读书笔记

《重构 改善既有代码的设计》

本文github地址：
https://github.com/YoungBear/MyBlog/blob/master/refactor.md

重构：在不改变软件可观察行为的前提下改善其内部结构。

refactoring

tips：

如果你发现自己需要为程序添加一个特性，而代码结构使你无法很方便地达成目的，那就先重构那个程序，使特性的添加比较容易进行，然后再添加特性。

重构前，先检查自己是否有一套可靠的测试机制。这些测试必须有自我检验能力。

重构技术就是以微小的步伐修改程序。如果你犯下错误，很容易便可发现它。

任何一个傻瓜都能写出计算机可以理解的代码。唯有写出人类容易理解的代码，才是优秀的程序员。(变量命名)

重构(名词):对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。

重构(动词):使用一系列重构方法，在不改变软件可观察可观察行为的前提下，调整其结构。

为何重构

1. 重构改进软件设计
2. 重构使软件更容易理解
3. 重构帮助找到Bug
4. 重构提高编程速度

何时重构

三次法则

第一次做某件事时只管去做；第二次做类似的事会产生反感，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。

tips：

事不过三，三则重构。

1. 添加新功能时重构
2. 修补错误时重构
3. 复审代码时重构

第3章 代码的坏味道

1. Duplicated Code 重复代码
2. Long Method 过长函数
3. Large Class 过大的类
4. Long Parameter List 过长参数列
5. Divergent Change 发散式变化 -- 软件能够更容易被修改
6. Shotgun Surgery 霰弹式修改 -- 在很多类中做出许多小的修改

7. Feature Envy 依恋情结 -- 函数对某个类的兴趣高过对自己所处类的兴趣
8. Data Clumps 数据泥团 -- 很多相同的参数(可以新建一个类来保存)
9. Primitive Obsession 基本类型偏执
10. Switch Statements switch 惊悚现身
11. Parallel Inheritance Hierarchies 平行继承体系
12. Lazy Class 冗赘类
13. Speculative Generality 夸夸其谈未来性
14. Temporary Field 令人迷惑的暂时字段
15. Message Chains 过度耦合的消息链
16. Middle Man 中间人
17. Inappropriate Intimacy 狎昵关系
18. Alternative Classes with Different Interfaces 异曲同工的类
19. Incomplete Library Class 不完美的类库
20. Data Class 纯稚的数据类
21. Refused Bequest 被拒绝的遗赠
22. Comments 过多的注释

第6章 重新组织函数

6.1 Extract Method 提炼函数

6.2 Inline Method 内联函数

6.3 Inline Temp 内联临时变量

6.4 Replace Temp with Query 以查询取代临时变量

6.5 Introduce Explaining Variable 引入解释性变量

6.6 Split Temporary Variable 分解临时变量

如果某一个临时变量被赋值超过一次，并且它既不是循环变量，也不是用于搜集计算结果，**则针对每次赋值，创造一个独立、对应的临时变量。**

做法：

1. 将新的临时变量声明为final
2. 以该临时变量的第二次赋值动作为界，修改此前对该临时变量的所有引用点，让它们引用新的临时变量
3. 在第二次赋值处，重新声明原先那个临时变量
4. 编译，测试
5. 逐次重复上述过程。每次都在声明处对临时变量改名，并修改下次赋值之前的引用点

6.7 Remove Assignments to Parameters 移除对参数的赋值

做法：

1. 建立一个临时变量，把待处理的参数值赋予它。
2. 以“对参数的赋值”为界，将其后所有对此参数的引用点，全部替换为“对此临时变量的引用”。
3. 修改赋值语句，使其改为对新建之临时变量赋值。
4. 编译，测试。

6.8 Replace Method with Method Object 以函数对象取代函数

如果一个函数中局部变量泛滥成灾，那么想分解这个函数是非常困难的。以**查询取代临时变量**可以助你减轻这个负担，但有时候你会发现根本无法拆解一个需要拆解的函数。这种情况下，你应该把手伸进工具箱的深处，祭出**函数对象**这件法宝。

