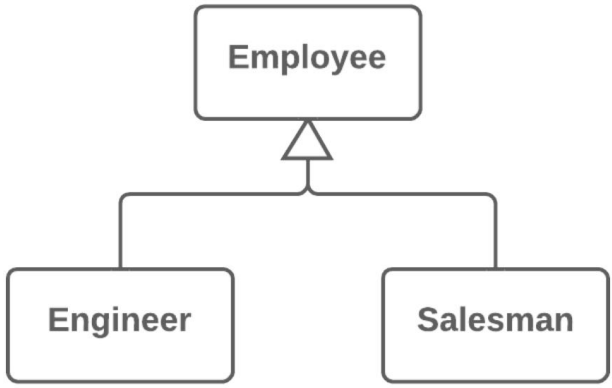
解决方案

为编码类型的每个值创建子类。然后
从原始类中提取相关行为到这些
子类。用多态替换控制流代码。



为什么要重构

这种重构技术是一个更复杂的转折
用类替换类型代码。

和第一种重构方法一样,你有一组简单的值,它们构成了一个字段的
所有允许值。虽然
这些值通常被指定为常量并具有可命名的名称,它们的使用使您的代
码非常容易出错
因为它们仍然是有效的原语。例如,您有
在参数中接受这些值之一的方法。

在某个时刻,而不是常量USER_TYPE_ADMIN
值为“ADMIN”，该方法接收相同的字符串

小写（“admin”），这将导致执行作者（您）不打算执行的其他操作。

在这里，我们正在处理控制流代码，例如 condi开关和?: 。全国如果，换句话说，带有编码的字段值（如$user->type === self::USER_TYPE_ADMIN ）是在这些运算符的条件内使用。如果我们要在这里用类替换类型代码，所有这些控制流最好将结构转移到负责的班级数据类型。最终，这当然会创建一个类型类与原来的非常相似，有同样的问题也是。

好处

- 删除控制流代码。而不是一个笨重的开关原始类，将代码移动到适当的子类。这提高对单一职责原则的遵守程度总体上使程序更具可读性。

- 如果您需要为编码类型添加新值，您只需要要做的是在不触及现有代码的情况下添加一个新的子类（参见开放/封闭原则）。

- 通过用类替换类型代码，我们为类型铺平了道路在编程级别提示方法和字段语言。使用简单的数字或编码类型中包含的字符串值。

什么时候不使用

- 如果您已经拥有类层次结构,则此技术不适用。您不能通过面向对象编程中的继承来创建双重层次结构。不过,您可以通过组合而不是继承来替换类型代码。为此,请使用将类型代码替换为状态/策略。

- 如果创建对象后类型代码的值会发生变化,请避免使用此技术。我们将不得不以某种方式即时替换对象本身的类,这是不可能的。不过,在这种情况下,另一种选择是用状态/策略替换类型代码。

如何重构

1. 使用 Self Encapsulate Field 为该字段创建一个 getter
   包含类型代码。

2. 将超类构造函数设为私有。创建一个与超类构造函数具有相同参数的静态工厂方法。它必须包含将采用编码类型的起始值的参数。根据这个参数,工厂方法将创建各种子类的对象。为此,您必须在其代码中创建一个大型条件,但至少,它是唯一真正需要的条件;否则,子类和多态就可以了。

