بسمه تعالی

محمدرضا سابقی ۹۹۲۰۱۴۲۱

الگوها در مهندسی نرم افزار

استاد: دکتر رامسین

تمرین سوم

تابستان ۱۴۰۰

فهرست مطالب

۱ موقعیت اول

۱.۱ بررسی اجمالی موقعیت

موقعیت در این سوال بدین گونه تعریف شده است که:
تعدادی operation در یک کلاس، وابستگی قوی ای به attribute ها و operation های کلاس دیگر دارند. این موقعیتی است که در کد بوجود آمده است و حال قصد داریم به کمک دو الگوی Hide Delegate و Inline Class به موقعیت نگاه کرده و طبق شرایطی که باید حاصل باشد، به حل موقعیت بپردازیم.

۲.۱ بررسی الگوی اول

الگوی Hide Delegate برای زمانی کاربرد دارد که شاهد نقض PLK هستیم. یعنی زنجیره دید ترایا وجود دارد و ما قصد داریم که دوباره اصل PLK را برگردانیم و درواقع message chain را از میان برداریم. اگر در نظر بگیریم که کلاس A (بعنوان کلاینت) از آبجکت کلاس B درخواستی داشته باشد و کلاس B در جواب، آبجکت کلاس C را برای کلاینت ارسال کند، آنگاه دید ترایا بوجود آمده است که مخالف اصول شیءگرایی است. این الگو پیشنهاد می دهد که کلاس B را سرور قرار داده و متدهایی در آن ایجاد کنیم که درصورت نیاز، واسپاری را به C داشته باشند، به جای آنکه آن را برگردانند. بدین ترتیب دید A بعنوان کلاینت، از C برداشته می شود و به نوعی آن را Hide یا مخفی کرده ایم. پس از اعمال راه حل الگو، از این به بعد هر درخواستی از کلاینت A به آبجکت B صرفا توسط متدهای ساخته شده در B ، به آبجکت C ارجاع داده می شود، بجای آنکه خود آبجکت C برگشت داده شده و در اختیار کلاینت قرار بگیرد و کلاینت روی حالت داخلی آن دید داشته باشد.

۳.۱ شرایط مفید بودن الگوی اول در موقعیت

در این موقعیت که وابستگی شدیدی روی attribute و operation بین دو کلاس موجود است و یکی قویا به متد و داده های دیگری نیاز دارد، زمانی الگوی Hide Delegate می تواند مفید باشد که این وابستگی از جنس زنجیره ی دید ترایا بوده و زنجیره ی پیغام (message chain) بین آبجکت های کلاس دیده شود. پس شرایطی را در نظر می گیریم که کلاس A عملیات و رفتاری دارد که در اجرای آن ها نیاز زیادی به attribute و operation های کلاسی مانند C دارد. برای رسیدن به کلاس C نیز درخواست خود را از طریق زنجیره پیام، ابتدا به B می فرستد و B هم آبجکت C را در جواب می فرستد. اگر در شرایط، کلاس B درخواست را به گونه ای پاسخ دهد که در جواب، آبجکتی از کلاس نوع سوم مانند C بفرستد(چون A شدیدا به آن نیاز دارد و وابسته ی آن است)، آنگاه می توانیم از فایده ی الگوی Hide Delegate صحبت کنیم. بدین گونه که کلاس A در نقش کلاینت برای کلاس B بوده و مدام از رفتارهای آن، آبجکت C را می گیرد تا عملکرد خود را تکمیل کند(چرا که شدیدا به رفتار و داده های آن وابسته است). اگر B دید C را برای A برقرار کند، در این شرایط، الگوی ذکر شده مفید خواهد بود. اینکه چگونه مفید خواهد بود را در قسمت بعد خواهیم گفت.

۴.۱ چگونگی مفید بودن الگوی اول در موقعیت

بعد از آنکه شرایط را برای اعمال الگو بیان کرده ایم، حال قصد داریم بیان کنیم در آن شرایط، الگو چه گونه می تواند موثر باشد. الگو پیشنهاد می دهد که کلاس B را سروری قرار دهیم که کلاس وابسته ی A که در نقش کلاینت بود، درخواست خود را به B فرستاده و B به جای برگرداندن آبجکتی از C و قرار دادن آن در معرض دید A ،کار را در صورت لزوم به C واسپاری کند.

دقت داریم که برای تحقق این کار، لازم است تا متدهایی برای B ساخته شود و بعنوان رفتار در آن تعبیه شود که در آن سرویس را اینگونه به A دهد که کار را به C واسپاری کند، نه اینکه C صرفا برگشت داده شود(کاری که قبلا در متدها می شد) . بدین صورت A از شدت وابستگی به C بی نیاز می شود و فقط سرور خود یعنی B را می شناسد و درخواست خود را به آن می دهد. اینگونه رفتارهایی را به B و C منتقل کردیم که نشان می دهد رفتار به داده ی خود نزدیک شده است. بدین شیوه، message chain از میان می رود و PLK برقرار می شود و از همه مهمتر آنکه، وابستگی در A به C کم می شود که دقیقا منطبق بر موقعیت است.