做法：

1. 建立一个**新类**，根据待处理函数的用途，为这个类命名。
2. 在新类中建立一个**final**字段，用以保存**原先大型函数所在的对象**。我们将这个字段成为“源对象”。同时，针对原函数的**每个临时变量**和**每个参数**，在新类中建立一个对应的字段保存之。
3. 在新类中建立一个构造函数，接收源对象及原函数的所有参数作为参数。
4. 在新类中建立一个compute()函数。
5. 将原函数的代码复制到compute()函数中。如果需要调用源对象的任何函数，请通过源对象字段调用。
6. 编译。
7. 将旧函数的函数本体替换为这样一条语句：“创建上述新类的一个新对象，而后调用其中的compute()函数”。

这项重构的好处是：我们可以轻松地对compute()函数采取Extract Method(提炼函数)，不必担心参数传递的问题。

6.9 Substitute Algorithm 替换算法

把某一个算法替换为另一个更清晰的算法。

第7章 在对象之间搬移特性

7.1 Move Method (搬移函数)

你的程序中，**有个函数**与其所驻类之外的另一个类进行更多交流：调用后者，或者被后者调用。

思路：
在该函数最常引用的类中建立一个有着类似行为的新函数。将旧函数变成一个单纯的委托函数，或是将旧函数完全移除。

7.2 Move Field （搬移字段）

你的程序中，**某个字段**被其所驻类之外的另一个类更多地用到。

思路：在目标类新建一个字段，修改源字段的所有用户，令它们改用新字段。

7.3 Extract Class (提炼类)

某各类做了应该由两个类做的事。

思路：建立一个新类，将相关的字段和函数从旧类搬移到新类。

7.4 Inline Class (将类内联化)

某个类没有做太多事情。

思路：将这个类的所有特性搬移到另一个类中，然后移除原类。

7.5 Hide Delegate (隐藏“委托关系”)

客户通过一个委托类来调用另一个对象。

思路：在服务类上建立客户所需求的所有函数，用以隐藏委托关系。

7.6 Remove Middle Man (移除中间人)

某个类做了过多的简单委托动作。

思路：让客户直接调用受托类。(和7.5刚好相反)

7.7 Introduce Foreign Method (引入外加函数)

你需要为提供服务的类增加一个函数，但你无法修改这个类。

思路：在客户类中建立一个函数，并以第一参数形式传入一个服务类实例。

7.8 Introduce Local Extension (引入本地扩展)

你需要为服务类提供一些额外函数，但你无法修改这个类。

思路：建立一个新类，使它包含这些额外函数。让这个扩展品成为源类的子类或包装类。

第8章 重新组织数据

8.1 Self Encapsulate Filed (自封装字段)

将属性声明为private，使用get/set函数来访问。

8.2 Replace Data Value with Object (以对象取代数据值)

你有一个数据项，需要与其他数据和行为一起使用才有意义。

思路：将数据项变成对象。

8.3 Change Value to Reference (将值对象改为引用对象)

你从一个类衍生出许多彼此相等的实例，希望将它们替换为同一个对象。

思路：将这个值对象变成引用对象。

8.4 Change Reference to Value (将引用对象改为值对象)

你有一个引用对象，很小且不可变，而且不易管理。

思路：将它变成值对象。

8.5 Replace Array with Object (以对象取代数组)

你有一个数组，其中的元素各自代表不同的东西。

思路：以对象替换数组，其中的数组中的每个元素，以一个字段来表示。

8.6 Duplicate Observed Data (复制“被监视数据”)

你有一些领域数据置身于GUI控件中，而领域函数需要访问这些数据。

思路：将该数据复制到一个领域对象中。建立一个Observer模式，用以同步领域对象和GUI对象内的重要数据。

8.7 Change Unidirectional Association to Bidirectional (将单向关联改为双向关联)

两个类都需要使用双方特性，但其间只有一条单向连接。

思路：添加一个反向指针，并使修改函数能够同时更新两条连接。

8.8 Change Bidirectional Association to Unidirectional (将双向关联改为单向关联)

两个类之间有双向关联，但其中一个类如今不再需要另一个类的特性。

思路：去除不必要的关联。

8.9 Replace Magic Number with Symbolic Constant (以字面常量取代魔法数)

你有一个字面数值，带有特别含义。

思路：创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

eg. 使用PI来代替3.14

8.10 Encapsulate Field （封装字段）

你的类中存在一个 public 字段。

思路：将它声明为 private， 并提供相应的访问函数。

8.11 Encapsulate Collection (封装集合)

有一个函数返回一个集合。

思路：让这个函数返回该集合的一个只读副本，并在这个类中提供添加/移除集合元素的函数。

动机：

我们常常会在一个类中使用集合(collection,可能是array,list,set或vector)来保存一组实例。这样的类通常也会提供指针对该集合的取值/设值函数。

