دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

تمرین سوم درس الگوها در مهندسی نرم افزار

استاد:

دکتر رامسین

تهیه کننده:

امید یعقوبی سامانی

۴۰۱۲۰۵۳۴۸

تابستان ۱۴۰۲

فهرست مطالب

موقعیت اول:

عبارات شرطی پیچیده داخل متدهای کلاس دیده می شود.

عبارات شرطی از ستون های اصلی زبان های برنامه نویسی هستند اما گاهی استفاده از آن ها می تواند باعث پیچیدگی برنامه ها و در نتیجه کاهش قابلیت خوانایی و فهم برنامه ها شود. برنامه ها هر چه طولانی تر باشند فهم آن ها سخت تر است و اگر عبارات شرطی به اینها اضافه شود خوانایی و فهم آنها را بیشتر کاهش می دهد. گاهی شرط ها آنقدر پیچیده هستند و حتی بدنه طولانی دارندکه زمانی که سراغ فهم else در شرط رفته ایم فراموش می کنیم در if چه اتفاقی می افتاد. هم بررسی شرط پیچیده است و هم بدنه آن و هم else آن. گاهی هم در عبارات شرطی خود شرط و بدنه پیچیده نیست و عبارات تو در تو باعث کاهش فهم می شود. در شرط ها معمولا باید if و else هم وزن باشند و کار مخالف هم را انجام بدهند ولی گاهی این ها تبدیل به کارهای بی ربط می شوند مثلا استثنا های برنامه ها در else قرار می گیرند. عبارات شرطی رایج ترین منابع پیچیدگی برنامه ها هستند و اغلب مشکلات در برنامه ها هم ناشی از عبارات شرطی است. عبارات شرطی می توانند علاوه بر اینکه خودشان ناخوانا هستند متدی که در آن هستند را هم پیچیده کنند و باعث شوند فهم متد دشوار شود. مشکل در عبارات شرطی معمولا از اینجا ناشی می شود که عبارات شرطی می گویند طبق چه شرایطی چه اتفاقی بیفتد اما در مورد چرایی اتفاق چیزی نمی گویند و چرایی در عبارات شرطی مبهم است.

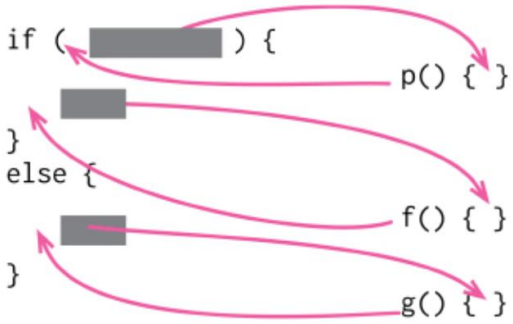
بررسی موقعیت:

الگوی اول: Decompose Conditional

بررسی الگوی اول:

مسئله: عبارات های شرطی (if-else) پیچیده داریم که خوانایی برنامه را سخت کرده اند.

راه حل: با استخراج متد از شرط، بدنه if و بدنه else خوانایی را افزایش می دهیم.



منطق های شرطی پیچیده رایج ترین منابع پیچیدگی در برنامه ها هستند. در زمان کد نویسی چیزهای زیادی وابسته به شرط ها می شوندکه این در نهایت منجر به یک تابع طولانی خواهد شد. طولانی بودن توابع از عواملی است که به خودی خود فهم را دشوار و خوانایی را کم می کند و عبارات شرطی، این پیچیدگی و دشواری فهم کد را بیشتر هم می کنند. مشکل معمولا در اینجا این است که هم عبارت شرطی و هم بدنه عبارت شرطی تنها کاری که اتفاق می افتد را مشخص می کند اما چرایی را بیان نمی کند که می تواند باعث شود علت وقوع مبهم شود و فهم آن دشوارگردد.

این مشکل یعنی داشتن منطق های شرطی پیچیده در متد ها معمولا زمانی اتفاق می افتدکه کد طولانی تر شده و دلیل طراحی نامناسب هم شرط ها در عبارات شرطی و هم بدنه آنها پیچیده شده اندکه نتیجه آن معمولا طولانی شدن متد ها وکاهش خوانایی است. خوانایی قابلیت نگهداری را افزایش می دهد وکاهش آن کار نگهداری سیستم را دشوار خواهدکرد.

برای ساده کردن این بخش پیچیده کد، باید مقصود منطق شرطی را شفاف کنیم و باید وقتی وارد بدنه شرط می شویم به آسانی چرایی ورود و چرایی انجام کار را بفهمیم. برای این کار با اعمال extract function بخش های پیچیده را استخراج و با دادن نام مناسب به آنها مشخص می کنیم دقیقا چه کاری انجام می دهند.

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

چگونگی بازآرایی:

۱. با کمک Extract Function، شرط عبارت شرطی را استخراج کرده و تابع استخراج شده را که دارای نام مناسب است به جای آن قرار می دهیم.

۲. این کار را روی بدنه انشعابات عبارت شرطی شامل if، then یا else هم تکرار می کنیم.

معایب اعمال الگو:

- یک لایه indirection ایجاد شده که فراخوانی تابع دارد، می تواندکارایی را کاهش بدهد و سربار به کلاس اضافه کند.
- Trace کردن را سخت می کند و برای فهم کد باید مسیر را دنبال کرد و به جاهای مختلف منتقل شد و برگشت و در نتیجه زمان دیباگ افزایش خواهد یافت.
- افراط در اعمال آن می تواند کلاسی که این متد ها در آن هستند را بسیار پیچیده کند و ناخوانایی را از متد به کلاس انتقال دهدکه نتیجه آن کاهش قابلیت نگهداری خواهد بود.
- در صورتی که مواردی که استخراج می شوند طولانی باشند می تواند Long Function را به جای دیگر یعنی تابع استخراج شده منتقل کند.

