# 重构

原作者github: https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

# 一、第一个案例

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，那就先重构这个程序。

在重构前，需要先构建好可靠的测试环境，确保安全地重构。

重构需要以微小的步伐修改程序，如果重构过程发生错误，很容易就能发现错误。

**案例分析**

影片出租店应用程序，需要计算每位顾客的消费金额。

包括三个类：Movie、Rental 和 Customer。


```
class Customer {

    private List<Rental> rentals = new ArrayList<>();

    void addRental(Rental rental) {
        rentals.add(rental);
    }
```

```
8.
9.      double getTotalCharge() {
10.         double totalCharge = 0.0;
11.         for (Rental rental : rentals) {
12.             switch (rental.getMovie().getMovieType()) {
13.                 case Movie.Type1:
14.                     totalCharge += rental.getDaysRented();
15.                     break;
16.                 case Movie.Type2:
17.                     totalCharge += rental.getDaysRented() * 2;
18.                     break;
19.                 case Movie.Type3:
20.                     totalCharge += rental.getDaysRented() * 3;
21.                     break;
22.             }
23.         }
24.         return totalCharge;
25.     }
26. }
```

```
1.  class Rental {
2.      private int daysRented;
3.
4.      private Movie movie;
5.
6.      Rental(int daysRented, Movie movie) {
7.          this.daysRented = daysRented;
8.          this.movie = movie;
9.      }
10.
11.     Movie getMovie() {
12.         return movie;
13.     }
14.
15.     int getDaysRented() {
16.         return daysRented;
17.     }
18. }
```

```
1.  class Movie {
2.
3.      static final int Type1 = 0, Type2 = 1, Type3 = 2;
4.
```

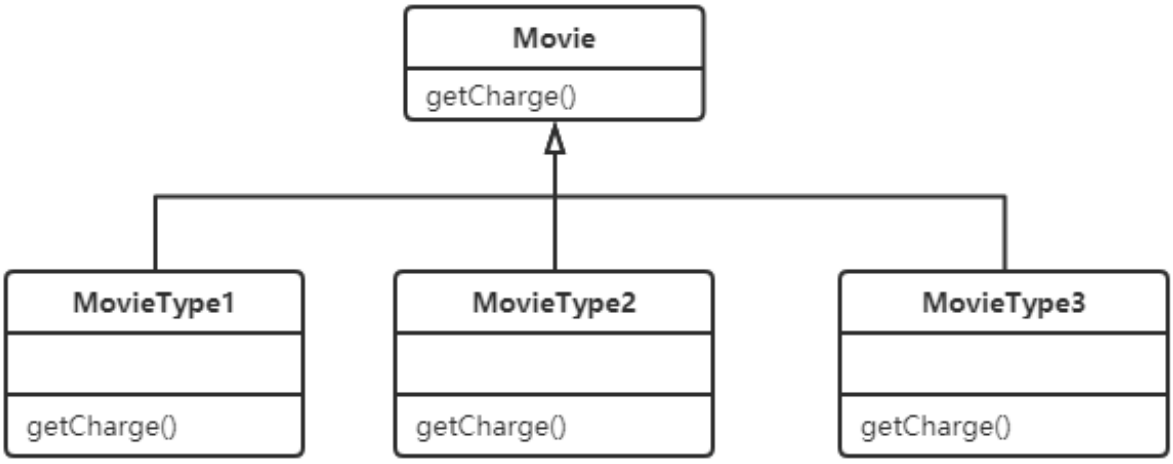
```
    private int type;

    Movie(int type) {
        this.type = type;
    }

    int getMovieType() {
        return type;
    }
}
```