但是，集合的处理方式应该和其他种类的数据略有不同。取值函数不该返回集合自身，因为这会让用户得以修改集合内容而集合拥有者却一无所悉。这也会对用户暴露过多对象内部数据结构的信息。如果一个取值函数确实需要返回多个值，它应该避免用户直接操作对象内所保存的集合，并隐藏对象内与用户无关的数据结构。至于如何做到这一点，视你使用的 Java 版本不同而有所不同。

另外，不应该为这整个集合提供一个设值函数，但应该提供用以为集合添加/移除元素的函数。这样，集合拥有者(对象)就可以控制元素的添加和移除。

如果你做到以上几点，集合就可以很好地封装起来了，这便可以降低集合拥有者和用户之间的耦合度。

8.12 Replace Record with Data Class (以数据类取代记录)

你需要面对传统编程环境中的记录结构。

思路：**为该记录创建一个“哑”数据对象。**

8.13 Replace Type Code with Class (以类取代类型码)

类之中有一个数值类型码，但它并不影响类的行为。

思路:**以一个新的类替换该数值类型码。**

8.14 Replace Type Code with Subclasses (以子类取代类型码)

你有一个不可变的类型码，它会影响类的行为。

思路：**以子类取代这个类型码。**

8.15 Replace Type Code with State/Strategy (以 State/Strategy 取代类型码)

你有一个类型码，它会影响类的行为，但你无法通过集成手法消除它。

思路：**以状态对象取代类型码。**

8.16 Replace Subclass with Fields (以字段取代子类)

你的各个子类的唯一差别只在“返回常量数据”的函数身上。

思路：**修改这些函数，使它们返回超类中的某个(新增)字段，然后销毁子类。**

动机：

建立子类的目的，是为了增加新特性或变化其行为。有一种变化行为被称为“常量函数(constant method)”，它们会返回一个硬编码的值。这东西有其用途：你可以让不同的子类中的同一个访问函数返回不同的值。你可以在超类中将反问函数声明为抽象函数，并在不同的子类中让它返回不同的值。

尽管常量函数有其用途，但若子类只有常量函数，实在没有足够的存在价值。你可以在超类中设计一个与常量函数返回值相应的字段，从而完全去除这样的子类。如此一来就可以避免因继承而带来的额外复杂性。

第9章 简化条件表达式

9.1 Decompose Conditional (分解条件表达式)

你有一个复杂的条件 (if-then-else) 语句。

思路：**从 if,then,else 三个段落中分别提炼出独立函数。**

动机

程序之中，复杂的条件逻辑是最常导致复杂度上升的地点之一。你必须编写代码来检查不同的条件分支、根据不同的分支做不同的事，然后，你很快就会得到一个相当长二代函数。大型函数自身就会使代码的可读性下降，而条件逻辑则会使代码更难阅读。在带有复杂条件逻辑的函数中，代码(包括检查条件分支的代码和真正实现功能的代码)会告诉你发生的事，但常常让你弄不清楚为什么会发生这样的事，这就说明代码的可读性的确大大降低了。

和任何大块头代码一样，你可以将它分解为多个独立函数，根据每个小块代码的用途，为分解而得的新函数命名，并将原函数中对应的代码改为调用新建函数，从而更清楚地表达自己的意图。对于条件逻辑，将每个分支条件分解成新函数还可以给你带来更多好处：可以突出条件逻辑，更清楚地表达每个分支的作用，并且突出每个分支的原因。

