重构-改善既有代码的设计：重新组织数据的 16 种方法（六）

1.Self Encapsulate Field 自封装字段

间接访问类的属性：你直接访问一个字段，但与字段之间的耦合关系逐渐变得笨拙。为这个字段建立取值/设值函数，并且只以这些函数来访问字段。

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

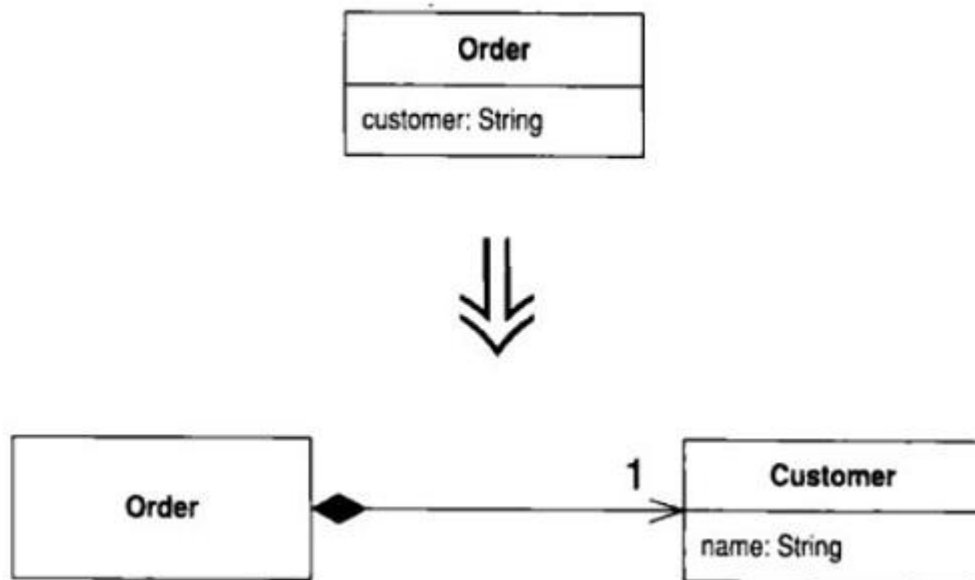
```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

间接访问变量的好处是，子类可以通过覆写一个函数而改变获取数据的途径；它还支持更灵活的数据管理方式，例如延迟初始化。

如果你想访问超类中的一个字段，却又想子类中将对这个变量的访问改为一个计算后的值，这就是使用 Self Encapsulate Field （自封装字段）的时候。“字段自我封装”只是第一步。完成自我封装后，你可以在子类中根据自己的需要随意覆写取值/设值函数。

2. Replace Data Value with Object 对象取代数据值

聚合改为组合：你有一个数据项，需要与其他数据和行为一起使用才有意义。将数据项变成对象。

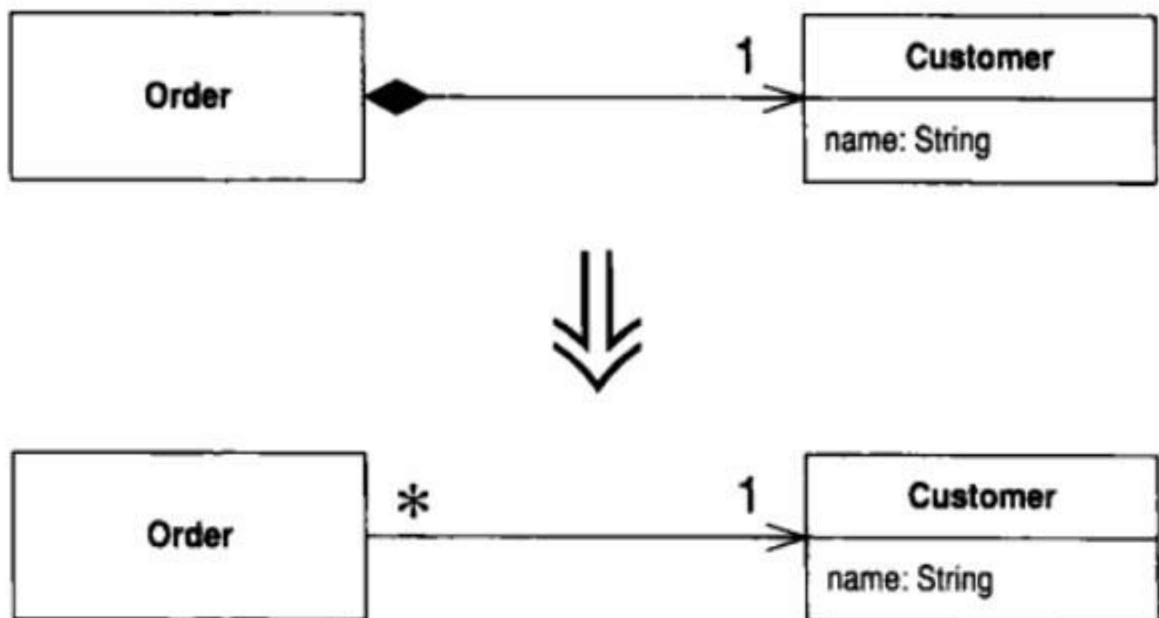


开发初期，你往往决定以简单的数据项表示简单的情况。但是，随着开发的进行，你可能会发现，这些简单数据项不再那么简单了。如果这样的数据项只有一两个，你还可以把相关函数放进数据项所属的对象里；但是重复代码（Duplicated Code）坏味道和 依恋情结（Feature Envy）坏味道很快就会从代码中散发出来，当这些坏味道开始出现，就应该将数据值变成对象。