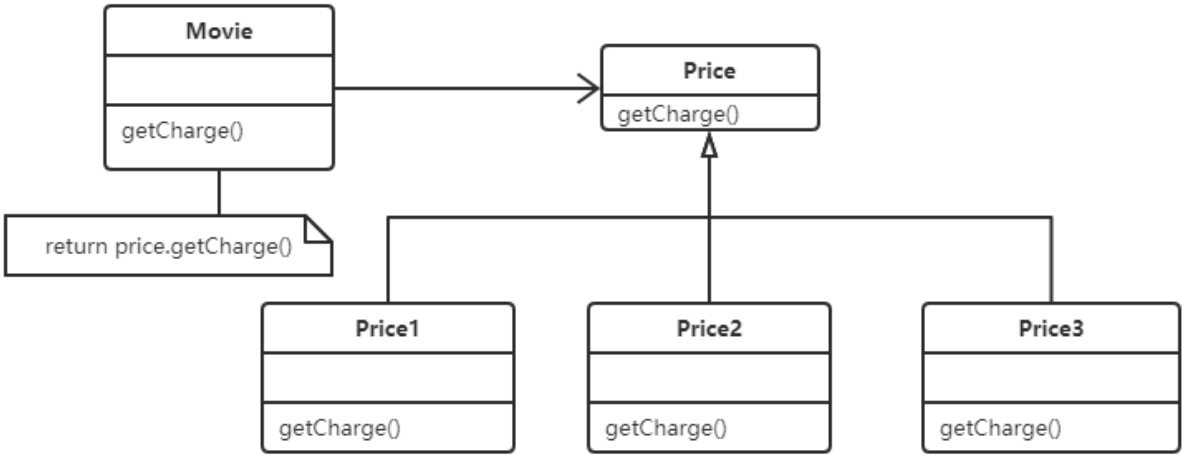
```
public class App {
    public static void main(String[] args) {
        Customer customer = new Customer();
        Rental rental1 = new Rental(1, new Movie(Movie.Type1));
        Rental rental2 = new Rental(2, new Movie(Movie.Type2));
        customer.addRental(rental1);
        customer.addRental(rental2);
        System.out.println(customer.getTotalCharge());
    }
}
```