做法

- 将 if 段落提炼出来，构成一个独立函数。
- 将 then 段落和 else 段落都提炼出来，各自构成一个独立函数。

9.2 Consolidate Conditional Expression (合并条件表达式)

你有一系列条件测试，都得到相同结果。

思路：**将这些测试合并为一个条件表达式，并将这个条件表达式提炼成为一个独立函数。**

动机

有时你会发现这样一串条件检查：检查条件各不相同，最终行为却一致。如果发现这种情况，就应该使用“逻辑或”和“逻辑与”将它们合并为一个条件表达式。

之所以要合并条件代码，有两个重要原因。首先，合并后的条件代码会告诉你“实际上只有一次条件检查，只不过有多个并列条件需要检查而已”，从而使这一次检查的用意更清晰。当然， 合并前和合并后的代码有着相同的效果，但原先代码传达出的信息却是“这里有一些各自独立的条件测试，它们只是恰好同时发生”。其次，这项重构往往可以为你使用 Extract Method 做好准备。将检查条件提炼成一个独立函数对于理清代码意义非常有用，因为它把描述“做什么”的语句换成了“为什么这样做”。

条件语句的合并理由也同时指出了不要合并的理由：**如果你认为这些检查的确彼此独立，的确不应该被视为同一次检查，那么就不要使用本项重构。**因为在这种情况下，你的代码已经很清楚表达出自己的意义。

9.3 Consolidate Duplicate Conditional Fragments (合并重复的条件片段)

在条件表达式的**每个分支**上有着相同的一段代码。

思路：**将这段重复代码搬移到条件表达式之外。**

动机：

有时你会发现，一组条件表达式的所有分支都执行了相同的某段代码。如果是这样，你就应该将这段代码搬移到条件表达式外面。这样，代码才能更清楚地表明哪些东西随条件的变化而变化、哪些东西保持不变。

9.4 Remove Control Flag (移除控制标记)

在一系列布尔表达式中，某个变量带有“控制标记”(control flag) 的作用。

思路：**以 break 语句或 return 语句取代控制标记。**

9.5 Replace Nested Conditional with Guard Clauses (以卫语句取代嵌套条件表达式)

函数中的条件逻辑使人难以看清正常的执行路径。

思路：**使用卫语句表现所有特殊情况。**

eg.

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isReetired) result = retiredAmount();
            else result = normalPayAmount();
        }
    }
    return result;
}
```

重构之后：

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
}
```

动机：

根据我的经验，条件表达式通常有两种表现形式。第一种形式是：所有分支都属于正常行为。第二种形式是：条件表达式提供的答案中只有一种是正常行为，其他都不是不常见的情况。

这两类条件表达式有不同的用途，这一点应该通过代码表现出来。如果两条分支都是正常行为，就应该使用形如 if...else... 的条件表达式；如果**某个条件极其罕见**，就应该**单独检查该条件**，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为“卫语句” (guard clauses)。

Replace Nested Conditional with Guard Clauses 的精髓就是：给某一条分支以特别的重视。如果使用 if-then-else 结构，你对 if 分支
和 else 分支的重视是同等的。这样的代码结构传递给阅读者的消息就是：各个分支有同样的重要性。卫语句就不同了，它告诉阅读者：“这种情况很罕见，如果它真的发生了，请做一些必要的整理工作，然后退出。”

“每个函数只能有一个入口和一个出口”的观念，根深蒂固于某些程序员的脑海里。我发现，当我处理他们编写的代码时，经常需要使用这项重构。现今的编程语言都会强制保证每个函数只有一个入口，至于“单一出口”规则，其实不是那么有用。在我看来，**保持代码清晰才是最关键的**：如果单一出口能使这个函数更清楚易读，那么就使用单一出口；否则就不必这么做。

嵌套条件代码往往由那些深信“每个函数只能有一个出口”的程序员写出。我发现那条规则实在有点太简单粗暴了。如果对函数剩余部分不再有兴趣，当然应该立刻退出。引导阅读者去看一个没有用的else区段，只会妨碍他们的理解。

范例：将条件反转

我们常常可以将条件表达式反转，从而实现该项重构。

初始代码：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital > 0.0) {
        if (_intRate > 0.0 && _duration > 0.0) {
            result = (_income / _duration) * ADJ_FACTOR;
        }
    }
    return result;
}
```

我们将逐一进行替换。不过这次在插入卫语句时，我们需要将相应的**条件反转**过来：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;//将这个条件反转，并使用卫语句(Guard Glauses)
    if (_intRate > 0.0 && _duration > 0.0) {
            result = (_income / _duration) * ADJ_FACTOR;
    }
    return result;
}
```

下一个条件稍微复杂一点，所以我们分两步进行逆反。首先加入一个**逻辑非**操作：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (!(_intRate > 0.0 && _duration > 0.0)) return result;//加入逻辑非操作，并使用卫语句
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