۵.۱ معایب اعمال الگوی اول

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- سربار اجرایی به دلیل delegation های مکرر که در متدهای ایجاد شده در سرور مشهود است، سبب می شود تا پردازش بیشتری برای یک درخواست کلاینت، مصرف شود.
- سربار زمانی که در اثر طول کشیدن زمان اجرای یک درخواست کلاینت به دلیل واسپاری درخواست از سرور به Delegate Class
- زمانی که صرف می شود تا متدهای delegation در سرور نوشته شوند و مورد آزمون قرار بگیرند.
- احتمال رخدادن Inappropriate Intimacy اگر دقت کافی در واسپاری انجام نشود و نتوان آن وابستگی قبلی را کاهش داد و به همان میزان، وابستگی را بالا برد.

۶.۱ مزایای اعمال الگوی اول

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- کاهش متناسب وابستگی یک کلاس به داده ها و رفتارهای کلاس دیگری که با واسطه به آن ها می رسد. دقت داریم که کلاس C از دید A مخفی می شود که نکته ی مثبتی برای کاهش وابستگی است.
- انتقال یافتن رفتارها در نزدیکی داده هایی که برای اجرا به آن ها نیاز دارند. این کار کمک می کند تا کلاس های همبسته تری در مجموعه داشته باشیم که منطبق بر اصل ایجاد کپسول های داده رفتار می باشد.
- از بین رفتن دید ترایا که یکی از موارد مهم برای بخش نگهداری یا maintainability می باشد. دید ترایا، نقض اصول شیء گرایی است که وجود آن در سیستم سبب مرگ تدریجی آن می شود.
- تغییر در کلاس C سبب انتشار محدود به B می شود که در نتیجه، A بی نیاز از دانستن تغییر خواهد بود مادامی که B دستخوش تغییر نشود.

۷.۱ بررسی الگوی دوم

الگوی Inline Class را زمانی به کار می بریم که ارتباطی تنگاتنگ از یک کلاسی مانند A به کلاس دیگری مانند B موجود است و در این میان، کلاس A به اصطلاح lazy می باشد. بدین معنا که مجموعه رفتار و ساختار آن بسیار سبک می نماید و توجیه وجودی به عنوان یک کلاس مستقل برای آن وجود ندارد. الگو می گوید که این دو کلاس باید با یکدیگر ادغام شوند. بدین صورت که تمام ساختار کلاس A را به کلاس B منتقل کنیم و سپس کلاس A را حذف کنیم. بدین ترتیب حال فقط یک کلاس خواهیم داشت(کلاس B که درنتیجه ی ادغام، کامل شد) که وابستگی قبلی دیگر پیدا نیست. کلاس A معمولا اینگونه بوجود می آید که از قبل در سیستم هویت مشخص و مستقلی را داشته اما در اثر refactoring بخش های مختلف از داخل آن جدا شدند، به مرور تبدیل به کلاس کوچکی شده و به وضعی رسیده که هیچ توجیهی برای وجود به عنوان یک کلاس مستقل و هویت دار ندارد. حال در صورت صلاحدید می توان آن را به کلاس دیگری که وابستگی بیشتری به آن دارد منتقل کرد؛ به شرطی که همبستگی و cohesion قربانی نشود. اطمینان از این موضوع بسیار مهم است.

۸.۱ شرایط مفید بودن الگوی دوم در موقعیت

موقعیتی را که در بخش اول توضیح دادیم دوباره مدظر می گیریم. وضعیت به گونه ای است که وابستگی از یک کلاس به داده/رفتار های کلاس دیگر بسیار بالاست. یعنی برای اجرای متدها در یک کلاس، لزوم دسترسی به داده یا رفتار در کلاس دیگر مورد نیاز است. حال اگر بخواهیم این الگو را در چنین موقعیتی اعمال کنیم، در شرایطی می توان از فایده ی آن گفت که کلاس وابسته، به صورت lazy وجود داشته باشد. بدین معنا که ساختار و رفتاری بسیار ساده دارد و هیچ توجیه وجودی برای آن(مثلا پس از refactoring های قبلی) در

نظر گرفته نمی شود. شرایط به گونه ای است که در این موقعیت، کلاسی غیرمستقل و وابسته که اساسا به کلاس دیگری برای بروز عملکرد خود نیاز دارد، باید ساختار و همچنین رفتار درونی ساده ای داشته باشد تا بتوان به فایده ی الگوی Inline Class در موقعیت فکر کرد. مثلا کلاس مذکور، حاوی یک یا دو feature و همچنین متدهای کم با خطوط کد کوتاه و قابل فهم که گاها کارها را نیز واسپاری می کنند. اگر چنین شرایطی وجود داشت، می توان مفید بودن الگوی Inline Class را محرز دانست و آن را اعمال کرد. راه حل آن برای چگونگی اثرگذاری و فایده ی خود، در قسمت بعد مطرح می شود، هرچند ساده و پیش پا افتاده به نظر می رسد. نکته ای در مورد دقت به پایدار ماندن cohesion وجود دارد که در قسمت بعد به آن می پردازیم.

