好处

这里的选择是在添加新参数和
添加一个包含所需数据的新私有字段
方法。当您需要一些没有意义的偶尔或频繁更改的数据时，最好使用
字段
一直把它放在一个物体里。在这种情况下，一个新的
参数将比私有字段更合适，并且重构将得到回报。否则，添加一个私有
字段并填充它
调用方法之前的必要数据。

缺点

- 添加一个新参数总是比删除它更容易，
  这就是为什么参数列表经常膨胀到怪诞的原因
  尺寸。这种气味被称为长参数列表。

- 如果您需要添加新参数，有时这意味着
  您的类不包含必要的数据或现有的参数不包含必要的相关数据。在

  在这两种情况下，最好的解决方案是考虑将数据移动到
  主类或对象已经是访问权限的其他类
  sible 从方法内部。

如何重构

1.查看方法是定义在超类还是子类中。
  如果该方法存在于其中，您将需要重复所有
  这些类中的步骤也是如此。

2. 以下步骤对于在重构过程中保持程序正常运行至关重要。通过复制旧方法创建一个新方法并向其添加必要的参数。

   将旧方法的代码替换为对新方法的调用。您可以将任何值插入新参数（例如对象的null或数字的零）。

3.找到对旧方法的所有引用并将它们替换为对新方法的引用。

4.删除旧方法。如果旧方法是公共接口的一部分，则无法删除。如果是这种情况，请将旧方法标记为已弃用。

反重构

§ 移除参数

类似的重构

§ 重命名方法

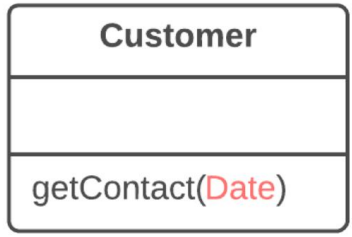
帮助其他重构

§ 引入参数对象

B移除参数

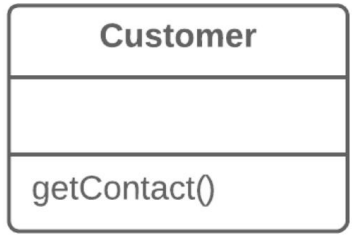
问题

方法体中不使用参数。



解决方案

删除未使用的参数。



为什么要重构

方法调用中的每个参数都迫使程序员阅读它以找出在该参数中找到的信息。

如果一个参数在方法体中完全没有使用,那么这种“头疼”就是徒劳的。

在任何情况下,额外的参数都是额外的代码
要运行。

有时我们会着眼于未来添加参数,对可能需要参数的方法进行预期的更改。尽管如此,经验表明最好只在真正需要时才添加参数。毕竟,预期的变化通常只是预期的。

好处

一个方法只包含它真正需要的参数。

什么时候不使用

如果该方法在子类或超类中以不同的方式实现,并且您的参数在这些实现中使用,请保持参数不变。

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,那里使用的参数是什么?如果在这些实现之一中使用了该参数,请推迟使用此重构技术。

2. 下一步对于在重构过程中保持程序功能很重要。通过复制旧方法创建新方法并从中删除
相关参数。

用对新方法的调用替换旧方法的代码。

3.找到对旧方法的所有引用并将它们替换为
对新方法的引用。

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。在这种情况下,将旧方法
标记为已弃用。

反重构

§ 添加参数

类似的重构

§ 重命名方法

帮助其他重构

§ 用方法调用替换参数

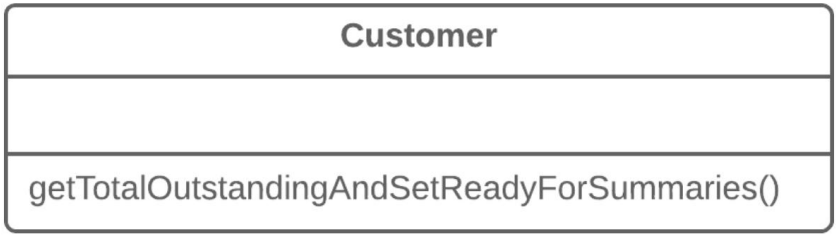
消除异味

§ 推测的普遍性

B将查询与修饰符

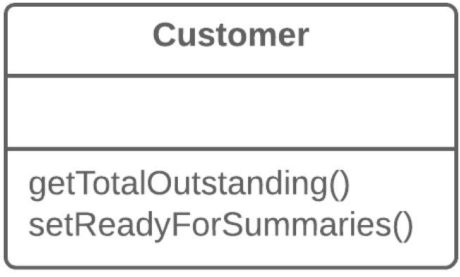
问题

您是否有一个方法可以返回一个值但也可以更改对象内部的某些内容?



