به نام او

درس الگوها در مهندسی نرم‌افزار

استاد: دکتر رامان رامسین

تمرین اول

پدرام شاطری - ۴۰۰۲۱۱۳۹۸

نیمسال دوم ۰۰-۰۱

فهرست مطالب

مقایسه الگوهای **Command, Proxy** و **Mediator**

دسته - نوع

Command – این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

Proxy – این الگو در درسته الگوهای ساختاری می‌باشد. الگوهای ساختاری با در کنار هم قراردادن کلاس‌ها و آبجکت‌ها باعث تولید ساختارهای بزرگ‌تر می‌شود درحالی‌که این ساختارها را انعطاف‌پذیر و کارا نگه می‌دارد. در واقع با شناسایی ارتباطات باعث ساده شدن ساختار می‌شود.

Mediator - این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

مقایسه: دو الگوی Command و Mediator در دسته الگوهای رفتاری می‌باشد و هدف اصلی این الگوها تخصیص مسئولیت به آبجکت‌ها است درحالی‌که الگوی Proxy در دسته الگوهای ساختاری می‌باشد و این الگوها بیشتر دغدغهٔ تولید ساختارهای بزرگ از طریق کنار هم قراردادن کلاس‌ها و آبجکت‌ها را دارند.

هدف

Command – الگوی کامند برای تبدیل ریکوئســـت‌ها به آبجکت‌های مســـتقل و مانا که تمام اطلاعات ریکوئســـت را در بر دارد به کار می‌رود. با اســـتفاده از این الگو می‌توان ریکوئســـت‌ها را به‌عنوان یک پارامتر (آبجکت) به متدها ارسـال کرد, صف‌بندی یا تأخیر در اجرای ریکوئسـت ایجاد کرد یا از عملیات هایی که نیاز به بازگردانی دارند پشـــتیبانی کرد. هم چنین امکان logging یا ذخیره‌ســـازی را به‌راحتی فراهم می‌آورد. بعد از اعمال الگو به‌جای ارســـال پیام از آبجکت A به B, در واقع به آبجکت Command می‌گوید Execute شو. (ارتباط مستقیم بین کلاینت و سرور از بین می‌رود.)

Proxy - این الگو می‌تواند یک آبجکت را جایگزین یا به‌اصـطلاح Placeholder یا نایب برای یک آبجکت دیگر قرار دهد. این امکان را فراهم می‌کند تا عملیات هایی قبل و بعد از ارســـال ریکوئســـت به آبجکت اصـــلی انجام داد. در واقع یک سـطحی از indirection به آبجکت اصلی اضافه می‌کند و باعث می‌شـود باری از رو دوش آبجکت اصـــلی برداشـــته شـــود و بعد از واردشـــدن آبجکت اصـــلی در حافظه, وظیفهٔ یک واســـطه یا Delegator را دارد.

Mediator – این الگو به ما کمک می‌کند تا ارتباطات بی‌نظم و پر هرج‌ومرج و پیچیدهٔ بین آبجکت‌ها را از بین ببریم و ارتباطات بین آن‌ها را طریق آبجکت واسط (Mediator) محدود و مدیریت کنیم.

مقایسـه: می‌توان هر ۳ الگو کار متفاوتی انجام می‌دهند به این صورت که الگوی Command برای تبدیل ریکوئســـت‌ها به آبجکت‌های مانا و مســـتقل, الگوی Proxy برای داشـــتن یک جایگزین به آبجکت اصـــلی و برداشـتن باری از روی دوش آبجکت اصـلی و الگوی Mediator برای سـاماندهی ارتباطات پیچیده و نامنظم بین آبجکت‌ها به کار می‌روند.

حوزه

Command – این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Proxy - این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Mediator - این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

مقایسه: هر سه الگو در زمان اجرا و از طریق delegation تحقق می‌یابند.

کاهش وابستگی

Command – با اعمال این الگو Coupling که مستقیم بین Receiver و Invoker وجود داشت از بین می‌رود و باعث می‌شود این ۲ از هم خبر نداشته باشند. فقط ارتباط بین پیکربند که باید انواع کلاس‌های Command را بشناسد تا بتواند آن‌ها را نمونه‌سازی کند و در اختیار Invoker قرار دهد. در واقع پیکربند باید دید به Receiver و Invoker و هم چنین Concrete Commandها برای پیکربندی داشته باشد این دید را می‌توان به‌صورت Dependency یعنی غیر مانا در نظر گرفت زیرا لازم نیست همیشه به آن‌ها دید داشته باشد و فقط برای پیکربندی از آن‌ها استفاده می‌کند. هم چنین ارتباط بین Command و Invoker نیز غیرمستقیم و فقط از طریق اینترفیس آن هست.

Proxy - با استفاده از این الگو می‌توان وابستگی بین کلاینت و آبجکت اصلی را با استفاده از اینترفیس و محقق‌سازی آن توسط آبجکت اصلی و جایگزین و استفاده از آبجکت جایگزین برای عملیات, از بین برد. یعنی یک اینترفیس تعریف کرد و آن را توسط کلاس اصلی و جایگزین که همان پراکسی هست محقق کرد (در این صورت کمی Coupling کمتر می‌شود و با تغییرات در کلاس اصلی تغییرات به‌جایی منتشر نمی‌شود). اما خود پیکربند به ساختن انواع پراکسی‌های نیاز دارد یعنی باید کلاس‌های عینی اصلی را بشناسد و آن را در متد سازندهٔ هر پراکسی به او پاس بدهد. پس از آن کلاینت با اینترفیس آن کار می‌کند.

Mediator – با اســتفاده از این الگو وابســتگی همکاران که قبلاً به‌صــورت مســتقیم به یکدیگر بود از بین می‌رود و فقط به آبجکت Mediator خود وابسته می‌شوند. اما این وابستگی در سطح بالا و فقط در سطح اینترفیس اسـت, اما میانجی باید کلاس‌هایی که در داخل خود هسـت (ممکن اسـت با یک همکار کار نکند, در واقع به آن, دیگر همکار یا Colleague نمی‌گویند.) را در ســطح Concrete بشــناســد, در کل می‌توان گفت باعث کاهش وابستگی می‌شود چون حالت قبلی این الگو خیلی وابستگی بالاست.

مقایسه: می‌توان گفت هر ۳ الگو در کاهش وابستگی نقش مثبت داشته اگرچه هنوز هم در الگوی کامند پیکربند یا همان Client به Invoker و Receiver دید دارد اما این اجتناب‌ناپذیر اســت. هم چنین در مورد الگوی Mediator باید گفت که Mediator به تمام همکاران Coupled شـده اسـت چون در صـورت ارسـال یک Event از یک همکار باید آبجکت میانجی, همکاران را بشـناسـد تا با اسـتفاده از چک کردن این Event جواب مناسـب را به آن‌ها ارسـال کند. هم چنین در Proxy پیکربند نیز باید برای configuration بتواند انواع آبجکت اصـلی را بشــناســد و آن را به نایب که همان پراکســی هســت پاس دهد (اما در virtual proxy داسـتان کمی متفاوت هسـت و در زمان پیکربندی آبجکت اصـلی سـاخته نمی‌شـود و فقط در صـورت نیاز هست که این اتفاق می‌افتد.).

افزایش چسبندگی

Command – این الگو در واقع آبجکت‌های متخصــص برای ارســال و دریافت Commandها دارد و این کلاس‌ها و آبجکت‌ها همبســتگی یا چســبندگی بالایی دارند. این الگو بیشــتر در کاهش وابســتگی عمل می‌کند ولی کمی کلاس‌های Invoker و Receiver ســاده و یکپارچه می‌شــوند. ازآنجایی‌که آبجکت‌های Invoker درگیر خود عملیات نیسـتند و این کار به Command که خود متخصـص انجام کار هسـت سـپرده شده, پس تک کارگی در این الگو افزایش‌یافته است.

Proxy - این الگو فقط به‌عنوان جایگزین برای آبجکت اصلی می‌باشد و تأثیر زیادی در افزایش چسبندگی ندارد و در واقع همان کارهایی که آبجکت اصلی انجام می‌دهد را واسپاری می‌کند و هر دو یک اینترفیس را محقق می‌سازند. می‌توان گفت چون کلاس پراکسی فقط وظیفه واسپاری دارد از وابستگی خوبی برخوردار است.

Mediator – خود کلاس Mediator که کار یکتایی دارد و آن هم ارتباط بین همکاران هست, خود همین کلاس باعث می‌شود تا دغدغهٔ کلاس‌ها و همکاران دیگر برای برقراری ارتباط با یکدیگر کمتر شده و فقط بر روی کار خود تمرکز کنند.

مقایسه: الگوی Mediator که به‌صورت واضح و کاملاً مشخص در افزایش چسبندگی تأثیر دارد اما الگوی Proxy تأثیر آن‌چنانی در افزایش چسبندگی بعد از اعمال الگو ندارد. هم چنین الگوی Command نیز در این شاخص مثبت عمل کرده است.

OCP

Command – در این الگو اصل OCP برای Invoker و Receiver و خود کامند به حد خوبی رعایت شده, زیرا کلاینت با Invoker کار می‌کند و خود این کلاس با اینترفیس کلاس Command کار می‌کند و به‌این‌ترتیب از تغییرات در امان است. برای Receiver می‌توان گفت چون معمولاً این کلاس‌ها برای بیزینس لاجیک ها نوشته شده و از اینترفیس Command یا Invoker خبر ندارد. اما کلاینت باید از تغییرات Invoker خبر داشته باشد تا بتواند با آن به‌درستی کار کند (تا حدی غیرقابل‌اجتناب هست.). هم چنین کلاینت باید انواع کلاس‌های Concrete Command را برای پیکربندی بشناسد.

Proxy - در این الگو نیز چون می‌توان به‌راحتی بدون تغییر قسمت‌های اصلی این الگو, Proxyهای جدید تعریف کرد که همه با همان اینترفیس کار می‌کنند می‌توان گفت این اصل را رعایت می‌کند.

Mediator – این الگو چون همکاران فقط از طریق اینترفیس با آبجکت Mediator در ارتباط هستند, تغییرات از Mediator به همکاران منتشر نمی‌شود ولی تغییرات همکاران باعث تغییرات در متد Notify یا به عبارتی خود Mediator خواهد شد زیرا Mediator تمام کلاس‌های همکار را در سطح پایین و Concrete می‌بیند و به نظر نمی‌توان در هر دو سو اصل OCP را رعایت کرد.

مقایسه: تقریباً می‌توان گفت که هر ۳ الگو تا حدی این اصل را رعایت کرده‌اند و بعضی از رعایت نشدن‌ها اجتناب‌ناپذیر خواهد بود مثلاً در الگو Mediator از طرف همکاران به Mediator این اصل رعایت نشده و تغییرات در همکاران باعث تغییر در Mediator می‌شود. یا در الگوی Command از سمت پیکربند به Receiver, Invoker و Concrete Command ها OCP برقرار نشده است.

LSP

Command – کاملاً برقرار است چون زیر کلاس‌های کامند می‌تواند به‌عنوان کلاس پدر استفاده شود چون یک اینترفیس را محقق می‌سازد.

Proxy - کاملاً برقرار است چون زیر کلاس‌های پراکسی می‌تواند به‌عنوان کلاس پدر استفاده شود چون یک اینترفیس را محقق می‌سازد.

Mediator – کاملاً برقرار است چون زیر کلاس‌های همکار می‌تواند به‌عنوان کلاس پدر خود یعنی Component استفاده شود چون یک اینترفیس را محقق می‌سازد. هم چنین از سمت کلاس Mediator نیز به همین دلیل کاملاً این اصل برقرار است.

مقایسه: هر ۳ الگو این اصل را برقرار می‌سازند.

DIP

Command – همان‌طور که قبل‌تر اشاره شد ارتباط بین Invoker و Command از طریق اینترفیس آن‌ها است و این اصل برقرار است. اما ارتباط بین کلاینت که نقش پیکربند را دارد با Receiver یا Invoker یا کامند از طریق کلاس‌های عینی و Concrete هست و باید آن‌ها را ببیند پس DIP نقض شده است.

Proxy - در این الگو چون آبجکت Proxy ارتباطی که با آبجکت اصلی دارد به‌صورت محقق‌سازی یک اینترفیس یکسان هست و کلاینت متوجه نمی‌شود که با کدام آبجکت دارد کار می‌کند, پس این اصل برقرار است.

Mediator – همان‌طور که پیش‌تر گفته شد اگر از سمت خود آبجکت‌های همکار به Mediator باشد با استفاده از Interface هست این اصل رعایت شده است ولی اگر از سمت Mediator به همکاران را نگاه کنیم متوجه نقض DIP می‌شویم.

مقایسه: هر ۳ الگو در حد خوبی این اصل را رعایت کرده‌اند اما در الگوی Mediator از طرف خود آبجکت میانجی به سمت همکاران ارتباط از طریق Concretion و هم چنین در الگوی Command ارتباط کلاینت با Invoker یا Receiver نیز از طریق Interface نیست و هم چنین باید زیر کلاس‌های عینی Command را ببیند پس DIP را نقض می‌کنند.

ISP

Command – هر آبجکت Command فقط اینترفیس خود که متد مربوط به Execute یا log یا ... را دارا می‌باشد و محقق می‌سازد پس این اصل رعایت می‌شود.

Proxy - چون هر دو آبجکت اصلی و فرعی هر ۲ یک اینترفیس را محقق می‌سازند و کار اضافه‌ای انجام نمی‌دهد این اصل رعایت می‌شود.

Mediator – در این الگو چون به‌صورت معمولی یک متد برای کارچرخانی و ارتباط بین همکاران وجود دارد ISP به‌خوبی رعایت می‌شود. حال اگر به‌جای چندین Mediator, فقط یکی داشته باشیم ممکن است با بزرگ‌شدن و جنرال شدن اینترفیس آن این اصل نیز نقض شود.

مقایسه: هر ۳ الگو به‌خوبی این اصل را رعایت می‌کنند مگر الگوی Mediator آن هم در صورت Generic شدن یک کلاس Mediator به‌جای چندین Mediator کوچک.

CRP

Command – این اصل در این الگو رعایت می‌شود چون در کلاس Invoker ارتباط با هر Command از طریق Delegation انجام می‌شود و همچنین ارتباط بین Command و Receiver نیز به همین صورت است و این الگو در زمان اجرا محقق می‌شود.

Proxy - به‌وضوح مشخص هست که ساختار توارثی خاصی در این الگو دیده نمی‌شود و آبجکت اصلی و جایگزین فقط در محقق کردن یک اینترفیس شرکت دارند. می‌توان گفت این اصل رعایت شده است. هم چنین آبجکت اصلی به‌صورت aggregate در داخل پراکسی هست.

Mediator – در این الگو هم که در زمان اجرا محقق می‌شـــود می‌توان گفت که ارتباط بین Mediator و Colleagues فقط از طریق Delegation هســت و ســاختار توارثی بین آن‌ها دیده نمی‌شـود پس این اصـل رعایت شـده اسـت.

مقایسه: هر ۳ الگو به‌خوبی این اصل را رعایت می‌کنند.

PLK

Command – این اصــل در این الگو رعایت شــده چرا که ارتباط Command و Invoker از طریق ارســال آبجکت انجام نمی‌شــود و فقط از طریق صــدا زدن و دانســتن اینترفیس‌های یکدیگر می‌باشــد, هم چنین ارتباط Command و Receiver نیز به همین صورت هست.

Proxy - این اصــل در این الگو به‌خوبی رعایت شــده زیرا آبجکت Proxy آبجکت اصــلی را در اختیار کلاینت قرار نمی‌دهد و فقط به‌واســطه ی اینترفیس به آن درخواســت می‌زند و جواب را به کلاینت می‌دهد. اگرچه طراح می‌تواند آبجکت اصلی را در اختیار کلاینت قرار دهد (نقض غرض)

Mediator – در این الگو می‌توان گفت این اصل رعایت شده چرا که ارتباطات بین آبجکت‌ها در داخل کلاس Mediator فقط از طریق ارســـال پیام به و از آن‌هاســـت و هم چنین ارتباط همکاران و کلاس میانجی به همین صــورت اســت. اگرچه طراح می‌تواند همکاران را در کلاس میانجی به همکاران دیگر پاس دهد. (این کار کاملاً اشتباه است اما می‌تواند باعث نقض PLK شود.)

مقایســـه: هر ۳ الگو این اصـــل را رعایت می‌کنند اگرچه با اشـــتباه طراح می‌تواند این اصـــل در الگوهای Mediator و Proxy نقض شود.

بسته‌بندی

Command – در این الگو چون Invoker و Receiver از وجود یکدیگر خبر ندارند و هم چنین Invoker از ماهیت Command خبر ندارد و فقط از طریق اینترفیس آن را اجرا می‌کند و هم چنین خود Command به Receiver فقط از طریق ارسال پیام کار می‌کند و به حالت داخلی آن کاری ندارد به‌خوبی Encapsulation انجام شده است.

Proxy - در این الگو چون ارتباط بین آبجکت اصلی و فرعی فقط از طریق اینترفیس سطح بالای مشترک آن‌ها و از طریق ارسال پیام انجام می‌شود می‌توان گفت که بسته‌بندی رعایت شده است.

Mediator – در این الگو ارتباط بین همکاران و کلاس میانجی فقط از طریق اینترفیس سطح بالای Mediator هست و بسته‌بندی برقرار است اما ارتباط میانجی با کلاس‌های کارمند به‌صورت مستقیم و با کلاس‌های Concrete هست این ارتباط به تغییر نام یا خروجی متدهای کلاس‌های همکار از بین می‌رود و فقط در این صورت هست که بسته‌بندی نقض شده است.

مقایسه: هر ۳ الگو به‌خوبی بسته‌بندی را انجام می‌دهند.

انعطاف‌پذیری

Command – در این الگو چون ارتباط بین Invoker و Receiver از بین رفته و در واقع Invoker دارد با اینترفیس Command کار می‌کند و می‌توان به‌راحتی بدون اینکه Invoker متوجه شود اینستنس Command را عوض کرد که این ۲ مورد به انعطاف‌پذیری کمک شایانی می‌کند. هم چنین Client که آبجکت‌های Receiver را به Command ارسال می‌کند می‌تواند با ارسال آبجکت‌های مختلف از Receiverها آن‌ها را به‌صورت های مختلف پیکربندی کند (اگر در اینجا دریافت‌کننده‌ها هم از یک اینترفیس

سطح بالا پیروی کنند انعطاف‌پذیری به بالاترین حد ممکن می‌رسد.). اما فقط برای پیکربندی در واقع Client باید زیر کلاس‌های عینی Command و انواع Receiver را بشناسد.

Proxy - این الگو چون از یک اینترفیس سطح بالا برای ساخت آبجکت جایگزین برای آبجکت اصلی استفاده می‌کند می‌توان از Proxyهای مختلف برای یک آبجکت اصلی استفاده کرد. می‌توان گفت تا حد خوبی انعطاف‌پذیر است.

Mediator – ازآنجایی‌که این الگو رفتارهای پیچیده بین آبجکت‌های همکار را ساده کرده و به‌راحتی می‌توان همکار کموزیاد کرد و تغییرات کمی در کلاینت داد (فقط نوع دیگری را از آبجکت میانجی نمونه‌سازی می‌کند) یعنی به‌راحتی حتی در زمان اجرا بازپیکربندی را انجام داد و Mediator دیگری استفاده کرد. به‌راحتی برای تغییر رفتار بین کلاس‌ها می‌توان یک Subclass از Mediator تعریف کرد بدون آن که این تغییرات به کلاینت یا به همکاران انتشار یابد.

مقایسه: هر ۳ الگو به‌خوبی بسته‌بندی را انجام می‌دهند. الگوی Proxy نیز تا حد خوبی انعطاف‌پذیر می‌باشد. در الگوی Command اگر Receiver ها یک اینترفیس سطح بالا داشته باشند تمام Command ها بدون درنظرداشتن نوع دریافت‌کننده و یا تغییرات درونی آن‌ها با آن‌ها کار می‌کند بدون اینکه نیاز به تغییر داشته باشد.

تغییرپذیری/ انتشار تغییرات

Command – در این الگو تغییرات بین Invoker و Receiver به یکدیگر منتشر نمی‌شود جون ارتباط آن‌ها کاملاً قطع شده است, هم چنین Invoker از تغییرات Command در امان است زیرا فقط با استفاده از اینترفیس با آن کار می‌کند و فقط کلاینت هست که باید از انواع Command ها یا Receiver ها برای پیکربندی خبر داشته باشد. این یعنی تغییرپذیری بالا و منتشر نشدن تغییرات زیرا این وابستگی کلاینت و

بقیه کلاس‌های مذکور مانا نیســـت (ارتباط بین پیکربند و زیر کلاس‌های عینی Command به‌صـــورت Dependency هست و ارتباط بین پیکربند و Receiver و Invoker نیز می‌تواند به همین صورت باشد.).

Proxy - در این الگو نیز به دلیل استفاده از یک اینترفیس سطح بالا آبجکت Proxy از تغییرات درون آبجکت اصلی در امان است و تغییرات منتشر نمی‌شود. فقط کلاینت باید از اینترفیس پراکسی باخبر باشد.

Mediator – در این الگو تغییرات در کلاس میانجی به کلاس‌های همکار منتشر نمی‌شود چون کلاس‌های همکار فقط از اینترفیس کلاس میانجی استفاده می‌کند و همچنین این تغییرات به همکاران دیگر نیز منتقل نمی‌شود, اما در صورت تغییر در متدهای کلاس‌ها همکار (کم‌وزیاد شدن آن‌ها) این تغییر به کلاس میانجی منتقل می‌شود.

مقایســه: الگوی Proxy تغییرپذیری بالا و انتشــار تغییرات پایین دارد, در الگوی Command نیز این ویژگی به‌خوبی وجود دارد ولی پیکربند باید از این تغییرات خبردار شـــود و هم چنین تغییرات در عملیات های Receiver به کلاس‌های Command که از آن Receiver اســـتفاده می‌کند منتشـــر می‌شـــود. در الگوی Mediator نیز فقط تغییرات در همکارها باعث انتشــار تغییرات می‌شــود در غیر این صـورت می‌توان گفت تا حد خوبی این ۳ الگو تغییرپذیر هستند.

پیکربندی

Client – Command مسـئول پیکربندی این الگو می‌باشـد با سـاختن آبجکت از Command و پاس دادن آبجکت Receiver در صـــورت نیاز به آن و هم چنین اســـتفاده از Invoker برای اجرای این Command پیکربندی را در زمان اجرا انجام می‌دهد. ارتباط بین پیکربند و زیر کلاس‌های عینی Command به‌صـــورت Dependency هســـت و ارتباط بین پیکربند و Receiver و Invoker نیز می‌تواند به همین صـــورت باشـــد

(یعنی یک ارتباط مانا و به‌صـورت Association نباشــد). در واقع یک ثالث نیاز داریم که اینجا همان کلاینت هست.

Proxy - در این الگو نیز ثالث آبجکت اصــلی را به آبجکت فرعی می‌دهد و پیکربندی آن را انجام می‌دهد و از طریق اینترفیس با آبجکت فرعی کار می‌کند و آن را در اختیار کلاینت قرار می‌دهد. کلاینت فقط اینترفیس سطح بالا را می‌شناسد و با آن کار می‌کند.

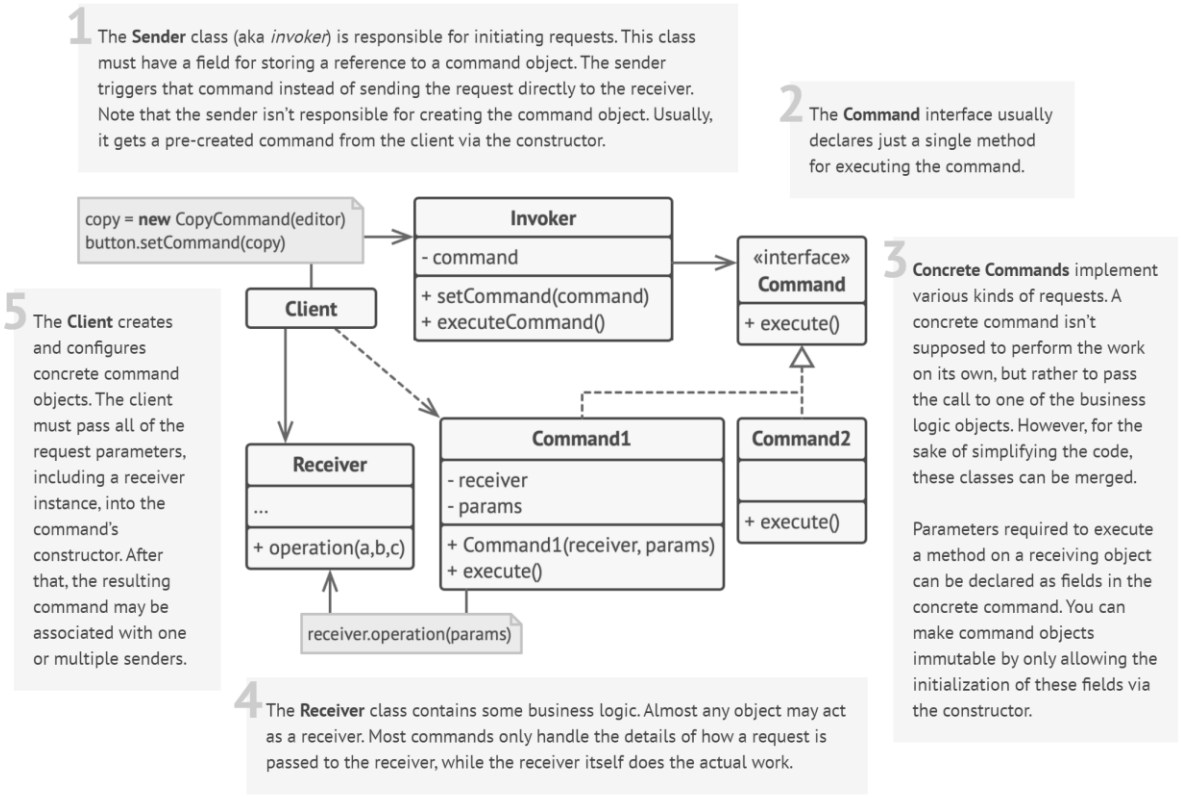
Mediator – یک ثالث که انواع همکاران و انواع میانجی‌ها را می‌شناسد آن‌ها را پیکربندی کرده و در اختیار کلاینت قرار می‌دهد.

مقایسه: هر ۳ نیاز به پیکربند نیاز دارند.

ساختار و رفتار

در این قسمت توضیحات تکمیلی در شکل موجود است.

Command

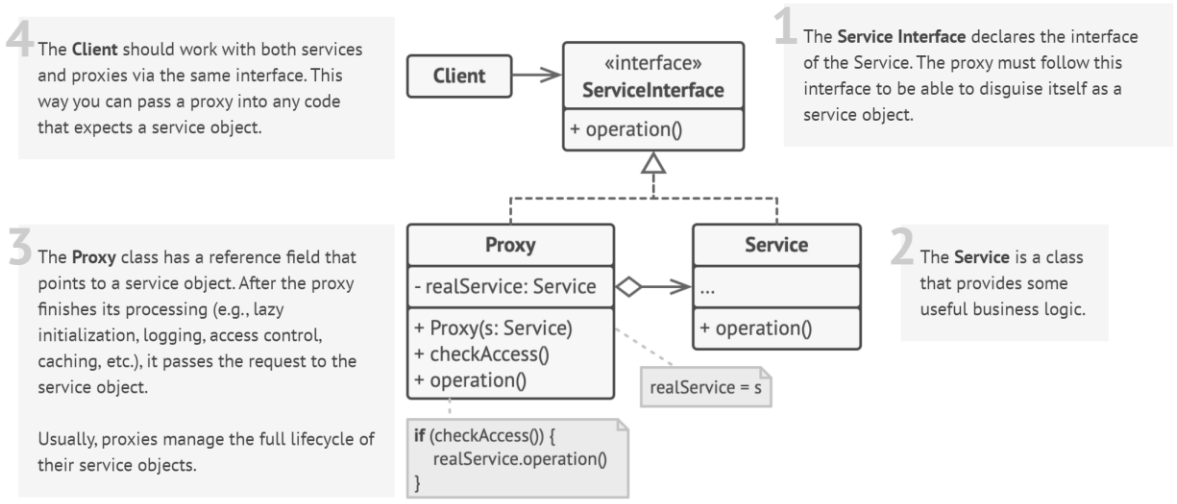


شکل ۱ ساختار الگوی Command ۱[۱]

همان‌طور که در شکل ۱ می‌بینید کلاینت که در واقع پیکربند این مجموعه است کلاس Receiver و کلاس‌های فرزند Command و هم چنین کلاس Invoker را می‌بیند. ارتباط بین پیکربند و زیر کلاس‌های عینی Command به‌صورت Dependency هست و ارتباط بین پیکربند و Receiver و Invoker نیز می‌تواند به همین صورت باشد (یعنی یک ارتباط مانا و به‌صورت Association نباشد). به این صورت که ابتدا یک Concrete Commandمی‌سازد و سپس Receiver لازم را می‌سازد و به کامند پاس می‌دهد. سپس این Command به Invoker پاس داده می‌شود و بعد از آن وقتی Invoker متد Execute را صدا بزند کامند اجرا می‌شود. هم چنین خود پیکربند می‌تواند مسئول صدا زدن Invoker و اجرای کامند آن باشد (invoker.executeCommand()).

ساختاری که در اینجا آمده تفاوت خاصی با الگویی که در کتاب هست ندارد. به همین دلیل از فقط یکی از آنها ذکر شده است.