3. 为编码类型的每个值创建一个唯一的子类。其中，重新定义编码类型的getter，使其返回编码类型的对应值。

4. 从超类中删除带有类型代码的字段。使其吸气剂抽象。

5. 现在你有了子类，你可以开始将字段和方法从超类移动到相应的子类（借助下推字段和下推方法）。

6. 当所有可能的东西都被移动时，使用用多态替换条件以一劳永逸地摆脱使用类型代码的条件。

反重构

§ 用字段替换子类

类似的重构

§ 用类替换类型代码

§ 用状态/策略替换类型代码

消除异味

§ 原始痴迷

B用状态/策略替换类型代码

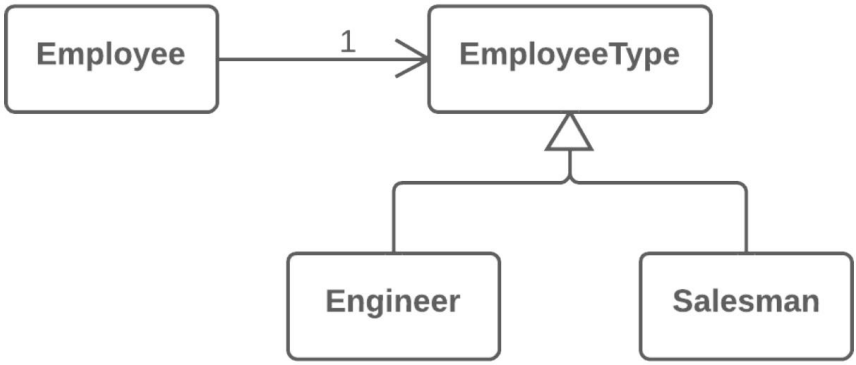
什么是类型代码?当您拥有一组数字或字符串,而不是单独的数据类型时,就会出现类型代码,这些数字或字符串构成了某个实体的允许值列表。
通常这些特定的数字和字符串通过常量被赋予易于理解的名称,这就是原因
为什么会遇到如此多的类型代码。

问题

您有一个影响行为的编码类型,但您不能使用子类来摆脱它。

解决方案

用状态对象替换类型代码。如果需要用类型代码替换字段值，则“插入”另一个状态对象。



为什么要重构

您有类型代码，它会影响类的行为，因此我们不能使用用类替换类型代码。

类型代码会影响类的行为，但由于现有的类层次结构或其他原因，我们无法为编码类型创建子类。因此意味着我们不能用子类替换类型代码。

好处

- 这种重构技术是一种解决当一个具有编码类型的字段在对象的运行过程中改变它的值的情况的方法。

寿命。在这种情况下,通过替换原始类的状态对象来替换值

- 如果您需要添加编码类型的新值,您只需添加新的状态子类而不更改现有代码（参见打开/关闭原则）。

缺点

如果您有一个简单的类型代码案例,但无论如何都使用了这种重构技术,那么您将拥有许多额外的（和不需要的)类。

很高兴知道

这种重构技术的实现可以利用两种设计模式之一:状态或策略。无论您选择哪种模式,实现都是相同的。那么在特定情况下您应该选择哪种模式呢?

如果您尝试拆分控制算法选择的条件,请使用策略。

但是,如果编码类型的每个值不仅负责选择算法,而且负责类的整体条件、类状态、字段值和许多其他操作,那么 State 更适合这项工作。

如何重构

1. 使用 Self Encapsulate Field为该字段创建一个 getter
   包含类型代码。

2. 创建一个新类,并给它一个符合类型代码目的的易于理解的名称。这个类将扮演状态（或
   策略)的角色。在其中,创建一个抽象编码字段 getter。

3.为编码的每个值创建状态类的子类
   类型。在每个子类中,重新定义编码字段的getter,使其返回编码类型的对应值。

4. 在抽象状态类中,创建一个静态工厂方法,接受编码类型的值作为参数。根据这个参数,工
   厂方法将创建各种状态的对象。为此,在其代码中创建一个大条件 al;重构完成后,它
   将是唯一的一个。