۹.۱ چگونگی مفید بودن الگوی دوم در موقعیت

اینکه چگونه این الگو می تواند در موقعیت بیان شده مفید باشد، پس از بررسی شرایط مهیا بودن اعمال الگو بحث می شود. اگر تمام شرایط که در قسمت قبلی بیان شد، برقرار بود، الگو پیشنهاد می دهد که کلاس lazy را به کلاسی که وابسته ی آن است، منتقل کنیم. بدین صورت که همه ی آنچه از feature ها و operation های موجود در کلاس وابسته را به کلاسی منتقل کنیم که وابستگی شدید به آن داشت. با اعمال این الگو وابستگی ای که قبلا اجرای هر رفتار کلاس lazy به کلاس مستقل داشت از بین رفته و coupling در حد مطلوبی کم می شود. فقط ذکر این نکته خالی از لطف نیست که در نحوه ی اعمال الگو، خیلی باید دقت داشته باشیم که کلاس ها از cohesive بودن نیافتند. مثل کلاسی که قرار است به آن، انتقال داده و رفتار داده باشیم؛ خیلی اهمیت دارد که یا در هنگام بررسی شرایط و یا به هنگام اعمال الگو، هیچ گونه کاهش همبستگی در کلاس مستقل که پیش از این در حد خوبی وجود داشت، رخ ندهد. اگر این مورد نیز رعایت شد، آنگاه می توانیم بگوییم که Class

Inline اثر مطلوب و فایده ی خود را نشان داده است. مزایای آن را در قسمت های دیگر، موردی تر بررسی خواهیم کرد.

۱۰.۱ معایب اعمال الگوی دوم

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- احتمال خراب شدن همبستگی مجموعه به گونه ای که اگر دقت نکنیم، cohesion لطمه ی سنگینی ممکن است ببیند؛ چرا که دو کلاس در حال ادغام با یکدیگر که قبل از آن هر کدام دلیل وجودی مستقلی داشت.
- حوصله سر بر بودن کار انتقال ویژگی ها و رفتارها از کلاسی به کلاس دیگر ضمن رعایت انسجام و به هم نخوردن استایل کد
- پتانسیل بروز کد متد تکراری که در اثر انتقال متد ها بوجود آمده است. مثل کدی که قبلا در کلاس lazy فقط واسپاری می کرده است و حال دیگر به آن نیاز نیست و در عین حال بر اثر بی دقتی به کلاس مستقل، منتقل شده است.

۱۱.۱ مزایای اعمال الگوی دوم

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- یک کلاس lazy که برای نگهداری هزینه بر هست(مخصوصا از نظر زمانی) کنار گذاشته می شود.
- انتقال پیغام های بین دو کلاس از حالت قبلی که شدید بود، ازبین رفته است.

- وابستگی زیادی که بین دو کلاس بوده است، با اعمال الگو، از میان می رود.
- کاهش تعداد کلاس ها که سبب کاهش پیچیدگی کد و مدل ها می شود. همچنین کار را برای maintenance راحتتر می کند(از نظر زمان و انرژی صرف شده).
- از میان رفتن بروز احتمالی کلاس های موسوم به Data Class

۱۲.۱ مقایسه شرایط و نحوه اعمال الگوها در موقعیت

در مقایسه ای اجمالی می توان گفت که هر دو الگو به صورت کارآ می توانند به موقعیت کمک کنند اما Inline Class در صورتی مفید است که کلاس وابسته ی مجموعه، خیلی سنگین نبوده و توجیهی برای وجود ندارد. پس می توان با ترکیب شدن با کلاس مستقل، از بروز وابستگی زیاد جلوگیری کرد.

در حالی که Hide Delegate در صورتی اثر خود را می گذارد که وابستگی به شکل واسط برقرار بوده و با زنجیره ی دید ترایا، کلاس وابسته، به داده ی دور افتاده ی خود می رسد. این شدت وابستگی از طریق ساخت متدهای واسپاری در واسط های زنجیره ی دید، برطرف خواهد شد و کلاس وابسته، تنها با سرور خود ارتباط خواهد داشت.

۲ موقعیت دوم

۱.۲ بررسی اجمالی موقعیت

موقعیت در این سوال بدین گونه تعریف شده است که:

تغییر یا گسترش بخشی از کد سبب آن می شود که بقیه ی بخش های کد دستخوش تغییر بشوند که از آن با عنوان change propagation نیز یاد می شود. حال قصد داریم از زاویه دید دو الگوی Change Value to Reference و Remove Middle Man به این موقعیت نگاه کنیم و بسته به شرایطی که باید برقرار باشد به حل آن بپردازیم.

۲.۲ بررسی الگوی اول

الگوی Change Value to Reference زمانی کاربرد دارد که کپی های تغییرناپذیر از یک آبجکت یا ساختار داده ای در سیستم موجود است که توسط بخش های مختلف مورد استفاده قرار می گیرند؛ اما حال نیاز هست تا درنتیجه ی تکامل سیستم، این کپی ها آپدیت شوند و کاربرانی که از آن ها استفاده می کنند لازم هست تا بتوانند آن ها را آپدیت کنند. بدیهی است با آپدیت شدن یکی از آن ها، بقیه نیز باید متعاقب همین تغییر، آپدیت شوند تا بتوان سازگاری آن ها را با یکدیگر تامین کرد. این مشکلی است که الگو سعی در رفع آن دارد تا از بروز این تغییر منتشرشونده جلوگیری کند. کاری که انجام می شود این است که از ارجاع آن آبجکت یا ساختار داده ای استفاده می کنیم به جای اینکه از value آن ها استفاده کنیم. همه ی آن ها را به reference تبدیل می کنیم و دسترسی مستقیم به خود آبجکت اصلی بوسیله ی reference برقرار می شود. وقتی value کنار گذاشته شود و خود آبجکت اصلی دراختیار گذاشته شود، آنگاه هر تغییری روی آن، برای همه ارجاعات، تغییریافته می شود. پس

مشکل اصلی انتشار تغییرات ناشی از آپدیت کپی ها، مرتفع می شود.

