تمرین سوم درس الگوها در مهندسی نرم افزار

مقایسه نتایج اعمال الگوهای متفاوت بازآرایی کد در شرایط مشابه

استاد: دکتر رامان رامسین

وحید رحیمیان

شماره دانشجویی: ۹۶۳۰۱۶۳۲

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

تابستان ۱۳۹۷

فهرست مطالب

۱. موقعیت اول

اگرچه ارث‌بری از امکانات فوق العاده زبان‌های شئ‌گرا است، در برخی از شرایط به آن احتیاج نداریم. در واقع ممکن است در سیر تحول کد از یک کلاس ارث‌بری کنیم اما طی زمان متوجه شویم که بسیاری از رفتارهای کلاس پدر برای کلاس فرزند بی‌معنا هستند؛ یعنی داده‌ها یا رفتارهایی به ارث رسیده که برای کلاس فرزند فاقد معنا هستند یا به آنها هیچ‌گونه احتیاجی ندارد. در این حالت interface کلاس پدر نمایش دهنده رفتار واقعی کلاس فرزند نیست.

اگر چه در این شرایط همچنان میتوان رابطه ارث‌بری را حفظ کرد، در این حالت همیشه باید در نظر داشت که علیرغم رابطه وراثت تنها از رفتار کلاس پدر به کلاس فرزند به ارث رسیده است است که ای کار به معنای وجود یک ابهام در معنای کد (برای دیگران یا حتی خود نویسنده کد پس از گذشت زمان است).

در این شرایط اصل LSP[1] نقض شده است، زیرا از ارث بری جهت code reuse استفاده شده است نه به دلیل اینکه کلاس فرزنده توسعه دهنده (extension) رفتارهای کلاس پدر است.

در این بخش به شرایطی که در آن کلاس‌های فرزند به مسئولیت‌هایی که کلاس پدر بر عهده آنها قرار داده است پایبند نیستند (میراث مردود[2]) خواهیم پرداخت. برای حل مشکل دو الگوی زیر را بررسی خواهیم کرد:

- جایگزین کردن وراثت (Inheritance) با واسپاری (Delegation)
- استخراج کردن Interface

هر دوی این الگوها برای مدیریت ارث‌بری[3] هستند

۱,۱ بررسی الگوی اول: جایگزین کردن وراثت با واسپاری

۱,۱,۱ مقدمه ای بر الگو

این الگو دقیقا برای حل مساله میراث مردود به وجود آمده است، زمانی که ما از یک کلاس ارث‌بری کرده ایم، اما تنها بخشی از رفتارهای کلاس پدر را استفاده می‌کنیم یا تمایلی به ارث بری داده‌های کلاس پدر نداریم. در واقع بزرگ‌ترین اشتباه در سیستمهای شیء‌گرا این است که به منظور استفاده مجدد از کد[4] از ارث بری استفاده

[1] Liskov Substitution Principle

[2] Refuesed Bequest

[3] Dealing with Generalization

[4] Code Reuse

کنند؛ در بسیاری از اوقات استفاده از ارث‌بری جهت استفاده مجدد از کد در دراز مدت موجب بروز مشکل میراث مردود می‌گردد.

در این راه حل استفاده از واسپاری (delegation) نشان دهنده آن است که تنها بخشی از رفتارهای یک کلاس دیگر (که قبلا کلاس پدر بود) مورد استفاده است. در نتیجه کنترل اینکه از چه جنبه هایی از کلاس استفاده می‌شود (دید یا interface) و از چه مواردی صرف نظر شده است در دست کلاس استفاده کننده خواهد بود.

هزینه این فرآیند نوشتن متدهای واسپاری است (سختی اصلی اعمال این الگو نوشتن متدهای زیادی برای انجام delegation های ساده است) که به فعالیت برنامه‌نویسی اضافه می‌شود؛ اگرچه به دلیل سادگی زیاد امکان بروز خطا در آن بسیار کم است.

۱,۱,۲ چگونگی اعمال الگو

یک رابطه Association (به صورت یک فیلد) به کلاس پدر ایجاد می‌کنیم و متدهای کلاس را به گونه ای تغییر میدهیم که رفتار مرتبط را به فیلد از نوع کلاس پدر واسپاری کنند. سپس رابطه ارث‌بری را از بین می‌بریم. در این حالت با از بین رفتن ارث بری به طور کلی میراث[1] وجود نخواهد داشت که ملزم به پذیرش آن باشیم.

۱,۱,۳ مزایا

- با اعمال الگو کلاس هیچ متد اضافی را به ارث نمی‌برد؛ طی زمان متدهای اضافی به ارث رسیده توسط کلاینت‌ها یا متدهای خود کلاس به اشتباه فراخوانی خواهد شد و با اعمال این الگوی refactoring جلوی این اتفاق گرفته می شود.
- در عین حال با استفاده از واسپاری (delegation) امکان قرار دادن اشیاء مختلف در فیلد delegate (و حتی تغییر آن در زمان اجرا) وجود خواهد داشت. زیرا رابطه ارث‌بری صلب است و در زمان کامپایل تثبیت می‌شود اما رابطه delegation در زمان اجرا قابل تغییر است (انعطاف پذیری بسیار بالا).
- پس از اعمال الگو، در صورت نیاز به وجود رفتارهای مختلف بر اساس شرایط، استفاده از الگوی طراحی Strategy امکان پذیر می‌گردد.
- در نتیجه اعمال این الگو تغییرات واگرا کمتر می‌شود، زیرا با واسپاری رفتارها، cohesion بالا می‌رود و هر کلاس تنها رفتار تخصصی لازم را دارد (رفتارهای اضافی و زائد به ارث رسیده نداریم).

[1] Bequest

همچنین بالاترین نوع coupling در شیء گرایی رابطه Gen/Spec است که در این الگو آن را با رابطه Delegation که انعطاف پذیری زمان اجرا دارد جایگزین کرده‌ایم.

- همچنین با اعمال این الگو Shotgun Surgery کمتر می‌شود زیرا coupling کمتر شده است. پیش از اعمال الگو اضافه کردن متدهای جدید به کلاس پدر نیازمند تغییر برخی کلاس‌های فرزند جهت بی اعتبار کردن رفتار اضافی به ارث رسیده بود که یک تغییر منتشر شوند بود.

۱,۱,۴ معایب

- سختی اضافه کردن متدهای ساده delegate که هر چند نوشتن آنها ساده است و لذا احتمال بروز خطا در آنها کم است، اما ممکن است حوصله برنامه نویس را سر ببرد
- کاهش کارایی (هر چند جزیی) در نتیجه اضافه شدن یک سطح از indirection در واسپاری عملیات
- امکان ایجاد Transitive Visibility (Message Chain) به صورت بالقوه، در صورتی که پس از اعمال الگو واسپاری ایجاد شده از کلاینت‌های کلاس پنهان نباشد (فید تعریف شده جهت واسپاری public باشد)
- پتانسبل بروز Feature Envy اگر واسپاری به درستی صورت نگیرد: احتمال واسپاری برخی رفتارهای ذاتی کلاس به کلاس delegate وجود دارد. در این صورت فراخوانی چنین متدهایی احتمالا لیست پارامترهای طولانی هم می‌خواهد (چون دیگر دسترسی به فیلدهای کلاس اصلی وجود ندارد)
- پتانسیل بروز Inappropriate Intimacy، اگر مرز واسپاری (تفکیک کلاس‌ها از هم) به درستی صورت نگیرد

۱,۱,۵ گام های اعمال الگو

۱. قرار دادن یک فیلد در subclass از که به instance ی از superclass اشاره کند و مقدار دهی اولیه این فیلد با this

۲. تغییر هر متد در subclass به صورتی که به جای استفاده از متدهای this که از superclass به ارث رسیده اند از فراخوانی متد روی فیلد از نوع superclass استفاده کند. پس از تغییر هر متد باید کامپایل و تست شود.

توجه: اگر متدی به صورت abstract در کلاس پدر تعریف شده باشد و پیاده سازی آن در کلاس فرزند انجام شده باشد، یا رفتار کلاس پدر در کلاس فرزند override شده باشد، ممکن است با انجام گام ۲ فراخوانی recursive به صورت نامحدود پیش آید. در این حالت باید پس از شکستن رابطه ارث برای این متدها کد جایگزین نوشت.

۳. حذف رابطه ارث‌بری و جایگزینی مقداردهی delegate با یک شیء جدید (new object) از کلاس پدر
٤. اضافه کردن متدهای ساده delegate کننده برای هر یک از متدهای کلاس پدر که توسط کلاینت‌های کلاس مورد استفاده قرار می‌گیرند.

در ادامه این گام ها را در یک مثال خواهیم دید.