将逻辑非简化：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate <= 0.0 || _duration <= 0.0)) return result;//简化逻辑非操作
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

这时候，我比较喜欢在卫语句内返回一个明确值，因为这样我们可以一目了然地看到卫语句返回的失败结果。此外，这种时候我们也会考虑使用 Replace Magic Number with System Constant。

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return 0.0;//在卫语句中返回明确值
    if (_intRate <= 0.0 || _duration <= 0.0)) return 0.0;//在卫语句中返回明确值
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

完成替换之后，我们同样可以将临时变量移除：

```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0)) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

9.6 Replace Conditional with Polymorphism (以多态取代条件表达式)

你手上有个条件表达式，它根据对象类型的不同而选择不同的行为。

思路：**将这个条件表达式的每个分支放进一个子类内的覆写函数中，然后将原始函数声明为抽象函数。**

动机

在面向对象术语中，听上去最高贵的词非“多态”莫属。多态最根本的好处就是：如果你需要根据对象的不同类型而采取不同的行为，多态使你不必编写明显的条件表达式。

正因为有了多态，所以你会发现：“类型码的 switch 语句”以及“基于类型名称的 if-then-else 语句” 在面向对象程序中很少出现。

多态能够给你带来很多好处。如果同一组条件表达式在程序许多地点出现，那么使用多态的收益是最大的。使用条件表达式时，如果你想添加一种新类型，就必须查找并更新所有条件表达式。但如果改用多态，只需建立一个新的子类，并在其中提供适当的函数就行了。类的用户不需要了解这个子类，这就大大降低了系统各部分之间的依赖，使系统升级更加容易。

9.7 Introduce Null Object (引入 Null 对象)

你需要再三检查某对象是否为null。

思路：**将null值替换为null对象。**

9.8 Introduce Assertion (引入断言)

某一段代码需要对程序状态做出某种假设。

思路：**以断言明确表现这种假设。**

```
double getExpenseLimit() {
    //shoule have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
            _expenseLimit : _primaryProject.getMemberExpenseLimit();
}
```

这项重构后：

```
double getExpenseLimit() {
    Assert.isTrue(_expenseLimit != NULL_EXPENSE || _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
            _expenseLimit : _primaryProject.getMemberExpenseLimit();
}
```

动机

常常会有这样一段代码：只有当某个条件为真时，该段代码才能正常运行。例如平方根计算只对正值才能进行，又例如某个对象可能假设其字段至少有一个不等于null。

这样的假设通常并没有在代码中明确表现出来，你必须阅读整个算法才能看出。有时程序员会以注释写出这样的假设。而我要介绍的是一种更好的技术：使用断言明确标明这些假设。

断言是一个条件表达式，应该总是为真。如果它失败，表示程序员犯了错误。因此断言的失败应该导致一个非受控异常(unchecked exception)。断言绝对不能被系统的其他部分使用。实际上，程序最后的成品往往将断言统统删除。因此，标记“某些东西是个断言”是很重要的。

断言可以作为交流与调试的辅助。在交流的角度上，断言可以帮助程序阅读者理解代码所做的假设；在调试的角度上，断言可以在距离bug最近的地方抓住它们。当我编写自我测试代码的时候发现，断言在调试方面的帮助变得不那么重要了，但我仍然非常看重它们在交流方面的价值。

第10章 简化函数调用

10.1 Rename Method (函数改名)

函数的名称未能揭示函数的用途。

思路：**修改函数名称。**

10.2 Add Parameter (添加参数)

某个函数需要从调用端得到更多信息。

思路：**为此函数添加一个对象参数，让该对象带进函数所需信息。**

10.3 Remove Parameter (移除参数)

函数本体不再需要某个参数。

思路：**将该参数去除。**

10.4 Separate Query from Modifier (将查询函数和修改函数分离)

某个函数既返回对象状态值，又修改对象状态。

思路：**建立两个不同的函数，其中一个负责查询，另一个负责修改。**

10.5 Parameterize Method (令函数携带参数)

若干函数做了类似的工作，但在函数本体中却包含了不同的值。

思路：**建立单一函数，以参数表达那些不同的值。**

动机：

你可能会发现这样的两个函数：它们做着类似的工作，但因少数几个值致使行为略有不同。这种情况下，你可以将这些各自分离的函数统一起来，并通过参数来处理那些变化情况，用以简化问题。这样的修改可以去除重复的代码，并提高灵活性，因为你可以用这个参数处理更多的变化情况。

范例

一个最简单的例子：