Proxy

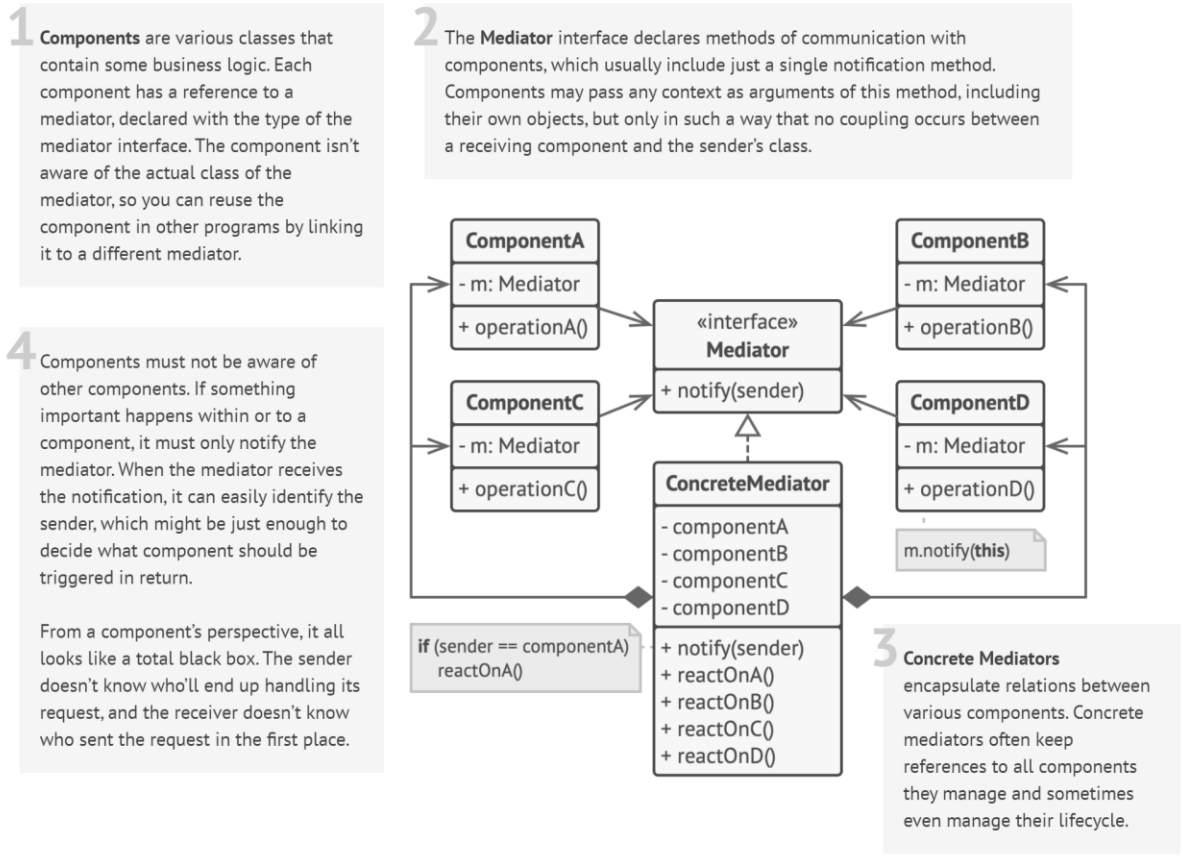


شکل ۲ ساختار الگوی Proxy [۲]

همان‌طور که در شکل ۲ مشخص است کلاینت فقط با اینترفیس Subject کار می‌کند یعنی بسته به نوع پراکسی این آبجکت‌ها ساخته شده و کلاینت با آن‌ها کار می‌کند مثلاً Virtual Proxy آبجکت ته زنجیره که آبجکت اصلی هست را خود Proxyمی‌سازد.

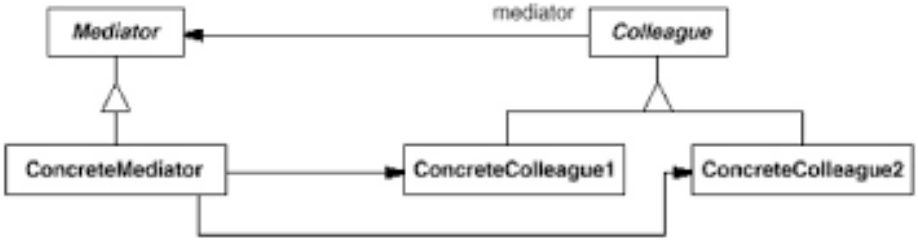
ساختاری که در اینجا آمده تفاوت خاصی با الگویی که در کتاب هست ندارد. به همین دلیل از فقط یکی از آنها ذکر شده است.

Mediator



شکل ۳٫۱ ساختار الگوی Mediator [۳]

همان‌طور که از شـــکل۳٫۱ پیداســـت دید کلاس‌های Concrete همکار به آبجکت میانجی از اینترفیس پدر آن‌ها و برعکس از کلاس Concrete میانجی به آبجکت‌های Concrete همکار هســـت یعنی می‌تواند با همهٔ آن‌ها تعامل نداشـــته باشـــد. این الگو به این صـــورت رفتار می‌کند که تعاملات بین آبجکت‌های همکار فقط از طریق Notify کردن کلاس میـانجی انجـام می‌پذیرد و خود کلاس میـانجی بـا اســـتفـاده از دیـد مســـتقیمی که به کلاس‌های همکار دارد, تعاملات بین آن‌ها را انجام می‌دهد اما باید از تبدیل‌شـــدن آن به یک God Class جلوگیری کنیم.



شکل ۳٫۲ ساختار الگوی Mediator در کتاب

همان‌طور که می‌بینید تنها تفاوتی که بین این دو ساختار هست, این هست که برای کلاس‌های همکار یک فوق کلاس در نظر گرفته شده تا از یک اینترفیس پیروی کنند ولی در هر دو حالت کلاس‌های Mediator که به‌صورت Concrete هستند با زیر کلاس‌های عینی همکار باید به‌صورت مستقیم کار کنند.

مقایسه: در الگوی Command در واقع با استفاده از یک Invoker و پیکربندی کردن Command های مختلف روی Receiver های متعدد می‌توان دستورات زیادی را اجرا کرد و هم چنین کلاینت باید از کلاس‌های Concrete دستور و دریافت کنند باخبر باشد, در الگوی Proxy در واقع با تولید یک آبجکت جایگزین می‌توان باری از روی دوش آبجکت اصلی برداشت (همان‌طور که بالاتر گفته شد بسته به نوع پراکسی ساخته‌شدن آن‌ها توسط کلاینت متفاوت است) و در آخر در الگوی Mediator تعاملات همکاران از میانجی انجام می‌شود و پیکربند از کلاس‌های Concrete میانجیان و همکاران باخبر است (اگرچه میانجی لازم نیست از تمام همکاران باخبر باشد و با آن‌ها ارتباط داشته باشد.).

کارایی از منظر حافظه

Command – ازآنجایی‌که قبلاً پیام‌ها بین Invoker و Receiver انجام می‌شد و به‌صورت موقت بود و ما آن را با این الگو به یک هویت مانا تبدیل کردیم مصرف حافظه بالا می‌رود مخصوصاً اگر بخواهیم قابلیت برگشت‌پذیری اضافه کنیم و اطلاعات سیستم را در قبل از اجرای دستور در حافظه نگه داریم.

Proxy - در این الگو باتوجه‌به این که ممکن است برای بعضی از عملیات ها لازم نباشد آبجکت اصلی درون حافظه بیاید و با استفاده از Proxy به آن پاسخ دهیم مصرف حافظه کم شده است.

Mediator – در این الگو فقط یک یا چند آبجکت میانجی اضافه شده و می‌توان گفت کمی مصرف حافظه افزایش میابد (تنها در صورت زیاد بودن میانجی‌ها)

مقایسه: در الگوی Command به‌وضوح زیاد شده, در الگوی Proxy باتوجه‌به نوع پراکسی متفاوت است اما به دلیل مذکور در بالا می‌تواند کاهش دهد و هم چنین در Mediator می‌توان گفت تغییر چندانی نکرده است.

موارد کاربرد

Command – وقتی می‌خواهیم مثلاً یک کال کردن متد را تبدیل به آبجکت Stand-alone کنیم که اطلاعات در آن ذخیره شده, وقتی می‌خواهیم عملیات ها را صف‌بندی یا تأخیر را به‌صورت ریموت اجرا کنیم و یا وقتی می‌خواهیم عملیات های قابل بازگردانی داشته باشیم از این الگو استفاده می‌کنیم (به‌این‌ترتیب که کامندها را می‌توان load و Store کرد و بعد از مثلاً کرش کردن سیستم آن را restore کنیم.). هم چنین می‌توان یک سیستم خیلی بزرگ و به‌اصطلاح پیچیده را با استفاده از کامندهای ریزدانه و ابتدایی ساده ساخت.

Proxy - بسته به نوع Proxy کاربردهای متفاوتی دارد مثلاً Lazy Initialization (Virtual Proxy), Access Control (Protection Proxy), Logging (logging Proxy), Local execution of Remote Service (remote Proxy و یا dismiss کردن آبجکت وقتی کسی از آن استفاده نمی‌کند (Smart Reference)

Mediator – جاهایی که Coupling بین کلاس‌ها زیاد هســـت و تغییرات بســـیار ســـخت هســـت, وقتی می‌خواهیم از یک کامپوننت reuse کنیم اما به بقیه کامپوننت‌ها وابســـته اســـت و نمی‌توانیم این کار را انجام دهیم (چون بعد از تحقق این الگو همکاران از یکدیگر خبر ندارند و فقط از طریق اینترفیس میانجی با هم ارتباط دارند.) جاهایی که رفتار تعاملی زیاد هســـت و می‌خواهیم Extension انجام دهیم (با تعریف میانجی جدید می‌توان رفتار جدید تعریف کرد). جاهایی که ارتباطات بین آبجکت‌ها زیاد اســـت و باعث Shotgun Surgery می‌شود.

مقایســـه: می‌توان گفت این ۳ الگو هرکدام هدف و موارد اســـتفاده متفاوت و مجزایی دارند و شـــباهت و تفاوت خاصی دیده نمی‌شود.

الگوهای مرتبط

Command

۱- این الگو با Chain of responsibility, Mediator, Observer ارتباط دارد زیرا راه‌های ارتباطی بین Receiver و Sender ایجـاد می‌کنـد امـا بـا تفـاوت‌هـای خود مثلاً Chain of responsibility درخواســـت در یک زنجیره داینامیک حرکت می‌کند تا یکی به آن جواب دهد, یا مثلاً کامند ارتباط Invoker و Receiver را قطع کرده ولی آن‌ها با هم به‌صـورت یک‌طرفه کار می‌کند یا Observer از طریق Subscribe یا Unsubscribe شدن Receiver ها این کار را انجام می‌دهد.

۲- در Chain of Responsibility هندلر ها می‌تواند Command باشـد, یا خود درخواسـت به‌صـورت Command باشـد در این صـورت می‌توان در جا و Context مشـخص مناسـب دسـتورات متفاوت را اجرا کرد.

۳- از Memento و Command می‌توان برای پیاده‌سازی Undo استفاده کرد که Memento اطلاعات و State را بازگردانی می‌کند و Command دستورات و عملیات لازم را برای این کار انجام می‌دهد.

٤- Prototype می‌تواند برای کپی‌هایی از Command برای هیستوری دستورات استفاده شود.

٥- می‌توان Visitor را نسخه قدرتمندتر از Command در نظر گرفت چون عملیات مختلف را بر روی آبجکت‌های کلاس‌ها متفاوت اجرا می‌کند.

٦- می‌توان از الگوی Composite استفاده کرد تا از Command های پیچیده پشتیبانی کرد.

٧- می‌توان گفت هم Command و هم Strategy هدف‌های متفاوت دارند اما آبجکت‌ها را با رفتارهای مختلف پارامتریزه می‌کنند.

٨- می‌توان Invoker ها را در این الگو به‌صورت Singleton نوشت.

Proxy

١- Proxy در واقع یک Wrapper با همان اینترفیس هست اما Adapter اینترفیس متفاوت و Decorator در واقع اینترفیس بهبودیافته برای آبجکت Wrap شده است.

٢- مانند Façade در واقع هر ٢ یک Entity را Wrap کرده‌اند و هر ٢ نمونه‌سازی را به‌خودی‌خود انجام می‌دهند.

Mediator

١- مورد اول برای الگوهای مرتبط با الگوی Command برای این الگو نیز صادق است.

٢- Facade و Mediator در واقع ارتباطات کلاس‌های Tightly Coupled را مدیریت می‌کند, Mediator ارتباطات را به‌صورت مرکزی کنترل می‌کند و Facade نیز یک اینترفیس ساده شده از زیرسیستم تعریف می‌کند, زیرسیستم از وجود Facade بی‌خبر است, آبجکت‌ها در Facade می‌توانند مستقیماً ارتباط داشته باشند.

٣- تفاوت بین Observer و Mediator تقریباً خیلی کم است در خیلی از موارد می‌توان هر ٢ را برای یک Context پیاده‌سازی کرد حتی پیاده‌سازی ای از Mediator که بر مبنای Observer هست

هم وجود دارد, در واقع هر ۲ ارتباطات بین آبجکت‌ها را مدیریت می‌کنند یکی از طریق میانجی مرکزی و دیگری از طریق Subscription که ممکن است یک آبجکت زیردست دیگری باشد.

مقایسه: موارد مرتبط ذکر شد و واضح است که Mediator و Command هر ۲ تعریف نحوه ارتباطی جدید بین ارسال‌کننده و پذیرنده در اختیارمان قرار می‌دهند.

مزایا معایب

Command

۱- ارتباط بین Receiver و Invoker از بین می‌رود, این مزیت است.

۲- اضافه‌کردن Command های جدید راحت است, این مزیت است.

۳- از Command های Composite می‌توان پشتیبانی کرد, این مزیت است.

٤- دستورات مانا می‌شوند و می‌توان مانند آبجکت‌ها با آن کار کرد, این مزیت است.

٥- چون داریم اطلاعات دستورات را در حافظه اصلی می‌ریزیم تا با آن‌ها مانند آبجکت رفتار کنیم سربار حافظه زیاد می‌شود, این عیب است.

Proxy

۱- یک لول indirection اضافه می‌کند که این مزیت است.

۲- می‌توان از Smart reference استفاده کرد تا زمانی که از آبجکت استفاده نمی‌شود آن را Dispose کرد, این مزیت است.

۳- می‌توان از Virtual Proxy استفاده کرد و تا زمانی که لازم نیست آبجکت اصلی را به حافظه نیاورد, این مزیت است.

٤- کلاینت از این که آبجکت اصلی در فضای آدرس دیگری است خبر ندارد, این مزیت است.

Mediator

۱- ارتباط بین همکاران را انتزاعی می‌کند و آن‌ها از نحوه کار آبجکت میانجی خبر ندارد, این مزیت است.

۲- وابستگی بین همکاران غیرمستقیم می‌شود و تقریباً از بین می‌رود, این مزیت است.

۳- Subclassing محدود می‌شود و فقط از طریق Subclassing خود Mediator می‌توان رفتار را تغییر داد. این مزیت است.

٤- کنترل مرکزی باعث پیچیدگی کلاس خود Mediator می‌شود. این عیب است.

٥- Mediator را می‌توان به‌عنوان Single Point of Failure در نظر گرفت که این عیب است.

٦- می‌تواند باعث ایجاد God Class شود, این عیب است.

مقایسه: هرکدام مزایا و معایب خود را دارند و باید با توجه با Context مسئله تصمیم گرفت.

شیء جعلی

Command – کلاس و آبجکت‌های Command در جهت تحقق این الگو تولید شده است و جعلی است.

Proxy - آبجکت و کلاس Proxy در قلمرو مسئله نیست و برای تحقق الگو تولید شده و جعلی است.

Mediator – خود کلاس Mediator برای ساماندهی ارتباط تولید شده و قبل از آن ارتباط کلاس‌ها با یکدیگر مستقیم بوده پس جعلی است.

مقایسه: هر ۳ شیء جعلی تولید کرده‌اند.

جداسازی دغدغه‌ها

Command – این الگو تقریباً می‌توان گفت هرکدام از عملیات ها را از Invoker جدا کرده و وظیفهٔ انجام آن را فقط به Invoker سپرده و آبجکت‌های متخصص برای عملیات که همان Command هست تولید کرده. این یعنی جداسازی دغدغه‌ها. هم چنین این عملیات ها اطلاعی از این که عملیات را روی کدام دریافت‌کننده انجام می‌دهد, ندارند.

Proxy - این الگو تغییری در کار اصلی آبجکت اصلی انجام نمی‌دهد اما آبجکت اصلی و فرعی را مجبور می‌کند تا اینترفیس سطح بالای مشترک را محقق سازند.

Mediator – به‌وضوح کار ارتباطی بین همکاران را در دست گرفته و آن را از کار اصلی که همکاران انجام می‌دهد جدا می‌کند.

مقایسه: Mediator به‌خوبی این کار را انجام داده, الگوی Command نیز تا حد خوبی این کار را انجام می‌دهد, علاوه بر این که ارتباط بین Receiver و Invoker را Decoupled می‌کند, ولی Proxy کاری برای جداسازی دغدغه‌ها انجام نمی‌دهد.

پیاده‌سازی

Command – با استفاده از یک اینترفیس که متدهای مربوطه به کامند را داراست مثل Execute, Log, Undo و ... می‌توان آبجکت‌های از آن (Command)ساخت که به‌راحتی به Invoker داد که آن را اجرا کند. هم چنین در این زیر کلاس‌ها از Command می‌توان یک آبجکت Receiver قرار داد تا عملیات را روی آن اجرا کنند (Association). هم چنین فیلدهایی مثل on Start, on Finish وMain Command را

می‌توان در Invoker گذاشت و به‌صورت عمومی از آن استفاده کرد که همان‌طور که نام آن پیداست قبل و بعد از Main Command در صورت وجود اجرا می‌شوند.

Proxy - همان‌طور که بارها اشاره شد باتوجه‌به نوع Proxy می‌تواند پیاده‌سازی متفاوتی داشت مثلاً برای Logging علاوه بر محقق کردن اینترفیس سطح بالای یکسان قبل و بعد از دسترسی به آن باید Logging را انجام داد. هم چنین می‌توان مثلاً دسترسی‌های کلاینت به آبجکت اصلی را در این مرحله و توسط Proxy انجام داد. تمام Proxy ها باید اینترفیس یکسان با آبجکت اصلی داشته باشند.

Mediator – وقتی داریم نمونه‌ای از آبجکت میانجی می‌سازیم و نمونه‌های همکار را به آن پاس می‌دهیم در Constructor در واقع خود کلاس میانجی را به کلاس‌های همکار پاس می‌دهیم. پیاده‌سازی پیچیده‌ای ندارد اما باید در نظر داشت که یک میانجی را سخت و بزرگ نکنیم چون پتانسیل تبدیل‌شدن به God Class را دارد.

مقایسه: به نظر می‌توان گفت Proxy و Command در یک حد پیچیدگی دارند و برای پیاده‌سازی Mediator که مقداری نسبت به بقیه پیچیده‌تر می‌باشد باید از تبدیل‌شدن به God Class جلوگیری کرد.

بررسی الگوهای **GRASP** برای ۳ الگوی **Command, Proxy** و **Mediator**

Information Expert

Command – در این الگو کپسول‌های داده رفتار برای Invoker, Receiver و Command به‌خوبی ساخته شده است. درواقع برای Invoker فقط اطلاعات اینترفیس کامند و اطلاعات مربوط به شروع Command را دارد, Receiver اطلاعات و عملیات مربوطه به بیزینس لاجیکی که قرار است اجرا کند را دارد و هم چنین کامند هم به همین صورت است پس به‌خوبی رعایت شده است.

Proxy - باتوجه‌به نوع Proxy کارهای پسین و پیشین هر عملیات با آبجکت اصلی در این آبجکت انجام می‌شود که اطلاعات مربوط به آن یا در کلاس Proxy هست یا به‌صورت Loosely Coupled در کلاس مربوط به وظیفه خودش هست که فقط این کلاس از اینترفیس آن استفاده می‌کند. پس کپسول داده رفتار به‌خوبی رعایت می‌شود چرا که هر ۲ آبجکت اصلی و فرعی هر دو یک اینترفیس را محقق می‌سازند و Proxy به‌صورت یک Wrapper عمل می‌کند و داده رفتار مخصوص به خود را در کپسول خود دارد.

Mediator – ازآنجایی‌که رفتار برای ارتباط بین همکاران از آن‌ها جدا شده و به آبجکت متخصص این کار که همان Mediator است سپرده می‌شود برای همکاران پس به‌خوبی این الگو رعایت شده است. هم چنین Mediator خود متخصص ارتباطات بین آبجکت‌هاست و داده رفتار مخصوص به خود را دارد پس این الگو را رعایت کرده.

Creator

Command – در این الگو همان‌طور که مشخص است رابطهٔ بین Command و Receiver و هم چنین رابطه بین Invoker و Command به این صورت هست که اولی دارد با سرویس‌های دومی کار می‌کند

پس باید مسئول Creation باشد یعنی شرط ٤ اما چون پیکربند این کار را انجام می‌دهد در واقع شرط ٥ در نظر گرفته شده پس یعنی این الگو نقض شده است.

Proxy - در این الگو جون در واقع رابطه بین آبجکت اصلی و آبجکت جایگزین به‌صورت Aggregation هست پس باید شرط ۲ برقرار باشد و آبجکت فرعی مسئول تولید آبجکت اصلی باشد اما پیکرند در واقع این کار را انجام می‌دهد یعنی این الگو نقض شده است (به‌غیراز Virtual Proxy که آبجکت فرعی این کار را انجام می‌دهد.)

Mediator – ازآنجایی‌که کلاینت یا پیکربند دارد کلاس Mediator را می‌سازد پس الگو در واقع با شرط پنجم هست که این اشتباه است زیرا در واقع Mediator دارد با سرویس‌های کلاس‌های همکاران کار می‌کند یعنی شرط چهارم. پس این الگو نقض شده است.

Coupling

Command – در این الگو در واقع ارتباط مستقیم بین Invoker و Receiver از بین رفته هم چنین Invoker فقط در سطح انتزاعی و از طریق اینترفیس با Command کار می‌کند پس Coupling پایینی دارد اما هنوز Command ها با کلاس‌های Concrete یک Receiver کار می‌کنند و هم چنین پیکربند نیز با هر ۳ این کلاس‌ها در سطح Concrete کار می‌کند.

Proxy –در این الگو چون هر دو آبجکت اصلی و فرعی یک اینترفیس را محققمی‌سازدند و آبجکت فرعی به‌صورت یک Wrapper عمل می‌کند و ارتباط به داخل آبجکت اصلی ندارد Coupling پایینی دارد. کلاینت نیز با این اینترفیس کار می‌کند و نمیفهمد که اصلی است یا فرعی .

Mediator – همان‌طور که قبلاً هم اشـاره شـده ارتباطات بی‌نظم بین همکاران از بین رفته و همکاران فقط اینترفیس میانجی را می‌شــناســند و با آن کار می‌کنند ولی برعکس میانجی برای کار کردن با همکاران در واقع با آبجکت‌های Concrete این کار را انجام می‌دهد. می‌توان گفت در ســطح همکاران و ارتباطات آن به میانجی این الگو رعایت شده اما برعکس خیر.

Cohesion

Command – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر تقریباً خوبی برای این شاخص دارد و می‌توان گفت باعث افزایش Cohesion می‌شود.

Proxy - با رجوع به قســمت افزایش چســبندگی برای این الگو متوجه می‌شــویم که تأثیر زیادی برای این الگوی Grasp ندارد و تقریباً از Cohesion بالایی برخوردار است.

Mediator – با رجوع به قســمت افزایش چســبندگی برای این الگو متوجه می‌شــویم که تأثیر زیادی برای این الگوی Grasp دارد و آن را محقق می‌سازد.

Controller

Command – این الگو در واقع کارچرخانی ندارد و Controller ندارد. هم چنین همان کلاینت, پیکربندی را انجام می‌دهد.

Proxy - می‌توان Proxy را در انواع مختلف پراکســـی به‌عنوان کـارچرخـان در نظر گفـت زیرا از کلاینـت درخواست‌ها را به آبجکت اصلی انتقال می‌دهد و جواب را برمی‌گرداند.

Mediator – کلاس Mediator در واقع وظیفه کارچرخانی را دارد زیرا که ارتباطات میان همکاران را اداره می‌کند در واقع Controller است.

Polymorphism

Command – این الگو با داشـــتن فوق کلاس برای Command ها و اجرای متفاوت از زیر کلاس‌های متفاوت که بر اساس نوع کلاس, رفتار متفاوتی دارند این الگو را رعایت می‌کند.

Proxy - این الگو با داشتن اینترفیس مشترک برای آبجکت اصلی و فرعی و بسته به نوع Proxy دستورات متفاوت را با اینترفیس یکسان و دارای رفتار متفاوت را اجرا کرد (با اینترفیس یکسان).

Mediator – در واقع چون در این الگو همکاران با اینترفیس میانجی کار می‌کند می‌توان میانجی گرهای متفاوت ایجاد کرد و به‌راحتی این الگو را رعایت کرد یعنی رفتار متفاوت بر اساس type یا class داشت.

Indirection

Command – همان‌طور که در بحث الگوهای مرتبط گفته شـد ارتباط مسـتقیم بین Invoker و Receiver در این الگو شـکسـته شـده و از طریق Command آن هم از طریق اینترفیس آن توسـط Invoker در هر زمان که لازم هسـت Execute می‌شـود و این یعنی غیرمسـتقیم کردن ارتباط بین Invoker و Receiver و ارتبـاط آن‌هـا از طریق اینترفیس ســـطح بـالای کلاس Command. ولی خود Command اگر از طریق اینترفیس با Receiver کار می‌کرد خیلی عالی می‌شـــود (در حال حاضـــر با کلاس‌های ســـطح پایین Receiver است.)

Proxy - این الگو با اضافه‌کردن یک سطح جدیدی از Indirection بسته به نوع پراکسی باعث می‌شود که کلاینت به‌صورت مستقیم به آبجکت اصلی دسترسی نداشته باشد.

Mediator – این کلاس در واقع یک سطحی از Indirection را به سیستم اضافه می‌کند و همکاران دیگر با یکدیگر کار نمی‌کنند و در واقع رفتار تعاملی آن‌ها از طریق کلاس میانجی انجام می‌گیرد.

Pure Fabrication

Command – همان‌طور که در بحث آبجکت‌های جعلی گفته شد در واقع این الگو کلاس Command را که در مسئله نیست می‌سازد.

Proxy - همان‌طور که در بحث آبجکت‌های جعلی گفته شد این الگو برای جداسازی آبجکت اصلی از کلاینت آبجکت Proxy را که در فضای مسئله نیست را می‌سازد.

Mediator – همان‌طور که در بحث آبجکت‌های جعلی گفته شد این الگو در واقع برای ساماندهی ارتباطات بین همکاران آبجکت Mediator را می‌سازد.

Protected Variations

Command – همان‌طور که در بحث تغییرپذیری گفته شد چون ارتباط بین Invoker و Receiver از طریق اینترفیس Command می‌باشد و آن‌ها نمی‌داند کدام کامند را اجرا کرده‌اند و در سطح انتزاعی کار می‌کنند در واقع می‌توان گفت که تغییرات منتشر شونده نداریم اما درصورتی‌که Receiver ها از اینترفیس سطح بالایی استفاده نکنند با تغییرات آن‌ها باید Command های آن‌ها نیز تغییر کند زیرا در سطح

Concrete کـار می‌کننـد. هم چنین پیکربنـد بـایـد از تغییرات Invoker و Receiver و انواع کلاس‌هـای Command باخبر باشد.

Proxy - همان‌طور که در بحث تغییرپذیری گفته شـد ارتباط بین آبجکت اصـلی و فرعی که در اختیار کلاینت گذاشــته شــده فقط در ســطح اینترفیس آن‌هاســت و با تغییر در آبجکت اصـلی یا فرعی هیچ‌کدام و حتی کلاینت از تغییرات آن‌ها مطلع نمی‌شـود و نمی‌داند که داخل آن‌ها چه خبر اسـت و تغییرات منتشـر شـونده نداریم. فقط کلاینت باید از اینترفیس پراکسـی باخبر باشد.

Mediator – همان‌طور که در بحث تغییرپذیری گفته شــد این الگو نیز مانند دو الگوی دیگر چون همکاران با اینترفیس کلاس Mediator کار می‌کند در واقع تغییرات منتشـر شـونده نداریم ولی برعکس هر Mediator در واقع با کلاس‌های Concrete همکاران کار می‌کند و در صـــورت تغییر آن‌ها کلاس Mediator دچار تغییر می‌شود.

مقایسه الگوهای State, Strategy و Visitor

دسته - نوع

State – این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

Strategy - این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

Visitor - این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

مقایسه: هر ۳ الگو جزو دسته الگوهای رفتاری می‌باشد و دغدغه و وظیفه‌های یکسانی دارند.

هدف

State – این الگو برای تغییر رفتار یک آبجکت در زمان اجرا به کار می‌رود (با تغییر State داخلی). همانند آن است که انگار یک آبجکت کلاس خود را تغییر داده است. این الگو رفتار وابسته به حالت را در کلاس‌های جدای Stateمی‌سازد و آبجکت اصلی را مجبور می‌سازد تا از طریق Delegation و از طریق یکی از این کلاس‌ها کار را انجام دهد (در ادامه توضیح داده می‌شود.).

Strategy - این الگو مجموعه‌ای از رفتارها را درون آبجکت‌ها می‌سازد تا به نحوی قابل Interchange شدن در آبجکت زمینهٔ اصلی شوند. در واقع این آبجکت Context به‌صورت Aggregation نمونه از این آبجکت‌های Strategy برای انجام رفتار دارد. در واقع مجموعه‌ای از الگوریتم‌ها داریم تا برای پیاده‌سازی متدها از الگوریتم‌های متفاوت بر اساس شرایط استفاده کرد.

Visitor – این الگو به ما کمک می‌کند تا رفتار جدید را به کلاس‌هایی که موجود هست (ساختاری از اشیا) اضافه کنیم بدون اینکه کد آن‌ها را تغییر بدهیم. یعنی الگوریتم‌ها را از آبجکت‌هایی که بر روی آن‌ها قرار است اجرا شود جدا می‌کنیم.

