الگوها در مهندسی نرم افزار

دکتر رامان رامسین

تمرین سوم

مهدی حاجی‌محمدعلی - 402211615

تابستان ۱۴۰۳

فهرست مطالب

فهرست مطالب 2
مقدمه 1
موقعیت اول: پیچیدگی کد و حلقه 2
توصیف Bad Smell 2
الگوی اول: Split Loop 3
بررسی الگو 3
شرایط اعمال الگو 3
چگونگی مفید بودن الگو در موقعیت 4
چگونگی پیاده‌سازی 5
مزایا 5
معایب 6
الگوهای مرتبط 6
الگوی دوم: Replace Loop with Pipeline 7
بررسی الگو 7
شرایط اعمال الگو 7
چگونگی مفید بودن الگو در موقعیت 8
چگونگی پیاده‌سازی 8
مزایا 10
معایب 10
الگوهای مرتبط 10
مقایسه‌ی دو الگو 11
شباهت‌ها 11
تفاوت‌ها 11
موقعیت دوم: رعایت نکردن اصول شی‌گرایی 12
توصیف Bad Smell 12
الگوی اول: Encapsulate Collection 13
بررسی الگو 13
شرایط اعمال الگو 13
چگونگی مفید بودن الگو در موقعیت 14
چگونگی پیاده‌سازی 14

مزایا 15
معایب 15
الگوهای مرتبط 15
الگوی دوم: Convert Procedural Design to Objects 16
بررسی الگو 16
شرایط اعمال الگو 16
چگونگی مفید بودن الگو در موقعیت 17
چگونگی پیاده‌سازی 17
مزایا 19
معایب 19
الگوهای مرتبط 19
مقایسه‌ی دو الگو 20
شباهت 20
تفاوت 20
موقعیت سوم: انتشار تغییرات 21
توصیف Bad Smell 21
الگوی اول: Separate Domain from Presentation 22
بررسی الگو 22
شرایط اعمال الگو 22
چگونگی مفید بودن الگو در موقعیت 23
چگونگی پیاده‌سازی 23
مزایا 24
معایب 25
الگوهای مرتبط 25
الگوی دوم: Extract Interface 26
بررسی الگو 26
شرایط اعمال الگو 26
چگونگی مفید بودن الگو در موقعیت 27
چگونگی پیاده‌سازی 27
مزایا 28
معایب 28
الگوهای مرتبط 29

مقایسه‌ی دو الگو........ 29
شباهت 29
تفاوت 29

مقدمه

برای پاسخ‌گویی به سوالات، از کتاب Refactoring: improving the design of existing code 2nd edition اثر martin fowler بهره برده شده است. البته چندی از الگوها در کتاب دوم قرار ندارند که به همین منظور از کتاب اول استفاده کرده‌ام. در بخش‌هایی از پاسخ که به صورت مشخص از آن استفاده شده، صفحات استفاده شده نام برده شده است (برای مثال به صورت "کتاب فاولر صفحه‌ی ۱۲۴"). البته اشاره به کتاب قدیمی به علت در دسترس نبود ورژن مناسب آن، بدون نام بردن از صفحه صورت گرفته است.

موقعیت اول: پیچیدگی کد و حلقه

توصیف Bad Smell

یکی از bad smellهای مهم شی گرا، loops است که این موقعیت، به آن اشاره می‌کند. حلقه‌ها، از برنامه‌سازی ساخت‌یافته وارد شی گرایی شده اند اما ساختار و کد را ممکن است پیچیده کنند. معمولا فهم منطق حلقه‌ها سخت است و به خصوص ممکن است چند کار در یک حلقه انجام شود که cohesion آن را کاهش داده و فهم منطق آن را سخت می‌کند.
یکی از جاهایی که استفاده از حلقه بسیار رایج است، پیمایش یک کالکشن است. پیمایش به کمک اندیس‌ها و یا اشیا داخل یک کالکشن و با یک حلقه انجام می‌شود. اما امروزه به کمک pipelineها که در زبان‌های برنامه‌نویسی شی گرای امروزی تعریف شده اند می‌توان منطق آن‌ها را قابل فهم کرد. به این صورت که عملیات‌های اجرا شده در حین پیمایش روی یک کالکشن را به ریزعملیات‌هایی تقسیم می‌کنیم که هر کدام مجزا قابل فهم هستند و این ریزعملیات‌ها را به صورت متوالی و در یک خط لوله روی کالکشن اجرا می‌کنیم. fowler نیز یک لیست مفصلی از آن را داخل وبسایت خود قرار داده است.
همچنین جداسازی کارهای مختلفی که در یک حلقه در حال انجام است می‌تواند به فهم بهتر آن کمک شایانی کند.

الگوی اول: Split Loop

بررسی الگو

این الگو از مجموعه الگوهای بازآرایی moving features است. این مجموعه الگوها به حرکت دادن المان‌های مختلف کد در میان contextهای مختلف کد می‌پردازند. (کتاب فاولر، صفحه ۱۹۷)
این الگو به جداسازی بخش‌های مختلف یک حلقه می‌پردازد. در اینجا ما حلقه‌ای داریم که چند کار مختلف با ماهیت‌های مختلف را در یک حلقه انجام می‌دهد.
پیش از اعمال این الگو، فهم حلقه و رفتار آن پیچیده و سخت است چرا که حلقه بیش از یک کار را انجام می‌دهد. همچنین هر بار که بخواهیم کد را برای یک وظیفه‌مندی تغییر دهیم، لازم است تا هر دو کار اساسی حلقه را متوجه شده و سپس اعمال کنیم. در نتیجه تغییرات در کد سخت است و فهم آن پیچیده است.
به صورت ذاتی، ما این کار را انجام نمی‌دهیم و کارهایی که بتوانند در یک حلقه با یکدیگر انجام شوند را با هم در همان حلقه انجام می‌دهیم و حلقه‌ها را جدا نمی‌کنیم. چرا که به نظر می‌آید که دوباره کاری انجام می‌دهیم.
اینجا ممکن است بگوییم که کد کند می‌شود و دوباره کاری انجام می‌شود و duplication داریم. اما واقعیت این است که این duplication کم تبعات است و تنها حلقه تکرار می‌شود و چون دغدغه‌ها جدا شده اند، تغییرات راحت اعمال می‌شود و مکانیزم کاری آن راحت مشخص می‌شود.
یکی از جاهایی که منجر به سختی فهم کد و گیجی می‌شود، همین loop است. حلقه آزار دهنده است و بخصوص nested آن منجر به فهم سخت برنامه می‌شود. کدهای cryptic خیلی اوقات به کمک همین loop انجام می‌شود.

شرایط اعمال الگو

در حالتی این الگو را اعمال می‌کنیم که حلقه‌ی ما، cohesive نباشد. در واقع چندین کار مختلف را در یک حلقه انجام می‌دهیم به گونه‌ای که فهم حلقه سخت شده است و پیچیدگی کد را افزوده است. برای مثال، فرض کنید که یک لیست از همکاران در یک شرکت داریم و می‌خواهیم روی این افراد، میانگین سنی، مجموع حقوق دریافتی افراد و موارد دیگر را محاسبه کنیم. این دو مفهوم (محاسبه‌ی میانگین سنی و محاسبه‌ی مجموع حقوق دریافتی) ذاتا دو کار مجزا از یکدیگر هستند که به واسطه‌ی پیمایش روی لیست افراد، در کنار یکدیگر قرار داده می‌شوند تا به اصطلاح اضافه کاری و کندی کد به وجود نیاید.
در این حالت، هر گاه بخواهیم کد این بخش را بخوانیم، بایستی هم نحوه‌ی محاسبه‌ی مجموع حقوق و هم نحوه‌ی محاسبه‌ی میانگین سنی را مجزا درک کنیم در حالی که هر دو در یک حلقه انجام می‌شوند و در نتیجه پیچیدگی کد را افزایش داده اند و فهم آن برای ما سخت می‌شود.
در نتیجه لازم است فرایند محاسبه‌ی میانگین سنی و مجموع حقوق را از یکدیگر جدا کرده و هر کدام را در یک حلقه‌ی مجزا انجام دهیم. بعدتر درباره مزایا و معایب این کار نیز صحبت می‌کنیم.

البته باید توجه کرد که در صورتی که حلقه روی یک لیست بسیار بزرگی پیمایش می‌کند به طوری که جداسازی حلقه‌ها، آن را به گلوگاه performance کد تبدیل می‌کند، نباید این کار را انجام دهیم.

