‌شوند. به این صورت به راحتی می‌توان در زمان اجرا عملایت تعریف شده را تغییر داد و برای این کار کافی است تا instanceهایی از visitorهای مختلف در اختیار پیمایش‌گر قرار گیرد.

مقایسه

دو الگوی visitor و decorator به خوبی قابل پیکربندی هستند و می‌توان در زمان اجرا رفتارهای انجام شده را تغییر داد. Decorator علاوه بر تغییر رفتارها، می‌تواند رفتارهای جدیدی را به مجموعه اضافه کند و برای این کار کافی است تا در زنجیره‌ی آن، decoratorهای جدید تعریف کند.

انعطاف‌پذیری

Decorator

این الگو بسیار منعطف است و اساسا حالت پیشین خود را به کلی متحول می‌کند. در حالت پیشین اگر از ساختار صلب توارثی بهره می‌بردیم، افزودن و تغییر در رفتار یک معضل بزرگی می‌شد. چرا که انفجار ترکیبی در کمین بود و مجبور بودیم برای ترکیب‌های متنوع رفتاری زیرکلاس‌های مختلفی تعریف کنیم و یک ساختار بسیار غیر منعطف به وجود می‌آمد. در صورتی که از feature leiden class استفاده می‌شد، با افزودن featureها می‌توانستیم منتظر یک god class باشیم که غیر منعطف است و کار را برای ادامه‌ی مسیر افزودن feature دشوار می‌کند.
اما این الگو امکان افزودن، کاستن و تغییر در ترکیب رفتارها را به ما می‌دهد بدون آن که نیاز به تغییر در کد باشد و صرفا با ساختن زنجیره‌های مختلف اشیا decorator، می‌توان به این هدف مهم دست یافت. افزودن componentهای جدید نیز در این الگو بسیار راحت و بی دردسر است و انعطاف در بالاترین شکل ممکن قرار دارد.

Flyweight

انعطاف پذیری در این الگو مناسب است. هر چند افزودن flywieghtهای جدید منجر به شناخت کلاینت از آن برای ساختن کلید می‌شود و همچنین factory باید از آن با خبر باشد. اما به طور کلی تغییر در اشیا flyweight به راحتی قابل انجام است و می‌توان گفت تا حد خوبی flyweight انعطاف پذیری مناسبی دارد. هرچند تمرکز این الگو در بالا بردن انعطاف نبوده است.

Visitor

این الگو انعطاف‌پذیری ساختار پیشین را بسیار افزایش می‌دهد. پس از اعمال این الگو، به راحتی می‌توان با تعریف زیرکلاس های جدید از ساختار Visitor، عملیات‌های متنوع و جدیدی را به پیمایش افزود. چرا که در زمان اجرا با delegation الگو محقق شده و عملیات انجام می‌شود.
اما باید توجه کرد که انعطاف‌پذیری برای تعریف المان‌های جدید مناسب نیست چرا که به ازای هر المان جدید، interface مربوط به Visitorها دستخوش تغییر شده و بایستی متدهای جدیدی به تمامی کلاس‌های Visitor افزوده شود. اما هدف اصلی این الگو بالا بردن انعطاف‌پذیری در تعریف عملیات‌های متنوع و جدید برای پیمایش است و در نتیجه انعطاف به طور کلی بهبود می‌یابد.

مقایسه

دو الگوی visitor و decorator تمرکز بسیاری بر روی انعطاف پذیری دارند و هر دو انعطاف‌پذیری را بسیار می‌افزایند. هر چند visitor نمی‌تواند از افزایش انواع المان جدید پشتیبانی کند، اما تغییر و افزودن رفتارهای جدید به کمک تعریف visitorهای جدید امکان پذیر است و در زمان اجرا نیز بسیار منعطف است.

Decorator اما بر خلاف visitor هم در component و هم در decoratorها امکان تعریف انواع جدید را به ما می‌دهد و بسیار ساختار منعطف‌تری دارد. علاوه بر آن در زمان اجرا نیز بسیار منعطف است و می‌توان زنجیره‌های متنوع رفتاری تعریف کرد.

Flyweight نیز منعطف است اما تمرکز آن برخلاف دو الگوی دیگر بر انعطاف نیست و نمی‌توان از این جهت با آن دو مقایسه کرد.

کارایی

Decorator

در این الگو کارایی به شدت پایین است. در حالت پیشین به کمک یک feature leiden class یا زیرکلاس‌های متعدد لازم بود که مشکل کارایی کمی داشتند اما مشکلات انعطاف پذیری ساختاری دارند و cohesion کمی دارند. اما در اینجا، پس از اعمال الگو می‌توان زنجیره‌ی از decoratorها درست کرد که یک پیام در طول زنجیره حرکت کرده و سپس باز می‌گردد و این انتقال پیام، بزرگتری ضربه زننده به performance است. در نتیجه این الگو می‌تواند تاثیر بسیار منفی روی کارایی برنامه بگذارد و در این زمینه از بدترین الگوها به شما می‌رود.

Flyweight

این الگو در کارایی تاثیر منفی کمی دارد. چرا که کلاینت در حالت پیشین به صورت مستقیم یک شی flyweight ایجاد و استفاده می‌کرد اما پس از اعمال الگو، یک انتقال پیام به factory داریم که بایستی مدیریت pool را انجام دهد. در نتیجه کمی اثر منفی دارد. اما روی حافظه اثر مثبت جدی دارد که در بخش مناسب به آن می‌پردازیم. اثر منفی آن نیز تا حد زیادی قابل چشم‌پوشی است.

Visitor

در این الگو، به جای پیاده‌سازی رفتارها و عملیات‌های پیمایش داخل کلاس المان، آن را به visitorها منتقل کردیم و از طریق delegation آن را محقق می‌کنیم. در نتیجه یک انتقال پیام اضافی نسبت به حالت پیشین ایجاد شده و کارایی الگو کم می‌شود. در نتیجه اگر یک مجموعه‌ی اشیا داشته باشیم، برای پیمایش و اعمال رفتار روی آن‌ها بایستی انتقال پیام‌های اضافی متعددی ببینیم که کارایی الگو را کاهش می‌دهد.

