Lecture7 重构 Refactoring

1. 介绍

什么是重构

语义保留式的程序转换

- 对程序内部结构所做的改变，而不修改程序的可观察行为
  - 更容易理解
  - 更简单修改

重构模式 Refactoring Patterns

- 在代码编写完成后改进设计
  - 这似乎有点奇怪，因为我们通常先设计然后编写代码
  - 重构通常需要进行较小的更改，但会产生较大的累积效应
- 非正式的（它们不可能是正式的，因为无法确定程序的等价性）
- 与设计模式相似，定义了一些共享的词汇

为什么要重构

- 代码在维护过程中会退化
- 代码从一开始就写得很糟糕

推荐参考书

- Bad Smells：http://blog.codinghorror.com/code-smells/
- 重构的分类：http://www.refactoring.com/catalog/

2. 风格问题 Style Smells

注释 Comments

- 清晰的注释和令人费解的注释存在一条界限
- 确保你的注释是真正需要的，如果可以的话，重构一些代码，这样就不需要一些注释了
- 大量的注释可能意味着糟糕的代码

命名 Naming

- 避免在方法和变量的命名中提到与类有关的信息，否则如果你改变了类，你还需要重写这些名字
- 确保变量和方法的命名简洁地描述了目的
- 选择一个标准的命名方式，确保一些模拟行为的函数有模拟行为的名称
  - 如果你可以调用 `open()`，那也应该能 `close()`

没用的代码 Dead code

- 删了
- 使用版本控制

3. 过长 Too Long

长方法 Long Method

- 两段长方法更可能出现重复的逻辑
    - 如果你不理解一个很长的方法，把它分解成更小的、命名良好的方法可以帮助你
- 具有较小方法的系统往往更容易扩展和维护
- 在其他条件相同的情况下，更短的方法更容易阅读，更容易理解，更容易解决问题

潜在重构方法

- 提取方法 Extract method

长类 Long Class

- 这种情况经常发生在我们在仔细设计或设计原型之前编码，然后继续构建的时候
- 非常多的实例变量
    - 一个类要做的太多了
    - 类有太多的责任

潜在重构方法

- 提取类 Extract class
- 提取子类 Extract subclass
- 观察者模式 Observer

长参数列表 Long Parameter List

- 很长的参数列表（在过程式编程中很常见）很可能是易变的 `volatile`
- 考虑哪些参数是必须的
    - 如果需要，将其余的留给对象跟踪

潜在重构方法

将参数替换成方法 Replace parameter with method

对象调用方法，然后将结果作为方法的参数传递

- 接收方也可以调用它

为什么要间接这么多？移除多余的参数，让接收者调用方法

```
public double getPrice(){
    int basePrice = quality * itemPrice;
    int discountLevel;
    if(quantity > 100) discountLevel = 2:
    else diecountLevel = 1;
    double finalPrice = discountedPrice(basePrice, discountLevel);
    return finalPrice;
}

private double discountedPrice(int basePrice, int discountLevel){
    if(discountLevel == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

可以修改为

```
public double getPrice(){
    int basePrice = quality * itemPrice;
    double finalPrice = discountedPrice(basePrice);
    return finalPrice;
}

private int getDiscountLevel(){
    int discountLevel;
    if(quantity > 100) discountLevel = 2:
    else diecountLevel = 1;
    return discountLevel;
}

private double discountedPrice(int basePrice){
    if(getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

继续修改

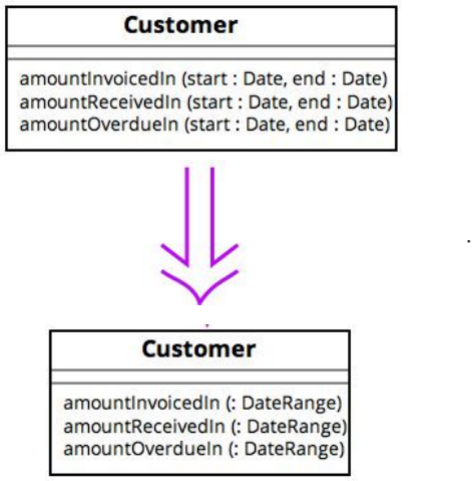
```
public double getPrice(){
    return discountedPrice();
}

private int getDiscountLevel(){
    int discountLevel;
    if(quantity > 100) discountLevel = 2:
    else diecountLevel = 1;
    return discountLevel;
}

```

```
12  private double discountedPrice(){
13      if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
14      else return getBasePrice() * 0.05;
15  }
16
17  private double getBasePrice(){
18      return quantity * itemPrice;
19
```