```
class Employee {
    void tenPercentRaise() {
        salary *= 1.1;
    }

    void fivePercentRaise() {
        salary *= 1.05;
    }
}
```

这段代码可以替换如下：

```
void raise(double factor) {
    salary *= (1 + factor);
}
```

本项重构的要点在于：以“可将少量数值视为参数”为依据，找出带有重复性的代码。

10.6 Replace Rarameter with Explicit Methods (以明确函数取代参数)

你有一个函数，其中完全取决于参数值而采取**不同行为。**

思路：**针对该参数的每一个可能值，建立一个独立函数。**

eg.

```
void setValue(String name, int value) {
    if (name.equals("height")) {
        _height = value;
        return;
    }
    if (name.equals("width")) {
        _width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

经过这项重构后：

```
void setHeight(int arg) {
    _height = arg;
}

void setWidth(int arg) {
    _width = arg;
}
```

动机：

Replace Parameter with Explicit Methods 恰恰相反于 Parameterize Method 。如果某个参数有多种可能的值，而函数内又以条件表达式检查这些参数值，并根据**不同参数值做出不同的行为**，那么就应该使用本项重构。调用者原本必须赋予参数适当的值，以决定该函数做出何种响应。现在，既然你提供了不同的函数给调用者使用，就可以避免出现条件表达式。此外你还可以获得编译期检查的好处，而且接口也更清楚。如果以参数值决定函数行为，那么函数用户不但需要观察该函数，而且还要判断参数值是否合法，而“合法的参数值”往往很少在文档中被清楚地提出。

就算不考虑编译期检查的好处，只是为了获得一个清晰的接口，也值得你执行本项重构。哪怕只是给一个内部的布尔变量赋值，相比较之下，Switch.beOn() 也比 Switch.setState(true) 要清楚得多。

但是，**如果参数值不会对函数行为有太多影响**，你就不应该使用 Replace Parameter with Explicit Methods。如果情况这是这样，而你也只需要通过参数为一个字段赋值，那么直接使用设置函数就行了。如果的确需要条件判断的行为，可考虑使用 Replace Conditional with Polymorphism。

做法：

- 针对参数的每一种可能值，新建一个明确函数。
- 修改条件表达式的每个分支，使其调用合适的新函数。
- 修改每个分支后，编译并测试。
- 修改原函数的每一个被调用点，改而调用上述的某个合适的新函数。
- 编译，测试。
- 所有调用端都修改完毕后，删除原函数。

10.7 Preserve Whole Object (保持对象完整)

你从某个对象中取出若干值，将它们作为某一次函数调用时的参数。

思路：**改为传递整个对象。**

eg.

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

经过这项重构后：

```
withinPlan = play.withinRange(daysTempRange());
```

动机：

有时候，你会将来自同一对象的若干项数据作为参数，传递给某个函数。这样做的问题在于：万一将来被调用函数**需要新的数据项**，你就必须查找并修改对此函数的所有调用。如果你把这些数据所属的**整个对象**传给函数，可以避免这种尴尬的处境，因为被调用函数可以向那个参数对象请求任何它想要的信息。

除了可以使参数列更稳固之外，Preserve Whole Object 往往还能提高代码的可读性。过长的参数列很难使用，因为调用者和被调用者都必须记住这些参数的用途。此外，不使用完整对象也会造成重复代码，因为被调用函数无法利用完整对象中的函数来计算某些中间值。

不过事情总有两面。如果你传的是数值，被调用函数就只依赖于这些数值，而不依赖它们所数的对象。但如果你传递的是整个对象，被调用函数所在的对象就需要依赖参数对象。如果这会使你的依赖结构恶化，那么就不该使用Preserve Whole Object。

还有一种不使用 Preserve Whole Object 的理由：如果被调用函数只需要参数对象的其中一项数值，那么只传递那个数值会更好。我并不认同这种观点，因为传递一项数值和传递一个对象，至少在代码清晰度上是等价的(当然对于按值传递的参数来说，性能上可能有所差异)。更重要的考量应该放在对象之间的依赖关系上。