- در صورتی متد قبلا دارای Long Parameter Function باشد و اینها در موارد استخراج شده استفاده شده باشند می تواند اینها را در جاهای دیگر انتشار دهد. پس اعمال این الگو باید با طراحی درست و دقیق و حساب شده باشد.
- افزایش تعداد متد ها که حاصل اعمال الگو است نیازمند تلاش بیشتر برای فهم و یادگیری آنها است.
- اجرای بدون طراحی دقیق و بی حساب درکلاس می تواند به cohesion کلاس آسیب بزند.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- چون شرط ها و بدنه ها را با اعمال Extract Function بازآرایی می کنیم توابع ایجاد شده می توانند در جاهای دیگر هم استفاده شوند و قابلیت استفاده مجدد بالا می رود.
- آزمون پذیری افزایش پیدا می کند چون توابعی که استخراج شده اند خودشان جداگانه باید تست شده باشند برای همین تست کردن این تابع ساده تر خواهد بود.
- در صورتی این شرط ها یا بدنه انشعاب در چندین متد یا تابع استفاده شده باشند می تواند duplication را هم از بین ببرد.

الگوهای مرتبط:

- Extract Function

Bad smell هایی که حذف می کند:

- Long Function

شرایط مفید بودن الگو:

زمانی که داخل توابع یا متد ها عبارت شرطی پیچیده باشد این پیچیدگی می تواند به این صورت باشد که خود شرط پیچیده باشد و بدنه انشعابات شرطی هم پیچیده باشند.

این مورد باعث می شود خوانایی کاهش یابد و در نتیجه انعطاف پذیری و قابلیت نگهداری هم کاهش می یابد و بعد از مدتی دیگرکسی نمی تواند به کد آن دست بزند چون آن را نمی فهمد و عدم فهم باعث می شود نتواند تغییرات لازم را درکد انجام بدهد. این مورد علاوه بر این باعث بروز Long Function هم می شود که پیچیده بودن و به تبع آن طولانی شدن نتیجه آن است. در زمان بازآرایی کد هر جا دیدیم شرط در عبارات شرطی و همچنین بدنه آنها غیر قابل فهم است باید حتما برای سادگی فهم این الگو اعمال شود. این افزایش خوانایی و فهم کد می تواند حتی مقدمه ای بر بازآرایی های بیشتر و مفید تر باشد. پس باید با دقت خوبی انجام بپذیرد.

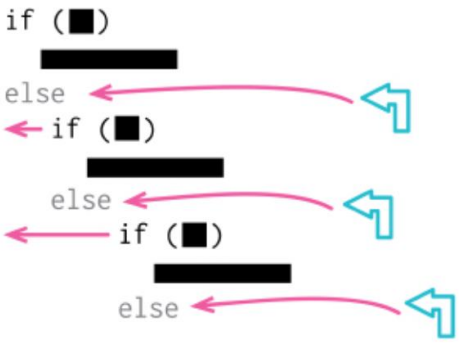
چگونگی مفید بودن الگو:

با اعمال Decompose Conditional روی عبارت شرطی که با اعمال آن روی خود شرط و بدنه انشعاب انجام می شود خوانایی افزایش یافته، فهم آن ساده می شود و انعطاف پذیری بالا خواهد رفت و در نتیجه قابلیت نگهداری افزایش پیدا می کند. با اجرای extract function روی شرط، شرط در عبارت شرطی ساده می شود. با اعمال روی بدنه های انشعاب هم آن بخش ها بسیار ساده خواهند شد و علاوه بر اینکه bad smell ها می توانند از بین بروند خوانایی کد افزایش یافته و فهم آن برای برنامه نویسان و نگهداری کنندگان از سیستم ساده تر می شود. در نهایت با این کار تست کردن سیستم خیلی ساده تر می شود و این هم نتیجه مثبتی روی کار نگهداری کنندگان از سیستم دارد و باعث می شود نتیجه بهتری بگیرند.

الگوی دوم: Replace Nested Conditional with Guard

مسئله: عبارات های شرطی تو در تو داریم که باعث شده مسیر عادی اجرای برنامه مشخص نباشد.

راه حل: برای حالات خاص و استثناها از عبارت های نگهبان (guard) استفاده می کنیم.



عبارات شرطی اغلب دو حالت دارند. یک حالت این است که در عبارات شرط همه انشعاب شرط ها رفتار معمول دارند. حالت دیگر این است که برخی انشعاب شرط ها حالات نامعمول که خاص و استثنایی هستند را انجام می دهند.

این حالات عبارت شرطی هرکدام مقاصد مختلفی دارند و این باید درکد نمود داشته باشد. اگر یکی از انشعابات شرط رفتار نامعمولی انجام می دهد باید آن را با عبارت نگهبان جایگزین کرد. این الگو روی این تاکید دارد که انشعابات عبارات شرطی که رفتار معمول را انجام می دهند به یک اندازه مهم هستند و عبارات نگهبان در واقع استثنا هستند و وظیفه مندی اصلی متد نیستند. نتیجه این کار افزایش خوانایی خواهد بود که قابلیت نگهداری را هم بهتر می کند.

چگونگی بازآرایی:

۱. تمام عبارت نگهبان که می تواند باعث اجرای exception یا return سریع یک مقدار در متد شوند را ایزوله کرده و اینها را در ابتدای متد قرار می دهیم.

۲. بعد از این که تست کردیم برنامه درست کار می کند بررسی می کنیم می توان روی آن الگوی Consolidate Conditional Expression را اجرا و عبارت نگهبان را تجمیع کرد یا خیر. در صورتی تعدادی از عبارات نگهبان یک نتیجه یکسان برگردانند امکان اعمال آن وجود خواهد داشت.

```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```

```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

معایب اعمال الگو:

- افراط در استفاده از عبارات نگهبان ممکن است باعث کاهش فهم کد شود و در صورتی متدی تعداد زیادی عبارت نگهبان داشته باشد می توان نشان دهنده وجود مشکل دیگری در طراحی متد باشد.
- عبارات نگهبان می توانند باعث بروز duplication code شوند پس حتما باید در صورت لزوم Consolidate Conditional Expression را اعمال کرد.
- عبارات نگهبان در صورتی خیلی زیاد باشند می توانندکارایی را کاهش دهند به خصوص اینکه امکان درست بودن آنها کم باشد ولی با این حال همیشه ابتدای اجرای متد بررسی می شوند.
- عبارات نگهبان نیاز به تلاش بیشتر برای اعمال دارند چون باید ترتیب اجرای آنها را هم در مواردی به درستی رعایت کرد و نیازمند طراحی دقیق تر هستند.