引入参数对象 Introduce parameter object

如果有一组参数它们经常要一起传入使用

- 把它们替换成一个对象



维护整个对象 Preserve whole object

- 从一个对象中获得一堆值，然后将这些对象一起传递给另一个方法
    - 也许你可以把整个对象都传过去

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

可修改为

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

4. Code Smell

重复的代码 Duplicated Code

重复的情况

- 同一个类中的两个方法中的相同表达式
- 在兄弟类的两个方法中使用相同的表达式
- 在两个不相关的类中使用相同的表达式

明确和微妙的重复

- 明显的：完全相同的代码
- 微妙的：看起来不同但是实际上一样

潜在的重构方法

- 提取方法 Extract Method
- 提取类 Extract Class
- 模板方法模式 Template Method Pattern
- 策略模式 Strategy Pattern

提取方法 Extract Method

当某个代码块的代码片段可以经常形成一个组的时候

- 将片段转换为一个方法，其名称解释块的用途

```
void printOwing(){
    printBanner();
    // print details
    System.out.println("name: " + _name);
    System.out.println("amount " + getOutstanding());
}
```

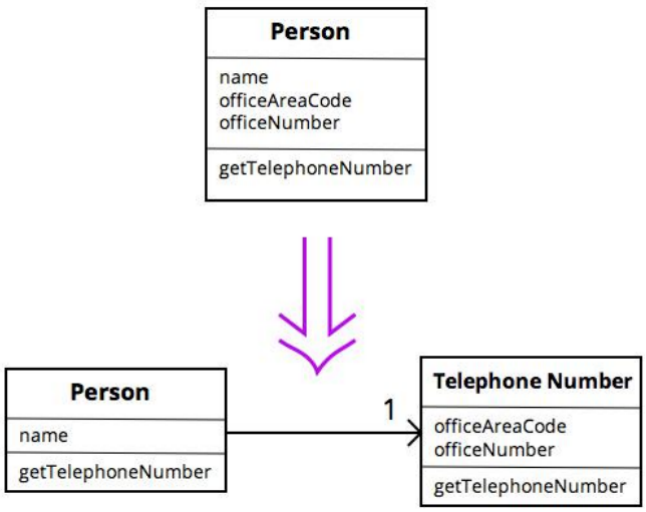
重构为：

```
void printOwing(){
    printBanner();
    printDetails(getOutstanding())
}

void printDetails(double outstanding){
    System.out.println("name: " + _name);
    System.out.println("amount " + outstanding);
}
```

提取类 Extract Class

当一个类完成的工作可以被两个不同的类完成时

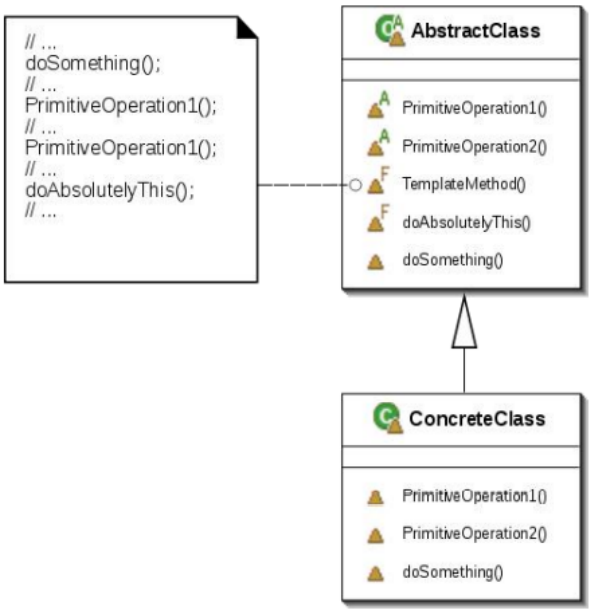
- 创建一个新类，并将相关字段和方法从旧类移动到新的类




```
class Person{
    private String name;
    private TelephoneNumber officeTelephone = new TelephoneNumber("86","123456");

    public String getName(){
        return name;
    }

    public String getTelephoneNumber(){
        return officeTelephone.getTelephoneNumber();
    }
}

class TelephoneNumber{
    private String number;
    private String areacode;
    public String getTelephoneNumber(){
        return ("(" + areacode + ")" + number);
    }
    public TelephoneNumber(String num, String area){
        number = num;
        areacode = area;
    }
}
```

模板方法模式 Template Method Pattern

- 模板方法描述了一个方法的骨架行为
  - 将一些子步骤延迟到子类
- 通过定义包含模板方法的“基本操作”，子类提供了不同的行为



多样的变化 Divergent Change

通常会因为不同的原因以不同的方式改变一个特定的类

- 将不同的职责分开减少了一个变化对不同职能产生负面影响的机会
- 例如，在类 X 中
  - 当我们每次换一个新的数据库时，`mA()`，`mB()`，`mC()` 就会改变
  - 当我们每次添加一个新的金融工具时，`mD()`，`mE()`，`mF()` 就会改变