如果被调用函数使用了来自另一个对象的很多数据，这可能意味着该函数实际上**应该本定义在那些数据所属的对象**中。所以，考虑 Preserce Whole Object 的同时，你也应该考虑Move Method。

运用本项重构之前，你可能还没定义一个完整对象。那么就应该先用 Introduce Parameter Object 。

还有一种常见情况：调用者将自己的若干数据作为参数，传递给被调用函数。这种情况下，如果该对象有合适的取值函数，你可以使用**this**取代这些参数值，并且无需担心对象依赖问题。

10.8 Replace Parameter with Methods (以函数取代参数)

对象调用某个函数，并将所得结果作为参数，传递给另一个函数。而接受该参数的函数本身也能够调用前一个函数。

思路：**让参数接受者取出该项参数，并直接调用前一个函数。**

eg.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

通过本项重构后：

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);//在discountedPrice内部调用getDiscountLevel()
```

动机

如果函数可以通过其他途径获得参数值，那么它就不应该通过**参数**取得该值。过长的参数列会增加程序阅读者的理解难度，因此我们应该尽可能缩短参数列的长度。

10.9 Introduce Parameter Object(引入参数对象)

某些参数总是很自然地同时出现。

思路：**以一个对象取代这些参数。**

动机

你经常会看到特定的一组参数总是一起被传递。可能有好几个函数都使用这一组参数，这些函数可能隶属同一个类，也可能隶属于不同的类。这样一组参数就是所谓的 Data Clumps (数据抱团)，我们可以**运用一个对象包装所有这些数据**，再以该对象取代它们。哪怕只是为了把这些数据组织在一起，这样做也是值得的。本项重构的价值在于缩短参数列，而你知道，过长的参数列总是难以理解的。此外，新对象所定义的访问函数还可以使代码更具一致性，这又进一步降低了理解和修改代码的难度。

本项重构还可以带给你更多好处。当你把这些参数组织到一起之后，往往很快可以发现一些可被移至新建类的行为。通常，原本使用那些参数的函数对这一组参数会有一些共通的处理，如果将这些共通行为移到新对象中，你可以减少很多重复代码。

10.10 Remove Setting Method (移除设值函数)

类中的某个字段应该在对象创建时被设值，然后就不再改变。

思路：**去掉该字段的所有设值函数。**

动机

如果你为某个字段提供了设值函数，这就按时这个字段值可以被改变。如果你不希望在对象创建之后此字段还有机会被改变，那就不要为它提供设值函数 (同时将该字段设为final)。这样你的意图会更加清晰，并且可以排除其值被修改的可能性————这种可能性往往是非常大的。

如果你保留了间接访问变量的方法，就可能疆场有程序员盲目使用它们。这些人甚至会在构造函数中使用设值函数！我猜想他们或许是为了代码的一致性，但却忽略了设值函数往后可能带来的混淆。

10.11 Hide Method (隐藏函数)

有一个函数，从来没有被其他任何类用到。

思路：**将这个函数修改为private。**

动机

重构往往促使你修改函数的可见度。提高函数可见度的情况很容易想象：另一个类需要用到某个函数，因此你必须提高该函数的可见度。但是要指出一个函数的可见度是否过高，就稍微困难一些。理想状况下，你可以使用工具检查所有函数，指出可被隐藏起来的函数。即使没有这样的工具，你也应该时常进行这样的检查。

一种特别常见的情况是：当你面对一个过于丰富、提供了过多行为的接口时，就值得将**非必要**的取值函数和设值函数隐藏起来。尤其当你面对的是一个只有简单封装的数据容器时，情况更是如此。随着越来越多行为被放入这个类，你会发现许多取值/设值函数不再需要公开，因此可以把它们隐藏起来。如果你把取值/设值函数设为private，然后在所有地方都直接访问变量，那就可一个放心移除取值/设值函数了。

10.12 Replace Constructor with Factory Method (以工厂函数取代构造函数)

你希望在创建对象时不仅仅是做简单的建构动作。

思路：**将构造函数替换为工厂函数。**

动机

使用 Replace Constructor with Factory Method 的最显而易见的动机，就是在派生子类的过程中以工厂函数取代类型码。你可能常常需要根据类型码创建相应的对象，现在，创建名单中还得加上子类，那些子类也是根据类型码来创建。然而由于构造函数只能返回单一类型的对象，因此你需要将构造函数替换为工厂函数。

此外，如果构造函数的功能不能满足你的需要，也可以使用工厂函数来代替它。工厂函数也是 Change Value to Reference 的基础。你也可以令你的工厂函数根据参数的个数和类型，选择不同的创建行为。

10.13 Encapsulate Downcast (封装向下转型)

某个函数返回的对象，需要由函数调用者执行向下转型(downcast)。

思路：**将向下转型动作移到函数中。**

eg.