مقایسه: با این که هدف دو الگوی State و Strategy متفاوت است و اولی برای جداسازی رفتار وابسته به حالت از Context و دومی برای تغییر الگوریتم در زمان اجرا است, می‌توان گفت الگوی State و Strategy در واقع دارند تغییر رفتار یک آبجکت در زمان اجرا را به‌وسیله Delegation و از راه Aggregation انجام می‌دهند با این تفاوت که در Strategy در واقع این آبجکت‌های Strategy کاملاً از هم جدا و مستقل هستند اما در State می‌تواند این‌گونه نباشد. الگوی Visitor نیز در واقع رفتار جدید به آبجکت‌هایی که موجود هست اضافه می‌کند.

حوزه

State – این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Strategy - این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Visitor - این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

مقایسه: هر سه الگو در زمان اجرا و از طریق delegation تحقق می‌یابند.

کاهش وابستگی

State – با اعمال این الگو Coupling که مستقیم بین الگوریتم و کلاس Context هست از بین می‌رود, به‌این‌ترتیب که بعد از اعمال الگو دیگر کلاس زمینه نمی‌داند کدام حالت را در خود دارد و فقط از طریق اینترفیس یا فوق کلاس انتزاعی آن عملیات ها را انجام می‌دهد, هم چنین می‌توان درون کلاس‌های State در واقع Back reference به آبجکت Context نداشت (برای تغییر State ها به یکدیگر لازم می‌شود) و در این حالت Coupling به‌شدت پایین می‌آید. فقط و فقط در اینجا ثالث یا پیکربند باید کلاس‌های Concrete حالت‌ها را برای پیکربندی بشناسد.

Strategy - این الگو نیز مانند State در واقع ارتباط مستقیم بین الگوریتم و کلاس Context از بین رفته و فقط از طریق اینترفیس سطح بالای کلاس‌های Strategy استفاده می‌شود (توسط Context) و می‌توان گفت که Coupling کمی دارد ولی از طرفی کلاینت باید تمام کلاس‌های Concrete را برای پیکربندی ببیند.

Visitor – ازآنجایی‌که آبجکت‌های Element در واقع با اینترفیس Visitor کار می‌کند و تغییرات در آن به کلاس‌های Element منتشر نمی‌شود, در اینجا Coupling بسیار پایین هست. اما وقتی آبجکت‌های

Visitor به حالت داخلی Concrete Element ها می‌تواند دسترسی داشته باشد و ارتباط آن‌ها در سطح انتزاعی نیست چون لازم است بعضی از عملیات ها را اجرا کند در اینجا Coupling وجود دارد و تغییرات در Element ها به Visitor منتشر می‌شود (می‌توان برای Element ها هم ساختار مشخص در نظر گرفت یعنی علاوه بر متد Accept دیگر متدها نیز در اینترفیس تعریف شود و در اینترفیس Visitor برای متدهای Visit از این اینترفیس استفاده کرد تا Coupling به پایین‌ترین حد ممکن برسد.). پیکربند نیز فقط از طریق اینترفیس با Element ها کار می‌کند پس Coupling پایین هست, اما برای پیکربندی و به‌اصطلاح Visit کردن لازم هست به انواع مختلف Visitor دید داشته باشد (این دید Dependency هست و مانا نیست.).

مقایسه: برای دو الگوی تقریباً مشابه State و Strategy می‌توان گفت که Coupling پایینی دارند مگر وابستگی بین Client و کلاس‌های مجرد State یا Strategy که باید آن‌ها را علاوه بر Context بشناسد, اما در Visitor از طرف Element ها به Visitor ارتباط از طریق اینترفیس می‌باشد و Coupling پایینی دارد اما برعکس از سمت Visitor ارتباط با Concrete Element ها است و باعث می‌شود تا حدی به هم Coupled باشند, هم چنین پیکربند در این الگو از طریق اینترفیس با Element ها کار می‌کند.

افزایش چسبندگی

State – با اعمال این الگو چند کارگی بودن Context که قبلاً الگوریتم‌های مختلف در آن بود کم شده و الگوریتم‌ها و رفتارها در کلاس‌های متخصص نوشته شده است, این باعث شده که کلاس‌های State فقط یک وظیفه داشته باشد و هم چنین Context نیز از چندکارگی بودن آن کم شود به‌این‌ترتیب بسیار Cohesive شده است. هم چنین Switch Statement های طولانی از بین می‌رود و رفتار وابسته به حالت با این الگو محقق می‌شود.

Strategy - این الگو نیز مانند الگوی State نیز Cohesive شـده اسـت. درسـت اسـت که هدف دو الگو متفاوت اسـت اما چون خانواده‌ای از الگوریتم‌ها قبل از اعمال الگو درون Context توسـط عبارات شـرطی طولانی تعیین می‌شـد, بعد از اعمال الگو این الگوریتم‌ها در کلاس‌های Strategy پیاده‌سـازی شـده و در زمان اجرا به Context داده شـده و می‌توانند تعویض شـوند, پس می‌توان گفت تک کارگی روی این کلاس‌های Strategy به‌خوبی رعایت شده.

Visitor – با اعمال این الگو در واقع عملیات هایی که در کلاس‌های Element بوده جدا شـده و در آبجکت‌های متخصص Concrete Visitor قرار گرفته اسـت و باعث شـده این کلاس‌ها منسـجم باشـد و چندکارگی از کلاس‌های Element کم شده و منسجم باشند. پس Cohesion بالایی دارد.

مقایسه: هر سه الگو هرکدام به صورتی Cohesion بالایی دارند یا باعث افزایش آن شده‌اند, در واقع هر ۳ با کاهش چندکارگی در کلاس‌های Context (دو الگوی State و Strategy) و Element (در الگوی Visitor) باعث افزایش Cohesion شده‌اند و هم چنین با تولید آبجکت‌های متخصص اجرای الگوریتم و رفتار باعث افزایش Cohesion شده‌اند.

OCP

State – در این الگو در واقع چون رفتار از کلاس Context جدا شـده و به‌صـورت State به آن Delegate شـده (بهتر اسـت بگوییم State درون یک Context به‌صـورت Aggregated آمده اسـت) و این رفتار با استفاده از اینترفیس یا کلاس سطح بالای State انجام می‌پذیرد این اصل رعایت شده ولی برعکس برای ارتباط بین State ها (مثلاً بعد از انجام کاری به State دیگر تبدیل شـود) نیاز هسـت تا کلاس Context را به‌صـورت Concrete داشـته باشـد و حالت داخلی آن را تغییر دهد این نقض OCP اسـت (می‌تواند این قابلیت را نداشت).

Strategy - در این الگو نیز چون خود کلاس Context الگوریتم‌ها را به‌صـــورت Aggregate شـــده دریافت می‌کند و آن‌ها را از طریق اینترفیس ســـطح بالای آن‌ها اجرا می‌کند و تغییرات در آن‌ها به کلاس Context منتشـر نمی‌شـود به‌خوبی این اصـل رعایت شـده اسـت. ولی فقط پیکربند باید از انواع Strategy ها باخبر باشـد و بتواند آن‌ها را به Context واسپاری کند.

Visitor – همان‌طور که قبلاً اشـــاره شـــد چون رفتارهای چندگانه از کلاس‌های Element جدا شـــده و در آبجکت‌های متخصص Visitor قرار گرفته اسـت و این Element ها در واقع از طریق اینترفیس Visitor با آن کار می‌کند تغییرات در Visitor ها به Element منتقل نمی‌شـــود یعنی این اصـــل رعایت شـــده اما چون Visitor ها برای انجام عملیات های خاص هر Element دید مســـتقیم به آبجکت‌های Concrete دارد تغییرات در آن‌ها باعث تغییر در Visitor ها خواهد شد که این نقض این اصل است.

مقایسـه: همان‌طور که گفته شـد در Strategy به‌خوبی این اصل رعایت شـده, در State بدون درنظرگرفتن ارتباط بین State ها برای تغییر حالت نیز به‌خوبی رعایت شـــده, اما در Visitor همان‌طور که در بالا گفته شـــد چون تغییرات در Element ها باعث تغییرات در Visitor ها می‌شـــود می‌توان گفت این‌ســـوی ارتباط باعث نقض شدن این اصل می‌شود.

LSP

State – کاملاً برقرار اسـت چون زیر کلاس‌های State به‌راحتی می‌تواند جایگزین Super class آن شوند و Context نیاز نیســـت از چیزی خبر داشـــته باشـــد. فقط پیکربند باید انواع مختلف آن را برای پیکربندی یا بازپیکربندی بشـــناســـد. همچنین می‌توان درون کلاس‌های State در واقع Back reference به آبجکت Context داشـت (برای تغییر State ها به یکدیگر لازم می‌شود).

Strategy - کاملاً برقرار اســت چون زیر کلاس‌های Strategy می‌توانند به‌جای پدر خود اســتفاده شــوند بدون این که تغییری در رفتار یا کد ایجاد شود.

Visitor – برای Visitor ها کاملاً برقرار است زیرا از اینترفیس سطح بالا یا از super class استفاده می‌کند و به‌نوعی مانند دو الگوی بالا رابطه is-a برقرار اســـت و هم چنین این مورد برای Element ها نیز برقرار است پس این اصل برقرار است.

مقایسه: هر ۳ الگو این اصل را برقرار می‌سازند.

DIP

State – همان‌طور که قبل‌تر اشـاره شـد ارتباط بین Context و State از طریق اینترفیس آن‌ها اسـت و این اصل برقرار اسـت, اما وقتی لازم اسـت برای تغییر State ها به یکدیگر بعد یا قبل از انجام عملیاتی در واقع فیلد State در آبجکت Context را عوض کنیم پس لازم اسـت کلاس Concrete که از Context داریم را به State پاس دهیم و این نقض DIP است (می‌توان این قابلیت را نداشت).

Strategy - ازآنجایی‌که ارتباط بین کلاس‌های Context و Strategy فقط از طریق اینترفیس آن اســـت و Context در واقع نمی‌داند کدام را دارد اجرا می‌کند و تغییرات به آن منتشر نمی‌شود در واقع DIP به‌خوبی رعایت شده است.

Visitor – همان‌طور که پیش‌تر گفته شــد ازآنجایی‌که آبجکت‌های Element ها با اینترفیس ســطح بالای Visitor کار می‌کنند و از تغییرات آن بی‌خبر هســتند DIP برقرار اســت اما از طرف Visitor ها به Element ها چون هرکدام می‌تواند رفتار مخصــوص به خود را داشـــته باشـــد ارتباط از طریق کلاس‌های Concrete

هســـت و این نقض DIP اســـت. پیکربند نیز با اینترفیس المنت‌ها کار می‌کند اما باید انواع Concrete Visitor ها را بشناسد و در سطح پایین با آن کار کند.

مقایسه: در الگوی Strategy به‌خوبی این اصل رعایت شده است, در الگوی State اگر تغییرات الگوها به یکدیگر را در نظر نگیریم و در نتیجه نیازی به ارتباط State ها به کلاس‌های سطح پایین Context نداشته باشد نیز این اصل به‌خوبی رعایت می‌شود اما در الگوی Visitor همان گونه که در بالا گفته شد از سمت Visitor ها به سمت Element ها این اصل رعایت نمی‌شود.

ISP

State – هر آبجکت State فقط اینترفیس یا فوق کلاس ســـطح بالای خود که دارای متد Context Set یا انجام الگوریتم‌های مربوطه است پس به‌خوبی این اصل رعایت شده.

Strategy - در این الگو چون الگوریتم‌های لازم در هر کلاس Strategy در اینترفیس آن هست به‌خوبی این اصل رعایت شده است.

Visitor – در این الگو برای المنت‌ها فقط متد Accept در اینترفیس ســطح بالای Element نوشــته شــده است که این نشان‌دهندهٔ Cohesion و Specific بودن آن است که بسیار عالی است و این اصل را رعایت می‌کند, اما در ســـمت Visitor در اینترفیس آن مجبوریم برای ویزیت کردن هر کلاس متدهای جداگانه بنویســـیم که هم می‌توان گفت این اصـــل نقض شـــده هم نه, زیرا این الگو ایجاب می‌کند که این‌گونه پیاده‌سازی شود اما با تکه‌تکه کردن اینترفیس‌ها می‌توان آن‌ها را Specific تر کرد.

مقایسه: می‌توان گفت هر ۳ الگو این اصل را تا حد خوبی رعایت کرده است اما برای Visitor همان‌طور که گفته شد برای قسمت اینتفریس Visitor ها می‌تواند کمی این اصل را نقض کند و شاید بتوان با توسعه

دادن چندین اینترفیس Specific تر به‌جای یک اینترفیس General اصـــل را رعایت کرد اما به پیچیدگی کد اضافه می‌شود.

CRP

State – به حد عالی این اصل رعایت شده و به‌جای ساختار توارثی از واسپاری استفاده شده است.

Strategy - به حد عالی این اصـل رعایت شـــده و به‌جای ســاختار توارثی از حالت Delegation اســتفاده شده است.

Visitor – چون در این الگو رفتارهای مختلف که در زمان اجرا به آبجکت‌های اضافه می‌شـود از خود آبجکت جدا شـده و به متخصص سـپرده شـده اسـت, می‌توان گفت که از سـاختارهای توارثی که برای reuse در Element ها صورت می‌گیرد جلوگیری کرده است و به Visitor ها واسپاری شده است. (اما خود Visitor ها از ساختارهای توارثی استفاده می‌کنند و تقریباً همه یک اینترفیس سطح بالا را محقق می‌سازند.)

مقایسه: هر ۳ الگو به‌خوبی این اصل را رعایت می‌کنند.

PLK

State – این اصـل در این الگو رعایت شـده زیرا ارتباط بین Context و کلاس‌های State از طریق اینترفیس هسـت و ارتباطات از طریق ارسـال خود آبجکت‌ها صورت نمی‌گیرد, اما همان‌طور که گفته شـده برای تغییر State ها به یکدیگر در زمان اجرا نیاز هســـت که خود آبجکت Context هنگام ســـاخت State ها به آن‌ها پاس داده شـود (می‌توان این قابلیت را نداشـت و از پیکربند اسـتفاده کرد که در آن صورت به‌خوبی این اصل رعایت شده است).

Strategy - این اصـل در این الگو به‌خوبی رعایت شـده زیرا فقط از طریق اینترفیس و انتقال پیام با یکدیگر کار می‌کنند نه ارسال خود آبجکت‌ها به یکدیگر.

Visitor – جون برای Accept هر Element آبجکت Concrete Visitor ارسـال شـده و همان جا برای خود Visitor در واقع خود Concrete Element ارسـال می‌شـود این اصـل به‌کلی نقض شـده اسـت زیرا که Visitor به حالت داخلی Element دسترسی دارد.

مقایسـه: دو الگوی State و Strategy به‌خوبی این اصـل را رعایت کرده اما Visitor این اصـل را نقض می‌کند.

بسته‌بندی

State – در این الگو چون رفتارها و داده‌هایی که قبلاً در کلاس Context بوده یا حالا به‌صـورت سـاختار توارثی نوشـته شـده بوده, حال به‌صورت کلاس‌های مجزا که به آن State می‌گوییم در می‌آیند و داده رفتار آن‌ها مسـتقل از هم اسـت, حال کلاس Context فقط از اینترفیس آن اسـتفاده می‌کند تا آن‌ها را اجرا کند. به‌خوبی بسـته‌بندی یا کپسـوله‌سـازی انجام شـده اسـت. می‌توان درون کلاس‌های State در واقع Back reference به آبجکت Context داشـت (برای تغییر State ها به یکدیگر لازم می‌شـود) که این باعث نقض بسته‌بندی می‌شود.

Strategy - این الگو نیز مانند الگوی State چون مجموعه‌ای از الگوریتم‌ها را در کلاس‌های Strategy دارد و کلاس Context به‌صـورت Aggregation و فقط از این کلاس‌ها و از اینترفیس آن‌ها اسـتفاده می‌کند. به‌خوبی بسته‌بندی یا کپسوله‌سازی انجام شده است.

Visitor – در این الگو چون در متدهای Accept که در Element ها پیاده‌سازی شده, هر متد خود را به Visitor مربوطه می‌فرستد و در این حال این Visitor که به‌صورت Concrete به State داخلی این Element دسترسی دارد و این باعث نقض Encapsulation می‌باشد.

مقایسه: دو الگوی State و Strategy به‌خوبی بسته‌بندی را انجام داده اما الگوی Visitor چون در واقع این رفتارها از آبجکت‌ها جدا شده و می‌تواند بر روی آن‌ها اجرا شود, می‌توان گفت که هم بسته‌بندی نقض شده و هم به خاطر داشتن چنین قابلیتی که الگوریتم‌ها از آبجکت‌ها جدا شود نیاز هست که این ویژگی نقض شود.

انعطاف‌پذیری

State – در این الگو چون رفتارها به‌صورت توارثی نیست و به‌راحتی می‌توان رفتار جدید در کلاس State جدید تعریف کرد و کلاس Context به‌راحتی از آن استفاده کند بدون آنکه متوجه آن شود (چون از اینترفیس آن استفاده می‌کند). هم چنین در زمان اجرا نیز می‌توان رفتارها (یعنی State را عوض کرد). فقط پیکربند هست که باید از انواع کلاس‌های State برای پیکربندی Context باخبر باشد (یعنی کلاس‌های Concrete و سطح پایین). پس از انعطاف‌پذیری بالایی برخوردار است.

Strategy - این الگو نیز همانند الگوی State چون در زمان اجرا می‌توان الگوریتم یا آبجکت Strategy را عوض کند و هم چنین با تعریف Strategy جدید نیاز نیست Context تغییری کند و به‌راحتی از طریق اینترفیس با آن کار می‌کند و فقط پیکربند هست که برای پیکربندی با کلاس‌های سطح پایین در ارتباط است, می‌توان گفت انعطاف‌پذیری بالایی دارد.

Visitor – این الگو برای جدا کردن الگوریتم‌ها از آبجکت‌ها به‌صورت خوبی عمل کرده اما برای تعریف Element جدید اینترفیس Visitor باید تغییر کند و تمام Concrete Visitor ها باید برای آن آبجکت

Implementation برای متد Visit داشته باشند که این بسیار بد هست, اما برای تعریف Visitor جدید راحت کار انجام می‌شود و فقط همان اینترفیس پیاده‌سازی می‌شود. هم چنین پیکربند باید از تمام کلاس‌های سطح پایین Visitor باخبر باشد. هم چنین اضافه‌کردن یک عملیات جدید ساده است.

مقایسه: دو الگوی State و Strategy انعطاف‌پذیری بسیار خوبی دارند اما الگوی Visitor همان‌طور که اشاره شده برای تولید Element جدید انعطاف‌پذیری بسیار پایینی دارد.

تغییرپذیری/ انتشار تغییرات

State – در این الگو رفتارها از کلاس Context جدا شده و به‌صورت کلاس‌های Concrete و با نام State درآمده و این کلاس‌ها به‌صورت Aggregation در اختیار Context قرار می‌گیرد پس به‌راحتی می‌توان State جدید تعریف کرد بدون آنکه Context متوجه آن باشد. همان‌طور که قبلاً هم اشاره شد برای جابه‌جایی بین State ها وقتی عملیاتی انجام شده یا قبل از انجام یک عملیات طی یک شرط خاص, لازم هست که State در واقع Reference به آبجکت Context داشته باشد (می‌توان این قابلیت را نداشت). هم چنین پیکربند باید از کلاس‌های سطح پایین State برای پیکربندی خبر داشته باشد. در کل انتشار تغییرات خیلی کم هست و تغییرپذیری بالا.

Strategy - در این الگو نیز به دلیل جداشدن الگوریتم‌ها از آبجکت‌هایی که قرار هست بر روی آن انجام شود و قراردادن در کلاس‌های Strategy به‌راحتی می‌توان آن‌ها را تغییر داد و یا Strategy جدید تعریف کرد و از آن‌ها در کلاس Context استفاده کرد بدون آنکه تغییری لازم باشد. یعنی تغییرپذیری در بالاترین حالت خود می‌باشد. فقط پیکربند باید از انواع کلاس‌های Concrete Strategy خبر داشته باشد.

Visitor – در این الگو همان‌طور که قبلاً اشاره شد وقتی بحث اضافه‌کردن یک Visitor جدید باشد به‌راحتی این کار انجام می‌شود و فقط لازم است پیکربند این کلاس سطح پایین را برای پیکربندی‌های بعدی خود

بشناسد, ولی اگر بحث اضافه‌کردن Element جدید باشد این تغییر در تمام کلاس‌های Visitor و همچنین اینترفیس آن منتشر می‌شود.

مقایسه: الگوی Strategy تغییرپذیری بالا و انتشار تغییرات پایین دارد, الگوی State نیز همین‌طور, اما برای الگوی می‌توان این Trade-off را در نظر گرفت که اضافه‌کردن Visitor های جدید ساده و بدون انتشار تغییرات اما اضافه‌کردن یک Element جدید منتشرکننده تغییرات است, اما هدف الگو که جدا کردن الگوریتم‌ها از آبجکت‌هایی که روی آن اجرا می‌شود مهم‌تر است.

پیکربندی

State – برای پیکربندی می‌توان یک ثالث داشت تا انواع State ها را بشناسد و با استفاده از آن‌ها پیکربندی را با Context انجام دهد, اما می‌توان یک State دیفالت تعریف کرد و سپس درون کلاس‌های State در واقع Back reference به آبجکت Context داشت و از آن برای تغییر State ها به یکدیگر استفاده کرد.

Strategy - یک پیکربند ثالث لازم است وجود داشته باشد و از انواع Strategy ها خبر داشته باشد و بتواند Context را با آن, برای اجرای عملیات پارامتریزه کند.

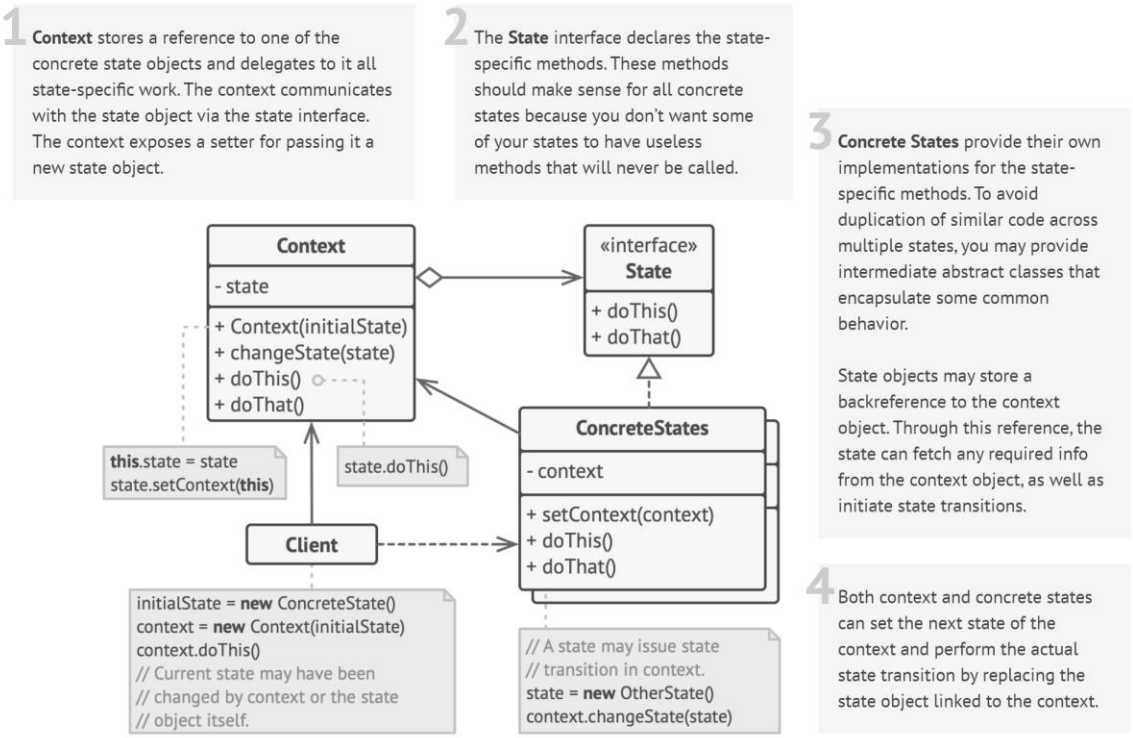
Visitor – پیکربند که همان کلاینت است لازم است تا انواع Element ها را در دست داشته باشد و از طریق اینترفیس مشترک آن‌ها متد Accept را صدا بزند و یکی از Concrete Visitor ها را به او پاس دهد, پس یعنی باید کلاس‌های سطح پایین Visitor ها را بشناسد.

مقایسه: همان‌طور که گفته شد State می‌تواند به پیکربند نیاز نداشته باشد. اما برای Strategy یک پیکربند ثالث لازم است و هم چنین Visitor نیز پیکربند خارج از مجموعه الگو دارد.

ساختار و رفتار

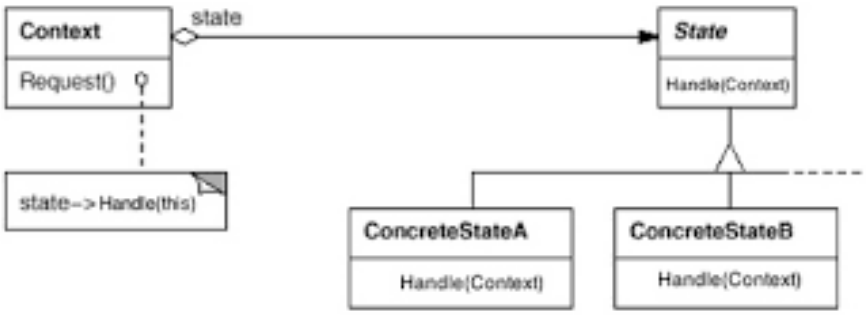
در این قسمت توضیحات تکمیلی در شکل موجود است.

State



شکل ٤٫١ ساختار الگوی State [٤]

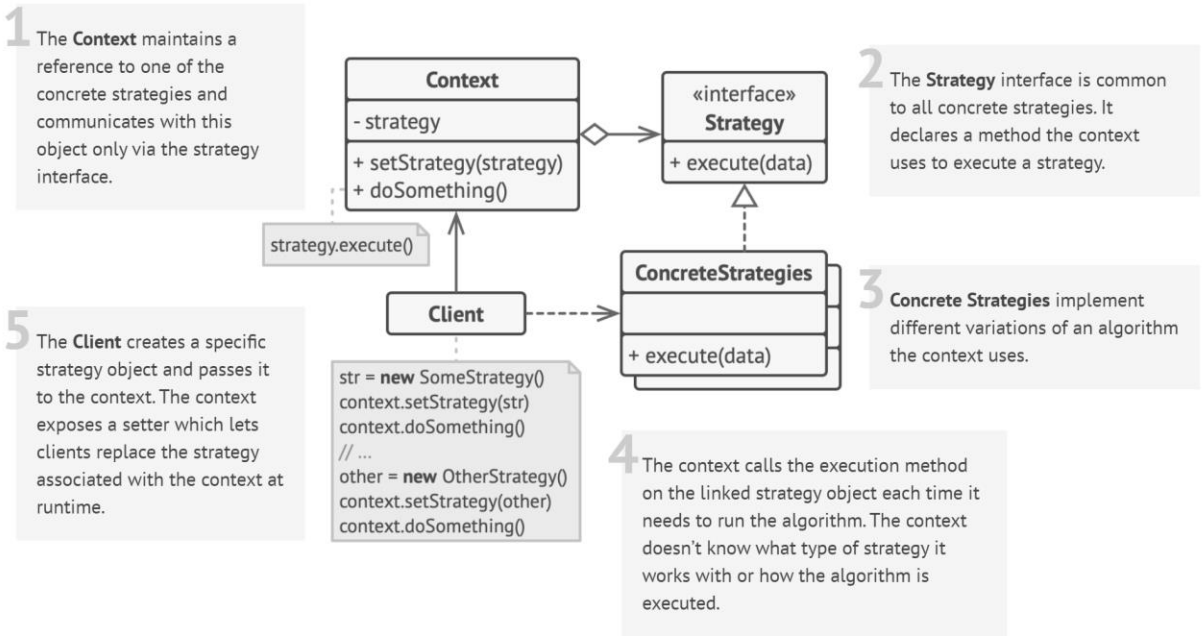
همان‌طور که در شـــکل ٤٫١ می‌بینید Client که در واقع همان پیکربند اســـت رابطه Association دارد به Context و Dependency دارد به کلاس‌های Concrete که از State ساخته شده است (که مانا نیست و بیشـــتر برای نمونه‌ســـازی و پیکربندی اســـت) , با این الگو می‌توان در واقع حالت‌های مختلف رفتار را در کلاس‌های State ساخت و در زمان اجرا و بسته به شرایط رفتار آبجکت Context تغییر می‌کند. این کار از طریق Delegation انجام می‌پذیرد (به‌جای ساختارهای توارثی). Context فقط از طریق اینترفیس سطح بالای State با آن‌ها کار می‌کند. همچنین می‌توان درون کلاس‌های State در واقع Back reference به آبجکت Context داشت (برای تغییر State ها به یکدیگر لازم می‌شود).



شکل ۳٫۲ ساختار الگوی State کتاب

همان‌طور که مشـخص اسـت تنها تفاوت این ۲ سـاختار ارتباط Association بین Concrete State ها و Context در ساختار ۳٫۱ هسـت که آن هم برای همان Back Reference برای تغییر State ها به یکدیگر است.