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。



为什么要重构

这种分解技术实现了命令和查询职责分离。这个原则告诉我们将负责获取数据的代码与更改对象内部内容的代码分开。

获取数据的代码称为查询。更改对象可见状态的代码称为修饰符。

组合查询和修饰符时，您无法在不更改其条件的情况下获取数据。换句话说，您提出一个问题，并且即使在收到答案时也可以更改答案。当调用查询的人可能不知道该方法的“副作用”时，这个问题变得更加严重，这通常会导致运行时错误。

但请记住，只有在修改器会改变对象的可见状态时，副作用才是危险的。例如，这些可能是从对象的公共接口、数据库中的条目、文件中可访问的字段等。如果修饰符只缓存一个复杂的操作并将其保存在类的私有字段中，它几乎不会导致任何副作用。

好处

如果您有一个不改变程序状态的查询，您可以随意调用它多次，而不必担心仅仅因为您调用该方法而导致结果发生意外变化。

缺点

在某些情况下,执行命令后获取数据很方便。例如,从数据库中删除某些内容时,您想知道删除了多少行。

如何重构

1.新建一个查询方法,返回原来的内容
方法做到了。

2.改变原来的方法,让它只返回结果
调用新的查询方法。

3. 将所有对原始方法的引用替换为对查询方法的调用。在此行之前,调用修饰符方法。如果在条件运算符或循环的条件下使用原始方法,这将使您免受副作用的影响。

4.去掉原来方法中的返回值代码,现在变成了合适的修饰符方法。

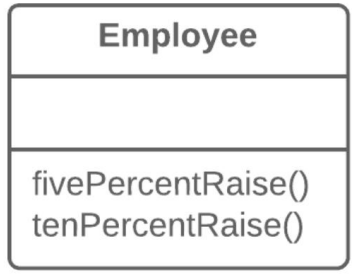
帮助其他重构

§ 用查询替换 Temp

B参数化方法

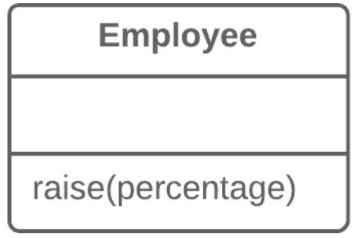
问题

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。



为什么要重构

如果你有类似的方法,你可能有重复的代码,这会带来所有的后果。

更重要的是,如果您需要添加此功能的另一个版本,您将不得不创建另一个方法。
相反,您可以简单地使用不同的参数运行现有方法。

缺点

- 有时这种重构技术可能太过分了,导致一个冗长而复杂的通用方法而不是多个更简单的方法。

- 将功能的激活/停用移动到参数时也要小心。这最终会导致创建一个大型条件运算符,需要通过使用显式方法替换参数来处理该运算符。

如何重构

1. 创建一个带有参数的新方法,并通过应用提取方法将其移动到对所有类都相同的代码中。
   请注意,有时只有某些方法实际上是相同的。在这种情况下,重构包括仅将相同部分提取到新方法中。

2. 在新方法的代码中,将特殊/差异值替换为参数。

3. 对于每个旧方法,找到调用它的位置,将这些调用替换为对包含参数的新方法的调用。然后删除旧方法。

反重构

§ 用显式方法替换参数

类似的重构

§ 提取方法

§ 表格模板法

消除异味

§ 重复代码

B替换参数
显式
方法

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。

```
1 void setValue（字符串名称,整数值） {
2         if (name.equals(    height    )) {
3             高度=价值;
4             返回;
5         }
6         if (name.equals(    width    )) {
7             宽度=值;
8             返回;
9         }
10        断言.shouldNeverReachHere();
11 }
```

解决方案

将方法的各个部分提取到自己的
方法并调用它们而不是原始方法。

```
1无效setHeight(int arg) {
2         高度 = arg;
3 }
4 void setWidth(int arg) {
5         宽度 = 参数;
6 }
```