۳.۲ شرایط مفید بودن الگوی اول در موقعیت

شرایطی را در نظر می گیریم که مجموعه زیادی از کپی های تغییرناپذیر یک آبجکت یا یک ساختار داده ای مانند یک Customer موجود هستند که براساس تغییرات وارد شده به سیستم، باید بروزرسانی شوند. در چنین شرایطی می توان از فایده ای الگو Change Value to Reference سخن به میان آورد. وقتی آپدیت شدن مقدار آبجکت یا ساختار داده ای Customer مستلزم این هست که تغییرات منتشرشونده روی همه کپی ها داشته باشیم تا همه ی آن ها را نیز به مقدار جدید ببریم(مقداردهی جدید داشته باشیم)، پس این مشکلی است که ما را به سمت این الگو می کشاند. اینکه چه گونه می تواند مفید باشد در قسمت بعدی تدقیق خواهد شد اما آگاهی از شرایط قبل از اعمال بسیار مهم است. پس درنهایت جمع بندی می کنیم اگر موقعیت داده شده که تغییر در بخشی از کد، انتشار تغییرات را بدنبال دارد، از جنس این هست که مقدار یک آبجکت اصلی تغییر می کند و به تبع آن، مقدار در کپی های تغییرناپذیر از آن باید دستخوش تغییر شوند، این الگو در چنین شرایطی بسیار کاربردی است. دقت داریم که ما شرایطی را در موقعیت در نظر گرفتیم که مثلا از یک Customer آبجکتی با مشخصات:

علی محمدی، تهران، ۲۲ سال

داریم که کپی های متعددی از این آبجکت که هرکدام ارجاعات متفاوت خود را به حافظه دارند، موجود است. حال تغییر در قسمتی از آن در داده ی اصلی مثلا با مشخصات زیر:

علی محمدی، تهران، ۲۳ سال

سبب می شود که همه ی کپی ها در هر بخش دیگری از کد، این تغییر را اعمال کنند. لذا شرایط اینگونه با موقعیت باید برقرار باشد تا بتوان پی به فایده ی الگو برد.

۴.۲ چگونگی مفید بودن الگوی اول در موقعیت

اگر دقت داشته باشیم متوجه هستیم که زمانی که شرایط اعمال الگو در موقعیت کشف شد، به راحتی می توان value را به reference تغییر داد. بدین ترتیب باید داده های کپی شده را به یک تک‌ـارجاع آدرس دهی کرد. از این طریق می توان گفت وقتی آبجکت جدیدی در حافظه نباید بوجود بیاید و نیازی به نگهداری مقادیر برای آن نیست(تا تغییر منتشرشونده نداشته باشیم)، پس ترجیح آن هست که از reference استفاده کنیم و در همه جا و همه ی قسمت های کد، به آن داده ی اصلی، ارجاع بدهیم. بدین سبب با اعمال این گام های الگو، هر تغییری در داده های آبجکت بوجود بیاید، چون همه جا به همین مقادیر، ارجاع داده شده است، پس نگرانی از بابت یکپارچه نبودن مقادیر در جاهای مختلف وجود نخواهد داشت و از طرفی تغییر منتشرشونده برای آپدیت کردن مقادیر، حذف می شود. خلاصه آنکه از موقعیت دور می شویم و با این الگو دیگر احتیاجی به تغییر بخش های مختلف ناشی از تغییر یک بخش کد نیست. در واقع اینگونه به فایده ی الگوی Change Value to Reference رسیدیم و از change propagation جلوگیری کردیم.

۵.۲ معایب اعمال الگوی اول

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- احتمال بی دقتی در مرجع سازی همه ی کپی های موجود و از دست دادن یکپارچگی حالت آبجکت های یکسان

- خطا در فهم و برداشت از ارجاع به یک آبجکت و مقدار یک آبجکت
- فراموشی از اینکه چرا و درنتیجه چه چیزی این الگو اعمال شد و کدام آبجکت را به کمک reference ارجاع دادیم، صرف اینکه شاید بخواهیم الگوی ضد آن را عمل کنیم.
- اشتباه در آپدیت کردن و مقادیر نادرست دادن به آبجکت اصلی، سبب انتشار آنی تغییر می شود که می تواند آپدیتی خطرناک باشد و پیغامی را در سیستم فعال کند. مثلا به مشتری، تخفیفی داده شود که مستحق آن نبوده و فقط در اثر یک اشتباه سهوی، این آپدیت برای همه ی ارجاعات به آن مشتری رفته است.

۶.۲ مزایای اعمال الگوی اول

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- جلوگیری از تغییرات منتشرشونده، مهم ترین مزیت آن است.
- حفظ یکپارچگی داده و عدم رخداد تناقض برای یک داده به هنگام آپدیت کردن داده ی اصلی
- کاهش سربار حافظه ای در سیستم از این جهت که تمام کپی ها از روی حافظه برداشته می شود.
- افزایش بازدهی در بهبود سرعت آپدیت شدن