5.在原类中,将编码字段的类型改为状态类。在字段的 setter 中,调用工厂状态方法来获取
   新的状态对象。

6. 现在您可以开始将字段和方法从超类移动到相应的状态子类（使用Push Down Field
   和Push Down Method）。

7. 当所有可移动的东西都被移动时,使用用多态替换条件来彻底摆脱使用类型代码的条件。

类似的重构

§ 用类替换类型代码

§ 用子类替换类型代码

实现设计模式

§ 状态

§ 策略

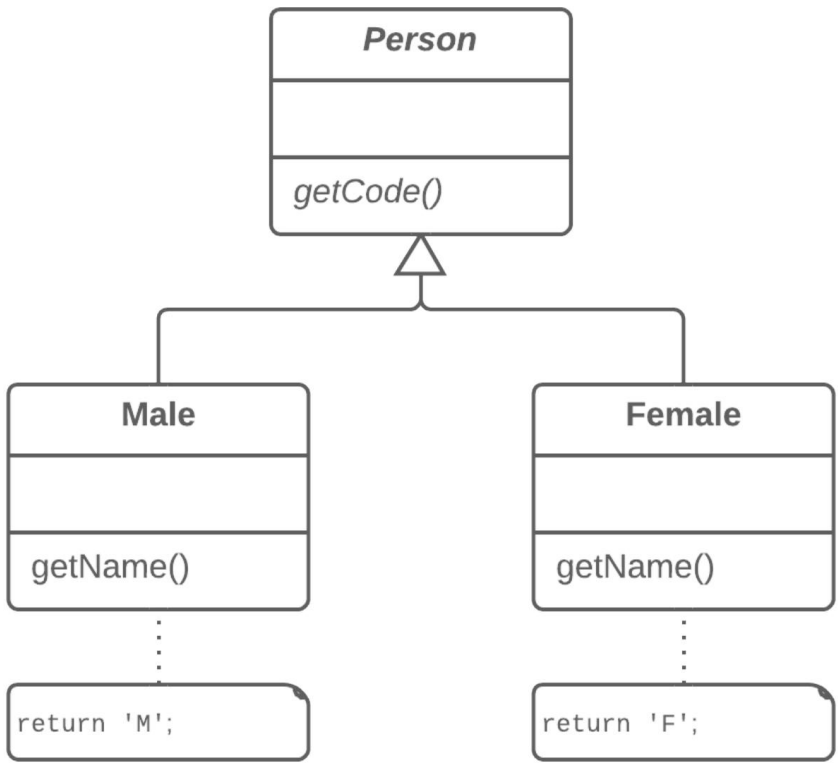
消除异味

§ 原始痴迷

B用字段替换子类

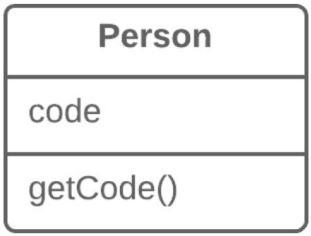
问题

您的子类仅在它们的（常量返回)方法上有所不同。



解决方案

用父类中的字段替换方法并删除
子类。



为什么要重构

有时重构只是避免类型代码的门票。

在一种这样的情况下,子类的层次结构可能仅在特定方法返回的值上有所不同。这些方法甚至不是计算的结果,而是在方法本身或方法返回的字段中严格规定。为了简化类体系结构,可以根据情况将该层次结构压缩为包含一个或多个具有必要值的字段的单个类。