为什么要重构

一种包含参数相关变体的方法已经发展
大量的。非平凡代码在每个分支和新变量中运行
蚂蚁很少添加。

好处

提高代码可读性。更容易理解
目的 的 `启动引擎（）` 比
`setValue( engineEnabled , true)` 。

什么时候不使用

如果方法是，不要用显式方法替换参数
很少更改，并且不会在其中添加新变体。

如何重构

1. 对于方法的每个变体,创建一个单独的方法。根据 main 中的参数值运行这些方法

方法。

2.找到所有调用原始方法的地方。在这些地方,调用新的参数依赖之一

变体。

3. 当没有保留对原始方法的调用时,将其删除。

反重构

§ 参数化方法

类似的重构

§ 用多态替换条件

消除异味

§ 切换语句

§ 长法

B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。

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

解决方案

相反,请尝试传递整个对象。

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

为什么要重构

问题是每次调用你的方法之前,都必须调用future参数对象的方法。
如果这些方法或为该方法获得的数据量

改变了,你需要在程序中仔细找到十几个这样的地方,并在每个地方实施这些改变
其中。

应用这种重构技术后,用于获取所有必要数据的代码将存储在一个地方 方法
本身。

好处

- 您看到的不是杂乱无章的参数,而是具有易于理解的名称的单个对象。

- 如果该方法需要来自对象的更多数据,则无需重写所有使用该方法的地方 只需
在方法本身内部。

缺点

有时这种转换会导致方法变成
不太灵活:以前该方法可以从许多不同的来源获取数据,但现在,由于重构,我们将其使用限制为仅具有特定接口的对象。

如何重构

1. 在方法中为对象创建一个参数
您可以获得必要的值。

2. 现在开始从方法中一一移除旧参数,代之以对参数对象相关方法的调用。每次替换参数后测试程序。

3. 删除参数对象的getter代码
放弃了方法调用。

类似的重构

§ 引入参数对象

§ 用方法调用替换参数

消除异味

§ 原始痴迷

§ 长参数列表

§ 长法

§ 数据块

B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。

```
1 int basePrice = 数量 * itemPrice; 2双季折扣 =
this.getSeasonalDiscount (); 3双倍费用 = this.getFees();

4 double finalPrice = discountedPrice(basePrice, seasonDiscount, fee);
```

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。

```
1 int basePrice = 数量 * itemPrice;
2双finalPrice = discountedPrice(basePrice);
```

为什么要重构

一长串参数很难理解。此外,对此类方法的调用通常类似于一系列级联,具有难以导航但必须传递给方法的曲折和令人振奋的值计算。因此,如果可以借助方法计算参数值,请在方法本身内部执行此操作并去掉参数。

