Refactoring

احسان رضایی

NAGHOOS
PUBLICATION

سرشناسه	: رضايي , احسان , 1371
عنوان و نام پديدآور	: Rfactoring/ مولف احسان رضايي
مشخصات نشر	: تهران : ناقوس، 1400.
مشخصات ظاهری	: 80ص. مصور، جدول , نمودار
شابک نسخه چاپي	: 978-600-473-409-7 :450.000ريال
وضعيت فهرست‌نويسي	: فيپا
يادداشت	: برای گرد آوری کتاب حاضر از کتاب Refactoring اثرمارتین فاولر استفاده شده است
موضوع	: فاکتور گیری مجدد نرم افزارهای کامپیوتری
موضوع	: Software Refactoring
شناسه افزوده	: فاولر، مارتین ، 1963- م
شناسه افزوده	: Fowler , Martin
رده بندی کنگره	: QA76/758
رده بندی ديويی	: 005/14
شماره کتابشناسی ملی	: 8516362

NAGHOOS PUBLICATION

انتشارات ناقوس

نام کتاب	: Rfactoring
ناشر	: انتشارات ناقوس
مولف	: احسان رضايي
چاپ اول	: 1400
تيراژ	: 300 جلد
چاپ و صحافي	: حيدری
ناظر چاپ	: ياسمن تجملي محدث
قيمت	: 450.000ريال
شابك	: 978 -600-473 - 409 -7

كليه حقوق براي سرمايه گذار محفوظ است. تكثير تمامي يا قسمتي از اين اثر به‌صورت حروفچيني يا چاپ مجدد، چاپ افست، پلي‌كپي، فتوكپي و انواع ديگر چاپ ممنوع است و پيگرد قانوني دارد.

انتشارات ناقوس
خيابان انقلاب- ابتدای وصال شيرازی-ساختمان شماره 7 طبقه اول
تلفاكس : 66478951 - 66478954

درباره مؤلف

احسان رضایی، متولد سال 1371 از 16 سالگی برنامه‌نویسی تحت وب با php را شروع کردم، به عنوان زبان برنامه‌نویسی دوم و رفتن به سمت تجربه‌های جدید golang را یاد گرفتم. نتیجه این تجربه جدید جاب ترندز شد. ابزاری برای تجزیه و تحلیل هوشمند بازار کار ایران.

https://job-trend.ir

به علت این‌که در شهر خودم شغل مناسب پیدا نکردم و مخالف مهاجرت به شهرهای دیگه مثل تهران بودم دورکار شدم.

به نوشتن علاقه دارم و مدت زیادی هست که داخل وبلاگم این کار را انجام می‌دهم. علاوه بر این، سال پیش هم کتاب الگوهای طراحی به بیان ساده را منتشر کردم.

https://www.developit.ir/design-patterns

می‌توانید اخبار و به‌روزرسانی‌های جدید کتاب را داخل وبلاگ یا توییترم دنبال کنید.

blog: http://www.developit.ir
email: me@developit.ir
twitter: @developit121

سخن مؤلف

برای گردآوری این کتاب از وبسایت refactoring.guru، کتاب Refactoring اثر Martin Fowler و نهایتاً تجربه خودم استفاده کردم.

با این‌که تکنیک‌ها تعاریف مشخصی دارند اما محتوای کتاب ترجمه نیست. در مورد مثال‌ها (کدهای برنامه‌نویسی) حدود 30 درصد مثال‌های آورده شده دقیقاً از کتاب Refactoring آمده است.

مثال‌ها، مربوط به زبان برنامه‌نویسی خاصی نمی‌شوند و سینتکس به خصوصی ندارند. سعی کردم با شبه کد تکنیک‌های refactoring را توضیح بدهم تا همه بتوانند از کتاب استفاده کنند.

فهرست مطالب

فصل اول- مقدمه 13
مقدمه 13
بدهی فنی/ Technical debt 15
فصل دوم- Code Smells 17
تعریف بوی بد کد/ Code Smell 17
1) Bloaters 17
1-1) Long Method 18
1-2) Large Class 18
1-3) Long Parameter List 18
1-4) Primitive Obsession 18
1-5) Data Clumps 18
2) Object-Orientation Abusers 18
2-1) Switch Statements 19
2-2) Temporary Field 19
2-3) Refused Bequest 19
2-4) Alternative Classes with Different Interfaces 19
3) Change Preventers 19
3-1) Divergent Change 20
3-2) Shotgun Surgery 20

20 .....Parallel Inheritance Hierarchies (3-3
21 ..... Dispensables(4
21 .....Lazy class (1-4
21 ..... Data class (2-4
22 ..... Duplicate Code (3-4
22 .....Dead Code(4-4
22 ..... Speculative Generality (5-4
22 .....Comments(6-4
22 ..... Couplers (5
22 .....Feature Envy (1-5
23 .....Inappropriate Intimacy (2-5
23 .....Message Chains (3-5
23 ..... Middle Man (4-5
25 ..... Composing Methods -فصل سوم
25 .....Composing Methods
25 ..... Extract Method
26 ..... Inline Method
27 .....Inline Temp
27 .....Extract Variable
28 .....Replace Temp with Query
29 ..... Split Temporary Variable
29 .....Remove Assignments to Parameters
30 ..... Replace Method with Method Object
32 .....Substitute Algorithm
33 ..... Moving Features between Objects -فصل چهارم
33 .....Moving Features between Objects
33 ..... Move Method
35 ..... Move Field

35 ..... Extract Class
36 .....Inline Class
37 ..... Hide Delegate
40 .....Remove Middle Man
42 .....Introduce Foreign Method
43 .....Introduce Local Extension
فصل پنجم- Organizing Data 45
45 .....Organizing Data
45 ..... Self Encapsulate Field
46 .....Encapsulate Field
47 .....Encapsulate Collection
48 ..... Replace Data Value with Object
49 .....Change Value to Reference
50 .....Change Reference to Value
50 .....Replace Array with Object
51 .....Duplicate Observed Data
51 ..... Change Unidirectional Association to Bidirectional
52 ..... Change Bidirectional Association to Unidirectional
52 .....Replace Magic Number with Symbolic Constant
53 .....Replace Type Code with Class
54 .....Replace Type Code with Subclasses
57 .....Replace Subclass with Fields
فصل ششم- Simplifying Conditional Expressions 59
59 ..... Simplifying Conditional Expressions
59 ..... Decompose Conditional
60 ..... Consolidate Conditional Expression
60 .....Consolidate Duplicate Conditional Fragments
61 .....Remove Control Flag

62 ..... Replace Nested Conditional with Guard Clauses
62 ..... Replace Conditional with Polymorphism
64 ..... Introduce Null Object
65 ..... Introduce Assertion
67 ..... Simplifying Method Calls -فصل هفتم
67 ..... Simplifying Method Calls
67 ..... Rename Method
68 ..... Remove Parameter
68 ..... Separate Query from Modifier
69 ..... Parameterize Method
69 ..... Replace Parameter with Explicit Methods
70 ..... Preserve Whole Object
71 ..... Replace Parameter with Method Call
71 ..... Introduce Parameter Object
72 ..... Remove Setting Method
73 ..... Hide Method
73 ..... Replace Constructor with Factory Method
74 ..... Replace Error Code with Exception
75 ..... Replace Exception with Test
77 ..... Dealing with Generalization -فصل هشتم
77 ..... Dealing with Generalization
77 ..... Pull Up Field
77 ..... Pull Up Method
77 ..... Pull Up Constructor Body
78 ..... Push Down Field
78 ..... Push Down Method
78 ..... Extract Subclass
78 ..... Extract Superclass

Extract Interface 78
Collapse Hierarchy 79
Form Template Method 79
Replace Inheritance with Delegation 79
Replace Delegation with Inheritance 79

فصل اول- مقدمه

مقدمه

Refactoring مجموعه‌ای از تکنیک‌هاست که به منظور اصلاح و بهبود کدهای قبلی بدون تغییر در عملکرد و رفتارشان جهت خوانایی، کارآمدی و قابلیت نگهداری بیشتر انجام می‌شود.

در کتاب Refactoring اثر Martin Fowler نوشته شده: refactoring تکنیک مرتب/منظم سازی برای تجدید ساختار کد موجود است. تغییر ساختار داخلی کد بدون تغییر رفتار خارجی آن.

یک نرم‌افزار برای ادامه حیات خود همواره نیاز به نگهداری، رفع مشکلات و به‌روزرسانی دارد. اغلب این به‌روزرسانی‌ها روی کدهای قبلی انجام می‌شود. کد کثیف و افزایش بدهی فنی، توسعه نرم‌افزار را کند و هزینه‌های آن را بالا خواهد برد.

refactoring یک سرمایه‌گذاری و راه‌حلی برای مقابله با کد کثیف و بدهی فنی است که باعث کاهش هزینه‌های توسعه نرم‌افزار در آینده خواهد شد.

refactoring به طور منظم در فرآیند توسعه نرم‌افزار نیاز است. دلایل زیادی برای این کار وجود دارد که اینجا به مهم‌ترین آن‌ها می‌پردازیم.