在将大量功能从类层次结构移动到另一个位置后,这些更改可能变得必要。当前的层次结构不再那么有价值,它的子类现在只是沉重的负担。

好处

简化系统架构。如果你想做的只是在不同的地方返回不同的值,那么创建子类就大材小用了
方法。

如何重构

1. 将 Replace Constructor with Factory Method应用于
子类。

2.用超类工厂替换子类构造函数调用
方法调用。

3.在超类中,声明存放各个值的字段
返回常量值的子类方法。

4. 创建一个受保护的超类构造函数来初始化
新领域。

5.创建或修改已有的子类构造函数,使其调用父类的新构造函数,并传递

与之相关的价值观。

6、在父类中实现每个常量方法,使其返回对应字段的值。然后删除

子类的方法。

7. 如果子类构造函数具有附加功能,请使用
内联方法将构造函数合并到超类工厂方法中。

8. 删除子类。

反重构

§ 用子类替换类型代码

简化条件表达式

随着时间的推移,条件的逻辑往往会变得越来越复杂,还有更多的技术可以对抗

这也是。

§ 分解条件

问题:您有一个复杂的条件（ if-then / else或switch ）。

解决方案:将条件的复杂部分分解为单独的方法:条件,然后和其他。

§ 合并条件表达式

问题:您有多个导致相同结果或操作的条件。

解决方案:将所有这些条件合并到一个表达式中。

§ 合并重复的条件片段

问题:在所有分支中都可以找到相同的代码
有条件的。

解决方案:将代码移到条件之外。

§ 删除控制标志

问题:您有一个用作控件的布尔变量
多个布尔表达式的标志。

解决方案:代替变量,使用break , 继续和
返回。

§ 用保护子句替换嵌套条件

问题:你有一组嵌套条件,这很难
来确定代码执行的正常流程。

解决方案:将所有特殊检查和边缘情况隔离到单独的子句中,并将它们放在主要检查之前。理想的盟友,你应该有一个“平坦”的条件列表,一个接一个
另一个。

§ 用多态替换条件

问题:您有一个执行各种操作的条件
取决于对象类型或属性。

解决方案:创建与条件分支匹配的子类。在其中,创建一个共享方法并从

条件的相应分支。然后更换
带有相关方法调用的条件。结果是
正确的实现将通过多态性获得,具体取决于对象类。

§ 引入空对象

问题:由于某些方法返回null而不是 real
对象,您在代码中对null进行了许多检查。

解决方案:而不是null ，返回一个展示的空对象
默认行为。

§ 引入断言

问题:要让一部分代码正常工作,某些条件或值必须为真。

解决方案:用特定的断言替换这些假设
检查。

B分解有条件的

问题

您有一个复杂的条件（ if-then / else或switch ）。

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
        费用 = 数量 * 冬季价格 + 冬季服务费用；
}
其他{
        批次 = 数量 * SummerRate；
}
```

解决方案

将条件的复杂部分分解为
单独的方法:条件,然后和其他。

```
if (isSummer(日期)) {
        电荷 = 夏季电荷（数量）；
    }
其他{
        收费 = 冬季收费（数量）；
```

```
6 }
```

为什么要重构

一段代码越长,就越难理解。

当代码充满条件时,事情变得更加难以理解:

- 当您忙于弄清楚then块中的代码做了什么时,您忘记了相关的条件是什么。
- 当您忙于解析else时,您忘记了其中的代码
  然后做。

好处

- 通过将条件代码提取到明确命名的方法中,您可以让以后维护代码的人(比如您,两个月后!)的工作更轻松。
- 这种重构技术也适用于条件中的短表达式。字符串isSalaryDay()比用于比较日期的代码更漂亮且更具描述性。

如何重构

1.通过Extract将条件提取到单独的方法
方法。

2. 对then和else块重复该过程。

消除异味

§ 长法

B巩固有条件的表达

问题

您有多个导致相同结果的条件或行动。

```
1双disabilityAmount（） {
2        如果（资历< 2){
3            返回0；
4        }
5        if (monthsDisabled > 12) {
6            返回0；
7        }
8日      if (isPartTime) {
9            返回0；
10       }
11       // 计算残疾金额。
12       // ...
13 }
```

解决方案

将所有这些条件合并到一个表达式中。

```
1双disabilityAmount（） {
2       if (isNotEligableForDisability()) {
3           返回0；
4       }
5       // 计算残疾金额。
6       // ...
7 }
```

为什么要重构

您的代码包含许多交替执行的运算符
相同的动作。目前尚不清楚为什么运营商会被拆分。

合并的主要目的是提取条件
为了更清楚，另一种方法。

好处

- 消除重复的控制流代码。结合多个
  具有相同“目的地”的条件有助于显示
  你只做一个复杂的检查导致一个
  行动。

· 通过合并所有运算符,您现在可以将这个复杂的表达式隔离在一个新方法中,该方法的名称可以解释条件的用途。

如何重构

在重构之前,请确保条件没有任何“副作用”或以其他方式修改某些内容,而不是简单地返回值。副作用可能隐藏在运算符本身内部执行的代码中,例如当某些东西

根据条件的结果添加到变量中。

1. 通过使用将条件合并到一个表达式中
   和和或。作为合并时的一般规则:
   - 嵌套条件句使用和连接。
   - 连续条件句与or连接。

2.对运算符条件执行提取方法,并为方法指定一个反映表达式用途的名称。

消除异味

§ 重复代码

B巩固复制有条件的碎片

问题

在条件的所有分支中都可以找到相同的代码。