好处

我们摆脱了不需要的参数并简化了方法调用。
这些参数通常不会像现在那样为项目创建,而是着眼于可能永远不会出现的未来需求。

缺点

您明天可能需要该参数来满足其他需求……让您重写该方法。

如何重构

1. 确保获取值的代码不使用当前方法中的参数,因为它们在另一个方法中不可用。如果是这样,移动代码是不可能的。

2. 如果相关代码比单个方法或函数调用更复杂,使用Extract Method将这段代码隔离在一个新的方法中,使调用简单。

3.在main方法的代码中,将所有对被替换参数的引用替换为对获取的方法的调用

价值。

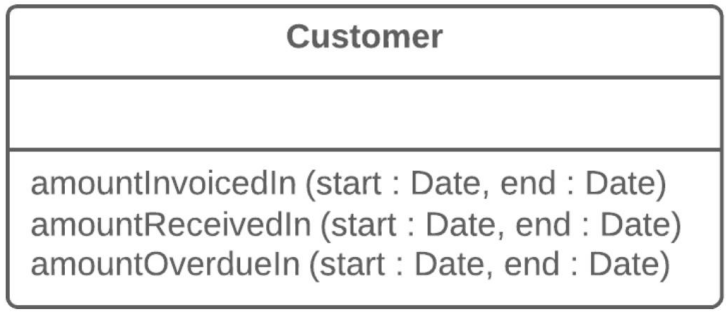
4. 使用 Remove Parameter消除现在未使用的

范围。

B引入参数对象

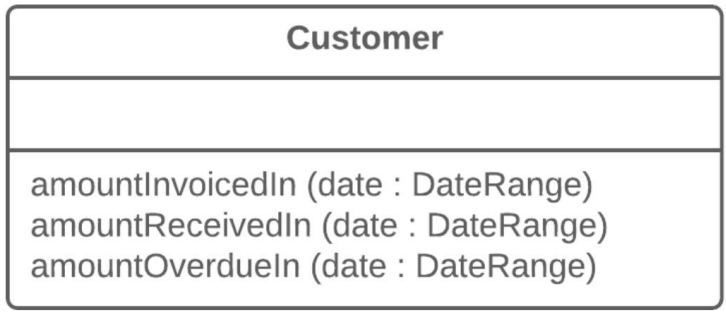
问题

您的方法包含一组重复的参数。



解决方案

将这些参数替换为对象。



为什么要重构

在多种方法中经常会遇到相同的参数组。这会导致参数本身和相关操作的代码重复。通过将参数合并到单个类中，您还可以将处理此数据的方法也移动到那里，从而释放其他方法

从这段代码。

好处

- 更易读的代码。您看到的不是大杂烩的参数，而是具有易于理解的名称的单个对象。

- 散布在各处的相同参数组会产生它们自己的代码重复：虽然没有调用相同的代码，但会经常遇到相同的参数组和参数。

缺点

如果您只将数据移动到一个新类并且不打算将任何行为或相关操作移到那里，这就会开始产生异味
的一个数据类。

如何重构

1. 创建一个代表你的参数组的新类
   三。使类不可变。

2. 在您要重构的方法中,使用 Add Parameter,这是您的参数对象将被传递的地方。在所有方法调用中,将从旧方法参数创建的对象传递给此参数。

3. 现在开始从方法中一一删除旧参数,将代码中的旧参数替换为参数对象的字段。每次更换参数后测试程序。

4. 完成后,看看将部分方法(有时甚至是整个方法)移动到参数对象类是否有任何意义。如果是这样,请使用移动方法或提取方法。

类似的重构

§ 保留整个对象

消除异味

§ 长参数列表

§ 数据块

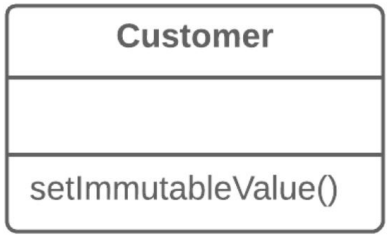
§ 原始痴迷

§ 长法

B删除设置方法

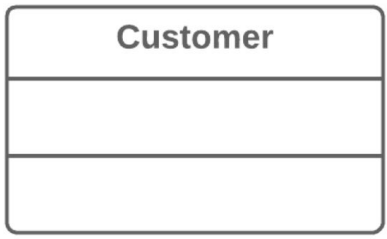
问题

字段的值应仅在创建时设置,之后不得更改。



解决方案

所以删除设置字段值的方法。



为什么要重构

您希望防止对字段值进行任何更改。

如何重构

1. 字段的值只能在构造函数中改变。如果构造函数不包含用于设置值的参数，则添加一个。

2. 查找所有 setter 调用。

   - 如果 setter 调用位于对当前类的构造函数的调用之后，则将其参数移动到构造函数调用并删除设置器。

   - 将构造函数中的 setter 调用替换为直接访问场。

3. 删除设置器。

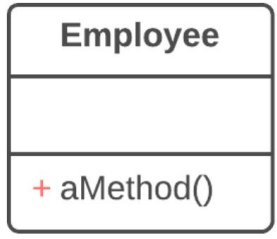
帮助其他重构

§ 更改对值的引用

B隐藏方法

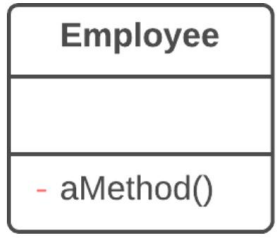
问题

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。



为什么要重构

很多时候,需要隐藏获取和设置值的方法是由于开发了更丰富的接口,该接口提供

额外的行为,特别是如果您从一个仅添加数据封装的类开始。

随着新行为内置到类中,您可能会发现不再需要公共 getter 和 setter 方法,并且可以将其隐藏。如果您将 getter 或 setter 方法设为私有并应用对变量的直接访问,则可以删除该方法。