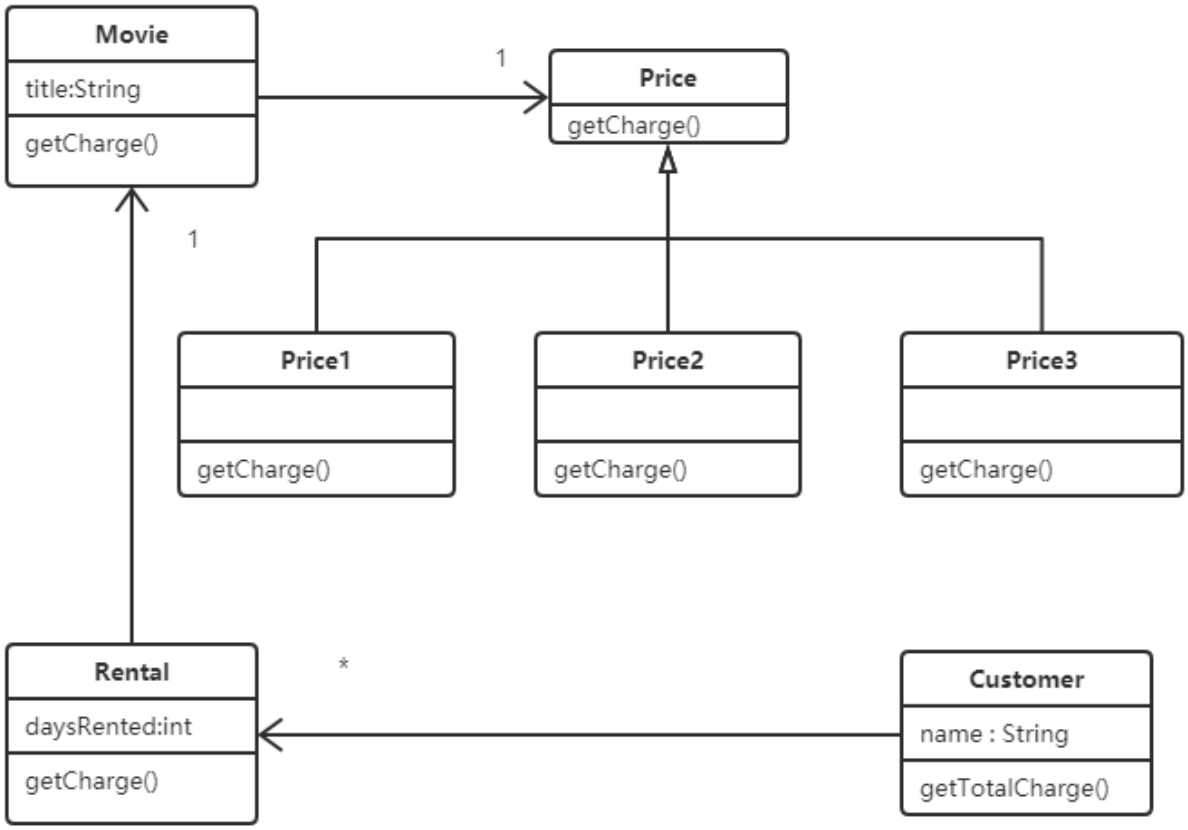
```
5
```

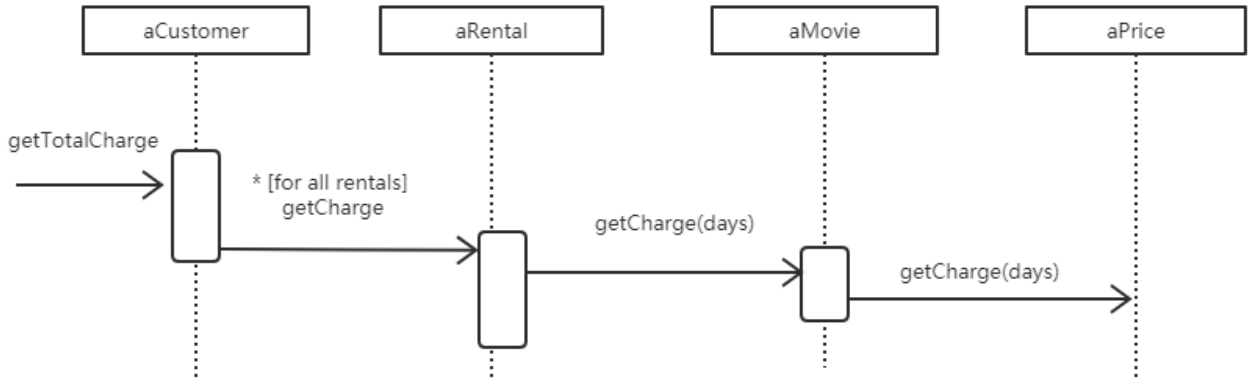
使用 switch 的准则是：只使用 switch 所在类的数据。解释如下：switch 使用的数据通常是一组相关的数据，例如 getTotalCharge() 代码使用了 Movie 的多种类别数据。当这组类别的数据发生改变时，例如增加 Movie 的类别或者修改一种 Movie 类别的计费方法，就需要修改 switch 代码。如果违反了准则，就会有多个地方的 switch 使用了这部分的数据，那么这些 swtich 都需要进行修改，这些代码可能遍布在各个地方，修改工作往往会很难进行。上面的实现违反了这一准则，因此需要重构。

以下是继承 Movie 的多态解决方案，这种方案可以解决上述的 switch 问题，因为每种电影类别的计费方式都被放到了对应 Movie 子类中，当变化发生时，只需要去修改对应子类中的代码即可。


Movie
getCharge()
MovieType1
getCharge()
MovieType2
getCharge()
MovieType3
getCharge()


Movie
getCharge()
return price.getCharge()
Price
getCharge()
Price1
getCharge()
Price2
getCharge()
Price3
getCharge()




```java
class Customer {
    private List<Rental> rentals = new ArrayList<>();

    void addRental(Rental rental) {
        rentals.add(rental);
    }

    double getTotalCharge() {
        double totalCharge = 0.0;
```

```
        for (Rental rental : rentals) {
            totalCharge += rental.getCharge();
        }
        return totalCharge;
    }
}
```

```
class Rental {
    private int daysRented;

    private Movie movie;

    Rental(int daysRented, Movie movie) {
        this.daysRented = daysRented;
        this.movie = movie;
    }

    double getCharge() {
        return daysRented * movie.getCharge();
    }
}
```

```
interface Price {
    double getCharge();
}
```

```
class Price1 implements Price {
    @Override
    public double getCharge() {
        return 1;
    }
}
```

```
class Price2 implements Price {
    @Override
    public double getCharge() {
        return 2;
    }
}
```

```
package imp2;
```

```
2.
3.  class Price3 implements Price {
4.      @Override
5.      public double getCharge() {
6.          return 3;
7.      }
8.  }
```

```
1.  class Movie {
2.
3.      private Price price;
4.
5.      Movie(Price price) {
6.          this.price = price;
7.      }
8.
9.      double getCharge() {
10.         return price.getCharge();
11.     }
12. }
```

```
1.  class App {
2.
3.      public static void main(String[] args) {
4.          Customer customer = new Customer();
5.          Rental rental1 = new Rental(1, new Movie(new Price1()));
6.          Rental rental2 = new Rental(2, new Movie(new Price2()));
7.          customer.addRental(rental1);
8.          customer.addRental(rental2);
9.          System.out.println(customer.getTotalCharge());
10.     }
11. }
```

# 二、重构原则

## 定义

重构是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。

# 为何重构

- 改进软件设计
- 使软件更容易理解
- 帮助找到 Bug
- 提高编程速度

# 三次法则

第一次做某件事时只管去做；第二次做类似事情时可以去做；第三次再做类似的事，就应该重构。

# 间接层与重构

计算机科学中的很多问题可以通过增加一个间接层来解决，间接层具有以下价值：

- 允许逻辑共享
- 分开解释意图和实现
- 隔离变化
- 封装条件逻辑

重构可以理解为在适当的位置插入间接层以及在不需要时移除间接层。

# 修改接口

如果重构手法改变了已发布的接口，就必须维护新旧两个接口。可以保留旧接口，让旧接口去调用新接口，并且使用 Java 提供的 @deprecation 将旧接口标记为弃用。