فرض کنید یک کلاس به صورت زیر دارید:

```
class MyStack extends Vector {
    public void push(Object element) {
        insertElementAt(element, 0);
    }
    public Object pop() {
        Object result = firstElement();
        removeElementAt(0);
        return result;
    }
}
```

با بررسی کلاس های استفاده کننده از کلاس MyStack (کلاینت‌های کلاس) مشخص می‌شود که چهار متد از کلاس MyStack مورد استفاده قرار می‌گیرد: push، pop، size، و isEmpty. دو متد آخر از کلاس پدر یعنی Vector به ارث رسیده اند و بدون تغییر در اختیار کلاینت‌ها قرار داده شده اند.

برای اعمال الگو در گام اول یک فیلد از نوع Vector تعریف میکنیم و مقدار this را به آن می‌دهیم. در واقع در میانه فرآیند refactoring هم ارث‌بری و هم واسپاری با هم وجود دارند:

```
private Vector _vector = this;
```

در گام دوم متدهایی از کلاس که از رفتارهای کلاس پدر استفاده میکنند را تغییر می دهیم تا از واسپاری استفاده کنند. پس از تغییر هر متد کد را کامپایل میکنیم و همچنین تست های برنامه را اجرا میکنیم:

```
public void push(Object element) {
    _vector.insertElementAt(element,0);
}

public Object pop() {
    Object result = _vector.firstElement();
    _vector.removeElementAt(0);
    return result;
```

```
}
```

در گام سوم رابطه ارث برای را از بین می‌بریم:

```
class MyStack extends Vector
private Vector _vector = new Vector();
```

در گام چهارم هم متدهای ساده ای را اضافه میکنیم تا رفتارهای superclass را در اختیار کلاینت‌های کلاس قرار دهند:

```
public int size() {
    return _vector.size();
}

public boolean isEmpty() {
    return _vector.isEmpty();
}
```

با انجام کامپایل و تست کد، خواهیم فهمید که آیا نوشتن یک متد delegate فراموش شده است یا کل فرآیند با موفقیت به اتمام رسیده است.

۱,۲ بررسی الگوی دوم: استخراج واسط

۱,۲,۱ مقدمه ای بر الگو

الگوی Extract Interface زمانی استفاده می‌شود که چندین کلاینت از یک زیر مجموعه از interface یک کلاس استفاده می‌کنند، یا اینکه دو کلاس بخشی از interface شان مشترک است. در این حالت زیر مجموعه را به عنوان یک واسط (interface) مستقل بیرون می‌کشیم.

در واقع استفاده کلاس‌ها از یکدیگر از چندین طریق ممکن است:

۱. استفاده از تمام قابلیت‌ها (رفتارها) یک کلاس دیگر
۲. استفاده گروهی از کلاینت ها از مجموعه مشخصی از قابلیت‌های کلاس
۳. نیاز یک کلاس برای کار کردن با هر کلاس سرویس‌دهنده‌ای که توانایی پردازش درخواست‌های مشخصی را دارد.

در حالت اول در بسیاری از شرایط از رابطه ارث‌بری استفاده می‌کنیم. در حالت دوم و سوم، بهتر است که مجموعه وظیفه مندی ها (رفتارها یا responsibility ها) یک هویت مستقل پیدا کند، تا مسئولیت‌های کلاس‌ها در سطح سیستم بهتر مدیریت شود.

برخی زبان‌های شیء گرا مانند C++ این حالت‌ها (۲ و ۳) را با قابلیت ارث‌بری چندگانه[1] پشتیبانی می‌کنند. در این زبان‌ها برای هر یک از بخش‌های رفتار یک کلاس ایجاد می‌کنند و با ترکیب ارث‌بری از آنها، مجموعه رفتار مورد نظر را پیاده‌سازی می‌کنند. اما با توجه به مشکلات ارث‌بری چندگانه، زبان‌های شیء گرای مدرن مانند Java و C# این نیازمندی را با استفاده از تعریف interface پشتیبانی می‌کنند.

در واقع با ارث‌بری کلاس فرزند از کلاس پدر، رفتار اضافی به ارث می‌رسد که مورد نیاز نیست. در این حالت یک روش شکستن کلاس پدر به چندین کلاس پدر کوچک تر بوده است که روش انجام آن در زبان‌های شیء گرای مدرن استفاده از interface برای تعریف مجموعه مشخصی از رفتارهاست.

الگوی Extract Interface با الگوی Extract Superclass شباهت دارد، اما در Extract Interface تنها واسط مشترک بیرون کشیده میشود نه کد مشترک. در نتیجه Extract Interface میتواند منجر به duplicated code شود که یک bad smell است. البته این رخداد را میتوان با استفاده از الگوی Extract Class جهت قراردادن رفتار در یک کلاس و واسپاری عملیات به آن کاهش داد. باید توجه داشت که اگر رفتار مشترک زیادی بین کلاس ها وجود اداشته باشد، اعمال الگوی Extract Superclass ساده تر است، اما در زبان‌هایی مانند Java هر کلاس تنها می‌تواند یک کلاس پدر داشته باشد.

از سوی دیگر interface ها در زمان‌هایی که یک کلاس در شرایط مختلف نقش‌های متفاوتی دارد (چیزی مشابه الگوی roles played از الگوهای هفت گانه Coad) بسیار مفید است. در این زمان‌ها می‌توان از الگوی Extract Interface برای هر نقش استفاده کرد.

همچنین در مواقعی که شما میخواهید واسط خارجی (outbound interface) یک کلاس، یعنی عملیاتی که آن کلاس از سرویس دهنده خود انتظار دارد، را مشخص کنید، استفاده از این الگو کارگشاست. با استفاده از interface می‌توان در آینده از سرویس دهندگان دیگر استفاده کرد (حتی در زمان اجرا سرویس دهنده را تغییر داد یا آن را از بیرون به کلاس تزریق[2] کرد). تنها کاری که این کلاس‌های سرویس‌دهنده باید انجام دهند پیاده‌سازی interface مشخص شده است.

[1] Multiple inheritance

[2] inject

۱,۲,۲ چگونگی اعمال الگو

مجموعه رفتار مورد نظر را در یک interface تعریف می‌کنیم و به آن رجوع می‌کنیم.

۱,۲,۳ مزایا

- کم شدن تغییرات واگرا به دلیل ارتباط DIP بین کلاس‌ها. دید به interface (و نه concrete class) موجب به وجود آمدن DIP می‌شود که از انتشار تغییرات جلوگیری می‌کند.
- بهبود تغییرپذیری سیستم (OCP) به دلیل کاهش وابستگی (coupling) میان کلاس‌ها در نتیجه رابطه DIP. در واقع به دلیل محدود شدن دید به interface تغییرات منتشر شونده کاهش می‌یابد.
- استفاده از Interface میتواند مشابه چندریختی[1] در رابطه وراثت، جلوی به وجود آمدن switch statement ها را بگیرد. زیرا در زبان‌های مدرن که ارث‌بری چندگانه نداریم اگر از interface هم استفاده نکنیم مجبور خواهیم بود در صورت استفاده از دو نوع[2] مختلف از پارامتر، یک نوع بالاتر (مثلا Object) برای پارامتر ارسال شده به متد تعریف کنیم و سپس با استفاده از switch بر اساس Type Check رفتار مورد نظر را از پارامتر دریافت شده بخواهیم.
- سرویس های مشترک بین چند کلاس در یک واسط پیاده شده و کلاینت فقط با واسط ارتباط برقرار می کند؛ علاوه بر این هر کلاسی به میزان دانش مورد نیازش دید خواهد داشت.

۱,۲,۴ معایب

- پتانسیل ایجاد کد تکرار شونده، زیرا در interface بر خلاف superclass تنها تعریف متد قرار دارد و هیچ کدی در آن قرار نمی‌گیرد. لذا احتمال تکرار کدهای پیاده‌سازی متدهای interface در کلاس‌های مختلف پیاده‌سازی کنند آن وجود دارد (که البته با واسپاری میتوان این رفتارهای مشترک را در یک مکان واحد پیاده‌سازی کرد)
- اضافه شدن تعداد interface ها ممکن است موجب افزایش انواع داده ای و در نتیجه افزایش پیچیدگی ورود به کد برنامه (به ویژه در صورت ورود افراد جدید به تیم) شود.
- پتانسیل بروز Alternative classes with different interface وجود دارد (کلاس‌هایی که رفتار یکسانی دارند با واسط‌های مختلف). زیرا اگر از واسط‌های از قبل تعریف شده استفاده نکنیم عملا ممکن است duplication در تعریف interface ها به وجود آید. لذا interface ها هم باید روابط پدر–فرزندی مناسب را داشته باشند.