好处

- 隐藏方法使您的代码更容易演变。当您更改私有方法时,您只需要担心如何不破坏当前类,因为您知道该方法不能在其他任何地方使用。

- 通过将方法设为私有,您可以强调类的公共接口和保持公共的方法的重要性。

如何重构

1.定期尝试寻找可以私有化的方法。静态代码分析和良好的单元测试覆盖率可以提供很大的帮助。

2. 使每个方法尽可能私有。

消除异味

§ 数据类

B替换构造函数使用工厂方法

问题

你有一个复杂的构造函数可以做更多的事情
不仅仅是在对象字段中设置参数值。

```
1类员工{
2        员工（int类型）{
3            this.type = 类型;
4        }
5        // ...
6 }
```

解决方案

创建一个工厂方法并使用它来替换构造函数调用。

```
1类员工{
2        静态员工创建（int类型）{
3            员工=新员工（类型）；
4            // 做一些繁重的工作。
5            返岗员工;
```

```
6       }
7       // ...
8 }
```

为什么要重构

使用这种重构技术的最明显原因与用子类替换类型代码有关。

您的代码中先前创建了一个对象,并将编码类型的值传递给它。使用重构方法后,出现了几个子类,您需要根据编码类型的值从它们中创建对象。更改原始构造函数以使其返回子类对象是不可能的,因此我们创建了一个静态工厂方法,该方法将返回必要类的对象,之后它会替换对原始构造函数的所有调用

构造函数。

当构造函数不能胜任任务时,工厂方法也可以在其他情况下使用。在尝试将值更改为参考时,它们可能很重要。它们还可用于设置超出参数数量和类型的各种创建模式。

好处

- 工厂方法不一定返回
  调用它的类。通常这些可能在其子类中
  es,根据提供给方法的参数进行选择。

- 工厂方法可以有一个更好的名称来描述它返回的内容和方式,例如

  部队::GetCrew(myTank) 。

- 工厂方法可以返回一个已经创建的对象,一个构造函数,它总是不同于创建一个新实例。

如何重构

1.创建工厂方法。调用当前构造函数
  在里面。

2. 将所有构造函数调用替换为对工厂方法的调用。

3. 将构造函数声明为私有。

4. 调查构造函数代码,尽量隔离与构造当前类的对象没有直接关系的代码,将这些代码移到
  工厂方法中。

帮助其他重构

§ 将值更改为参考

§ 用子类替换类型代码

实现设计模式

§ 工厂方法

B替换错误代码除了例外

问题

方法返回一个表示错误的特殊值？

```
int取款（int金额）{
    如果（金额> _balance）{
        返回-1;
    }
    否则{
        余额-=金额;
        返回0;
    }
}
```

解决方案

改为抛出异常。

```
无效取款（整数）抛出BalanceException {
    如果（金额> _balance）{
        抛出新的BalanceException();
    }
```

```
5         余额-=金额;
6 }
```

为什么要重构

返回错误代码是程序的过时保留
编程。在现代编程中,错误处理由称为异常的特殊类构成。如果一个

问题发生,你“抛出”一个错误,然后被“捕获”
由异常处理程序之一。特殊的错误处理代码,
在正常情况下被忽略,被激活以响应。

好处

- 将代码从大量检查条件中解放出来
  各种错误代码。异常处理程序是区分正常执行路径和异常执行路径
  的更简洁的方法。

- 异常类可以实现自己的方法,因此包含部分错误处理功能(例如

  发送错误消息)。

- 与异常不同,错误代码不能在构造函数中使用,
  因为构造函数必须只返回一个新对象。

缺点

异常处理程序可以变成类似 goto 的拐杖。避免这种情况!不要使用异常来管理代码执行。只应抛出异常以通知错误或严重

情况。

如何重构

尝试一次只针对一个错误代码执行这些重构步骤。这将使您更容易将所有重要信息保留在您的脑海中并避免错误。

1. 查找对返回错误代码的方法的所有调用,而不是
   检查错误代码,将其包装在try / catch块中。

2. 在方法内部,不是返回错误代码,而是抛出异常。

3.更改方法签名,使其包含信息
   关于抛出的异常（ @throws部分）。

B替换异常带测试

问题

你在一个简单的测试会抛出异常的地方做这份工作？

```
1双getValueForPeriod(int periodNumber) {
2        试试{
3            返回值[周期数];
4        }捕捉（ArrayIndexOutOfBoundsException e){
5            返回0；
6        }
7 }
```

解决方案

用条件测试替换异常。

```
1双getValueForPeriod(int periodNumber) {
2        if (periodNumber >= values.length) {
3            返回0；
4        }
```

```
5         返回值[周期数];
6 }
```