Strategy

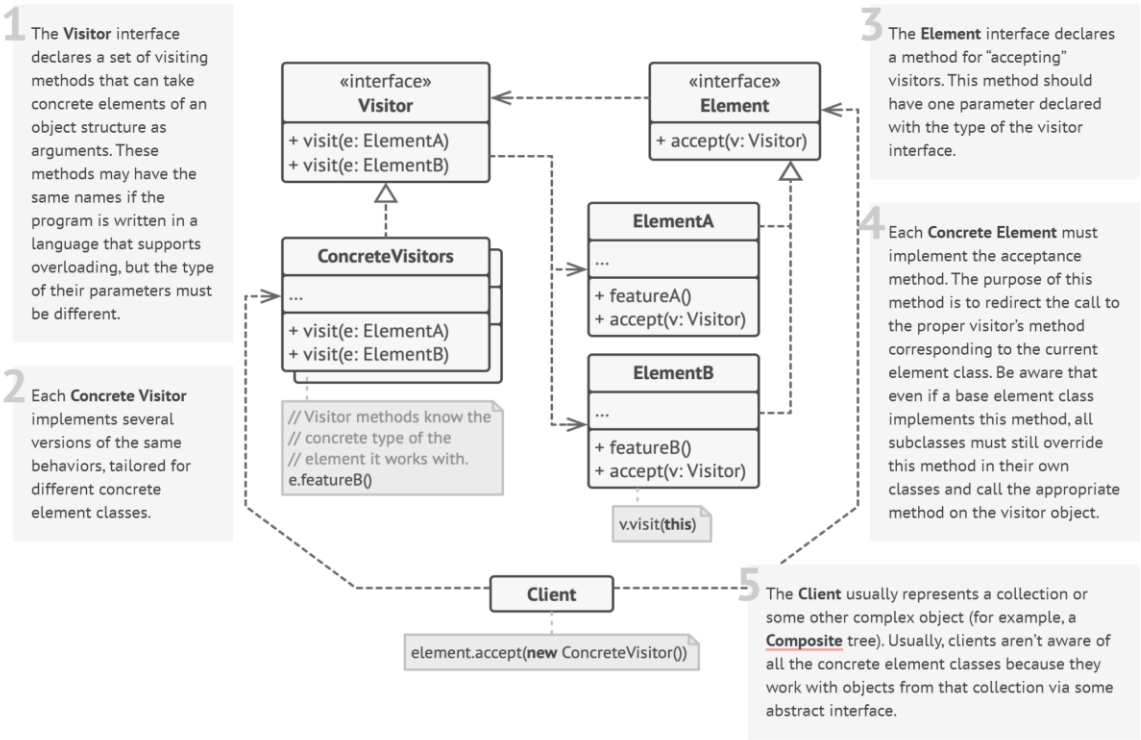


شکل ۵ ساختار الگوی Strategy [۵]

همان‌طور که در شکل ۵ مشخص است کلاینت با Delegate کردن Strategy های مختلف به کلاس زمینه باعث انجام الگوریتم‌های متفاوت بر روی این آبجکت می‌شـــود, تفاوت آن با State ارتباط نداشـــتن کلاس Context با Concrete Strategies هسـت زیرا تغییر حالت بین State های مختلف که در سـاختار State با نام Change State هســـت نداریم. این الگو در واقع با اینترفیس کلاس Strategy یکی از مجموعه الگوریتم‌های نوشـــته شـــده را روی آبجکت زمینه اجرا می‌کند. Context نمی‌داند کدام Strategy را اجرا می‌کند و فقط اینترفیس آن را می‌بیند. پیکربند انواع Strategy ها را برای پیکربندی باید بشـــناســـد (Dependency که مانا نیست). می‌توان مانند State نیز دید معکوس را با Back Reference ایجاد کرد.

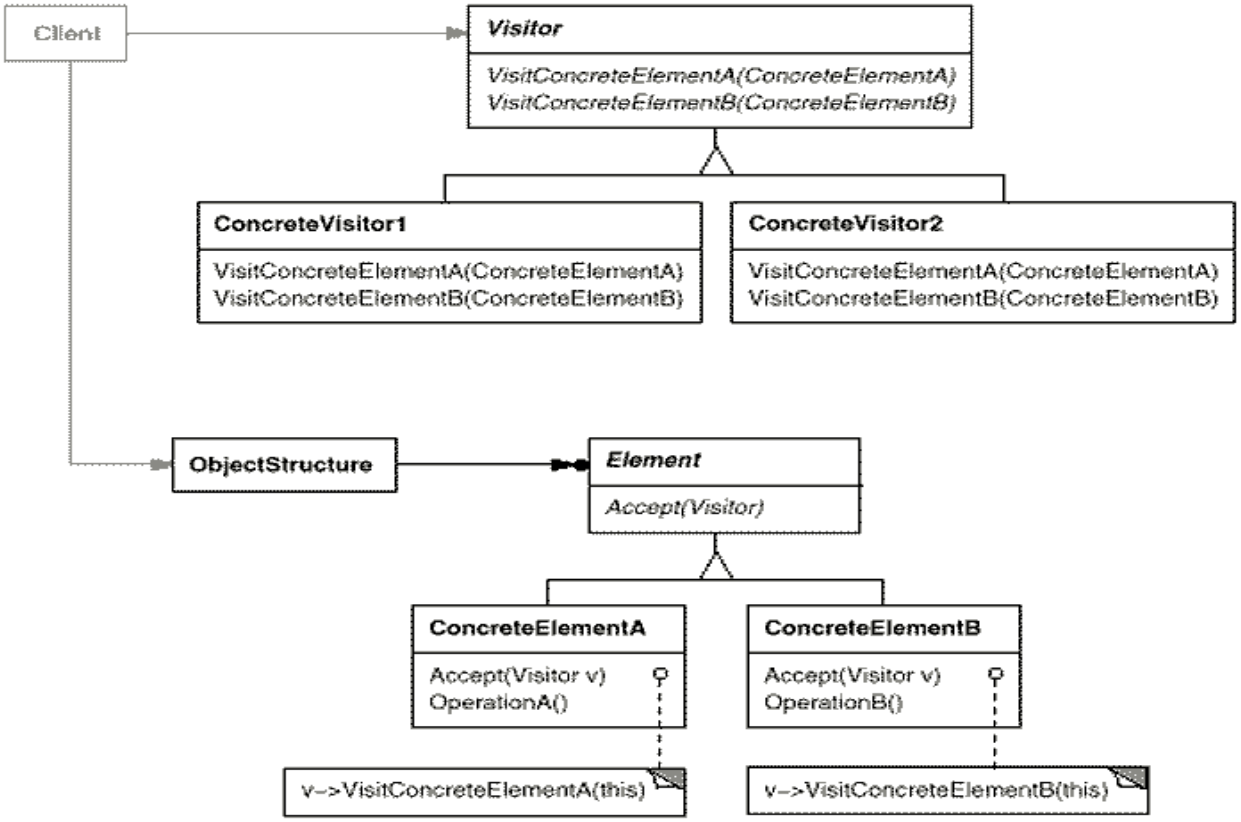
ساختاری که در اینجا آمده تفاوت خاصی با الگویی که در کتاب هست ندارد. به همین دلیل از فقط یکی از آنها ذکر شده است.

Visitor



شکل ۶٫۱ ساختار الگوی Visitor [۶]

همان‌طور که از شـــکل٦/١ پیداســـت کلاینت (در اینجا Object Structure در نظر گرفته نشـــده و کلاینت همان کار را انجام می‌دهد.) می‌تواند روی Element های مختلف که به‌صـورت Abstract به آن‌ها دید دارد (به کلاس‌های سـطح پایین دید مسـتقیم ندارد و از طریق اینترفیس آن با آن‌ها کار می‌کند) با اسـتفاده از Visitor الگوریتم‌های مختلف را اجرا کند یعنی در واقع الگوریتم‌ها از آبجکت‌هایی که قرار هسـت روی آن‌ها اجرا شـود جدا شـده اسـت. Visitor ها در واقع چندین ورژن از یک رفتار را می‌سـازند که برای Element ها تدارک دیده شده است. تمام کلاس‌های Element باید متد Accept را برای پاسخگویی Call و انجام رفتار مربوطه را Override کنند. همان‌طور که از شـــکل پیداســـت یک Element فقط با اینترفیس Visitor کار می‌کند. اما Visitor به حالت داخلی المان دید مستقیم دارد.



شکل ٦/٢ ساختار الگوی Visitor کتاب

ساختاری که در کتاب قابل‌مشاهده است همان‌طور که دکتر رامسین اشاره کردند, به‌صورت کامل صحیح نیسـت و مثلاً ارتباط بین کلاینت با Concrete Visitor ها به‌صورت Dependency اسـت و کلاینت ارتباطی

با اینترفیس Visitor ندارد. در نهایت پس از تصحیح کردن این ساختار کلاسی به شکل ٦/١ می‌رسیم, هم چنین تنها تفاوتی که بین این ٢ ساختار می‌توان گفت, وجود Object Structure در ساختار کتاب است که تفاوت زیادی نیست و کلاینت می‌تواند به‌صورت مستقیم به اینترفیس Element ها دسترسی داشته باشد.

مقایسه: در الگوی State در واقع آبجکت Context رفتارهای متفاوت را بر اساس State که به آن Delegate شده است اجرا می‌کند این کار از طریق اینترفیس آن و بدون خبر از این که چه چیز را اجرا می‌کند می‌باشد, آبجکت‌های State می‌تواند یک Back Reference به Context داشته باشد تا برای تغییر حالت بعد یا قبل از انجام رفتار استفاده شود و State آبجکت Context را عوض می‌کند. الگوی Strategy نیز مجموعه از الگوریتم‌ها را در کلاس‌های جدا می‌گذارد تا آبجکتی که قرار هست از آن استفاده کند آن‌ها را به‌صورت Delegation داشته باشد و از طریق اینترفیس آن‌ها را اجرا کند. هم چنین الگوی Visitor در واقع این‌گونه است که کلاینت بر روی Element ها Visitor های مختلف را ارسال کرده و متد Accept را با آن‌ها صدا می‌زند. در هر Visitor رفتارهای مختص به هر Element نوشته شده است.

کارایی از منظر حافظه

State – اگر قبل از انجام الگو ساختار توارثی داشته بودیم تعداد حالت‌های مختلف آبجکت‌ها زیاد بود و اگر در یک Context حالت‌های مختلف به‌صورت دستی نوشته شده بود یک کلاس بسیار بزرگ داشتیم اما بعد از انجام این الگو تعداد آبجکت‌های حالت زیاد شده در نتیجه برای Context در هر لحظه یک State در نظر داریم و اگر بعد از اتمام کار State را Dispose کنیم مصرف حافظه کم می‌شود. (٢ آبجکت کوچک و Specific به‌جای ١ آبجکت بزرگ). لازم به ذکر است که ممکن است یک Context چندین State لازم داشته باشد که در این صورت نسبت به حالت Before تعداد آبجکت‌های درون سیستم افزایش میابد.

Strategy - تعداد آبجکت‌های Strategy چون برای هر متد یک آبجکت Strategy داریم می‌تواند باعث افزایش تعداد آبجکت‌های درون حافظه شود.

Visitor – در حالت قبل از انجام الگو فقط Element ها را داریم اما بعد از اعمال الگو تعداد آبجکت‌ها به مقدار تعداد Visitor هایی که در حافظه است بیشتر می‌شود.

مقایسه: در الگوی State اگر برنامه‌نویس به‌صورت درست از آن استفاده کند مصرف حافظه کم شده (یعنی در هر لحظه یک State با یک Context داشته باشیم) ازاین‌جهت کم‌شدن را عنوان می‌کنیم که برای مثال فرض کنید رفتارهای مختلف نیاز به Import های سنگین دارد, حال این رفتارها هم زمان با هم در حافظه نیستند. لازم به ذکر است که ممکن است یک Context چندین State لازم داشته باشد که در این صورت نسبت به حالت Before تعداد آبجکت‌های درون سیستم افزایش میابد. الگوی Strategy باعث افزایش مصرف حافظه به دلیل مذکور در بالا می‌شود. الگوی Visitor به دلیل ذکر شده در بالا مصرف حافظه را افزایش می‌دهد.

موارد کاربرد

State

۱- وقتی می‌دانیم یک آبجکت بسته به حالت داخلی‌اش رفتارهای متفاوت دارد و این رفتارها با ساختار توارثی جایگشت زیادی دارد.
۲- رفتار لازم است که در زمان اجرا عوض شود.
۳- آبجکت بسته به مقدار بعضی از فیلدهایش عبارات شرطی طولانی و بزرگ دارد (برای تعیین رفتار).

Strategy

۱- وقتی انواع مختلفی از یـک الگوریتم برای یـک آبجکـت داریم و می‌خواهیم در زمـان اجرا این الگوریتم‌ها تغییر کنند.

۲- وقتی کلاس‌هایی داریم که تقریباً خیلی مشـــابه هســـتند اما در اجرای بعضـــی از رفتارها تفاوت دارند.

۳- وقتی عبارات شـرطی طولانی برای اجرای نسـخه‌های متفاوت از یک الگوریتم یا رفتارهای مختلف داریم.

٤- برای ایزوله کردن بیزینس لاجیک یا عملیات هایی رو داده که نمی‌خواهیم Client از پیاده‌ســـازی آن‌ها باخبر باشد.

Visitor

۱- وقتی می‌خواهیم بر روی یک Object Structure پیچیده ورژن خاصـی از یک الگوریتم را اجرا کنیم ولی بر روی هر نود این ساختار تفاوت‌هایی برای اجرای این الگوریتم دیده می‌شود.

۲- وقتی ساختار آبجکت‌ها زیاد تغییر نمی‌کند ولی لازم هست عملیات های جدید بر روی آن‌ها تعریف شود.

۳- وقتی عملیات هایی که در المان‌ها اســـت به یکدیگر مربوط نیســـتند و نمی‌خواهیم کلاس‌های Element را با آن‌ها شلوغ کنیم و cohesion را پایین بیاوریم.

مقایسـه: می‌توان گفت که دو الگوی Strategy و State در واقع رفتارهای متفاوت را بر روی آبجکت زمینه, در زمان اجرا انجام می‌دهند و از این بابت شـــبیه هم هســـتند. اما Visitor در واقع الگوریتم‌ها را از آبجکت‌هایی که قرار هست روی آن اجرا شود جدا می‌کند.

الگوهای مرتبط

State

۱- State را می‌توان توســـعه‌یافتهٔ Strategy در نظر گرفت, ازآنجایی‌که هر دو عملیات و رفتار را به آبجکت‌های خارجی Delegate می‌کنند, اما در Strategy این رفتارها کاملاً مستقل از هم هستند اما در State می‌توانند از یک حالت به حالت دیگر بروند (مثلاً بعد از انجام یک عملیات درون یک State)

۲- این الگو همانند Bridge یا Strategy در واقع رفتار را به آبجکت‌های دیگر Delegate می‌کند.

Strategy

۱- یکی از موارد مرتبط در اینجا (الگوی Command) به آن اشاره شده است.

۲- می‌توان Decorator ظاهر یک آبجکت را تغییر می‌دهد, Strategy درون آن را (رفتار و عملیات).

۳- یکی از الگوهای مرتبط نیز State هست که در بالا به آن اشاره شد.

٤- یکی از الگوهای مرتبط نیز State و Bridge هست که در بالا به آن اشاره شد.

Visitor

۱- این الگو می‌تواند نســخه قوی تر شــدهٔ Command باشــد. آبجکت‌های آن می‌تواند عملیات های مختلف را روی کلاس‌های مختلف اجرا کند.

۲- از Visitor می‌توان استفاده کرد تا عملیات را روی یک درخت Composite اجرا کرد.

۳- می‌توان از Iterator اســتفاده کرد تا یک ســاختار آبجکتی پیچیده را پیمایش کرد و روی المان‌های آن حتی اگر کلاس‌های متفاوت هم دارند عملیات اجرا کرد.

مقایسه: موارد مرتبط ذکر شد و واضح است که State و Strategy شباهت بسیار زیادی دارند.

مزایا معایب

State

۱- رفتارهای جدید و مختلف را می‌توان با زیر کلاس‌های State پیاده‌سازی کرد. این مزیت است.

۲- در زمان اجرا می‌توان حالت را تغییر داد. این مزیت است.

۳- رفتارهای وابسته به حالت Localize می‌شود. این مزیت است.

٤- تغییر حالت‌ها به‌صورت صریح است (در مقابل Switch-case وابسته به یک فیلد). این مزیت است.

٥- آبجکت‌های State می‌تواند به‌صورت اشتراکی استفاده شود. این مزیت است.

Strategy

۱- خانواده‌ای از یک الگوریتم می‌توان تولید کرد. این مزیت است.

۲- جایگزینی برای ساختار توارثی (subclassing). این مزیت است.

۳- پیکربند می‌توان پیاده‌سازی‌های مختلف از یک متد را در زمان اجرا عوض کند. این مزیت است.

٤- جلوگیری از عبارات شرطی. این مزیت است.

٥- تعداد آبجکت‌ها زیاد می‌شود. این عیب است.

٦- چون در اینترفیس سطح بالای Strategy به‌صورت اجتماع تمام استراتژی‌ها است ممکن است بعضی از Context ها آن را لازم نداشته باشد و به‌این‌ترتیب Overhead زیاد می‌شود, هم چنین به خاطر واسپاری و هم چنین سطحی از Indirection سربار زیاد می‌شود. این عیب است.

۷- پیکربند یا کلاینت باید انواع استراتژی‌ها را بشناسد. این عیب است. (تقریباً برای همه پیکربند ها صادق است زیرا در مورد آن‌ها DIP برقرار نیست.)

Visitor

۱- اضافه‌کردن عملیات جدید در Visitor ها ساده است. این مزیت است.

۲- عملیات های مرتبط را جمع و جدا از عملیات نامرتبط می‌کند. این مزیت است.

۳- اضافه‌کردن Element جدید سخت است چون باید اینترفیس و تمام کلاس‌های Visitor آپدیت شوند. این عیب است.

٤- همان‌طور که قبلاً اشـاره شـد بسـته‌بندی یا کپسـوله‌سـازی نقض می‌شـود. چون باید از داخل Visitor به داخل Element ها دسترسی پیدا کرد و این عیب است.

مقایسه: هر دو الگوی State و Strategy با واسپاری الگوریتم‌ها به آبجکت‌های خارجی باعث تغییر حالت در زمان اجرا می‌شـود. همچنین Visitor باعث تولید خانواده‌ای از الگوریتم‌ها که مسـتقل اسـت از آبجکت‌هایی که قرار است روی آن اجرا شوند, می‌شود. بقیه مزایا و معایب در بالا توضیح داده شد.

شیء جعلی

State – کلاس و آبجکت‌های State همه تصنعی هستند و در قلمرو مسئله نیستند.

Strategy - آبجکت و کلاس‌های Strategy در قلمرو مسئله نیست و برای تحقق الگو تولید شده و جعلی است.

Visitor – تمام کلاس‌ها و آبجکت‌های Visitor در قلمرو مسئله نیستند و مصنوعی هستند.

مقایسه: هر ۳ شیء جعلی تولید کرده‌اند.

جداسازی دغدغه‌ها

State – این الگوریتم رفتارهای مرتبط با حالت را در کلاس‌های State قرار داده و کلاس Context در واقع نمی‌داند که قرار است کدام حالت را اجرا کند.

Strategy - این الگو, الگوریتم عملیاتی که قرار است انجام شود را از آبجکت Context جدا کرده است و هم چنین از ایجاد عبارات شرطی جلوگیری کرده و وظیفه مشخص‌کردن الگوریتم مشخصاً به پیکربند محول شده است.

Visitor – به‌وضوح الگوریتم‌هایی که قرار بوده روی Element ها انجام شود را از آنها جدا کرده و در کلاس‌های جدا قرار داده و باعث شده جداسازی دغدغه‌ها به‌خوبی صورت گیرد.

مقایسه: هر ۳ الگو به‌خوبی این کار را انجام داده‌اند.

پیاده‌سازی

State – ابتدا عملیات هایی که در کلاس Context هست و رفتارهای مرتبط با حالت دارد را انتخاب کرده و این عملیات ها را از آن حذف کرده و اینترفیس برای آن تعریف می‌کنیم. یا از ساختارهای توارثی این حالت‌ها را استخراج می‌کنیم. سپس اینترفیس State را تعریف کرده و متناسب با هر حالت یک Concrete Stateمی‌سازدیم, سپس فیلد State را در کلاس زمینه به‌صورت Aggregate اضافه می‌کنیم و از طریق آن, عملیات های مذکور را اجرا می‌کنیم. برای تغییرات حالت‌ها به یکدیگر به‌اصطلاح یک Back reference به آبجکت Context در کلاس‌های State قرار می‌دهیم. حال پیکربند می‌تواند با واسپاری نوع مناسب State به کلاس Context به‌خوبی از این الگو استفاده کند. دقت داریم که Context فقط از اینترفیس State می‌تواند استفاده کند.

Strategy - در کلاس زمینه دنبال عبارات شرطی بزرگ که اجرای الگوریتم را با ورژن‌های مختلفی از آن یا کانفیگ خاصی از آن مشخص می‌کند را پیدا می‌کنیم, سپس اینترفیس انواع مختلف از این الگوریتم را می‌سازیم و به‌تبع آن برای هر نوعی از این الگوریتم کلاس‌های Concrete برای Strategy می‌نویسیم.

بعد از آن یک فیلد برای Strategy در کلاس Context تعریف می‌کنیم تا آن را به‌صورت Aggregated دریافت کند. حال پیکربند می‌تواند به‌درستی Strategy های مختلف را در اختیار Context قرار دهد تا از آن استفاده کند. دقت داریم که Context فقط از اینترفیس Strategy می‌تواند استفاده کند.

Visitor – اینترفیسی برای Visitor متشکل از متدی برای انجام الگوریتم برای هر Concrete Elementمی‌سازدیم, تمام این متدها المان را به‌عنوان ورودی میگرند. برای Element ها اینترفیسی با متد Accept که Visitor را به‌عنوان پارامتر می‌گیرد می‌سازیم. المان‌ها فقط با اینترفیس Visitor کار می‌کنند اما Visitor باید دید مستقیم و کامل به Concrete Element ها داشته باشد. حال برای هر رفتاری که می‌خواهیم داشته باشیم یک Visitor به‌صورت Concreteمی‌سازدیم. بعد از کلاینت می‌تواند نمونه‌هایی از Visitor ها را به متدها Accept در المان‌ها بفرستد و آن را صدا کند تا عملیات انجام شود.

مقایسه: به نظر می‌توان گفت Strategy و State مانند یکدیگر تقریباً ساده هستند و پیچیدگی خاصی ندارند. به نظر می‌رسد الگوی Visitor پیچیدگی در نحوه پیکربندی و ساختن یا نوشتن ساختار کلی دارد.

بررسی الگوهای **GRASP** برای ۳ الگوی **State, Strategy** و **Visitor**

Information Expert

State – قبل از اجرای الگو, حالت‌های مختلف یا توسط ساختار توارثی یا عبارات شرطی نوشته می‌شد که در اینجا رفتار در کنار داده‌های خود می‌باشد, اما بعد از اجرای الگو رفتارها و حالت‌های مختلف درون کلاس‌های State قرار می‌گیرد, اما داده‌های مربوطه هنوز در کلاس زمینه یا Context هست, این یعنی نقض این الگو.

Strategy - باتوجه‌به این که هر Context داده‌های مربوط به خود را هنگام اجرای یک Strategy به آن ارسال می‌کند (به‌عنوان پارامتر) می‌توان گفت که این الگو رعایت نشده است, زیرا Strategy ها متخصص اجرای الگوریتم هستند اما داده‌ها در Context جامانده‌اند. این یعنی نقض این الگو.

Visitor – در این الگو Information Expert نقض شده است زیرا که الگوریتم‌هایی که در Visitor های مختلف نوشته شده‌اند قبلاً در کنار Element ها بوده (دقیقاً همان جایی که داده‌های مربوطه وجود دارد) این یعنی جداسازی رفتار از داده‌ها و نقض این الگو.

Creator

State – این الگو چون آبجکت Context آبجکت‌های State را به‌صورت Aggregation دارد (اولویت ۲) باید آن‌ها را بسازد اما پیکربند این کار را انجام می‌دهد, این دلیل باعث نقض این الگو است.

Strategy - این الگو چون آبجکت Context آبجکت‌های Strategy را به‌صورت Aggregation دارد (اولویت ۲) باید آن‌ها را بسازد اما پیکربند این کار را انجام می‌دهد, این دلیل باعث نقض این الگو است.

Visitor – ازآنجایی‌که هم Element ها و هم Visitor ها از یکدیگر اســـتفاده می‌کنند (Concrete Element ها از اینترفیس Visitor اســـتفاده می‌کند, Concrete Visitor ها از خود آبجکت Concrete Element ها اســتفاده می‌کنند) پس هرکدام باید به‌صـورت Circular دیگری را Create کنند اما این امکان وجود ندارد و بهتر بود که Element ها Visitor ها را ایجاد کنند. به‌هرحال این وظیفه به پیکربند محول شده و این الگو نقض می‌شود.

Coupling

State – با رجوع به قســمت کاهش وابســتگی برای این الگو متوجه می‌شــویم که تأثیر زیادی برای این الگوی Grasp دارد و Coupling را کم می‌کند.

Strategy - با رجوع به قسـمت کاهش وابسـتگی برای این الگو متوجه می‌شـویم که تأثیر زیادی برای این الگوی Grasp دارد و Coupling را کم می‌کند.

Visitor – با رجوع به قسـمت کاهش وابسـتگی برای این الگو متوجه می‌شــویم از طرف Element ها به Visitor به‌شـدت Coupling پایین هسـت ولی برعکس چون Visitor ها به‌صـورت مسـتقیم به Concrete Element ها دسترسی دارند Coupling بالا است.

Cohesion

State – با رجوع به قســمت افزایش چســبندگی برای این الگو متوجه می‌شــویم که تأثیر زیادی برای این الگوی Grasp دارد و تقریباً Cohesion را به حد خوبی می‌رساند.

Strategy - با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و تقریباً Cohesion را به حد خوبی می‌رساند.

Visitor – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و آن را محقق می‌سازد.

Controller

State – این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Strategy - این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Visitor – این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Polymorphism

State – این الگو با داشتن فوق کلاس برای State ها و اجرای متفاوت از زیر کلاس‌های متفاوت که بر اساس Type رفتار متفاوتی دارند این الگو را رعایت می‌کند.

Strategy - این الگو با داشتن اینترفیس مشترک برای آبجکت اصلی و فرعی و بسته به نوع Strategy دستورات متفاوت را با اینترفیس یکسان و دارای رفتار متفاوت را اجرا کرد (با اینترفیس یکسان).

Visitor – برای استفاده از این الگو در واقع از Element ها متد Accept را صدا می‌زنیم که همهٔ Element ها آن را پیاده‌سازی کرده‌اند, سپس هر Visitor متد برای ویزیت کردن آن Element را اجرا می‌کند (بعد از

این که در متد Accept, این متد در Visitor صدا زده شد). پس درواقع کلاینت می‌تواند انواع Element ها را استفاده کرده (فقط با استفاده از اینترفیس آنها) و Visitor های مختلف را به آن‌ها پاس دهد. پس چندریختی رعایت شده است زیرا که انواع Visitor ها یک اینترفیس سطح بالا را محقق کرده‌اند و Element ها با این اینترفیس کار می‌کنند.

Indirection

State – همان‌طور که در ساختار دیده شد ارتباط بین Context و رفتارهایی که وابسته به حالت بود از بین رفته و حالت‌ها در کلاس‌های بیرونی State قرار گرفته, که ارتباط بین آن‌ها غیرمستقیم شده است. ارتباط فقط از طریق اینترفیس است. پس این الگو رعایت شده است.

Strategy - همان‌طور که در ساختار دیده شد ارتباط بین Context و الگوریتم‌هایی که تفاوت‌هایی داشته‌اند و با عبارات شرطی مشخص می‌شدند از بین رفته و الگوریتم‌ها در کلاس‌های بیرونی Strategy قرار گرفته, که ارتباط بین آن‌ها غیرمستقیم شده است. ارتباط فقط از طریق اینترفیس است. پس این الگو رعایت شده است.

Visitor – کلاینت یا پیکربند (در اینجا می‌تواند Object Structure هم باشد اما ما فرض می‌کنیم فقط کلاینت وظیفه آن را بر عهده گرفته است) فقط با استفاده از اینترفیس Element که متد Accept دارد کار می‌کند که قبلاً با خود المان‌ها کار می‌کرد در اینجا Indirection برقرار است, اما چون کلاس‌های سطح پایین Visitor با کلاس‌های سطح پایین Element کار می‌کند, اینجا Indirection برقرار نیست. اما اگر Object Structure را هم اینجا در نظر بگیریم می‌توان گفت ارتباط بین کلاینت و Element ها غیرمستقیم است. هم چنین می‌توان گفت ارتباط کلاینت با Element ها برای انجام عملیات های خود, غیرمستقیم و از طریق Visitor ها انجام می‌پذیرد.

Pure Fabrication

State – تمام کلاس‌های State جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Strategy - تمام کلاس‌های Strategy نیز جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Visitor – کلاس‌های Visitor برای راحتی حل مسئله تولید شده‌اند.

Protected Variations

State – همان‌طور که در بحث تغییرپذیری گفته شد ارتباط بین Context و کلاس‌های State در سطح بالا و از طریق اینترفیس آن‌ها می‌باشد, به‌این‌ترتیب تغییرات به کلاس Context منتشر نمی‌شود زیرا فقط از طریق اینترفیس صدا زده می‌شود. می‌توان درون کلاس‌های State در واقع Back reference به آبجکت Contextداشت) برای تغییر State ها به یکدیگر لازم می‌شود (که این خود دید سطح پایین است و باعث انتشار تغییرات از Context به State ها می‌شود .

Strategy - همان‌طور که در بحث تغییرپذیری گفته شد ارتباط بین آبجکت Context و خانواده‌ای از الگوریتم‌ها که در Strategy ها تعریف شده به‌صورت غیرمستقیم و فقط از طریق اینترفیس آن‌هاست و تغییرات به کلاس زمینه منتقل نمی‌شود. فقط پیکربند است که باید از انواع مختلف Strategy باخبر باشد.

Visitor – همان‌طور که در بحث تغییرپذیری گفته شد در این الگو برای تولید Element های جدید باید تمام اینترفیس‌ها و کلاس‌های سطح پایین Visitor آپدیت شوند و عملیات های مخصوص این Element نوشته شود. اما برای تولید رفتار جدید یا همان Visitor جدید در واقع با نوشتن یک Concrete Visitor می‌توان این عمل را انجام داد و این تغییر به Element ها منتشر نمی‌شود زیرا از طریق اینترفیس با آن‌ها در ارتباط

اســت, اما از طرف دیگر تعریف المان جدید تغییر منتشــر شــونده دارد هم به دلیل ذکر شــده در بالا و هم به دلیل اینکه Concrete visitor ها باید به حالت و داخلی المان‌ها دســترســی داشــته باشــد و ارتباط آن‌ها از طریق اینترفیس نیست.