[1] polymorphism

[2] Type

- پتانسیل به وجود آمدن Data Class وجود دارد، چون interface ها صرفا رفتار هستند (در زبان‌هایی مانند Java و C#) و شامل اقلام داده‌ای نمیباشند، لذا ممکن است در کنارشان Data Class به وجود آید.

۱,۲,۵ گام های اعمال الگو

۱. ساخت یک interface خالی
۲. تعریف رفتارهای مشترک درون interface
۳. تعریف کلاس های مرتبط به عنوان پیاده سازی کنندگان Interface
٤. تصحیح تعاریف نوع کلاینت[1] جهت استفاده از interface

در ادامه این گام ها را در یک مثال خواهیم دید.

فرض کنید یک کلاس تایم‌شیت داریم که حقوق کارمندان را محاسبه می‌کند. برای انجام این کار این کلاس باید رتبه کارمند و همچنین دارا بودن مهارت‌های ویژه وی را بداند:

```
double charge(Employee emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

کلاس Employee ویژگی‌ها و رفتارهای زیادی به غیر از رتبه و دارا بودن مهارت‌های ویژه دارد که در اینجا به آنها احتیاجی نداریم. تاکید بر این نکته که تنها به یک زیرمجموعه مشخص از رفتارهای Employee نیاز داریم را با تعریف یک Interface برای این رفتارها انجام می‌دهیم:

```
interface Billable {
    public int getRate();
    public boolean hasSpecialSkill();
}
```

سپس در گام سوم کلاس کارمند را پیاده سازی کننده این interface قرار می‌دهیم:

```
class Employee implements Billable ...
```

با تعریف interface می‌توانیم رفتار محاسبه حقوق را طوری تغییر دهیم که نشان دهیم تنها از این بخش از رفتار کلاس کارمند استفاده شده است:

[1] client type declaration

```
double charge(Billable emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

در این مثال منفعت اعمال الگو متوسط و موجب خواناتر شدن کد است. این منعفت برای یک متد چندان بزرگ نیست، اما در صورتی که چندین کلاس بخواهند از Billable interface بودن کلاس Employee استفاده کنند بسیار بزرگ است. بیشترین منعفت (و توجیه استفاده از الگو) زمانی است که بخواهیم برای Computer ها هم هزینه استفاده (مشابه حقوق) تعریف کنیم. برای Billable کردن کامپیوترها تنها کافی است که interface مزبور را پیاده سازی کنند، تا بتوانند در کلاس تایم‌شیت مورد استفاده قرار گیرند.

۱,۳ مقایسه دو الگو

۱,۳,۱ مقایسه زمان استفاده از دو الگو (when) در شرایط مساله

در شرایط میراث مردود، اگر رابطه ارث‌بری به صورت غیر واقعی (بدون وجود رابطه معنایی gen/spec) صرفا جهت code reuse استفاده شده باشد، الگوی Replace Inheritance with Delegation با حذف وراثت و جایگزین کردن واسپاری مورد استفاده قرار می‌گیرد. این الگو به ویژه در زمانی که به اشتباه در فرآیند refactoring به جای اعمال Extract Class از الگوی Extract Superclass استفاده کرده باشیم کاربرد دارد.

در شرایط میراث مردود، اگر رابطه ارث‌بری جهت ارائه مجموعه رفتارهایی از کلاس پدر به کلاینت‌ها باشند، می‌بایست این مجموعه از رفتارها به صورت interface تعریف شوند و با حذف رابطه وراثت، کل رفتار در کلاس فرزند پیاده‌سازی شود (و در صورت نیاز از واسپاری عملیات به کلاس‌های دیگر نیز استفاده شود). در اینجا مجموعه رفتارهای مد نظر کلاینت در کلاس پدر تعریف شده‌اند، اما ممکن است در کلاس فرزند override شده باشند یا حتی برخی رفتارهای مجموعه اصولا در کلاس پدر به صورت abstract تعریف شده باشند. لازم به ذکر است دید کلاینت ممکن است در شرایط دیگر به مجموعه مشخص دیگری از رفتارهای کلاس باشد.

شرایط مساله به این معناست که رابطه معنایی gen/spec بین کلاس فرزند و کلاس پدر وجود ندارد و لذا میراث مردود وجود دارد؛ استفاده از الگوی اول بیشتر زمانی است که جهت استفاده مجدد از کد رابطه

ارث‌بری به وجود آمده است اما استفاده از الگوی دوم بیشتر زمانی است که جهت استفاده مجدد از interface مشترک تعریف شده در کلاس پدر، رابطه ارث‌بری به وجود آمده است.

۱,۳,۲ مقایسه نحوه استفاده از دو الگو (how) در شرایط مساله

هر دو الگو رابطه ارث بری را از بین می‌رود. در الگوی اول به جای آن Delegation قرار می‌گیرد و code reuse با استفاده از واسپاری انجام می‌شود که مطلوب زبان‌های شیء گرا است.

در الگوی دوم به جای آن interface قرار می‌گیرد که کم کننده وابستگی بین کلاس ها و محدود کننده دید است اما ممکن است موجب بروز duplicated code شود. در اثر اعمال این الگو رابطه DIP برقرار می‌شود و کلاینت به جای ارتباط با کلاس concrete به interface وابسته می‌شود که موجب بهبود OCP می‌شود.

۲. موقعیت دوم

تغییرات یک کلاس ممکن است بر اثر تغییر در نیازمندی‌های وظیفه ای یا غیر وظیفه ای سیستم، رفع خطا، بهبود طراحی، اضافه کردن قابلیت های جدید، یا به دلیل سازگاری با تغییرات سایر کلاس‌های سیستم باشد. معمولا هر کلاس پس از پیاده سازی اولیه و گذشت مدت زمانی از عمر سیستم، باید به وضعیت پایداری برسد و تغییرات چندانی طی زمان نداشته باشد.

اما گاهی از اوقات کلاس‌هایی مشاهده می‌شوند که دائما، به دلایل مختلف نیاز به تغییر پیدا می‌کنند. در این بخش به شرایطی که در آن یک کلاس، به دلایل مختلف غیرمرتبط دائما نیازمند تغییر است خواهیم پرداخت. برای حل مشکل دو الگوی زیر را بررسی خواهیم کرد:

- شکستن God Class [۱]
- جداکردن ساختارهای وراثتی [۲]

الگوی Split Up God Class از الگوهای مهندسی مجدد، بازتعریف مسئولیت‌ها [۳] است. این الگو بیشتر در جهت مدیریت تغییرات یک کلاس بزرگ که نیازمندی‌های وظیفه ای مختلف را دارد و بخش قابل توجهی از تغییرات سیستم (یا زیر سیستم) عمدتا در این کلاس انجام می‌شود، استفاده می‌شود. الگوی Tease Apart Inheritance از الگوهای Big Refactoring می‌باشد که بیشتر برای جلوگیری از انتشار تغییرات بین ساختارهای توارثی استفاده می‌شود.

۲,۱ بررسی الگوی اول: شکستن God Class

۲,۱,۱ مقدمه ای بر الگو

این الگو زمانی استفاده میشود که یک کلاس وظایف زیاد و متنوعی را دارد (Cohesion پایین). معمولا این وظایف زیاد، رفتارهای زیادی را شامل می‌شود که داده‌های مرتبط با این رفتارها در کلاس‌های داده‌ای (Data Class ها) قرار می‌گیرد.

[1] Split Up God Class

[2] Tease apart inheritance

[3] Redistribute Responsibilities

برای حذف این نوع کلاس‌ها، به صورت افزایشی مسئولیت‌های کلاس را به کلاس‌های تعامل‌کننده با آن (که داده های لازم را در خود دارند) یا کلاس‌های جدیدی که از کلاس اصلی بیرون کشیده می‌شوند (یک مجموعه رفتارهای تخصصی[1] به همراه اقلام داده‌ای مرتبط) می‌سپاریم.

در انتهای این فرآیند از God Class هیچ چیزی جز یک Facade باقی نمی‌ماند که ممکن است به کلی زائد باشد (و می‌بایست Deprecate و سپس حذف شود). در عین حال در بیشتر اوقات وصل کردن مستقیم کلاینت‌ها به کلاس‌ها، باعث بالا رفتن Coupling خواهد شد و لذا توجیه دارد که Facade باقی بماند؛ البته به دلیل اصل ISP[2] لازم است که interface ها Cohesive باشند و لذا ممکن است تبدیل به چندین interface تخصصی گردد.