潜在重构方法

- 提取类 Extract Class

短枪手术 Shotgun Surgery

与**多样的变化**相反，通常是一个改变可能会影响到多个类，经常会在很多个类中做出很多小的改变

- 很容易忽略掉一个关键的改变

特例

- 并行继承层次结构 —— 每次你创建一个类的子类时，你必须创建另一个类的子类

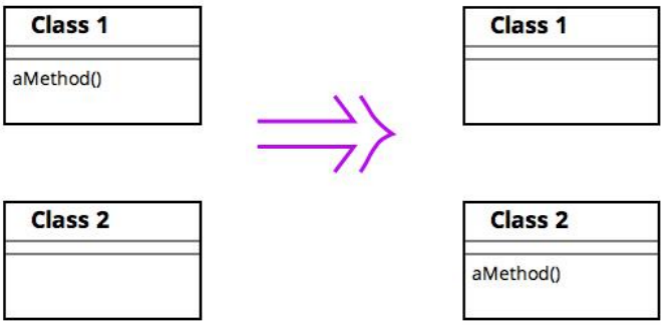
潜在的重构方法

- 移动方法 Move Method
- 移动字段 Move field
- 内部类 Inline class

移动方法 Move Method

一个方法更可能会被其它的类调用，而不是被它本身所在的类所使用

- 移动它，在它使用最多的类中创建一个具有相似主体的新方法
- 将旧方法转换为一个简单的委托或同时删除它




```
class Project {
  Person[] participants;
}

class Person {
  int id;
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
          if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```

```
class Project {
  Person[] participants;
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
          if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
}

... if (p.participate(x)) ...
```

特征嫉妒 Feature Envy

类中的方法似乎对其他类的内部结构比对自己的更感兴趣

- 最常见的嫉妒对象是数据
- 一个类反复调用其他类的 getter 和 setter 方法

潜在的重构方法

- 提取方法 Extract method
- 移动方法 Move method
- 移动字段 Move field

策略模式是一种例外，它刻意调用了别的类的 getter 和 setter 方法，为的是解决一些更大的问题

数据泥团 Data Clumps

一团团聚集在一起的数据应该被做成它们自己的对象

- 在好几个不同的类的字段，参数总是被连在一起

潜在的重构方法

- 提取类 Extract class
- 维护整个对象 Preserve whole object
- 引入参数对象 Introduce parameter object

基本数据类型的痴迷 Primitive Obsession

- 老一辈不喜欢用一些小的对象
- 反而会导致过分强调基本数据类型（例如，字符串、数组、整数等）
- 类通常比原语提供了一种更简单、更自然的方法来直接建模
    - 更高层次的抽象阐明代码

潜在的重构方法

- 将数据值替换为对象 Replace data value(s) with object
- 将类型码替换为类 Replace type code with class
- 用状态替换类型码 Replace type code with state/stratrgy

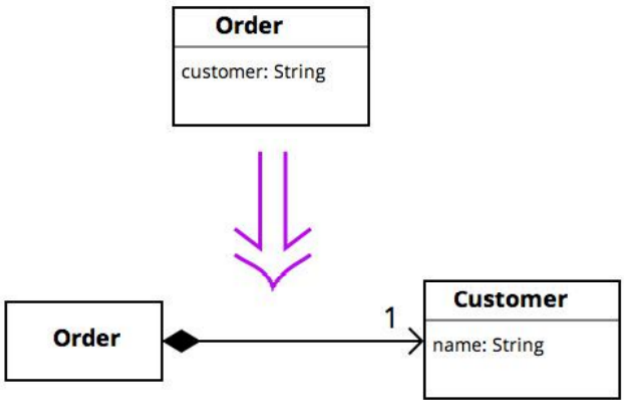
将数据值替换为对象 Replace data value(s) with object

有一个需要额外数据或行为的数据项

- 实际上，尽量不要从基本类型开始，然后添加越来越多概念性（但不是具体的）关联的基本类型

相反，将数据项转变成一个对象

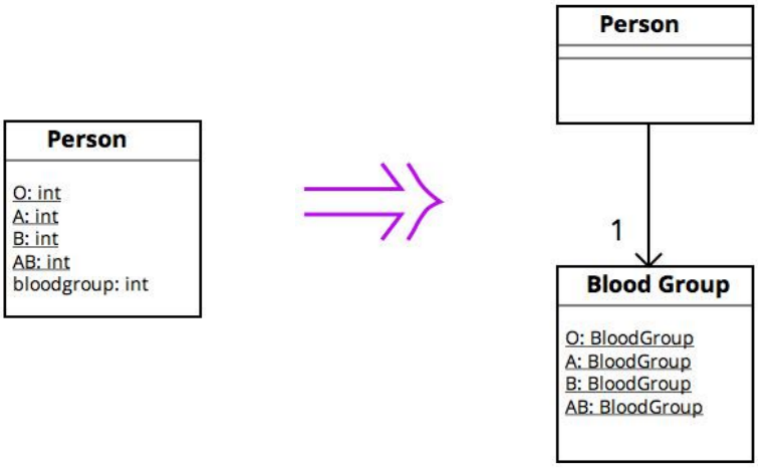
.



将类型码替换为类 Replace type code with class

类具有不影响其行为的（数字）类型代码

- 将数字替换成一个新的类