- ممکن است آزمون پذیری را کم کنند و برای تست نیاز به تلاش بیشتر باشد چون می توانند در Reachability به بخش هایی که قرار است تست شوند اثر منفی بگذارند.
- نیازمند آشنایی دقیق برنامه نویس با آن ها است و در صورت عدم آشنایی فهم آنها برای برنامه نویس دشوار خواهد بود.
- در شرط های تو در تو معمولا اشیایی در نتیجه درست بودن یا نبودن شرط ساخته می شوند ولی با عبارت نگهبان باید همه را از ابتدا ساخت تا بتوان آنها را ابتدای متد بررسی کرد و این ممکن است باعث ایجاد سربار و کاهش کارایی شود.
- در برخی زبان ها مثل C ممکن است garbage collection را انتهای تابع نوشته باشند و با خروج زودهنگام این مورد اجرا نخواهد شد.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- کاهش عبارات شرطی تو در تو و در صورت لزوم خروج زودهنگام از متدکارایی را افزایش می دهد.
- با کمک عبارت های نگهبان می توان خطاها را ابتدای متد به سادگی اداره کرد و از انتشار آنها جلوگیری کرد.
- عبارات نگهبان آزمون پذیری را افزایش می دهد چون مشخص کننده رفتار غیرعادی یا حالت خاص هستند می توان اینها را به سادگی تست کرد.
- جدا کردن موارد خاص و استثنا باعث می شود ادامه اجرای متد cohesive تر شود.
- اضافه کردن قابلیت fail fast روی ورودی های نامعتبر اضافه می کند.
- وابستگی انشعابات به همدیگر با خروج اینها از لابه لای انشعابات کاهش خواهد یافت.
- ایجاد جدایی دغدغه ها در متد.

الگوهای مرتبط:

- Consolidate Conditional Expression

Bad smell هایی که حذف می کند:

- Duplicated Code

شرایط مفید بودن الگو:

در شرایطی که شرط های تو در تو داشته باشیم فهم کد سخت و خوانایی پایین می آید. اگر همراه شرط های تو در تو return های زودهنگام هم داشته باشیم وضعیت بدتر هم خواهد شد. این مشکلات باعث خواهد شد نتوان متوجه جریان اصلی برنامه شد و نتیجه این است نتوان نگهداری را به درستی انجام داد. در عبارات شرطی باید انشعابات یک وزن داشته باشند و رفتار معمول انجام بدهند. اگر برخی رفتار نامعمول داشته باشد باید آنها را با عبارات نگهبان جایگزین کرد. هر زمان در توابع و متد ها عبارات شرطی دیدیم که تو در تو هستند در نگاه اول باید به فکر عبارات نگهبان افتاده و بررسی کنیم می توان آنها را اعمال کرد یا نه همچنین هر زمان عبارات شرطی تو در تو دیدیم و نتوانستیم جریان اصلی تابع وکاری که می خواهد انجام دهد را بفهمیم باید به فکر عبارات نگهبان افتاده و با بازآرایی کردن کد خوانایی را افزایش بدهیم که نتیجه آن فهم ساده کد و در نتیجه جریان اصلی اجرای یک تابع خواهد بود.

چگونگی مفید بودن الگو:

زمانی که استثناها و حالات خاص را در متد جدا کرده و ابتدای متد قرار می دهیم و در صورت برقراری آن ها از متد return می کنیم ادامه اجرای متد مسیر طبیعی و نیازمندی اصلی متد خواهد بود با این کار فهم متد هم ساده شده، خوانایی افزایش یافته و باعث بهبود قابلیت نگهداری می شود.

در صورتی که بتوان Consolidate Conditional Expression را هم اعمال کرد وضعیت می تواند بهتر هم بشود و خوانایی بیش از پیش افزایش پیدا کند.

هر جایی عبارات شرطی تو در تو دیدیم که نتوانستیم آن را بفهمیم یا فهم آن دشوار بود به سرعت می توان الگو را اعمال کرد. با اعمال الگو استثناها و حالات خاص به ابتدای متد یا تابع منتقل می شوند و در صورتی برقرار باشند و شرط آنها درست باشد به طور زودهنگام از تابع یا متد خارج می شویم. در نتیجه این کار متد خوانا تر شده و با اجرای جریان اصلی بعد از عبارات نگهبان این بخش از متد را cohesive تر می کنیم. اجرای این الگو باعث نوعی separation of concern در سطح متد هم می شود.

مقایسه:

هر دو الگو می خواهند عبارات شرطی پیچید داخل متد های کلاس را ساده کنند اما مقصود آنها متفاوت است. در Decompose Conditional شرط ها و انشعابات عبارات شرطی پیچیده هستند و فهم آن دشوار است و در Replace Nested Conditional with Guard پیچیدگی عبارات شرطی ناشی از وجود شرط های تو در تو و return های حالات خاص است که باعث می شود روند اصلی اجرای متد مشخص نباشد و فهم آن دشوار شود. در Decompose Conditional موارد پیچیده را استخراج می کنیم و توابع را در شرط و بدنه انشعاب شرطی قرار می دهیم و با این کار فهم آن را ساده می کنیم. در Replace Nested Conditional with Guard عبارات نگهبان را یافته و آنها را ابتدای متد قرار می دهیم تا اگر درست بودند سریع از متد خارج شود . بعد از اینها جریان اصلی متد خواهد بود و عبارات نگهبان که باعث پیچیدگی می شدند و در نتیجه فهم دشوار می شد را از آن جدا کردیم و بعد از عبارات نگهبان وظیفه مندی اصلی متد قرار خواهدگرفت. در اینجا ممکن است نیاز باشد Consolidate Conditional Expression را هم روی عبارات نگهبان اجرا کنیم. هر دو مورد حتما باید با انجام تست بازآرایی شوند تا به عملکرد اصلی آسیب نخورد به خصوص در مورد عبارات نگهبان این موضوع بیشتر مورد توجه است. در هر دو مورد نتیجه افزایش قابلیت فهم، خوانایی، انعطاف پذیری و در نتیجه بهبود قابلیت نگهداری خواهد بود.

موقعیت دوم:

در برخی متدها فهرست پارامترها بسیار طولانی است.