- وضوح/خوانایی

افزایش خوانایی و درک آسان‌تر کد به شما و سایر اعضای تیم برای توسعه بدون زجر نرم‌افزار کمک خواهد کرد.

- کارایی/ performance

کدهای refactor شده عملکرد بهتری پیدا می‌کنند. نتیجه رفع مشکلات گذشته کد سبک‌تر و سریع‌تری است که علاوه بر افزایش performance نرم‌افزار، باعث افزایش performance توسعه‌دهنگان نیز خواهد شد.

- صرفه‌جویی در زمان و هزینه

توسعه سریع‌تر، استفاده از منابع سخت افزاری کمتر، کار گروهی و... دلیل خوبی است برای سرمایه‌گذاریی که امروز آن را انجام می‌دهید و در آینده بهره می‌برید. refactoring سرمایه‌گذاری در زمان و هزینه‌هاست.

- یافتن و رفع اشکالات

یافتن خطا، رفع و یا جلوگیری از آن روی کد تمیز و قابل فهم به سهولت انجام می‌پذیرد.

- قابلیت نگهداری

refactoring به نرم‌افزار قابلیت نگهداری می‌دهد و باعث افزایش طول عمر آن خواهد شد.

- امنیت

technical debt یا بدهی‌های فنی ناخواسته اغلب به علت تجربه کم توسعه‌دهنده رخ می‌دهد. احتمال این‌که در technical debt مشکلات امنیتی باشد کم نیست. پیشتر توضیح دادم refactoring برای رفع یا کاهش technical debt موثر خواهد بود.

- به‌روز بودن

توسعه‌دهندگان حرفه‌ای همیشه از آخرین نسخه‌ها استفاده می‌کنند تا جایگاه خود را حفظ و ارتقا دهند. همچنین نیازمندی‌های خارجی پروژه مدام در

حال به‌روزرسانی ست. این موضوع دلایل زیادی دارد مثلاً کارایی، امنیت، رفع اشکالات، ویژگی‌های جدید و...

بدهی فنی/ Technical debt

technical debt وظیفه‌ایست که در آینده باید انجام شود زیرا امروز به دلایلی انجام نشده. مثلاً از نوشتن تست‌ها به دلیل کمبود وقت صرف نظر می‌کنیم تا به زمان تحویل پروژه برسیم.

به طور کلی technical debt زمانی اتفاق می‌افتد که به هر دلیلی تحویل کار از کیفیت آن مهم‌تر باشد.

می‌توان technical debt را به دو نوع کلی تقسیم کرد.

1- برنامه‌ریزی شده

این نوع از بدهی با برنامه‌ریزی تیم جهت رسیدن به اهداف مهم‌تر و همچنین در نظر گرفتن زمان مناسب برای پرداختن به آن‌ها صورت می‌گیرد. مثالی که در ابتدا زدم از همین نوع بود. تیم از نوشتن تست‌ها در مدتی مشخص صرف‌نظر می‌کند تا به deadline پروژه برسد و پس از آن زمانی را برای نوشتن تست‌ها در نظر می‌گیرد.

2- ناخواسته

هیچ برنامه ریزی برای آن وجود نداشته و به عنوان مثال تجربه کم علت بروز چنین اتفاقی شده.

technical debt هم خوب است و هم بد. فرض کنید تصمیم به خرید چیزی گرفته‌اید در حالی که به اندازه کافی پول برایش ندارید. فروشنده پیشنهاد خرید قسطی را می‌هد و قبول می‌کنید. خوب است زیرا شما صاحب آن شده‌اید و بد است چون تا چند سال آینده مشغول پرداخت قسط خواهید بود که شامل بازپرداخت اصل پول به همراه سود آن است. این اقساط قدرت خرید شما را در آینده محدود و پیشرفت‌تان را کند خواهد کرد.

refactoring برای مقابله با technical debt موثر است و به تأخیر انداختن آن یکی از مهم‌ترین دلایل به وجود آمدن بدهی فنی محسوب می‌شود.

فصل دوم- Code Smells

تعریف بوی بد کد/ Code Smell

code smellها نشانه‌هایی هستند که با مشکلات نرم‌افزار مرتبط‌اند و **ممکن** است به شما در یافتن مشکلات کمک کنند. محلی که قرار است از تکنیک‌های refactoring استفاده شود.

این اصطلاح اولین بار توسط Martin Fowler مطرح شد. اما دلیل این نام‌گذاری چیست؟ اول این‌که بوها قابل تشخیص هستند و بیشتر اوقات شما را به سمت مشکل خواهند برد و دوم، ممکن است گاهی نشانه‌ی مشکلی نباشند!

همچنین code smellها شاید در حال حاضر از دید فنی بدون ایراد به نظر برسند اما ممکن است در آینده باعث کند شدن توسعه و بروز مشکلات شوند.

برای درک بهتر smellها و داشتن یک تصویر کلی، آن‌ها به 5 گروه تقسیم شده‌اند. اینجا به توضیح مختصرشان می‌پردازیم و در ادامه با تکنیک‌های refactoring بازسازی/بازنویسی و اصلاح‌شان را انجام خواهیم داد.

1) Bloaters

آنقدر رشد کرده و بزرگ شده‌اند که نمی‌توان به درستی با آن‌ها کار کرد. مانند:

- Long Method
- Large Class
- Long Parameter List

- Primitive Obsession
- Data Clumps

1-1) Long Method

متدی شامل ده‌ها خط کد.

درک، عیب‌یابی و توسعه متدهای کوچک آسان‌تر است. در صورت امکان بهتر است متدهای طولانی را به چند متد کوچک‌تر تبدیل کرد.

2-1) Large Class

کلاس‌های بزرگ.

3-1) Long Parameter List

متدهایی با پارامترهای زیاد.

4-1) Primitive Obsession

استفاده بیش از حد مقادیر به جای اشیاء. مثلاً استفاده از ثابت‌ها، ADMIN_ROLE = 0 که نقش مدیر در کلاس User اشاره می‌کند.

5-1) Data Clumps

مجموعه‌ای از متغیرها که اغلب به هم وابسته‌اند و در بخش‌های مختلف نرم‌افزار استفاده یا تکرار می‌شوند. مانند تنظیمات پیکره‌بندی یا تاریخ شروع و پایان (rangeها). معمولاً اینطور نیست که فقط با تاریخ شروع یا پایان کار کنید بلکه اغلب هر دو را با هم لازم دارید. Data Clumpها می‌توانند با اشیا جایگزین شوند.

2) Object-Orientation Abusers

به طور کامل از امکانات و قابلیت‌های طراحی شی‌گرا استفاده نمی‌شود.

- Switch Statements

- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

1-2) Switch Statements

عملگرهای شرطی بزرگ، پیچیده، تودرتو یا متوالی. مانند switch با caseهای زیاد.

2-2) Temporary Field

اشیایی که شامل مقدار زیادی فیلد غیر ضروری یا اختیاری هستند که در شرایط خاصی از آن‌ها استفاده می‌شود و در بیشتر حالات بدون استفاده با مقدار پیش فرض مثل null باقی می‌مانند.

3-2) Refused Bequest

هنگامی که از قابلیت‌های(یعنی متدها و پراپرتی‌های) کلاس parent استفاده نمی‌شود یا تنها برخی از آن‌ها مورد استفاده قرار می‌گیرند.

4-2) Alternative Classes with Different Interfaces

کلاس‌های مشابه با Interface های متفاوت. کلاس‌هایی که عملکرد یکسانی دارند و کار مشابهی انجام می‌دهند اما شامل متدهای مختلف، متفاوت و تکراری هستند. این مورد هنگام ورود همکار جدید در صورتی که ارتباط بین اعضا ضعیف باشد زیاد اتفاق می‌افتد. اگر نداند کلاس‌های مورد نیازش موجود است دوباره آن را می‌نویسد.

3) Change Preventers

هنگام تغییر دادن ناچار هستید قسمت‌های زیادی را تغییر دهید.

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Divergent Change (1-3

زمان افزودن ویژگی جدید یا تغییر ناچار هستید تغییرات زیادی در کلاس/ ماژول انجام دهید.

Shotgun Surgery (2-3

تا حدودی به Divergent Change شباهت دارد.

Shotgun Surgery زمانی رخ می‌دهد که یک تغییر (اغلب کوچک) را در چند محل اعمال می‌کنید. Divergent Change به انجام تغییرات در یک کلاس اشاره می‌کرد.