مقایسه الگوهای **Abstract Factory, Iterator** و **Builder**

دسته - نوع

Abstract Factory - این الگو در درسته الگوهای آفرینشی می‌باشد. الگوهای آفرینشی بیشتر دغدغهٔ مکانیزم‌های تولید آبجکت‌ها را دارد تا انعطاف‌پذیری و Reuse افزایش یابد. این الگوها دو ایده اصلی دارند اول این که چطور آبجکت‌ها ساخته شوند و دیگری اینکه دانش را در مورد اینکه سیستم از چه Concrete کلاس‌هایی استفاده می‌کند پنهان کند.

Iterator – این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

Builder - این الگو در درسته الگوهای آفرینشی می‌باشد. الگوهای آفرینشی بیشتر دغدغهٔ مکانیزم‌های تولید آبجکت‌ها را دارد تا انعطاف‌پذیری و reuse افزایش یابد. این الگوها دو ایده اصلی دارند اول این که چطور آبجکت‌ها ساخته شوند و دیگری اینکه دانش را در مورد اینکه سیستم از چه Concrete کلاس‌هایی استفاده می‌کند پنهان کند.

مقایسه: دو الگوی Builder و Abstract Factory هر دو جزو الگوهای آفرینشی هستند که دغدغه‌های یکسانی دارد اما Iterator این‌گونه نیست و جزو الگوهای رفتاری می‌باشد.

هدف

Abstract Factory - این الگو برای تولید خانواده‌ای از آبجکت‌ها اســـتفاده می‌شـــود تا از عنوان کردن یا مشخص‌کردن کلاس Concrete آن جلوگیری شود (از اینترفیس آن‌ها استفاده شود).

Iterator – این الگو در واقع این امکان را فراهم می‌آورد تا بتوان بر روی المان‌های یک ســـاختار داده‌ای پیچیده پیمایش انجام داد بدون آن که جزئیات داخلی آن ساختار را بدانیم.

Builder – این الگو به ما کمک می‌کند تا آبجکت‌های پیچیده و بزرگ را قدم‌به‌قدم با کدهای یکســـان بســـازیم. یعنی با یک کدی که نوشـــته شـــده بســـته به Step هایی که بر روی یک آبجکت خام انجام می‌دهیم نمایش‌هایی متفاوتی خواهد داشـت. فرایند سـاخت از جزئیات جدا شـده اسـت. یعنی الگوریتم ســـاخت کلی را تعریف کنیم و آن را جدا بگیریم و ســـازنده را تحت هدایت یک کارگردان بگذاریم که فقط دستور می‌دهد.

مقایســـه: الگوی Abstract Factory و Builder در واقع هر دو بر روی آنکه یک یا چند آبجکت را چگونه بســـازند تمرکز دارند, مثلاً Builder با ســـاختن آبجکت‌های پیچیده با قدم‌های متفاوت اما کد یکســـان, و الگوی Abstract Factory می‌تواند با ســاختن خانواده‌ای از آبجکت‌ها به ما کمک می‌کند. بااین‌حال الگوی Iterator بر روی پیمایش المان‌های هر ساختار تمرکز دارد.

حوزه

Abstract Factory - این الگو در حوزهٔ شـیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Iterator – این الگو در حوزهٔ شـیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Builder - این الگو در حوزهٔ شـیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

مقایسه: هر سه الگو در زمان اجرا و از طریق delegation تحقق می‌یابند.

کاهش وابستگی

Abstract Factory - ازآنجایی‌که وابستگی بین کلاینت و محصولات از بین رفته و فقط از طریق کلاس‌های Factory محصولات را می‌سازیم آن هم از طریق اینترفیس آن, پس وابستگی این ۲ بسیار پایین هست, اما وابستگی بین Factory ها و محصولات بسیار بالاست زیرا کلاس‌های Factory باید کلاس‌های سطح پایین محصولات را ببیند و با آن کار کند. همچنین اضافه یا کم‌کردن محصولات جدید به اینترفیس و تمام Factory ها منتشر می‌شود. ولی اضافه و کم‌کردن یک Product در یک خانواده خاص به کلاینت منتشر نمی‌شود چون کلاینت فقط با اینترفیس آن کار می‌کند.

Iterator – وابستگی بین اشیا مرکب و کلاینت از بین رفته زیرا کلاینت فقط با پیمایشگر کار می‌کند اما از طرفی وابستگی بین اشیا و پیمایشگر بسیار بالاست. قبل از آن وابستگی بین اشیا و عمل پیمایش کردن آن بود که به عهده کلاینت بود که بسیار بالا بود.

Builder – از آنجایی ارتباط کلاینت با آبجکت‌های کوچک و بزرگ از بین رفته و فقط با Director کار می‌کند, این وابستگی بالایی نیست زیرا فقط یک نمونه از Builderمی‌سازد و در اختیار Director قرار می‌دهد, هم چنین خود Director با Builder از طریق اینترفیس آن کار می‌کند (ولی آن را به‌صورت Aggregate شده داخل خود دارد) باز هم وابستگی بالایی ندارد, هم چنین کلاینت نیز باید انواع کلاس‌های Builder را برای پیکربندی‌های مختلف بشناسد, اما وابستگی کلاس Builder و محصول بالاست, زیرا با تغییر در محصول این تغییر به تمام Builder ها منتشر می‌شود. هم چنین کلاینت باید متد Get Result را از کلاس‌های سطح پایین Concrete Builder را اجرا کند پس باید دید کامل به Concrete Builder ها دارد.

مقایسه: برای دو الگوی Builder و Abstract Factory وابستگی بین محصولات و کلاینت از بین رفته اما در هر دو بین Builder و محصول و هم چنین Abstract Factory و محصول بالاست زیرا هر دو با کلاس‌های سطح پایین محصولات کار می‌کند. هم چنین در الگوی Builder متد Get Result در سطح بالا و اینترفیس تعریف نشده و فقط کلاینت و آن هم با دید مستقیمی که به Concrete Builder ها دارد این کار را انجام دهد. هم چنین در Iterator وظیفه پیمایش به Iterator و اشیا مرکب به Aggregate Object داده شده است و این باعث می‌شود تا کلاینت فقط با پیمایشگر کار کند (آن هم از طریق اینترفیس) و با داخل آن کار ندارد.

افزایش چسبندگی

Abstract Factory - ازآنجایی‌که دیگر کلاینت درگیر ساخت محصولات نیست و این کار به Factory ها که متخصص این کار هستند محول شده از Cohesion بالایی برخوردار است.

Iterator – اگر فرض کنیم که اشیا شیء مرکب هرکدام به‌صورت Cohesive و تک کاره هستند, بعد از اعمال الگو چون در واقع یک پیمایشگر برای هر شیء مرکب داریم که فقط کار پیمایش آن را انجام می‌دهد پس چسبندگی بسیار خوبی برخوردار است. زیرا وظیفه پیمایش از کلاینت جدا شده و به Iterator داده شده و باعث می‌شود که Cohesive باشد.

Builder – با اعمال این الگو در واقع چون عملیات های مربوط به Step های ساخت به‌صورت مستقیم به Client محول نشده و Director این کار را انجام می‌دهد, از Cohesion خوبی برخوردار است. هم چنین خود کلاس Builder فقط مسئول ساخت ریز قطعه‌ها می‌باشد. هم چنین کارگردان مخصوص هدایت کردن و دستور دادن به Builder هست به‌صورت تک کاره و یکپارچه است.

مقایسه: هر سه الگو هرکدام به صورتی Cohesion بالایی دارند یا باعث افزایش آن شده‌اند, درست هست که هرکدام مقصود متفاوتی دارند اما مثلاً Abstract Factory با جداسازی مسئولیت ساخت از پیکربند, Iterator با جدا کردن وظیفهٔ پیمایش از کلاینت و هم چنین Builder با جدا کردن وظیفه ساخت ریز قطعه‌ها و آبجکت‌ها از کلاینت و محول کردن آن به‌Builder و دستور دادن از طریق Director به آن, باعث کاهش چندکارگی و افزایش Cohesion شده‌اند.

OCP

Abstract Factory - تغییرات (اضافه و کم‌شدن) در کلاس‌های Product در واقع به Factory ها منتشر می‌شود چون باید در اینترفیس Abstract Factory تعریف شود و پیاده‌سازی آن انجام شود اما تغییرات در Factory به Product به پیکربند یا کلاینت منتشر نمی‌شود چون ارتباط از طریق اینترفیس سطح بالای آن است. پس در جاهایی برقرار و در جاهای دیگر رعایت نشده است. ولی اضافه و کم‌کردن یک Product در یک خانواده خاص به کلاینت منتشر نمی‌شود چون کلاینت فقط با اینترفیس آن کار می‌کند.

Iterator – ازآنجایی‌که ارتباط بین کلاینت و پیمایشگر از طریق اینترفیس آن می‌باشد تغییرات راحت صورت می‌گیرد و باعث تغییر کدهای موجود نمی‌شود, اما تغییرات در شیء مرکب به پیمایشگر و برعکس منتشر می‌شود چون ارتباط آن‌ها در سطح پایین است. پس این اصل رعایت نشده.

Builder – ازآنجایی‌که ارتباط بین Director و Builder از طریق اینترفیس سطح بالای Builder هست تغییرات از Builder به Director منتشر نمی‌شود و برعکس پس این اصل برقرار است, هم چنین تغییرات به کلاینت منتشر نمی‌شود یعنی چون کلاینت با Director کار می‌کند و Director نیز از طریق اینترفیس با Builder کار می‌کند پس این اصل رعایت شده و برقرار است. اما از طرفی چون کلاینت برای پیکربندی و هم چنین Get Result لازم دارد تا انواع کلاس‌های Concrete Builder را ببیند و ارتباط آن‌ها از طریق

اینترفیس نیست و تغییرات به کلاینت منتشر خواهد شد (این وابستگی به‌صورت Dependency هست یعنی مانا نیست اما باز هم باید انواع Concrete Builder ها را بشناسد).

مقایسه: در الگوی Builder تا حد خوبی رعایت شده (به‌غیراز پیکربند) اما در Abstract Factory در کم یا اضافه‌شدن محصول جدید این تغییر باعث تغییر کدهای موجود می‌شود اما باز هم در حد خوبی رعایت شده ولی در Iterator تغییرات دو ساختار توارثی موازی به‌شدت باعث تغییر کدهای موجود قبلی می‌شود.

LSP

Abstract Factory - کاملاً برقرار است چون زیر کلاس‌های Abstract Factory به‌راحتی می‌تواند جایگزین Super class آن شوند و کلاینت به‌راحتی از آن‌ها استفاده می‌کند.

Iterator – کاملاً برقرار است چون زیر کلاس‌های Iterator می‌توانند به‌جای پدر خود استفاده شوند بدون این که تغییری در رفتار یا کد ایجاد شود.

Builder – برای Builder ها کاملاً برقرار است زیرا کلاینت فقط با Director کار می‌کند و خود Director فقط از اینترفیس Builder استفاده می‌کند و با تغییر کلاس Concrete Builder مشکلی برای او ایجاد نمی‌شود.

مقایسه: هر ۳ الگو این اصل را برقرار می‌سازند.

DIP

Abstract Factory - همان‌طور که قبل‌تر اشـــاره شـــد ارتباط بین کلاینت و Factory ها فقط از طریق اینترفیس آن‌ها اســـت پس این اصـــل برقرار شـــده اما تغییرات (کم یا زیادشـــدن) محصـــولات این تغییر به اینترفیس و زیر کلاس‌های Factory منتشـر می‌شـود زیرا به کلاس‌های Concrete وابسـته شـده اسـت در نتیجه این اصـل برای Factory برقرار نیسـت اما بقیه تغییرات به کلاینت منتشـر نمی‌شـود زیرا DIP برقرار اسـت.

Iterator – ازآنجایی‌که ارتباط بین کلاینت و پیمایشـگر فقط از طریق اینترفیس آن هسـت و از پیاده‌سازی آن اطلاعی ندارد پس آن را رعایت کرده اما ارتباط بین آبجکت مرکب و پیمایشـــگر به‌شـــدت در ســـطح پایین و Concrete اسـت و این اصل نقض شـده اسـت.

Builder – چون کلاینت در واقع با Director کار می‌کند و او با اینترفیس فقط با Builder کار می‌کند این اصل رعایت شـده, مگر بخواهیم به‌اصطلاح Part های جدید به اینترفیس Builder اضافه کنیم که این تغییر باید در اینترفیس هم انجام شـــود و پیاده‌ســـازی خود لازم دارد. پس در این اصـــل رعایت شـــده اســـت. اما ازآنجایی‌که کلاینت برای Get Result باید زیر کلاس‌های Builder را ببیند و این کار را از طریق اینترفیس و یا Director انجام نمی‌دهد باعث نقض DIP می‌شـــود اما رعایت DIP در جاهای دیگر که در بالا ذکر شـــد بیشتر مدنظر اسـت.

مقایســه: در دو الگوی اول ارتباط بین کلاینت و آبجکت اصـلی که به ترتیب Factory و Iterator هسـت در سـطح بالا و اینترفیس اسـت و این اصـل برقرار اسـت, اما ارتباط بین پیمایشـگر و شـیء مرکب در الگوی Iterator و هم چنین کم یا زیادشـــدن Product در Abstract Factory و هم چنین Part ها در Builder باعث تغییر در اینترفیس و ... می‌شـــود, پس در این حالات DIP را رعایت نکرده‌اند. اما در Builder مسـئله

دیگری هم وجود دارد که همان‌طور که اشاره شد همان دید مستقیمی است که کلاینت به زیر کلاس‌های Builder دارد.

ISP

Abstract Factory - در این الگو در حد بسیار خوبی این اصل رعایت شده زیرا که برای خانواده‌ای از محصولات یک اینترفیس نوشته شده است و خیلی مسئولیت Specific و تعریف شده‌ای است و این اصل را رعایت کرده.

Iterator – این الگو در واقع اینترفیس‌های پیمایشگر و شیء مرکب از هم جدا شده و به‌خوبی کارهای تقسیم شده و مسئولیت‌ها تخصیص‌یافته است, یعنی این اصل رعایت شده.

Builder – در این الگو برای هر قطعه‌ای که لازم هست به آبجکت اصلی اضافه شود در اینترفیس اصلی Builder تعریف شده و مسئولیت‌ها مشخص و به‌صورت Specific هست.

مقایسه: می‌توان گفت هر ۳ الگو این اصل را تا حد خوبی رعایت کرده است.

CRP

Abstract Factory - در این الگو چون وظیفه پیکربند است که از Factory موردنیاز استفاده کند تا خانواده‌ای از محصولات را بسازد (آن را در اختیار کلاینت قرار دهد) و از ساختار توارثی برای ساخت محصولات جلوگیری کرده و در زمان اجرا می‌تواند از Factory متفاوتی استفاده کند. پس این اصل رعایت شده است اما برای محصولات و Factory ها ساختار توارثی نیاز است.

Iterator – به حد عالی این اصل رعایت شده و به‌جای ساختار توارثی از حالت Delegation استفاده شده است. زیرا وظیفه پیمایش روی یک شیء مرکب به یک آبجکت دیگر به نام پیمایشگر محول شده است و از ساختار توارثی جلوگیری کرده است.

Builder – در این الگو چون در واقع به‌صورت Delegation که همان رابطه Association بین Director و Builder هست داریم از ساختار توارثی جلوگیری کرده و این اصل را رعایت کرده است.

مقایسه: هر ۳ الگو به‌خوبی این اصل را رعایت می‌کنند, باتوجه‌به اینکه بعضی از جاها مثلاً در Abstract Factory نیاز هست که ساختار توارثی محصولات حفظ شود ولی وظیفه اصلی الگو چیز دیگری است.

PLK

Abstract Factory - این اصل توسط این الگو رعایت شده زیرا کلاینت فقط دستور ساخت یک آبجکت از یک خانواده را به Factory مربوطه ارسال می‌کند و آن را دریافت می‌کند که این زنجیره دید ترایا نیست و هدف این الگو ساخت این آبجکت است.

Iterator – می‌توان گفت که هم خیلی خوب این اصل را رعایت کرده هم نکرده, زیرا پیمایشگر خود آبجکت مرکب را در اختیار کلاینت قرار نمی‌دهد و فقط به درخواست‌های کلاینت پاسخ می‌دهد اما از طرفی پیمایشگر دید کامل به آبجکت مرکب دارد که این اجتناب‌ناپذیر است.

Builder – در این الگو به‌خوبی این اصل برقرار است زیرا همان‌طور که مشخص است ارتباط بین Director و Builder از طریق اینترفیس آن هست و آبجکتی پاس داده نمی‌شود, از سوی دیگر برای متد Get Result که از طریق کلاینت و آن هم از کلاس‌های Concrete Builder هست انجام می‌شود محصول برگردانده می‌شود که این هم نقض PLK نیست.

مقایسه: هر ۳ الگو به‌خوبی این اصل را رعایت کرده مگر در الگوی Iterator که پیمایشگر دید کامل به آبجکت مرکب دارد و این اجتناب‌ناپذیر است.

بسته‌بندی

Abstract Factory - در این الگو چون Factory ها متخصص ساخت خانواده‌ای از محصولات هستند و داده رفتار آن‌ها در کنار یکدیگر است و این شاخص نقض نشده است. ولی در تمام کلاس‌های Concrete Factory با کلاس‌های سطح پایین محصولات Concrete Product ها کار شده, اما این کار فقط برای نمونه‌سازی است, نمی‌توان آن را نقض بسته‌بندی دانست.

Iterator – در این الگو علاوه بر این که دید کلاینت نسبت به پیمایشگر و شیء مرکب فقط از طریق اینترفیس و بدون اطلاع داشتن از اطلاعات داخلی آن است بسته‌بندی به‌خوبی رعایت شده, اما چون هر پیمایشگر به طور کامل به حالت داخلی یک شیء مرکب دید دارد بسته‌بندی نقض شده است.

Builder – در این الگو چون Director فقط از طریق اینترفیس یک Builder با آن کار می‌کند و اطلاعی از خالت داخلی آن ندارد کپسوله‌سازی به‌خوبی رعایت شده است. ولی پیکربند به‌صورت مستقیم متد Get Result را از Concrete Builder ها اجرا می‌کند که این را باتوجه‌به هدفی که الگو دارد می‌توان نقض بسته‌بندی در نظر نگرفت.

مقایسه: الگوی Builder و Abstract Factory هر دو تا حد بسیار بالایی کپسوله‌سازی را رعایت کرده‌اند مگر یکجا آن هم فقط در نمونه‌سازی از محصولات, اما Iterator به خاطر دید مستقیم به حالت داخلی آبجکت مرکب کپسوله‌سازی را نقض کرده است, اما این نقض برای رسیدن به هدف الگو یعنی جداکردن پیمایش و آبجکت مرکب, لازم است.

انعطاف‌پذیری

Abstract Factory - در این الگو چون کلاینت در واقع از اینترفیس سطح بالای هر Factory استفاده می‌کند به‌راحتی می‌تواند نوع آن را عوض کند و متوجه تغییری نشود, هم چنین تعریف خانواده جدیدی از محصولات راحت است ولی تعریف یک محصول جدید تغییرات منتشر شونده دارد که در بخش مربوط به تغییرپذیری توضیح داده شده است, در کل انعطاف‌پذیری خوبی دارد.

Iterator – در این الگو نیز چون کلاینت در واقع با اینترفیس سطح بالای یک پیمایشگر کار می‌کند و نمی‌داند که کدام نمونه از Concrete Iterator هست و هم چنین می‌توان از چندین پیمایشگر هم زمان روی یک آبجکت مرکب استفاده کرد. ولی تغییرات در پیمایشگر و آبجکت مرکب به یکدیگر منتشر می‌شود, در کل انعطاف‌پذیری خوبی برقرار است.

Builder – این الگو نیز چون Director به‌صورت Aggregate دارای یک Builder هست و از طریق اینترفیس با آن کار می‌کند, می‌توان گفت که به‌راحتی در زمان اجرا می‌توان Builder را عوض کرد و Director متوجه آن نمی‌شود, پس در کل انعطاف‌پذیری بالایی دارد.

مقایسه: همان‌طور که در بالا اشاره شد هر سه تا حد خوبی انعطاف‌پذیری دارند اما در شرایطی مثلاً برای Abstract Factory و تعریف محصول جدید, یا برای Iterator و تغییرات در بخش پیمایشگر یا آبجکت مرکب دارای تغییرات منتشر شونده هستیم که در پایین به آن اشاره شده است.

تغییرپذیری/ انتشار تغییرات

Abstract Factory - در این الگو به‌راحتی می‌توان خانواده جدیدی تولید کرد و تغییرات آن به‌جایی منتشر نمی‌شود (با تعریف یک Concrete Factory). اما برای اضافه‌کردن یک محصول جدید این تغییر به تمام کلاس‌های Factory و هم چنین پیکربند منتشر می‌شود.

Iterator – در این الگو نیز به دلیل جدا شدن پیمایشگر و آبجکت مرکب از دید کلاینت و ارتباط کلاینت با آن‌ها فقط از اینترفیس سطح بالای پیمایشگر هست که تغییرات منتشر نمی‌شود و به‌راحتی تغییرات انجام می‌پذیرد, اما در ۲ ساختار توارثی موازی پیمایشگر و آبجکت مرکب تمام تغییرات به یکدیگر منتشر می‌شود زیرا در سطح Concrete با یکدیگر کار می‌کند.

Builder – در این الگو همان‌طور که قبلاً اشاره شد چون کلاینت با Director کار می‌کند و از طرفی Director نیز با اینترفیس سطح بالای Builder کار می‌کند تغییرات در Builder به آن منتشر نمی‌شود, ولی از طرفی اضافه‌کردن یک Part جدید به یک محصول, در واقع باعث اضافه‌شدن متدهای Add Part به اینترفیس و کلاس‌های Concrete Builder خواهد شد. هم چنین پیکربند دید مستقیم به کلاس‌های سطح پایین Builder دارد و تغییراتی مثل اضافه و کم شدن یا تغییرات امضای متدها به او منتشر خواهد شد.

مقایسه: هر ۳ الگو تقریباً تغییرپذیری در حد خوبی است, تغییرات به کلاینت منتشر نمی‌شود, اما مثلاً Abstract Factory تغییرات برای اضافه‌شدن یک محصول جدید منتشر می‌شود یا در Iterator تغییرات از پیمایشگر به آبجکت مرکب منتشر می‌شود و برعکس.

پیکربندی

Abstract Factory - پیکربند ثالث لازم اســـت تا بتواند یک نمونه از کلاس‌های Concrete Factory را ساخته و در اختیار کلاینت قرار دهد.

Iterator – یک پیکربند ثالث لازم اســت تا یک لیســت یا آبجکت مرکب بســازد و از طریق آن متد Create Iterator را صـدا بزند و آن را در اختیار کلاینت قرار دهد, ارتباط کلاینت با پیمایشــگر و آبجکت مرکب فقط از طریق اینترفیس آن‌ها می‌باشد.

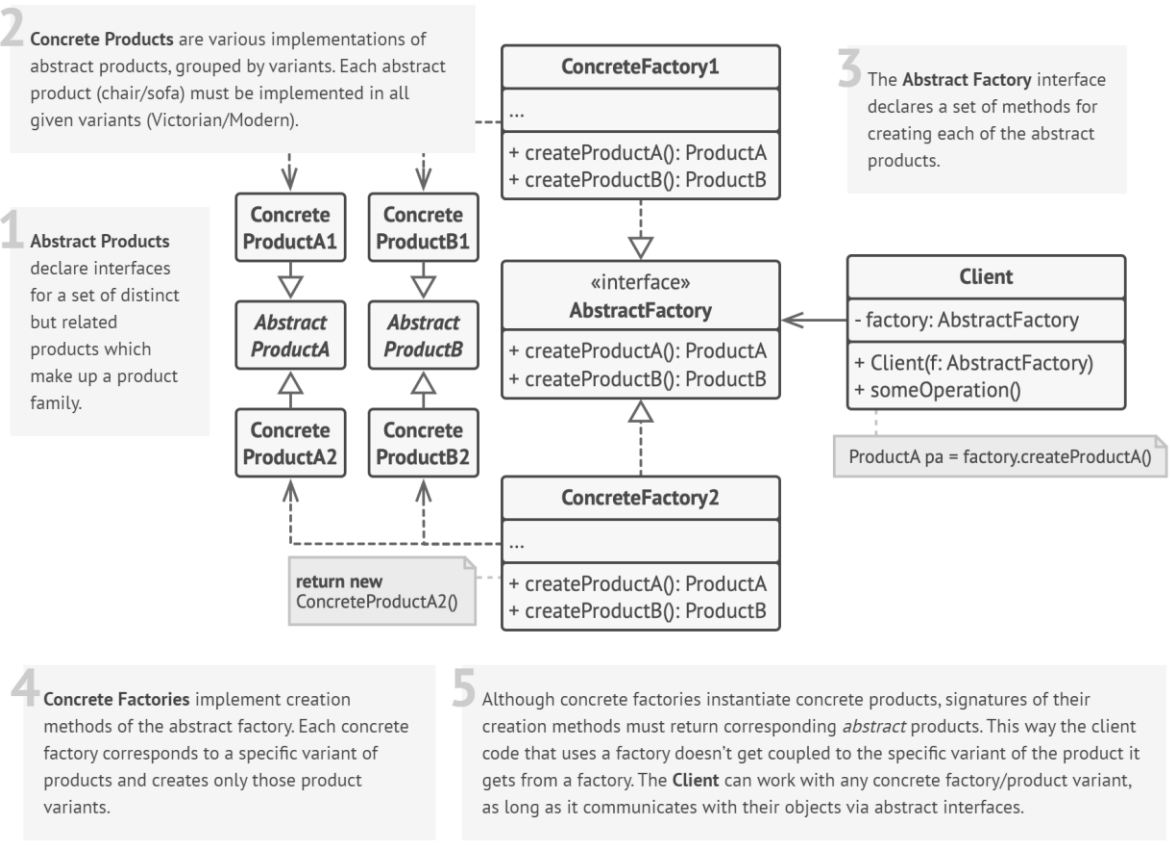
Builder – یک پیکربند که در واقع همان کلاینت هست انواع کلاس‌های Concrete Builder را می‌شناسد و آن را نمونه‌ســـازی می‌کند و در اختیار Director قرار می‌دهد تا با اســـتفاده از Director انواع مختلف آبجکت‌ها را می‌ســازد. در مورد این کلاینت DIP نقض اســت زیرا تمام کلاس‌های ســطح پایین Builder را می‌بیند.

مقایسه: هر ۳ الگو نیاز به پیکربند ثالث دارند.

ساختار و رفتار

در این قسمت توضیحات تکمیلی در شکل موجود است.

Abstract Factory

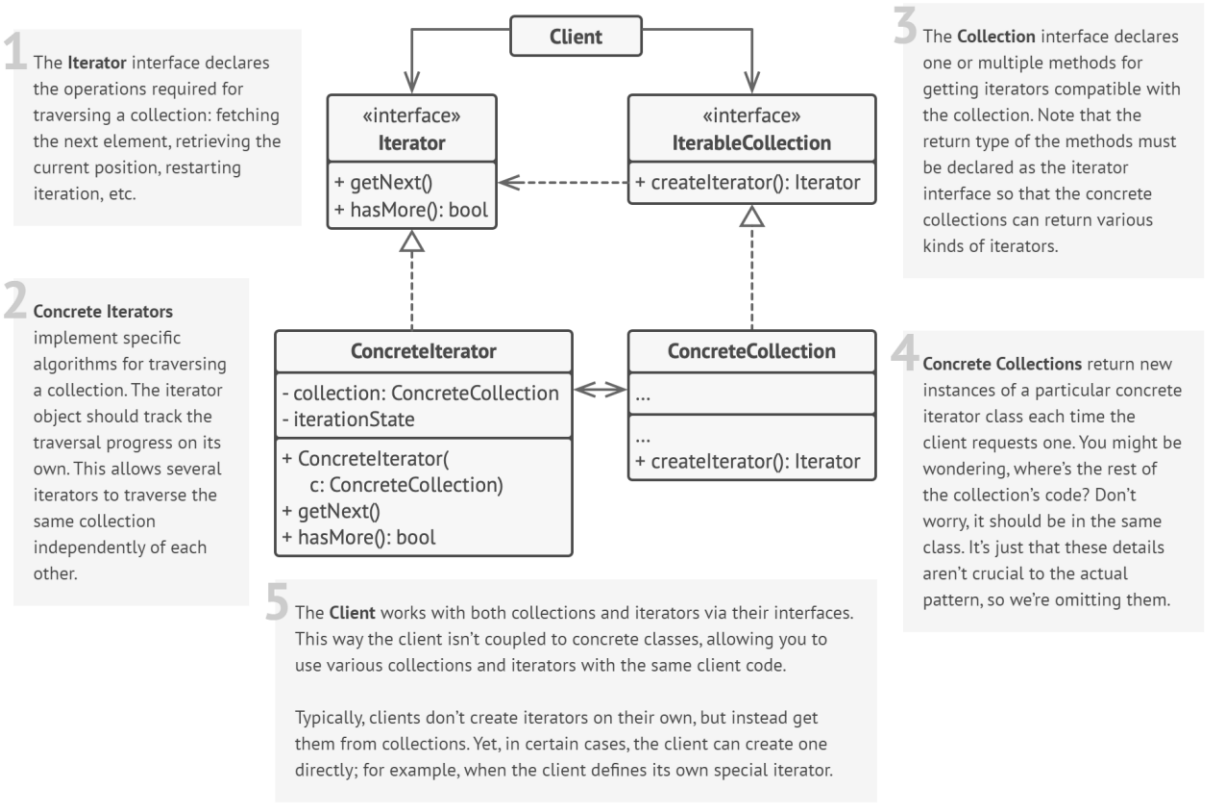


شکل ۷ ساختار الگوی Abstract Factory [۷]

همان‌طور که در شـــکل ۷ می‌بینید Client فقط از طریق اینترفیس Abstract Factory با آن کار می‌کند و متدهای Create را صـدا می‌زند, اما در سـطح پایین‌تر یعنی Concrete Factory ها, دسـترسـی به متد Constructor برای تولید Concrete Product ها دیده می‌شـــود, یعنی این ارتباط کمی در ســـطح پایین هسـت ولی نمی‌توان گفت Encapsulation را نقض کرده اسـت. پیکربند باید انواع Concrete Factory ها را بشـناسـد و آن را در اختیار کلاینت قرار دهد. بحث‌های مربوط به تغییرپذیری و DIP در قسـمت‌ها مربوطه به‌خوبی توضیح داده شده است.