پارامتر توابع نقاط تغییر پذیری یک تابع هستند و اولین جایی است که نشان می دهد رفتار تابع می تواند براساس آن ها تغییرکند. فهرست طولانی گاهی نشان دهنده عدم وجود cohesion در تابع است. در صورتی لیست متد ها طولانی باشد و تابع هم منسجم باشد نشان دهنده یک bad smell است و باید این موضوع را در تابع تصحیح کرد. در سیستم های تجاری چون توابع کارهای کوچک انجام می دهند معمولا لیست پارامترهای کوچک دارند. فهرست طولانی می تواند ناشی از این باشدکه مقادیر یک رکورد داده ای خاص را استخراج کرده و آن ها را جداگانه به تابع پاس می دهیم یا ممکن است به این خاطر باشدکه مقداری که خود تابع می تواند بدست آورد را مجددا به آن پاس داده ایم. البته این موارد می تواندکاملا با هدف انجام شده باشد و به خاطر مصالح دیگری این کارها صورت پذیرفته باشد و این هزینه بهبود چیز دیگری باشدکه پرداخت شده است.

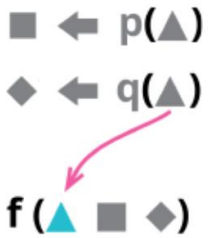
بررسی موقعیت:

الگوی اول:Preserve Whole Object

بررسی الگوی اول:

مسئله: یک متد پارامترهای زیادی دریافت می کند و همه اینها از یک رکورد داده ای یا object می آیند.

راه حل: خود رکورد داده یا object را به عنوان پارامتر به متد می دهیم.



اگرکدی داشته باشیم که در آن چند مقدار از یک رکورد دریافت و به طور جداگانه به عنوان پارامتر به یک تابع داده می شود بهتر است این مقادیر را با خود رکورد جایگزین کنیم و اجازه دهیم تابع در بدنه خودش این مقادیر را از رکورد دریافت کند. لیست طولانی پارامتر ها یک bad smell است و اینکه هربار مقادیر را از یک رکورد بگیریم و به تابع به عنوان پارامتر بدهیم درکارایی اثر منفی دارد. با دادن رکورد به تابع به عنوان پارامتر در صورتی در آینده تابع نیاز به مقادیر بیشتری از رکورد داشت نیاز نیست لیست پارامترها را تغییر بدهیم و تغییرات در بدنه تابع انجام خواهد شد. باکاهش تعداد پارامترهای یک تابع معمولا قابلیت خوانایی افزایش می یابد و فهم آن ساده تر می شود.

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (aPlan.withinRange(low, high))
```

```
if (aPlan.withinRange(aRoom.daysTempRange))
```

چگونگی بازآرایی:

۱. ابتدا یک تابع با پارامتر هایی که می خواهیم ایجاد می کنیم.

۲. نام مناسبی به آن می دهیم. البته این نام را در نهایت تعویض خواهیم کرد.

۳. بدنه تابع جدید را با فراخوانی به تابع قدیمی تکمیل کرده و از پارامتر های جدید به قدیمی نگاشت انجام می دهیم.

٤. Static check انجام می دهیم

٥. فراخواننده تابع را طوری تغییر می دهیم که از تابع جدید استفاده کند و بعد از هر بار انجام اینکار تست را اجرا می کنیم. اینکار در نهایت باعث ایجادکد هایی خواهد شدکه لازم نیستند پس باید Remove Dead Code را اجرا کنیم.

٦. وقتی تمام فراخواننده ها تغییرکردند Inline Function را روی تابع اصلی اجرا می کنیم.

۷. نام تابع جدید و تمام فراخواننده ها را تغییر می دهیم.

معایب اعمال الگو:

- این کار باعث coupling بین تابع و شی یا رکورد داده ای مورد نظر می شود و تابع باید شی را بشناسد.
- ممکن است باعث Feature Envy شود پس باید قبل از اعمال با بررسی دقیق و طراحی درست این کار را انجام داد تا این bad smell رخ ندهد.
- فهم تابع می تواند در برخی موارد دشوار شود چون پارامتر های تابع از اولین جاهایی هستند که می تواند نشان دهنده مقصود و هدف تابع باشد.
- در صورتی شی ای که به تابع داده می شود بزرگ باشد و تابع تنها به چند مقدار از آن نیاز داشته باشد اعمال الگو می تواند باعث شود بهره وری کاهش یابد.
- قابلیت استفاده مجدد تابع کاهش می یابد چون همیشه نیازمند داده یک شی به عنوان پارامتر به آن است و برخی کلاینت ها ممکن است به شی دسترسی نداشته باشند.
- کپسوله بودن و information hiding برای تابع نقض می شود چون تابع می تواندکاملا به شی ای که به آن داده می شود دسترسی داشته باشد.

- در صورت شی در اختیار بخش های دیگر هم باشد و مالکیت چندگانه داشته باشد تغییرات مقادیر ممکن است روی تابع اثر منفی بگذارد.
- در صورت عدم تعریف get و set برای شی و عدم اجبار تابع به استفاده از آنها تغییرات در مقادیر ممکن است در تابع منتشر شود.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- می تواندکد مربوط به فراخواننده تابع را کوتاه کند.
- کارایی را افزایش خواهد داد چون نیاز نیست پارامتر ها بازیابی شده و به تابع داده شوند.
- قابلیت آزمون را بهتر می کند و تست کردن تابع با ورودی های کمتر ساده تر خواهد بود.
- می تواند با انتقال بازیابی مقادیر از شی به داخل خود تابع باعث information hiding شود.
- این توابع معمولا در آینده نیاز به مقادیر دیگری از شی یا رکورد داده ای هم خواهند داشت و با این کار نیاز نیست در آینده تابع تغییرکند و تغییرات در خودش ایزوله خواهد بود.

الگوهای مرتبط:

- Introduce Parameter Object
- Extract Class
- Remove Dead Code
- Inline Function
- Extract Function
- Extract Variable

Bad smell هایی که حذف می کند:

- Long Function
- Data Clumps
- Long Parameter List

شرایط مفید بودن الگو:

زمانی که لیست پارامتر های یک تابع بزرگ است و این به این خاطر است که مقادیر را از یک رکورد داده ای گرفته و به تابع می دهیم ولی بهتر است کل شی یا رکورد داده ای را به تابع بدهیم. با اعمال الگو می توان لیست پارامتر های تابع را کوچک ترکرد که نتیجه آن افزایش خوانایی، فهم ساده تر و در نتیجه بهبود قابلیت نگهداری خواهد بود. هر زمان در کد دیدیم از یک شی مقادیری را دریافت می کنیم در متغیر می گذاریم و آنها را به یک تابع می دهیم باید به فکر این الگو بیفتیم وکد را بازآرایی کنیم. البته باید در اجرای آن دقت نمود. چون داده ای به تابع یا متد می دهیم، نوعی دانش اضافی به متد می دهد و مثلا اگر از یک شی بزرگ تنها دو یا سه مقدار را به یک تابع می دهیم و مطمئنیم در آینده هم این ها افزایش پیدا نمی کنند شاید اجرای این الگو به صرفه نباشد و برای اجرای الگو باید شرایط به دقت سنجیده شود تا باعث بروز مشکلات در آینده نشود.

چگونگی مفید بودن الگو:

زمانی که لیست پارامتر های یک تابع طولانی است و این ناشی از این است پارامتر زاید به آن داده می شود با اعمال الگو می توان این پارامتر زاید را حذف کرد وکل رکورد داده ای را به تابع داد و از این به بعد این خود تابع است که مقادیری که قبلا به عنوان پارامتر به آن داده می شد را بازیابی می کند. نتیجه این کار رفع Long Parameter List است و باعث افزایش خوانایی کد و فهم ساده تر آن می شود.

به علاوه باعث کوتاه تر شدن فراخواننده تابع هم می شود و دیگر مجبور نیست خودش مقادیر را بازیابی کرده و به تابع بدهد و Long Function که در جای دیگری است، مثلا در فرخواننده است را می تواند درست کند. اجرای الگو با کوتاه کردن لیست پارامتر های تابع می تواند تست کردن آن را هم ساده کند و تست هم یک مسئله مهم در زمان نگهداری از سیستم است.

الگوی دوم:Replace Parameter with Query

مسئله: مقداری را از یک شی بازیابی می کنیم و آن را به عنوان پارامتر به یک تابع می دهیم در حالی که خود تابع می تواند آن را بازیابی کند.

راه حل: باید پارامتر را حذف کنیم و به تابع اجازه دهیم خودش مقدار را بازیابی کند.

لیست پارامتر های یک تابع نقاط تغییرپذیری آن هستند و اولین جایی است که نشان می دهد تابع چگونه می تواند رفتار متفاوتی داشته باشد. با حذف پارامتر های زاید، تکرار ها حذف و لیست پارامترها کوتاه و در نتیجه خوانایی کد افزایش پیدا می کند. این که مقداری را به عنوان پارامتر به تابع می دهیم در حالی که خودش هم می تواند آن را بدست بیاورد نوعی تکرار است و می تواند پیچیدگی زاید اضافه کند. با حذف پارامتر زاید مسئولیت تعیین مقدار و بازیابی آن را به خود تابع انتقال می دهیم. اغلب تنها در مواردی که به بازیابی نیاز نیست باید مقدار را به عنوان پارامتر داد و در صورت نیاز به بازیابی متغیر در حالی که خود تابع می تواند این کار را کند خوب است این کار در بدنه خود تابع انجام بشود. البته باید توجه کرد انتقال مسئولیت درست باشد و این انتقال مسئولیت cohesion تابع را از بین نبرد. نتیجه نهایی کار بالارفتن خوانایی است.