```
class Person{
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 1;
    public static final int AB = 1;

    private int _bloodGroup;

    public Person(int bloodGroup){
        _bloodGroup = bloodGroup;
    }

    public void setBloodGroup(int arg){
        _bloodGroup = arg;
    }

    public int getBloodGroup(){
        return _bloodGroup;
    }
}
```

重构代码

```
class BloodGroup{
    public static final BloodGroup O = new BloodGroup(0);
    public static final BloodGroup A = new BloodGroup(1);
    public static final BloodGroup B = new BloodGroup(2);
    public static final BloodGroup AB = new BloodGroup(3);
    private static final BloodGroup [] _values = {O,A,B,AB};
    private final int _code;
    private BloodGroup(int code){
        _code = code;
    }

    public int getCode(){
        return _code;
```

```
    }

    public static BloodGroup code(int arg){
        return _values[arg];
    }
}
```

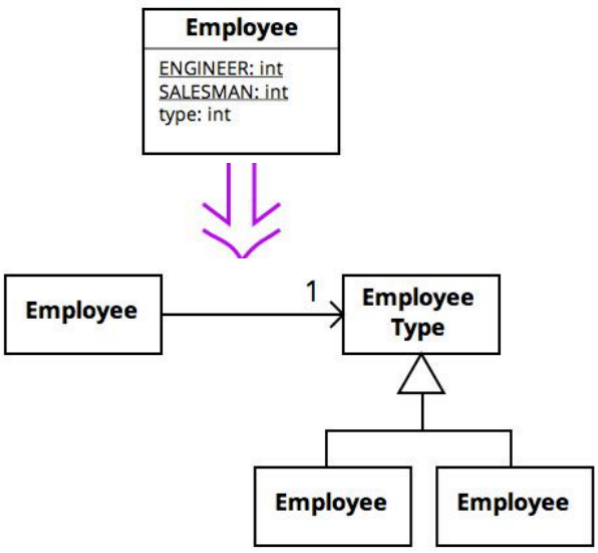
```
class Person{
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();

    private BloodGroup _bloodGroup;