چگونگی مفید بودن الگو در موقعیت

این الگو یکی از روش‌های حل bad smell نام برده شده یعنی پیچیدگی کد به واسطه‌ی حلقه‌ها هستند. البته در این الگو مطابق چیزی که در بخش شرایط گفته شد، تنها در صورتی به کار می‌رود که حلقه در حال انجام کارهای متفاوت است و برای باقی شرایطی که این bad smell را ایجاد می‌کنند مفید نیست.
پیش از اعمال این الگو، کارهای مختلف و متفاوت در قالب یک حلقه مطابق چیزی که در مثال بخش قبل آورده شد، انجام می‌شود. در نتیجه مطابق motivationی که در کتاب fowler (صفحه‌ی ۲۲۷ و ۲۲۸) برای این الگو آورده شده است، فهم کد سخت می‌شود و پیچیدگی کد بالا است. همچنین اعمال تغییرات دشوار است چرا که به واسطه‌ی چندکاره بودن حلقه، لازم است تا برای تغییر در کد، ابتدا حلقه را فهم کنیم که لازم است تا چندین کار مختلف را متوجه شویم که این کار را دشوار می‌سازد.
باید توجه کرد که ما هزینه‌ی performanceی می‌دهیم اما در قبال آن، انعطاف بیشتر و قابل تغییر شدن کد را به دست می‌آوریم. در نتیجه‌ی جداسازی حلقه و کارهای داخلی آن، کد پیچیده به یک کد ساده‌تر و قابل فهم‌تر تبدیل می‌شود.

چگونگی پیاده‌سازی

فاولر برای اعمال این الگو، سه مرحله اصلی و یک مرحله فرعی را مطابق زیر در نظر گرفته است:

1. کپی کردن حلقه
2. شناسایی و حذف تاثیرات جانبی duplicate شدن کد حلقه
3. تست
4. (اختیاری) با اعمال extract function، وظیفه‌مندی هر کدام از حلقه‌ها را به صورت یک تابع در می‌آوریم.

مثال مناسب نیز در شکل زیر آورده شده است.

```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

مزایا

- انعطاف و قابل تغییر شدن کد
- قابل فهم شدن و بالا بردن خوانایی کد
- در صورتی که حلقه یک کار را انجام دهد، می‌تواند یک خروجی مشخص را در قالب یک تابع داشته باشد اما اگر چند کاره باشد، لازم است تا اشیای پیچیده جدید تعریف شود تا خروجی‌های کارهای مختلف کد را در آن قرار دهیم.
- افزایش cohesion حلقه و حتی المان‌های دیگر (تابع و کلاس) در صورت استفاده از extract function

- بالا رفتن قابلیت استفاده‌ی مجدد یا reusability به دلیل استخراج توابع

معایب

- کند شدن احتمالی برنامه هر چند ناچیز. چرا که برای چند بار یک حلقه اجرا می‌شود و پیمایش انجام می‌شود.
- امکان وجود آمدن bottleneck در برنامه در صورت جداسازی حلقه‌های با اجرای بالا
- اگر تعداد عملیات‌های مختلفی که در حلقه انجام می‌شود زیاد باشد، جداسازی حلقه‌ها هم می‌تواند bottleneck شود و هم می‌تواند خود عامل پیچیدگی کد شود. در این حالت استفاده از extract method برای استخراج توابع و بالابردن انعطاف و خوانایی کد ضروری می‌شود.

الگوهای مرتبط

- الگوی extract function می‌تواند پس از اعمال این الگو برای استخراج تابع از وظیفه‌مندی حلقه‌ها به کار رود.
- در صورتی که داخل کد چند کار متفاوت انجام شود، ممکن است که ترتیب فراخوانی عملیات‌های داخلی حلقه‌ها مناسب نباشد. در این حالت لازم است با اجرای slide statement، ابتدا statementهای مرتبط با هر وظیفه‌مندی حلقه را در کنار یکدیگر قرار دهیم تا عملیاتی که حلقه انجام می‌شود را بهتر متوجه شده و سپس عملیات split loop را اعمال کنیم.
- الگوی دوم این مجموعه یعنی الگوی replace loop with pipeline نیز از جهت رفع bad smell مربوطه (loops) با این الگو در ارتباط است.

الگوی دوم: Replace Loop with Pipeline

بررسی الگو

این الگو از مجموعه الگوهای بازآرایی moving features است. این مجموعه الگوها به حرکت دادن المان‌های مختلف کد در میان contextهای مختلف کد می‌پردازند. (کتاب فاولر، صفحه ۱۹۷)
استفاده از حلقه‌ها برای پیمایش و اعمال توابع روی یک کالکشن از اشیا، در زبان‌های برنامه‌سازی یک امر رایج بوده است. حلقه‌ها از زبان‌های ساخت‌یافته وارد شی گرایی شدند اما استفاده از آن‌ها منجر به کاهش خوانایی کد می‌شود. در زبان‌های شی‌گرایی کنونی، یک ساخت جدیدی تحت عنوان pipeline تعریف شده است که امکان تعریف عملیات برای پیمایش روی یک کالکشن را در اختیار ما قرار می‌دهد که به کمک آن می‌توان مراحل مختلف عملیاتی تعریف کرد و با انعطاف و خوانایی بالا، مجموعه‌ای از عملیات‌ها را با ترتیب مشخص روی یک مجموعه‌ی شی اجرا کرد.
در این الگو، سعی می‌شود تا استفاده از حلقه برای پیمایش و اعمال عملیات روی مجموعه‌ای اشیا را با استفاده مناسب از عملیات‌های تعریف شده‌ی pipeline جایگزین کنیم. در نتیجه پیچیدگی‌هایی که حلقه بر روی کد و خوانایی آن ایجاد می‌کند را کاهش می‌دهیم.

شرایط اعمال الگو

این الگو در شرایطی استفاده می‌کنیم که برای پیمایش یک مجموعه‌ای از اشیا، عملیات‌های پیمایشی داخل یک حلقه تعریف شده است و در واقع از حلقه برای پیمایش collection استفاده کرده‌ایم. ممکن است در هنگام پیمایش collection، بخواهیم بر اساس شرط (یا شرایطی) اشیا را filter کنیم، آن‌ها را به اشیا جدیدی تبدیل کنیم و یا به ازای هر کدام، عملیات خاصی را انجام دهیم.
در این حالت، کد حلقه می‌تواند بیش از حد پیچیده و ناخوانا شود و رهگیری و تغییر در منطق آن سخت شود. چرا که پردازش‌های مختلفی روی مجموعه شی انجام می‌شود و فهم هر کدام از این پردازش‌ها می‌تواند با توجه به کد پیچیده باشد.
در نتیجه به جای پیمایش کالکشن به کمک حلقه، در صورتی که بتوانیم از عملیات‌های از پیش تعریف‌شده‌ی pipeline در زبان استفاده کنیم، از یک pipeline پردازشی بهره می‌بریم تا عملیات‌های پردازشی مختلفی که می‌خواهیم اجرا کنیم را به ترتیب و در یک خط لوله‌ی پرداشی اجرا کنیم.
باید توجه کرد که ممکن است عملیاتی را در حلقه تعریف کرده باشیم که در پردازش‌های پیش‌فرض خط لوله تعریف نشده اند. در این شرایط نمی‌توانیم از pipeline به جای حلقه استفاده کنیم. هر چند چنین عملیات‌هایی تقریبا کمیابند و اکثر عملیات‌های پیمایشی تاکنون تعریف شده اند.

چگونگی مفید بودن الگو در موقعیت

در حالت پیشین، از حلقه برای پیمایش یک مجموعه شی استفاده شده است. برای مثال فرض کنید می‌خواهیم در لیست کارمندان شرکت پیمایش کنیم و از میان کارمندان، افرادی را که برنامه‌نویس هستند را جدا کرده و نام آن‌ها را خروجی دهیم. ممکن است کارهای پیچیده‌تری نیز تعریف کنیم برای مثال برای آن‌ها میانگین حقوق حساب کنیم و یا سطح سازمانی و ... آن‌ها را با یکدیگر مقایسه کنیم.
انجام عملیات‌های گفته شده در حالت عادی به کمک یک حلقه و با استفاده از ساخت‌های زبانی مانند عملیات شرطی و همچنین تعریف یک لیست و افزودن نام افراد در هنگام پیمایش می‌تواند انجام شود.
این الگو پیشنهاد می‌دهد که حلقه را با تعریف عملیات‌های پردازشی جایگزین کنیم. مراحل پردازشی در قالب یک خط لوله و با ترتیب مناسب تعریف می‌شود. برای نمونه، در مثالی که بالاتر اشاره شد، ابتدا باید یک عملیات filter که از عملیات‌های پیش‌فرض pipeline هست استفاده می‌کنیم و شرط را داخل آن تعریف می‌کنیم. سپس به کمک عملیات map، می‌توانیم به ازای هر کدام از اشیایی که فیلتر شده‌اند، یک خروجی جدید داشته باشیم. در اینجا کافی است که نام افرادی که از فیلتر گذر کردند را در map به عنوان خروجی تعریف کنیم.
به این صورت، خوانایی کد بسیار بالا می‌رود. چرا که کاملا مشخص می‌شود که در هنگام پیمایش collection، دقیقا داریم به چه ترتیبی چه عملیات‌هایی را اعمال می‌کنیم. خوانایی pipeline پردازشی نسبت به حلقه بسیار بالاتر است و انعطاف‌پذیری کد را نیز بالا می‌برد. چرا که تغییرات کد نیز بسیار نسبت به گذشته ساده‌تر می‌شود.
مجموعه‌ی نسبتا کاملی از عملیات‌های پردازشی توسط fowler در وبسایت او تعریف شده که می‌تواند مفید باشد.

چگونگی پیاده‌سازی

در کتاب فاولر، مراحل زیر برای اعمال این الگو تعریف شده است:

- یک متغیر جدید به برای collection مورد پردازش تعریف می‌کنیم.
- از بالاترین بخش حلقه شروع می‌کنیم و عملیات‌های تعریف شده روی مجموعه را به ترتیب عملیات‌های پردازشی خط لوله روی متغیر تعریف شده‌ی collection تبدیل می‌کنیم و سپس آن را تست می‌کنیم. سپس عملیات را حذف می‌کنیم.
- در صورتی که تمام عملیات‌ها حذف شد، حلقه را حذف می‌کنیم.

یک مثال نیز به شکل زیر آمده است. در این مثال قصد داریم مجموعه‌ی داده‌ی ورودی officeهای مختلف یک سازمان را دریافت کنیم و سپس شرکت‌هایی که در هند هستند را جدا کرده و شهرها و شماره‌ی تلفن آن‌ها را خروجی دهیم. ورودی به شکل یک فایل csv است که بایستی پس از دریافت آن، کمی نیز تمیز شود.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  for (const line of lines) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

پس از اعمال الگو و استخراج خط لوله داریم:

```
function acquireData(input) {
  const lines = input.split("\n");
  return lines
        .slice  (1)
        .filter (line   => line.trim() !== "")
        .map    (line   => line.split(","))
        .filter (fields => fields[1].trim() === "India")
        .map    (fields => ({city: fields[0].trim(), phone: fields[2].trim()}))
        ;
}
```

همانطور که می‌بینید پس از اعمال تغییرات، به خوبی مشخص می‌شود که ابتدا سعی می‌کنیم که خطوطی که خالی هستند را فیلتر کنیم، سپس فیلدهای مختلف هر سطر فایل csv را به کمک یک تابع map جداسازی می‌کنیم و سپس بر اساس شهر آن‌ها را فیلتر کرده و در نهایت خروجی را به کمک یک map تولید می‌کنیم.
همانطور که مشخص است، خوانایی کد پس از بازآرایی به شکل فوق العاده‌ای بهتر می‌شود و ایجاد تغییرات نیز در آن ساده خواهد بود.

مزایا

- انعطاف‌پذیری بالا پس از اعمال الگو
- افزایش خوانایی
- تنوع بالا در تعریف عملیات‌های پردازشی روی مجموعه
- کوتاه‌تر شدن کد
- امکان جابجایی ترتیبی و افزودن یا تعریف عملیات‌های جدید پردازشی بدون کاهش کیفیت کد

معایب

- دیباگ کردن و تشخیص خطا در کد نسبت به حلقه سخت‌تر می‌شود.
- برخی از زبان‌های برنامه‌سازی شی گرا از آن پشتیبانی نمی‌کنند.
- تعریف تعداد عملیات زیاد در یک خط لوله خود موجب کاهش خوانایی کد می‌شود و پیچیدگی کد را افزایش می‌دهد که نهی می‌شود و بهتر است به جای آن، pipeline را مجددا ریفکتور کرده و بخش‌هایی از آن را جدا کنیم.

الگوهای مرتبط

- الگوی قبلی این مجموعه یعنی الگوی split loops نیز از جهت رفع bad smell مربوطه (loops) با این الگو در ارتباط است.

مقایسه‌ی دو الگو

شباهت‌ها

هر دو الگو در دسته‌ی moving features از الگوهای بازآرایی هستند که برای رفع bad smell حلقه‌ها به کار می‌روند. هر دو الگو به بهبود خوانایی و انعطاف بالای کد کمک می‌کنند.

تفاوت‌ها

هر دو الگو در شرایط اعمال تفاوت جدی دارند. الگوی split loops، در حالتی که یک حلقه با چند عملیات non cohesive داریم به کار می‌رود اما الگوی replace loop with pipeline، در حلقه‌هایی که برای پیمایش collection استفاده شده اند اعمال می‌شود.
در نتیجه‌ی اعمال الگوی اول، تعدادی حلقه به حلقه‌های موجود افزوده می‌شود و ممکن است حتی حلقه‌ها خود به کمک extract function، از کد جدا شوند اما در نتیجه‌ی اعمال الگوی دوم، حلقه حذف می‌شود و حلقه‌ی جدیدی افزوده نمی‌شود. در واقع الگوی دوم با حذف حلقه، پیچیدگی کد را کم می‌کند اما الگوی اول با افزودن حلقه، مشکل ناخوانا بودن حلقه‌ی اولیه‌ی بزرگ را حل می‌کند.
الگوی اول منجر به کاهش performance کد می‌شود چرا که سربار پردازشی با افزودن تعداد حلقه‌ها به وجود می‌آید اما الگوی دوم، منجر به سخت شدن debug و رهگیری خطا می‌شود.

موقعیت دوم: رعایت نکردن اصول شی‌گرایی

توصیف Bad Smell

موقعیت تعریف شده خود مستقیما یک bad smell نیست اما می‌تواند شامل چندین bad smell از مجموعه‌ی تعریف شده در درس شود. از مهم‌ترین اصول شی‌گرایی می‌توان به encapsulation و ساختن کپسول‌های داده رفتار، inheritance و همچنین polymorphism اشاره کرد. همچنین از اصول ۶گانه‌ی OCP DIP ISP LSP PLK CRP نیز می‌توان نام برد.

افرادی که به تازگی از فضای برنامه‌نویسی ساخت‌یافته و یا functional وارد شی گرایی می‌شوند، مسیر طولانی برای تعریف درست کپسول‌های داده رفتار، استفاده مناسب از توارث و کاربردی کردن چندریختی را طی می‌کنند. در این مسیر ابتدا کلاس‌های داده‌ای و کلاس‌های رفتاری شروع می‌کنند که خود می‌تواند به Large Class و Data Class تبدیل شوند که از bad smellهای مهم هستند. تعریف نادرست کلاس‌های داده رفتار علاوه بر این دو خود می‌تواند عامل feature envy و insider trading نیز باشد. تعریف نادرست توارث می‌تواند به refused bequest منجر شود و عدم استفاده از چند ریختی ممکن است به وقوع repeated switches بیانجامد.

Divergent change, shotgun surgery, message chain, middle man از دیگر bad smellهایی هستند که می‌توانند نتیجه‌ی استفاده‌ی نادرست از اصول شی‌گرایی ۶ گانه و مبانی شی‌گرایی باشند.

برای هر کدام از این bad smellها، موقعیت‌هایی متفاوتی می‌توان متصور بود که بسته به اصولی که در شی‌گرایی نقض شده، می‌توان برای آن‌ها راه حل ارائه داد که در ادامه به دو الگوی اشاره شده می‌پردازیم.

الگوی اول: Encapsulate Collection

بررسی الگو

این الگو از دسته‌ی الگوهای بازآرایی Encapsulation است. این دسته الگوها به یکی از اصول مهم شی‌گرایی یعنی کپسوله‌سازی داده رفتار می‌پردازند. مطابق توضیح کتاب فاولر (صفحه‌ی ۱۶۱)، یکی از مهمترین شاخص‌های تعریف ماژول‌های مختلف نرم افزار، تشخیص آن ا ست که هر ماژول چه داده‌هایی را باید از ماژول‌های دیگر پنهان کند. الگوی Encapsulate collection و encapsulate record از الگوهایی هستند که یک داده‌ساختار را کپسوله می‌کنند.

در هنگام کپسوله سازی اشیا، گاها کپسوله‌سازی مجموعه‌های یا collections به درستی صورت نمی‌پذیرد. در واقع در بسیاری از زبان‌های شی‌گرای امروزی، ارجاعات به صورت pass by reference هستند و در نتیجه، اگر برای collectionهای یک شی get تعریف کنیم، امکان تغییر روی تک تک اعضای collection را برای سرویس‌گیرنده فراهم می‌کنیم. در واقع با get کردن کل collection، سرویس‌گیرنده می‌تواند تک تک اعضا را تغییر دهد، به مجموعه شیئی بیافزاید و یا شیئی را modify کند.

البته یک نکته‌ای در این حوزه باید مورد توجه قرار گیرد. همانطور که پیش‌تر در بخش قبل هم اشاره شد، ممکن است پیمایش و استفاده از دستوراتی که interface مربوط به collection در اختیار کاربر قرار می‌دهد در سرویس‌گیرندگان مورد استفاده قرار گرفته باشد. در این حالت، لازم است تا امکانی را فراهم آوریم که علاوه بر جلوگیری از تغییر در collection، بتوان خود collection را نیز دسترسی پیدا کرد و عملیات‌های وابسته به واسط collection را به صورت write protected روی آن اعمال کرد.

شرایط اعمال الگو

در حالتی این الگو را اعمال می‌کنیم که بخواهیم یک شی حاوی یک collection را کپسوله‌سازی کنیم. در این حالت با توجه به مطالبی که گفته شد، استفاده از getter برای collection می‌تواند امکان تغییر ناخواسته را روی مجموعه‌ی اشیا فراهم آورد که مطلوب نیست. به همین دلیل، لازم است تا کل مجموعه به صورت pass by reference در اختیار سرویس گیرنده قرار نگیرد و از طرفی امکان افزودن و تغییر بر روی collection از طریق شی سرویس داده شود.

نکته‌ی دیگری که در شرایط اعمال این الگو باید توجه کرد این است که سرویس‌گیرندگان ممکن است به collection برای عملیات‌های پردازشی و پیمایشی نیاز داشته باشند. در این حالت، لازم است تا به گونه‌ای این امکان در اختیار سرویس‌گیرنده قرار گیرد که بدون امکان تغییر بر روی collection، بتواند به آن دسترسی داشته و روی اشیای آن عملیات تعریف شده را اجرا کند.

چگونگی مفید بودن الگو در موقعیت

ابتدا باید دقت کنیم که این الگو به مسئله‌ی کپسوله‌سازی اشیا و داده‌ساختارها که از اصول شی‌گرایی است می‌پردازد. در نتیجه اصل شی‌گرایی که در اینجا نقش شده، encapsulation است. همانطور که در بخش بررسی الگو نیز گفته شد، این الگو از دو جهت به مشکل کپسوله‌سازی collectionها می‌پردازد. مسئله‌ی اول، تغییر در اجزای collection است و مسئله‌ی دوم، نیاز سرویس‌گیرندگان بیرونی به واسط collection برای اعمال پردازش روی آن است.

برای حل مسئله‌ی تغییر در اجزای collection این الگو پیشنهاد می‌کند که توابع کمکی add و remove روی collection تعریف شود تا برای دسترسی سرویس‌گیرندگان به تغییر اجزا، بتوان از آن بهره برد. در این حالت می‌توان دسترسی سرویس‌گیرندگان به اجزای داخلی را کپسوله کرد و کنترل لازم بر روی آن اعمال شود و در نتیجه collection به خوبی نسبت به تغییر در اجزا encapsulate می‌شود.

برای حل مسئله‌ی دوم، در کتاب فاولر (صفحات ۱۷۰ و ۱۷۱) پیشنهاد می‌کند تا به گونه‌ای دید سرویس‌گیرندگان را برای write کردن روی collection محدود کنیم و یک protected view را در اختیار سرویس‌گیرنده قرار دهیم. برای این کار راه‌های مختلفی را می‌توان در نظر گرفت. به جای collection، یک copy از آن را در اختیار سرویس‌گیرنده قرار دهیم (pass by value) و یا یک read-only proxy از collection ایجاد کنیم که دسترسی write روی collection را محدود می‌کند و تنها امکان خواندن آن را در اختیار سرویس‌گیرنده قرار می‌دهد. همچنین استفاده از یک ImmutableCollection می‌تواند گزینه‌ی مناسبی باشد که امکان modify کردن نداشته باشد تا سرویس‌گیرندگان امکان تغییر روی collection را نداشته باشند.

چگونگی پیاده‌سازی

مراحلی که فاولر برای پیاده‌سازی این الگو نام می‌برد عبارتند از:

- ابتدا باید encapsulate variable را روی شی اعمال کنیم تا کل شی یک کپسوله‌سازی ابتدایی شود.
- توابع تغییر در اجزای collection را تعریف می‌کنیم (توابع add و remove)
- تمامی usageهای collection را در کد پیدا کرده و در سرویس‌گیرندگان به جای تغییر روی اشیا collection، از توابع add و remove بهره می‌بریم.
- عملگر getter برای collection را تغییر می‌دهیم و به جای collection، بایستی از ImmutableCollection یا یک کپی از آن را در اختیار کاربر قرار دهیم. گزینه‌ی دیگر نیز می‌تواند read-only proxy باشد.
- تست مناسب می‌نویسیم و آزمایش می‌کنیم.

مثال مناسب آن نیز در شکل زیر آورده شده است.