هنگامی‌که یک تکه کد بارها تکرار می‌شود این اتفاق خواهد افتاد. مانند فرآیند پرداخت که در قسمت‌های مختلفی وجود دارد. در صورت تکرار این کد با آپدیت بعدی که ممکن است تغییر درگاه بانکی باشد ناچار هستید در چند قسمت آن را اعمال کنید.

پس از شلیک با اسلحه Shotgun، غلاف پلاستیکی گلوله باز شده و ساچمه‌های کوچک داخل آن پخش و به قسمت‌های زیادی برخورد می‌کنند. اگر تغییر/update را شلیک و انجام تغییرات را گلوله فرض کنیم دلیل این نام‌گذاری مشخص می‌شود.

Parallel Inheritance Hierarchies (3-3

هنگام اضافه کردن subclass نیاز دارید به سایر کلاس‌ها نیز subclass اضافه کنید.

رشته‌های دانشگاهی را در نظر بگیرید، هر رشته مدیر گروه خود را دارد. از نظر طراحی دو کلاس اصلی/parent داریم. **رشته** و **مدیر گروه**.

در این ساختار هر یک از رشته‌ها subclassهایی هستند که از کلاس اصلی خود ارث می‌برند. مثلاً نرم‌افزار کامپیوتر.

```
Class FieldOfStudy
Class ComputerSoftwareField extends FieldOfStudy
```

و همین‌طور برای مدیر گروه.

```
Class Chairperson
Class ComputerSoftwareChairperson extends Chairperson
```

Parallel Inheritance Hierarchies اینجا اتفاق می‌افتد. در هنگام اضافه کردن subclass جدید به رشته نیاز به اضافه کردن subclass برای مدیر گروه پیدا می‌کنید.

4) Dispensables

چیزهای بیهوده و اضافی که وجودشان باعث دردسر می‌شود. نکته مشترک بین تمام Dispensableها این است که همه آن‌ها نمایانگر موارد غیر ضروری هستند و باید حذف یا جایگزین شوند.

- Lazy class
- Data class
- Duplicate Code
- Dead Code
- Speculative Generality
- Comments

4-1) Lazy class

کلاسی که به اندازه کافی کار نمی‌کند. ممکن است پس از مدتی به علت توسعه، یک کلاس به حاشیه رفته باشد و یا این‌که با آینده‌نگری برنامه‌نویس ایجاد شده و فعلاً استفاده نمی‌شود. یا باید حذف شود و یا مسئولیت‌هایش افزایش یابد.

4-2) Data class

کلاسی که شامل تعدادی فیلد و متد getter/setter است و توسط سایر کلاس‌ها مورد استفاده قرار می‌گیرد. Data Class باعث وابستگی بیش از اندازه می‌شود زیرا شما با دیتایی کار می‌کنید که در کلاس دیگری وجود دارد.

Duplicate Code (3-4

کدهای تکراری و یا کدهایی که عملکرد یکسانی دارند.

Dead Code (4-4

کدی که دیگر از آن استفاده نمی‌شود.

Speculative Generality (5-4

آینده‌نگری باعث می‌شود مواردی به کد اضافه کنید که در حال حاضر مورد نیاز نیست. در صورتی که بر پایه حدس و گمان باشند ممکن است هرگز اجرا نشوند. درک و نگهداری این موارد اغلب دشوارتر از فایده وجود آن‌هاست.

Comments (6-4

توضیحات اضافی و زیاد. در نظر داشته باشید اول خوانایی بعد comment. ایده‌آل است با نگاه کردن به کد آن را درک کنیم.
از توضیحات اضافی در کد بپرهیزید. اگر کد شما نیاز به commentهای زیادی دارد ممکن است تمیز نوشته نشده باشد.

Couplers (5

وابستگی بین کلاس‌ها یا ماژول‌ها زیاد است.

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man

Feature Envy (1-5

Feature Envy مربوط به متدهاست. متدی Feature Envy دارد که از ویژگی‌های سایر کلاس‌ها (فیلد و متدهای آن‌ها) بیشتر از ویژگی‌های کلاس خود استفاده کند.

Inappropriate Intimacy (2-5

هنگامی که یک کلاس از فیلدها و متدهای داخلی سایر کلاس‌ها استفاده کند.

توجه کنید Feature Envy از ویژگی‌های عمومی/public سایر کلاس‌ها استفاده می‌کرد و مربوط به متدها بود در حالی که Inappropriate Intimacy از ویژگی‌هایی استفاده می‌کند که قرار نیست و نباید مستقیماً به آن‌ها دسترسی داشته باشد و مربوط به کلاس‌هاست. این code smell کپسوله‌سازی کلاس‌ها را به خطر می‌اندازد.

Message Chains (3-5

متدی از یک شئ، متد شئ دیگری را صدا می‌زند و همین‌طور تکرار می‌شود...

```
userID = app.getEnv().getAuth().getUser().getID()
```

زنجیره‌های طولانی قانون دمیتر را نقض می‌کنند و نشان می‌دهند که شما به ساختار چندین کلاس وابستگی دارید و تغییر این ساختار (زنجیره) باعث تغییر کد شما خواهد شد.

https://en.wikipedia.org/wiki/Law_of_Demeter

Middle Man (4-5

کلاسی که بجای مشارکت در برنامه، تنها نماینده کلاس دیگری ست. متدهای آن کلاس را اجرا و نتیجه را به کلاس اصلی برمی‌گرداند.

```
class Person
{
    public getManager()
     {
         return Department.getManager();
     }
}
```

فصل سوم- Composing Methods

Composing Methods

در اولین بخش از تکنیک‌های Refactoring به Composing Methods می‌پردازیم. Composing Methods مجموعه‌ای از تکنیک‌ها برای ساده‌سازی متدهای بیش از حد بزرگ و طولانی هستند.

Extract Method

تبدیل متدهای بزرگ و طولانی به چند متد کوچک‌تر. این کار باعث افزایش خوانایی، استفاده مجدد، حذف کد تکراری و انسجام بهتر کلاس و متدها خواهد شد. متدهای بزرگی که داخل بدنه خود چند وظیفه دارند را به چند متد تک وظیفه‌ای تبدیل کنید.

```
public function delete(int id)
{
    order = this.find(id)
    order.delete()
}
```

وظایف را از هم تفکیک کنید و سپس برای هر وظیفه یک متد جدید بنویسید.

```
public function delete(Order order)
{
```

```
    order.delete()
}

public function findByID(int id)
{
    order = this.find(id)
    return order
}
```

Inline Method

کاهش متدهای غیر ضروری.

آقای Kent Beck در کتاب Responsive Design presentation می‌گوید: تکنیک‌های refactoring دوطرفه است! هنگامی که یک تکنیک را انجام می‌دهیم ممکن است از سایر موارد دور شویم.

زیاده‌روی در انجام Extract Method منجر به ایجاد متدهای کوچک، ساده و غیر ضروری خواهد شد.

```
class Challenge
{
      public function join(User user)
      {
            nickname = this._nickname(user)
            // ...
      }

      private function _nickname(User user)
      {
            return user.name + " (member)"
      }
}
```

همانند متدهای طولانی، متدهای کوچک و ساده هم، آزاردهنده و گیج‌کننده هستند.

```
class Challenge
{
      public function join(User user)
```

```
        {
                nickname = user.name + " (member)"
                // ...
        }
}
```

Inline Temp

متغیرهای temp با مقدار یا expressionهای ساده را حذف کنید.

```
public function getToday()
{
    today = DateTime.today()
    return today
}
```

دلیلی برای تعریف یک متغیر با چنین مقداری وجود ندارد. مگر این‌که بارها استفاده شود.

```
public function getToday()
{
    return DateTime.today()
}
```

Extract Variable

expressionهای پیچیده را تا جای ممکن ساده کنید.

```
public function generateUniqueID()
{
        dateTime = DateTime.now()
        return "J" + dateTime.getYear() +
                    (dateTime.getMonth() <= 9 ? "0" +
                    dateTime.getMonth() :
dateTime.getMonth()) +
                    dateTime.getSecond()
}
```

برای درک بهتر می‌توان با استفاده از چند متغیر جدید و شکستن expression به قسمت‌های مجزا آن را ساده کرد.

```
public function generateUniqueID()
{
      dateTime = DateTime.now()
      month = dateTime.getMonth() <= 9 ? "0" +
dateTime.getMonth() : dateTime.getMonth()
      return "J" + dateTime.getYear() + month +
dateTime.getSecond()
}
```

Replace Temp with Query

به جای نگهداری نتیجه expression در متغیرهای موقتی، از متد استفاده کنید.

```
public function calculatePrice()
{
    basePrice = this.quantity * this.itemPrice

    if (basePrice > 1000) {
        return basePrice * 0.95
    }
    return basePrice * 0.98
}
```