    public Person (int bloodGroup) {
        _bloodGroup = BloodGroup.code(bloodGroup);
    }
    public int getBloodGroup() {
        return _bloodGroup.getCode();
    }
    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code (arg);
    }
}
```

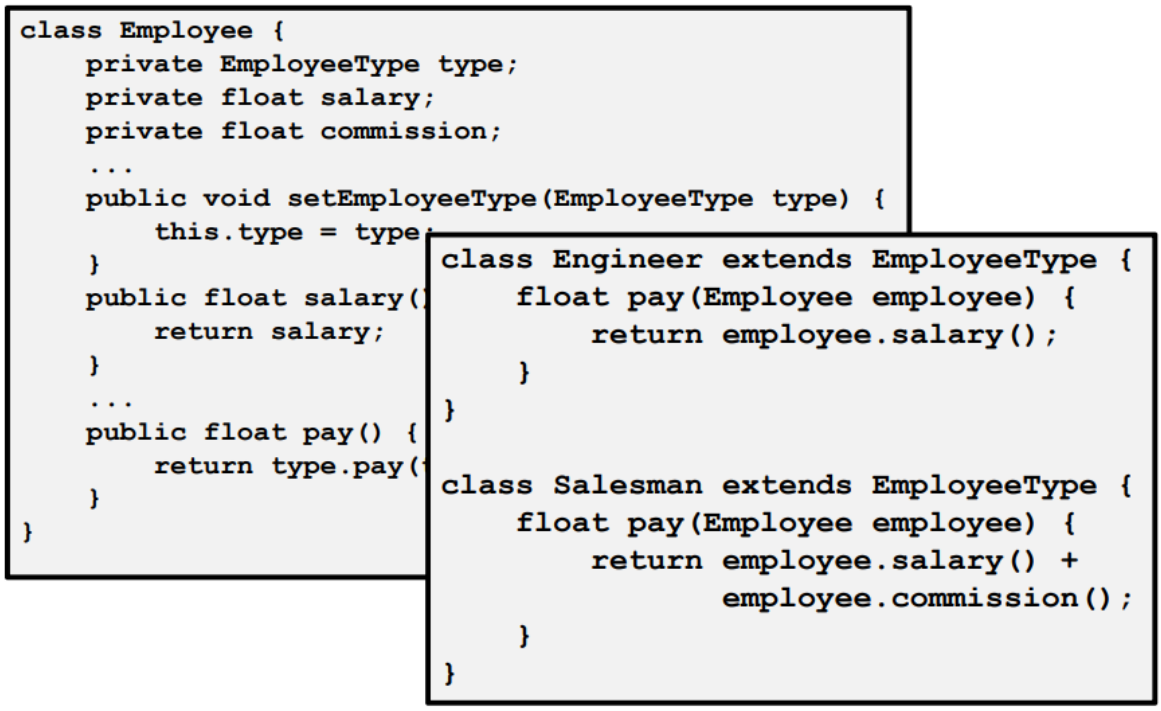
还能对 `BloodGroup` 做出什么改进吗?

用状态替换类型码 Replace type code with state/stratrgy

有一个影响类行为的类型代码，但你不能使用子类

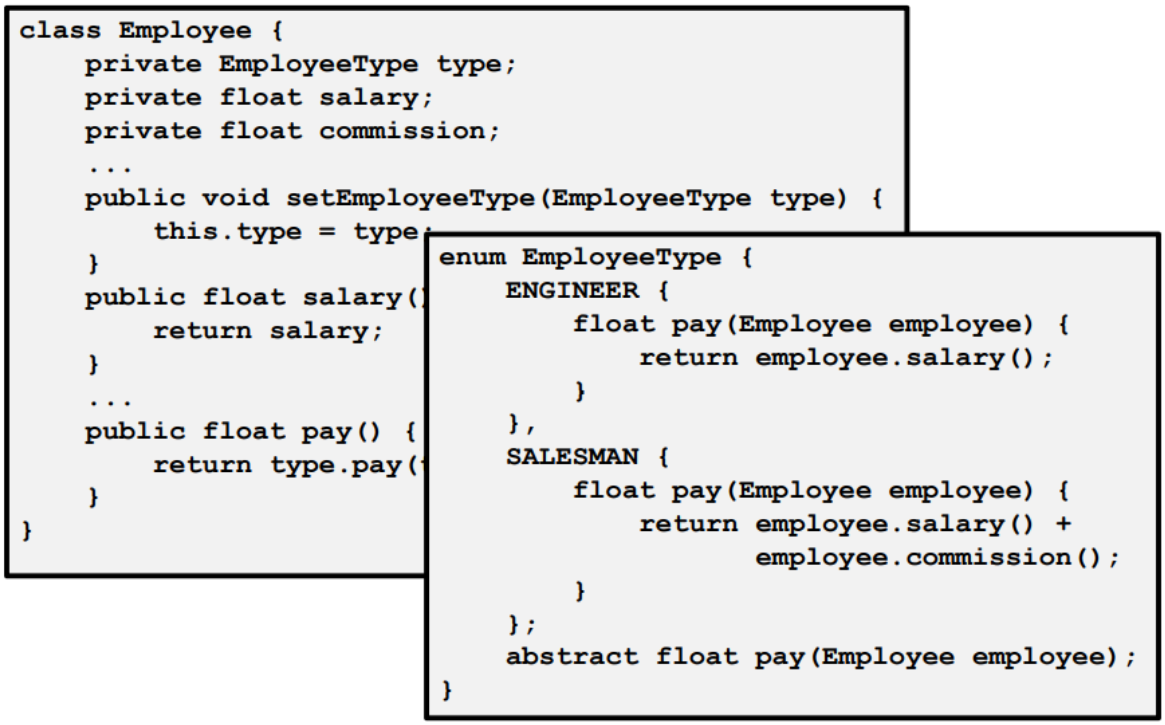
- 将类型码替换为状态对象





```
class Employee {
    private EmployeeType type;
    private float salary;
    private float commission;
    ...
    public void setEmployeeType(EmployeeType type) {
        this.type = type;
    }
    public float salary() {
        return salary;
    }
    ...
    public float pay() {
        return type.pay(this);
    }
}
```

```
class Engineer extends EmployeeType {
    float pay(Employee employee) {
        return employee.salary();
    }
}

class Salesman extends EmployeeType {
    float pay(Employee employee) {
        return employee.salary() +
               employee.commission();
    }
}
```



```
class Employee {
    private EmployeeType type;
    private float salary;
    private float commission;
    ...
    public void setEmployeeType(EmployeeType type) {
        this.type = type;
    }
    public float salary() {
        return salary;
    }
    ...
    public float pay() {
        return type.pay(this);
    }
}
```