مقایسه

به جز flyweight که تاثیر منفی کمی بر روی کارایی دارد، دو الگوی دیگر تاثیر منفی نسبتا بیشتری روی کارایی دارند. Decorator اما یکی از بدترین الگوها از نظر کارایی می‌تواند باشد چرا که انتقال پیام زنجیره‌ی متعدد خواهد داشت و در نتیجه اصلا از نظر کارایی الگوی مطلوبی نیست.

عواقب و تبعات

Decorator

اثرات مثبت

- از static inheritance خیلی بهتر است. اگر رفتار عوض شود لازم به کشتن آبجکت و ساخت آبجکت جدید نیست و همه‌ی ترکیبات ممکن لازم به پیش‌بینی نیستند.
- از feature leiden class بسیار بهتر است چرا که لازم نیست تمام فیچرها در یک کلاس پیاده‌سازی شود و god class شود.

اثرات منفی

- دکوراتور و component مربوطه‌اش یکی نیستند و اشیا متمایز هستند. اگر بر حسب identity داریم کد می‌زنیم نباید از این الگو استفاده کنیم هر چند استفاده از آن لازم نیست. به این معنی که از identity برای چک کردن آن‌که ابجکتی که در اختیار کلاینت هست عوض شده یا نه، نمی‌توان استفاده کرد. مثلا دفعه‌ی اول فقط textview دیده می‌شود اما بعدا border و ... اضافه می‌شود و آبجکت در دست کلاینت عوض می‌شود و چک کردن این شناسه‌ی آبجکت نمی‌تواند به کار برود.
- تعداد زیادی شی کوچک خواهیم داشت که همگی decorator هستند و هر کدام دنباله‌ای از ابجکت‌ها دارند که منابع زیادی مصرف می‌کند. همچنین وقتی دنباله‌ای داریم، پیغام دارد در زنجیره رفت و برگشتی می‌رود و کند می‌شود. در نتیجه انعطاف پذیری به دست آمده به دلیل هزینه‌ی performance است. اما وقتی تنوع decoratorها کم است می‌تواند بسیار مناسب باشد.

Flyweight

اثرات مثبت

- صرفه‌جویی در مصرف حافظه

اثرات منفی

- در زمان اجرا هزینه‌هایی داریم بخصوص مربوط به extrinsic که وجود دارد و در نتیجه ممکن است چالش ایجاد کند. ترجیح ما محاسبه‌ی آن به جای ذخیره‌سازی است.

Visitor

اثرات مثبت

- می‌توان operationهای جدید به پیمایش افزود بدون تغییر زیاد در سیستم. فقط کلاینت باید آن را بشناسد.
- باعث می‌شود که operationهای مرتبط همگی یکجا جمع شده و در قالب تخصصی Visitor جمع شوند و operationهایی که برای ذات المان هستند داخل المان قرار گیرند. در نتیجه cohesion بالا می‌رود چرا که شی متخصص ایجاد شده است. operationهای مربوط و نامربوط جدا می‌شود. (البته این اتفاق به قیمت بالا رفتن coupling است)

اثرات منفی

- افزودن المان‌های جدید سخت است و باید تغییرات بسیاری در interface مربوط به Visitor و زیرکلاس‌ها انجام داد.
- کپسوله‌سازی نقض می‌شود چرا که رفتاری که از داخل کلاس‌ها استخراج شده و به visitor نسبت داده شده نیازمندی اطلاعات داخلی کلاس‌ها است و در نتیجه لازم است از visitor دید به المان‌ها باشد.

مقایسه

هر کدام اثرات مثبت و منفی مربوط به خود را دارند. اثر مثبت مشترک visitor و component، افزودن رفتارهای متنوع و قابل تغییر است.
در باقی موارد اشتراکات زیادی ندارند که اشاره شده است.

encapsulation

Decorator

در این الگو encapsulation نقض نمی‌شود چرا که دید میان decoratorها، کلاینت و component به یکدیگر در سطح بالا است و state و مشخصه‌های داخلی آن‌ها در دسترسی دیگری قرار نمی‌گیرد.

Flyweight

در این الگو نیز مانند decorator، مشخصه‌های داخلی در دسترس یکدیگر قرار نمی‌گیرد. هر چند دید factory به ConcreteFlyweightها برقرار است اما مشخصه‌های آن در دسترسی factory نیستند و در نتیجه encapsulation به صورت کامل برقرار است.

Visitor

یکی از نقاط ضعف جدی این الگو نقض encapsulation است. در حالت پیشین عملیات‌ها در کلاس‌های المان تعریف شده و در نتیجه این عملیات‌ها به attributeها و مشخصات اشیا دسترسی کامل داشتند. پس از اعمال الگو لازم است تا کلاس‌های ConcreteVisitor، نسبت به ConcreteElementها نیز به صورت کامل دید داشته باشند تا بتوانند عملیات را پیاده‌سازی کنند و در نتیجه مشخصات المان‌ها در دسترس این کلاس قرار گرفته و encapsulation کاملا نقض می‌شود.

مقایسه

الگوی visitor به صورت آشکار encapsulation را نقض می‌کند اما دو الگوی دیگر به خوبی این قاعده را رعایت می‌کنند.

انتشار تغییرات

Decorator

تغییرات در component و decoratorها به کلاینت منتشر نمی‌شود چرا که dip کاملا برقرار است. از طرفی تغییرات در میان آن‌ها نیز منتشر نمی‌شود چرا که دید میان اشیا در سطح بالا و از طریق interface مربوط به component برقرار است.

Flyweight

تغییرات در ConcreteFlyweightها می‌تواند به factory منتشر شود. اما میان factory و کلاینت و همچنین flywiegh و کلاینت، dip برقرار است و در نتیجه تغییرات دیگر اشیا در کلاینت منتشر نمی‌شود. هر چند کلاینت لازم است تا کلید flyweightهای جدید را بداند.

Visitor

در این الگو به علت آنکه Elementها در سطح بالا به Visitor دید دارد در نتیجه تغییرات Visitor و عملیات‌ها به ان‌ها منتشر نمی‌شوند. همچنین رابطه‌ی میان Elements و کلاینت‌ها نیز سطح بالا است و ObjectStrcuture نیز نسبت به تغییرات المان‌ها منتزع می‌ماند.
اما تغییرات در المان‌ها منجر به تغییرات در Visitor خواهد شد چرا که دید Concrete به Concrete از Visitor به Elements وجود دارد و در نتیجه، به علت وابستگی شدید و نقض dip در رابطه از Visitor به Elements، تغییرات به راحتی منتشر می‌شوند. از طرفی کلاینت نیز لازم است تا به ConcreteVisitor دید داشته باشد تا از آن instantiate کند و تغییرات در Visitor می‌تواند به کلاینت نیز منتقل شود.