استفاده از متد باعث جلوگیری از تکرار کد و به‌روزرسانی آسان یک عبارت در تمام برنامه خواهد شد. کافیست بدنه متد به‌روز شود.

```
public function calculatePrice()
{
    basePrice = this.basePrice()
    if (basePrice > 1000)
    {
        return basePrice * 0.95
    }

    return basePrice * 0.98
```

```
}

public function basePrice() {
    return this.quantity * this.itemPrice
}
```

Split Temporary Variable

از یک متغیر موقت چند بار برای نگهداری مقادیر مختلف استفاده نکنید.

```
public function accounting()
{
      temp = this.revenue - this.costs
      print("Profit = " + temp)

      temp = (this.costs / this.revenue) * 100
      print("Margin = " + temp + "%")
}
```

در عوض برای عبارات و مقادیر مختلف از متغیرهای مختلف استفاده کنید به طوری که هر متغیر تنها مسئول نگهداری یک مقدار باشد. این کار تغییر و نگهداری کد را آسان می‌کند و اثر یک به‌روزسانی در قسمت‌های دیگر احساس نخواهد شد.

```
public function accounting()
{
      profit = this.revenue - this.costs
      print("Profit = " + profit)

      margin = (this.costs / this.revenue) * 100
      print("Margin = " + margin + "%")
}
```

Remove Assignments to Parameters

از تغییر پارامترهای متد داخل بدنه خودش پرهیز کنید.

```
public function getValue(number)
{
	if(number > 1000)
	{
		number = number * 10
	}

	return number
}
```

متغیرهای موقتی برای نگهداری و تغییر مقدار پارامترها مناسب هستند.

در مواقعی که پارامترهای متد به صورت by reference پاس داده شده‌اند، تغییرشان باعث به وجود آمدن اثرات جانبی در سایر بخش‌های دیگر می‌شود. حتی اگر پارامتر ها by reference نیستند هر عنصر در کد شما مانند کلاس، متد و یا جزئی‌تر، حتی یک متغیر باید مسئول انجام یک عمل باشد. در غیر این صورت درک و تشخیص این‌که عناصر ما در شرایط مختلف دقیقاً چه کاری انجام می‌دهند دشوار خواهد شد.

```
public function getValue(number)
{
	result = number
	if(number > 1000)
	{
		result = number * 10
	}

	return result
}
```

Replace Method with Method Object

یک متد بسیار بزرگ و غیر قابل درک شامل انبوهی از متغیرهای محلی را در نظر بگیرید.

```
class Exam
{
	// ...
	public function score()
	{
		// long body code ...
	}
}
```

کاری که باید انجام داد از نام این تکنیک پیداست. تبدیل آن متد به یک کلاس جداگانه.

Extract Method در صورتی که بتوان وظایف یک متد را تفکیک نمود قابل انجام است. زمانی از تکنیک Replace Method with Method Object استفاده می‌شود که آنقدر متغیرهای محلی زیاد، مرتبط و درهم تنیده‌ای داریم که امکان شکستن متد به متدهای کوچک‌تر وجود نداشته یا با این عمل باعث به وجود آوردن یک کلاس بزرگ و پیچیده خواهیم شد. در نظر داشته باشید Large Class و Long Method هر دو Code Smell هستند.

می‌توان پس از ایجاد کلاس جدید Extract Method را در صورتی که نیاز باشد گام بعدی در نظر گرفت.

```
class Exam
{
	// …

	public function score()
	{
		return new Scorer(this).execute()
	}
}

class Scorer
{
	public function Scorer(Exam exam)
	{
		// ...
```

```
        }

        public function execute()
        {
                // long body code ...
        }
}
```

Substitute Algorithm

جایگزین کردن الگوریتم‌های قدیمی. این کار معمولاً به دلیل ناکارامدی الگوریتم‌های قدیمی، پیچیده و غیر قابل فهم بودن، ناسازگاری با عناصر جدید، به‌روز شدن کتابخانه‌ها و... صورت می‌گیرد.

به طور کلی هدف از بازنویسی الگوریتم‌ها بهبود آن‌هاست. از نظر کارایی، وضوح و...

فصل چهارم- Moving Features between Objects

Moving Features between Objects

یکی از بخش‌های مهم Refactoring جابجایی ویژگی‌ها بین اشیاست. بررسی و انتقال آن‌ها به محل مناسب.

Move Method

متدی که در سایر کلاس‌ها بیشتر از کلاس خودش استفاده شود انسجام کلاس‌ها را کاهش و وابستگی بین‌شان را افزایش می‌دهد.

```
class Account
{
      public getOverdraft()
      {
            if(AccountType.IsPremium)
            {
                  return this.daysOverdrawn * 6
            }

            return this.daysOverdrawn * 8
      }
}

class AccountType
{
```

```
        public IsPremium()
}
```

Overdraft یکی از تسهیلات بانکی است. بدین صورت که منفی شدن حساب مشتری تا اندازه‌ای که توافق می‌کنند امکان‌پذیر خواهد شد. مثلاً توافق بین بانک و مشتری 100 دلار است. مشتری می‌تواند با موجودی 50 دلار هم قبض 150 دلاریش را پرداخت نمایید. در عوض به ازای هر روز منفی ماندن حساب باید مبلغی را بعدا به بانک بدهد. در مثال بالا این مبلغ را برای کاربران ویژه روزی 6 دلار و سایر کاربران 8 دلار در نظر گرفته‌ایم.

متد getOverdraft به کلاس AccountType منتقل می‌شود زیرا این کلاس شامل دیتایی است که getOverdraft از آن استفاده می‌کند.

```
class Account
{
        public getOverdraft()
        {
                return AccountType.getOverdraft(this)
        }
}

class AccountType
{
        public IsPremium()

        public getOverdraft(Account account)
        {
                if(this.IsPremium)
                {
                        return account.daysOverdrawn * 6
                }

                        return account.daysOverdrawn * 8
        }
}
```

Move Field

همانند Move Method اما در مورد فیلدهاست. فیلدهایی که در کلاس‌های دیگر بیشتر از کلاس خودش استفاده دارند.

در مرتبط‌ترین کلاسی قرارشان دهید که بیشترین استفاده از آن‌ها شود.

Extract Class

از ابتدایی‌ترین قوانین مربوط به کلاس این است که باید تنها یک هدف یا مسئولیت داشته باشد. تکنیک Extract Class راه‌حلی است برای کلاس‌هایی که چند مسئولیت دارند یا این‌که وظایف‌شان مبهم است.

```
class Comment
{
	public store(request)
	{
		data = this.htmlPurifier(request)
		this.save(data)
	}

	public htmlPurifier(html)
	{
			// ...
		return purifiedText
	}
}
```

به مرور کلاس‌های چند مسئولیتی بزرگ و پیچیده می‌شوند. در مثال بالا کلاس Comment قبل از ذخیره نظر جدید اقدام به تمیز کردن و حذف تگ‌های html متن ارسال شده می‌کند(دو مسئولیت/وظیفه).

متدهای کلاس را بر اساس وظیفه/ مسئولیت گروه‌بندی و به ازای هر گروه کلاس جدیدی ایجاد کنید.

```
class Comment
{
	public store(request)
```

```
        {
                data = HtmlPurifier(request)
                this.save(data)
        }
}

class HtmlPurifier
{
        public htmlPurifier(html)
        {
                        // ...
                return purifiedText
        }
}
```

Inline Class

در قسمت code smellها تعریفی به نام Lazy class داشتیم.

Lazy class، کلاسی بود که به اندازه کافی کار نمی‌کرد. ممکن است پس از مدتی به علت توسعه، یک کلاس به حاشیه رفته باشد و یا این‌که با آینده‌نگری برنامه‌نویس ایجاد شده و فعلاً استفاده نمی‌شود.

Inline Class، کلاس‌های اضافی هستند که مسئولیت مجزایی ندارند و قابل پیوستن به سایر کلاس‌ها می‌باشند. ادغامشان کنید یا مسئولیت‌های بیشتری به آن‌ها بدهید.

```
Class Person
{
        public name {get/set}
        public TelephoneNumber telephoneNumber

        public Person()
        {
                telephoneNumber = new TelephoneNumber
        }

        public phoneNumber
        {
```

```
                    this.telephoneNumber.phoneNumber
        }
}

Class TelephoneNumber
{
    public phoneNumber {get / set}
}
```

کلاس TelephoneNumber مسئولیت جدیدی ندارد که بتوان آن را در گروه دیگری جای داد. به همین دلیل متدهایش را به کلاس Person منتقل و سپس TelephoneNumber را حذف می‌کنیم.