```
enum EmployeeType {
    ENGINEER {
        float pay(Employee employee) {
            return employee.salary();
        }
    },
    SALESMAN {
        float pay(Employee employee) {
            return employee.salary() +
                   employee.commission();
        }
    };
    abstract float pay(Employee employee);
}
```

.

Switch 语句

Switch 语句经常在系统中重复出现

- 表示缺乏 OO 风格和多态性使用不足
- 特殊情况：根据对象的类型选择不同的行为的条件

潜在的重构方法

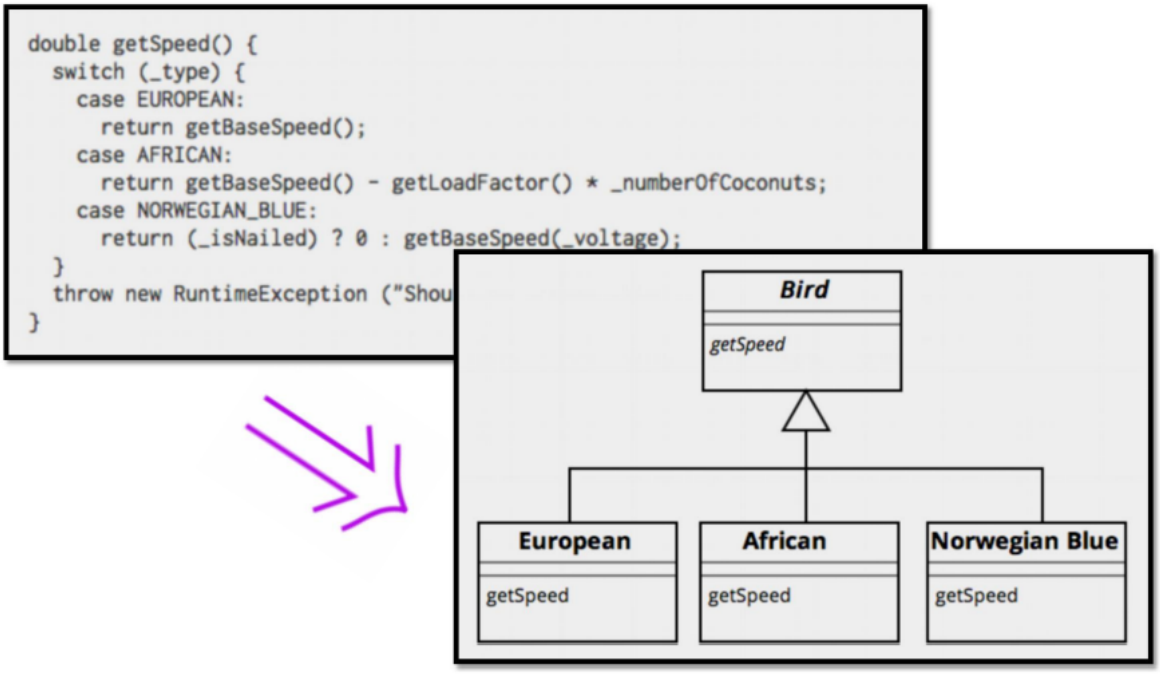
- 提取方法 Extract method
- 移动方法 Move method
- 将类型码替换成子类 Replace type code with subclasses

- 将类型码替换成状态类 Replace type code with state / strategy
- 将条件语句替换成多态 Replace conditional with polymorphism

将条件语句替换成多态 Replace conditional with polymorphism

根据对象的类型，有一个条件可以选择不同的行为

- 将条件语句的每个分支移动到子类中的重写方法
- 使原来的方法变成抽象的
    - 否则，你将引入一个叫做**拒绝请求 Refused Request** 的不好的代码实例



懒惰类 Lazy Class

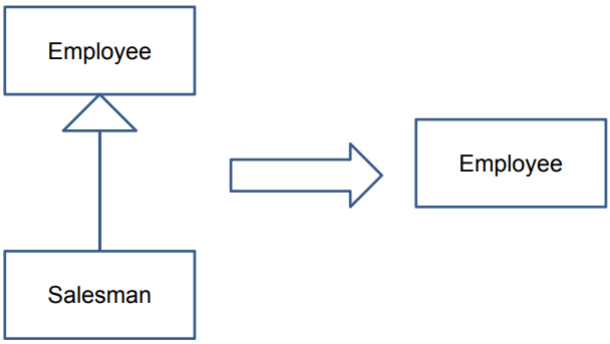
每一个类都需要花费一些时间来维护和理解

- 我们通常不会有意地创建惰性类，但它通常是由于缩减规模或投机地添加内容而导致的
- 有些类随着时间的推移，不怎么调用 / 使用某个类