مقایسه

Decorator از این منظر در بهترین شرایط قرار دارد. Flyweight تنها در factory مشکل انتشار تغییرات را دارد اما visitor، اشیا visitor کاملا نسبت به تغییرات element حساس هستند و dip نقض می‌شود.

میزان استفاده از منابع سیستمی

Decorator

این الگو از نظر منابع سیستمی نیز تعریف چندانی ندارد. مانند بخش کارایی، ممکن است برای تحقق الگو اشیا متعددی در قالب یک زنجیره‌ی composite و پشت سر هم قرار گیرند که هم انتقال پیام زیادی دارند که روی منابع پردازشی اثر منفی بسیار بدی دارد و هم حافظه‌ی بیشتری مصرف می‌کنند. در نتیجه این الگو تاثیر منفی جدی بر روی حافظه و منابع سیستمی دارد.

Flyweight

این الگو به بهبود مصرف حافظه کمک شایانی می‌کند و اساسا هدف الگو کاهش مصرف منابع سیستمی است. برای مثال تلاش می‌کنیم intrinsic stateها بیشینه ممکن باشد و همچنین هدف الگو ایجاد نکردن متعدد اشیا یکسان برای contextهای مختلف است به گونه‌ای که اطلاعات shared آن‌ها را تنها در یک شی نگهداری کنیم و در نتیجه، برای اشیا یکسان پس از اعمال الگو، اشیا مختلفی نمی‌سازیم و همگی را در یک pool نگهداری می‌کنیم. در نتیجه هدف اصلی این الگو بهبود مصرف حافظه است و نتیجه مثبت روی استفاده از منابع سیستمی دارد.

Visitor

در این الگو، می‌توان به ازای هر عملیات و برای کل پیمایش، یک visitor استفاده کرد. در نتیجه تا حدی از استفاده visitorهای متعدد برای یک کاربرد می‌توان اجتناب کرد اما در مجموع نسبت به حالت پیشین الگو، یک شی بیشتری وجود خواهد داشت. هرچند شایان ذکر است که این شی، خود اطلاعات زیادی را نگهداری نمی‌کند بلکه بیشتر کپسول‌های رفتاری هستند که متخصص انجام عملیات هستند. در نتیجه اثر منفی زیادی بر حافظه ندارد.

مقایسه

الگوی flyweight، مصرف منابع را به حد قابل توجهی کاهش می‌دهد و الگوی بسیار مناسبی است. Visitor تاثیر منفی قابل چشم‌پوشی بر مصرف حافظه دارد اما decorator هم منابع حافظه و هم منابع پردازشی جدی مصرف می‌کند در نتیجه اثر منفی چشم‌گیری دارد.

استفاده از شی جعلی

Decorator

در دامنه‌ی تحلیل، decoratorها که wrapper هستند دیده نمی‌شود و صرفا رفتارهای component مشاهده می‌شود. در زمان طراحی و برای بالا بردن انعطاف مجموعه و تحقق هدف الگو، decoratorها تعریف شده و در زمان اجرا این اشیا متخصص استفاده می‌شوند تا رفتارهایی را به مجموعه بیافزایند. در نتیجه در مجموع decoratorها اشیا جعلی هستند و این الگو از شی جعلی استفاده می‌کند.

Flyweight

اشیا ConcreteFlyweights و FlyweightFactory در دامنه‌ی مسئله و در مرحله‌ی تحلیل نیستند و در طراحی و برای بهبود استفاده از منابع سیستمی تعریف می‌شوند. در نتیجه این الگو حداقل دو شی جعلی جدید دارد.

Visitor

این الگو نیز مانند بسیاری از الگوها با تعریف ساختار cohesive جدید که در دامنه‌ی مسئله وجود نداشته و مختص مرحله‌ی طراحی است، operationهایی که در ذات المان‌ها نیستند را از آن‌ها جدا کرده و به کلاس‌های متخصص visitor منتسب می‌کند. در نتیجه این الگو نیز یک شی جعلی متخصص تعریف می‌کند.

مقایسه

هر سه الگو از اشیا جعلی جدیدی برای تحقق خود استفاده می‌کنند. در visitor و decorator اشیا جعلی نماینده‌ی عملیاتی هستند که می‌خواهد انجام شود و در flyweight، دو شی جعلی داریم که یکی آفرینشی و دیگر نماینده‌ی یک شی مورد استفاده است.

سادگی پیاده‌سازی

Decorator

این الگو با امکانات زبان‌های شی گرا به سادگی قابل پیاده‌سازی است و کافی است تا زبان از inheritance پشتیبانی کند و یا interface قابل تعریف باشد.
هر چند استفاده از این الگو یک ملاحظاتی دارد. باید توجه کرد که این الگو می‌تواند performance برنامه را با ساختن زنجیره‌ی طولانی اشیا دکوراتور و در نتیجه زنجیره‌ی طولانی از فراخوانی توابع و بازگشت آن‌ها کاهش دهد. در نتیجه باید در زمان‌هایی استفاده کرد که performance مسئله اصلی نیست و یا زنجیره طولانی نمی‌شود.

Flyweight

این الگو نیز به راحتی با امکانات شی گرایی قابل پیاده‌سازی است و پیچیدگی زیادی ندارد.

Visitor

این الگو به راحتی با امکانات شی‌گرایی مانند inheritance در زبان‌های شی گرا قابل پیاده‌سازی است و پیاده‌سازی آن پیچیدگی زیادی ندارد. کافی است تا عملیات‌های مربوط به پیمایش و غیرذاتی را از المان‌ها جدا کرده و در ساختار توارثی Visitor قرار دهیم.

مقایسه

هر سه الگو به سادگی به زبان‌های شی‌گرا قابل پیاده‌سازی هستند و پیاده‌سازی آن‌ها ساده است. هر چند مخاطرات performance در دکوراتور مهم است و باید در هنگام پیاده‌سازی و طراحی به آن توجه شود.