۷.۲ بررسی الگوی دوم

الگوی Remove Middle Man درواقع عکس الگوی Hide Delegate می باشد. پس در اصل این الگو زمانی کاربرد دارد که سرور در وسط تبدیل به Middle Man می شود. یعنی آنکه همه ی تغییراتی که در حال رخدادن در کلاس Delegate هست(کلاسی که سرور، درخواست را به آن واسپاری می کند) به خود کلاس سرور نیز منتشر می شود تا کلاینت بتواند از تغییرات داده شده، استفاده ی لازمه را بکند. دقت داریم که تا الان روابط اینگونه بود که کلاینت درخواستی را با واسطه ی سرور، از کلاس Delegate می گیرد. در واقع کلاینت اصلا دیدی به کلاس Delegate ندارد و نمی داند که این کلاس در حقیقت، درخواست را جواب می دهد(دید آن فقط به کلاس سرور هست) و سروری که کلاینت می شناسد، فقط واسپاری کننده ی درخواست است و تنها این کلاس هست که کلاس Delegate را می شناسد. حال در نظر می گیریم که کلاس Delegate در حال رشد هست و پایداری ندارد(به آن عملیات و رفتار اضافه می شود). خب این بدیهی است که با هر اضافه شدن عملکردی، یک یا تعدادی operation با همان نام و امضاء در سرور بسازیم که صرفا در بدنه ی خود، کار delegation انجام می دهد. درنتیجه اینگونه یکی از مهمترین معضلات نگهداری از کلاس سرور این هست که تغییرات گسترشی کلاس Delegate باید در آن منعکس بشود و این نمود تغییر منتشرشونده به صورت حاد می باشد. الگو می گوید که کلاس سرور در اینجا به حالت Middle Man درآمده و وابستگی به صورت زائد افزایش یافته است؛ چرا که اینترفیس کلاس سرور، به اینترفیس کلاس Delegate وابسته شده است. پس وقتی اینترفیس کلاس Delegate را عوض می کنیم، مجبور هستیم که اینترفیس کلاس سرور را نیز عوض کنیم که کلاینت از سرویس جدید در کلاس Delegate بتواند استفاده کند. الگو در چنین مواردی پیشنهاد می دهد که PLK نقض شود. یعنی آنکه دید کلاینت به هر دو موجود برقرار باشد و کلاس سرور، آبجکت خود Delegate را در پاسخ

به یک درخواست برگرداند تا خود کلاینت مستقیما بتواند از سرویس های جدید و قدیمی آن استفاده کند. نقض PLK باید ارزش داشته باشد و صلاحدید طراح جهت استفاده از آن الزاما می بایست با منفعت های جانبی همراه باشد، مثل همینکه جلوی فاجعه ی انتشار تغییرات از کلاس Delegate به کلاس سرور گرفته شود.

۸.۲ شرایط مفید بودن الگوی دوم در موقعیت

واضح است که در موقعیت وصف شده، الگوی Remove Middle Man زمانی فایده خواهد داشت که جنس تغییرات منتشرشونده به گونه ای باشد که وقتی اینترفیس یک آبجکت دچار تغییر(عمدتا به معنای گسترش و افزایش متد) شد، اینترفیس کلاس دومی که آن کلاس اول را در حالت درونی خود دارد نیز تغییر بکند و این تغییر در کلاس دوم، صرفا اضافه کردن operation با همان امضای operation جدید در کلاس اول می باشد که بدنه ای جز delegation به کلاس اول نیز ندارد. مثلا درنظر می گیریم کلاینتی به یک Shop دید دارد و خود Shop نیز دیدی را به Owner دارد. کلاینت درخواست های خود را وقتی می خواهد به Owner بدهد باید از کلاس میانی Shop این درخواست را بکند. سپس Shop درخواست را به Owner واسپاری کرده و جواب را به کلاینت پس می دهد. اگر هر گسترشی در Owner مستلزم این شد که آن را عینا در Shop منعکس کنیم و صرفا با یک delegation آن را پر کنیم تا کلاینت همچنان بتواند از سرویس های جدید Owner استفاده کند، آنگاه شرایط مهیاست تا از الگوی Remove Middle Man استفاده کنیم و فایده ی آن را ببینیم. اینکه چه گونه از آن استفاده می شود را در بخش بعدی خواهیم دید. پس می توان گفت که موقعیت اینگونه با شرایط، گره می خورد تا به فکر این باشیم برای حل آن، Middle Man را (شاید موقتی در ازای ارزش فعلی) حذف کنیم.

۹.۲ چگونگی مفید بودن الگوی دوم در موقعیت

پیشنهادی که الگو می دهد این است که به آسودگی اجازه دهید تا دید ترایا بتواند در مجموعه بوجود بیاید. بدین صورت که کماکان کلاس اول می تواند به کلاس دوم وابسته باشد، اما کلاینت به هر دو دید داشته باشد و در جواب درخواست از کلاس اول، آبجکت کلاس دوم را دراختیار بگیرد. با این کار وقتی الگو را اعمال کردیم مزیت عمده ای که می رسد آن است که هر تغییر و گسترشی در کلاس دوم نیازمند آن نیست که در کلاس اول انتشار یابد و اگر این قسمت با متدهای بیشتری تکمیل شد، صرفا خود کلاینت می داند که باید از آنها مستقیما استفاده کند و قسمت دیگری مثل کلاس اول، تغییر نمی یابند. اگر مثال را در نظر بگیریم اینگونه تفسیر می شود که کلاینت همزمان هم به Shop دید دارد و هم به Owner و درعین حال Owner بخشی از حالت درونی Shop نیز هست. بر این اساس، هم Shop و هم Owner می توانند جداگانه گسترش یابند و تغییر در یکی، بر دیگری اثرگذار نیست. کلاس Owner با سرویس های بیشتر، فقط کلاینت را مجاب می سازد تا آن ها را بشناسد و از آن ها استفاده کند. اینکه چگونه الگو در موقعیت مفید خواهد بود، تنها به همین شکل میسر است.