```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse)    { ... }
  removeCourse(aCourse) { ... }
```

مزایا

- جلوگیری از دسترسی ناخواسته‌ی سرویس‌گیرندگان به اجزای collection
- امکان خواندن collection و استفاده از interface آن بدون تغییر روی collection
- امکان اعمال کنترل دسترسی روی collection برای اشیا مختلف
- بهبود encapsulation در نتیجه‌ی اخفای collection و تغییر در آن از کلاینت‌ها

معایب

- در صورتی که لازم باشد تا بخش‌های مختلف کد روی collection و اجزای آن تغییر ایجاد کنند، با یک سطح indirection این کار را انجام می‌دهند.
- یکی از مواردی که pass by value را نامطلوب می‌کند، نیاز بخش‌هایی از سیستم به تغییر در شی است. در نتیجه ممکن است یک کد duplicate در بخش‌های مختلفی دیده شود که سعی در تغییر collection می‌کنند که این خود نامطلوب خواهد بود و منجر به بالا رفتن coupling است و در شی‌گرایی نامطلوب است.

الگوهای مرتبط

- این الگو با الگوهای encapsulate record و encapsulate variable از جهت بهبود وضعیت کپسوله‌سازی اشیا مرتبط است. همچنین خود می‌تواند در نتیجه‌ی encapsulate variable اتفاق بیفتد.

الگوی دوم: Convert Procedural Design to Objects

بررسی الگو

این الگو از دسته الگوهای بازآرایی big refactoring است. این دسته از الگوها، برخلاف الگوهای دیگر بازآرایی به سادگی و در یک موقعیت خاص معمولا به کار نمی‌روند بلکه به مدت زمان بیشتری برای پیاده‌سازی نیازمندند و معمولا به طور آنی به ثمر نمی‌رسند. حجم کدی که تغییر می‌کند معمولا زیاد است و بخش‌های مختلفی درگیر می‌شوند. اما اعمال آن‌ها بسیار سودمند است چرا که امکان تغییر و maintenance مناسب را فراهم می‌آورد.
هنگامی که مهندسین نرم افزار در ابتدای کار با object oriented از یک پارادایم procedural وارد می‌شود، ابتدا کلاس‌های خود را به شکل نامناسبی تعریف می‌کند. در واقع کلاس‌هایی با داده‌های کم و رفتار زیاد و کلاس‌هایی با داده‌های زیاد و رفتار کم دیده می‌شود.
در برنامه‌سازی شی‌گرا، هدف ایجاد کپسول‌های داده رفتاری است که رفتارهای مربوط به داده‌های یک شی، در نزدیک‌ترین حالت به شی خود (داخل خود کلاس) تعریف می‌شوند. در نتیجه به این صورت وابستگی کلاس‌های دیگر به دادگان یک کلاس بسیار کم می‌شود به خصوص اگر شی بتواند خود سرویس‌دهنده به دیگر اشیا باشد. در واقع اشیا دیگر که با دادگان یک شی سر و کار دارند، لازم است تا از خود شی سرویس بگیرند و شی با برقراری encapsulation مناسب، کنترل کافی روی داده‌ی خود را اعمال کند.
در نتیجه پیش از اعمال این الگو، معمولا کلاس‌ها به یکدیگر وابستگی شدید دارند و احتمالا cohesion کم در میان برخی قسمت‌هایی که بروز رفتاری بیشتری دارند دیده می‌شود. این الگو پیشنهاد می‌کند تا با نزدیک کردن داده به رفتار خود و حذف کلاس‌های داده‌ای و رفتاری و ایجاد کپسول‌های داده رفتار، این اصل مهم شی‌گرایی را برقرار کرد. البته اجرای این الگو همانطور که گفته شد به زمان زیاد و دقت بالا نیازمند است.

شرایط اعمال الگو

در حالت پیشین، معمولا تعداد زیادی کلاس با داده‌ی کم و رفتار زیاد و تعداد زیادی کلاس با رفتار کم و داده‌ی زیاد دیده می‌شود. در واقع می‌توان گفت پیش از اعمال الگو، می‌توان تعداد زیادی data class و large class در کد مشاهده کرد. Large classها کلاس‌های رفتاری هستند که cohesive نیستند و معمولا رفتارهای مختلفی را که به دادگان متعددی نیاز دارد در خود نگهداری می‌کنند. این کلاس‌ها وابستگی شدیدی به کلاس‌های داده‌ای دیگر دارند و در نتیجه وضعیت سیستم پیش از اعمال الگو از جهت وابستگی و همبستگی (cohesion & coupling) اصلا مناسب نیست.
بسته به شرایط، معمولا در این گونه سیستم‌ها feature envy نیز به شدت رایج است. چرا که در این bad smell نیز رفتار از داده جدا است و یک المان مدام به فیچرها یا داده‌های یک المان دیگر نیز چشم دارد. در نتیجه یک وابستگی شدید بین دو المان به وجود می‌آید.

مهم‌ترین نکته‌ای که باید توجه کرد، رعایت نکردن encapsulation و درست نشدن کپسول‌های داده رفتار است. در واقع این الگو در شرایطی اعمال می‌شود که این اصل مهم شی‌گرایی رعایت نشود. Encapsulation در این سیستم‌ها به راحتی نقض می‌شود چرا که کلاس‌های داده‌ای مدام اطلاعات خود را در اختیار کلاس‌های رفتاری دیگر قرار می‌دهند.

چگونگی مفید بودن الگو در موقعیت

این الگو به حل مسئله‌ی encapsulation و تعریف نادرست کپسول‌های داده رفتار می‌پردازد که یکی از مهمترین اصول شی‌گرایی است. همانطور که در بخش قبل نیز گفته شد، bad smellهای متعددی می‌توانند از حالت پیشین این الگو پدید آمده باشند.

این الگو با نزدیک کردن رفتار به داده‌ی خود و حذف کلاس‌های بزرگ non cohesive و large classها، مشکل encapsulation، وابستگی بالا و cohesion نامناسب در کد را حل می‌کند.

برای حل این مسئله، الگو پیشنهاد می‌دهد که از دو روش bottom up و top down بهره بگیریم. روش bottom up با شروع از پایگاه داده‌ی رابطه‌ای که به خوبی نرمال‌سازی شده و تا حد قابل قبولی جداسازی داده‌ها از یکدیگر در آن به درستی انجام شده، سعی می‌کند تا اشیا داده‌ای مناسب را تعریف کند. سپس با حرکت در کد و یافتن رفتارهای مختص اشیا، به کمک extract method و move method، رفتار را به داده‌ی خود نزدیک‌تر می‌کند. به این صورت اشیا خود سرویس‌دهنده به دیگر اجزای سیستم می‌شوند و مشکل encapsulation و coupling تا حد خوبی برطرف می‌شود. با ادامه‌ی این کار، نهایتا کلاس‌های رفتاری به تعدادی facade و یا واسط تبدیل می‌شوند و یا به کلی حذف می‌شوند. در نتیجه اصول encapsulation شی گرایی به خوبی برقرار می‌شود و کلاس‌های داده‌ای و رفتاری که با دید procedural نوشته شده اند، تبدیل به کپسول‌های داده رفتار می‌شوند. در نتیجه مسئله‌ی نقض اصول شی‌گرایی را به این شکل این الگو حل می‌کند.