موارد کاربرد

Decorator

- برای اینکه مسئولیت را در زمان اجرا به صورت پویا و شفاف بدون تاثیر در کلاینت، به اشیا اضافه کرد و کافی است که کلاینت به سر زنجیره اشاره کند.
- مسئولیت‌ها قابل حذف شدن هستند و می‌توان رفتار را حذف کرد و برداشت.
- وقتی که ترکیب‌های متعددی از رفتارها وجود دارد این کار بسیار کمک می‌کند و ویژگی جدید نیاز به subclassهای جدید ندارد بلکه با یک کلاس decorator انجام می‌شود. در صورتی که از subclass و extenstion استفاده کنیم، ممکن است که یک مجموعه زیادی از subclass تولید شود که منجر به explosion of subclass شود.

Flyweight

- وقتی که یک application از تعداد زیادی object استفاده می‌کند و این اشیا ریزدانه هستند.
- هزینه‌ی نگهداری آن‌ها بالا است به دلیل تعدد اشیا و حجم آن‌ها از نظر هزینه ذخیره سازی.
- بخشی از state را باید بتوان intrinsic کنیم که بتوانیم sharing داشته باشیم. نکته‌ای که در کتاب به صورت اشتباه آمده که extrinsic بسیار بزرگ باشد که درست نیست.
- تعداد زیادی از اشیا رو می‌توانیم با تعداد کمتری از اشیا مشترک جایگزین کنیم.
- البته باید دقت کنیم که نباید اپلیکیشن به identity آبجکت flyweight وابسته باشد. مانند decorator. چرا که در مثال حروف الفبای کتاب، همه‌ی aهایی که داریم در یک جا قرار دارند و اشیا aهای مختلف نباید یکسان در نظر گرفته شود ولی در واقع context آن‌ها موجب تفاوت می‌شود.

Visitor

- یک ساختار شیئی داریم که می‌خواهیم تعداد زیادی شی از کلاس‌های مختلف Elementها آن را تشکیل می‌دهند و می‌خواهیم operationهایی را در هنگام پیمایش این ساختار انجام دهیم که به کلاس آن‌ها بستگی دارد. در نتیجه خود کلاس‌ها که هویت مختلف دارند و تنها اشتراکشان المان یک ساختار شیئی بودن است. پیش از الگو مجبوریم زیرکلاس‌های element تعریف کنیم و برای هر کدام operationهای خاص را تعریف کنیم که ساختار صلب و سختی هستند و operation جدید به راحتی تعریف نمی‌شود.
- تعداد زیادی operation بی ربط به یکدیگر وجود دارد که باید روی اشیا یک ساختار شی اجرا شود و نمی‌خواهیم کلاس‌های element را با آن آلوده کنیم از این جهت که این operationها وابسته به پیمایش هستند و متعلق به المان‌ها نیستند در نتیجه cohesion المان‌ها را خراب می‌کنند.

- کلاس‌هایی که یک ساختار شیئی را تعریف می‌کنند خود به ندرت عوض می‌شوند و نوع جدیدی از آن‌ها تعریف می‌شود بلکه چیزی که تغییر می‌کند و جدیدتر می‌شود، operartionها هستند.

مقایسه

کاربرد مشترک میان decorator و visitor، افزودن عملیات یا رفتارهای جدید و قابلیت پیکربندی آن است. باقی موارد چندان قابل مقایسه نیستند و هر کدام به صورت مجزا به آن‌ها اشاره شده است.

الگوهای مرتبط

Decorator

به chain of responsibility بسیار شباهت دارد. هر دو با یک زنجیره‌ای از اشیا کار می‌کنند و یک پیام را در طول زنجیره منتقل می‌کنند هر چند در cor ممکن است در اواسط مسیر عملیات انجام شده و دیگر ادامه پیدا نکند اما decorator روی همه‌ی اشیا اجرا می‌شود.
با composite نیز در ارتباط است و در ساختار خود از composite بهره می‌برد.
از prototype نیز در کنار این الگو می‌توان استفاده کرد. چرا که اشیا متفاوت با خصوصیات را تولید کنیم بدون آنکه فرایند ایجاد را از ابتدا طی کنیم و صرفا کپی می‌گیریم.
از نظر ساختار الگو، بسیار به proxy شباهت دارد. در واقع هر دکوراتور component را جعل می‌کند و رفتاری می‌افزاید اما proxy زنجیره‌ای از اشیا را شامل نمی‌شود و تنها روی یک شی تعریف می‌شود.

Flyweight

به نوعی با singleton در ارتباط است. در singleton می‌خواهیم یک شی را تنها یک بار بسازیم. در flyweight اما یک flyweight pool داریم که از هر SharedConcreteFlyweight تنها یک شی می‌سازیم. هدف دو الگو اما کاملا متفاوت است و ساختارهای متفاوتی دارند.

Visitor

با command همانطور که گفته شد در ارتباط است. چرا که هر المان، خود را در اختیار visitor قرار می‌دهد تا یک عملیات انجام شود. Command نیز receiver را دریافت می‌کند تا یک عملیات روی آن انجام دهند.
بر روی یک زنجیره‌ی composite نیز می‌توان از visitor برای اعمال رفتار استفاده کرد.
با iterator نیز در ارتباط است چرا که می‌توان در کنار iterator از آن استفاده کرد و یک عملیات را در هنگام پیمایش بر روی یک شی مرکب اعمال کرد.

مقایسه

Visitor و decorator هر دو با composite در ارتباط هستند. در باقی موارد شباهت و نقاط یکسانی میان الگوها دیده نمی‌شود.

OCP

Decorator

در این الگو، کلاینت به component که واسط سطح بالا است دید دارد و نسبت به زیرکلاس‌های آن و decoratorها منتزع است. در نتیجه dip بین کلاینت و ساختار توارثی component وجود دارد و با گسترش component و decoratorها، کلاینت تغییری نمی‌کند.
همچنین گسترش decorator نیز تاثیری روی componentهای دیگر ندارد و در نتیجه، به خوبی ocp در این الگو برقرار است. البته نکته‌ی مهم، پیکربند ثالث است که برای تحقق الگو لازم است تا زنجیره‌ی decorator و component را بسازد. این پیکربند رابطه‌ی dip با ساختار ندارد و برای آن ocp نقض می‌شود. اما این موضوع پذیرفته است و در نتیجه در مجموع ocp برقرار است.