可见修改接口特别麻烦，因此除非真有必要，否则不要发布接口，并且不要过早发布接口。

## 何时不该重构

当现有代码过于混乱时，应当重写而不是重构。

一个折中的办法是，将代码封装成一个个组件，然后对各个组件做重写或者重构的决定。

## 重构与设计

软件开发无法预先设计，因为开发过程有很多变化发生，在最开始不可能都把所有情况考虑进去。

重构可以简化设计，重构在一个简单的设计上进行修修改改，当变化发生时，以一种灵活的方式去应对变化，进而带来更好的设计。

## 重构与性能

为了软代码更容易理解，重构可能会导致性能减低。

在编写代码时，不用对性能过多关注，只有在最后性能优化阶段再考虑性能问题。

应当只关注关键代码的性能，并且只有一小部分的代码是关键代码。

# 三、代码的坏味道

本章主要介绍一些不好的代码，也就是说这些代码应该被重构。

## 1. 重复代码

> Duplicated Code

同一个类的两个函数有相同表达式，则用 Extract Method 提取出重复代码；

两个互为兄弟的子类含有相同的表达式，先使用 Extract Method，然后把提取出来的函数 Pull Up Method 推入超类。

如果只是部分相同，用 Extract Method 分离出相似部分和差异部分，然后使用 Form Template Method 这种模板方法设计模式。

如果两个毫不相关的类出现重复代码，则使用 Extract Class 方法将重复代码提取到一个独立类中。

## 2. 过长函数

> Long Method

函数应该尽可能小，因为小函数具有解释能力、共享能力、选择能力。

分解长函数的原则：当需要用注释来说明一段代码时，就需要把这部分代码写入一个独立的函数中。

Extract Method 会把很多参数和临时变量都当做参数，可以用 Replace Temp with Query 消除临时变量，Introduce Parameter Object 和 Preserve Whole Object 可以将过长的参数列变得更简洁。