در واقع در شرایط ذکر شده در مساله (یک کلاس به دلایل مختلف و غیرمرتبط دائما نیازمند تغییر است) اگر عمده تغییرات لازم (یا منتشر شونده) بر روی یک کلاس واحد رخ می‌دهد، و بر اساس معیارهای زیر بتوانیم God Class شناسایی کنیم، باید آن را بشکنیم (Split Up کنیم). شناسایی God Class بر اساس روش‌های مختلفی امکان پذیر است:

- یک کلاس واحد بزرگ که از کلاس‌های زیادی به عنوان ساختارهای داده‌ای استفاده می‌کند.
- یک کلاس ریشه بزرگ، که نام آن شامل کلماتی مانند System، Subsystem، Manager، Driver، یا Controller است.
- کلاسی که تغییرات سیستم همیشه موجب تغییر در آن کلاس هم می‌شود.
- تغییر دادن کلاس بسیار دشوار است، زیرا نمی‌توان فهمید که چه بخش‌هایی از کلاس از تغییر متاثر می‌شود.
- استفاده مجدد از کلاس تقریبا غیرممکن است، زیرا دغدغه های زیادی را پوشش می‌دهد.
- یک کلاس در قلمرو دامنه مساله شامل عمده اقلام داده ای و رفتارهای (زیر)سیستم است.
- یک کلاس Cohesion پایینی دارد، و مجموعه‌های غیرمرتبط از رفتارها روی اشیاء مستقل جدا از هم عملیات انجام می‌دهند.
- زمان لازم برای کامپایل یک کلاس، حتی برای تغییرات کوچک، زیاد است.
- به دلیل مسئولیت‌های زیاد کلاس، تست آن دشوار است.
- کلاس حافظه بسیار زیادی مصرف می‌کند.

[1] Cohesive

[2] Interface Segregation Principle

- گفته می‌شود که یک کلاس *قلب* سیستم است.
- وقتی از مسئولیت‌های کلاس می‌پرسید، پاسخ‌های مختلف، طولانی، و مبهم دریافت می‌کنید.
- کلاس‌های God Class کابوس نگهداره[1]های سیستم هستند. پس از آنها بپرسید که چه کلاسهایی بسیار بزرگ هستند و نگهداری از آنها دشوار است. از آنها بپرسید چه کلاسی است که دوست ندارند روی آن کار کنند.

۲,۱,۲ چگونگی اعمال الگو

راه حل مقابله با God Class ها، انتقال تدریجی رفتار از کلاس، به کلاس‌های داده ای است: رفتارهایی که در حالت اولیه توسط God Class بر روی داده‌های آنها انجام می‌شود. همچنین تعدادی کلاس‌های جدید از God Class استخراج می‌شوند.

باید در نظر گرفت که در سیستم های مختلف ممکن است God Class ها ساختارهای داخلی متفاوتی داشته باشند و لذا برای تبدیل آنها تکنیک‌های متفاوتی مورد نیاز است. همچنین معمولا امکان حذف یکباره God Class وجود ندارد و لذا یک راه حل امن تبدیل God Class به یک Lightweight God Class، سپس یک Facade که عملیات را به کلاس‌های دیگر واسپاری می‌کند، و سپس حذف Facade و ارتباط دادن مستقیم کلاینت‌ها با کلاس‌های کوچک متخصص (در صورت نیاز) است.

۲,۱,۳ مزایا

- کنترل برنامه دیگر در یک موجودیت پیچیده مرکزیت نیافته است، بلکه بین موجودیت‌های با مجموعه مسئولیت‌های مشخص و محدود پخش شده است. در نتیجه طراحی نرم‌افزار نیز از الگوی طراحی رویه‌ای به طراحی شیء گرا که در آن اشیاء مستقل با هم تعامل می‌کنند بهبود پیدا می‌کند.
- فهم و نگهداری از بخش‌های مختلف کد God Class اولیه آسان‌تر می‌شود.
- بخش‌هایی از God Class اولیه (که اکنون در کلاس‌های مختلف قرار گرفته اند) نسبت به تغییرات stable تر می‌شوند.
- زمان کامپایل برنامه (به دلیل ساده شدن dependency ها) احتمالا کمتر می‌شود.
- قبل از اعمال الگو کپسوله سازی میان کلاس‌های داده‌ای و God Class نقض می‌شد اما بعد از الگو حفظ می‌شود.
- با کاهش Coupling و افزایش Cohesion کلاس‌ها تغییر منتشر شونده و کد تکراری رفع خواهد شد.

[1] maintainer

- قابلیت استفاده مجدد بیشتر می‌شود.
- اصول شئ‌گرای OCP، ISP و PLK تا حد خوبی برقرار می‌شوند.

۲,۱,۴ معایب

- شکستن God Class یک فرآیند سخت، زمان‌بر، و کُند است.
- تعداد کلاس‌ها و در نتیجه پیچیدگی اولیه ورود به کد برنامه (به ویژه برای افراد جدید) ممکن است افزایش پیدا کند. هر چند معمولا داشتن کلاس‌های کوچک اما تخصصی موجب سادگی کد و افزایش فهم آن می‌شود.
- نگهداری کنندگان از سیستم دیگر نمی‌توانند برای رفع مشکلی که باید اصلاح شود به سادگی به سراغ یک کلاس واحد بروند. هر چند در حال حاضر با پیشرفت ابزارهای IDE که کمک زیادی به فرآیند Feature Location و جستجو در کد کرده اند، این نیاز تا حد زیادی رفع شده است.
- پتانسیل بالا رفتن Coupling و پیچیدگی ارتباطات میان کلاس‌ها در صورت حذف Facade و ارتباط دادن مستقیم کلاینت‌ها با کلاس‌های کوچک متخصص

۲,۱,۵ گام های اعمال الگو

مراحل انجام کار به صورت زیر است و لازم است پس از هر مرحله تست‌های رگرسیون[1] صورت پذیرد:

۱. شناسایی مجموعه‌هایی Cohesive از متغیرهای شیء (instance variables) در God Class و تبدیل آنها به External Data Container ها. همچنین باید متدهای initialization کلاس تغییر داده شوند تا از این کلاس‌های جدید استفاده کنند.

۲. شناسایی تمامی کلاس‌های داده‌ای استفاده شده توسط God Class (شامل کلاس‌هایی که در گام یک ایجاد شدند) و اعمال الگوی حرکت دادن رفتار به سمت داده[2] بر روی آنها، تا این ظرف‌های داده‌ای تبدیل به ارائه‌کنندگان سرویس شوند. متدهای God Class به سادگی تغییر داده می‌شوند تا رفتار را به این متدهای منتقل شده واسپاری کنند.

۳. تکرار گام های اول و دوم تا زمانی که از God Class تنها یک Facade به همراه یک متد مقداردهی اولیه به متغیرها[3] باقی بماند. سپس وظیفه initialization به یک کلاس مستقل سپرده می‌شود تا تنها

[1] Regression Tests

[2] Move Behavior Close to Data

[3] initialization method

یک Facade باقی بماند. در اینجا می‌توان کلاینت‌ها را به اشیائی که God Class سابق برای آنها Facade است انتقال داد، تا بتوان Facade را deprecate یا حذف کرد.

۲,۲ بررسی الگوی دوم: Tease Apart Inheritance

۲,۲,۱ مقدمه ای بر الگو

ارث بری قابلیت بزرگی است که به برنامه شیء گرا امکان می‌دهد کدهای *فشرده شده* در زیرکلاس‌ها نوشته شود. در واقع بالا بودن موقعیت یک متد در ساختار توارثی، اهمیت آن را بالا می‌برد. در عین حال بسیار محتمل است که برای انجام یک مسئولیت، یک زیرکلاس کوچک اضافه کنید. مدتی بعد نیز زیرکلاس‌های دیگر را برای انجام همان کار در بخش‌های دیگر ساختار توارثی اضافه کنید. اضافه کردن زیرکلاس‌های برای انجام کارهای مختلف در بخش‌های مختلف ساختار توارثی ممکن است موجب به وجود آمدن کد اسپاگتی شود.

ساختارهای توارثی درهم (چند وجهی) منجر به code duplication می‌شود که نابود کننده سیستم‌های شیء گراست. در چنین ساختار توارثی، به دلیل اینکه استراتژی‌های حل یک مساله مشخص در جاهای مختلف پخش شده است، اعمال تغییرات بسیار دشوار است. همچنین فهم کد سخت است، بدین معنا که به عنوان مثال شما نمی‌توانید بگویید این ساختار توارثی که اینجاست نتیجه را محاسبه می‌کند، بلکه باید بگویید که خب این بخش نتایج را محاسبه می‌کند، و زیرکلاس‌هایی برای ارائه نتایج به صورت جدول وجود دارد، و هر یک از آن‌های زیرکلاس‌هایی برای هر یک از کشورها دارد.

به سادگی ممکن است یک ساختار توارثی در نرم‌افزار به وجود آید که همزمان دو کار مختلف را انجام می‌دهد. اگر تمام کلاس‌های یک سطح ساختار توارثی زیرکلاس‌هایی داشته باشند که با یک صفت یکسان نامگذاری شده است، احتمال زیادی وجود دارد که دو کار مختلف در یک ساختار توارثی واحد در حال انجام است.

