重构-改善既有代码的设计：简化函数调用 （八）

1. Rename Method 函数改名

函数的名称未能揭示函数的用途。**修改函数名称。**

大力提倡的一种编程风格是：将复杂的处理分解成小函数。但是，如果做得不好，这会使你费尽周折却弄不清楚这些小函数各自的用途。要避免这种麻烦，关键就在于给函数起一个好名称。函数的名称应该准确表达它的用途。给函数命名有一个好办法：首先考虑应该给这个函数写上一句怎样的注释，然后想办法将注释变成函数名称。

你常常无法第一次就给函数起一个好名称。如果你看到一个函数名称不能很好地表达它的用途，应该马上加以修改。你的代码首先是为人写的，其次才是为计算机写的。而人需要良好名称的函数。如果给每个函数都起一个良好的名称，也许你可以节约好多时间。起一个好名称并不容易，需要经验；要想成为一个真正的编程高手，起名的水平至关重要。当然，函数签名中的其他部分也一样重要。如果重新安排参数顺序，能够帮助提高代码的清晰度，那就大胆地去做。还有 Add Parameter （添加参数）和 Remove Parameter （移除参数）这 2 项武器。

2. Add Parameter 添加参数

某个函数需要从调用端得到更多信息。**为此函数添加一个对象参数，让该对象带进函数所需信息。**

Add Parameter （添加参数）是一个很常用的重构手法。使用这项重构的动机很简单：你必须修改一个函数，而修改后的函数需要一些过去没有的信息，因此你需要给该函数添加一个参数。

需要说明的是：不使用本项重构的时机。除了添加参数外，你常常还有其他选择。只要可能，其他选择都比添加参数要好，因为它们不会增加参数列的长度。过长的参数列是不好的味道，因为程序员很难记住那么多参数而且长参数列往往伴随着坏味道：数据泥团（Data Clumps）。

请看看现有的参数，然后问自己：你能从这些参数得到所需的信息吗？如果回答是否定的，有可能通过某个函数提供所需信息吗？你究竟把这些信息用于何处？这个函数是否应该属于拥有该信息的那个对象所有？看看现有参数，考虑一下，加入新参数是否合适？也许你应该考虑使用 Introduce Parameter Object （引入参数对象）。

3. Remove Parameter 移除参数

函数本体不再需要某个函数。**将该参数去除。**

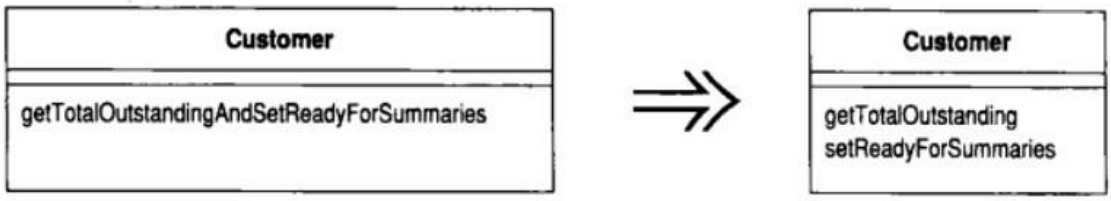
程序员可能检查添加参数，却往往不愿意去掉它们。他们打的如意算盘是：无论如何，多余的参数不会引起任何问题，而且以后还可能用上它。

参数代表着函数所需的信息，不同的参数值有不同的意义。函数调用者必须为每一个参数操心该传什么东西进去。如果你不去掉多余参数，就是让你的每一位用户多费一份心。是很不划算的，更何况“去除参数”是非常简单的一项重构。

但是，对于多态函数，情况有所不同。这种情况下，可能多态函数的另一份实现会使用这个参数，此时你就不能去除它。你可以添加一个独立函数，在这些情况下使用。不过你应该先检查调用者任何使用这个函数，以决定是否值得这么做。如果某些调用者已经知道他们正在处理的是一个特定的子类，并且已经做了额外工作找出自己需要的参数，或已利用对类体系的了解来避免取到 null，那么就值得建立一个新函数，去除那多余的参数。如果调用者不需要了解函数所属的类，你也可以继续保持调用者无知而幸福的状态。