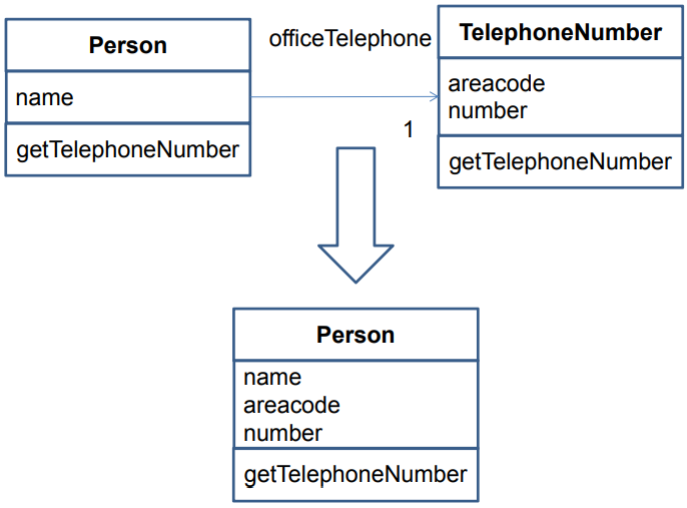
潜在的重构方法

- 折叠层次结构 Collapse hierarchy
- 内联类 Inline class

合并继承层次 Collapse hierarchy



内联类 Inline class



投机的概括性 Speculative Generality

有时候我们会去创建一些特殊样例，来处理一些可能甚至都不会发生的事情

- 我们可能以后需要一种方法来做 X
- 当您拥有实际上不需要（至少现在还不需要）的通用或抽象代码时，这一点就很明显了

潜在的重构方法

- 折叠层次结构 Collapse hierarchy
- 重命名方法 Rename method
- 移除参数 Remove parameter
- 内联类 inline class

临时字段 Temporary Field

- 只在某些实例中设置的实例变量
- 其余的时候，字段是空的，或者（更糟）包含不相关的数据
    - 这阻碍了可理解性，并可能导致基于上下文的意外错误

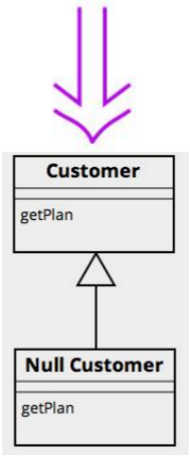
潜在的重构方法

- 提取类 Extract class
- 引出空对象 Introduct null object

引出空对象 Introduct null object

所以将空值替换为空对象

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



信息链 Message Chains

当您看到一长串的方法调用或临时变量来获取一些数据时发生

一个长串的 string，比如 `xxx.getThis().getThat().getSomething()`

- 使代码依赖于导航组件之间关系的算法
- 未能保护外部对象不受内部实现的影响

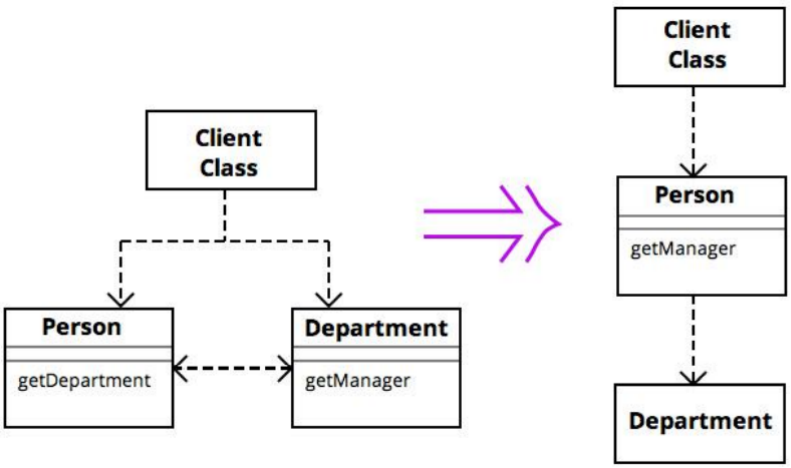
潜在的重构方法

- 隐藏委托 Hide delegate
- 提取方法 Extract method
- 移动方法 Move method

隐藏委托 Hide delegate

客户端正在调用一个委托类

- 在服务类中创建方法来隐藏委托