ساختاری که در اینجا آمده تفاوت خاصی با الگویی که در کتاب هست ندارد. به همین دلیل از فقط یکی از آنها ذکر شده است.

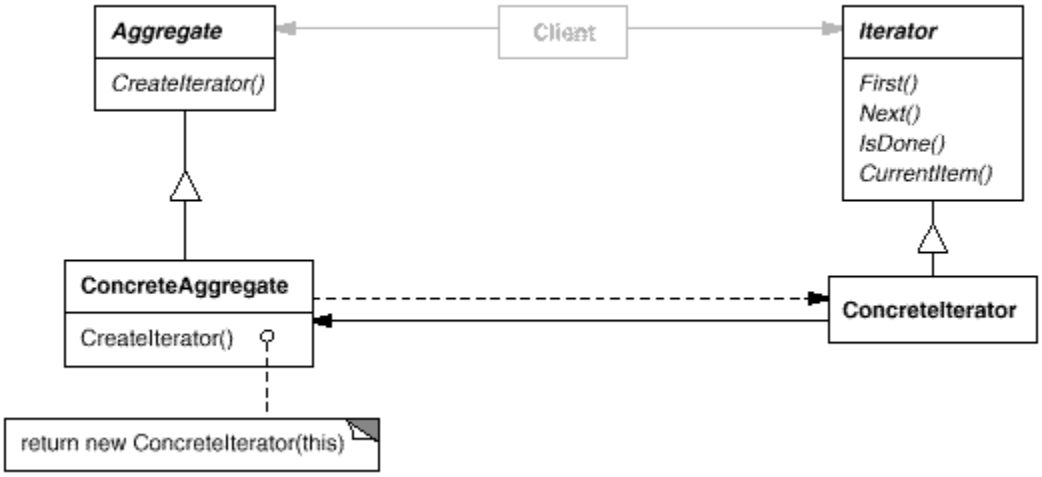
Iterator



شکل ۸٫۱ ساختار الگوی Iterator [۸]

همان‌طور که در شکل ۸٫۱ مشخص است کلاینت با اینترفیس‌های کلاس‌های Iterator و Collection که در واقع به ترتیب همان پیمایشگر و آبجکت مرکب هست کار می‌کند به‌این‌ترتیب متوجه تغییرات آن‌ها نمی‌شود. حال می‌بینیم که در ۲ ساختار توارثی موازی رابطه نیز وجود دارد, زیرا هر Collection متد ساخت و دریافت پیمایشگر را در خود دارد, هر پیمایشگر حالت پیمایش را در خود نگه می‌دارد تا بتوان از چندین پیمایشگر بر روی یک Collection یا همان آبجکت مرکب استفاده کنیم. ارتباط بین کلاینت و آبجکت مرکب که به آن در این شکل عنوان Collection داده‌ایم, می‌تواند به‌صورت غیر مانا و Dependency باشد

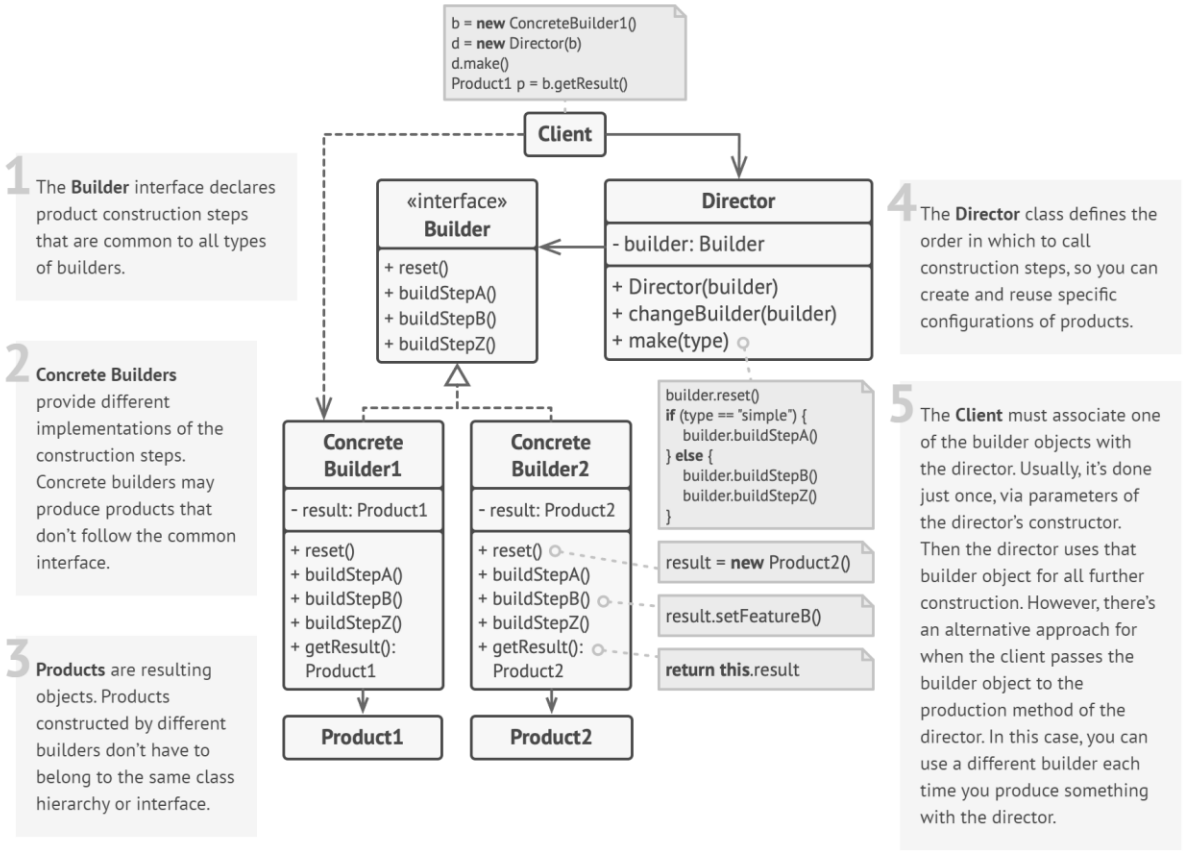
زیرا فقط می‌تواند پس از دریافت از پیکربند متد Create Iterator را صـــدا بزند و با آن کار کند. هم چنین ارتباط از سمت آبجکت مرکب به پیمایشگر هم به همین دلیل می‌تواند به‌صورت Dependency باشد.



شکل ۸٫۲ ساختار الگوی Iterator کتاب

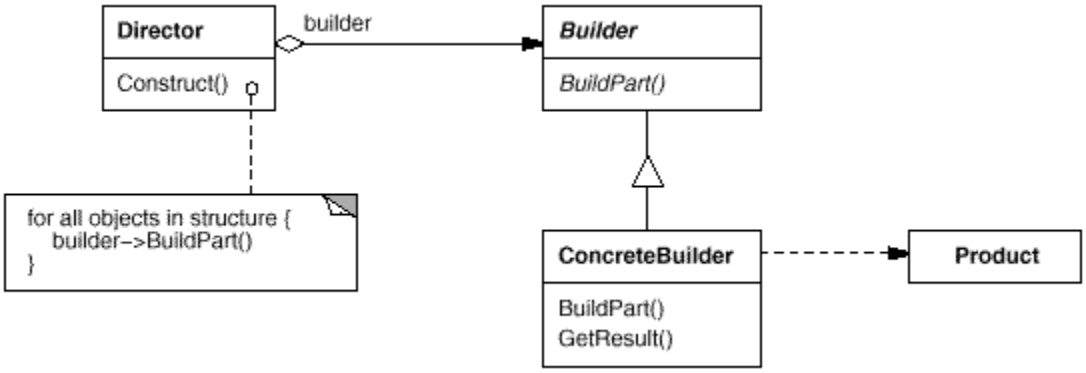
در ساختاری که در کتاب هست ارتباط بین Concrete Iterator ها و Concrete Aggregate ها به‌صورت Association ۲ طرفه نیست و از سمت Concrete Aggregate به سمت Concrete Iterator به‌صورت Dependency که غیر مانا هست کشیده شده است, این ارتباط درست‌تر است زیرا فقط Concrete Iterator توسط Concrete Aggregate مربوطه ساخته شده, و پس از آن ارتباطی بین آن‌ها نیست. (مگر از سمت Concrete Iterator به Concrete Aggregate که آن ماناست.)

Builder



شکل ۹٫۱ ساختار الگوی Builder [۹]

همان‌طور که از شکل۹٫۱ پیداست کلاینت با کلاس Director کار می‌کند و در واقع یکی از Concrete Builder ها را در آن قرار می‌دهد و بعد از طریق متدهایی که تعریف شده برای انواع آبجکت‌های درشت‌دانه, از اینترفیس Builder استفاده می‌کند تا از عملیات های Build Part استفاده کند و آن آبجکت را بسازد, در اینجا تغییرات در Builder به Director منتشر نمی‌شود چون از اینترفیس سطح بالای آن استفاده می‌شود. متد Get Result در واقع آبجکت موردنظر را برمی‌گرداند و این کار از طریق داشتن دید مستقیم غیر مانا که کلاینت به Concrete Builder ها دارد, انجام می‌پذیرد, لازم به ذکر است که این دید به‌صورت Dependency و غیر مانا هست, زیرا الزامی به همیشگی بودن آن نیست.



شکل ۹٫۲ ساختاری الگوی Builder کتاب

همان‌طور که مشخص است تنها تفاوت ساختار ۹٫۱ با ساختار کتاب, رابطه Dependency بین کلاینت و Concrete Builder ها است زیرا علاوه بر نمونه‌سازی اولیه و پاس دادن آن به Director نیاز دارد تا در صورت لزوم متد Get Result را صدا کند. پس این ارتباط به نظر صحیح می‌باشد.

مقایسه: در الگوی Abstract Factory و Builder در حال ساخت آبجکت‌های موردنظر با مکانیزم‌هایی هستیم که مدنظر داریم, یعنی به ترتیب برای ساخت خانواده‌ای از محصولات یا ساخت آبجکت‌های متفاوت با ساختار کدهای یکسان اما با Step یا Part های متفاوت. اما Iterator برای پیمایش آبجکت‌های مرکب استفاده شده است. توضیحات در مورد رفتار و ساختار این ۳ الگو در بالا و در کنار دیاگرام‌ها قابل‌مشاهده است.

کارایی از منظر حافظه

Abstract Factory - قبل از اعمال الگو در کلاس‌های Factory وجود نداشت اما بعد از انجام آن اضافه می‌شوند و این تعداد آبجکت‌های موجود در حافظه را افزایش می‌دهد اما این آبجکت‌ها زیاد نیستند چون می‌توان مثلاً Singleton هم با آن ترکیب کرد.

Iterator – در این الگو نیز همانند الگوی Abstract Factory قبل از اعمال الگو کلاس‌های Iterator وجود نداشتند اما بعد از آن کمی از اندازه و سنگینی آبجکت‌های مرکب کم شده و این وظیفه به آبجکت‌های پیمایشگر محول شده پس تعداد آبجکت‌های موجود در حافظه بیشتر می‌شود. البته این موضوع که می‌توانیم تعدادی پیمایشگر هم زمان داشته باشیم می‌تواند مشکل‌ساز باشد (تعداد خیلی‌خیلی زیاد بسازیم و Dispose نکنیم).

Builder – در حالت قبل از انجام الگو Director و Builder ها موجود نیستند و از طریق ساختار توارثی این کار انجام می‌شود, در حالت قبل تعداد کلاس‌ها می‌تواند به‌شدت زیاد باشد (تعداد کلاس با تعداد آبجکت متفاوت است) ولی تعداد آبجکت‌ها افزایش چشمگیری نداشته است.

مقایسه: هر ۳ الگو به مقدار کمی تعداد آبجکت‌های موجود در سیستم را افزایش داده‌اند اما به توجه به انعطاف‌پذیری آن‌ها و هم چنین کمکی که در آن Context خاص به ما می‌کند قابل نادیده‌گرفتن است. اما همان‌طور که در بالا اشاره شد در الگوی Iterator در واقع باید دقت لازم در نمونه‌سازی از پیمایشگرها را داشته باشیم.

موارد کاربرد

Abstract Factory

۱- وقتی لازم داریم تا سیستم از اینکه آبجکت‌های چگونه به وجود می‌آیند یا Compose می‌شوند و یا ساختار داخلی آن چگونه است, مستقل باشد.
۲- وقتی نیاز داریم چندین خانواده از محصولات داشته باشیم و سیستم را لازم است با آن کانفیگ کنیم.
۳- وقتی خانواده‌ای از محصولات طراحی شده تا باهم استفاده شوند که سیستم را با آن Factory کانفیگ می‌کنیم.

٤- می‌خواهیم یک Class Library از پروداکت‌های بســـازیم ولی فقط می‌خواهیم اینترفیس آن‌ها معلوم باشد نه پیاده‌سازی آن‌ها.

Iterator

١- وقتی می‌خواهیم از duplicate code برای پیمایش جلوگیری کنیم.

٢- وقتی می‌خواهیم از polymorphic traverse اســـتفـاده کنیم یعنی یـک پیمـایش رو داده ساختارهای متفاوت اجرا کنیم.

٣- وقتی می‌خواهیم چندین پیمایش روی یک آبجکت مرکب داشته باشیم.

٤- وقتی می‌خواهیم به اطلاعات درون یک آبجکت مرکب دســـترســـی داشـــته باشـــیم بدون آنکه مستقیماً به حالت داخلی آبجکت دسترسی داشته باشیم.

٥- وقتی می‌خواهیم یک داده ساختار پیچیده را از کلاینت مخفی کنیم (برای راحتی یا امنیت).

Builder

١- وقتی می‌خواهیم الگوریتم ساخت رو از بخش‌های مختلف یک آبجکت پیچیده جدا و مستقل کنیم.

٢- وقتی می‌خواهیم نمایش‌های داخلی متفاوت از یک آبجکت داشته باشیم.

٣- وقتی می‌خواهیم از پدیدهٔ Telescopic constructor دوری کنیم.

مقایسه: برای هر ٣ الگو تقریباً تمام موارد کاربرد مهم ذکر شد, همان‌طور که قبلاً گفته شد تقریباً می‌توان گفت که Builder و Abstract Factory برای تولید آبجکت‌های مختلف (حال به‌صـــورت خانواده یا چندین آبجکت با نمایش‌های داخلی متفاوت) استفاده می‌شود.

الگوهای مرتبط

Abstract Factory

١- در واقع با Factory Method برای تولید محصـولات شـروع می‌شـود (سـاده‌تر اما با subclassing بهتر هندل می‌شود) و به Abstract Factory, Builder می‌رسیم (پیچیده‌تر اما انعطاف‌پذیرتر)

۲- همان‌طور که قبلاً گفته شد Abstract Factory خانواده‌ای از محصولات را به‌صورت آنی می‌سازد و برمی‌گرداند اما Builder یک آبجکت پیچیده را از طریق Assemble کردن و اضافه‌کردن Partمی‌سازد.

۳- Abstract Factory, Builder, Prototype را می‌توان به‌صورت Singleton پیاده‌سازی کرد.

٤- می‌توان به‌عنوان Facade از آن استفاده کرد, منتها در جایی که فقط تولید آبجکت‌ها در زیرسیستم می‌خواهیم از دید کلاینت مخفی کنیم.

٥- می‌توان همراه با Bridge استفاده کرد در جایی که یک سری انتزاعات فقط با سری خاصی از پیاده‌سازی‌ها کار می‌کند و می‌خواهیم این پیچیدگی را از دید کلاینت مخفی کنیم.

Iterator

۱- از Iterator می‌توان استفاده کرد تا درخت‌های Composite را پیمایش کرد.

۲- از Factory Method می‌توان استفاده کرد تا پیمایشگرهای متفاوت بر اساس نوع آبجکت مرکب برگردانده شود.

۳- از Memento می‌توان استفاده کرد تا حالت کنونی پیمایش را ذخیره کرد و آن را بازگردانی کرد.

٤- مورد سوم الگوی Visitor نیز برای این الگو صادق است و در ارتباط با یکدیگر هست.

Builder

۱- مورد اولی که در Abstract Factory ذکر شد برای این الگو نیز صادق است.

۲- مورد دومی که در Abstract Factory ذکر شد برای این الگو نیز صادق است.

۳- مورد سومی که در Abstract Factory ذکر شد برای این الگو نیز صادق است.

٤- می‌توان از Builder هنگام تولید درخت‌های Composite که فرایند ساخت پیچیده‌ای دارد استفاده کرد.

مقایسه: موارد مرتبط ذکر شد و واضح است که Abstract Factory و Builder شباهت و الگوهای مرتبط زیادی دارند.

مزایا معایب

Abstract Factory

۱- تغییر خانواده‌ای از محصولات بسیار راحت و ساده است و با تغییر Concrete Factory ها انجام می‌پذیرد. این مزیت است.

۲- کلاس‌های Concrete کاملاً ایزوله هستند و کلاینت می‌تواند از Instance های متفاوت استفاده کند بدون آنکه متوجه شود.

۳- سازگاری بین محصولات ارتقا پیدا می‌کند. این مزیت است.

٤- اضافه‌کردن نوع جدید محصول باعث تغییر اینترفیس و کلاس‌های Factory و هم چنین کد پیکربند تغییر می‌کند. این عیب است.

Iterator

۱- چندین پیمایش را می‌توان بر روی یک آبجکت مرکب داشت. این مزیت است.

۲- انواع پیمایش را می‌توان بر روی یک آبجکت مرکب داشت. این مزیت است.

۳- اینترفیس آبجکت مرکب ساده‌تر می‌شود زیرا وظیفه پیمایش به Iterator سپردیم. این مزیت است.

٤- اگر سیستم فقط با آبجکت‌های مرکب ساده استفاده کند باعث Overkill اپلیکیشن می‌شود. این عیب است.

٥- تغییرات منتشر شونده بین Iterator و Aggregate Object زیاد است. این عیب است.

Builder

۱- می‌توان آبجکت‌ها با نمایش‌های داخلی متفاوت ایجاد کرد. این مزیت است.

۲- در واقع فرایند ساخت و نمایش داخلی جدا شده است. این مزیت است.

۳- کنترل بهتری روی فرایند ساخت داریم زیرا Director قدم‌به‌قدم این کار را انجام می‌دهد. این مزیت است.

مقایسه: موارد در بالا ذکر شده است و به‌راحتی می‌توان باتوجه‌به Context مسئله و trade-off که می‌توان در نظر داشته باشیم الگوی موردنظر را انتخاب کنیم.

شیء جعلی

Abstract Factory - کلاس و آبجکت‌های Factory همه تصنعی هستند و در قلمرو مسئله نیستند.

Iterator – آبجکت و کلاس‌های Iterator در قلمرو مسئله نیست و برای تحقق الگو تولید شده و جعلی است.

Builder – تمام کلاس‌ها و آبجکت‌های Builder و هم چنین Director در قلمرو مسئله نیستند و مصنوعی هستند.

مقایسه: هر ۳ شیء جعلی تولید کرده‌اند.

جداسازی دغدغه‌ها

Abstract Factory - این الگوریتم در واقع وظیفه ساختن آبجکت‌ها را از کلاینت گرفته و به Factory ها داده و کلاینت از طریق اینترفیس با آن‌ها کار می‌کند.

Iterator – این الگو وظیفه پیمایش را از آبجکت‌های مرکب گرفته و به کلاس پیمایشگر داده و کلاینت با اینترفیس آن‌ها کار می‌کند.

Builder – به‌وضوح فرایند ساخت که به کلاینت محول شده بود جدا شده و به Builder داده شده است تا با اســتفاده از دســتور دادن از Director به متدهای Add Part که به‌نوعی فرایند ســاخت هســتند و در Builder ها موجود هستند آبجکت‌های پیچیده را ساخته تا فرایند ساخت از نمایش داخلی آن جدا شود.

مقایسه: هر ۳ الگو به‌خوبی این کار را انجام داده‌اند.

پیاده‌سازی

Abstract Factory - ابتدا انواع مختلف پروداکت ها و خانواده‌های آن‌ها را شناسایی می‌کنیم.
یک اینترفیس برای آن‌ها تعریف و آن را در کلاس‌های محصول پیاده‌سازی می‌کنیم.
اینترفیس برای تولید هر خانواده از پروداکت با نام Abstract Factoryمی‌ســازدیم و بعد آن را در کلاس‌های Factory پیاده‌سازی می‌کنیم. هر پیاده‌سازی برای یک خانواده می‌باشد.

Iterator – یک اینترفیس برای پیمایشــگر با حداقل متد get Nextمی‌ســازدیم (بســته به کار و نیازها این اینترفیس پیچیده‌تر می‌شود.).
اینترفیس برای آبجکت مرکب می‌سازیم و در آن متدی برای دریافت پیمایشگر می‌نویسیم.
ســپس پیاده‌ســازی پیمایشــگر را برای آبجکت مرکب می‌نویســیم, هر پیمایشــگر باید با یک آبجکت مرکب لینک باشد که این کار از طریق سازندهٔ پیمایشگر (معمولاً) انجام می‌شود.
بعد اینترفیس آبجکت مرکب را باهدف اینکه یک پیمایشگر را به‌راحتی در اختیار کلاینت قرار دهد پیاده‌سازی می‌کنیم.

Builder – ابتدا اطمینان پیدا می‌کنیم که می‌توان Step های همه نمایش‌های ممکن محصــولات را تعریف کنیم.
سپس این قدم‌ها را در اینترفیس Builder می‌نویسیم.

برای هر نمایش محصول یک Concrete Builderمی‌سازدیم و این قدم‌ها را پیاده‌سازی می‌کنیم.

حال می‌توانیم یک Director بسازیم و محصولات ممکن را در آن با استفاده از اینترفیس Builder و آبجکت Builder که به آن پاس داده شده است بسازیم.

مقایسه: به نظر می‌توان گفت Iterator به دلیل آنکه دو ساختار توارثی موازی دارد که به همدیگر هم Coupled هستند (زیرا پیمایشگر لازم دارد با حالت داخلی آبجکت مرکب کار کند) و هم چنین در بعضی از زبان‌ها مانند C# لازم است از کلاس‌های موجود در قسمت Enumeration ارث‌بری کند, پیچیدگی بیشتری دارد.

اما Builder و Abstract Factory تقریباً پیچیدگی متوسطی دارند, اما Abstract Factory به دلیل ساختارهای محصولات و تعداد اینترفیس‌ها و کلاس‌های زیاد, پیچیدگی بیشتری دارد.

بررســـی الگوهـای **GRASP** برای ۳ الگوی **Abstract Factory, Iterator** و **Builder**

Information Expert

Abstract Factory - این الگو برای تولید خانواده‌ای از محصولات استفاده می‌شود و هر Factory اطلاعات لازم برای تولید آن‌ها را دارد و این کار را انجام می‌دهد پس این الگو نقض نشده است.

Iterator – باتوجه‌به این که هر آبجکت مرکب در واقع به‌صورت Aggregate درون یک پیمایشـگر وجود دارد و در واقع داده و رفتار در کنار هم نیست این الگو نقض شده است.

Builder – در این الگو اگر بخواهیم وظیفه ســـاخت را به Director نســـبت دهیم (خود فقط از اینترفیس Builder اســـتفاده می‌کند) این الگو را نقض می‌کند اما اگر این وظیفه را به Builder نســـبت دهیم و Director را بیشـتر میانجی قرار دهیم, این الگو را نقض نمی‌کند زیرا اطلاعات سـاخت محصول و رفتارهای آن در کلاس Builder هسـت. (در هر ۲ صورت می‌توان Director را حذف کرد و کلاینت به‌صورت مسـتقیم با اینترفیس Builder کار کند.)

Creator

Abstract Factory - این الگو با اولویت پنجم این الگو را رعایت می‌کند یعنی اطلاعات لازم برای تولید و initialize کردن یک محصول داخل Factory موجود هست و هر Factory محصولات خود را می‌سازد.

Iterator – ازآنجایی‌که کلاینت داده‌های لازم برای آبجکت‌های مرکب را دارد و آن را می‌سازد (اولویت پنجم) و در اینجا الگو رعایت شده. ولی ازآنجایی‌که هر پیمایشگر یک آبجکت مرکب در خود به‌صورت Aggregation دارد باید آن را بسازد, اما این کار برعکس اتفاق افتاده پس این الگو در اینجا نقض شده.

Builder – در این الگو کلاینت نمونه‌ای از Builder را در اختیار Director قرار می‌دهد, این یعنی اولویت ٥ که اولویت بالاتری وجود دارد, زیرا Director به‌صورت مستقیم در حال استفاده از Builder است باید آن را بسازد و این یعنی اولویت ٤ که نقض شده است.

Coupling

Abstract Factory - با رجوع به قسمت کاهش وابستگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و Coupling را کم می‌کند.

Iterator – با رجوع به قسمت کاهش وابستگی برای این الگو متوجه می‌شویم که تأثیر متوسطی برای این الگوی Grasp دارد و Coupling را کم می‌کند اما ارتباط بین آبجکت مرکب و پیمایشگر هنوز بسیار Coupled هست.

Builder – با رجوع به قسمت کاهش وابستگی برای این الگو که تأثیر زیادی برای این الگوی Grasp دارد و Coupling را کم می‌کند.

Cohesion

Abstract Factory - با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و تقریباً Cohesion را به حد خوبی می‌رساند.

Iterator – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و تقریباً Cohesion را به حد خوبی می‌رساند.

Builder – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و آن را محقق می‌سازد.

Controller

Abstract Factory - این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Iterator – این الگو در واقع کارچرخانی ندارد و Controller ندارد ولی بعد از اینکه پیمایشگر نمونه‌سازی شد, بعد از دریافت هر پیام از کلاینت وظیفه اعمال عملیات روی آبجکت مرکب به عهده Iterator است.

Builder – در این الگو Director به‌عنوان use-case controller می‌توان در نظر گرفته شود زیرا درخواست ساخت را از کلاینت گرفته و به Builder می‌دهد و پس از آن پاسخ می‌تواند با فراخوانی Get Result در اختیار کلاینت قرار بگیرد.

Polymorphism

Abstract Factory - این الگو با داشتن فوق کلاس برای Factory ها و پیکربندی‌های متفاوت بر اساس خانواده محصولی موردنظر, این الگو را رعایت کرده.

Iterator – این الگو با داشتن اینترفیس مشترک برای پیمایشگر, کلاینت به‌راحتی و بدون مشکل از آن برای داده ساختارهای متفاوت و آبجکت‌های مرکب متفاوت استفاده می‌کند. این الگو رعایت شده است.

Builder – درون کلاس Director ارتباط بین او و Builder در سطح بالا و اینترفیس است به همین دلیل می‌توان بدون اینکه Director متوجه شود آبجکت Builder را عوض کرد, در نتیجه این الگو رعایت شده است.

Indirection

Abstract Factory - همان‌طور که در ساختار دیده شد ارتباط بین کلاینت با محصولات از طریق اینترفیس Factory ها انجام گرفته و این یعنی Indirection داریم. ولی رابطه بین محصولات و Factory ها به‌صورت مستقیم است.

Iterator – همان‌طور که در ساختار دیده شد ارتباط بین پیمایشگر و آبجکت مرکب خود به‌صورت مستقیم و در سطح پایین است اما ارتباط بین کلاینت و آبجکت مرکب از طریق پیمایشگر و اینترفیس آن است و این یعنی Indirection داریم.

Builder – همان‌طور که در ساختار دیده شد ارتباط بین کلاینت و محصولات به‌واسطه کلاس Builder (در نبود Director) یا به‌واسطه ی خود Director غیرمستقیم شده و آن‌ها را نمی‌بینند. این یعنی Indirection داریم. یعنی دیگر کلاینت به‌صورت مستقیم محصولات را نمی‌سازد.

Pure Fabrication

Abstract Factory - تمام زیر کلاس‌های Abstract Factory جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Iterator – تمام کلاس‌های Iterator نیز جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Builder – کلاس‌های Builder و کلاس Director برای راحتی حل مسئله تولید شده‌اند.

Protected Variations

Abstract Factory - همان‌طور که در بحث تغییرپذیری گفته شـــد ازآنجایی‌که ارتباط بین Factory و محصولات در سطح پایین و Concrete هست و کم یا اضافه‌کردن یک محصول باعث انتشار تغییرات و تغییر اینترفیس و تمام کلاس‌های Factory و کلاینت می‌شـــود. اما اضـــافه‌کردن یک خانواده جدید از محصـــولات بســـیار ســـاده و راحت اســـت و تغییری جایی منتشـــر نمی‌شـــود. هم چنین تغییرات به کلاینت منتشـــر نمی‌شود چون به‌خوبی DIP را رعایت کرده است.

Iterator – همان‌طور که در بحث تغییرپذیری گفته شـــد ارتباط بین کلاینت و آبجکت مرکب از طریق اینترفیس سطح بالا و بدون انتشـار تغییر اسـت, اما در دو سـاختار موازی توارثی برای پیمایشـگر و آبجکت مرکب, چون ارتباط به‌صورت مستقیم و در سطح Concrete هست این تغییر منتشر می‌شود.