۱۰.۲ معایب اعمال الگوی دوم

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- نقض صریح یکی از اصول شیءگرایی به نام PLK
- ایجاد زنجیره ی دید ترایا که بالقوه به پیچیدگی سیستم می افزاید
- احتمال رخدادن message chain که یکی از آفت های کد می باشد.

- خطر نقض encapsulation از آنجا که دید کلاینت به کلاس Delegate مستقیم می شود و می تواند همه ی عملکرد آن را ببیند.

۱۱.۲ مزایای اعمال الگوی دوم

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- ازبین رفتن تغییرات منتشرشونده ناشی از گسترش یافتن بخشی از کد
- کاهش وابستگی رشدیافته ی زائد بین سرور و کلاس Delegate
- حذف Middle Man هایی که درک کد را سخت و خوانایی را ضعیف می کنند.
- کاهش سربار اجرایی و پردازشی به جهت اینکه واسپاری های متعدد ازبین رفته اند.

۱۲.۲ مقایسه شرایط و نحوه اعمال الگوها در موقعیت

جهت مقایسه دو الگو باید گفت که الگوی Remove Middle Man وقتی اینترفیس یک کلاس به کلاس دیگر وابسته شده است، فایده خواهد داشت و اینگونه عمل می کند که وابستگی را از طریق ارتباط برقرار کردن به صورت دید مستقیم از کلاینت به کلاس Delegate کاهش می دهد.

در حالی که Change Value to Reference زمانی مفید خواهد بود که در سیستم کپی های تغییرناپذیر از یک ساختار داده ای در اثر تغییر داده اصلی، باید بروزرسانی شوند و اینگونه عمل می کند که به ازای مقدار در کپی ها، همه را به داده ی اصلی روی حافظه ارجاع می دهد.

۳ موقعیت سوم

۱.۳ بررسی اجمالی موقعیت

موقعیت در این سوال بدین گونه تعریف شده است که:

داده های تغییرپذیری در سیستم موجود هستند که آن را بی ثبات کرده اند. حال قصد داریم از زاویه دید دو الگوی Change Reference to Value و Separate Query from Modifier به این موقعیت نگاه کنیم و بسته به شرایطی که باید برقرار باشد به حل آن بپردازیم.

۲.۳ بررسی الگوی اول

الگوی Change Reference to Value زمانی کاربرد دارد که داده های تغییرپذیر در سیستم وجود دارد. تغییرپذیری داده اکثرا مشکل زا می باشد بنابراین ترجیح آن است که کپی های مختلف از یک آبجکت داشته باشیم ولی به داده با reference دسترسی نداشته باشیم. چون وقتی داده ی تغییرپذیر داشته باشیم، از هر جایی ممکن است که تغییر روی آن اعمال شود و اگر یک تغییر اشتباه باشد، باعث انتشار خطا روی همه دسترسی های به آن آبجکت خواهد شد و در این مواقع، محل بروز خطا را نیز نمی توان به راحتی پیدا کرد. اما اگر بوسیله value با داده کار کنیم، یعنی داده ی اصلی در جایی است که تغییرناپذیر است و هر جا به آن نیاز شد کپی های آن را دراختیار می گذاریم، داده ی اصلی محفوظ می ماند و نمی تواند از جاهای مختلف مورد استفاده قرار بگیرد. در مواردی که داده فقط جهت خواندن مورد استفاده قرار می گیرد و تغییر دادن آن بلاموضوع است، تاکید آن است که داده، تغییرناپذیر لحاظ شود. پس اگر آبجکت یا ساختار داده ای تغییر پذیر داشتیم که احیانا در داخل آبجکت دیگر موجود است، باید کپی های تغییرناپذیر از آن را تامین و به هر

درخواست کننده و استفاده کننده، ارسال کرد. این پیشنهاد الگو می باشد تا سیستم از بی ثباتی دربیاید. اینگونه از هر جایی نمی توان روی داده ی اصلی نوشت و آن را تغییر داد و سیستم را از ثبات انداخت؛ چرا که داده را value by در سیستم در دسترس می گذاریم.

۳.۳ شرایط مفید بودن الگوی اول در موقعیت

در شرایطی می توان از فواید الگوی Change Reference to Value سخن گفت که علت ناپایداری سیستم ناشی از داده های تغییرپذیر(ذکر شده در موقعیت) در اثر این باشد که داده ی اصلی by reference در سیستم قابل دسترس است. یعنی آن که هر قسمتی از سیستم می تواند به آبجکت یک داده ی اصلی دسترسی داشته و در خانه ی حافظه ی آن، هر مقداری بنویسد و آن را به کلی تغییر دهد و اشتباهات ناخواسته ای را پیش بیارد. این در حالی خواهد بود که ما نیز نمی دانیم که محل بروز اشتباه، ناشی از کجاست؛ چرا که از هر جایی ممکن است این تغییر رخ داده باشد. پس سیستم یک بی ثباتی را تجربه می کند. در چنین شرایطی، الگوی ذکر شده می تواند مفید باشد اما اینکه چگونه می تواند موقعیت را برطرف کند، در قسمت بعدی به آن خواهیم پرداخت. پس با وجود اینکه آبجکت یا ساختار داده ای تغییرپذیر، آماده کننده ی محیط برای بی ثبات کردن سیستم هست، در ادامه اگر متوجه شویم که دسترسی ها به آن از طریق ارجاع به محل حافظه ی داده می باشد، می توان گفت که شرایط برای پیاده سازی این الگو مهیا می باشد.