Hide Delegate

هنگامی که یک متد به صورت متوالی متدی از یک کلاس دیگر را صدا می‌زند ممکن است منجر به ایجاد زنجیره‌ای از ارتباطات ضعیف و شکننده شود. در این مدل ارتباطات یک تغییر در بسیاری از قسمت‌ها ثاتیر می‌گذارد.

```
manager = Employee.getDepartment().getManager()
```

وارد جزئیات مثال بالا شویم.

```
class Employee
{
    Department department

    public Employee(Department department)
    {
            this.department = department
    }

    public Department getDepartment()
    {
            return department
    }
}
```

```
class Department
{
    Employee manager

    public Department(Employee manager)
    {
            this.manager = manager
    }

    Employee getManager()
    {
            return manager
    }
}

// use
john = new Employee
manager = john.getDepartment().getManager()
```

در یکی از بندهای قانون دمیتر یا قاعده حداقل دانش این‌طور نوشته شده:
هر واحد تنها با دوست نزدیک خود حرف بزند و از غریبه‌ها دوری کند!

دمیتر می‌گوید هر واحد (بخش، ماژول یا...) حداقل ارتباط با سایر واحدها را داشته باشد.

زنجیره‌ای از متدها به خودی خود چیز بدی نیست و ما به وفور آن را مشاهده کرده‌ایم. اما به شرطی که وابستگی بین آن‌ها کنترل شده باشد.

مشکل اساسی مثال بالا این است که جزئیات کلاس Department بیش از حد قابل مشاهده است. **هر چه اشیا اطلاعات کمتری در مورد جزئیات هم داشته باشند، با تاثیر کمتری بر دیگران قابل تغییر خواهند بود.**

ابتدای برنامه‌نویسی شئ‌گرا می‌آموزیم کپسوله‌سازی یعنی مخفی نگاه داشتن و کنترل دسترسی فیلدها. اما در این کتاب خواهیم دید کپسوله‌سازی چیزی بیشتر از آن است.

دو تعریف:

- Delegate: نماینده. متدهایی که اشاره به متد دیگری دارند. آن را صدا زده و خروجی را پاس می‌دهند. این کار می‌تواند به دلیل کپسوله کردن انجام شود. delegate ها جزئیات کلاس را مخفی نگاه می‌دارند.

- server: از delegateها استفاده می‌کنند و در مورد ساختار کلاس‌ها اطلاعی ندارند.

در مثال بالا، کلاس Department از متدی در کلاس Employee گرفته شده و سپس متدی در کلاس Department صدا می‌زند!

با وجود پیاده سازی server و delegate به منظور کاهش جزئیات، همچنان زمان استفاده ساختار کلاس‌ها بیش از اندازه دیده می‌شود و در صورت تغییر delegate ناچارید تمام قسمت‌هایی که از این زنجیره استفاده شده است را ویرایش کنید.

```
// use
john = new Employee
manager = john.getDepartment().getManager()
```

برای حل این مشکل delegate را مخفی کنید. داخل server برای delegate یک متد delegate دیگر بسازید.

```
class Employee
{
	Department department;

	public Employee(Department department)
	{
		this.department = department
	}

	public getManager()
	{
		return this.department.manager;
	}
}
```

```
class Department
{
      Employee manager

      public Department(Employee manager)
      {
            this.manager = manager;
      }
}

// use
john = new Employee
manager = john.getManager()
```

Remove Middle Man

در فصل code smellها گفته شد Middle Man کلاسی است که جای مشارکت در برنامه، تنها نماینده کلاس دیگری شده. یعنی فقط متدهای کلاس دیگر را صدا و نتیجه را به کلاس اصلی برمی‌گرداند.

تکنیک Hide Delegate و مثال آن را به خاطر بیاورید. ممکن است متوجه شوید کلاس server تنها متدهای delegate را صدا می‌زند. و شما به ازای ایجاد یک متد جدید در delegate، متدی هم در server می‌سازید.

```
class Person
{
      public Department department
      public getManager()
      {
            return this.department.getManager()
      }
}

class Department
{
      private Person _manager;
      public department(Person manager)
      {
```

```
            this._manager = manager
      }

      public getManager()
      {
            return this._manager
      }
}

 // use
person = new Person();
manager = person.getManager();
```

در این صورت مستقیماً از خود delegate استفاده کنید.

```
class Person
{
      public department
      public getDepartment()
      {
            return this.department
      }
}

class Department
{
      private this._manager
      public department(manager)
      {
            this._manager = manager
      }

      public getManager()
      {
            return this._manager
      }
 }
```

```
// use
person = new Person()
manager = person.department.getManager()
```

Introduce Foreign Method

مناسب‌ترین کلاس برای تعریف یک متد جدید، کلاسی است که در آن فیلدهای مورد نیاز متد وجود داشته باشد. این کار باعث کاهش وابستگی خواهد شد زیرا دیگر نیاز به استفاده از فیلد و متد سایر کلاس‌ها ندارید.

اما همیشه این کار قابل انجام نیست، به عنوان مثال اضافه کردن متد به کتابخانه‌ها و کلاس‌های کمکی وجود ندارد یا این‌که نباید چنین کاری را انجام داد.

```
class Report
{
	public sendReport()
	{
		nextDay = new Date(previousEnd.getYear(),
previousEnd.getMonth(), previousEnd.getDate() + 1)
		// ...
	}
}
```

در مثال بالا هر چند با استفاده از پارامترهای کتابخانه Date به روز بعد رسیدیم اما این روش درست نیست زیرا ممکن است در بخش‌های مختلف برنامه به آن نیاز داشته باشیم و تکرار شود.

حال آنکه نمی‌توان متد جدید را داخل کتابخانه Date افزود، در کلاس کلاینت خود یک متد با پارامتری از جنس کلاس Date تعریف و استفاده کنید.

```
class Report
{
	public sendReport()
	{
		Date newStart = this._nextDay(previousEnd)
		// ...
	}
```

```
	private _nextDay(Date arg)
	{
		return new Date(arg.getYear(), arg.getMonth(),
arg.getDate() + 1)
	}
}
```

Introduce Local Extension

یک کلاس کمکی مانند کتابخانه Date متدهای مورد نیازتان را شامل نمی‌شود و شما نمی‌توانید آن‌ها را به کلاس اصلی اضافه کنید. تفاوتش با Introduce Foreign Method در تعداد متدهاست. در صورتی که متدهای مورد نیاز شما افزایش یافت یا این‌که در سایر کلاس‌ها مورد نیاز بود مانند روش قبلی از اضافه کردن به کلاینت خودداری کنید و به جای آن یک subclass بسازید.

```
class MyDate extends Date
{
	public function nextDay()
	{
		//...
	}

	public function addDays(int day)
	{
		//...
	}

	public function addWeeks(int week)
	{
		//...
	}

	public function startOfDay()
	{
		//...
	}
```

```
        public function endOfDay()
        {
                //...
        }
    //...
}
```

فصل پنجم- Organizing Data

Organizing Data

در این فصل به سازماندهی بهتر داده‌ها می‌پردازیم. کنترل دسترسی، جایگزینی مقادیر با اشیاء و... .

Self Encapsulate Field

از فیلدهای خصوصی کلاس به طور مستقیم استفاده نکنید.
بدین ترتیب دسترسی به آن‌ها محدود و قابل کنترل خواهد شد و علاوه بر این‌که می‌توان مشخص کرد کدام فیلدها بروزسانی شوند یا قابل دسترس باشند، امکان اجرای یک رفتار یا تایید مقدار و... نیز را خواهید داشت.
کپسوله‌سازی داده‌ها یک اصل مهم در برنامه‌نویسی شی‌گراست.

```
class Charge
{
      private _unit
      private _rate

      public Charge(unit, rate)
      {
            this._units = unit
            this._rate = rate
      }
```

```
        public getAmount()
        {
                return this._unit * this._rate
        }
}
```

به جای دسترسی مستقیم به فیلد از متدهای getter/setter استفاده کنید.

```
class Charge
{
        private _ unit
        private _rate

        public Charge(unit, rate)
        {
                this._units = units
                this._rate = rate
        }

        public getUnits() { return this._unit }
        public getRate() { return this._rate }

        public getAmount()
        {
                return getUnits() * getRate()
        }
}
```

Encapsulate Field

مانند Self Encapsulate Field است اما برای فیلدهای عمومی.

```
class Person
{
        public name
}
```

برای کنترل دسترسی ابتدا فیلد را به حالت خصوصی تغییر و از متدهای getter / setter استفاده کنید.

```
class Person
{
	private _name

	public getName()
	{
		return this._name
	}

	public setName(arg)
	{
		this._name = arg
	}
}
```

Encapsulate Collection

مجموعه‌ها (مثلاً یک آرایه) کمی متفاوت‌تر از فیلدهای معمولی کپسوله‌سازی می‌شوند. نکته مهم این است که نباید دسترسی مستقیم به ساختار مجموعه و اجازه تغییر کامل آن را بدهید.