```
availableVacation(anEmployee, anEmployee.grade);

function availableVacation(anEmployee, grade) {
  // calculate vacation...
```

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // calculate vacation...
```

چگونگی بازآرایی:

۱. اگر لازم بود برای پارامتر هایی که نیاز به محاسبه دارند extract function را اجرا می کنیم.

۲. در بدنه تابع ارجاع به پارامتر را با ارجاع به عبارتی که حاصل آن مقدار پارامتر است تغییر می دهیم.

۳. با استفاده از Change Function Declaration پارامتر زاید را از لیست پارامتر ها حذف می کنیم.

معایب اعمال الگو:

- در صورتی بازیابی توسط خود متد های شی جاری یعنی شی ای که متد در آن است انجام نشود باعث ایجاد coupling با شی دیگری خواهد شد. یعنی الگو زمانی که مقداری که بازیابی می کند اگر قلمرو کلاس باشد در خود کلاس باشد بسیار مفید است، در غیر این صورت coupling ایجاد می کند.

- در صورتی بازیابی توسط خود متد های شی جاری یعنی شی ای که متد در آن است انجام نشود می تواند باعث کاهش کارایی شود.
- فهم تابع می تواند در برخی موارد دشوار شود چون پارامتر های تابع از اولین جاهایی هستند که می تواند نشان دهنده مقصود و هدف تابع باشد.
- قابلیت استفاده مجدد تابع کاهش می یابد چون همیشه نیازمند است شی آن ساخته شود و مثلا به طور استاتیک نمی توان از تابع استفاده کرد. البته استفاده استاتیک از متد ها غیر از constructor نقض اصول شی گرایی است ولی می تواند ضعفی باشدکه در نتیجه الگو بروز می کند.
- در صورتی که مقادیری که تابع خودش بازیابی می کند توسط بخش های دیگر تغییرکنند ممکن است باعث بروز مشکل شود.
- تست white box تابع دشوار تر می شود.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- با کاهش تعداد پارامتر ها تست کردن تابع ساده تر می شود.
- می تواند با انتقال بازیابی مقادیر به داخل خود تابع باعث information hiding شود.
- با کپسوله کردن بخشی از داده درون تابع قابلیت استفاده مجدد در برخی موارد افزایش پیدا می کند.
- می تواند دغدغه ساخت شی را از روی دوش فراخواننده تابع بردارد.

الگوهای مرتبط:

- Extract Function
- Replace Temp with Query

Bad smell هایی که حذف می کند:

- Long Parameter List

شرایط مفید بودن الگو:

زمانی که مقادیری را بازیابی می کنیم و به عنوان پارامتر به یک تابع می دهیم ولی خود تابع هم می تواند مقادیر را بازیابی کند در این حالت لیست پارامتر های تابع بی مورد طولانی شده و می توان به سپردن بازیابی مقدار به خود تابع Long Parameter List را برطرف کرد و در نتیجه خوانایی بهبود می یابد. هر جا دیدیم شی ای به تابعی می دهیم و مقداری از آن را هم بازیابی کرده و به تابع می دهیم به فکر استفاده از این الگو می افتیم و با حدف پارامتر اجازه می دهیم خود تابع کار را انجام دهد. البته بهترین حالت برای اعمال الگو زمانی است که تابع درکلاس باشد یعنی متد باشد و مقداری که بازیابی می کند عنصری از نمونه کلاس باشد و این حالت بهترین حالت استفاده از الگو است. در غیر این صورت coupling دارد. البته این coupling از قبل هم بوده است و چیزی اضافه نمی شود ولی در هر صورت کم کردن coupling بسیار مفید خواهد بود.