۴.۳ چگونگی مفید بودن الگوی اول در موقعیت

اگر شرایطی که برای موقعیت در نظر گرفته شد وجود داشت آنگاه به سراغ اعمال توصیه ی الگو می رویم. الگو اینگونه تاثیر مفید خود را می گذارد که برای

دسترسی از هرجایی به داده اصلی، آبجکت های کپی تغییرناپذیر از آن را تامین می کند و آن ها را در سیستم پاس می دهد. این چنین داده هایی آبجکت های هستند که مقادیر مشابه با آبجکت اصلی دارند اما ارجاعی به آن ندارند و به خانه ی حافظه ی متفاوتی اشاره می کنند. چنین کاری دسترسی by value نامیده می شود. اگر دستوری که الگو می دهد را بدین شکل پیاده کنیم، آنگاه سیستم را از بی ثباتی نجات داده ایم؛ زیرا دلیل اصلی بی ثباتی زمانی بود که از هر جایی می توانستیم به آبجکت تغییرپذیر اشاره کنیم و آن را تغییر دهیم که این حالت سیستم را دگرگون می ساخت و آن را باثبات جلوه نمی داد. حال اما از بیرون نمی توان به آبجکت اصلی دسترسی داشت و آن را در هر صورتی تغییر داد. بلکه فقط کپی های تغییرناپذیر از آن تهیه و در اختیار درخواست کنندگان قرار می گیرد.

۵.۳ معایب اعمال الگوی اول

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- وجود انتشار تغییرات روی همه ی کپی های آبجکت اصلی به دلیل تغییر آبجکت اصلی
- کاهش کارآیی سیستم به خاطر لزوم آپدیت شدن همه ی کپی های آبجکت اصلی
- اضافه کردن سرباز حافظه ناشی از تولید کپی از آبجکت اصلی
- احتمال بروز عدم یکپارچگی داده و رخداد تناقض برای یک داده به هنگام آپدیت کردن داده ی اصلی

۶.۳ مزایای اعمال الگوی اول

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- تغییرناپذیر کردن داده و تهیه کپی از آن برای درخواست کنندگان بیرونی که کمک می کند تا احتمال بروز اشتباه و درنتیجه بی ثباتی در سیستم به حداقل برسد.
- گرفتن دسترسی برای تغییر داده از هر جای دیگر
- عدم انتشار خطا در صورت اشتباه در تغییر مقدار داده ی اصلی
- با رسیدن به بی ثباتی، می توان محل بروز خطا را بهتر کشف کرد؛ چون دسترسی کمتری(توانایی ارجاع به حافظه روی داده اصلی محدود تر است) برای تغییر مقدار داده وجود دارد.

۷.۳ بررسی الگوی دوم

الگوی Separate Query from Modifier به طور کلی قصد دارد که اینترفیس کلاس ها را ساده کند. بدین معنا که متدها را ساده تر کرد و مجموعه ی متدها را کم کرد. این الگو زمانی کاربرد دارد که متدی در سیستم هم از نوع Query و هم از نوع Modifier موجود باشد. یعنی هم مقدار برمی گرداند و در عین حال نباید حالت آبجکت را عوض کند(به دلیل این که Query است) و هم از طرفی حالت آبجکت را عوض می کند(به دلیل اینکه Modifier است). بدیهی می نماید متدی که خالصا از نوع Query می باشد مطلوب می باشد چون تاثیری در قسمت های دیگر کد نمی گذارد و حالتی را عوض نمی کند. از هر جایی می توان آن را فراخواند چون فقط محاسبه ای انجام داده و آن را در اختیار می گذارد. در واقع این Modifier ها هستند که مشکل زا هستند. درنتیجه بهتر آن است که این دو مفهوم، کاملا مجزا درنظر گرفته بشوند و اگر متدی، مقداری را برمی گرداند نباید حالت راعوض کند. از آن طرف نیز Modifier ها نباید مقدار برگشتی داشته باشند. در این الگو، متدی موجود است که یک value برمی گرداند ولی state آبجکت(هر آبجکتی اعم از خودش و آبجکت بیرونی) را نیز تغییر می دهد و یک اثر مانا در سیستم می گذارد. الگو پیشنهاد می دهد که برای رفع چنین مشکلی، دو تا متد ساخته شود:

- یک متد برای اینکه محاسبات لازم را انجام بدهد و مقداری را برگرداند.
- دومین متد آنکه کار تغییر در حالت آبجکتی را انجام می دهد و هیچ مقدار بازگشتی ای نیز ندارد.

۸.۳ شرایط مفید بودن الگوی دوم در موقعیت

موقعیت بیان می کند که سیستم به وسیله ی داده های تغییرپذیر، دچار بی ثباتی شده است. حال در نظر می گیریم متدهایی در یک کلاس موجود است