Flyweight

در این الگو، کلاینت دید به flyweightFactory و کلاس سطح بالا (یا واسط) flyweight دارد و نسبت به ConcreteFlyweightها منتزع است. در نتیجه dip برقرار بوده و گسترش ساختار flyweight، منجر به تغییر در کلاینت نمی‌شود. شایان ذکر است هرچند که کلاینت نیازی به شناخت زیرکلاس‌های flyweight ندارد، اما باید انواع آن‌ها را بداند تا از طریق factory و با در اختیار دادن کلید مناسب، بتواند flyweight مناسب را بدست آورد. اما در مجموع می‌توان گفت ocp برقرار است.
از دید factory اما dip برقرار نیست. برخلاف شکل ساختار این الگو در کتاب، factory باید به ConcreteFlyweightها دید مستقیم داشته باشد تا آن‌ها را بسازد. در نتیجه گسترش Flyweight منجر به تغییر در factory می‌شود. می‌توان گفت ocp برای factory برقرار نیست.

Visitor

در این الگو کلاینت دید سطح بالا به object structure دارد اما از آن‌جایی که نقش پیکربندی را برعهده دارد، بایستی زیرکلاس‌های visitor را نیز ببیند تا پیمایش را اغاز کند. در نتیجه dip نسبت به visitor ندارد و گسترش ساختار visitor، منجر به تغییر در کلاینت می‌شود.
در رابطه‌ی بین visitor و Elementها، باید گفت که مستقیم با یکدیگر در ارتباط نیستند. گسترش visitorها بدون تغییر در Element یکی از اهداف این الگو بوده و در نتیجه Elementها رابطه‌ی ocp با visitor دارند. اما همانطور که پیش‌تر اشاره شد، گسترش ساختار Elementها منجر به تعریف operation جدید در واسط visitor می‌شود که بایستی توسط visitorهای مختلف یا در کلاس پدر پیاده‌سازی شود و در نتیجه، گسترش Element منجر به تغییر در visitor می‌شود. همچنین visitor وابستگی شدیدی به المان دارد به حدی که encapsulation

نقض می‌شود. در مجموع می‌توان گفت ocp برقرار نیست و با گسترش ساختارها، تنها Elementها تغییری نمی‌کنند.

مقایسه

تنها برای decorator، این قاعده برقرار است. در flyweight توسط factory و در visitor، در رابطه‌ی کلاینت با visitor و visitor با Elementها، ocp نقض می‌شود.

LSP

Decorator

بستگی به طراحی، ممکن است decoratorها از نظر منطقی رابطه‌ی is a با component نداشته باشند. اما در مثال‌های مورد بررسی و کاربردهای معمول، اینطور به نظر می‌رسد که decoratorها خود یک component هستند و در نتیجه می‌توانند به جای آن به کار روند. همچنین زیرکلاس‌های decorator خود نیز رابطه‌ی is a با decorator بایستی داشته باشند. در مجموع می‌توان گفت که lsp برقرار است.

Flyweight

در این الگو، ConcreteFlyweightها همگی رابطه‌ی is a با flyweight دارند. در واقع زیرکلاس‌های flyweight به جای flyweight می‌توانند به کار روند. در باقی موارد نیز به ساختار توارثی دیده نمی‌شود و در نتیجه این الگو از lsp پیروی می‌کند.

Visitor

در این الگو نیز ConcreteElementها همگی رابطه‌ی is a با Element دارند و ConcreteVisitorها نیز رابطه‌ی is a با Visitor دارند و در نتیجه lsp در این الگو برقرار است.

مقایسه

در هر سه الگو، lsp برقرار است.

DIP

Decorator

در این الگو، همانطور که در ocp بیان شد، رابطه‌ی میان کلاینت با component و decoratorها در سطح بالا است. در نتیجه dip برقرار است. تنها نکته‌ی منفی این الگو که در اکثر الگوهای کتاب دیده می‌شود، وابستگی شدید پیکربند به decoratorها و componentها است که قابل چشم پوشیست.

Flyweight

در این الگو در رابطه‌ی میان کلاینت و flyweightها و factory، رابطه‌ی سطح بالا است و در نتیجه دید concrete به concrete وجود ندارد و dip برقرار است. اما factory لازم است تا زیرکلاس‌های flyweight را بشناسد و در نتیجه، dip در این رابطه برقرار نیست.

Visitor

در این الگو، dip میان کلاینت و elementها برقرار است. اما کلاینت که نقش پیکربندی ساختار شی با visitor مناسب را دارد، بایستی به زیرکلاس‌های visitor دید داشته باشد و در نتیجه dip نقض می‌شود. از طرفی visitorها نیز با گسترش ساختار elementها، لازم است تا در operationها واسط خود تغییر ایجاد کرده و خود نیز آن را پیاده‌سازی کنند. در نتیجه dip نقض می‌شود.

مقایسه

تنها در decorator، این قاعده برقرار است و در دو الگوی دیگر نقض می‌شود.

ISP

Decorator

در این الگو، component و decorator هر کدام interfaceهای تک مسئولیتی هستند که cohesion مناسبی دارند. در واقع componentهای کارهای نا مرتبط با خود را انجام نمی‌دهند و پیاده‌سازی decoratorها نیز ذیل یک composite و زیرکلاس decorator تعریف شده اند و در نتیجه isp به خوبی دیده می‌شود.

Flyweight

در این الگو نیز به خوبی واسط ها از یکدیگر جداسازی شده و کلاس‌ها نقش‌های مربوط به خود را ایفا کرده و cohesive هستند در نتیجه isp برقرار است.

Visitor

در سمت elementها، به خوبی کارهای مرتبط با آن‌ها قرار گرفته و از کلاس‌های بزرگ جلوگیری می‌شود. در حالت پیشین لازم بود برای هر element، نحوه‌ی پیمایش را نیز پیاده سازی کنیم که در واقع وظیفه‌ی visitor خود برعهده‌ی element بود. در سمت visitor نیز می‌توانیم بگوییم که تا حدی برقرار است. هر چند با گسترش ساختار element مجبور به تعریف operationهای جدید می‌شویم، اما این ساختار خود در نتیجهٔ جداسازی واسط visitor از element بوده که در مجموع به cohesion مجموعه کمک کرده است. در نتیجه می‌توان گفت افزودن operationهای جدید نیز از cohesion مربوط به visitor کم نمی‌کند و isp برقرار است.