条件和循环语句往往也需要提取到新的函数中。

## 3. 过大的类

> Large Class

应该尽可能让一个类只做一件事，而过大的类做了过多事情，需要使用 Extract Class 或 Extract Subclass。

先确定客户端如何使用该类，然后运用 Extract Interface 为每一种使用方式提取出一个接口。

## 4. 过长的参数列表

> Long Parameter List

太长的参数列表往往会造成前后不一致，不易使用。

面向对象程序中，函数所需要的数据通常能在宿主类中找到。

## 5. 发散式变化

> Divergent Change

设计原则：一个类应该只有一个引起改变的原因。也就是说，针对某一外界变化所有相应的修改，都只应该发生在单一类中。

针对某种原因的变化，使用 Extract Class 将它提炼到一个类中。

## 6. 散弹式修改

> Shotgun Surgery

一个变化引起多个类修改。

使用 Move Method 和 Move Field 把所有需要修改的代码放到同一个类中。

## 7. 依恋情结

> Feature Envy

一个函数对某个类的兴趣高于对自己所处类的兴趣，通常是过多访问其它类的数据，

使用 Move Method 将它移到该去的地方，如果对多个类都有 Feature Envy，先用 Extract Method 提取出多个函数。

## 8. 数据泥团

> Data Clumps

有些数据经常一起出现，比如两个类具有相同的字段、许多函数有相同的参数，这些绑定在一起出现的数据应该拥有属于它们自己的对象。

使用 Extract Class 将它们放在一起。

## 9. 基本类型偏执

> Primitive Obsession

使用类往往比使用基本类型更好，使用 Replace Data Value with Object 将数据值替换为对象。

## 10. switch 惊悚现身

> Switch Statements

具体参见第一章的案例。

## 11. 平行继承体系

> Parallel Inheritance Hierarchies

每当为某个类增加一个子类，必须也为另一个类相应增加一个子类。

这种结果会带来一些重复性，消除重复性的一般策略：让一个继承体系的实例引用另一个继承体系的实例。

## 12. 冗余类

> Lazy Class

如果一个类没有做足够多的工作，就应该消失。

## 13. 夸夸其谈未来性

> Speculative Generality

有些内容是用来处理未来可能发生的变化，但是往往会造成系统难以理解和维护，并且预测未来可能发生的改变很可能和最开始的设想相反。因此，如果不是必要，就不要这么做。

## 14. 令人迷惑的暂时字段

> Temporary Field

某个字段仅为某种特定情况而设，这样的代码不易理解，因为通常认为对象在所有时候都需要它的所有字段。

把这种字段和特定情况的处理操作使用 Extract Class 提炼到一个独立类中。

## 15. 过度耦合的消息链

> Message Chains

一个对象请求另一个对象，然后再向后者请求另一个对象，然后...，这就是消息链。采用这种

方式，意味着客户代码将与对象间的关系紧密耦合。

改用函数链，用函数委托另一个对象来处理。

## 16. 中间人

> Middle Man

中间人负责处理委托给它的操作，如果一个类中有过多的函数都委托给其它类，那就是过度运用委托，应当 Remove Middle Man，直接与负责的对象打交道。

## 17. 狎昵关系

> Inappropriate Intimacy

两个类多于亲密，花费太多时间去探讨彼此的 private 成分。

## 18. 异曲同工的类

> Alernative Classes with Different Interfaces

两个函数做同一件事，却有着不同的签名。

使用 Rename Method 根据它们的用途重新命名。

## 19. 不完美的类库

> Incomplete Library Class

类库的设计者不可能设计出完美的类库，当我们需要对类库进行一些修改时，可以使用以下两

种方法：如果只是修改一两个函数，使用 Introduce Foreign Method；如果要添加一大堆额外行为，使用 Introduce Local Extension。

## 20. 幼稚的数据类

> Data Class

它只拥有一些数据字段，以及用于访问这些字段的函数，除此之外一无长物。

找出字段使用的地方，然后把相应的操作移到 Data Class 中。

## 21. 被拒绝的馈赠

> Refused Bequest

子类不想继承超类的所有函数和数据。

为子类新建一个兄弟类，不需要的函数或数据使用 Push Down Method 和 Push Down Field 下推给那个兄弟。

## 22. 过多的注释

> Comments

使用 Extract Method 提炼出需要注释的部分，然后用函数名来解释函数的行为。

# 四、构筑测试体系

Java 可以使用 Junit 进行单元测试。

测试应该能够完全自动化，并能检查测试的结果。

小步修改，频繁测试。

单元测试的对象是类的方法，而功能测是以客户的角度保证软件正常运行。

应当集中测试可能出错的边界条件。

# 五、重新组织函数

## 1. 提炼函数

> Extract Method

将这段代码放进一个独立函数中，并让函数名称解释该函数的用途。

## 2. 内联函数

> Inline Method

一个函数的本体与名称同样清楚易懂。

在函数调用点插入函数本体，然后移除该函数。

## 3. 内联临时变量

> Inline Temp

一个临时变量，只被简单表达式赋值一次，而它妨碍了其它重构手法。

将所有对该变量的引用替换为对它赋值的那个表达式自身。

```
double basePrice = anOrder.basePrice();
return basePrice > 1000;
```

```
return anOrder.basePrice() > 1000;
```

# 4. 以查询取代临时变量

> Replace Temp with Query

以临时变量保存某一表达式的运算结果，将这个表达式提炼到一个独立函数中，将所有对临时变量的引用点替换为对新函数的调用。

Replace Temp with Query 往往是 Extract Method 之前必不可少的一个步骤，因为局部变量会使代码难以提炼。

```
double basePrice = quantity * itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;

// ...
double basePrice(){
    return quantity * itemPrice;
}
```

# 5. 引起解释变量

> Introduce Explaining Variable

将复杂表达式（或其中一部分）的结果放进一个临时变量， 以此变量名称来解释表达式用途。

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
  (browser.toUpperCase().indexOf("IE") > -1) &&
  wasInitialized() && resize > 0) {
    // do something
}
```

```
final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrower = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOS && isIEBrower && wasInitialized() && wasResized) {
    // do something
}
```

# 6. 分解临时变量

> Split Temporary Variable

某个临时变量被赋值超过一次，它既不是循环变量，也不是用于收集计算结果。

针对每次赋值，创造一个独立、对应的临时变量，每个临时变量只承担一个责任。

# 7. 移除对参数的赋值

> Remove Assigments to Parameters

以一个临时变量取代对该参数的赋值。

```
int discount (int inputVal, int quentity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    ...
```