که با بررسی آن ها متوجه می شویم این متدها هم از نوع Query و هم از نوع Modifier هستند. یعنی آنکه در عین حالی که محاسباتی را انجام می دهند و مقداری را برمی گردانند، حالت آبجکت یا آبجکت هایی را هم تغییر داده و روی سیستم اثر مانایی می گذارند. این که یک متد در درون خود بتواند مقدار آبجکتی را تغییر دهد(حالت آن را عوض کند) نشان از بروز خطر می دهد. اما در جایی خطر قطعی است که همان متد، خروجی ای را حاصل از یک محاسبه برگرداند. دلیل خطر همان رخداد بی ثباتی در سیستم است. وقتی معلوم نیست متد از چه نوعی است(آیا از نوع Query است و یا از نوع Modifier می باشد)، آنگاه نمی توان انتظار داشت که سیستم در حالت خطا نرفته و پیدا کردن محل خطا کار آسانی باشد. این شرایط همان جایی است که الگوی from Modifier Separate Query فایده ی خود را نشان می دهد. زمانی که بتوان مناطق پرخطر از کم خطر را شناسایی و تفکیک کرد، آنگاه ثبات به سیستم باز می گردد، چرا که محل های بروز خطا مشخص تر هستند(عمدتا در Modifier ها هستند) و هر متدی در هر جایی نمی تواند روی سیستم اثرگذار باشد. پس اگر جنس بی ثباتی از این دست بود که متدها از نظر دسته بندی Query-Modifier تقسیم بندی نشده اند، آنگاه شرایط در موقعیت، ما را به استفاده از این الگو ترغیب می کند.

۹.۳ چگونگی مفید بودن الگوی دوم در موقعیت

پس از احراز شرایط در موقعیت تعریف شده، الگو توصیه می کند که به راحتی متدها از نظر Query-Modifier تقسیم بندی شوند. هر متدی که عضو هر دو طبقه بود، به ازای آن دو متد ساخته شود که یکی برای Query و دیگری برای Modifier باشد. به عنوان مثال اگر متدی هم وظیفه ی محاسبه ی مقدار نهایی قیمت فاکتور را دارد و هم وظیفه ی بردن درخواست به حالت خروج از انبار را بر دوش داشته باشد، می توان آن را به دو متد شکست. متد اول

تنها مقدار نهایی قیمت فاکتور را محساسبه کرده و برمی گرداند و متد دوم، درخواست را به حالت خروج از انبار تغییر می دهد. پس از این کار متوجه می شویم که ثبات به سیستم برمی گردد و در صورت بروز رخداد در اینکه چرا درخواست، حالت درست را نگرفت فقط روی متد دوم متمرکز خواهیم شد و با دقت روی آن، کاری می کنیم که سیستم به بی ثباتی نرود. اینگونه می توانیم فایده ی این الگو را در موقعیت داده شده بخوبی تشریح کنیم و از آن استفاده کنیم.

۱۰.۳ معایب اعمال الگوی دوم

معایبی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- تعداد متدها را زیاد می کند که سبب حجیم شدن اندازه ی کد می شود.
- اینترفیس کلاس، شلوغ تر از قبل شده و خوانایی کد، تحت الشعاع قرار می گیرد.
- حوصله سر بر بودن فرآیند جداسازی متدها و ایجاد و ساخت دو متد جدید از متد قبلی که ممکن است زمانی را از ایجادکننده بگیرد و او احساس اتلاف وقت کند.
- وجود خطای بالقوه در ساختن دو متد که یکی برای Query باشد و یکی برای Modifier به اینصورت که این دو متد، کار متد قبلی را به همانگونه، انجام ندهند و نقص و کاستی در آن بوجود بیاید. این خطا می تواند از ایجاد کننده به دلیل کم اهمیت در نظرگرفتن مساله سر بزند.

۱۱.۳ مزایای اعمال الگوی دوم

مزایایی که در اثر اعمال این الگو پدید می آید را می توان به شرح زیر دانست:

- ساده تر کردن اینترفیس ها(به منظور جداسازی نیت های متفاوت موجود در یک متد)، جمع وجور کردن و cohesive کردن متدها
- کمک زیاد به debugging چون مکان های بالقوه خطر، همان Modifier ها هستند که سریعا locate می شوند و باگ را می توان زودتر شناسایی کرد.
- کیفیت دادن به کد به دلیل جداسازی بخش های بالقوه خطرناک از بخش های کم خطر
- شناسایی سریع متدهایی که ارزش وقت گذاشتن برای کیفیت و آزمون پذیری دارند.

۱۲.۳ مقایسه شرایط و نحوه اعمال الگوها در موقعیت

اگر بخواهیم مقایسه ی کوتاهی بین این دو الگو در موقعیت داشته باشیم باید ذکر کنیم که الگوی Separate Query from Modifier برای بی ثباتی سیستم، دنبال شرایطی است که متدها از نظر Query–Modifier تفکیک نشده باشند و اینگونه راه حل خود را ارائه می دهد که هر کدام را از این منظر تفکیک کرد. بدین ترتیب، تمرکز برای خراب نشدن سیستم و ونرفتن به وضع ناپایداری عمدتا روی متدهای Modifier بوده و با این نظم و انضباطی که به سیستم وارد می شود، ثبات به آن برمی گردد.

در حالی که الگوی Change Reference to Value زمانی مثمرثمر خواهد بود که دسترسی به آبجکت اصلی by reference باشد و هر قسمتی از برنامه به مقدار آن دسترسی داشته باشد. اینگونه بی ثباتی عجیبی در سیستم پیدا می شود که معلوم نیست عامل تغییر دادن داده ی اصلی و خراب کردن سیستم کجاست. الگو پیشنهاد می دهد که از داده ی اصلی کپی های تغییرناپذیر ساخته

و در اختیار بقیه قسمت ها قرار دهیم(دسترسی را by value کنیم) تا اینگونه نتوان مقدار داده ی اصلی از هر راهی تغییر کند. بدین شکل، دوباره سیستم، باثبات خواهد شد.