در روش top down که بالاتر اشاره شد، با توجه به دامنه‌ی مسئله و با شروع از آن و مدل‌سازی آن، سعی می‌کنیم تا مدل‌ها و کد ایجاد شده در اثر روش bottom up را بهبود دهیم. این دو روش را در کنار یکدیگر قرار داده و در نهایت یک ساختار مناسب از اشیا که با دید object oriented و به صورت کپسول‌های داده‌رفتار نوشته شده اند ایجاد می‌شود.

چگونگی پیاده‌سازی

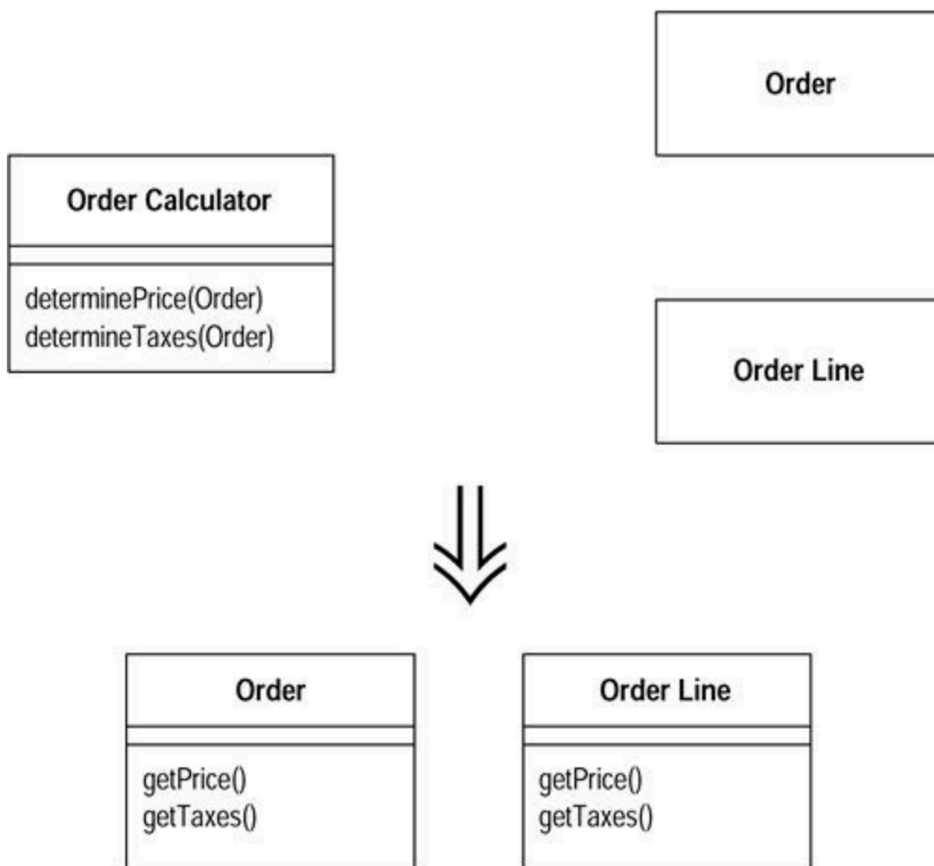
بخشی از چگونگی پیاده‌سازی در بخش قبل و با تعریف روش‌های top down و bottom up توضیح داده شد. در کتاب چاپ اول فاولر، او مراحل زیر را برای اعمال الگو توصیف می‌کند:

- ابتدا کلاس‌های داده‌ای را باید بر اساس داده‌هایی که داریم تعریف کنیم. معمولا برای این کار در صورت وجود یک پایگاه‌داده‌ی رابطه‌ای به آن مراجعه می‌کنیم.
- تمامی کدهای procedural را به یک کلاس منتقل می‌کنیم.

- توابع بزرگ را با کمک extract method کوتاه می‌کنیم. پس از هر جداسازی، آن را با move method به کلاس داده‌ی مربوط به خود منتقل می‌کنیم.
- مرحله‌ی قبل را انقدر ادامه می‌دهیم تا تمامی کدها به کلاس‌های مربوطه‌ی خود منتقل شوند. اگر کلاس اولیه که تمامی کدها را در خود داشت یک کلاس تمام procedural بود، قاعدتا کلاس باید حذف شود و اگر نمودهای شی‌گرا داشت، بایستی تبدیل به یک کپسول داده رفتار شود.

بایستی توجه کرد که همانطور که در کلاس گفته شد. این مراحل کافی نیست و لازم است تا پس از اعمال این مراحل، با کمک روش top down و با شروع از دامنه‌ی مسئله، طراحی جدید را بهبود و بازآرایی مجدد کرد تا در نهایت به بهترین طراحی رسید.

مثال کتاب نیز در شکل زیر آورده شده است. در این مثال، order و order line فرض شده است که هر کدام از قبل وجود داشته اند. اما این ممکن است که این کلاس‌ها خود کاملا کلاس داده‌ای نباشند و از ابتدا با توجه به پایگاه داده آن‌ها را تعریف کنیم. پس از تعریف کلاس‌های داده‌ای با نزدیک‌سازی رفتارهای کلاس رفتاری Order Calculator به دو کلاس‌داده‌ای، این کلاس حذف می‌شود:



مزایا

- کاهش coupling میان کلاس‌های رفتاری و کلاس‌های داده‌ای
- افزایش cohesion با حذف کلاس‌های large class
- بهبود خوانایی و انعطاف‌پذیری کد
- جلوگیری از نقض encapsulation با تعریف مناسب کپسول‌های داده رفتار
- نزدیکی رفتار به داده و جلوگیری از feature envy در سیستم
- بهبود maintainability در سیستم
- بهبود agility با توجه به امکان‌پذیر کردن تغییرات روی کد و بهبود perfective maintenance

معایب

- پیچیدگی و زمان‌بر بودن اعمال الگو با توجه به ماهیت مبهم و حجم گسترده‌ی کد احتمالی. باید توجه داشت که این الگو خود شبیه به یکی از الگوهای reengineering است که ممکن است ماه‌ها به طول بیانجامد.
- در صورتی که مراحل به درستی انجام نشود و برای مثال کپسول‌های داده رفتار به شکل نامناسبی تعریف شوند (داده‌ها و رفتارهایی در کنار یکدیگر قرار گرفته باشند که ذاتا به یکدیگر مرتبط نیستند)، در نتیجه‌ی اعمال این الگو، ممکن است برخی bad smellها مانند feature envy و insider trading رخ دهد که مطلوب نیست.

الگوهای مرتبط

- الگوهای extract method و move method به کرات در این الگو به کار می‌روند.
- این الگو با الگوی Move behaviour close to data از دسته‌ی الگوهای reengineering بسیار ارتباط تنگاتنگی دارد. در واقع می‌توان گفت مفاهیم بسیار نزدیکی را هر دو به کار می‌برند و حتی مراحل اعمال این الگو نیز بسیار شبیه به یکدیگر است.
- الگوی split up god class نیز از دیگر الگوهای بازمهندسی است که با این الگو ارتباط جدی دارد. در واقع می‌توان گفت الگوهای دسته‌ی redistribute responsibilities از الگوهای بازمهندسی بسیار با این الگو در ارتباط هستند.

مقایسه‌ی دو الگو

شباهت

هر دو الگو با وجود شرایط و نحوه‌ی اعمال متفاوت، در بهبود وضعیت encapsulation در شی گرایی سیستم نقش جدی دارند.
پس از اعمال هر دو الگو، تعدادی متد ممکن است به کد افزوده شود. در مواردی، در الگوی اول نیز ممکن است کلاس‌های جدید (مانند CollectionProxy یا ImmutableCollection) تعریف شود و در الگوی نیز ممکن است کلاس‌های داده‌ای جدیدی در حین اعمال الگو پدید آید.

تفاوت

الگوی encapsulate collection از الگوهای ریزدانه‌ی بازآرایی در دسته‌ی encapsulation است اما الگوی دوم، یکی از الگوهای بزرگ بازآرایی است.
الگوی اول، جزیی است و به حل مسئله‌ی تغییرات ناخواسته بر روی collection در صورت رعایت نشدن درست encapsulation در اشیا می‌پردازد. اما الگوی دوم بسیار کلی‌تر است و به صورت کلی به حل مسئله‌ی طراحی procedural و فقدان کلاس‌های داده‌رفتار شی‌گرا می‌پردازد.

موقعیت سوم: انتشار تغییرات

توصیف Bad Smell

تغییرات در یک بخش از کد منجر به انتشار و تغییر کردن بخش‌های دیگر کد می‌شود. این موقعیت بسیار به bad smell معروف shotgun surgery مربوط است. این bad smell نشان می‌دهد که تغییر در کد که بعضا برای بهبود کیفیت و در واقع جراحی و گاهی اوقات برای افزودن ویژگی‌های جدید به کد است، منجر به انتشار تغییرات و تغییر بخش‌های متعدد می‌شود که به آن به آن تغییر منتشر شونده نیز گفته می‌شود. علاوه بر تغییر، در این موقعیت extension نیز منجر به تغییرات متعدد نامربوط می‌شود.