```
}
```

```
int discount (int inputVal, int quentity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    ...
}
```

## 8. 以函数对象取代函数

> Replace Method with Method Object

当对一个大型函数采用 Extract Method 时，由于包含了局部变量使得很难进行该操作。

将这个函数放进一个单独对象中，如此一来局部变量就成了对象内的字段。然后可以在同一个对象中将这个大型函数分解为多个小型函数。

## 9. 替换算法

> Subsititute Algorithn

# 六、在对象之间搬移特性

## 1. 搬移函数

> Move Method

类中的某个函数与另一个类进行更多交流：调用后者或者被后者调用。

将这个函数搬移到另一个类中。

## 2. 搬移字段

> Move Field

类中的某个字段被另一个类更多地用到，这里的用到是指调用取值设值函数，应当把该字段移到另一个类中。

## 3. 提炼类

> Extract Class

某个类做了应当由两个类做的事。

应当建立一个新类，将相关的字段和函数从旧类搬移到新类。

## 4. 将类内联化

> Inline Class

与 Extract Class 相反。

## 5. 隐藏委托关系

> Hide Delegate

建立所需的函数，隐藏委托关系。

```
class Person {
    Department department;

    public Department getDepartment() {
```

```
        return department;
    }
}

class Department {
    private Person manager;

    public Person getManager() {
        return manager;
    }
}
```

如果客户希望知道某人的经理是谁，必须获得 Department 对象，这样就对客户揭露了 Department 的工作原理。

```
Person manager = john.getDepartment().getManager();
```

通过为 Peron 建立一个函数来隐藏这种委托关系。

```
public Person getManager() {
    return department.getManager();
}
```

# 6. 移除中间人

> Remove Middle Man

与 Hide Delegate 相反，本方法需要移除委托函数，让客户直接调用委托类。

Hide Delegate 有很大好处，但是它的代价是：每当客户要使用受托类的新特性时，就必须在服务器端添加一个简单的委托函数。随着受委托的特性越来越多，服务器类完全变成了一个“中间人”。

# 7. 引入外加函数

> Introduce Foreign Method

需要为提供服务的类添加一个函数，但是无法修改这个类。

可以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的实例，让客户类组合服务器实例。

## 8. 引入本地扩展

> Introduce Local Extension

和 Introduce Foreign Method 目的一样，但是 Introduce Local Extension 通过建立新的类来实现。有两种方式：子类或者包装类，子类就是通过继承实现，包装类就是通过组合实现。

# 七、重新组织数据

## 1. 自封装字段

> Self Encapsulate Field

为字段建立取值/设值函数，并用这些函数来访问字段。只有当子类想访问超类的一个字段，又想在子类中将对这个字段访问改为一个计算后的值，才使用这种方式，否则直接访问字段的方式简洁明了。

## 2. 以对象取代数据值

> Replace Data Value with Object

在开发初期，往往会用简单的数据项表示简单的情况，但是随着开发的进行，一些简单数据项

会具有一些特殊行为。比如一开始会把电话号码存成字符串，但是随后发现电话号码需要“格式化”、“抽取区号”之类的特殊行为。

## 3. 将值对象改成引用对象

> Change Value to Reference

将彼此相等的实例替换为同一个对象。这就要用一个工厂来创建这种唯一对象，工厂类中需要保留一份已经创建对象的列表，当要创建一个对象时，先查找这份列表中是否已经存在该对象，如果存在，则返回列表中的这个对象；否则，新建一个对象，添加到列表中，并返回该对象。

## 4. 将引用对象改为值对象

> Change Reference to value

以 Change Value to Reference 相反。值对象有个非常重要的特性：它是不可变的，不可变表示如果要改变这个对象，必须用一个新的对象来替换旧对象，而不是修改旧对象。

需要为值对象实现 equals() 和 hashCode() 方法。

## 5. 以对象取代数组

> Replace Array with Object

有一个数组，其中的元素各自代表不同的东西。

以对象替换数组，对于数组中的每个元素，以一个字段来表示，这样方便操作，也更容易理解。

# 6. 赋值被监视数据

> Duplicate Observed Data

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。

将该数据赋值到一个领域对象中，建立一个 Oberver 模式，用于同步领域对象和 GUI 对象内的重复数据。

# 7. 将单向关联改为双向关联

> Change Unidirectional Association to Bidirectional

当两个类都需要对方的特性时，可以使用双向关联。

有两个类，分别为订单 Order 和客户 Customer，Order 引用了 Customer，Customer 也需要引用 Order 来查看其所有订单详情。

```
class Order {
    private Customer customer;
    public void setCustomer(Customer customer) {
        if (this.customer != null)
            this.customer.removeOrder(this);
        this.customer = customer;
        this.customer.add(this);
    }
}
```

```
class Curstomer {
    private Set<Order> orders = new HashSet<>();
    public void removeOrder(Order order) {
        orders.remove(order);
    }
```