بنابراین به جای یک متد که تمام مجموعه را set می‌کند، دو متد add و remove برای اضافه و حذف آیتم از مجموعه تعریف می‌کنیم.

```
class Order
{
	private _orders
	public function get()
	{
		return this._orders
	}

	public function addOrder(Order order)
	{
		this._orders.add(order)
```

```
        }

        public function removeOrder(Order order)
        {
                this._orders.find(order).remove()
        }
}
```

Replace Data Value with Object

گاهی یک فیلد بین کلاس‌های مختلف، مشترک و مورد نیاز است که خود می‌تواند شامل رفتار یا جزئیات بیشتری باشد یا این‌که به مرور چنین اتفاقی خواهد افتاد. مثلاً متغیر نام مشتری/customer.

```
class Order
{
        public customer

        public Order(customerName)
        {
                this.customer = customerName
        }

        public count(orders, customer)
        {
                return orders.count("customer_name", = ,
this.customer)
        }
}
```

برای جلوگیری از تکرار کد مانند کاری که در Extract Class انجام دادیم این فیلد را تبدیل به یک شئ مجزا می‌کنیم.

```
class Order
{
        public count(orders, Customer customer)
        {
```

```
            return orders.count(”customer_name”, = ,
customer.name)
      }
}

class Customer
{
      public name
      public Customer(name)
      {
            this.name = name
      }
}
```

Change Value to Reference

ممکن است در طول برنامه از یک شئ تعداد زیادی ساخته شود.

```
class Order
{
      public Customer customer

      public Order(customerName)
      {
            this.customer = new Customer(customerName)
      }

      public count(orders)
      {
            return orders.count("customer_name", = ,
this.customer.name)
      }
}

class Customer
{
      public name
```

```
        public Customer(name)
        {
                this.name = name
        }
}
```

در مثال بالا اگر نام مشتری تکراری باشد چند شئ تکراری هم ساخته می‌شود. این یک حالت بود و ممکن است حالات دیگری نیز اتفاق بیفتد که این مشکل ایجاد گردد. مشخصاً باید از تکرار کلاس‌های مشابه جلوگیری کرد و جای آن یک کلاس مرجع داشت. به عنوان راه‌حل می‌توان خود کلاس Customer را در متد مورد نظر پاس داد.

```
class Order
{
        public count(orders, Customer customer)
        {
                return orders.count("customer_name", = ,
customer.name)
        }
}
```

Change Reference to Value

در مورد قبلی(Change Value to Reference) کلاس Customer یک چرخه حیاط طولانی داشت. احتمالا تا زمانی که کار با سفارشات تمام شود به Customer نیاز داشتیم. Change Reference to Value عکس آن است. گاهی کلاس‌های ما خیلی کم به‌روزسانی می‌شوند و نیازی به وجود آن‌ها تا انتهای برنامه نیست. در این صورت ایجاد یک نمونه از آن، استفاده و سپس رهاسازی کفایت خواهد کرد.

Replace Array with Object

آرایه‌هایی شامل مقادیر ناهمگن(غیر مشابه) خوانایی کافی را ندارد.

```
String[] row = new String[3]
row[0] = "refactoring"
```

```
row[1] = "refactoring@mail.com"
row[2] = "157298"
```

همچنین ممکن است مانند آخرین عنصر مجبور شوید عدد را به عنوان رشته تعریف کنید در صورتی که تایپ آن مناسب نیست.

در چنین حالتی آرایه را با یک شئ جایگزین و برای هر عنصر یک فیلد در نظر بگیرید.

```
Performance row = new Performance()
row.setName("refactoring")
row.setEmail("refactoring@mail.com")
row.setID(157298)
```

Duplicate Observed Data

در سیستمی که شامل رابط کاربری است مخصوصاً زمانی که چند رابط کاربری برای دیتای شما ایجاد می‌شود مانند اپلیکیشن وب و موبایل، به منظور جلوگیری از تکرار کد منطق برنامه باید از رابط کاربری جدا باشد.

نمایش تعداد نتایج جستجو را در نظر بگیرید.

```
<h4>About { count(results) } results</h4>
<div>
      <!-- results -->
<div>
```

منطق محاسبه تعداد نتایج را به کلاسی که وظیفه نمایش دیتای شما را دارد ببرید.

```
<h4>About { results.count() } results</h4>
```

Change Unidirectional Association to Bidirectional

گاهی دو کلاس به دلیل استفاده از ویژگی‌های یکدیگر نیاز به رابطه معکوس خواهند داشت.

```
class User
{
      public Profile profile
      public function profile()
```

```
        {
                // ... get and return profile class
        }
}

class Profile
{
        public name
        public phone
        public address
        // ...
}
```

می‌توانیم به Profile از طرف User دسترسی داشته باشیم اما به صورت عکس چنین کاری قابل انجام نیست. برای ایجاد یک رابطه معکوس در کلاس Profile متد جدیدی به منظور دسترسی به User تعریف کنید.

```
class Profile
{
        // …
        public function user()
        {
                // ... get and return user class
        }
}
```

Change Bidirectional Association to Unidirectional

برعکس مورد قبلی است و تأکید دارد اگر بین دو شئ رابطه دوطرفه یا معکوس دارید اما از آن استفاده نمی‌کنید، باید حذف شود.

Replace Magic Number with Symbolic Constant

اعداد خاصی که در برنامه استفاده می‌شوند را به صورت ثابت تعریف کنید.

```
public function area(radius)
{
```

```
        return 3.14 * radius * radius
}
```

با قرار گرفتن این اعداد داخل برنامه خوانایی کاهش پیدا می‌کند، پس بهتر است به صورت زیر نوشته شود.

```
const PI = 3.14
public function area(radius)
{
        return PI * radius * radius
}
```

Replace Type Code with Class

فیلدهایی که برای تایپ، نوع ، وضعیت و... داخل کلاس تعریف شده‌اند را به کلاس جدیدی منتقل کنید.

```
class User
{
        const ENABLE_STATUS = 1
        const DISABLE_STATUS = 2
        const BLOCKED_STATUS = 3

        // ...

        public function forgotPassword()
        {
                if(this.status == ENABLE_STATUS)
                {
                        sendLink()
                }
        }
        // ...
}
```

این کار باعث کنترل بهتر مقادیر نامعتبر و کاهش خطا و کد خواهد شد.

```
class User
{
	// ...
	public UserStatus status { get / set }
	public function User(UserStatus userStatus)
	{
		this.status = userStatus
	}

	public function forgotPassword()
	{
		if(this.status == status.enable.code)
		{
			sendLink()
		}
	}
	// ...
}

class UserStatus
{
	public static UserStatus enable = new UserStatus(1)
	public static UserStatus disable = new UserStatus(2)
	public static UserStatus blocked = new UserStatus(3)

	public int code { get / set }

	public UserStatus(int statusCode)
	{
		this.code = statusCode
	}
}
```

Replace Type Code with Subclasses

هر کدام از فیلدهایی که برای تایپ، نوع ، وضعیت و... داخل کلاس تعریف شده و **باعث تغییر در رفتار برنامه می‌شوند** را همراه با رفتارشان به subclass منتقل کنید. این همان تکنیک قبلی با یک شرط اضافی است.

```
public class Employee
{
	//...
	const ENGINEER = 0
	const SALESMAN = 1
	const MANAGER = 2

	private int type

	public int type (int type) { get }

	public getInformation(int type)
	{
		switch (type)
		{
				case ENGINEER
					return room +"\n" + phone +
"\n" + hours.toString()
				case SALESMAN
					return phone
				case MANAGER
					return room + "\n" +
phone.toString()
			}
	}
}
```

خروجی getInformation بر اساس نوع تغییر می‌کند. برای هر نوع یک subclass بسازید. با این کار دو اصل Single Responsibility Principle و Open/Closed رعایت می‌شود.

```
abstract class Employee
{
	//...
	const ENGINEER = 0
	const SALESMAN = 1
	const MANAGER = 2
```

```
      private int type

      public int type (int type) { get }

      public getInformation(int type)

       //...
}

class Engineer extends Employee
{
      public function type()
      {
            return ENGINEER
      }

      public getInformation()
      {
            return room +"\n" + phone + "\n" +
hours.toString()
      }
}

class Salesman extends Employee
{
      public function type()
      {
                  return SALESMAN
      }

      public string getInformation()
      {
            return phone
      }
}

class Manager extends Employee
{
      public function type()
```

```
        {
                        return SALESMAN
        }

        public string getInformation()
        {
                return room + "\n" + phone.toString()
        }
}

// use
Employee  manager = new  manager()
print(manager.getInformation())
```

Replace Subclass with Fields

اگر subclass ها فقط در مقادیری که بازمی‌گردانند متفاوت هستند، آن‌ها را با فیلدها جایگزین کنید.

```
abstract class Person
{
        abstract function isMale()
        abstract function getCode()
}

class Male extends Person
{
        function getCode()
        {
                        return "M"
        }
}

class Female extends Person
{
        function getCode()
        {
                return "F"
        }
}
```

در این حالت ایجاد زیرکلاس کار اضافی است.

```
class Person
{
	private isMale
	private code

	static function createMale()
	{
		return new Person(true, "M");
	}

	static function createFemale()
	{
		return new Person(false, "F");
	}

	protected function __construct(isMale, code)
	{
		this.isMale = isMale;
		this.code = code;
	}

	function isMale()
	{
			return this.isMale
	}

	function getCode()
	{
			return this.code
	}
}
```

فصل ششم- Simplifying Conditional Expressions

Simplifying Conditional Expressions

در این قسمت به ساده‌سازی و refactor عبارات شرطی خواهیم پرداخت.