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。**建立 2 个不同的函数，其中一个负责查询，另一个负责修改。**



如果某个函数只是向你提供一个值，没有任何看得到的副作用，那么这是个很有价值的东西。你可以任意调用这个函数，也可能把调用动作搬到函数的其他地方。明确表现出”有副作用”与“无副作用”2 种函数之间的差异，是个很好的想法。任何有返回值的函数，都不应该有看得到的副作用。有些程序员甚至将此作为一条必须遵守的规则。

如果你遇到一个“既有返回值又有副作用”的函数，就应该试着将查询动作从修改动作中分割出来。

有一种常见的优化办法是：将查询所得结果缓存于某个字段中，这么一来后续的重复查询就可以大大加快速度。虽然这种做法改变了对象的状态，但这一修改是觉察不到的，因为不论任何查询，你总是获得相同的结果。

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。**建立一个单一函数，以参数表达那些不同的值。**

动机：你可能会发现这样的 2 个函数：它们做着类似的工作，但因少数几个值致使行为略为不同。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，用以简化问题。这样的修改可以去除重复代码，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。

6.Replace Parameter 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香味。**针对该参数的每个可能值，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;
}
```

Replace Parameter with Explicit Methods （以明确函数取代参数）恰恰相反于 Parameterize Method （令函数携带参数）。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，而且接口也很清楚。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地接口，也值得执行本项重构。哪怕只是给一个内部的布尔变量赋值，相较之下，switch。BeOn( )也比 Switch.SetState() 要清楚的多。

但是，如果参数值不会对函数行为有太多影响，就不应该使用 Replace Parameter with Explicit Methods （以明确函数取代参数）。如果情况真是这样，而你也只需要通过参数为一个字段赋值，那么直接使用设值函数好了。如果的确需要条件判断的行为，可考虑使用 Replace Conditional with Polymorphism （以多态取代条件表达式）。

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。**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());
```

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外，Preserve Whole Object （保持对象完整）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 2 面：如果你传的是数值，被调用函数就只依赖于这些数值，而不依赖它们所属的对象。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（保持对象完整）。

有的观点认为：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。这个观点不能被认同：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一致的。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项，这可能意味着该函数实际上应该被定义在那些数据所属的对象中。所以，考虑使用 Preserve Whole Object （保持对象完整）同时，你也该考虑 Move Method（搬移函数）。

运用本项重构前，你可能还没有定义一个完整对象，那么就应该先使用 Introduce Parameter Object （引入参数对象）。

还有一种常见情况：调用者将自己的若干数值作为参数，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，并且无需操心对象依赖问题。

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。**让参数接受者去除该项参数，并直接调用前一个函数。**

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

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

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，因此应该尽可能缩短参数列的长度。

缩减参数列的办法之一就是：看看参数接受端是否可以通过与调用端相同的计算来取得参数值。如果调用端通过其所属对象内部的另一个函数来计算参数，并在计算过程中未曾引用调用端的其他参数，那么就应该可以将这个计算过程转移到被调用端，从而去除该项参数。如果所调用的函数隶属另一个对象，而该对象拥有调用端所属对象的引用，前面所说的这些也同样适用。