```
if (isSpecialDeal()) {
        总计 = 价格 * 0.95;
        发送 () ;
}
其他{
        总计 = 价格 * 0.98;
        发送 () ;
}
```

解决方案

将代码移到条件之外。

```
if (isSpecialDeal()) {
        总计 = 价格 * 0.95;
}
其他{
        总计 = 价格 * 0.98;
}
发送（）；
```

为什么要重构

在条件的所有分支中发现重复的代码，
通常是条件内代码演变的结果

国家分支机构。团队发展可以是一个贡献因素
通往此的大门。

好处

代码重复数据删除。

如何重构

1.如果重复的代码在条件的开头
分支，将代码移动到条件之前的位置。

2.如果代码在分支的末尾执行，放在后面
有条件的。

3.如果重复代码随机位于分支es内部，首先尝试将代码移动到分支es的开头或结尾

分支,取决于它是否改变了子顺序代码的结果。

4. 如果合适且重复代码超过一行,
尝试使用 ExtractMethod。

消除异味

§ 重复代码

B删除控制标志

问题

你有一个布尔变量作为控制标志
多个布尔表达式。

解决方案

代替变量,使用break , 继续并返回。

为什么要重构

控制标志可以追溯到过去,那时“合适的”程序员总是有一个函数
入口点（
函数声明行)和一个出口点（在最后
函数）。

在现代编程语言中,这种风格的 tic 已经过时了,
因为我们有特殊的操作符来修改控制流
在循环和其他复杂结构中：

- break : 停止循环
- continue : 停止当前循环分支的执行和
去检查下一次迭代中的循环条件

- return : 停止整个函数的执行并返回它的

如果在运算符中给出结果

好处

控制标志代码通常比代码笨重得多
用控制流运算符编写。

如何重构

1. 找到导致控制标志的值分配
   退出循环或当前迭代。

2、换成break , 如果这是一个循环的退出;如果需要,请继续 ,
   如果这是迭代的退出,或者返回 ,
   从函数返回这个值。

3. 删除与
   控制标志。

B替换嵌套有条件的保护条款

问题

你有一组嵌套条件,很难
确定代码执行的正常流程。

```
1公共双getPayAmount() {
2       双重结果;
3       如果（死亡){
4           结果 = deadAmount();
5       }
6       否则{
7           if (isSeparated){
8               结果 = 分离金额（）；
9           }
10          否则{
11              如果（已退休){
12                  结果 = 退休金额（）；
13              }
14              别的{
15                  结果 = normalPayAmount();
16              }
```

```
        }
    }
    返回结果;
}
```

解决方案

将所有特殊检查和边缘情况隔离到单独的子句中
并将它们放在主要检查之前。理想情况下，您应该
有一个“平坦”的条件列表，一个接一个。

```
公共双getPayAmount() {
    如果（死亡）{
        返回死量（）；
    }
    if (isSeparated){
        返回分离金额（）；
    }
    如果（已退休）{
        返回退休金额（）；
    }
    返回正常支付金额（）；
}
```

为什么要重构

发现“来自地狱的条件”相当容易。每一层嵌套的缩进形成一个箭头，
指向
在痛苦和不幸的方向上的权利：

```
1
2如果 (){
3        如果 (){
4            做{
5                如果 (){
6                    如果 (){
7                        如果 (){
8                            ...
9                        }
10                   }
11                   ...
12               }
13               ...
14           }
15           而 () ;
16           ...
17       }
18       否则{
19           ...
20       }
21 }
```

很难弄清楚每个条件的作用和方式，
因为“正常”的代码执行流程不是立即执行的
明显的。这些条件表明仓促的进化，
每个条件都添加为权宜之计，没有任何
着力优化整体结构。

为了简化情况，将特殊情况隔离为单独的速率条件，这些条件立即结束执行并返回一个

如果保护子句为真,则为空值。实际上,你的使命
这是使结构平坦。

如何重构

尝试消除代码的副作用 从修饰符中分离查询可能有助于达到此目的。该解决方案
对于下面描述的改组是必要的。

1. 隔离所有导致调用异常的保护子句或
从方法中立即返回一个值。放置这些骗局
方法开始时的版本。

2.重排完成并且所有测试都成功完成后,看看是否可以对导致相同异常的保护子句使用
Consolidate Conditional Expression
或返回值。

B替换条件具有多态性

问题

您有一个执行各种操作的条件
取决于对象类型或属性。