```
6.      public void addOrder(Order order) {
7.          orders.add(order);
8.      }
9.  }
```

注意到，这里让 Curstomer 类来控制关联关系。有以下原则来决定哪个类来控制关联关系：如果某个对象是组成另一个对象的部件，那么由后者负责控制关联关系；如果是一对多关系，则由单一引用那一方来控制关联关系。

## 8. 将双向关联改为单向关联

> Change Bidirectional Association to Unidirectional

和 Change Unidirectional Association to Bidirectiona 为反操作。

双向关联维护成本高，并且也不易于理解。大量的双向连接很容易造成“僵尸对象”：某个对象本身已经死亡了，却保留在系统中，因为它的引用还没有全部完全清除。

## 9. 以字面常量取代魔法数

> Replace Magic Number with Symbolic Constant

创建一个常量，根据其意义为它命名，并将字面常量换为这个常量。

## 10. 封装字段

> Encapsulate Field

public 字段应当改为 private，并提供相应的访问函数。

## 11. 封装集合

> Encapsulate Collection

函数返回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。如果函数返回集合自身，会让用户得以修改集合内容而集合拥有者却一无所知。

# 12. 以数据类取代记录

> Replace Record with Data Class

# 13. 以类取代类型码

> Replace Type Code with Class

类中有一个数值类型码，但它并不影响类的行为，就用一个新类替换该数值类型码。如果类型码出现在 switch 语句中，需要使用 Replace Conditional with Polymorphism 去掉 switch，首先必须运用 Replace Type Code with Subcalss 或 Replace Type Code with State/Strategy 去掉类型码。

# 14. 以子类取代类型码

> Replace Type Code with Subcalsses

有一个不可变的类型码，它会影响类的行为，以子类取代这个类型码。

## 15. 以 State/Strategy 取代类型码

> Replace Type Code with State/Strategy

有一个可变的类型码，它会影响类的行为，以状态对象取代类型码。

和 Replace Type Code with Subcalsses 的区别是 Replace Type Code with State/Strategy 的类型码是动态可变的，前者通过继承的方式来实现，后者通过组合的方式来实现。因为类型码可变，如果通过继承的方式，一旦一个对象的类型码改变，那么就要改变用新的对象来取代旧对象，而客户端难以改变新的对象。但是通过组合的方式，改变引用的状态类是很容易的。

## 16. 以字段取代子类

> Replace Subclass with Fields

各个子类的唯一差别只在“返回常量数据”的函数上。

# 八、简化条件表达式

## 1. 分解条件表达式

> Decompose Conditional

对于一个复杂的条件语句，可以从 if、then、else 三个段落中分别提炼出独立函数。

```
if (data.befor(SUMMER_START) || data.after(SUMMER_END))
    charge = quantity * winterRate + winterServiceCharge;
else
    charge = quantity * summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge(quantity);
```

# 2. 合并条件表达式

> Consolidate Conditional Expression

有一系列条件测试，都得到相同结果。

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。

```
double disabilityAmount() {
    if (seniority < 2) return 0;
    if (monthsDisabled > 12 ) return 0;
    if (isPartTime) return 0;
    // ...
}
```

```
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;
    // ...
}
```

# 3. 合并重复的条件片段

> Consolidate Duplicate Conditional Fragments

在条件表达式的每个分支上有着相同的一段代码。

将这段重复代码搬移到条件表达式之外。

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

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

# 4. 移除控制标记

> Remove Control Flag

在一系列布尔表达式中，某个变量带有“控制标记”的作用。

用 break 语句或 return 语句来取代控制标记。

# 5. 以卫语句取代嵌套条件表达式

> Replace Nested Conditional with Guard Clauses

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）。

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。第二种形式则是：条件表达式提供的答案中只有一种是正常行为，其他都是不常见的情况，可以使用卫语句表现所有特殊情况。

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

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

# 6. 以多态取代条件表达式

> Replace Conditional with Polymorphism

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。需要先使用 Replace Type Code with Subclass 或 Replace Type Code with State/Strategy 来建立继承结果。

```
double getSpeed() {
    switch (type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed()- getLoadFactor()* numberOfCoconuts;
```

```
            case NORWEGIAN_BLUE:
                return isNailed ? 0 : getBaseSpeed(voltage);
        }
        throw new RuntimeException("Should be unreachable");
    }
```

# 7. 引入 Null 对象

> Introduce Null Object

将 null 值替换为 null 对象。这样做的好处在于，不需要询问对象是否为空，直接调用就行。

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

# 8. 引入断言

> Introduce Assertion

以断言明确表现某种假设。断言只能用于开发过程中，产品代码中不会有断言。

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (expenseLimit != NULL_EXPENSE) ? expenseLimit :  primaryProj
ect.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue (expenseLimit != NULL_EXPENSE || primaryProject != nu
ll);
    return (expenseLimit != NULL_EXPENSE) ? expenseLimit :  primaryProj
ect.getMemberExpenseLimit();
```

```
4.  }
```