为什么要重构

应使用异常来处理与意外错误相关的不规则行为。它们不应该作为测试的替代品。如果可以通过在运行前简单地验证条件来避免异常，那么就这样做。应为真正的错误保留异常。

比如你进入了雷区，在那里触发了地雷，导致异常；例外是成功处理完毕，你被从空中升到雷区以外的安全地带。但是你可以通过简单地阅读雷区前面的警告标志来避免这一切。

好处

一个简单的条件有时可能比异常处理代码更明显。

如何重构

1. 为边缘情况创建条件并将其移动到 try/catch 块之前。

2. 从这个条件内的catch部分移动代码。

3.在catch部分,放置一个通常的抛出代码
未命名的异常并运行所有测试。

4. 如果在测试过程中没有抛出异常,去掉try / catch操作符。

类似的重构

§ 用异常替换错误代码

处理泛化

抽象有自己的一组重构技术,主要与沿类继承层次结构移动功能、创建新类和接口以及用委托替换继承相关,反之亦然。

§ 引体向上

问题:两个类具有相同的字段。

解决方案:从子类中删除该字段并将其移动到超类。

§ 引体向上法

问题:您的子类具有执行类似工作的方法。

解决方案:使方法相同,然后将它们移动到相关的超类。

§ 拉起构造函数体

问题:你的子类的构造函数的代码是大多相同。

解决方案:创建一个超类构造函数,并将子类中相同的代码移到它上面。调用超类 con

子类构造函数中的构造函数。

§ 下推法

问题:行为是否在仅由一个（或几个)子类使用的超类中实现?

解决方案:将此行为移至子类。

§ 下推场

问题:一个字段是否只在几个子类中使用?