زمانی که یک ساختار توارثی داریم که همزمان دو کار را انجام می‌دهد، از طریق اعمال الگوی Tease Apart Inheritance دو ساختار توارثی مستقل ایجاد می‌کنیم و با استفاده از واسپاری[1] از درون یک ساختار توارثی، دیگری را فراخوانی می‌کنیم. در واقع nested generalization داریم (ساختارهای توارثی چند وجهی) که با اعمال الگو از هم کنده می شوند و با association به صورت DIP به هم متصل می‌شوند. در نتیجه اعمال این

[1] delegation

الگو اصل DIP برقرار می‌شود و جلوگیری از تغییرات منتشر شوند موجب بالا رفتن OCP می‌شود. البته تغییر ساختارهای توارثی ممکن است موجب فرآیند دشواری باشد و موجب بروز تغییرات زیادی در کدهای موجود شود.

این الگو یک الگوی درشتانه است، چون سیستم‌های legacy معمولا طی زمان ساختارهای توارثی پیچیده پیدا می‌کنند. الگوی طراحی Bridge نوع خاصی از این الگوست. همچنین الگوی State و Role Played هم نمونه‌های این الگو هستند.

۲,۲,۲ چگونگی اعمال الگو

مهم‌ترین مسئولیت ساختار توارثی را به عنوان وجه اصلی آن در نظر می‌گیریم. وجه (یا وجوه) دیگر ساختار توارثی را به صورت ساختارهای توارثی مستقل جدا میکنیم و رفتار مورد نظر را به آنها واسپاری می‌کنیم. این رابطه Association ممکن است یک طرفه (از ساختار توارثی اصلی به وجوه فرعی) یا دو طرفه (با نگهداری همزمان یک ارجاع مانا از ساختار فرعی به ساختار توارثی اصلی) باشد.

۲,۲,۳ مزایا

- برقراری DIP
- بالا رفتن Cohesion (سپردن کارهای تخصصی به ساختارهای توارثی مستقل)
- ساده شدن کد و بالا رفتن قابلیت فهم سیستم
- کاهش انتشار تغییرات (بهبود OCP) و بالا رفتن تغییر پذیری سیستم با کاهش Coupling که امکان گسترش یا تغییر ساختارهای توارثی به صورت مستقل (بدون انتشار تغییرات) را می‌دهد
- جلوگیری از Code Duplication با استفاده از واسپاری عملیات تخصصی به کلاس فراهم آورنده آن سرویس
- حذف Parallel Inheritance Hierarchies که هدف اصلی الگوست.
- افزایش قابلیت استفاده مجدد از ساختارهای توارثی
- افزایش انعطاف‌پذیری به علت استفاده از Delegation

۲,۲,۴ معایب

- پتانسیل به وجود آمدن زنجیره دید ترایا، در صورتی که الگو به درستی اعمال نشود. زیرا واسپاری عملیات به ساختارهای توارثی مستقل ممکن است با Message Chain همراه شود.

- پتانسیل بروز Long Parameter List: اگر متدهای ساختار توارثی جدا شده نیاز به متغیرهای زیادی از کلاس اصلی داشته باشند. البته دید دو طرفه (یا ارسال خود شیء فراخوانی کننده به عنوان پارامتر به متد delegate) یا تعریف interface ها برای محدود کردن دید می‌تواند این مشکل را حل کند.
- پتانسیل بروز Feature Envy: اگر رفتارهایی که به هر دو وجه ساختار توارثی مرتبط هستند به خوبی جانمایی نشوند.
- پتانسیل به وجود آمدن Lazy Class: اگر کلاس‌هایی که مرتبط با وجه جدا شده از ساختار توارثی اصلی هستند به درستی بررسی و حذف نشوند.
- پتانسیل بروز Inappropriate Intimacy: اگر وجوه مختلف ساختار توارثی طوری تفکیک شوند که ساختار تفکیک شده برای انجام مسئولیت‌های خود همچنان وابستگی شدید به ساختار اصلی داشته باشد.

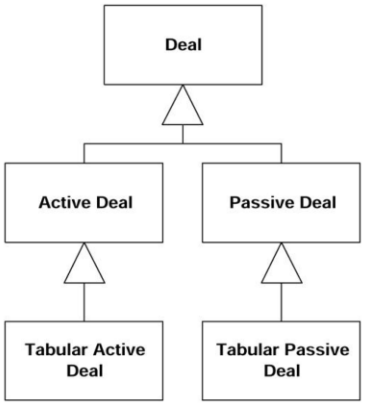
۲,۲,۵ گام های اعمال الگو

۱. شناسایی کارهای مختلفی که توسط ساختار توارثی در حال انجام است. یک جدول دو بعدی (یا چند بعدی اگر کد شما مشکلات جدی در این حوزه دارد) بکشید و هر محور را با یک کار خاص نامگذاری کنید. ما فرض می‌کنیم هر بعد اضافی نیازمند یک بار تکرار این فرآیند بازآرایی (در کدام در یک زمان) است.
۲. تصمیم بگیرید که کدام کار مهم‌تر است و در ساختار توارثی فعلی باقی می‌ماند، و کدام کار می‌بایست به ساختار توارثی دیگری منتقل شود.
۳. از الگوی Extract Class در Superclass مشترک استفاده کنید تا یک کلاس برای کار جانبی بسازید و یک متغیر شیء[1] اضافه کنید که ارجاع به شیء از این کلاس را نگهداری خواهد کرد.
٤. برای هر یک از زیرکلاس‌های ساختار توارثی اولیه، یک زیرکلاس برای کلاس استخراج شده بسازید. متغیر شیء ساخته شده در گام قبل را با نمونه این کلاس مقداردهی اولیه کنید.
۵. از الگوی Move Method در هر یک از زیرکلاس‌ها استفاده کنید تا رفتار زیرکلاس را به شیء استخراج شده منتقل کنید.
٦. زمانی که زیرکلاس هیچ کد دیگری ندارد، آن را حذف کنید.

[1] instance variable

۷. این کار را تا زمانی که تمامی زیرکلاس‌های مد نظر از ساختار توارثی اولیه حذف شوند ادامه دهید. سپس ساختار توارثی جدید را بررسی کنید تا در صورت امکان refactoring های بعدی مثل Pull Up Method و Pull Up Field را بر روی آن انجام دهید.

در ادامه این گام ها را در یک مثال خواهیم دید.



این ساختار توارثی به این شکل درآمده است زیرا Deal در ابتدا صرفا جهت مدیریت یک Single Deal استفاده میشد. سپس کسی ایده نمایش جدولی از Deal ها را داد. با مقدار کمی تلاش اضافه کرد یک جدول به کلاس Active Deal انجام شد. برای اضافه شدن جدول‌ها به Deal های Passive نیز یک زیرکلاس کوچک به آن اضافه شد. دو ماه بعد کد جدول پیچیده‌تر شده بود اما یک جای واحد برای قرار گرفتن آن وجود نداشت. بدلیل وجود فورس‌های زمانی پروژه، داستان تکرار شد. در حال حاضر اضافه کردن یک نوع جدید Deal کار دشواری است، زیرا منطق Deal با منطق نمایش آن گره خورده است.

جهت اعمال الگو، بر اساس گام‌های بالا اولین قدم شناسایی کارهای مختلفی است که توسط ساختار توارثی انجام می‌شوند. یک کار رفتارهای متغیر بر اساس نوع Deal است، یک کار دیگر رفتارهای مرتبط با نحوه نمایش[1] است که جدول زیر را نتیجه می‌دهد:

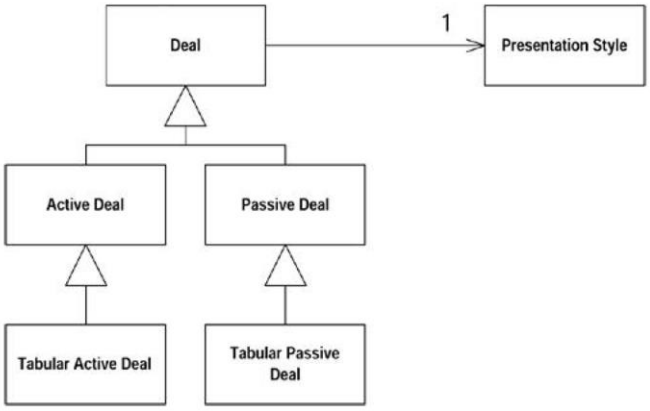
Deal	Active Deal	Passive Deal
Tabular Deal		

گام بعد تصمیم روی این است که کدام کار مهم‌تر است. اهمیت Deal بودن شیء خیلی مهم‌تر از نحوه نمایش آن است، پس ما Deal را سر جای خود می‌گذاریم و نحوه نمایش را در یک ساختار توارثی جدا قرار می‌دهیم.