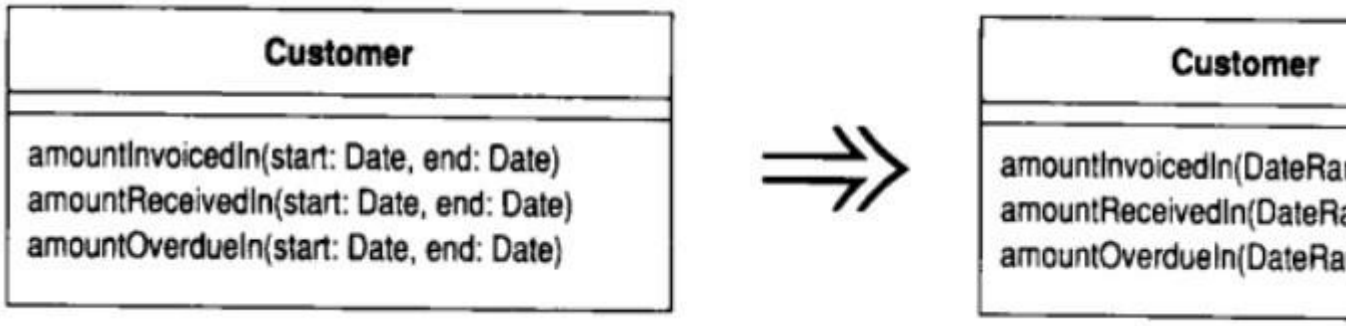
但是，如果参数值的计算过程依赖于调用端的某个参数，那么就无法去掉被调用端的参数，因为每次调用动作中，该参数值可能不同。另外，如果参数接受端并没有参数发送端对象的引用，而你也不想加上这样一个引用，那么也无法去除参数。

有时候，参数的存在是为了将来的灵活性。这种情况下仍然可以把这种多余参数拿掉。你应该只在必要关头才添加参数，预先添加的参数很可能并不是你所需要的。对于这条规则，有个例外：如果修改接口会对整个程序造成非常痛苦的结果，那么可以考虑保留前人预先加入的参数。如果真是这样，应该首先判断修改接口究竟会造成多严重的后果，然后考虑是否

应该降低给部位之间的依赖，以减少修改接口所造成的影响。稳定的接口确实很好，但是被冻结在一个不良接口上也是一个问题。

9. Introduce Parameter Object 引入参数对象

某些参数总是很自然地同时出现。**以一个对象取代这些参数。**



你常常会看到特定的一组参数总是被一起传递。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，也可能隶属不同的类。这样一组参数就是所谓的 Data Clumps（数据泥团），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，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.Remove setting Method 移除设置函数

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。**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，这就暗示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。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。

如果你保留了间接访问变量的方法，就可能经常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。

11.Hide Method 隐藏函数

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。**

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。理想状态下，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，你会发现许多设值/取值函数不再需要被公开，因此可以将它们隐藏起来。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，那就可以放心移除取值/设值函数了。

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。**将构造函数替换为工厂函数。**

```
Employee (int type) {
_type = type;
}
```

```
static Employee create(int type) {
return new Employee(type);
}
```

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代替它。工厂函数也是 Change Value to Reference （将值对象改为引用对象）的基础。你也可以令你的工厂函数根据参数的个数和类型，选择不同的构建行为。

做法：1、新建一个工厂函数，让它调用现有的构造函数。

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）。**将向下转型动作移到函数中。**

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

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

动机：向下转型也许是无法避免的，但你仍然应该尽可能少做。如果你的某个函数返回一个值，并且你知道所返回的对象类型比函数签名所昭告的更特化，你便是在函数用户身上强加了非必要的工作。这种情况下你不应该要求用户承担向下转型的责任，应该尽量为他们提供准确的类型。

以上所说的情况，常会在返回迭代器或集合的函数身上发生。此时你就应该观察人们拿这个迭代器干什么用，然后针对性地提供专用函数。

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。**改用异常**。

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

⇓

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

程序中发现错误的地方，并不一定知道如何处理错误。当一段子程序发现错误时，它需要让它的调用者知道这个错误，而调用者也可能将这个错误继续沿着调用链传递上去。许多程序都使用特殊输出来表示错误。

可以使用更好的错误处理方式：异常。它清楚地将“普通程序”和“错误处理”分开了，这使得程序更容易理解：代码的可理解性应该是我们追求的目标。

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];
}
```

动机：异常的出现是程序语言的一大进步。但是，就像许多好东西一样，异常会被滥用，从而变得不再让人愉快。“异常”只应该被用于异常的、罕见的行为，也就是那些产生意料之外的错误的行为，而不应该成为条件检查的替代品。如果你可以合理期望调用者在调用函数之前检查某个条件，那么就应该提供一个测试，而调用者应该使用它。