解决方案:将字段移至这些子类。

§ 提取子类

问题:一个类具有仅在 cer 中使用的特性
tain 案件。

解决方案:创建一个子类并在这些情况下使用它。

§ 提取超类

问题:您有两个具有公共字段的类和
方法。

解决方案:为它们创建一个共享超类,并将所有相同的字段和方法移至其中。

§ 提取接口

问题:多个客户端使用类接口的同一部分。另一种情况:两个类中的部分接口是

相同。

解决方案:将此相同部分移至其自己的界面。

§ 折叠层次结构

问题:您有一个类层次结构,其中子类实际上与其超类相同。

解决方案:合并子类和超类。

§ 表格模板法

问题:您的子类以相同的顺序实现包含相似步骤的算法。

解决方案:将算法结构和相同的步骤移到超类中,将不同步骤的实现留在子类中。

§ 用委托代替继承

问题:您的子类只使用了其超类的部分方法(或者无法继承超类数据)。

解决方法:创建一个字段,在里面放一个超类对象,把方法委托给超类对象,去掉

遗产。

§ 用继承代替委托

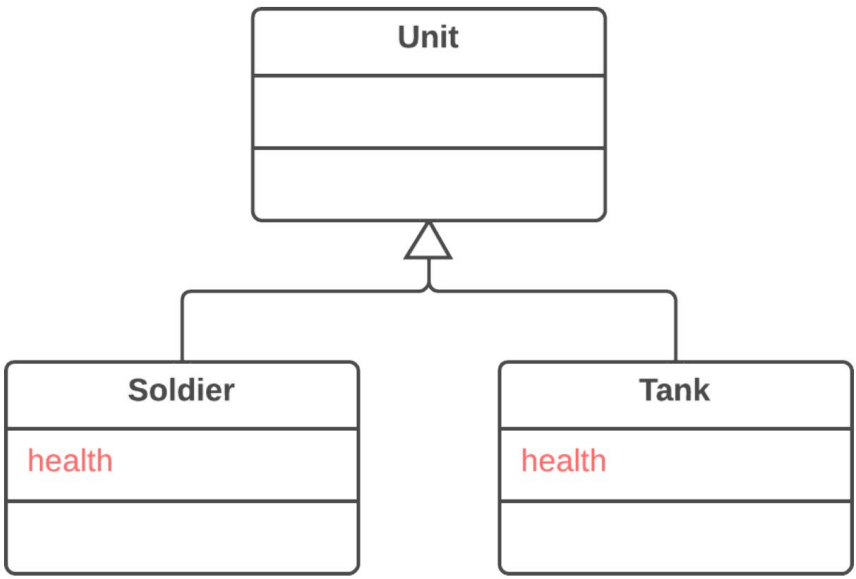
问题:一个类包含许多简单的方法,这些方法委托给另一个类的所有方法。

解决方案:使类成为委托继承者,这使得委托方法变得不必要。

B上拉场

问题

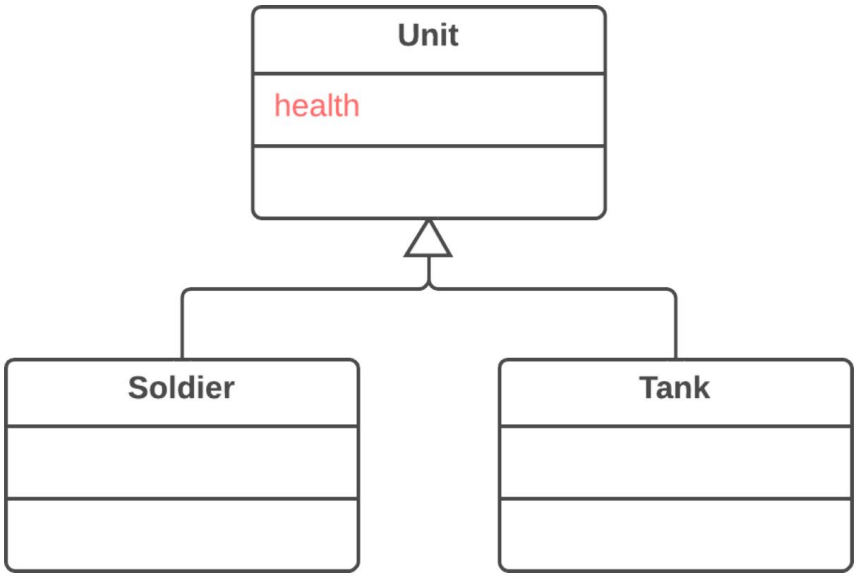
两个类具有相同的字段。



解决方案

从子类中删除该字段并将其移动到

超类。



为什么要重构

子类分别成长和发展,导致出现相同（或几乎相同)的领域和方法。

好处

- 消除子类中字段的重复。
- 简化重复方法的后续重定位,如果它们
  存在,从子类到超类。

如何重构

1.确保字段用于相同的需求
  子类。

2.如果字段有不同的名字,给他们相同的名字
并替换现有代码中对字段的所有引用。

3. 在超类中创建一个同名字段。请注意,如果字段是私有的,则超类字段应该受到保护。

4. 从子类中删除字段。

5. 您可能需要考虑使用自封装字段
新字段,以便将其隐藏在访问方法后面。

反重构

§ 下推场

类似的重构

§ 引体向上法

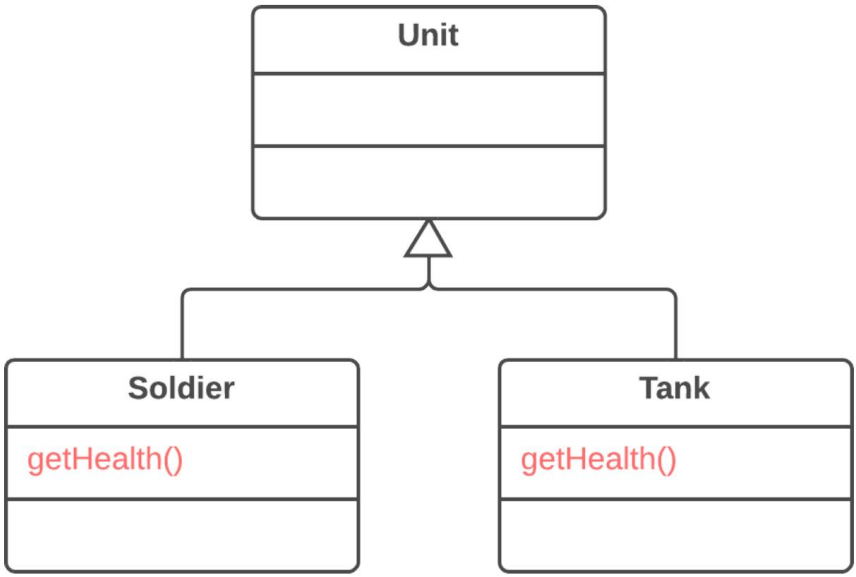
消除异味

§ 重复代码

B引体向上法

问题

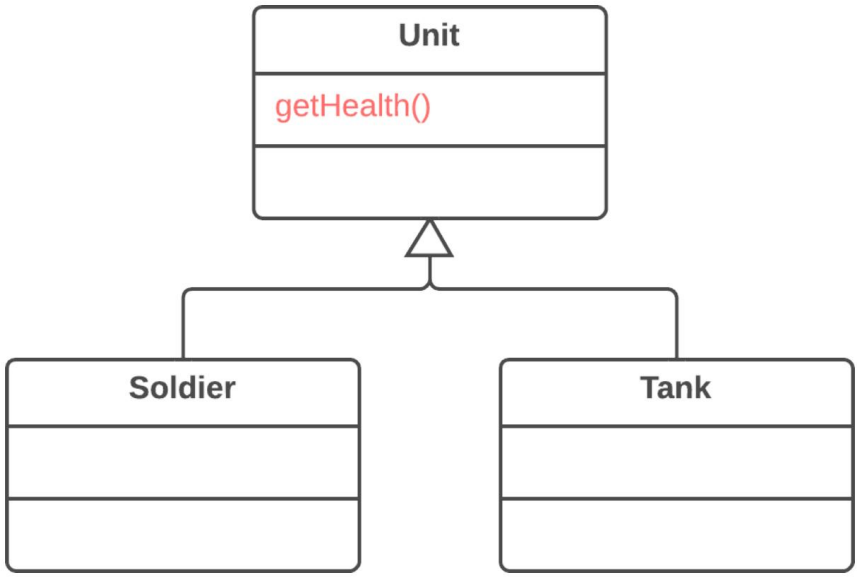
您的子类具有执行类似工作的方法。



解决方案

使方法相同,然后将它们移至

相关的超类。



为什么要重构

子类彼此独立地成长和发展,导致相同（或几乎相同)的领域和方法。

好处

- 摆脱重复代码。如果您需要对方法进行更改,最好在一个地方进行,而不是在子类中搜索该方法的所有重复项。