این موقعیت می‌تواند نشانه‌ی انعطاف پایین کد و coupling بالا در سطح کد باشد. به گونه‌ای که کلاس‌های متعدد ارتباط تنگاتنگی با یکدیگر دارند و این ارتباط در سطح concrete برقرار است و در نتیجه به ازای تغییر در هر کلاس، کلاس‌هایی که با او در ارتباط هستند نیز بایستی تغییر کند.

نقض قاعده‌ی OCP و همچنین DIP، بسیار در رخ دادن این وقوع این موقعیت موثر باشد. در واقع با برقراری OCP و DIP، به خوبی وابستگی ماژول‌های مختلف کد در سطوح مختلف (سطح تابع، کلاس، کامپوننت، زیرسیستم و ...) به یکدیگر کنترل می‌شود و در صورتی که وابستگی در سطح concrete باشد و یا ماژول‌ها extension را نپذیرند، این موقعیت می‌تواند رخ دهد.

از آنجایی که در این موقعیت، coupling بسیار شدید است، ممکن است در سیستم message chain نیز دیده شود. هر چند برخلاف موارد پیشین که از علل وقوع این موقعیت هستند، وجود message chain از مواردی است که با این موقعیت می‌تواند correlation داشته باشد.

الگوی اول: Separate Domain from Presentation

بررسی الگو

این الگو از دسته الگوهای بازآرایی big refactoring است. این دسته از الگوها، برخلاف الگوهای دیگر بازآرایی به سادگی و در یک موقعیت خاص معمولا به کار نمی‌روند بلکه به مدت زمان بیشتری برای پیاده‌سازی نیازمندند و معمولا به طور آنی به ثمر نمی‌رسند. حجم کدی که تغییر می‌کند معمولا زیاد است و بخش‌های مختلفی درگیر می‌شوند. اما اعمال آن‌ها بسیار سودمند است چرا که امکان تغییر و maintenance مناسب را فراهم می‌آورد.
هنگامی که مهندسین نرم افزار در ابتدای کار با object oriented از یک پارادایم procedural وارد می‌شود، ابتدا کلاس‌های خود را به شکل نامناسبی تعریف می‌کند. در واقع کلاس‌هایی با داده‌های کم و رفتار زیاد و کلاس‌هایی با داده‌های زیاد و رفتار کم دیده می‌شود.
در این الگو با شرایطی مواجه هستیم که کلاس‌هایی که وظیفه‌ی رابط کاربری (User Interface) را برعهده دارند و نوعا کلاس‌هایی برای نمایش داده و اشیا هستند، منطق کد را نیز در خود قرار داده اند. با فرض معماری پرکاربرد سه ستونه، که شامل سه لایه یا ستون اصلی presentation, business logic و data access است، در این الگو بر خلاف حالت معمول با معماری دو ستونه مواجه هستیم. در واقع business logic با لایه‌ی presentation ادغام شده و وجوه کسب و کاری و رفتاری سیستم در یک لایه قرار گرفته است. هدف این الگو، جداسازی بخش‌های presentation و business logic به منظور بهبود کیفیت کد، کاهش وابستگی و همچنین امکان ایجاد تغییرات مطلوب است.

شرایط اعمال الگو

این الگو را در شرایطی اعمال می‌کنیم که در معماری کد، بخش‌های presentation و business logic با یکدیگر ادغام شده و اشیا presentation که بایستی صرفا وظیفه‌ی واسط کاربری را داشته باشند، وظیفه‌ی اشیا domain را نیز بر عهده می‌گیرند. در واقع پیاده‌سازی‌های business logic نیز در presentation و اشیا view انجام شده است.
باید توجه کرد که در معماری سه ستونه، با تعریف یک واسط میان business و presentation، امکان تعریف viewهای مختلف و واسط‌های کاربری متنوع برای مجموعه‌ی domain objectها به سادگی فراهم است چرا که منطق کد از واسط کاربری منتزع شده و با تعریف واسط‌های کاربری، نیازی به پیاده‌سازی منطق business در آن‌ها نیست. به همین دلیل انعطاف و گسترش‌پذیری بالایی در این معماری برای تعریف viewهای متنوع دیده می‌شود.
اما در شرایط پیش از اعمال این الگو، به علت در هم تنیدگی کدهای business و presentation، امکان تعریف viewهای متعدد دیده نمی‌شود. Coupling بالا است و همچنین حجم کدهای duplicate بالا است. چرا که بخش‌های businessی در ذات خود یکسانند و رفتار مشابهی از آن‌ها انتظار داریم اما بایستی این کدها را به ازای

viewهای مختلف پیاده‌سازی کنیم. در نتیجه هر شی presentation ممکن است کدهای duplicate داشته باشد و این duplication به علت یکسان بودن business logic در سرتاسر presentation دیده می‌شود.

چگونگی مفید بودن الگو در موقعیت

با توجه به مطالبی که تاکنون گفتیم، به علت وجود duplication بالا، یک وابستگی نهانی میان بخش‌های کد وجود دارد. در واقع برای تغییر و گسترش business logic، لازم داریم تا این تغییرات را در اشیا مختلف presentation که آن business logic را پیاده‌سازی و از یکدیگر کپی کرده اند نیز اعمال کنیم. در نتیجه‌ی آن، گسترش و تغییرات منجر به تغییرات متعدد در سطح presentation می‌شود.
علاوه بر business logic، گسترش viewها نیز می‌تواند پیچیده باشد. چرا که برای تعریف viewهای جدید بایستی تمامی business logicهای مربوط به آن را نیز پیاده‌سازی کنیم. تغییر در viewها لازم به دقت در عدم تغییر در business logic است. در واقع به دلیل non cohesive بودن کلاس‌های presentation، ایجاد تغییر بسیار سخت می‌شود.
پس از اعمال این الگو، بخش‌های business logic و domain objects از بخش presentation جدا می‌شود. در نتیجه‌ی آن، اشیا presentation از business logic سرویس می‌گیرند و به همین دلیل، اشیا view بسیار سبک‌تر از قبل می‌شوند و تنها وجه نمایشی و واسط کاربری را برعهده می‌گیرند. به همین سبب، انعطاف‌پذیری و گسترش‌پذیری کد بسیار بالا می‌رود.