# 九、简化函数调用

## 1. 函数改名

> Rename Method

使函数名能解释函数的用途。

## 2. 添加参数

> Add Parameter

使函数不需要通过调用获得某个信息。

## 3. 移除参数

> Remove Parameter

与 Add Parameter 相反，改用调用的方式来获得某个信息。

## 4. 将查询函数和修改函数分离

> Separate Query from Modifier

某个函数即返回对象状态值，又修改对象状态。

应当建立两个不同的函数，其中一个负责查询，另一个负责修改。任何有返回值的函数，都不

应该有看得到的副作用。

```
getTotalOutstandingAndSetReadyForSummaries();
```

```
getTotalOutstanding();
setReadyForSummaries();
```

# 5. 令函数携带参数

> Parameterize Method

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。

建立单一函数，以参数表达那些不同的值。

```
fivePercentRaise();
tenPercentRaise();
```

```
raise(percentage);
```

# 6. 以明确函数取代参数

> Replace Parameter with Explicit Methods

有一个函数，完全取决于参数值而采取不同行为。

针对该参数的每一个可能值，建立一个独立函数。

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

```
    if (name.equals("width")){
        width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg){
    height = arg;
}
void setWidth(int arg){
    width = arg;
}
```

# 7. 保持对象完整

> Preserve Whole Object

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。

改为传递整个对象。

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

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

# 8. 以函数取代参数

> Replace Parameter with Methods

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。

让参数接收者去除该项参数，而是直接调用前一个函数。

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

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

# 9. 引入参数对象

> Introduce Parameter Object

某些参数总是很自然地同时出现，这些参数就是 Data Clumps。

以一个对象取代这些参数。

# 10. 移除设值函数

> Remove Setting Method

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。

去掉该字段的所有设值函数，并将该字段设为 final。

# 11. 隐藏函数

> Hide Method

有一个函数，从来没有被其他任何类用到。

将这个函数修改为 private。

## 12. 以工厂函数取代构造函数

> Replace Constructor with Factory Method

希望在创建对象时不仅仅是做简单的建构动作。

将构造函数替换为工厂函数。

## 13. 封装向下转型

> Encapsulate Downcast

某个函数返回的对象，需要由函数调用者执行向下转型（downcast）。

将向下转型动作移到函数中。

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

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

## 14. 以异常取代错误码

> Replace Error Code with Exception

某个函数返回一个特定的代码，用以表示某种错误情况。

改用异常，异常将普通程序和错误处理分开，使代码更容易理解。

## 15. 以测试取代异常

> Replace Exception with Test

面对一个调用者可以预先检查的条件，你抛出了一个异常。

修改调用者，使它在调用函数之前先做检查。

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

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

# 十、处理概括关系

## 1. 字段上移

> Pull Up Field

两个子类拥有相同的字段。

将该字段移至超类。

## 2. 函数上移

> Pull Up Method

有些函数，在各个子类中产生完全相同的结果。

将该函数移至超类。

## 3. 构造函数本体上移

> Pull Up Constructor Body

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。

在超类中新建一个构造函数，并在子类构造函数中调用它。

```
class Manager extends Employee...

public Manager(String name, String id, int grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
}
```

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

## 4. 函数下移

> Push Down Method

超类中的某个函数只与部分子类有关。

将这个函数移到相关的那些子类去。

## 5. 字段下移

> Push Down Field

超类中的某个字段只被部分子类用到。

将这个字段移到需要它的那些子类去。

## 6. 提炼子类

> Extract Subclass

类中的某些特性只被某些实例用到。

新建一个子类，将上面所说的那一部分特性移到子类中。

## 7. 提炼超类

> Extract Superclass

两个类有相似特性。

为这两个类建立一个超类，将相同特性移至超类。

## 8. 提炼接口

> Extract Interface

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。

将相同的子集提炼到一个独立接口中。

## 9. 折叠继承体系

> Collapse Hierarchy

超类和子类之间无太大区别。

将它们合为一体。

## 10. 塑造模板函数

> Form Template Method

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。

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。(模板方法模式)

## 11. 以委托取代继承

> Replace Inheritance with Delegation

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。

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。

## 12. 以继承取代委托

> Replace Delegation with Inheritance

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。

让委托类继承受托类。

# 参考资料


- MartinFowler, 福勒, 贝克, 等. 重构: 改善既有代码的设计 [M]. 电子工业出版社, 2011.


---

github: https://github.com/sjsdfg/Interview-Notebook-PDF