چگونگی مفید بودن الگو:

با اعمال این الگو و انتقال مسئولیت بازیابی مقدار به خود تابع لیست پارامتر های تابع کوچک می شود و پارامتر زاید به آن داده نمی شود و نتیجه آن افزایش خوانایی و بهبود انعطاف پذیری و در نتیجه بهتر شدن قابلیت نگهداری است و Long Parameter List هم از بین می رود.

پارامتر را از تابع حذف می کنیم و اجازه می دهیم مقدار راکه تابع به آن نیاز دارد خودش بازیابی کند. اینکه مقداری از شی ای را بازیابی کنیم و به تابعی بدهیم در حالی که خودش هم می تواند آن را بازیابی کند یعنی به آن شی دسترسی دارد احتمال اینکه در آینده هم این کار راکنیم افزایش می دهد با اعمال الگو از انتشار تغییرات در آینده در صورت نیاز به اضافه کردن پارامتر بیشتر پیشگیری خواهد شد.

به علاوه با کاهش تعداد پارامتر ها تست کردن ساده تر می شود و این در ساده شدن و سریع شدن کار نگهداری کنندگان تاثیر مثبت خواهد داشت.

مقایسه:

در اینجا لیست پارامتر های متد طولانی است. در Preserve Whole Object یک رکورد داده ای یا شی داریم که مقادیر را از آن می گیریم و به عنوان پارامتر به تابع می دهیم. الگو می گوید به جای اینکار خود شی یا رکورد داده را به تابع بدهید و پارامتر های زاید را حذف کنید. در Replace Parameter with Query مقادیر داده را بیرون از تابع بازیابی می کنیم و عنوان پارامتر به تابع می دهیم. این در حالی است که خود تابع دسترسی و توانایی بازیابی این مقادیر را دارد . الگو می گوید بازیابی مقادیر را به خود تابع بسپارید و لیست پارامتر ها را کوتاه کنید. در هر دو مورد لیست پارامتر ها در نهایت کوتاه می شود و این مورد می تواند فهم کد را ساده کند، انعطاف پذیری را بالا ببرد و قابلیت نگهداری را بهبود دهد. تفاوتی که این دو دارند این است که در Preserve Whole Object تابع شی را از ابتدا نمی شناسد و ما به داده شی به آن برای آن به شی دید ایجاد می کنیم اما در Replace Parameter with Query تابع خودش شی ای که قرار است از آن مقدار بازیابی کند و معمولا خود کلاس جاری است را می شناسد و دید اضافی ایجاد نمی شود. اما در هر دو مورد چون با شی سر وکار دارند و از ان مقدار بازیابی می کنند تغییرات توسط بخش های دیگر می تواند روی کار آنها اثر بگذارد در حالی که قبلا تنها یک مقدار را در پارامتر می گرفتند و این موضوع نیازمند دقت و تلاش بیشتر و طراحی دقیق تر است.

موقعیت سوم:

تغییر ساختارهای توارثی به دلیل انتشار تغییرات مشکل شده است.

ساختارهای توارثی به دلیل خاصیت ذاتی شان می توانند منشا تغییرات در سیستم های نرم افزاری باشند. توارث قوی ترین نوع coupling است پس انتشار تغییرات در ساختارهای توارثی می تواند بسیار زیاد باشد.

گاهی به مرور زمان و در ایجاد ساختارهای توارثی تکرار بوجود می آید. خود تکرار یا duplication از عوامل اصلی انتشار تغییرات است و می تواند تغییر در یکجا باعث شود مجبور باشیم دیگر مکان ها را هم تغییر بدهیم. گاهی به مرور و در زمان تکامل ساختارهای توارثی به مرحله ای می رسیم که تغییرات در فوق کلاس باعث ایجاد مشکل در زیرکلاس ها می شود یا زیرکلاس مجبور می شود پیاده سازی پدر را خالی کند و این زمان بروز مشکلات در ساختار توارثی است و تغییرات درکلاس پدر در فرزندان منتشر می شود ولی فرزندان نباید تغییرکنندکه نشان می دهد ساختار توارثی دچار مشکل است و در هر دو مورد باید این مشکلات را برطرف کرد.

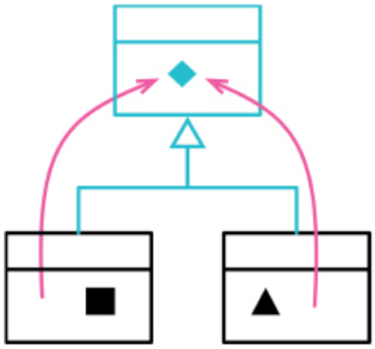
بررسی موقعیت:

الگوی اول:Extract Superclass

بررسی الگوی اول:

مسئله: کلاس هایی داریم که ویژگی های مشابه دارند.

راه حل: برای آنها فوق کلاس تعریف و ویژگی های مشترک را به فوق کلاس منتقل می کنیم.



اگر دوکلاس داشته باشیم که کار مشابه انجام می دهند با استفاده از توارث این اشتراکات را به فوق کلاس منتقل می کنیم. هم داده های مشترک و هم رفتارهای مشترک را باید به فوق کلاس منتقل کنیم. این کار در واقع همان generalization است و از تکرار جلوگیری می کند و قابلیت نگهداری را هم بالا می برد. این کار در برنامه های تکاملی مکانیزمی برای ایجاد و تشکیل ساختارهای توارثی است.

چگونگی بازآرایی:

۱. یک فوق کلاس خالی ساخته وکلاس اصلی را زیرکلاس آن می کنیم. اگر لازم بود روی constructor الگوی change Function deceleration را اعمال می کنیم.

۲. تست می کنیم.

۳. به ترتیب با استفاده از pull up constructor body، pull up method و pull up field موارد مشترک را به فوق کلاس منتقل می کنیم.