```
1类鸟{
2       // ...
3       双getSpeed（）{
4           开关（类型）{
5               案例欧洲：
6                   返回getBaseSpeed();
7               非洲案例：
8                   return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
9               案例NORWEGIAN_BLUE：
10                  返回（被钉）？ 0 :getBaseSpeed（电压）；
11          }
12          throw new RuntimeException(   应该无法访问   );
13      }
14 }
```

解决方案

创建与条件分支匹配的子类。
在其中,创建一个共享方法并从
条件的相应分支。然后更换
以相关方法调用为条件。结果是
正确的实现将通过多态性来实现
取决于对象类别。

```
1个抽象类鸟{
2        // ...
3        抽象双getSpeed();
4 }
5
6类欧式延伸鸟{
7        双getSpeed （） {
8日          返回getBaseSpeed();
9        }
10 }
11类非洲延伸鸟{
12       双getSpeed （） {
13           return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
14       }
15 }
16类挪威蓝扩展鸟{
17       双getSpeed （） {
18           返回（被钉） ？ 0 :getBaseSpeed（电压）；
19       }
20 }
21
22 // 客户端代码中的某处
```

```
23速度 = bird.getSpeed();
```

为什么要重构

如果您的代码包含执行各种任务的运算符,这种重构技术会有所帮助,这些任务基于:

- 它实现的对象或接口的类
- 对象字段的值
- 调用对象方法之一的结果

如果出现新的对象属性或类型,您将需要在所有类似条件下搜索并添加代码。就这样
如果有多个条件分散在对象的所有方法中,那么这种技术的好处就会成倍增加。

好处

- 这种技术遵循Tell-Don t-Ask原则:与其向对象询问其状态然后基于此执行操作,不如简单地告诉对象它需要做什么并让它自己决定要容易得多怎么做。

- 删除重复代码。你摆脱了许多几乎相同的条件。

- 如果您需要添加新的执行变体,您只需要
  做的是在不触及现有代码的情况下添加一个新的子类
  （开/关原则）。

如何重构

准备重构

对于这种重构技术,您应该有一个现成的包含替代行为的类层次结构。如果你
没有这样的层次结构,创建一个。其他技术
将有助于实现这一目标：

- 用子类替换类型代码。将为特定对象属性的所有值创建子类。这种方法
  很简单但不太灵活,因为您不能为
  对象的其他属性。

- 将类型代码替换为状态/策略。一个类将专用于一个特定的对象属性,并且将从它为属性的每个值创建子类。现在的班级

  将包含对此类对象的引用和委托
  处决他们。

以下步骤假设您已经创建了
●

重构步骤

1. 如果条件在执行其他操作的方法中,执行提取方法。

2.对于每个层次子类,重新定义包含条件的方法,并复制对应条件的代码

国家分支机构到该位置。

3. 从条件中删除此分支。

4.重复替换,直到条件为空。然后删除
条件并声明方法摘要。

消除异味

§ 切换语句

B引入零对象

问题

由于某些方法返回null而不是真实对象,因此您在您的代码中对null进行多次检查。

```
1如果（客户 ==空） {
2        计划 = BillingPlan.basic();
3 }
4其他{
5        计划 = customer.getPlan();
6 }
```

解决方案

而不是空，返回一个显示默认值的空对象行为。