Builder – همان‌طور که در بحث تغییرپذیری گفته شد در این الگو چون Director با اینترفیس سطح بالای Builder کار می‌کند تغییرات به آن منتشر نمی‌شود ولی اضافه یا کم‌کردن قطعه جدید باعث منتشر شدن این تغییر می‌شـود که در واقع اینترفیس و زیر کلاس‌های Builder دچار تغییر می‌شـوند. هم چنین کلاینت نیز به‌ناچار به خاطر متد Get Result به زیر کلاس‌های Builder دید مســـتقیم اما غیر مانا دارد و اگر تغییراتی مثل اضـافه و کم شـدن زیر کلاس‌ها (برای پیکربندی) یا تغییرات در امضـای متد Get Result (و یا حتی بقیه متدهای Builder در حالتی از الگو که Director نداریم) , این تغییرات به پیکربند منتشـــر می‌شود.

مقایسه الگوهای **Memento, Flyweight** و **Item Description**

دسته - نوع

Memento – این الگو در درسته الگوهای رفتاری می‌باشد. الگوهای رفتاری بیشتر دغدغهٔ تخصیص مسئولیت‌ها به آبجکت‌ها, یا کپسوله‌سازی رفتار در یک آبجکت یا تفویض کردن ریکوئست‌ها به آبجکت و مدیریت بهتر تعامل آبجکت‌ها در زمان اجرا با کم‌کردن Coupling و بالابردن Cohesion را دارد.

Flyweight – این الگو در درسته الگوهای ساختاری می‌باشد. الگوهای ساختاری با در کنار هم قراردادن کلاس‌ها و آبجکت‌ها باعث تولید ساختارهای بزرگ‌تر می‌شود درحالی‌که این ساختارها را انعطاف‌پذیر و کارا نگه می‌دارد. در واقع با شناسایی ارتباطات باعث ساده شدن ساختار می‌شود.

Item Description – این الگو جزو دسته الگوهای ۷تایی Coad می‌باشد.

مقایسه: همان‌طور که دیدیم الگوی Memento در دسته الگوهای رفتاری و Flyweight در دسته الگوهای ساختاری و هم چنین Item description در دسته الگوهای Coad هستند. می‌توان گفت چون Item description در واقع ساختار کلاسی را ساده‌تر کرده و هم چنین این ساختارها انعطاف‌پذیر هستند, می‌توان آن را در مجموعه الگوهای ساختاری نیز در نظر گرفت.

هدف

Memento – ذخیره‌سازی حالت یک آبجکت بدون نقض کردن Encapsulation از اهداف این الگو است. یعنی می‌توان حالت آبجکت را ذخیره و بازیابی کرد بدون آنکه جزئیات داخلی آن نمایان شود. یعنی می‌توان حالت آبجکت را ذخیره کرد تا بعداً بتوان به آن Restore کرد. در واقع یک Snapshot از آبجکت نگه می‌داریم.

Flyweight – الگویی ساختاری است که این امکان را می‌دهد تا در یک فضای مشخص با استفاده از مکانیزم Sharing که State های مشترک را بین آبجکت‌ها به اشتراک می‌گذارد, آبجکت‌های بیشتری ذخیره کند. (به‌جای ذخیره‌سازی بخش‌های مشترک در هر آبجکت).

Item Description – از این الگو در جایی می‌توان استفاده کرد که Value های چندین Attribute ممکن است در چندین کلاس تکرار شود, و این الگو باهدف Sharing این Attribute ها تحت عنوان Item Description به‌خوبی این کار را انجام می‌دهد.

مقایسه: Memento برای ذخیره‌سازی حالت داخلی آبجکت بدون نقض کردن بسته‌بندی و بازیابی آن حالت در صورت نیاز استفاده می‌شود. درحالی‌که هدف دو الگوی Flyweight و Item Description تقریباً متفاوت است, اما در اشتراک‌گذاری و Sharing این شباهت بین آن‌ها وجود دارد. Flyweight با جدا کردن Intrinsic State که ویژگی‌های ذاتی آبجکت هست, از Extrinsic State که وابسته به Context می‌باشد این کار را انجام می‌دهد و باعث صرفه‌جویی در مصرف حافظه می‌شود.

حوزه

Memento – این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Flyweight – این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

Item Description - این الگو در حوزهٔ شیء می‌باشد و در زمان اجرا و از طریق delegation می‌باشد.

مقایسه: هر سه الگو در زمان اجرا و از طریق delegation تحقق می‌یابند.

کاهش وابستگی

Memento – ازآنجایی‌که قبلاً Memento ها در دل خود Originator بوده و الان جدا شده‌اند لازم است که ارتباط بین آن دو در سطح پایین حفظ شود و به‌نوعی Encapsulation نقض شود و این یعنی وابستگی بالا, اما در طرف مقابل رابطه بین Caretaker و Originator به‌شدت کاهش پیدا کرده زیرا Caretaker حالت‌ها را فقط از طریق اینترفیس محدود Memento در اختیار دارد. البته هنوز رابطه Association بین Caretaker و Originator موجود هست.

Flyweight – ارتباط بین کلاینت و Flyweight ها, چه برای نمونه‌سازی که از بین رفته و چه برای انجام عملیات ها که از طریق اینترفیس انجام شده, بعد از اعمال الگو وابستگی مستقیم ندارد و یا از طریق اینترفیس است یا از بین رفته است. ولی فقط Flyweight Factory برای ساختن و نمونه‌سازی Flyweight ها نیاز دارد تا با کلاس‌های سطح پایین Flyweight کار کنند.

Item Description – ازآنجایی‌که قبل از اعمال الگو وابستگی ای به‌صورت کلی وجود نداشت و فقط یک کلاس Item داشتیم, و بعد از اعمال الگو هر کلاس Item در داخل خود یک کلاس Item Description دارد و برای واکشی اطلاعاتی که به‌صورت Shared داخل Item Description هست, نیاز هست تا به حالت داخلی او دسترسی داشته باشد, پس وابستگی افزایش‌یافته است.

مقایسه: برای الگوی Memento از یک سمت وابستگی کاهش و از سمت دیگر تقریباً کاهش چندانی نداشته است (زیرا Caretaker هنوز باید عملیات های Save یا Restore را از Originator صدا بزند و علت رابطه Association بین آن‌ها نیز همین است.). هم چنین در Flyweight از طرف کلاینت به سمت آبجکت‌های ریزدانه وابستگی کاهش یافته است ولی از سمت Factory به سمت Flyweight ها وابستگی در سطح کلاس‌های Concrete داریم. اما در Item Description وابستگی افزایش‌یافته است.

افزایش چسبندگی

Memento – ازآنجایی‌که دیگر Originator درگیر ذخیره و بازیابی حالت نیست و Caretaker با استفاده از آبجکت‌های Memento این کار را انجام می‌دهد, تک کارگی افزایش‌یافته است و به‌نوعی Cohesive شده‌اند.

Flyweight – قبل از اعمال الگو کلاینت وظیفه ساختن و جستجو در آبجکت‌های Flyweight را دارد, اما بعد از اعمال الگو این وظیفه به Flyweight Factory داده شده است که کاملاً نشان‌دهندهٔ کاهش چندکارگی و افزایش چسبندگی است.

Item Description – ازآنجایی‌که به قولی قبل از اعمال الگو همانند آن بوده است که دو کلاس را ترکیب کرده‌ایم و باعث این تکرار Attribute های مشترک شده‌ایم و بعد از اعمال الگو این ۲ کلاس جدا و به‌صورت Single-purpose شده‌اند, می‌توان گفت چسبندگی افزایش‌یافته است زیرا تک کارگی افزایش‌یافته است.

مقایسه: هر ۳ الگو به‌نوعی Cohesion را افزایش داده‌اند و به حد خوبی رسانده‌اند.

OCP

Memento – تغییرات در Memento ها به Originator منتشر می‌شود زیرا در سطح پایین با یکدیگر کار می‌کنند و برعکس, پس در اینجا OCP نقض شده است. اما از سمت Caretaker چون فقط با اینترفیس Memento کار می‌کند این اصل رعایت شده است.

Flyweight – ازآنجایی‌که ارتباط بین کلاینت و Flyweight از طریق اینترفیس سطح بالای آن است پس DIP برقرار است و در نتیجه OCP داریم. اما رابطه بین Factory و Flyweight ها در سطح پایین و در سطح کلاس‌های Concrete است پس OCP برقرار نیست.

Item Description – همان‌طور که در بحث وابستگی گفته شد چون یک رابطه Association بین Item و Item Description داریم و در واقع با نقض DIP این کار انجام می‌پذیرد پس در نتیجه OCP نقض شده است.

مقایسه: همان‌طور که اشاره شد در الگوی Memento از یک سمت این اصل نقض شده ولی در سمت دیگر به‌خوبی رعایت شده زیرا ارتباط Caretaker و Memento در سطح اینترفیس است و DIP برقرار است. در الگوی Flyweight نیز همان‌طور که در بالا گفته شد از سمت Factory به Flyweight این اصل برقرار نیست. در الگوی Item Description نیز به دلیل اینکه کلاس‌ها به‌صورت مستقیم و در سطح پایین با یکدیگر کار می‌کنند OCP برقرار نیست.

LSP

Memento – همان‌طور که در ساختار و رفتار این الگو اشاره خواهد شد ازآنجایی‌که اینترفیس‌های Wide & Narrow برای Originator و Caretaker برای کار با Memento ها ساخته می‌شود و آن دو با این

اینترفیس‌ها کار می‌کنند, می‌توان از زیر کلاس‌های مختلف Memento اســـتفاده کرد بدون آنکه متوجه شـوند, زیرا هر دو با اینترفیس Memento کار می‌کنند. (درسـت اسـت که Originator با زیر کلاس عینی Memento رابطه دارد, اما این رابطه مانا نیســـت و از نوع Dependency هســـت و فقط برای Instance ســاختن اســت, و برای get State اینترفیس تعریف می‌کنیم). در این صـورت LSP برقرار خواهد شـد اما به‌خودی‌خود در این الگو این اصل برقرار نیست.

Flyweight – کاملاً برقرار اسـت ارتباط کلاینت با Flyweight ها از طریق اینترفیس سـطح بالای آن اسـت و فقط از طریق او عملیات انجام می‌دهد, پس Extension و اضافه و کم‌کردن زیر کلاس‌ها به کلاینت منتشـر نمی‌شود و به‌راحتی با آن‌ها می‌تواند کار کند و این یعنی LSP برقرار است.

Item Description – برای این الگو خیلی نمی‌توان راجع به این اصـل صـحبت کرد زیرا که سـاختار توارثی خاصـی دیده نمی‌شـود, اما درصـورتی‌که مثلاً برای Item Description ها بخواهیم با تعریف یک سـاختار توارثی این Attribute هـا را ســـاختـاردهی کنیم, می‌توان در کلاس Item هم از کلاس پـدر Item Description استفاده کرد و LSP برقرار باشد. اما در حالت کلی نمی‌توان نظر خاصی داد.

مقایسه: در الگوی Memento همان‌طور که اشاره شد این اصل در شرایطی که اینترفیس‌ها تعریف شوند, برقرار خواهد شـــد. برای الگوی Flyweight همان‌طور که گفته شـــد این اصـــل برقرار اســـت. برای الگوی Item Description هم همان‌طور که گفته شـد نمی‌توان صـحبت خاصـی کرد مگر در شـرایطی که الگو به‌صورت دیگری (مثلاً پیشرفته‌تر از حالت معمول) استفاده شود.

DIP

Memento – از سمت Originator به Memento ها برقرار نیست چون با کلاس‌های سطح پایین یکدیگر در حال ارتباط هستند, اما از سمت Caretaker به Memento ها برقرار است زیرا فقط اینترفیس او را می‌شناسد.

Flyweight – ازآنجایی‌که ارتباط بین کلاینت و Flyweight در سطح اینترفیس است این اصل برقرار است. اما به‌ناچار برای نمونه‌سازی از Flyweight در داخل Factory نیاز هست تا این ارتباط در سطح کلاس‌های Concrete باشد و در اینجا این اصل نقض شده است.

Item Description – همان‌طور که در بحث OCP و وابستگی توضیح داده شد چون ارتباط کلاس‌ها در سطح پایین و بدون استفاده از اینترفیس هست, DIP نقض شده است.

مقایسه: در الگوی Memento همان‌طور که اشاره شد از سمت Caretaker به Memento برقرار است (در صورت تعریف اینترفیس‌های مذکور). همان‌طور که در بالا گفته شد برای Flyweight از سمت Factory به Flyweight این اصل برقرار نیست. هم چنین در Item Description این اصل نقض شده است.

ISP

Memento – همان‌طور که می‌دانیم چون دو اینترفیس Wide و Narrow تعریف می‌شود و عملیات های هرکدام با دیگری متفاوت است می‌توان گفت این اصل برقرار است. مثلاً در Wide که گسترده‌تر است عملیات های get State و set State که مختص Originator هست تعریف می‌شود. هرکدام از این اینترفیس‌ها به‌صورت Cohesive و Specific هستند پس این اصل رعایت شده است.

Flyweight – این الگو به‌صـــورت کلی برای انجام عملیات های خود از اینترفیس‌های Specific و تک کاره اسـتفاده می‌کند مگر در داخل Factory که بیشـتر برای نمونه‌سـازی هسـت که از Flyweight ها اسـت. پس می‌توان گفت این اصل نیز برقرار است.

Item Description – در این الگو ارتباط در سـطح پایین و بدون اسـتفاده از اینترفیس هسـت پس در مورد این اصل نمی‌توان صحبت خاصی کرد یا می‌توان گفت این اصل رعایت نشده است.

مقایسه: در الگوهای Memento و Flyweight این اصل برقرار است.

CRP

Memento – در این الگو به‌وضوح از Delegation به‌جای Inheritance استفاده شده پس این اصل برقرار است.

Flyweight – سـاختار توارثی خاصی در این الگو دیده نمی‌شـود مگر برای توسـعهٔ اینترفیس یا فوق کلاس Flyweight ها. بقیه کارها از طریق واسپاری انجام می‌گیرد, مثلاً لیستی از Flyweight ها درون Factory به‌صـورت Aggregate آمده اسـت, یا State ها در Context یا کلاس‌های Flyweight به‌صـورت واسـپاری شده هستند و نه ساختار توارثی. پس این اصل برقرار است.

Item Description – به‌وضوح مشـخص اسـت که به‌جای سـاختار توارثی از Delegation اسـتفاده شـده است و این اصل به‌خوبی رعایت شده است.

مقایسه: در هر ۳ الگو این اصل برقرار است.

PLK

Memento – می‌دانیم که هنگام Save و Restore که Caretaker آن‌ها را فراخوانی می‌کند آبجکت‌های Memento پاس داده می‌شود, اما این هدف الگو هست تا حالت‌ها درون آبجکت‌ها باشند و این نقض این اصل نیست و زنجیره دید ترایا ایجاد نمی‌کند, پس این اصل برقرار است.

Flyweight – به‌غیراز Factory که نمونه‌هایی از Flyweight ساخته و بر می‌گرداند جایی آبجکتی پاس داده نمی‌شود و زنجیره دید ترایا دیده نمی‌شود. (در Factory چون وظیفهٔ ساختن و کارخانه را دارد زنجیره دید ترایا به وجود نمی‌آید.) پس این اصل برقرار است.

Item Description – در هیچ جای الگو زنجیره دید ترایا دیده نمی‌شود.

مقایسه: در هر ۳ الگو این اصل برقرار است.

بسته‌بندی

Memento – در این الگو چون Memento ها در واقع در دل Originator بوده‌اند و از آن بیرون کشیده شده‌اند, در اینجا چون نیاز دارد به حالت درونی Memento برای ذخیره و بازیابی دسترسی داشته باشد, بسته‌بندی نقض شده است (می‌تواند این کار را از طریق Wide Interface انجام دهد تا بسته‌بندی نقض نشود) , اما از طرف دیگر چون قبل از اعمال الگو Caretaker به‌صورت مستقیم با حالت داخلی Originator کار می‌کرد ولی الان فقط از طریق اینترفیس محدود Memento کار ذخیره بازیابی را انجام می‌دهد (یعنی به Originator می‌گوید save یا restore) بسته‌بندی تا حد خوبی رعایت شده است.

Flyweight – در این الگو ارتباط بین کلاینت و Flyweight ها از طریق اینترفیس است پس کپسوله‌سازی به‌خوبی رعایت شده. از طرف دیگر ارتباط بین کلاینت و Factory در سطح پایین است اما می‌توان برای Factory هم اینترفیس تعریف کرد, زیرا در حالت عادی هم کلاینت به حالت داخلی Factory دسترسی ندارد (مگر نقض غرض باشد). هم چنین Factory به حالت داخلی Flyweight ها دسترسی ندارد و فقط آن‌ها را نمونه‌سازی می‌کند. به‌صورت کلی بسته‌بندی در جایی نقض نشده است.

Item Description – همان‌طور که قبلاً عنوان شد, ازآنجایی‌که Attribute ها از دل Item بیرون کشیده شده‌اند و خود Item به‌صورت مستقیم به حالت درونی Item Description دسترسی دارد, بسته‌بندی نقض شده است.

مقایسه: الگوی Memento در شرایط معمول بسته‌بندی را از سمت Originator به Memento کاملاً نقض می‌کند و یا در سمت Caretaker و Memento حفظ می‌شود, اما در قسمت ساختار و رفتار به حالتی از این الگو می‌پردازیم که بسته‌بندی نقض نشده است. در الگوی Flyweight همان‌طور که گفته شد, بسته‌بندی رعایت شده است. در Item Description نیز Encapsulation رعایت نشده است.

انعطاف‌پذیری

Memento – در این الگو چون دیگر خود Originator مجبور نیست تا حالات خود را ذخیره و بازیابی کند و این کار را Memento ها به‌واسطه Caretaker انجام می‌دهد, می‌توان گفت انعطاف‌پذیری برای نحوه ذخیره‌سازی و ... به Memento محول شده و می‌تواند انعطاف‌پذیری را افزایش دهد.

Flyweight – در این الگو نیز چون کلاینت در واقع با اینترفیس سطح بالای Flyweight کار می‌کند به‌راحتی می‌توان آن را عوض کرد و کلاینت متوجه تغییرات او نمی‌شود. فقط برای Factory باید کدها را

بازنویســـی کنیم زیرا این کلاس با کلاس‌های ســـطح پایین Flyweight کار می‌کند. به‌صـــورت کلی انعطاف‌پذیری خوبی دارد.

Item Description – در مورد انعطاف‌پذیری نمی‌توان صـــحبت خاصـــی کرد, ازآنجایی‌که فقط اطلاعات و Attribute های مشترک از داخل Item بیرون کشیده شده‌اند و برای تغییر نام این فیلدها یا کم‌وزیاد شدن آن‌ها باید کلاس Item تغییر کند (این عیب را در مقابل اسـتفاده درسـت از فضـای ذخیره‌سـازی وsharing می‌پذیریم.). هم چنین همان‌طور که قبل‌تر اشـاره شـد اگر برای Item Description یک سـاختار توارثی در نظر بگیریم, به‌راحتی می‌توان انواع مختلف آن را حتی در زمان اجرا به Item داد و او بدون آنکه متوجه شود می‌تواند با آن کار کند.

مقایســه: همان‌طور که در بالا اشــاره شــد هر ســه تا حد خوبی انعطاف‌پذیری دارند اما در شــرایطی مثلاً برای Flyweight و تعریف یک Flyweight جدید, این تغییرات منتشــر می‌شــود و باید بازنویســی انجام داد, برای الگوی Item Description به‌تفصیل در بالا توضیح داده شد.

تغییرپذیری/ انتشار تغییرات

Memento – ازآنجایی‌که Memento ها در واقع درون دل Originator بوده‌اند و بیرون کشــیده شــده‌اند و همان‌طور که گفته شــد, وابســتگی زیادی بین آن‌ها دیده می‌شــود, همین دلیل باعث می‌شــود تا تغییرات بین آن‌ها منتشر شود, زیرا با کلاس‌های سطح پایین یکدیگر کار می‌کنند.

Flyweight – در این الگو نیز به دلیل ارتباط ســـطح پایی Factory و Flyweight تغییرات از Flyweight به Factory منتشـر می‌شـود, اما تغییرات در Factory (در صورت تعریف اینترفیس سـطح بالا) و یا Flyweight (در حالت عادی اینترفیس دارد) به کلاینت منتشر نمی‌شود, زیرا در سطح بالا کار می‌کنند.

Item Description – در این الگو همان‌طور که قبلاً اشاره شد چون اطلاعات مشترک از داخل Item بیرون کشیده شده و در کلاس Item Description هست و این ۲ در سطح پایین با یکدیگر کار می‌کنند, تغییرات در Item Description مانند کم شدن یا تغییر نام Attribute ها به کلاس Item منتشر می‌شود.

مقایسه: هر ۳ الگو تقریباً تغییرپذیری در حد خوبی است, تغییرات به کلاینت منتشر نمی‌شود, اما مثلاً Memento و یا Item Description به ترتیب به خاطر ارتباط سطح پایین Originator و Memento و هم چنین ارتباط Item با Item Description تغییرات را منتشر می‌کنند. هم چنین در Flyweight بین خود Flyweight و Factory به خاطر ارتباط سطح پایین نیز همین‌گونه است.

پیکربندی

Memento – نیاز به پیکربند ثالث ندارد و خود این مجموعه کار پیکربندی و ذخیره‌سازی حالت‌ها را انجام می‌دهند.

Flyweight – یک پیکربند ثالث نیاز است تا حالت‌های Extrinsic را درون Context ذخیره کند.

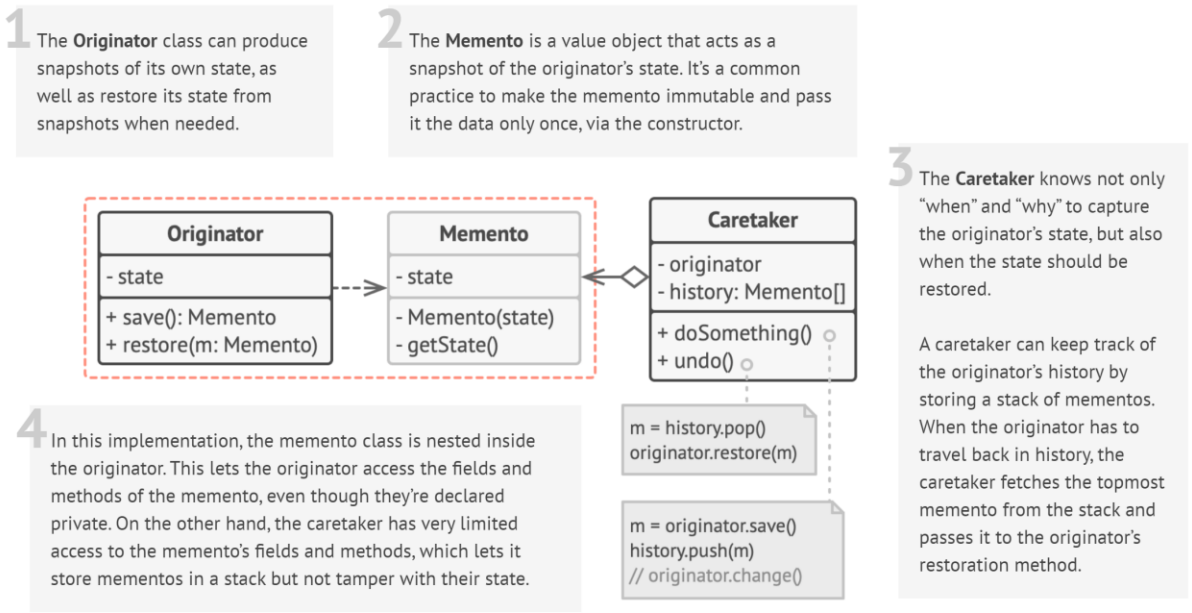
Item Description – این الگو پیکربندی خاصی نیاز ندارد و خود ساختار اصلی, هدف الگو را محقق می‌سازد.

مقایسه: همان‌طور که از ساختار مشخص است فقط Flyweight است که نیاز به پیکربند ثالث دارد. (البته بدون پیکربند خارجی هم می‌تواند از طریق Factory این کار را انجام دهد.)

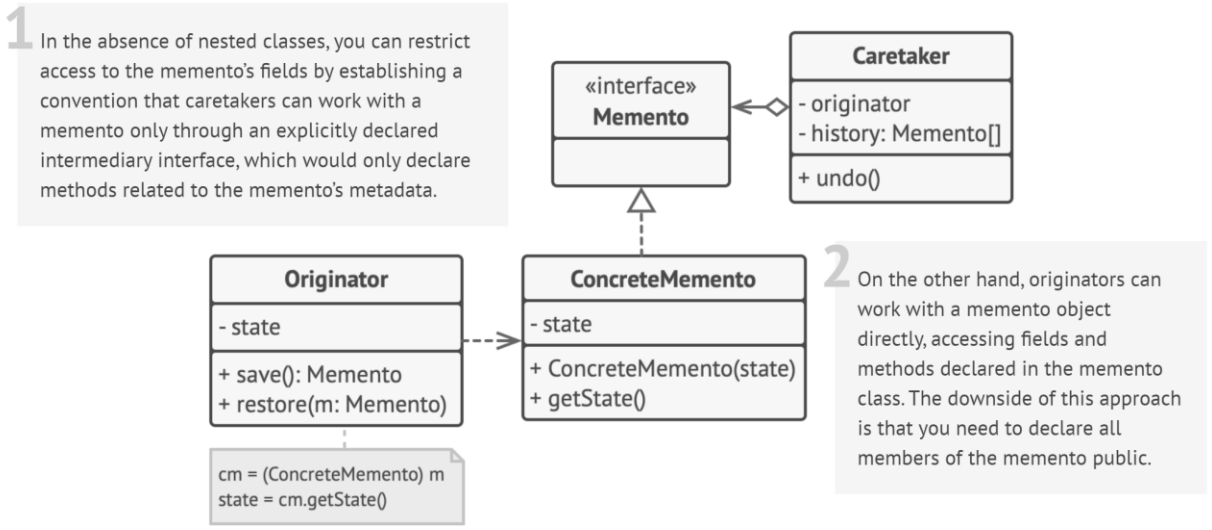
ساختار و رفتار

در این قسمت توضیحات تکمیلی در شکل موجود است.

Memento



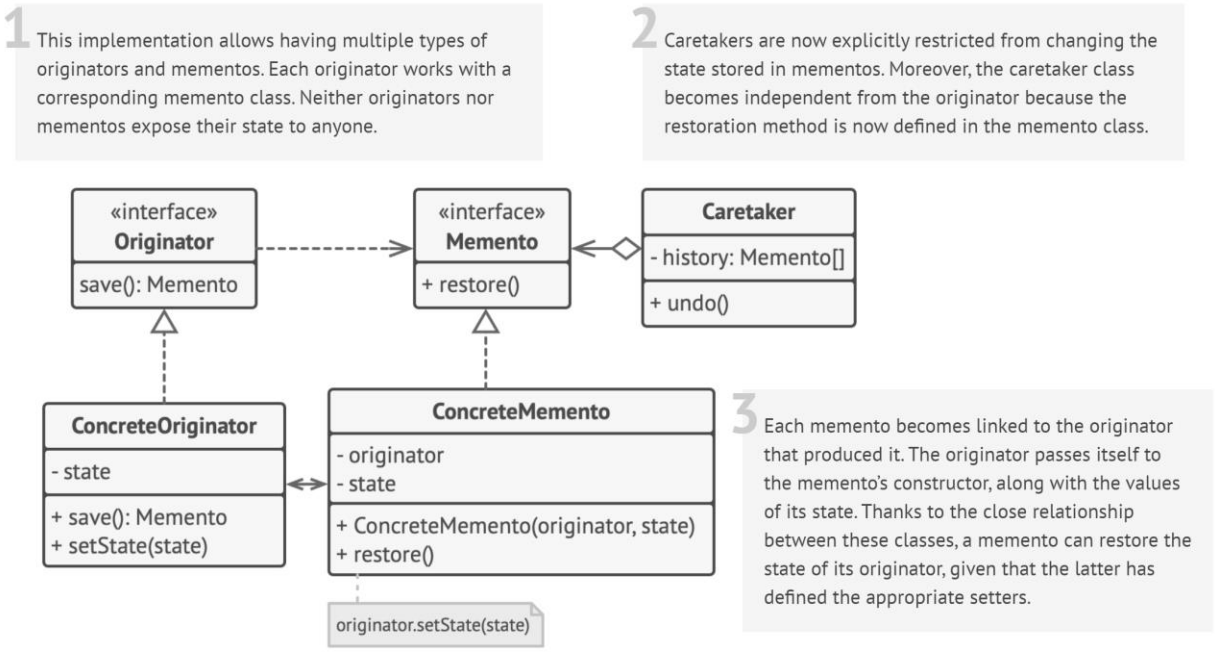
شکل ۱۰٫۱ ساختار الگوی Memento [۱۰]



شکل ۱۰٫۲ ساختار الگوی Memento با اینترفیس برای Memento(Wide & narrow) [۱۰]

در این شکل‌ها لازم است یک Association از Caretaker به سمت Originator اضافه شود.

همان‌طور که در شکل ۱۰٫۱ میبینید کلاینت که همان Caretaker هست فقط به Originator می‌گوید که الان State داخلی ات را به من بده تا ذخیره کنم یا این State را که در اختیارت میگذارم بازگردانی کن (این حالت ها را از Memento هایی که به‌صورت Aggregation داخل خود دارد به Originator می‌دهد). اما رابطه ی بین Originator و Memento باید از کلاس‌های سطح پایین باشد تا بتواند از متدهایی مثل get State آن استفاده کند (همان‌طور که در شکل ۱۰٫۲ میبینید در واقع اینترفیس‌های wide & narrow تعریف شده اند یعنی Caretaker داخل Memento را نمیبیند زیرا اینترفیسی که با آن کار می‌کند narrow است اما خود Originator آن را می‌بیند و در واقع باعث نقض Encapsulation نیز شده است.). در هنگام ذخیره‌سازی State در واقع Originator یک Memento به Caretaker می‌دهد و هم چنین هنگام بازگردانی یک Memento دریافت می‌کند.