```
Object lastReading() {
    return readings.lastElement();
}
```

通过这项重构后：

```
Reading lastReading() {
    return (Reading) readings.lastElement();
}
```

10.14 Replace Error Code with Exception (以异常取代错误码)

某个函数返回一个特定的代码，用以表示某种错误情况。

思路：**改用异常。**

eg.

```
int withdraw(int amount) {
    if (amount > _balance) {
        return -1;
    } else {
        _balance -= amount;
        return 0;
    }
}
```

通过这项重构后：

```
void withdraw(int amount) throws BalanceExceprion {
    if (amount > _balance) {
        throw new BalanceException();
    }
    _balance -= amount;
}
```

10.15 Replace Exception with Test (以测试取代异常)

面对一个调用者可以预先检查的条件，你抛出了一个异常。

思路：**修改调用者，使它在调用函数之前先做检查。**

eg.

```
double getValueForPeriod(int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```

通过这项重构后：

```
double getValueForPeriod(int periodNumber) {
    if (periodNumber >= _values.length) {
        return 0;
    }
    return _values[periodNumber];
}
```

第11章 处理概括关系

11.1 Pull Up Field (字段上移)

两个子类拥有相同的字段。

思路：**将该字段移至超类。**

11.2 Pull Up Method (函数上移)

有些函数，在各个子类中产生完全相同的结果。

思路：**将该函数移至超类。**

11.3 Pull Up Constructor Body (构造函数本体上移)

你在各个子类中拥有一些构造函数，它们的本体几乎完全一致。

思路：**在超类中新建一个构造函数，并在子类构造函数中调用它。**

eg.

```
class Manager extends Employee...
    public Manager(String name, String id, int grade) {
        _name = name;
        _id = id;
        _grade = grade;
    }
```

通过这项重构后：

```
public Manager(String name, String id, int grade) {
    super(name, id);
    _grade = grade;
}
```

11.4 Push Down Method (函数下移)

超类中的某个函数只与部分(而非全部)子类有关。

思路：**将这个函数移到相关的那些子类去。**

11.5 Push Down Field (字段下移)

超类中的某个字段只被部分(而非全部)子类用到。

思路：**将这个字段移到需要它的那些子类去。**

11.6 Extract Subclass (提炼子类)

类中的某些特性只被某些 (而非全部)实例用到。

思路：**新建一个子类，将上面所说的那一部分特性移到子类中。**

11.7 Extract Superclass (提炼超类)

两个类有相似特性。

思路：**为这两个类建立一个超类，将相同特性移至超类。**

11.8 Extract Interface (提炼接口)

若干客户使用类接口中的同一子集，或者两个类的接口有部分相同。

思路：**将相同的子集提炼到一个独立接口中。**

11.9 Collapse Hierarchy (折叠继承体系)

超类和子类之间无太大区别。

思路：**将它们合为一体。**

11.10 Form Template Method (塑造模板函数)

你有一些子类，其中相应的某些函数以相同顺序执行类似的操作，但各个操作的细节上有所不同。

思路：**将这些操作分别放进独立函数中，并保持它们都有相同的签名，于是原函数也就变得相同了。然后将原函数上移至超类。**

11.11 Replace Inheritance with Delegation (以委托取代继承)

某个子类只使用超类接口中的一部分，或是根本不需要继承而来的数据。

思路：**在子类中新建一个字段用以保存超类；调整子类函数，令它改而委托超类；然后去掉两者之间的继承关系。**

11.12 Replace Delegation with Inheritance (以继承取代委托)

你在两个类之间使用委托关系，并经常为整个接口编写许多极简单的委托函数。

思路：**让委托类继承受托类。**