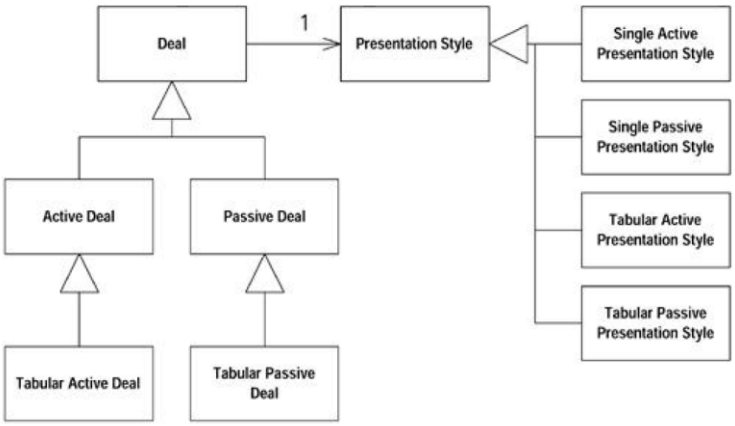
[1] presentation style

در عمل، معمولا کاری که حجم کد بیشتری با آن مرتبط است باقی می‌ماند و کار با حجم کد کمتر به ساختار توارثی جدید منتقل می‌شود.

گام بعد به ما می‌گوید که از الگوی Extract Class برای ساختن نحوه نمایش استفاده کنیم:



گام بعد به ما می‌گوید که برای هر یک از زیرکلاس‌های ساختار توارثی اصلی، یک زیرکلاس برای کلاس استخراج شده بسازیم (شکل زیر) و متغیرهای مرتبط را با کلاس متناسب مقداردهی کنیم:

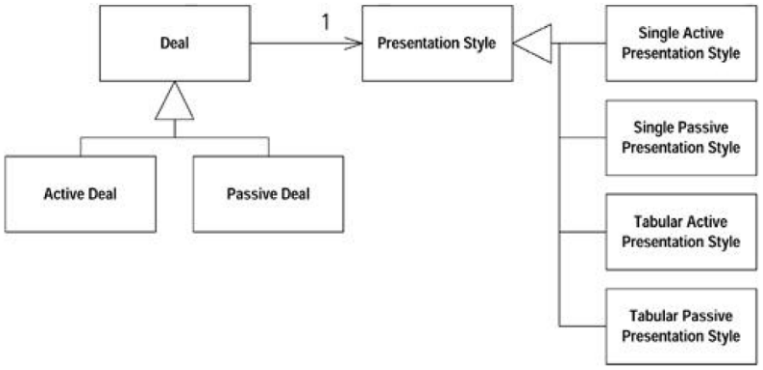



```
ActiveDeal constructor
  ...presentation= new SingleActivePresentationStyle();...
```

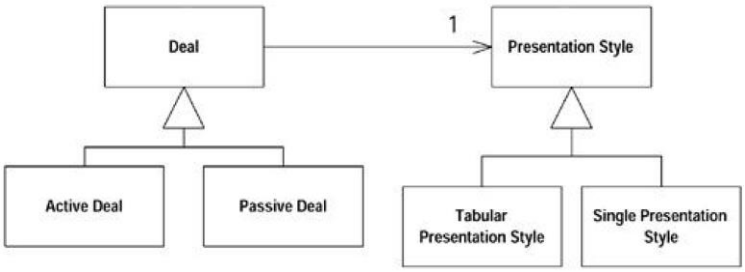
توجه کنید که در این مقطع زمانی، با هدف ساده کردن ساختار یک گام به عقب برداشته‌ایم و تعداد کلاس‌ها بیشتر شده است. زیرا امن تر است که در هر زمان یک گام از فرآیند refactoring را برداریم، به جای آنکه با عجله چندین گام جلو برویم.

حال از الگوهای Move Method و Move Field استفاده می‌کنیم تا رفتارها و داده‌های مرتبط با نحوه نمایش را از زیرکلاس‌های Deal به زیرکلاس‌هایی جدید که برای نحوه نمایش ساخته‌ایم منتقل کنیم. با انجام این کار

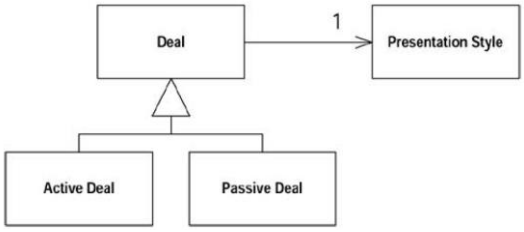
هیچ کدی در کلاس‌های Tabular Active Deal و Tabular Passive Deal باقی نمی‌ماند، بنابراین آنها را حذف می‌کنیم:



حال که دو کار مختلف را از هم جدا کرده‌ایم، می‌توانیم هر یک را به تنهایی مورد بررسی قرار دهیم و ساده کنیم. گام بعدی این است که از متمایز کردن Active و Passive در نحوه نمایش خلاص شویم:



حتی تمایز میان نمایش‌های Single و Tabular نیز ممکن است بدون توارث و به سادگی قابل مدیریت باشد و به ساختار بسیار ساده شده زیر منتج شود:



۲,۳ مقایسه دو الگو

۲,۳,۱ مقایسه زمان استفاده از دو الگو (when) در شرایط مساله

الگوی Split Up God Class زمانی که در یک کلاس تغییر زیادی داشته باشیم، یا تغییرات مختلف در سیستم منجر به ایجاد تغییر در یک کلاس خاص شوند مورد استفاده قرار می‌گیرد. در واقع در شرایطی از این الگو استفاده می‌کنیم که:

- یک کلاس به دلیل داشتن مسئولیت‌های بسیار زیاد کنترل کل برنامه را در دست دارد . در این حالت هر تغییر در برنامه (نیازمندی‌ها) موجب تغییر برخی از مسئولیت‌های این کلاس می‌شود.
- شناسایی اینکه تغییرات لازم برای پیاده سازی یک قابلیت باید کجا صورت پذیرند تا سایر کلاس‌های سیستم را تحت تاثیر قرار ندهند دشوار است. همچنین تغییر سایر کلاس‌ها معمولا موجب ایجاد تغییر در یک کلاس خاص (God Class) می‌شود.
- به دلیل پیچیدگی زیاد God Class، امکان تغییر بخشی از رفتار آن به صورت Black Box غیر ممکن است.

همچنین تفکر رویه‌ای[1] برنامه نویس ممکن است موجب دور شدن رفتار از داده، و به وجود آمدن God Class گردد. در اینجا معمولا تغییرات منتشر شونده به معنای نیاز به اضافه کردن کلاس جدید نیست، بلکه تغییرات به God Class منتشر می‌شود.

در الگوی Tease Apart Inheritance ساختارهای توارثی چند وجهی داریم که به دلیل پخش بودن یک مسئولیت واحد در چندین کلاس، تغییرات منتشر شونده داریم. همچنین اضافه کردن زیر کلاس به کلاسی از یک بخش از ساختار توارثی، اضافه شدن زیر کلاس‌هایی به بخش‌های دیگر ساختار توارثی را موجب می‌شود.

۲,۳,۲ مقایسه نحوه استفاده از دو الگو (how) در شرایط مساله

الگوی Split Up God Class طی یک فرآیند چند مرحله‌ای مسئولیت‌های God Class را به سمت Data Class ها حرکت می‌دهد. کاهش Coupling و افزایش Cohesion موجب خواهد شد دیگر تغییرات مختلف، به صورت دائم موجب تغییر God Class نباشند. زیرا God Class پس از اعمال الگو به شکل یک Facade در می‌آید (یا به کلی حذف می‌شود).

[1] Procedural

الگوی Tease Apart Inheritance با تفکیک جنبه‌های مختلف و متمرکز کردن جنبه‌های مختلف مسئولیت در ساختارهای توارثی مستقل، Cohesion را بالا می‌بود و از Duplicated Code (که نوعی از Coupling است) جلوگیری می‌کند، لذا انتشار تغییرات در سیستم کمتر خواهد شد.

هر دو الگو بر روی یک سیستم موجود (عمدتا در زمان نگهداری سیستم)، تغییرات درشتانه ایجاد می‌کنند. البته الگوی Split Up God Class قطعا بر روی یک سیستم موجود است، اما الگوی Tease Apart Inheritance ممکن است اواخر توسعه سیستم باشد و یا اینکه بر روی یک سیستم موجود اعمال شود.