پس از اعمال الگو، با تغییر business logic، دیگر تغییرات به اشیا متعدد view که در presentation هستند منتشر نمی‌شوند. در واقع جزییات بیزینسی از دید presentation و واسط کاربری منتزع می‌شود. تغییرات بیزینسی دیگر منتشر نمی‌شوند مگر آنکه واسط business تغییر کند. چرا که دید از presentation به business یک دید مبتنی بر interface است و در سطح concrete دید وجود نخواهد داشت. احتمالا تعدادی facade نیز میان این دو لایه‌ی کد دیده می‌شود. تغییرات در viewها نیز به دیگر بخش‌ها منتشر نمی شوند و همچنین گسترش viewها نیز دیگر سخت نخواهد بود. چرا که viewها برای منطق خود از لایه‌ی business logic سرویس می‌گیرند.

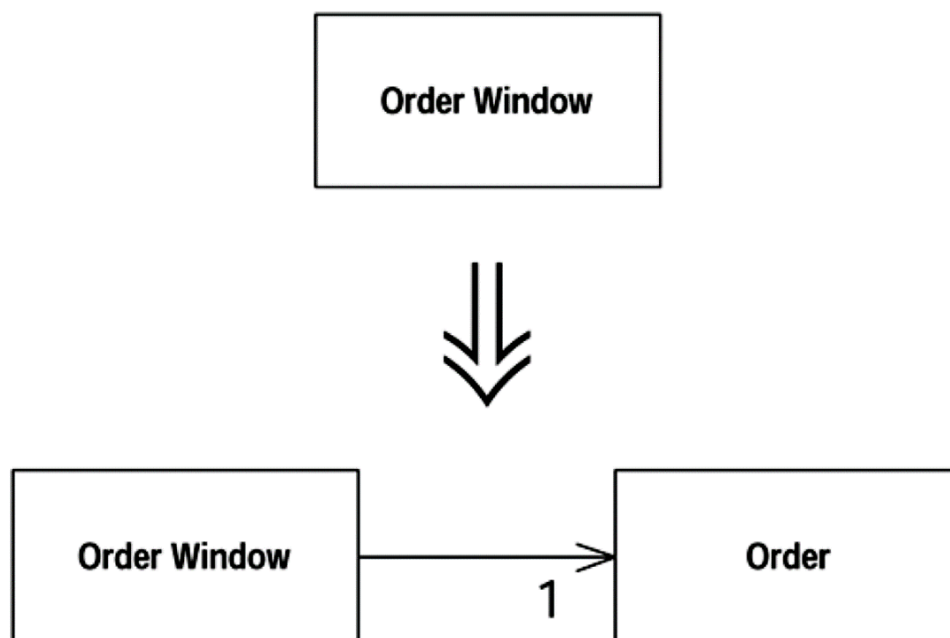
چگونگی پیاده‌سازی

روش توصیف شده در کتاب چاپ اول فاولر چندان ملموس نیست. اساسا اجرای این الگو چندان کار راحتی نیست. چرا که کدهای presentation و business در هم تنیده شده اند و بعضا منجر به ایجاد god classهایی شده اند که جداسازی آن نیاز به درک بهتر از دامنه‌ی بیزینسی دارد. به همین دلیل یک روش کلی بر اساس آن ارائه می‌کنیم:

- باید duplication را در سطح کلاس‌های presentation تشخیص دهیم. بر اساس آن می‌توانیم فهم اولیه نسبت به کلاس‌های دامنه بیزینس پیدا کنیم.

- کلاس‌های دامنه را باید استخراج کنیم. این کار با توجه به کدهای duplicate و همچنین توجه به دامنه‌ی کسب و کار، بایستی انجام شود. این مرحله بسیار سخت و پیچیده است و مهم‌ترین قسمت اجرای این الگو است.
- به کلاس‌های view نگاه می‌کنیم. اگر داده‌ای که مورد استفاده قرار گرفته است، تنها برای واسط کاربری استفاده شده آن را نگه می‌داریم و رفتار آن نیز نگهداری می‌شود. در غیر این صورت با اجرای extract method و move method آن را به کلاس‌های دامنه‌ای مربوط به خود نزدیک می‌کنیم.
- در صورتی که با انتهای این مراحل همچنان بخش‌هایی از business در presentation قرار دارند، با انجام عملیات‌های ریزدانه‌ی refactoring لازم است تا داده‌ها و رفتارهای بیزینسی را از presentation جدا کرده و به business logic منتقل کنیم.

مثالی که در اینجا استفاده شده است را در زیر می‌بینیم:



مزایا

- کد منعطف و گسترش پذیر می‌شود.
- امکان تعریف viewهای مختلف برای یک logic فراهم می‌شود.
- Cohesion و coupling افزایش می‌یابد.
- Duplication کم می‌شود.
- سرعت افزودن ویژگی‌های و UIهای جدید به سیستم افزایش می‌یابد.
- Portability افزایش می‌یابد.

معایب

- به دلیل جداسازی business logic از presentation، چند سطح indirection ممکن است اضافه شود که به کاهش performance منجر شود.
- در ابتدا ممکن است به نظر برسد که سرعت ایجاد کاهش یافته است. چرا که در حالت پیشین، طراحی ساده‌تر بوده و همه‌ی کارها در مجموعه‌ی محدودی از کلاس صورت می‌گرفت. اساسا وضعیت پیشین این الگو به علت تنبلی ایجاد کنندگان نرم افزار است که در ابتدای کار، سرعت ایجاد نرم افزار با حذف یک لایه از معماری افزایش می‌یابد. اما در بلند مدت و با بزرگ شدن سیستم، با کندی شدید مواجه می‌شویم.
- این الگو از الگوهای بازآرایی بزرگ است و در نتیجه اعمال آن می‌تواند زمان بسیاری از تیم ایجاد بگیرد و اساسا اعمال آن کار راحتی نیست.

الگوهای مرتبط

- از الگوهای extract method و move method در حین اعمال این الگو بهره برده می‌شود.
- از الگوی extract interface نیز برای تعریف interfaceهای لایه‌ی business logic ممکن است بهره برده شود.
- با الگوهای tease apart inheritance و convert procedural design to objects از جهت نوع الگوی بازآرایی در دسته‌ی big refactoring هستند.

الگوی دوم: Extract Interface

بررسی الگو

این الگو از مجموعه الگوهای dealing with inheritance است. البته این الگو در کتاب چاپ دوم نیامده و در کتاب چاپ اول به آن اشاره شده است. مجموعه الگوهای این دسته به یکی از مهم‌ترین ویژگی‌های شی گرایی یعنی توارث می‌پردازند. مفهوم توارث ساده و بسیار کاربردی است اما به راحتی امکان استفاده اشتباه از آن وجود دارد. بسیاری اوقات از توارث برای reuse استفاده می‌شود که درست نیست و اصل مهم در رابطه‌ی توارث، وجود رابطه‌ی is a یا gen/spec میان دو کلاس است.
این الگو به شرایط و چگونگی تعریف interface روی یک کلاس و یا تعدادی کلاس با سرویس‌های شبیه به یکدیگر می‌پردازد. تعریف interface در uml با برخی زبان‌ها مانند java یکسان نیست و تفاوت کمی دارند. در UML، می‌توان گفت که interface هم شامل attribute است هم operation اما همه چیز specification است و implementation وجود ندارد. در صورت وجود implementation یکسان، بایستی به جای interface، یک abstract class تعریف شود.
در فاز طراحی شی گرا توصیه می‌شود به طور افراطی interface تعریف کنیم. با دیدن مشترکات و جایی که حتی ساختار توارثی وجود ندارد نیز interface ایجاد می‌کنیم برای اینکه دید را محدود کنیم.
یکی از مهم‌ترین ثمرات این الگو، برقراری dip میان کلاینت‌ها و کلاس‌های concrete است. چرا که وابستگی میان کلاینت و سرورهای concrete با تعریف interface کمتر می‌شود.

شرایط اعمال الگو

این الگو در دو موقعیت متفاوت به کار می‌رود.