٤. متدهای باقی مانده را در زیرکلاس بررسی می کنیم. اگر بخش مشترکی وجود داشت ابتدا extract function را اعمال و بعد از آن pull up method را انجام می دهیم.

٥. کلاینت کلاس اصلی را بررسی کرده و آن را طوری تنظیم می کنیم که از اینترفیس فوق کلاس استفاده کند.

```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```

```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id() {...}
}
```

معایب اعمال الگو:

- با توجه به اینکه توارث قوی ترین نوع coupling است این الگو باعث ایجاد coupling بین فوق کلاس و زیرکلاس ها می شود.
- در صورتی کلاس فوق کلاس داشته باشد در برخی زبان ها که امکان زیرکلاس گرفتن از چند کلاس نیست نمی توان الگو را اعمال کرد.
- استفاده افراطی از الگو می تواند فهم کد را دشوارترکند.
- در صورت عدم رعایت دقیق اصول شی گرایی و موارد مربوط به توارث اینکار می تواند باعث بروز bad smell های دیگری بشود.
- در صورتی کلاس از قبل فوق کلاس داشته باشد اعمال الگو دشوار و پیچیده است.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- کاهش تکرارکد در نتیجه انتقال اشتراکات به فوق کلاس
- قابلیت استفاده مجدد افزایش پیدا می کند چون فوق کلاس abstract تر شده است.
- DIP برقرار می شود و در نتیجه تغییرات انتشار پیدا نمی کند.
- می توان برخی رفتار را در فوق کلاس کپسوله کرد و فهم آن برای استفاده کنندگان از زیرکلاس ها ساده تر خواهد بود.

الگوهای مرتبط:

- Pull Up Method
- Pull Up Field
- Extract Class
- Extract Function

Bad smell هایی که حذف می کند:

- Large Class
- Alternative Classes with Different Interface

شرایط مفید بودن الگو:

زمانی که تعدادی کلاس داریم که در حالات و رفتار اشتراک دارند وکدشان در برخی حالات مشابه هم است هرگونه تغییری باید در تمامی این کلاس ها اعمال شود و این باعث می شود قابلیت نگهداری پایین باشد و تغییر در همه این کلاس ها منتشر شود در اینجا باید ساختار توارثی ایجادکنیم و اشتراکات را به فوق کلاس ببریم.

هر زمان تعدادی کلاس داشتیم و هر زمان خواستیم تغییری در برنامه ایجادکنیم مجبور می شدیم همه را تغییر دهیم نشان می دهد چیزی بین اینها مشترک است و اینجا به فکر استفاده از الگو می افتیم. یک کلاس به عنوان فوق کلاس تعریف می کنیم و اشتراکات را به آن منتقل وکلاس ها را فرزندان آن قرار می دهیم. به این طریق ایجاد ساختار توارثی تغییرات در فوق کلاس ایزوله می شود و تغییرات محلی هم محدود در زیرکلاس ها خواهد بود.

چگونگی مفید بودن الگو:

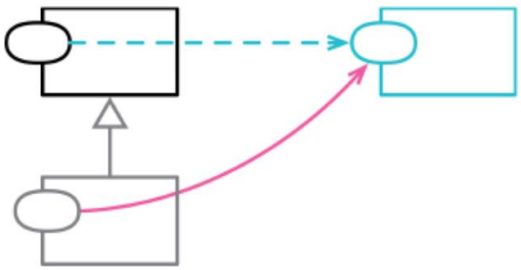
زمانی که درکلاس ها تکرار داریم وکلاس ها با هم مشترک هستند با اعمال این الگو اشتراکات به فوق کلاس منتقل می شوند و ساختار توارثی ایجاد می شود. این فوق کلاس می تواند اینترفیس هم داشته باشد. از این پس با هرگونه تغییری در موارد مشترک تنها کافی است فوق کلاس تغییرکند به علاوه با وابسته کردن کلاینت های استفاده کننده ازکلاس ها به اینترفیس می توان از انتشارات تغییرات به آنها هم جلوگیری نمود.

با انتقال اشتراکات به فوق کلاس duplication از بین می رود به علاوه گاهی می توان با استاندارد کردن متد ها وکارهایی که درکلاس است code bloat راکه تکرار عینی کد نیست را هم از بین برد و آن را درکلاس پدر قرار داد البته انجام آن پیچیده تر از حالت عادی خواهد بود.

الگوی دوم:Replace Superclass with Delegate

مسئله: زیرکلاسی داریم که از تمام اینترفیس فوق کلاس استفاده نمی کند و تنها از بخش هایی از داده های فوق کلاس استفاده می کند.

راه حل: ارتباط توارثی بین اینها را قطع کرده، زیرکلاس تبدیل به سرور شده وکار مورد نظرش را به چیزی که قبلا فوق کلاس بود delegate می کند.



اگر اشیایی داشته باشیم که رفتارشان از دسته ای به دسته دیگر متفاوت است یک مکانیزم برای بیان اینها توارث است. مشترکات را در فوق کلاس قرار داده و به زیرکلاس های آن اجازه می دهیم ویژگی های مورد نیازشان را override کنند. اما توارث معایبی هم دارد. اگر بیش از یک دلیل برای تنوع چیزی داشته باشیم از توارث تنها برای یک وجه آن می توان استفاده کرد و برای هرکدام باید زیرکلاس تعریف کرد و نمی توان هر دو تنوع را همزمان داشت.

مشکل دیگر این است که توارث رابطه قوی بین کلاس ها ایجاد می کند و هر تغییر در فوق کلاس می تواند در فرزندان آن اثر منفی بگذارد و در آنها تغییر مخرب ایجاد کند. راه حل هر دو این مشکلات delegation است. می توان برای دلایل مختلف به تعداد زیادی کلاس delegate انجام داد.

این یک رابطه عادی بین کلاس ها است و با برقراری رابطه به کمک اینترفیس می توان coupling را خیلی کمتر از مکانیزم فوق کلاس و زیر کلاس کرد. برای همین جایگزین کردن توارث با delegation بسیار رایج است. نتیجه این کار در نهایت افزایش خوانایی کد خواهد بود.

```
class Order {
  get daysToShip() {
    return this._warehouse.daysToShip;
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return this._priorityPlan.daysToShip;
  }
}
```

```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
      ? this._priorityDelegate.daysToShip
      : this._warehouse.daysToShip;
  }
}

class PriorityOrderDelegate {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

چگونگی بازآرایی:

۱. فیلدی در زیرکلاس برای نگهداری فوق کلاس ایجاد می کنیم. طی مراحل اولیه خود شی جاری را در آن قرار می دهیم.