۳. موقعیت سوم

در برخی از سیستم‌ها بین اشیاء در زمان اجرا زنجیره دید ترایا[1] وجود دارد، یعنی به جای آنکه از اشیاء مسئولیت خواسته شود، اشیائ دیگری که در اختیار دارند (چه به عنوان فیلد داخلی و چه در نتیجه فراخوانی یکی از متدهای شیء) از آنها دریافت می‌شود و عملیات مورد نظر مستقیما روی این اشیاء فراخوانی می‌شود.

دید ترایا معمولا نشانه قرارگیری نادرست مسئولیت‌ها و نقض Encapsulation است. در واقع state داخلی object به اشیاء بیرونی expose می‌شود که نقض Encapsulation است و موجب coupling بالا و انتشار تغییرات می‌شود. این امر تغییرپذیری سیستم را پایین می‌آورد. راه حل کلی مقابله با این شرایط حرکت دادن رفتار به سمت داده است.

در شرایط ذکر شده در مساله، به دلیل وجود navigation در فراخوانی متدها میان اشیاء، Coupling بالایی بین آنها برقرار می‌شود و تغییر در interface یک کلاس نه تنها کلاینت‌های مستقیم آن کلاس را تحت تاثیر قرار می‌دهد، بلکه کلاینت‌های غیرمستقیم (که از طریق navigation به این اشیاء از نوع این کلاس دسترسی پیدا می‌کنند) را نیز تحت تاثیر قرار می‌دهد.

راه حل کلی این است که در چندین مرحله، رفتار تعریف شده توسط کلاینت غیرمستقیم را به کلاسی که داده‌های مورد نیاز برای انجام آن رفتار را در اختیار می‌گذارد انتقال دهیم.

برای تشخیص دید ترایا باید به دنبال سرویس‌دهندگان غیرمستقیم[2] باشیم:

- هر زمان که چنین کلاسی تغییر می‌کند، مثلا با تغییر ساختار داخلی یا اشیائی که با آنها تعامل می‌کند، نه تنها کلاینت‌های مستقیم بلکه کلاینت‌های غیر مستقیم آن نیز باید تغییر کنند.

[1] Transitive Visibility Chain

[2] indirect providers

- چنین کلاس‌هایی تعداد زیادی public attribute دارند، یا متدهایی دارند که مقادیر attribute های آنها را به کلاینت‌ها ارائه می‌دهند.
- معمولا در کنار این سرویس‌دهندگان غیر مستقیم، ساختارهای بزرگ اتصال کلاس‌ها (که عمدتا کلاس‌های داده‌ای هستند) وجود دارند.

همچنین برای تشخیص دید ترایا می‌توان به دنبال کلاینت‌های غیرمستقیم[1] بود که شامل حجم زیادی از navigation code هستند. این کد به دو صورت است:

- یک سلسله از دسترسی به attribute ها: مثلا a.b.c.d؛ نتیجه این سلسله دسترسی می‌تواند در یک متغیر ریخته شود یا یک متد روی آخرین شیء فراخوانی شود (مثلا a.b.c.d.op()). در صورت private کردن attribute ها جلوی این نوع از دسترسی‌ها گرفته می‌شود.
- یک سلسله از فراخوانی متدهای accessor: مثلا object.m1().m2().m3() که m1 یک متد از object است، m2 یک متد از شیءی است که نتیجه فراخوانی m1 است، و m3 یک متد از شیءی است که به عنوان خروجی فراخوانی m2 برگردانده شده است.

باید در نظر داشت که بعضی اوقات وجود متغیرهای میانی ممکن است پیدا کردن کدهای navigation را سخت کنند. به عنوان مثال کد زیر در نگاه اول شامل زنجیره دید ترایا نیست:

```
Token token;
token = parseTree.token();
...
if (token.identifier() != null) {
...
```

اما در واقع در کد بالا زنجیره دید ترایا به شکل زیر وجود دارد:

```
if (parseTree.token().identifier() != null) {
    ...
```

معمولا دید ترایا به یکی از علت‌های زیر در برنامه به وجود می‌آید:

- دید رویه‌ای برنامه نویس که موجب می‌شود داده و رفتار به صورت جدا از هم تعریف شوند
- در فرآیند نگهداشت سیستم، ممکن است فورس زمانی موجب شود که رفتار در بین کلاس‌های داده‌ای شکسته نشود، بلکه به صورت متمرکز پیاده سازی گردد (این امر می‌تواند علت به وجود آمدن God Class در نتیجه عدم کارچرخانی در Facade یا Mediator هم باشد)

[1] indirect clients

- عدم رعایت Encapsulation: استفاده از الگوهایی مانند Encapsulate Field، Self-Encapsulate Field، و Encapsulate Collection می‌تواند از به وجود آمدن زنجیره دید ترایا جلوگیری کند.

در این بخش به شرایطی که بین اشیاء زنجیره‌های دید ترایا وجود دارد خواهیم پرداخت. برای حل مشکل دو الگوی زیر را بررسی خواهیم کرد:

- Eliminate Navigation Code
- Inline Class

الگوی Eliminate Navigation Code از الگوهای مهندسی مجدد، بازتعریف مسئولیت‌ها[1] است. این الگو دقیقا با هدف برقراری اصل PLK[2] و از بین بردن دید ترایا به وجود آمده است. الگوی Inline Class که نسبتا الگوی ریزدانه‌ای می‌باشد از الگوهای Refactoring در شاخه Move Features Between Objects است که بیشتر برای حذف کلاس‌های زائد (Lazy Class ها) به وجود آمده است. الگوی اول درشتانه تر از الگوی دوم است و معمولا می‌بایست طی چندین Iteration اعمال گردد.

۳,۱ بررسی الگوی اول: Eliminate Navigation Code

۳,۱,۱ مقدمه ای بر الگو

از بین بردن دید ترایا مشابه از بین بردن God Class از الگوی کلی Move Behavior Close to Data استفاده می‌کند، یعنی طی چندین مرحله رفتار را به سمت داده حرکت می‌دهیم، در عین حال در هر مرحله قدری از رفتار باقی می‌ماند. با انجام سرویس‌گیری به جای رد کردن object به عنوان خروجی، Encapsulation برقرار می‌شود.

این الگو در جهت برقراری اصل PLK از اصول شیء گرایی (حفظ قانون دیمیتر) است که با انتقال مسئولیت‌ها به سمت انتهای یک زنجیزه فراخوانی، میزان انتشار تغییرات را کاهش می‌دهد.

هر چند navigation code را می‌توان با الگوریتم‌های Pattern Matching ساده پیدا کرد، هدف ما پیدا کردن سلسله‌های فراخوانی است که منجر به بروز Coupling میان کلاس‌ها شده باشد. در واقع فراخوانی‌هایی که یک شیء را به شیء دیگر تبدیل می‌کنند[3] معمولا مشکل زا نیستند. مثلا:

```
leftSide().toString();
```

[1] Redistribute Responsibilities

[2] Principle of Least Knowledge

[3] object conversion

```
i.getValue().isShort();
```

لذا باید به دنبال سلسله‌های فراخوانی با طول بیش از ۲ باشیم و همچنین متدهای شناخته شده و استاندارد برای تبدیل اشیاء به هم یا تبدیل از / به انواع اولیه[1] را صرف نظر کنیم.

۳,۱,۲ چگونگی اعمال الگو

در این الگو در هر مرحله، رفتار را یک گام به سمت داده حرکت می‌دهیم. در عین حال احتمالا بخشی از رفتار (شامل پیش‌پردازش‌ها و پس‌پردازش‌ها) در متد فراخوانی کننده باقی می‌ماند.

۳,۱,۳ مزایا

- حذف زنجیره‌های وابستگی میان کلاس‌ها؛ در نتیجه coupling کمتر می‌شود و تغییرات منتشر شونده به کلاینت‌ها کمتر خواهد بود.
- حذف Data Class ها و تبدیل آنها به سرویس‌دهندگان
- برقراری Encapsulation برای کلاس‌های داده‌ای که موجب بالا رفتن قابلیت نگهداری و خطایابی سیستم می‌شود.
- تجمیع شدن کد کلاینت‌های غیر مستقیم در خود کلاس حاوی داده‌ها، احتمالا موجب حذف duplicated code خواهد شد.
- قرار گرفتن رفتار در کنار داده موجب افزایش Cohesion و برقراری اصل Information Expert از اصول شیء گرایی می‌شود.
- قابلیت‌های ضمنی[2] در سیستم نام‌گذاری می‌شوند و به صورت صریح[3] در اختیار کلاینت‌ها قرار می‌گیرند. این کار فهم‌پذیری سیستم را بالا می‌برد.
- افزایش قابلیت استفاده مجدد از کلاس‌ها چون رفتارشان به کنار خود داده هدایت شده است.