```
1类NullCustomer扩展客户{
2        布尔isNull() {
3            返回真；
4        }
```

```
5         计划 getPlan() {
6             返回新的NullPlan();
7         }
8         // 其他一些 NULL 功能。
9 }
10
11 //用Null对象替换空值。
12客户 = (order.customer != null) ?
13         order.customer :新的NullCustomer();
14
15 // 像普通子类一样使用 Null 对象。
16计划 = customer.getPlan();
```

为什么要重构

对null的数十次检查使您的代码更长更丑陋。

缺点

摆脱条件的代价是创造另一个
新班。

如何重构

1. 从有问题的类中,创建一个将执行的子类
   空对象的作用。

2.在两个类中,创建方法isNull() ，哪个想要
   对空对象返回true ,对真实类返回false 。

3. 查找代码可能返回null而不是 a
   真实的物体。更改代码，使其返回一个空对象。

4. 找到所有真实类的变量与null进行比较的地方。用呼吁替换这些检查
   isNull() 。

5. ◦ 如果原始类的方法在此条件下运行
   当一个值不等于null 类中的 null ods 并插入else中的代，重新定义这些方法
   码时
   那里的条件的一部分。然后你可以删除整个
   有条件的和不同的行为将通过
   多态性。
   ◦ 如果事情不是那么简单并且无法重新定义方法，请查看是否可以
   简单地提取之前的运算符
   应该在空值的情况下执行
   空对象的新方法。改为调用这些方法
   else中的旧代码默认作为操作。

类似的重构

§ 用多态替换条件

实现设计模式

§ 空对象

消除异味

§ 切换语句

§ 临时场地

B引入断言

问题

为了使部分代码正常工作,某些条件或
值必须为真。

```
1双getExpenseLimit() {
2        // 应该有费用限制或
3        // 一个主要项目。
4        返回（费用限制!= NULL_EXPENSE） ?
5            费用限制：
6            primaryProject.getMemberExpenseLimit();
7 }
```

解决方案

用特定的断言检查替换这些假设。