۲. متدهای زیرکلاس را طوری تغییر می دهیم که با جای this از نمونه فوق کلاس استفاده کنند.

۳. برای متدهایی که از فوق کلاس به ارث رسید و در کلاینت فراخوانی شده اند یک متد delegation ساده در زیرکلاس می سازیم.

٤. رابطه توارث زیرکلاس را با فوق کلاس قطع می کنیم.

٥. فیلدی که ابتدا ساختیم و هم اکنون فوق کلاس در آن است را با نمونه سازی ازکلاس آن جایگزین می کنیم.

معایب اعمال الگو:

- ایجاد یک لایه indirection کارایی را کاهش می دهد.
- نیاز است کلاس تعداد زیادی متد ساده برای delegate کارها پیاده سازی کند.
- کلاس سرور می تواند تبدیل به Middleman شود.
- نیاز به نمونه سازی از delegate می تواند باعث ایجاد سربار شود.
- تغییر از توارث به delegation نیازمند صرف تلاش بیشتر و انجام تست های بیشتر است تا عملکرد کلاس مختل نشود.
- در صورت عدم طراحی مناسب امکان بروز Feature Envy وجود خواهد داشت.

مزایای اعمال الگو:

- افزایش خوانایی، قابلیت فهم و انعطاف پذیری و در نتیجه بالا رفتن قابلیت نگهداری.
- با حذف رابطه توارث coupling بین آنها کم تر می شود.
- کلاس لازم نیست متد هایی که نیاز ندارد را خالی کند و اینترفیس بی مورد در اختیارکلاینتش قرار دهد و ISP برای کلاس برقرار می شود.
- با اعمال الگو جدایی دغدغه ها برقرار می شود و cohesion هم بالا می رود.

- کپسوله بودن و information hiding با انجام delegation برای کلاس سرور اجرا و نمی تواند به جزئیات delegate دسترسی داشته باشد.

الگوهای مرتبط:

- Change Value to Reference

Bad smell هایی که حذف می کند:

- Middle Man
- Insider Trading
- Refused Bequest

شرایط مفید بودن الگو:

زمانی که در ساختار توارثی زیرکلاس داریم که فرزند خلف پدرش نیست و رفتارهای پدر را خالی می کند و به بخشی از اینترفیس پدر هم احتیاج ندارد. در اینجا هرگونه تغییر در پدر می تواند باعث ایجاد مشکل در این زیرکلاس شود و زیرکلاس هم نیازمند تغییر می شود.

در اینجا دیگر رابطه is-a بین فوق کلاس و زیرکلاس برقرار نیست و باید با اعمال الگو و با کنار گذاشتن توارث و استفاده از delegation این مشکل را رفع کرد نتیجه آن این است قابلیت نگهداری بالاتر رفته و تغییرات منتشر شونده نخواهد بود. در اینجا زیرکلاس را نمی توان دیگر با پدر جایگزین کرد و LSP هم نقض شده است و فرزند تعهدات پدر را انجام نمی دهد.

چگونگی مفید بودن الگو:

این مشکل ناشی از استفاده نادرست از توارث است که احتمالا برای reuse بوده است. با اعمال الگو و تبدیل زیرکلاس به سرور و قراردادن فوق کلاس به عنوان delegate و واسپاری کارها به آن مشکل را رفع و خوانایی را بهبود و قابلیت نگهداری را افزایش می دهیم.

بعد از اعمال الگو چون ساختار توارثی نیست نیاز نیست کلاسی به تعهدات پدر پایبند باشد و نیاز نیست رابطه is-a برقرار باشد و نیاز نیست بتواند جایگزین پدر بشود. به علاوه قوی ترین نوع coupling هم که توارث است دیگر نیست و coupling به سطح delegation کاهش پیدا کرده است و این مفید است. به علاوه زیرکلاسی که الان سرور شده cohesive تر هم شده است.

مقایسه:

تغییر در ساختار های توارثی به دلیل تغییرات دشوار شده است. در Extract Superclass ساختار توارثی نداریم و نبود آن مشکل ساز شده است و با ایجاد فوق کلاس و با انتقال اشتراکات به فوق کلاس ساختار توارثی ایجاد می کنیم. در Replace Superclass with delegate ساختار توارثی داریم و مشکل ساز شده است و باید آن را از بین ببریم. در Extract Superclass در نتیجه اعمال الگو زیرکلاس ها دیگر تکرار ندارند پس انجام تغییرات در آنها ساده خواهد بود و هرکدام نیاز به تغییر داشته باشند را در همان کلاس انجام می دهیم. تغییرات مشترک هم محدود به فوق کلاس خواهد بود به علاوه DIP هم می تواند برقرار شود و کلاینت هم از تغییرات مصون بماند. در Replace Superclass with delegate ساختار توارثی معیوب که احتمالا ایجاد آن به خاطر reuse بوده است باعث شده زیرکلاس با تغییر در فوق کلاس دچار مشکل شود. علت هم این است که زیرکلاس رابطه is-a با فوق کلاس ندارد و فرزند خلف پدرش نیست و تعهدات پدر را انجام نمی دهد.

در اینجا ساختار توارثی را از بین می بریم و با تبدیل فوق کلاس به delegate و زیرکلاس به سرور و واسپاری کارهایی که در آن زیرکلاس به فوق کلاس نیاز دارد به delegate مشکل را برطرف می کنیم. در نتیجه اجرای هر دو مورد به دلیل اینکه تغییرات انتشار نمی یابد قابلیت نگهداری بهتر می شود. در Extract Superclass حتما باید با طراحی دقیق این کار را انجام داد وگرنه در صورت عدم رعایت اصول شی گرایی مشکلی که نتیجه آن اعمال Replace Superclass with delegate است پیش خواهد آمد. در مورد Replace Superclass with delegate هم باید مراقب بود دچار مشکل Middleman نشد و با دقت این کار را انجام داد.

منابع:

۱. Refactoring: Improving the Design of Existing Code, Martin Fowler, with Kent Beck ۲۰۱۸

۲. Dive Into Refactoring Dive Into Refactoring ۲۰۱۹