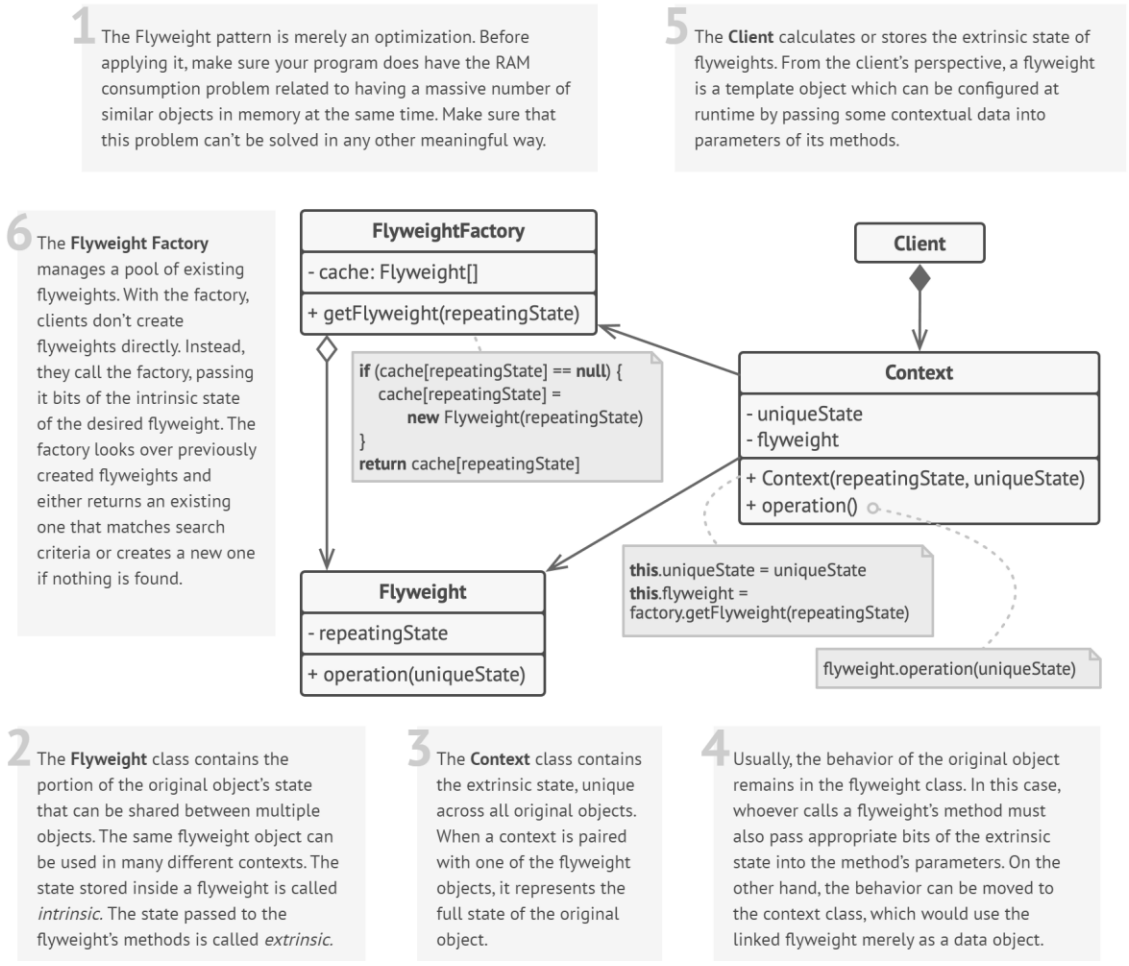


شکل ۱۰٫۳ ساختار الگوی Memento با رعایت کپسوله‌سازی [۱۰]

اما همان‌طور که در شکل بالا می‌بینید این پیاده‌سازی از این الگو باعث شده تا Encapsulation نقض نشود.

ساختار ۱۰٫۱ همانند ساختار کتاب می‌باشد (به همین دلیل دوباره به آن اشاره نمی‌کنیم) و همان‌طور که دکتر رامسین در کلاس اشاره کردند ارتباط Association بین Caretaker و Originator لازم است.

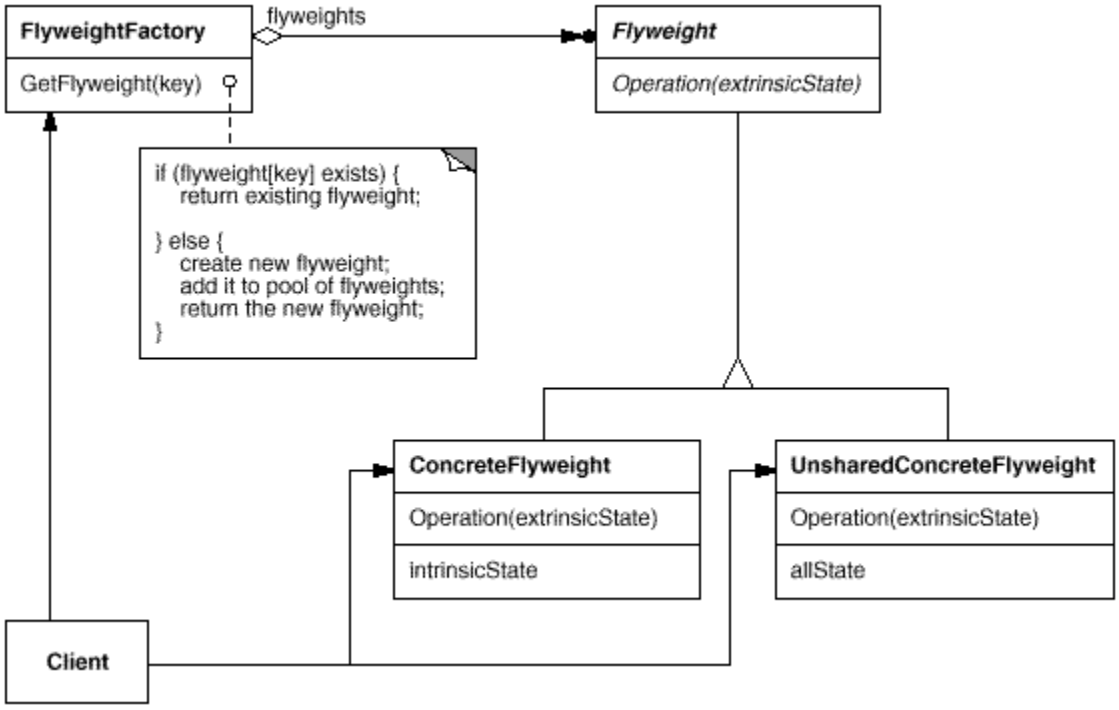
Flyweight



شکل ۱۱٫۱ ساختار الگوی Flyweight [۱۱]

همان‌طور که در شـــکل ۱۱٫۱ مشـــخص اســـت در واقع کلاینت با اســـتفاده از Context که در برگیرندهٔ Extrinsic State یا همان Unique State هســـت با اســـتفاده از Flyweight Factory که در برگیرندهٔ Flyweight Pool هســـت و داشـــتن دید مســـتقیم به Flyweight و عملیات های آن (این دید از طریق Context به وجود می‌آید.) , الگوی Flyweight را پیاده‌ســـازی کرده, و با ارســـال Unique State به متد

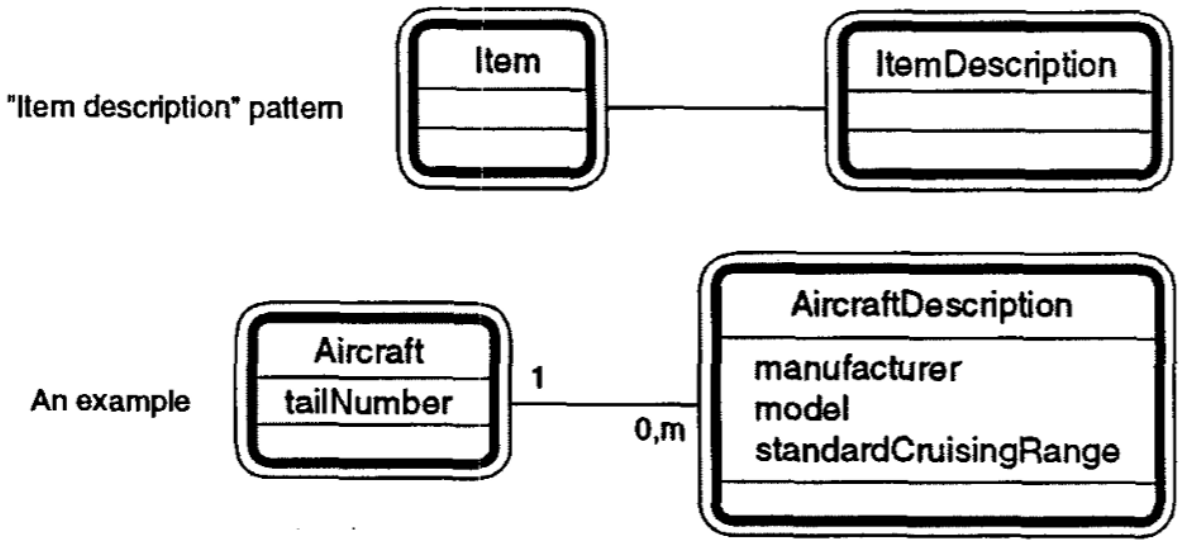
operation برای هر Flyweight, حالت ذاتی و وابســـته به Context او را ترکیب کرده و عملیات را انجام می‌دهد.



شکل ۱۱٫۲ ساختار الگوی Flyweight کتاب

این ساختار الگو در کتاب است و همان‌طور که دکتر رامسین در کلاس اشاره کردند ارتباط کلاینت با زیر کلاس‌های Flyweight نیست و در واقع با اینترفیس آن‌ها است, هم چنین Flyweight Factory ارتباط با اینترفیس آن‌ها ندارد بلکه با زیر کلاس‌های عینی Flyweight کار می‌کند.

تفاوت این ساختار با ساختار شکل ۱۱٫۱, تعریف اینترفیس برای Flyweight و همچنین تعریف کلاس جدید برای Extrinsic State به‌جای Compute کردن یا نگهداری آن در Context می‌باشد.



شکل ۱۲ ساختار الگوی Item Description

همان‌طور که از شـــکل۱۲ (Coad Yourdon Notation) پیداســـت هر Item می‌تواند یک Item Description داشـته باشـد و هر Item Description می‌تواند به‌صورت Shared بین Item ها اسـتفاده شـود, این مزیت اسـت. این الگو این‌گونه عمل می‌کند که Attribute هایی که Value های یکسـان دارد را از Item بیرون کشـــیده و در داخل Item Description می‌گذارد و این آبجکت می‌تواند در Item ها نیز Share شود.

مقایسه: به ساختار و رفتار این ۳ الگو به‌تفصیل پرداخته شد. می‌توان گفت دو الگوی Item Description و Flyweight در واقع در صـرفه‌جویی در مصـرف حافظه و هم چنین جداسـازی اطلاعاتی که تکرارشـونده هستند و Share کردن آن‌ها شباهت دارند.

کارایی از منظر حافظه

Memento – ازآنجایی‌که آبجکت‌های Memento هم به تعدد در سیستم ممکن است ثبت شوند و هم این آبجکت‌ها ممکن است بسیار سنگین باشند, می‌تواند مصرف حافظه را به‌شدت زیاد کند, پس یک مکانیزم Garbage Collection خوب برای مدیریت Memento ها نیاز داریم.

Flyweight – در این الگو با مکانیزم Sharing که Intrinsic State که وابسته به ویژگی‌های ذاتی آبجکت است, مصرف حافظه کم می‌شود.

Item Description – در حالت قبل از انجام الگو آبجکت‌ها بسیار بزرگ و Value های تکراری داشته‌اند اما پس از اعمال الگو بخش‌های مشترک به‌صورت اشتراکی درآمده‌اند و مصرف حافظه کم می‌شود. اما تعداد آبجکت‌های درون حافظه بیشتر می‌شود, زیرا قبل از اعمال الگو فقط Item ها را داریم.

مقایسه: در الگوی Memento همان‌طور که گفته شد می‌تواند باعث افزایش مصرف حافظه شود و نیاز به مکانیزم خوب برای Garbage Collection هست. اما در Item Description و Flyweight بدون شرط باعث کاهش مصرف حافظه شده است.

موارد کاربرد

Memento **(این شرط‌ها بینشان AND هست یعنی همه باهم باید وجود داشته باشند.)**

۱- وقتی یک Snapshot یا بخشی از حالت آبجکت را نیاز هست ذخیره کنیم تا بعداً بخواهیم به آن بازگردانی داشته باشیم.

۲- زمانی که یک اینترفیس مستقیم برای دسترسی به حالت روی خود آبجکت تعریف کنیم, باعث نقض بسته‌بندی و دسترسی به حالت داخلی آبجکت می‌شود.

Flyweight **(این شرط‌ها بینشان AND هست یعنی همه باهم باید وجود داشته باشند.)**

۱- یک اپلیکیشن تعداد خیلی زیادی آبجکت دارد.

۲- هزینه ذخیره‌سازی آبجکت‌ها صرفاً به دلیل تعداد زیاد آبجکت‌ها بالا است.

۳- اگر حجم Intrinsic State و Extrinsic State داریم و حجم خوبی هم دارند و هم چنین می‌توان Extrinsic را Compute کرد, در این حالت صرفه‌جویی خوبی هم داریم.

٤- گروه‌هایی از آبجکت‌ها را می‌توان با تعداد کمی از آبجکت‌ها با مکانیزم Sharing جایگزین کرد.

٥- اپلیکیشن نباید به هویت آبجکت مبتنی باشد.

Item Description

۱- زمانی که مقدارهای بعضی از Attribute ها به چندین آبجکت دیده می‌شود و تکراری هستند.

مقایسه: برای هر ۳ الگو تقریباً تمام موارد کاربرد مهم ذکر شد.

الگوهای مرتبط

Memento

۱- مورد استفاده ۳ برای الگوی Command برای این الگو نیز صادق است.

۲- مورد ۳ برای الگوی Iterator برای این الگو نیز صادق است.

۳- بعضی از مواقع می‌توان از جایگزین ساده‌تری مثل Prototype به‌جای Memento استفاده کرد. به شرطی که در حالت داخلی آبجکت لینک‌های سنگین به منابع خارجی یا آبجکت‌ها وجود نداشته باشد, یا به‌راحتی بتوان این لینک‌ها را دوباره برقرار کرد.

Flyweight

۱- می‌توان برگ‌های یک درخت Composite را به‌صورت Flyweight پیاده‌سازی کرد تا کمی از حافظه Save شود.

۲- هم چنین مورد اول در الگوی Item Description برای این الگو نیز صدق می‌کند.

Item Description

۱- این الگو مانند Flyweight از مکانیزم Sharing برای آبجکت‌ها و صـــرفه‌جویی از حافظه اســـتفاده می‌کند.

مقایسه: موارد مرتبط در بالا ذکر شد.

مزایا معایب

Memento

۱- در صورت استفاده بسته‌بندی به‌خوبی رعایت می‌شود (از سمت Caretaker بسته‌بندی به‌خوبی رعایت می‌شـــود و محتویات originator یا Memento را نمی‌بیند, ولی خود Originator محتویات داخلی Memento را باید ببیند.). این مزیت است.

۲- Originator ساده می‌شود, زیرا مسئولیت نگهداری Memento ها و ذخیره‌سازی و بازیابی آن‌ها و یا حتی Garbage Collection از او جدا شده است. این مزیت است.

۳- اســتفاده از Memento ها می‌تواند هزینه‌ها را زیاد کند چون ممکن اســت مکرراً آن‌ها تولید شــده, خود این آبجکت‌ها می‌تواند خیلی سنگین باشد. این عیب است.

٤- باید اینترفیس‌های باز و بســـته تولید کرد (narrow & wide) برای اینکه مطمئن شـــویم فقط Originator فقط Memento ها را می‌بیند. این عیب است.

٥- باید الگوریتم‌های سـختگیرانه برای Garbage Collection در Caretaker داشـته باشـیم زیرا خود Caretaker نمی‌داند هزینهٔ Memento ها چقدر اســـت و باید به‌موقع این عمل انجام شـــود. این عیب است.

Flyweight

۱- صرفه‌جویی در مصرف حافظه, مزیت است.

۲- واکشی یا محاسبه یا پیداکردن Extrinsic State باعث هزینه اضافی می‌شود, این عیب است.

۳- پیچیدگی در زمان پیاده‌سازی زیاد می‌شود. این عیب است.

Item Description

۱- صرفه‌جویی در مصرف حافظه که این مزیت است.

۲- از Sharing آبجکت‌ها می‌توان استفاده کرد که این نیز مزیت است.

۳- کلاس‌ها به هم Coupled هستند, این عیب است.

مقایسه: موارد در بالا ذکر شده است و به‌راحتی می‌توان باتوجه‌به Context مسئله و trade-off که می‌توان در نظر داشته باشیم الگوی موردنظر را انتخاب کنیم.

شیء جعلی

Memento – کلاس و آبجکت‌های Memento همه تصنعی هستند و در قلمرو مسئله نیستند.

Flyweight – آبجکت و کلاس‌های Flyweight و Flyweight Factory در قلمرو مسئله نیست و برای تحقق الگو تولید شده و جعلی است.

Item Description – تمام آبجکت‌های Item Description در قلمرو مسئله نیستند و مصنوعی هستند.

مقایسه: هر ۳ شیء جعلی تولید کرده‌اند.

جداسازی دغدغه‌ها

Memento – این الگوریتم در واقع وظیفه ذخیره و بازیابی حالت یک آبجکت به Caretaker و با استفاده از Memento ها محول شده است. این یعنی جداسازی دغدغه‌ها به‌خوبی انجام شده است.

Flyweight – در مورد این الگو زیاد نمی‌توان در مورد Separation of Concerns زیاد صحبت کرد, زیرا آبجکت‌ها قبل از اعمال الگو کار خود را انجام می‌داده‌اند, اما بعد از اعمال الگو همان کار را با کمی

صـــرفه‌جویی در مصـــرف حافظه انجام می‌دهند. ولی وظیفه جســـتجو در آبجکت‌ها و نمونه‌ســـازی از Flyweight (در صـــورت عدم وجود آن) به عهده Flyweight Factory اســـت و این وظیفهٔ جســـتجو و نمونه‌سازی قبلاً به Context محول شده بوده است.

Item Description – همان‌طور که قبلاً اشاره شد حالت قبل از اعمال الگو همانند آن است که دو کلاس ترکیب شده و به‌تبع آن Attribute Value های تکراری داریم, می‌توان گفت بعد از اعمال الگو با جداسازی این خصیصه‌ها, دغدغه‌های دو کلاسی که ترکیب شده بودند نیز جداسازی می‌شود.

مقایسه: الگوی Memento و Item Description به‌خوبی این کار را انجام داده‌اند, اما در مورد Flyweight همان‌طور که اشاره شد نمی‌توان نظر دقیقی داد.

پیاده‌سازی

Memento – ابتدا کلاسی که قرار است حالت‌های آن ذخیره شود را تعیین می‌کنیم.

کلاس Memento را با فیلدهای کلاس مذکور در بالا می‌ســـازیم. (بهتر اســـت این کلاس به‌صـــورت Immutable باشد و فقط State را از Constructor بگیرد و set State نداشته باشد.)

می‌توان اینترفیس‌های Wide & Narrow برای Memento ایجاد کرد تا متدهایی مثل get State را فقط در اختیار Originator قرار دهیم.

متدی برای تولید Memento توسط Originatorمی‌سازیم که State خود را به متد سازندهٔ Memento می‌دهد.

سپس Caretaker را با یک Reference به Originator و لیستی از Memento هامی‌سازیم.

Attribute – Flyweight های کلاس را به دو دســـته که یکی ذاتی (Intrinsic) و دیگری بســـته به Context است (Extrinsic) تقسیم می‌کنیم.

Intrinsic State را داخل کلاس می‌گذاریم اما توجه داشــته باشــیم که بهتر اســت به‌صورت Immutable باشد و فقط از طریق Constructor مقداردهی شوند.

یک کلاس Flyweight Factoryمی‌ســازدیم که به‌اصطلاح یک Flyweight Pool دارد و عملیات جســتجو و Instantiation را در صورت نبود Flyweight قبلی, را انجام می‌دهد.

بعد از کلاینت برای کار با آبجکت‌های ریزدانه فقط از طریق Factory این کار را انجام می‌دهد.

Item Description – ابتدا Attribute Value های تکراری را شناسایی می‌کنیم.

آن‌ها را در یک کلاس دیگر می‌گذاریم و آن را Item Description می‌نامیم.

سپس در کلاس اصلی Item این خصیصه‌ها را پاک کرده و سپس یک Reference به Item Description در این کلاس قرار می‌دهیم. (Aggregation)

مقایســـه: پیاده‌ســـازی Memento انواع مختلفی دارد و برای حالتی که بخواهیم اینترفیس‌های بســـته و گســترده بســازیم کمی پیچیدگی وجود دارد چون باید داخل Memento در واقع کلاس Originator ای که آن Memento را هم ســاخته پاس بدهیم. پیاده‌ســازی Flyweight هم در بحث‌های جداســازی حالت‌های Intrinsic و Extrinsic و مکانیزم جســتجویی که در Factory پیاده‌ســـازی می‌شـــود, پیچیدگی‌های خاص خودش را دارد, اما از نظر ســـاختار کلاســـی و پیاده‌ســـازی پیچیدگی متوســـطی دارد. پیاده‌ســـازی Item Description بسیار ساده است.

بررســـی الگوهای **GRASP** برای ۳ الگوی **Memento, Flyweight** و **Item Description**

Information Expert

Memento – ازآنجایی‌که Memento ها از دل Originator بیرون کشـیده شـده‌اند می‌توان گفت این عمل باعث نقض این الگو می‌شـود چون داده و رفتار در کنار یکدیگر نیسـتند (یک نظر دیگر هسـت که Memento ها نه عملیات داخلی Originator هســـتند و نه حالت داخلی بلکه کلاس‌هایی برای ذخیره حالت یک آبجکت بدون نقض Encapsulation هستند, پس این الگو برقرار است).

هم چنین از سمت Caretaker این الگو برقرار است.

Flyweight – همانند Item Description که در پایین به آن اشـاره شـده اسـت چون قسـمتی از داده که Extrinsic State نامیده می‌شود, از آن جدا شده است, این الگو را نقض کرده است.

Item Description – در این الگو Information Expert نقض شـده اسـت زیرا قسـمتی از داده‌ها از رفتار آبجکت اصلی که آن را Item نامیده‌ایم, جدا شده است.

Creator

Memento – این الگو نقض شـده اسـت زیرا با اولویت ۵ این کار انجام می‌شـود یعنی Originator حالت‌ها را در داخل Memento ها می‌گذارد و آن‌ها را می‌سـازد, اما اولویت ۲ برقرار اسـت زیرا Caretaker به‌صورت Aggregation آبجکت‌های Memento را نگه می‌دارد.

Flyweight – این الگو چون آبجکت Flyweight Factory آبجکت‌های Flyweight را به‌صورت Aggregation دارد (اولویت ۲) باید آن‌ها را بسازد و اولویت بالاتری وجود ندارد, پس این الگو رعایت شده است.

Item Description – ازآنجایی‌که به‌صورت کلی یک Item می‌تواند Description داشته باشد یا نه, و تعریف Aggregation هم می‌توان به‌صورت یک رابطه has-a تعریف کرد و هر آیتم یک یا چند Item Description را به‌صورت Aggregate در داخل خود دارد, می‌توان گفت اولویت ۲ برقرار است, اما در این الگو صحبتی از آن به میان نیامده است.

Coupling

Memento – با رجوع به قسمت کاهش وابستگی برای این الگو متوجه می‌شویم که از سمت Caretaker و Memento کم شده اما در طرف دیگر زیرا ارتباط Memento و Originator در سطح پایین هست به‌شدت Coupled هستند.

Flyweight – با رجوع به قسمت کاهش وابستگی برای این الگو متوجه می‌شویم که تأثیر خوبی برای این الگوی Grasp دارد و Coupling را کم می‌کند.

Item Description – با رجوع به قسمت کاهش وابستگی برای این الگو متوجه می‌شویم که Coupling افزایش‌یافته است.

Cohesion

Memento – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و تقریباً Cohesion به حد خوبی می‌رساند.

Flyweight – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر خوبی برای این الگوی Grasp دارد و تقریباً Cohesion به حد خوبی می‌رساند.

Item Description – با رجوع به قسمت افزایش چسبندگی برای این الگو متوجه می‌شویم که تأثیر زیادی برای این الگوی Grasp دارد و آن را محقق می‌سازد.

Controller

Memento – این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Flyweight – این الگو در واقع کارچرخانی ندارد و Controller ندارد و بیشتر برای صرفه‌جویی در مصرف حافظه نوشته شده است.

Item Description – این الگو در واقع کارچرخانی ندارد و Controller ندارد.

Polymorphism

Memento – همان‌طور که در پیاده‌سازی سوم از این الگو در بخش ساختار و رفتار صحبت شد, می‌توان دو اینترفیس Wide و Narrow را پیاده‌سازی کرد تا ارتباط بین Originator و Memento نیز در سطح اینترفیس و هم چنین Caretaker و Memento در سطح اینترفیس باشد و هیچ آبجکتی حالت داخلی آبجکت دیگر را نبیند, مزیت دیگر این کار این هست که آبجکت‌ها فقط با اینترفیس سطح بالای یکدیگر کار می‌کنند (مگر موقع نمونه‌سازی از Memento) که باعث می‌شود چندریختی در مورد زیر کلاس‌های Originator و Memento داشته باشیم و این الگو رعایت شود.

Flyweight – این الگو با داشـــتن اینترفیس ســـطح بالا برای آبجکت Flyweight و کار کردن کلاینت و Factory با آن این الگو را رعایت کرده اســـت. اما درون Factory ما نیاز به دید ســـطح پایین به کلاس‌های Concrete Flyweight برای Instantiation داریم, در نتیجه در این قسمت چندریختی برقرار نیست.

Item Description – در مورد این الگو نمی‌توان زیاد راجع به چندریختی صـحبت کرد زیرا از اینترفیس‌های سـطح بالا اسـتفاده نمی‌کند, مگر آنکه برای Item Description همان‌طور که در قسـمت LSP مربوط به این الگو اشاره شد, ساختار توارثی در نظر بگیریم.

Indirection

Memento – به‌وضـوح قابل‌مشـاهده اســت که Indirection برقرار اســت زیرا به‌صـورت مســتقیم از Originator حالت آن را ذخیره نمی‌کنیم و این کار توسـط درخواسـتی که از کلاینت آمده و یا Caretaker دستور آن را داده, با کمک آبجکت‌های Memento انجام می‌پذیرد.

Flyweight – همان‌طور که در ساختار دیده شد و قبلاً توضیح داده شد ازآنجایی‌که برای ارتباط بین کلاینت و Flyweight یک اینترفیس سطح بالا نوشته شده است, یک سطح از Indirection اضافه شده است, اما درون Factory چون ارتباط در سـطح پایین و به‌صـورت مسـتقیم به Flyweight وجود دارد, Indirection دیده نمی‌شود.

Item Description – به‌صـورت کلی باتوجه‌به هدف الگو نمی‌توان صـبحت خاصـی کرد اما اگر یک کلاینت بخواهد به اطلاعاتی که به‌صـورت مشـترک این Item با بقیه Item ها دارد دسـترسـی پیدا کند, این کار به‌صورت غیرمستقیم و از طریق Item Description انجام می‌پذیرد.

Pure Fabrication

Memento – تمام کلاس‌های Memento جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Flyweight – تمام کلاس‌های Flyweight و Flyweight Factory نیز جعلی هستند و برای راحتی حل مسئله تولید شده‌اند.

Item Description – کلاس‌های Item Description برای راحتی حل مسئله تولید شده‌اند.

Protected Variations

Memento – همان‌طور که در بحث تغییرپذیری گفته شد ارتباط بین Memento و Originator در سطح پایین یا در سطح اینترفیس گسترده است و این دو بسیار به یکدیگر Coupled شده‌اند پس تغییرات به‌راحتی به آن‌ها منتشر می‌شود, اما ازآنجایی‌که Caretaker به‌صورت غیرمستقیم و با اینترفیس بسته با Memento کار می‌کند (در نتیجه به‌صورت غیرمستقیم با Originator کار می‌کند) , تغییرات به او منتشر نمی‌شود.

Flyweight – همان‌طور که در بحث تغییرپذیری گفته شد ارتباط بین آبجکت Factory و Flyweight در سطح پایین و Concrete است و تغییراتی که در Flyweight صورت پذیرد, به Factory منتشر می‌شود. اما کلاینت و Flyweight چون در سطح بالا و اینترفیس کار می‌کند, تغییرات به‌جایی منتشر نمی‌شود.

Item Description – همان‌طور که در بحث ساختار دیدیم در این الگو Item در سطح پایین و به‌صورت مستقیم به حالت داخلی Item Description دید دارد و با آن کار می‌کند پس تغییرات بین آن‌ها منتشر

می‌شـود. یعنی مثلاً نام یک فیلد مشـترک که در Item Description اسـت, تغییر کند این تغییر به Item منتشـر می‌شـود.

منابع

١. Refactoring. (٢٠٢٢). *Command Design Pattern*. Retrieved from Refactoring guru: https://refactoring.guru/design-patterns/command
٢. Refactoring. (٢٠٢٢). *Proxy design pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/proxy
٣. Refactoring. (٢٠٢٢). *Mediator Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/mediator
٤. Refactoring. (٢٠٢٢). *State Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/state
٥. Refactoring. (٢٠٢٢). *Strategy Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/strategy
٦. Refactoring. (٢٠٢٢). *Visitor Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/visitor
٧. Refactoring. (٢٠٢٢). *Abstract Factory Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/abstract-factory
٨. Refactoring. (٢٠٢٢). *Iterator Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/iterator
٩. Refactoring. (٢٠٢٢). *Builder Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/builder
١٠. Refactoring. (٢٠٢٢). *Memento Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/Memento
١١. Refactoring. (٢٠٢٢). *Flyweight Design Pattern*. Retrieved from Refactoring: https://refactoring.guru/design-patterns/flyweight