```
class Person{
    Department depart;
    public Department getDepartment(){
```

```
        return depart;
    }

    public void setDepartment(Department arg){
        depart = arg;
    }
}

class Department{
    private String _chargeCode;
    private Person _manager;

    public Department(Person manager){
        _manager = manager;
    }

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

现在，如果你想要获取一个 `manager`

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

修改

```
class Person{
    Department depart;
    public Department getDepartment(){
        return depart;
    }

    public Person getManager(){
        return depart.getManager();
    }

    public void setDepartment(Department arg){
        depart = arg;
    }
}

class Department{
    private String _chargeCode;
    private Person _manager;

    public Department(Person manager){
        _manager = manager;
```

```
22      }
23
24      public Person getManager(){
25          return _manager;
26      }
27  }
```

现在再调用

```
1   manager = john().getManager();
```

中间人 Middle Man

委托是一个好的方法，也是我们为什么创造对象

但是有些时候，我们最终得到一个设计，一个对象所做的一切就是将调用传递给另一个对象

- 没有明显的原因（例如，适配器）

在信息隐藏和委托开销之间有一条细微的界线

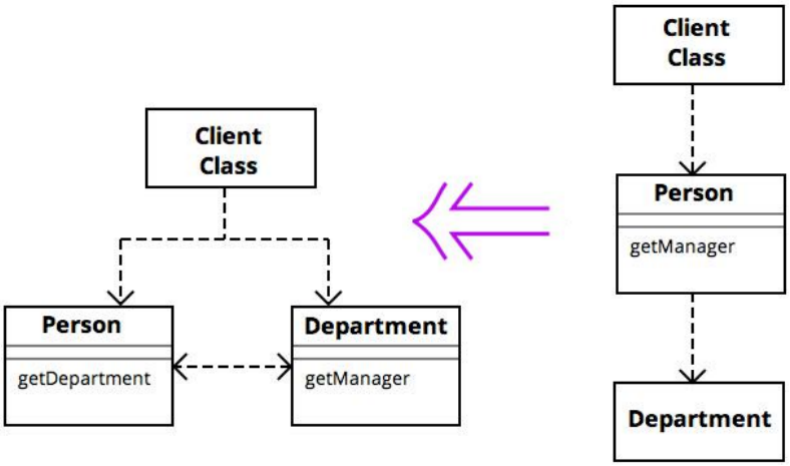
潜在的重构方法

- 移除中间人 Remove middle man
- 内联方法 Inline method
- 将委托替换成继承 Replace delegation with inheritanc

移除中间人 Remove middle man

一个类做了太多的简单委托

- 让客户端调用委托



内联方法 Inline method

方法的主体就像它的名字一样清楚

- 因此，将方法的主体放到它的调用者的主体中，并删除该方法

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
  return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

不恰当的亲密 Inappropriate intimacy

类有时会过度钻研彼此的私有方法和字段

相关联：数据类 —— 只有字段，getter 和 setter

- 几乎可以肯定的是，被其他类在太多细节上操纵了

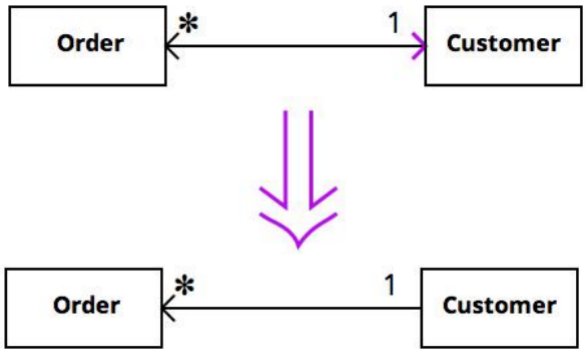
潜在的重构方法

- 移动方法 Move method
- 移动字段 Move field
- 将双向关联改为单向关联 Change bidirectional association to unidirectional association
- 提取类 Extract class
- 隐藏委托 Hide delegate
- 封装集合 Encapsulate collection

将双向关联改为单向关联 Change bidirectional association to unidirectional association

有一个双向关联，但一个类不再需要访问另一个类

- 所以删除不必要的连接

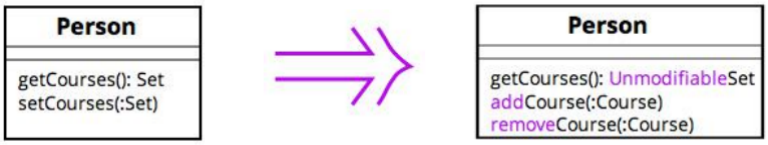


封装集合 Encapsulate collection

返回一个集合的方法

- 这可能会令人困惑，因为对调用者来说，他似乎可以对收集者进行更改

让它返回一个只读视图，并提供添加 / 删除方法



具有不同接口的互替代类 Alternative Classes with Different Interfaces

类可以在外部完全不同，但最终在内部是相同的

基本上，你应该找到这两个类的相似之处，重构它们以共享其中的共同点

潜在的重构方法

- 提取父类 Extract superclass
- 使用适配器统一接口 Unify interfaces with adapter

拒绝继承 Refused Bequet

当你继承了你不喜欢的代码

- 子类只使用了父类的非常少的功能

最糟糕的情况是，子类想要重新实现父类所有的行为

潜在的重构方法

- 字段下移 Push down field
- 方法下移 Push down method
- 用委托代替继承 Replace inheritance with delegation

用委托代替继承 Replace inheritance with delegation

子类只使用超类接口的一部分，或者不希望继承数据

- 父类创建一个字段，调整方法以委托给父类，并删除子类