```
1双getExpenseLimit() {
2        Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject != nul
3
4        返回（费用限制!= NULL_EXPENSE） ?
5            费用限额：
6            primaryProject.getMemberExpenseLimit();
7 }
```

为什么要重构

假设代码的一部分假设了一些事情,例如,对象的当前条件或参数或局部变量的值。通常,除非发生错误,否则此假设将始终成立。

通过添加相应的断言使这些假设变得明显。与方法参数中的类型提示一样,这些断言可以充当代码的实时文档。

作为查看代码在哪里需要断言的指南,请检查描述 par 的条件的注释

特定的方法将起作用。

好处

如果假设不正确,因此代码给出了错误的结果,最好在这导致致命后果和数据损坏之前停止执行。这也意味着您在设计执行程序测试的方法时忽略了编写必要的测试。

缺点

- 有时异常比简单的断言更合适。您可以选择异常的必要类,并让剩余的代码正确处理它。

· 什么时候异常比简单断言更好?如果
异常可能由用户或系统的操作引起
你可以处理异常。另一方面,通常未命名和未处理的异常基本上等同于简单的断言
你不处理它们,它们是

完全由于程序错误导致
应该发生的。

如何重构

当您看到假设条件时,添加一个断言
这个条件才能确定。

添加断言不应改变程序的行为。

不要过度使用断言来处理你的所有内容
代码。仅检查代码正确运行所必需的条件。如果您的代码即使在特定断言为假的情况
下也能正常工作,您可以安全地

删除断言。

消除异味

§ 评论

简化方法调用

这些技术使方法调用更简单、更容易理解。这反过来又简化了 interac 的接口

类之间的化。

§ 重命名方法

问题:方法的名称不能解释方法的作用。

解决方法:重命名方法。

§ 添加参数

问题:方法没有足够的数据来执行某些操作。

解决方案:创建一个新参数以传递必要的数据。

§ 移除参数

问题:方法体中没有使用参数。

解决方案:删除未使用的参数。

§ 将查询与修饰符分开

问题:你有没有一个方法可以返回一个值,但也改变了对象内部的某些东西?

解决方案:将该方法拆分为两个单独的方法。如您所料,其中一个应该返回值,另一个修改对象。

§ 参数化方法

问题:多个方法执行相似的操作,只是它们的内部值、数字或操作不同。

解决方案:通过使用将传递必要的特殊值的参数组合这些方法。

§ 用显式方法替换参数

问题:一个方法被分成几个部分,每个部分的运行取决于一个参数的值。

解决方案:将方法的各个部分提取到自己的方法中,并调用它们而不是原来的方法。

§ 保留整个对象

问题:你从一个对象中获取几个值,然后将它们作为参数传递给一个方法。

解决方案:相反,尝试传递整个对象。

§ 用方法调用替换参数

问题:调用查询方法并将其结果作为另一个方法的参数传递,而该方法可以直接调用查询。

解决方案:不要通过参数传递值,而是尝试在方法主体内放置查询调用。

§ 引入参数对象

问题:您的方法包含一组重复的
参数。

解决方案:将这些参数替换为一个对象。

§ 删除设置方法

问题:字段的值只能在创建时设置,之后任何时候都不能更改。

解决方案:因此删除设置字段值的方法。

§ 隐藏方法

问题:方法不被其他类使用或仅在其自己的类层次结构中使用。

解决方案:将方法设为私有或受保护。

§ 用工厂方法替换构造函数

问题:你有一个复杂的构造函数,它做的不仅仅是在对象字段中设置参数值。

解决方法:创建一个工厂方法并用它来替换con
构造函数调用。

§ 用异常替换错误代码

问题:方法返回一个表示错误的特殊值?

解决方案:改为抛出异常。

§ 用测试替换异常

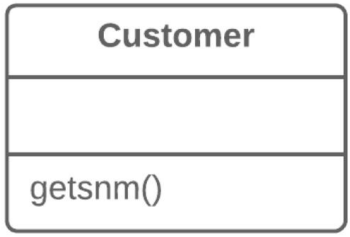
问题:你在一个简单的测试就可以完成工作的地方抛出一个异常?

解决方案:将异常替换为条件测试。

B重命名方法

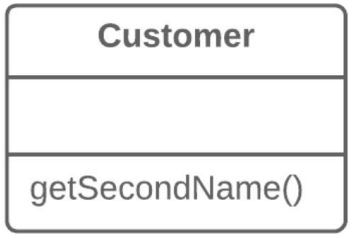
问题

方法的名称并不能解释该方法的作用。



解决方案

重命名方法。



为什么要重构

可能一个方法从一开始就没有命名 - 例如,有人匆忙创建了该方法,然后

没有适当注意命名它。

或者,也许该方法一开始命名得很好,但由于它的
功能增长,方法名称不再是一个好的
描述符。

好处

代码可读性。尝试为新方法命名
反映了它的作用。像createOrder()这样的东西 ,
渲染客户信息（） , 等等。

如何重构

1.查看方法是定义在超类还是子类中。
如果是这样,您也必须重复这些课程中的所有步骤。

2. 下一个方法对于维护功能很重要
在重构过程中程序的 ty。创建一个新的
具有新名称的方法。复制旧方法的代码
给它。删除旧方法中的所有代码,而不是它,
插入对新方法的调用。

3.找到对旧方法的所有引用并将它们替换为
引用新的。

4.删除旧方法。如果旧方法是公共接口的一部分,请不要执行此步骤。相反,标记旧的

不推荐使用的方法。

类似的重构

§ 添加参数

§ 移除参数

消除异味

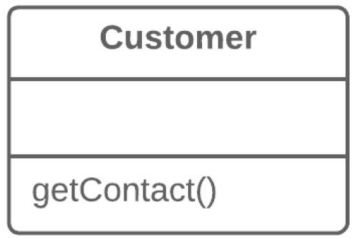
§ 具有不同接口的替代类

§ 评论

B添加参数

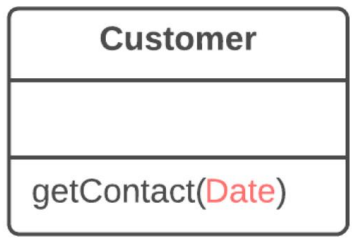
问题

方法没有足够的数据来执行某些操作。



解决方案

创建一个新参数以传递必要的数据。



为什么要重构

您需要对方法进行更改,而这些更改需要添加以前没有的信息或数据

可用于该方法。