Decompose Conditional

شرط‌های پیچیده را تجزیه و ساده کنید.

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

با ساده‌سازی یک تکه کد درک آن را برای خود و سایرین آسان می‌کنید. عبارات پیچیده را داخل متد تعریف و تا جای ممکن آن‌ها را ساده‌تر بنویسید. حتی اگر امکان ساده کردن نبود، متدی به نام summer یا Summer is به مراتب قابل درک‌تر از قراردادن خود عبارت داخل شرط است.

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

Consolidate Conditional Expression

شرط‌هایی که نتیجه مشابهی دارند را در یک متد تعریف کنید.

```
function disabilityAmount()
{
      if (anEmployee.seniority < 2) return 0
      if (anEmployee.monthsDisabled > 12) return 0
      if (anEmployee.isPartTime) return 0
}
```

بدین ترتیب علاوه بر افزایش خوانایی، کدهای اضافی نیز حذف می‌گردند.

```
function disabilityAmount()
{
      if (isNotEligableForDisability()) return 0
}

function isNotEligableForDisability()
{
    return ((anEmployee.seniority < 2)
        || (anEmployee.monthsDisabled > 12)
        || (anEmployee.isPartTime))
}
```

Consolidate Duplicate Conditional Fragments

کدهای مشترک داخل بلوک‌های شرطی را به خارج از آن‌ها منتقل کنید.

```
if (isSpecialDeal())
{
      total = price * 0.95
      send()
}
else
{
      total = price * 0.98;
      send()
}
```

.refactor

```
if (isSpecialDeal())
{
      total = price * 0.95
}
else
{
    total = price * 0.98
}
send()
```

Remove Control Flag

استفاده از متغیرهای flag توصیه نمی‌شود، آن‌ها را حذف کنید.

```
while (i < itemsCount && !found)
{
      if (…)
      {
            found = true;
      }
}

if(found)
{
      print("item found!")
}
```

همیشه جایگزین بهتری به جای استفاده از flag ها وجود دارد. داخل عبارات شرطی می‌توانید از break, continue یا return استفاده کنید.

```
while (i < itemsCount)
{
      if (…)
      {
            print("item found!")
            break
      }
}
```

Replace Nested Conditional with Guard Clauses

شرط‌های تو در تو را جهت درک بهتر جریان کد، ساده کنید.

```
if (number >= 0)
{
      if (number == 0)
      {
            return "the number is zero"
      }
      else
      {
                  return "positive number"
      }
}
else
{
        return "negative number"
}
```

حالت ایده‌آل عبارات شرطی، تعدادی عبارت هم سطح است.

```
if (number < 0)
{
    return "negative number"
}

if (number == 0)
{
    return "the number is zero"
}

return "positive number"
```

Replace Conditional with Polymorphism

اگر در حالت‌های مختلف شرط، رفتار برنامه تغییر می‌کند به جای آن از subclass ها استفاده کنید.

```
class Employee
{
      public getPaymentAmount(EmployeeType type)
      {
            switch(type)
            {
                  case ENGINEER
                              return monltySalary
                  case SALESMAN
                              return monltySalary +
commission
                  case MANAGER
                              return monltySalary + bonus
            }
      }
}
```

با این کار دو اصل Tell-Don't-Ask و Open/Closed رعایت می‌شود.

```
abstract public Employee
{
      // ...
      getPaymentAmount()
}

class Engineer extends Employee
{
      // ...
      public getPaymentAmount()
      {
            return monthlySalary
      }
}

class Salesman extends Employee
{
      // ..
      public getPaymentAmount()
```

```
        {
                return monthlySalary + commission
        }
}

class Manager extends Employee
{
        // ..
        public getPaymentAmount()
        {
                return monthlySalary + bonus
        }
}
```

Introduce Null Object

گاهی مقدار برگشتی از متدها null است و نیاز داریم این مقدار را برای ادامه برنامه بررسی کنیم.

```
Customer customer = new Customer()
if(customer.getByID(1) !== null)
{
        name = customer.getName()
}
```

این تکنیک بیشتر به دیزاین‌پترن مرتبط است و به عنوان یک الگو در نظر گرفته می‌شود.

به جای استفاده از شرط‌ها یک کلاس null object ایجاد کنید. این کار باعث حذف کدهای تکراری و شرط‌ها می‌شود.

```
class Customer
{
        public function getByID(int id)
        {
                Customer customer = this.find(id)
                if(customer !== null)
                        {
                                return customer
```

```
            }

            return new NullCustomer()

      }

      public function getName()
      {
            return this.name
      }
}

class NullCustomer extends Customer
{
      boolean isNull()
      {
            return true
      }

      public function getName()
      {
            return "Guest"
      }
      // ...
}

// use
Customer customer = new Customer()
name = customer.getName()
```

Introduce Assertion

برای ادامه برنامه، یک شرط حتما باید true باشد در غیر این صورت منجر به نتیجه نامعقولی خواهد شد. در این حالت بهتر است ادامه برنامه متوقف گردد.

```
public selectionSort(Collection collection)
{
      // sort  collection
}
```

البته نه در همه جا. اگر کد با حالت اشتباه کار می‌کند، Assertion مفید خواهد بود.

```
public selectionSort(Collection collection)
{
      assert(collection.isEmpty() = true)
      // sort  collection
}
```

برای گزارش دقیق و توضیحات بیشتر خطا می‌توانید از Exception استفاده کنید.

فصل هفتم- Simplifying Method Calls

Simplifying Method Calls

این فصل مربوط به ساده کردن فراخوانی متدها، پارامترها، انتقال اشیاء بین آن‌ها و ارتباط بین کلاس‌ها می‌باشد.

Rename Method

نام یک متد باید بدنه خود را شرح دهد. این‌که به چه منظوری ایجاد شده و چه کاری انجام خواهد داد. از نام‌های کلی، عمومی و مختصر پرهیز کنید.

```
function cntBigWrds(data, length)
{
	i = 0
	foreach (data as iter)
	{
		if (length(iter) > length)
		{
			i++
		}
	}
	return i
}
```

این تکنیک را در همه جا استفاده کنید، نام متد، متغیر، کلاس، پکیج، فایل و...

در مرحله اول از کامنت برای حل این ابهام نباید استفاده کرد. مناسب است برنامه‌نویس با نگاه کردن به راحتی یک تکه کد را درک بنماید.

```
function countWordsLongerThan(words, minimumLength)
{
      count = 0
      foreach (words as word)
      {
            if (length(word) > minimumLength)
                  {
                  count++
            }
      }
      return count
}
```

Remove Parameter

پارامترهایی که در بدنه متد استفاده نمی‌شوند و یا این‌که به دلیل آینده‌نگری توسعه‌دهنده به وجود آمده‌اند را حذف کنید.

Separate Query from Modifier

متدی که نتیجه یک query را برمی‌گرداند و سپس چیزی داخل شئ را تغییر می‌دهد، به دو متد مجزا تقسیم کنید. یکی برای query و دیگری برای تغییر شئ.

```
public function delete(int id)
{
    order = this.find(id)
    order.softDelete()
}
```

بدین صورت اصل Command Query Responsibility Segregation یا به طور خلاصه CQRS رعایت می‌شود و همچنین می‌توان از متد query بارها استفاده کرد.

```
public function delete(Order order)
{
    order.softDelete ()
}

public function findByID(int id)
{
    order = this.find(id)
    return order
}
```

Parameterize Method

متدهایی که عمل مشابهی انجام می‌دهند اما در مقادیر بدنه خود با هم متفاوت‌اند را ادغام کنید.

```
date = new Date
date.addOneWeek()
date.addTwoWeek()
date.addThreeWeek()
date.addFourWeek()
```

متدهای تکراری یعنی کد تکراری.

```
date = new Date
date.addWeeks(1)
date.addWeeks(2)
date.addWeeks(3)
date.addWeeks(4)
```

Replace Parameter with Explicit Methods

اگر متد بر اساس مقدار پارامترها کار متفاوتی انجام می‌دهد، به ازای هر مقدار یک متد جدید تعریف کنید.

```
function setValue(name, value)
{
      if (name == "height")
```

```
        {
                this.height = value
            return
        }
        if (name == "width")
        {
                this.width = value
            return
        }

        assert("Should never reach here");
}
```

این متد مبهم و بزرگ است یا خواهد شد و بی‌دلیل با هر بار فراخوانی شرط‌های اضافی در بدنه آن بررسی می‌شود.

```
function setHeight(arg)
{
        this.height = arg
}

function setWidth(arg)
{
        this.width = arg
}
```

Preserve Whole Object

به جای ارسال مقادیر شی به عنوان فهرستی از پارامترها، خود شئ را به متد پاس بدهید.

```
x = circle.getX()
y = circle.getY()
radius = circle.getRadius()
makeCircle(x, y, radius)
```

بدین ترتیب اگر مقادیر بیشتری از شئ در بدنه متد نیاز شود، احتیاجی به تغییر متد در جاهایی که استفاده شده نیست و فقط بدنه خود متد تغییر می‌کند. همچنین با کاستن پارامترها مساله Long Parameter List حل خواهد شد.

```
makeCircle(circle)
```

Replace Parameter with Method Call

به جای فراخوانی متدها و ارسال مقادیرشان به عنوان پارامتر به متدی دیگر، آن‌ها را داخل متد نهایی صدا بزنید.