۳,۱,۴ معایب

- اعمال سیستماتیک این الگو ممکن است منجر به تولید interface های بزرگ شود. به طور خاص اگر یک کلاس تعدادی متغیر از نوع collection داشته باشد، اعمال الگو ممکن است موجب تعریف

[1] primitive types

[2] implicit

[3] explicit

رفتارهای زیادی برای محافظت از این collection ها (و در عین حال سرویس‌دهی به کلاینت‌ها جهت انجام عملیات بر روی آنها) شود.

- پتانسیل بروز Middle Man: اعمال سیستماتیک الگو ممکن است موجب به وجود آمدن متدهای صرفا واسپاری کننده رفتار در زنجیره بشود. در واقع در برخی شرایط کلاینت می‌بایست مستقیما با کلاس‌های انتهای زنجیره در ارتباط قرار گیرد. در صورت لزوم باید از الگوی Remove Middle Man استفاده شود.
- غیر مستقیم کردن روابط می تواند باعث انتشار تغییرات در کلاس‌های میانی شود زیرا بعد از اعمال الگو آنها در انجام کار کلاینت دخیل هستند.
- افت کارایی به علت غیر مستقیم کردن ارتباط کلاینت با سرویس‌دهنده ها

۳,۱,۵ گام های اعمال الگو

فرآیند حذف دید ترایا در واقع با اعمال چندین بار الگوی Move Behavior Close to Data انجام می‌شود و شامل گام‌های زیر است:

۱. شناسایی کد navigation که باید انتقال داده شود
۲. اعمال الگوی Move Behavior Close to Data جهت حذف یک سطح از navigation
در این نقطه باید تست‌های رگرسیون اجرا شوند تا از صحت عملکرد خود اطمینان پیدا کنیم.
۳. تکرار گام‌های بالا در صورت نیاز

باید توجه داشت که این گام‌ها کدهای عملیات را از کلاینت به سمت سرویس‌دهنده واقعی حرکت می‌دهند. وجود فیلد یا متدهای دسترسی[1] تفاوتی در مشکل دید ترایا ندارد و نباید به اشتباه با تعریف متدهای دسترسی به فیلد، گمان کنیم که مشکل دید ترایا را حل کرده‌ایم.

۳,۲ بررسی الگوی دوم: Inline Class

۳,۲,۱ مقدمه ای بر الگو

الگوی Inline Class بر عکس الگوی Extract Class است و زمانی استفاده می‌شود که مسئولیت‌های کلاس کمتراز آن است که نیازی به حفظ آن باشد. معمولا این اتفاق نتیجه فرآیند refactoring است که عمده وظایف کلاس را به خارج از کلاس انتقال می‌دهد و لذا مقدار کنی مسئولیت (داده و رفتار) در کلاس باقی می‌ماند. در

[1] accessor methods

این زمان ترکیب کردن این کلاس در یک کلاس دیگر (کلاسی که بیشترین استفاده را از این کلاس دارد) منطقی به نظر می‌رسد.

همچنین احتمال دارد که بخشی از داده‌ها و رفتارهای ذاتی کلاس به کلاس دیگری داده شده باشد که این موجب coupling بالا بین آنها و همچنین Inappropriate Intimacy خواهد داشت. در این حالت با Inline کردن کلاس داده و رفتارها را در یک کلاس تجمیع می‌کنیم.

در واقع با استفاده از این الگو کلاس‌هایی که کار بسیار کمی انجام می‌دهند (Lazy Class ها) را به صورت Inline درون کلاس‌های دیگر جا می‌دهیم.

۳,۲,۲ چگونگی اعمال الگو

استفاده از این الگو در زمانی است که یک کلاس کار زیادی انجام نمی‌دهد. در این شرایط تمام مسئولیت‌های آن (داده ها و رفتارها) را به کلاس دیگری انتقال می‌دهیم (احتمالا استفاده کننده اصلی از کلاس) و این کلاس را حذف می‌کنیم.

۳,۲,۳ مزایا

- از بین بردن واسپاری اضافی (مثلا جهت از بین بردن Speculative Generality)
- قرار دادن مجموعه‌ای از رفتارها در کنار هم (در راستای جلوگیری از Shotgun Surgery)
- بالا رفتن فهم‌پذیری کد و قابلیت نگهداشت آن بر اساس ساده شدن کد
- کاهش Coupling با حذف یک کلاس و انتقال مسئولیت‌های آن به کلاس دیگر
- پتانسیل از بین بردن Inappropriate Intimacy که ممکن است قبل از اعمال الگو در ارتباط اضافی کلاس مبدا و نهایی وجود داشته باشد
- از بین بردن واسپاری اضافی
- پتانسیل از بین بردن Middle Man و Transitive Visibility
- بهبود جزئی در کارایی با کاهش یک سطح از واسپاری

۳,۲,۴ معایب

- در صورتی که کلاس مبدا چندین کلاینت مستقل داشته باشد، ممکن است Coupling اضافی به کلاس نهایی به وجود آید.
- پتانسیل کاهش Cohesion در کلاس نهایی با اضافه شدن مسئولیت‌های کلاس مبدا به آن

- پتانسیل پیچیده شدن interface کلاس نهایی با اضافه شدن public interface کلاس مبدا به آن (و در نتیجه افزایش پیچیدگی کد)
- اعمال زیاد الگو می‌تواند باعث ایجاد God class شود، زیرا مرز مشخصی برای تشخیص اینکه چه کلاسی می‌بایست در دیگری ادغام شود وجود ندارد.

۳,۲,۵ گام های اعمال الگو

۱. تمامی متدهای public کلاس مبدا (کلاسی که آن را به صورت inline در کلاس دیگر جا می‌دهیم و در نهایت حذف خواهد شد) را در کلاس نهایی تعریف کنید. تمامی این متدها را به کلاس نهایی واسپاری کنید.

توجه: اگر یک interface مجزا برای متدهای کلاس مبدا منطقی به نظر می‌رسد، قبل از انجام فرآیند inlining از الگوی Extract Interface استفاده کنید.

۲. تمامی ارجاعات به کلاس مبدا را تبدیل به ارجاع به کلاس نهایی کنید.

توجه: کلاس مبدا را private کنید و سپس نام آن را نیز تغییر دهید تا کامپایلر تمامی ارجاعات به آن را تشخیص دهد.

۳. کد را کامپایل و تست کنید.

٤. از الگوی Move Method و Move Field برای انتقال قابلیت‌های کلاس مبدا به کلاس نهایی استفاده کنید تا زمانی که چیزی باقی نماند.

۵. کلاس مبدا را حذف کنید.

در ادامه این گام ها را در یک مثال خواهیم دید.

بر اساس مثال Person و TelephoneNumber در ابتدا کلاس‌های زیر را داریم:

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }
    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

```
class TelephoneNumber...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
```

در گام اول تمامی متدهای public کلاس TelephoneNumber را در Person تعریف می‌کنیم:

```
class Person...
    String getAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
    String getNumber() {
        return _officeTelephone.getNumber();
    }
    void setNumber(String arg) {
        _officeTelephone.setNumber(arg);
    }
```

سپس کلاینت‌های TelephoneNumber را پیدا می‌کنیم و آنها را تغییر می‌دهیم تا از Person استفاده کنند، بنابراین کد زیر

```
Person martin = new Person();
martin.getOfficeTelephone().setAreaCode ("781");
```

به این کد تبدیل می‌شود:

```
Person martin = new Person();
```

```
martin.setAreaCode ("781");
```

حال از الگوهای Move Method و Move Field استفاده می‌کنیم تا زمانی که کلاس TelephoneNumber هیچ داده یا رفتاری نداشته باشد و قابل حذف باشد.

۳,۳ مقایسه دو الگو

۳,۳,۱ مقایسه زمان استفاده از دو الگو (when) در شرایط مساله

هدف الگوی Eliminate Navigation Code اصولا از بین بردن زنجیره دید ترایا است، اما الگوی Inline Class عمدتا جهت حذف Lazy Class است که در عین حال ممکن است برای از بین بردن زنجیره دید ترایا هم مورد استفاده قرار گیرد (در زمانی که کلاس مبدا برخی از رفتارهای خود را به کلاس‌های دیگر واسپاری کرده است).

الگوی اول درشتانه‌تر از الگوی دوم است.

۳,۳,۲ مقایسه نحوه استفاده از دو الگو (how) در شرایط مساله

الگوی Eliminate Navigation Code با حرکت دادن رفتار به سمت داده، زنجیره‌های دید ترایا را به صورت مرحله به مرحله از بین می‌برد، اما یکبار اعمال الگوی Inline Class حداکثر یک سطح از زنجیره دید ترایا را از بین می‌برد و برای اصلاح مشکل دید ترایا در سیستم در ادامه ممکن است نیاز به اعمال الگوی اول را همچنان داشته باشیم.

۴. مراجع

[GoF 95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[Fowler 99] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.