مقایسه

در هر سه الگو isp برقرار است و به خوبی تقسیم وظایف میان interfaceها صورت گرفته است.

CRP

Decorator

این الگو به کمک ساختن یک زنجیره‌ی decorator و component محقق می‌شود که هر decorator، به decorator بعدی دسترسی دارد تا انتهای زنجیره که یک component قرار می‌گیرد. این الگو به کمک delegation انجام می‌شود. در نتیجه crp دیده می‌شود.

Flyweight

در این الگو نیز ساختارهای delegation با تخصیص شی از flyweight به کلاینت و در زمان اجرا تحقق می‌یابد و در نتیجه این الگو crp را رعایت می‌کند و delegation بر ساختارهای صلب توارثی ترجیح دارد.

Visitor

در این الگو، در حالت پیش از اعمال الگو پیاده‌سازی روش‌های عملیات در زمان پیمایش ساختار شیئی داخل element ها و به کمک ساختارهای صلب توارثی محقق می‌شد. اما پس از اعمال الگو، با delegate کردن یک شی visitor و استفاده از آن برای پیمایش، منجر به تحقق الگو در زمان اجرا و رعایت crp می‌شود.

مقایسه

هر سه الگو از crp به بهترین شکل بهره می‌برند.

PLK

Decorator

در این الگو یک زنجیره‌ی شی ساخته می‌شود تا decoratorهای مختلف در کنار component عمل کنند در نتیجه مستعد نقض کردن این قاعده و وقوع دید ترایا هستند اما در این الگو دید ترایا دیده نمی‌شود و message chain به وجود نمی‌آید چرا که پیام در طول زنجیره منتقل می‌شود و هیچ decoratorی شی بعدی خود را به شی قبلی پاس نمی‌دهد.

Flyweight

در این الگو نیز دید ترایا دیده نمی‌شود. اساسا زنجیره‌ی شی نداریم و در نتیجه هیچ شیئی، شی بعدی را به قبلی پاس نمی‌دهد و message chain به وجود نمی‌آید و در نتیجه plk رعایت می‌شود.

Visitor

در این الگو نیز message chain نداریم. مانند دو الگوی قبلی، دید ترایا وجود ندارد. هر چند یک سلسله شی داریم که روی آن عملیات را به کمک visitor پیاده می‌کنیم اما این سلسله‌ی اشیا قاعده‌ی plk را نقض نمی‌کنند. شاید در نگاه اول اینطور به نظر بیاید که ConcreteElement خود را در اختیار visitor قرار می‌دهد تا عملیات را انجام دهد در نتیجه دید ترایا به وجود آمده است. اما این درست نیست چرا که دید غیر مانا از visitor به element برای تحقق الگو لازم است و در صورتی دید ترایا به وجود می‌آمد که visitor یک عملیاتی را برای کلاینت تعریف می‌کرد و در نتیجه‌ی آن عملیات، element را در اختیار کلاینت قرار می‌داد. در آن صورت دید ترایا به وجود می‌آمد چرا که علتی برای دید مستقیم میان کلاینت و element نبود. در نتیجه در مجموع می‌توان دید که message chain وجود ندارد و دید ترایا به وجود نمی‌آید.

مقایسه

هر سه الگو، plk را رعایت می‌کنند. هر چند decorator مستعد نقض این الگو است و visitor در ظاهر آن را نقض می‌کند، اما با بررسی دقیق‌تر و متوجه شدن روابط میان سلسله‌ی اشیا می‌توان دید که این قاعده در هر سه الگو برقرار است.

مقایسهٔ الگوهای Decorator، Flyweight و Visitor بر اساس grasp

Information Expert

Decorator

در این الگو، اشیا decorator و component همگی اطلاعات لازم برای اعمال رفتار مد نظر خود را دارند. همچنین برای اعمال رفتار شی بعدی در سلسله‌ی شی، فراخوانی روی شی بعدی انجام می‌شود. در نتیجه information expert به خوبی رعایت شده و همه‌ی اشیا کپسول‌های داده رفتار هستند.

Flyweight

در این الگو می‌توان نقض این قاعده را دید. چرا که extrinsic state، یک مثال از داده‌ای است که برای اعمال رفتار flyweight لازم است اما مستقیما در اختیار آن قرار نمی‌گیرد تا بتوان در مصرف حافظه صرفه جویی کرد و دریاچه‌ی اشیا شکل بگیرد. در نتیجه این قاعده با یک مصلحت مهم‌تری نقض می‌شود.

Visitor

در این الگو نیز این قاعده نقض شده است. چرا که visitorها به اطلاعاتی برای انجام عملیات نیاز دارد که در اختیار elementها قرار دارد. حتی در حالت پیشین الگو نیز رفتار داخل element تعریف شده است اما پس از اعمال الگو، رفتار را از داده جدا کرده‌ایم تا انعطاف‌پذیری الگو بالاتر رود. در نتیجه این الگو با مصلحت بالاتری، information expert را نقض می‌کند.

مقایسه

در flyweight و visitor این الگو استفاده نشده است اما در decorator استفاده از آن به خوبی دیده می‌شود.

Creator

Decorator

در این الگو وظیفه‌ی ساخت سلسله‌ی اشیا با پیکربند ثالث است. در حالی که رابطه‌ی decoratorها با یکدیگر و component، یک رابطه‌ی aggregation (حتی شاید composition) است که اولویت ۱ یا ۲ دارد. حتی کلاینت یک دید مانا به component با اولویت ۴ دارد. در نتیجه این الگو در decorator نقض می‌شود.

Flyweight

در این الگو وظیفه‌ی ساخت اشیا ShareConcreteFlyweightها با FlyweightFactory است که از آن‌جایی که flyweightFactory خود وظیفه‌ی کنترل pool اشیا را برعهده دارد، در نتیجه یک رابطه‌ی aggregation با آن‌ها دارد. در نتیجه بالاترین اولویت را دارد و creator را عملیاتی می‌کند. یک دید مانا از کلاینت به سطح بالای flyweightها نیز وجود دارد که از آن جایی که اولویت پایین‌تری نسبت به factory دارد، این الگو از creator بهره می‌برد.