- 如果出于某种原因,子类重新定义了超类方法但执行的工作基本相同,则也可以使用这种重构技术。

如何重构

1. 研究超类中的类似方法。如果他们不是
   相同,格式化它们以相互匹配。

2. 如果方法使用不同的参数集,请将参数
   您希望在超类中看到的形式。

3. 将方法复制到超类。在这里,您可能会发现方法代码使用仅存在于子类中的字段和方法,
   因此在超类中不可用。要解决此问题,您可以:

   - 对于字段:使用 Pull Up Field或 Self-Encapsulate Field 在子类中创建
     getter 和 setter;然后在超类中抽象地声明这些 getter。

   - 对于方法:使用 Pull Up Method 或在超类中为它们声明抽象方法(请注意,如果
     之前没有,您的类将变为抽象)。

4. 从子类中删除方法。

5. 检查调用方法的位置。在一些
   您可以用超类替换子类的使用。

反重构

§ 下推法

类似的重构

§ 引体向上

帮助其他重构

§ 表格模板法

消除异味

§ 重复代码

B上拉构造函数身体

问题

您的子类的构造函数的代码主要是
完全相同的。

```
1类经理扩展员工{
2        公共经理（字符串名称,字符串ID,整数等级） {
3            this.name = 名称;
4            这个.id = id;
5            this.grade = 等级;
6        }
7        // ...
8 }
```

解决方案

创建一个超类构造函数并移动代码
在它的子类中相同。调用超类构造函数
子类构造函数。

```
1类经理扩展员工{
```

```
2      公共经理（字符串名称,字符串ID,整数等级）{
3          超级（姓名,身份证）；
4          this.grade = 等级;
5      }
6      // ...
7  }
```

为什么要重构

这种重构技术与 Pull Up 有何不同
方法？

1.在Java中,子类不能继承构造函数,所以不能
只需将 Pull Up Method 应用于子类构造函数,然后
在将所有构造函数代码删除到超类后将其删除。除了在超类中创建构造函数之外

必须在子类中使用简单的构造函数
委托给超类构造函数。

2. 在 C++ 和 Java 中（如果你没有显式调用超类
构造函数)超类构造函数被自动调用
在子类构造函数之前,这使得它有必要
仅从开头移动公共代码
子类构造函数（因为你不能调用
来自子类中任意位置的超类构造函数
构造函数）。

3. 在大多数编程语言中,子类构造函数可以
有自己的参数列表,不同于参数