مثال قبلی را با این فرض ببینید که مقادیر مورد نیاز از کاربر و با استفاده از متد request دریافت می‌شود.

```
x = request("x")
y = request("y")
radius = request("radius")
makeCircle(x, y, radius)
```

رعایت این تکنیک همان مزیت Preserve Whole Object را دارد.

```
function makeCircle()
{
      x = request("x")
      y = request("y")
      radius = request("radius")
      // ...
}
```

همه جا این تکنیک و تکنیک قبلی قابل استفاده نیست، مثلاً نیاز دارید بعد از دریافت یک سطر از دیتابیس متد بالا را صدا بزنید. در چنین شرایطی پارامترها را به متد اضافه کنید.

Introduce Parameter Object

اگر چند متد با پارامترهای تکراری دارید، پارامترها را در غالب یک کلاس تعریف و به متدها پاس بدهید.

```
function amountInvoiced(Date startDate, Date endDate)
function amountReceived(Date startDate, Date endDate)
function amountOverdue(Date startDate, Date endDate)
```

کنترل پارامترها به عنوان یک شئ به مراتب آسان‌تر از مدیریت فهرستی از پارامترهاست.

```
function amountInvoiced(DateRange aDateRange)
function amountReceived(DateRange aDateRange)
function amountOverdue(DateRange aDateRange)
```

Remove Setting Method

متدهای setter که در طول برنامه استفاده‌ای ندارند یا فقط هنگام ایجاد شئ صدا زده می‌شوند را حذف کنید. نیازی به setter برای مقداری که قرار نیست در طول برنامه تغییر کند یا فقط در constructor ست می‌شود نیست.

```
Class Order
{
      private _vat
      public function setVAT(int vat)
      public function getVAT()

      public function order(int vat)
      {
            this.setVAT(vat)
      }
}
```

بدین ترتیب از مقداری که قرار نیست در طول برنامه تغییر کند محافظت خواهد شد و کلاس خلوت‌تر می‌شود.

```
Class Order
{
      private _vat
      public function getVAT()
```

```
        public function order(int vat)
        {
                    this._vat = vat
        }
}
```

Hide Method

متدی که در سایر کلاس‌ها استفاده نمی‌شود را private یا protected کنید. این موضوع درک اجزای کلاس را آسان‌تر خواهد کرد.

Replace Constructor with Factory Method

اگر constructor کلاس کاری بیش از تنظیم مقادیر انجام می‌دهد، آن را با Factory Method جایگزین کنید.

```
Class Employee
{
        const ENGINEER = 0
        const SALESMAN = 1
        const MANAGER = 2

        private _type

        public function Employee(type)
        {
                this._type = type
        }
}

// use
manager = new Employee(2)
```

.refactor

```
Class Employee
{
        public function Employee(type)
        {
```

```
            //...
        }

        public function createEngineer()
        {
        return new Engineer
        }

        public function createSalesMan()
        {
                    return new SalesMan
        }

        public function createManager()
        {
                    return new Manager
        }
}

class Engineer extends Employee
class SalesMan extends Employee
class Manager extends Employee
```

Replace Error Code with Exception

برای نمایش خطا به جای بازگردادن کد از Exception استفاده کنید.

```
if(data !== null)
{
        return data
}
return -1
```

این روش قدیمی و منسوخ شده.

```
if(data !== null)
{
        return data
```

```
}
throw new dataException()
```

Replace Exception with Test

از Exception بـرای مـدیریت خطاهـای غیرمنتظـره و غیرقابـل پیش‌بینـی استفاده و تا جای ممکن از رسیدن به آن‌ها و اجراییشان جلوگیری نمایید.

```
foreach(element in values)
{
	try
	{
		item[element.Key] = element.Value
	}
	catch {}
}
```

این یـک نمونـه بـد و نامناسـب اسـت. زیـرا بـا شـرط و بررسـی سـاده می‌تـوان کـد واضح‌تری نوشت.

```
foreach(element in values)
{
	if(!item.IsValidValue(element))
	{
			continue
	}

	item[element.Key] = element.Value
}
```

فصل هشتم- Dealing with Generalization

Dealing with Generalization

در این فصل ساختار و انسجام کلاس‌ها را بهبود می‌دهیم. انتقال فیلد و متد بین کلاس‌های پدر و فرزند و...

Pull Up Field

فیلدهای تکراری در subclassها را به کلاس پدر منتقل کنید.

Pull Up Method

متدهای تکراری در subclassها را به کلاس پدر منتقل کنید.
فیلد و متد تکراری ممکن است در زمان توسعه و هنگام کار گروهی ایجاد شود. مخصوصاً جایی که زیرکلاس‌ها جداگانه توسعه داده می‌شوند. مثلاً همکار شما بدون اطلاع از موجود بودن یک متد دوباره آن را تعریف کرده.

Pull Up Constructor Body

constructor subclassهایی که مشابه constructor کلاس پدر عمل می‌کنند را ویرایش و به جای آن constructor پدر را صدا بزنید.

```
class Employee
{
	public function Employee(data)
```

```
        {
                parentConstructor(data) // call parent construct
        }
}
```

Push Down Field

فیلدهایی که در کلاس اصلی تعریف شده‌اند اما داخل تمام subclassها مورد استفاده قرار نمی‌گیرند را به subclass منتقل کنید.

Push Down Method

متدی که در کلاس اصلی تعریف شده اما داخل تمام subclassها استفاده نمی‌شود را به subclass مربوطه منتقل کنید.

انتقال فیلد و متد به subclass ها باعث انسجام بهتر آن‌ها می‌شود. اگر یک فیلد یا متد در بیش از یک subclass استفاده می‌شود (نه در تمام زیرکلاس‌ها)، از یک subclass میانی برای قرارداد متد و فیلدهای مشترک آن‌ها استفاده کنید.

Extract Subclass

برای ویژگی‌هایی از کلاس که زیاد مورد استفاده قرار نمی‌گیرند یک subclass جدید تعریف و آن ویژگی‌ها را به subclass منتقل کنید.

Extract Superclass

برای کلاس‌های اصلی/ پدر که شامل متد و فیلدهای مشترکی هستند یک کلاس پدر ایجاد و مشترکات آن‌ها را به کلاس پدر جدید منتقل کنید.

Extract Interface

مانند تکنیک قبلی است اما در مورد Interfaceها. مشترکات Interfaceها را به یک کلاس پدر ببرید.

Collapse Hierarchy

برنامه شما رشد کرده و یک زیر کلاس با کلاس اصلی خود برابر شده است. زیر کلاس مذکور را حذف و به جای آن از کلاس اصلی استفاده کنید.

Form Template Method

subclassها فرآیندی را به ترتیب و با گام‌های مشابه انجام می‌دهند. این همان Template Pattern دیزاین‌پترن‌هاست. در این الگو اسکلت و مراحل کار تعریف می‌شود و کلاس‌های فرزند موظف به رعایت ترتیب آن هستند.

فرض کنید قرار است یک خانه بسازیم. هیچ وقت نمی‌توان از سقف خانه شروع کرد. مراحل کار مشخص است. زیرسازی، دیوارها، سقف و...

Template Pattern مراحل بالا را تعریف می‌کند. حالا دو زیرکلاس داریم. سازنده خانه چوبی و خانه بتنی. هر دو بر اساس مراحل عمل می‌کنند. اول زیرسازی سپس دیوارها و... اما جنس کارشان (بدنه متدها) متفاوت است.

Replace Inheritance with Delegation

اگر subclass از تمام ویژگی‌های کلاس پدر استفاده نمی‌کند به جای وراثت از delegation استفاده کنید.

Replace Delegation with Inheritance

عکس تکنیک بالا.

اگر delegation از تمام یا بیشتر ویژگی‌های یک کلاس استفاده می‌کند، به جای delegation از وراثت استفاده کنید.

پایان