Visitor

در این الگو وظیفه‌ی ساخت visitor با کلاینت است که اولویت ۵ را دارد اما دید مانا میان element و visitor منجر می‌شود که element از اولویت ۳ برخوردار باشد که در اینجا منجر می‌شود الگوی creator توسط این الگو نقض می‌شود.

مقایسه

تنها flyweight از visitor استفاده می‌کند و در باقی الگوها، creator نقض می‌شود.

Low Coupling

Decorator

در این الگو، coupling میان کلاینت و componentها و decorator در پایین ترین سطح است چرا که کلاینت تنها واسط component را می‌بیند. در میان decorator و component اما یک رابطه‌ی inheritance وجود دارد که coupling دارد اما طبیعت الگو است و چون رابطه‌ی is a برقرار است، coupling قابل پذیرفتن است.
هر چند پیکربند بایستی به تمام زیرکلاس‌های decorator و component دید دارد، اما این مورد قابل چشم پوشی است و در مجموع این الگو از coupling پایینی برخوردار است.

Flyweight

این الگو coupling بالایی میان flyweightFactory و SharedConcreteFlyweightها برقرار است. اما در میان کلاینت و اشیا flyweight دید سطح بالا وجود دارد و coupling کم است. در مجموع باید گفت coupling به صورت مطلق پایین و بالا نیست اما در سطح قابل قبولی قرار دارد.

Visitor

در این الگو، رابطه‌ی میان کلاینت با زیرکلاس‌های visitor، وابستگی شدیدی وجود دارد چرا که باید instantiate شود. دید elementها به visitor در سطح بالا است اما visitor دید مستقیم به المان‌ها دارد به حدی که encapsulation را نقض می‌کند. در نتیجه این الگو نیز low coupling را نقض می‌کند. هر چند باید توجه کرد که در حالت پیشین، ساختار صلب توارثی عملیات‌های مختلف را پیاده‌سازی می‌کرد و چون inheritance بالاترین نوع coupling است، پس از اعمال الگو، coupling کمتر شده است اما همچنان coupling نسبتا بالایی دارد.

مقایسه

Decorator از coupling کمتری نسبت به دو الگوی دیگر برخوردار است. الگوی flyweight یک وابستگی شدید میان factory و flyweightهای shared دارد و کمی از decorator وابستگی بیشتری دارد. اما visitor وابستگی به مراتب بیشتری نسبت به دو الگوی دیگر دیده می‌شود.

High Cohesion

Decorator

این الگو cohesion بالایی دارد. در حالت پیشین لازم بود تا یا از feature leiden class و یا تعداد زیادی زیر کلاس استفاده کنیم. در حالت feature leiden class، مجبور می‌شدیم رفتارهای مختلف مربوط و نامربوط را تجمیع کنیم که cohesion را از بین برده و منجر به god class می‌شد.
اما در این الگو هر decorator خود یک کلاس متخصص است که یک feature یا قابلیت جدید را نمایندگی می‌کند و در نتیجه cohesion بالا است.

Flyweight

این الگو نیز تغییری در cohesion حالت پیشین خود ایجاد نمی‌کند ولی از cohesion مناسبی برخوردار است. در واقع کلاس‌های flyweight کاملا cohesive هستند و در نتیجه cohesion بالا است.

Visitor

در این الگو با تعریف یک شی متخصص، cohesion به شدت افزایش می‌یابد چرا که Visitor وظیفه‌ی جداسازی operationهایی از زیرکلاس‌های المان را برعهده دارد که وظیفه‌ی ذاتی‌شان نیست و مختص پیمایش است. در نتیجه نسبت به حالت پیشین، cohesion بالاتری دارد و در نتیجه وظیفه‌مندی المان‌ها به ذات آن‌ها وابسته شده است و visitorها نیز کلاس‌های cohesive هستند که عملیات خاصی را پیاده‌سازی می‌کنند. این cohesion با فدا کردن coupling که در بخش قبل توضیح داده شد به دست آمده است.

مقایسه

هر سه الگو از cohesion مناسبی برخوردار هستند. Decorator و visitor هر کدام کلاس‌های متخصص رفتار را تعریف می‌کنند که خود منجر به بهبود cohesion نسبت به حالت پیشین می‌شود. در flyweight هم cohesion بالا است حتی پیش از اعمال الگو.

Controller

Decorator

در این الگو، controller دیده نمی‌شود چرا که وظیفه‌ی کارچرخانی در الگو دیده نمی‌شود و هر شی دکوراتور، پیغامی را به شی بعدی پاس می‌دهد و پیش یا پس از دریافت پاسخ، عملیات مربوط به خود را انجام می‌دهد. در نتیجه از controller بهره برده نمی‌شود.

Flyweight

در این الگو، factory هر چند واقعا وظیفه‌ی کارچرخانی ندارد، اما مدیریت دریاچه‌ی SharedConcreteFlyweight، برعهده‌ی factory است و factory اشیا را می‌سازد و یا اشیا موجود را باز می‌گرداند. در واقع می‌توان گفت factory وظیفه‌ی آفرینش را علاوه بر کنترل و مدیریت رابطه‌ی کلاینت با flyweight pool را برعهده دارد. با این وجود نمی‌توان گفت دقیقا وظیفه‌ی controller تعریف شده در grasp دارد چرا که شروع کننده‌ی یک زنجیره‌ی تعاملی نیست بلکه تنها مدیریت pool را برعهده دارد.

Visitor

در این الگو نیز controller وجود ندارد چرا که زنجیره‌ی تعاملی دیده نمی‌شود و در این الگو استفاده نشده است.

مقایسه

در هر سه الگو controller دیده نمی‌شود. هر چند flyweight یک شی factory دارد که وظیفه‌ی مدیریت روابط کلاینت با flyweight pool را دارد اما وظیفه‌ی controller را کاملا برعهده ندارد و در نتیجه هیچ کدام از controller استفاده نمی‌کنند.

Polymorphism

Decorator

تمامی اشیا decorator به خوبی از این ویژگی شی گرایی استفاده می‌کنند. به واسطه‌ی همین ویژگی می‌توان decoratorهای متنوع تعریف کرد که با رفتارهای مختلفی را به ازای یک فراخوانی مشترک انجام می‌دهند. حتی تعریف componentهای مختلف نیز امکان پذیر است و این الگو از polymorphism به خوبی بهره می‌برد.