موقعیت اول آنکه یک کلاس داریم که کلاینت‌های مختلف، از یک زیرمجموعه‌ای از سرویس‌ها و operationهای آن استفاده می‌کنند. برای این زیرمجموعه، یک interface تعریف می‌کنیم. در نتیجه‌ی این کاربرد، دید کلاینت به کلاس اصلی را محدود می‌کنیم. در واقع کارکرد اصلی این الگو برای این موقعیت این است که encapsulation را به خوبی برای کلاس سرور برقرار کنیم تا دید کلاینت به تمامی داده‌ها و سرویس‌های آن برقرار نباشد.
در موقعیت دیگر، مجموعه‌ای از کلاس‌های مختلف داریم که بخشی از واسط‌های این کلاس‌ها مشترک است. برای این بخش، interface تعریف می‌کنیم. در نتیجه‌ی آن، یک ساختار interface به وجود می‌آید که یک configurer بایستی کلاینت‌ها را با کلاس concrete مناسب interface تغذیه کند.
در نتیجه‌ی هر دو شرایط، وابستگی و coupling میان کلاینت و سرورها کاهش می‌یابد که نتیجه‌ی برقراری dip است. به همین ترتیب امکان گسترش و تعریف سرورهای جدید بدون تغییر در کلاینت به وجود می‌آید.

چگونگی مفید بودن الگو در موقعیت

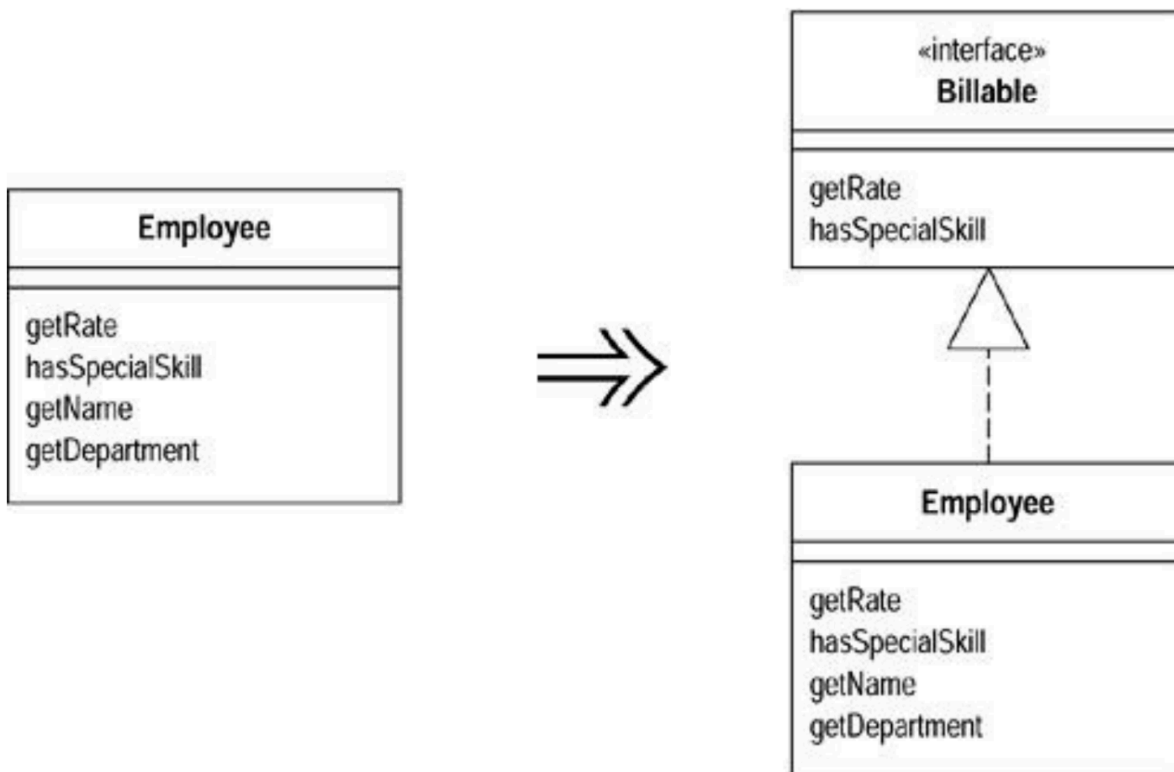
در هر دو شرط گفته شده برای اعمال الگو، dip برقرار شده و دید کلاینت به سرور یا سرورها محدودتر و از طریق واسط می‌شود. در نتیجه، تغییرات سرورها به کلاینت منتقل نمی‌شود. هرچند باید توجه کرد که تغییرات در interface به دلیل استفاده‌ی کلاینت‌ها و دید آن‌ها، منجر به تغییر در کلاینت می‌شود.
اما به علت برقراری dip، هر گونه تغییر در سرورها به کلاینت منتشر نمی‌شود. همچنین استفاده از این الگو امکان گسترش را به ما می‌دهد. در واقع می‌توانیم سرورهای مختلفی که interface تعریف شده را implement می‌کنند تعریف کنیم و کلاینت‌ها بایستی توسط یک پیکربند، پیکربندی شوند. در نتیجه با گسترش ساختار interface و implementationهای آن و همچنین تغییر در کلاس‌های concrete، هیچ‌گونه تغییری در کلاینت‌ها را منجر نمی‌شود.
برای مثال فرض کنید یک کلاس A داریم که از بخشی از سرویس‌های ارائه شده در کلاس B، بایستی استفاده کند. حال می‌خواهیم سرویس‌های ارائه شده توسط B را به شکل دیگری و در قالب کلاس C ارائه کنیم. در این صورت، اگر هیچگونه interfaceای روی دو کلاس B و C تعریف نکنیم، کلاس A لازم است تا هر دو کلاس را بشناسد و شرایط تغییر وضعیت و استفاده از هر کدام از کلاس‌ها را بایستی خود دانسته و پیاده‌سازی کند. پس با گسترش ساختار و تعریف C، کلاینت نیز تغییر می‌کند. حال اگر کلاینت‌های متعددی برای این کلاس داشته باشیم، گسترش ساختار بسیار سخت می‌شود چرا که باید تمامی کلاینت‌ها تغییر کنند. اما اگر یک interface روی این کلاس‌ها تعریف کنیم، کافیست تا configurer بسته به شرایط، A را پیکربندی کند. اینگونه تغییرات به A و دیگر کلاس‌ها منتشر نمی‌شود.

چگونگی پیاده‌سازی

بر اساس کتاب چاپ اول فاولر، مراحل زیر می‌تواند در تعریف interface مفید باشد:

- یک interface خالی تعریف می‌کنیم.
- همه‌ی operationهایی که بین کلاس‌های کلاینت (در موقعیت اول) و یا کلاس‌های سرور (موقعیت دوم) مشترک است را در interface تعریف می‌کنیم.
- کلاسی(هایی) که این interface را پیاده‌سازی می‌کنند را تغییر می‌دهیم و آن‌ها را ذیل interface تعریف می‌کنیم.
- در کلاینت‌ها نیز به جای کلاس concrete، استفاده از interface را جایگزین می‌کنیم و type مورد استفاده‌ی آن‌ها را تغییر می‌دهیم.

نمونه‌ای که در کتاب فاولر آمده نیز به شرح زیر است:
در این نمونه، می‌خواهیم حقوق یک کارمند را محاسبه کنیم. به این منظور، بسته به رتبه و همچنین مهارت کارمند، حقوق محاسبه می‌شود. برای محاسبه، یک interface روی employee تعریف می‌کنیم تا محاسبه‌گر تنها به دو سرویس getRate و hasSpecialSkill دید داشته باشد.



مزایا

- برقراری encapsulation و محدود کردن دید کلاینت‌ها به سرور
- کاهش coupling و برقراری dip با برقراری دید در سطح انتزاع بالا
- جلوگیری از انتشار تغییرات در سرور به کلاینت
- افزایش گسترش‌پذیری و انعطاف‌پذیری

معایب

- نیاز به یک پیکربند برای پیکربندی کلاینت‌ها با سرور مناسب
- وابستگی بالای پیکربند به کلاس‌های concrete
- Interfaceها بایستی کمتر تغییر کنند. تغییرات زیاد در interface تعریف شده نشان دهنده‌ی نابالغ بودن طراحی است و خود منجر به انتشار تغییرات می‌شود و هدف استفاده از این الگو در موقعیت گفته شده (انتشار تغییرات) را نقض می‌کند.

الگوهای مرتبط

- این الگو با الگوی extract superclass در نزدیکی جدی قرار دارد. البته شرایط اعمال دو الگو متفاوت است چرا که superclass می‌تواند برای reuse به کار رود اما interface هدف تحدید دید کلاینت به سرور را دارد.

مقایسه‌ی دو الگو

شباهت

الگوی اول، می‌تواند از الگوی extract interface بهره ببرد تا بین لایه‌ی business logic و لایه‌ی presentation یک جداسازی صورت گیرد.
هر دو الگو به بهبود انعطاف‌پذیری و گسترش‌پذیری می‌انجامند و از انتشار تغییرات جلوگیری می‌کنند.

تفاوت

الگوی اول، از دسته‌ی الگوهای big refactoring است و معمولا زمان زیادی برای اجرای آن مورد نیاز است. اما الگوی دوم از الگوهای نسبتا ریزدانه در دسته‌ی dealing with inheritance است.
الگوی اول بیشتر وجهه‌ی معماری دارد. در واقع یک معماری نادرست دو ستونه منجر به کاهش انعطاف و گسترش‌ناپذیری سیستم شده و ایجاد تغییر در سیستم سخت شده است. در الگوی دوم، تمرکز بر تعریف یک انتزاع بالاتر برای انعطاف‌پذیری و گسترش پذیری است.
در الگوی اول، کلاس‌ها و پیاده‌سازی‌های جدید ممکن است به وجود آید و جابجایی کلاس‌ها و متدها امکان‌پذیر است. اما در الگوی دوم، کلاس یا متد جدیدی تعریف نمی‌شود و تنها یک specification تعریف می‌شود.