Flyweight

در این الگو نیز به کمک polymorphism، می‌توان اشیا SharedConcreteFlyweight و UnsharedConcreteFlyweight متعدد تعریف کرد و در نتیجه در این الگو نیز از polymorphism بهره برده شده است. در واقع یک operation مشترک داریم که به شکل متفاوتی پیاده‌سازی می‌شود.

Visitor

در این الگو نیز به کمک چندریختی می‌توانیم ConcreteVisitorهای مختلف یا ConcreteElementهای مختلف تعریف کنیم. اساسا این الگو برای تحقق تعریف عملیات‌های متعدد تعریف شده است و visitorها هر یک عمیات متفاوتی را روی یک element خاص انجام می‌دهند که همگی تحت یک operation واحد در واسط visitor تعریف شده اما پیاده‌سازی مختلف دارد. در نتیجه در این الگو نیز از چندریختی استفاده شده است.

مقایسه

در هر سه الگو از polymorphism استفاده شده است.

Indirection

Decorator

در این الگو وظیفه‌ی واسطه‌گری دیده نمی‌شود اما indirection به مقدار جدی ایجاد شده است. با تعریف زنجیره‌ی اشیا decorator، هر کدام واسطه‌ی انتقال پیام به بعدی هستند و در نتیجه چندین سطح indirection می‌تواند برای انتقال پیام به component ایجاد شود. در نتیجه indirection در این الگو دیده می‌شود. هدف از ایجاد این دسترسی غیر مستقیم نیز بالا بردن انعطاف‌پذیری مجموعه برای تعریف decoratorهای متنوع بوده است.

Flyweight

در این الگو، factory به نوعی واسطه‌ی دسترسی کلاینت به شی flyweight است. با ایجاد یک سطح indirection، می‌توانیم flyweight pool را کنترل کنیم که این وظیفه برعهده‌ی factory است و در نتیجه از این الگو به خوبی بهره برده شده است.

Visitor

در این الگو واسطه گری و دیده نمی‌شود. در واقع کلاینت مستقیما با element می‌خواهد ارتباط برقرار کند و المان اما وظیفه‌ی خود را به visitor محول کرده است. از انجایی که هدف کلاینت ارتباط با visitor نبوده است در نتیجه indirection انجام نشده است بلکه وظیفه محول شده است و در این الگو از indirection استفاده نمی‌شود.

مقایسه

در دو الگوی decorator و flyweight دیده می‌شود اما در visitor با وجود انتقال پیام به visitor، اما indirection با هدف واسطه‌گری دیده نمی‌شود چرا که انتقال پیام غیر مستقیم نداریم.

Pure Fabrication

Decorator

در دامنه‌ی تحلیل، decoratorها دیده نمی‌شود و صرفا رفتارهای component مشاهده می‌شود. در زمان طراحی و برای بالا بردن انعطاف مجموعه و تحقق هدف الگو، decoratorها تعریف شده و وظایفی که پیش‌تر به شکل feature leiden class انجام می‌شد، به کمک delegation و در زمان اجرا توسط این اشیا متخصص انجام می‌شود تا رفتارهایی را به مجموعه بیافزایند. در نتیجه در مجموع decoratorها اشیا جعلی هستند و این الگو از شی جعلی استفاده می‌کند.

Flyweight

اشیا ConcreteFlyweights و FlyweightFactory در دامنه‌ی مسئله و در مرحله‌ی تحلیل نیستند و در طراحی و برای بهبود استفاده از منابع سیستمی تعریف می‌شوند. در نتیجه این الگو حداقل دو شی جعلی جدید دارد.

Visitor

این الگو نیز visitor را تعریف می‌کند. در دامنه‌ی تحلیل visitor وجود ندارد و فقط عملیات‌های مختلف روی elementها دیده می‌شود و تعریف آن مختص مرحله‌ی طراحی است، operationهایی که در ذات المان‌ها نیستند را از آن‌ها جدا می‌کنیم و به کلاس‌های متخصص visitor منتسب می‌کنیم. در نتیجه این الگو نیز اشیا جعلی متخصص تعریف می‌کند.

مقایسه

هر سه الگو از اشیا جعلی جدیدی برای تحقق خود استفاده می‌کنند. در visitor و decorator اشیا جعلی نماینده‌ی عملیاتی هستند که می‌خواهد انجام شود و در flyweight، دو شی جعلی داریم که یکی آفرینشی و دیگر نماینده‌ی یک شی مورد استفاده است.

Protected Variations

Decorator

در این الگو با تعریف واسط decorator و همچنین component، کلاینت نسبت به تغییرات و تعریف component و decoratorهای جدید مصون شده است. Component نیز از تغییرات decorator تاثیری نمی‌پذیرد که همگی به تعریف واسط مناسب و جداسازی بخش‌های مختلف است و در نتیجه از protected variations استفاده می‌شود.

Flyweight

واسط flyweight بر روی Shared و Unsharedها کشیده شده است و در نتیجه، کلاینت از تغییرات و تعریف flyweightهای جدید منتزع می‌شود. وظیفه‌ی آفرینش برعهده‌ی factory است اما تعریف واسط flyweight دید کلاینت و تغییرات flyweight را از یکدیگر جدا می‌کند و protected variation دیده می‌شود.

Visitor

در حالت پیشین الگو، عملیات‌های متعدد جدیدی نیاز و قابل تغییر بود در نتیجه بخش عملیات‌های ممکن روی المان‌ها به شدت unstable و قابل تغییر بود. به همین منظور، این رفتارها را از اشیا جدا کردیم و ذیل واسط visitor تعریف می‌کنیم در نتیجه به کمک protected variation، المان‌ها را نسبت به تغییرات عملیات‌ها مصون کردیم. هر چند در حالت پس از اعمال الگو، زیرکلاس‌های visitor به elementها وابستگی شدید دارند و تغییرات آن‌ها روی visitor تاثیر می‌گذارد. اما فرض اعمال الگو تغییر نکردن elementها هست و در نتیجه برای جداسازی elementها از تغییرات visitor، این interface تعریف شده است.

مقایسه

هر سه الگو از protected variations بهره برده اند. هر چند این الگو در decorator و visitor نمود جدی‌تری دارد چرا که تغییرات flyweight چندان محتمل نیست اما در هر صورت این الگو در هر سه دیده می‌شود.