3.Change value to Reference 将值对象改为引用对象

组合改为重数性关联（另一个类的一个对象与一个或多个该类对象有关系 **Customer 类对象可以包含多个 order 类的对象。）：从一个类中衍生出许多彼此相等的实例，希望将它们替换为一个对象。将这个值对象变成引用对象。（Customer 是一个实值对象 就算多份订单属于一个客户每个 Order 对象还是拥有各自的 Customer 对象。**

重构的结果是所有的订单对象共享一个客户名称。也就是，一个客户对象只对应一个客户名称。）



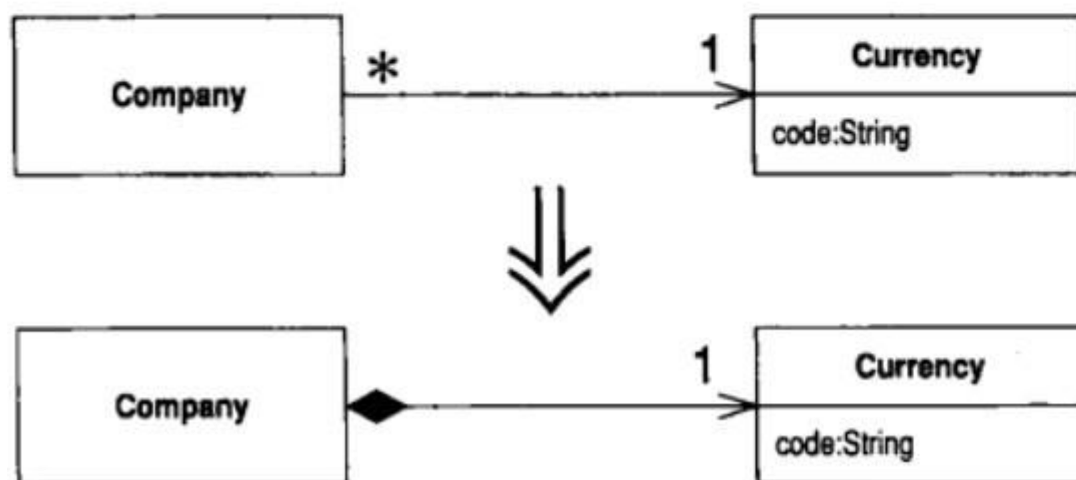
可以将对象分成两类 :reference object(引用对象)和 value object(实值对象)。有一个实值对象，在其中保存了少量的不可修改的数据。你需要给这个对象加入一些可修改的数据 ,并确保对任何一个对象的修改都能影响到所有引用此对象的地方。这是后就需要将 value object（实值对象）变成一个 reference object（引用对象）。

在许多系统中，都可以对对象做一个有用的分类：引用对象和值对象。要在引用对象和值对象之间做选择有时并不容易。有时候，你会从一个简单的值对象开始，在其中保存少量不可修改的数据。而后，你可能会希望给这个对象加入一些可修改数据，并确保对任何

一个对象的修改都能影响到所引用此对象的地方。这时候你就需要将这个对象变成引用对象。

4. Change Reference to Value 将引用对象改为值对象

你有一个引用对象，很小且不可变，而且不易管理。将它变成一个值对象。



要在引用对象和值对象之间做选择，有时并不容易。做出选择后，你常会需要一条回头路，

如果引用对象开始变得难以使用，也许就应该将它改为值对象。引用对象必须被某种方式控制，你总是必须向其控制者请求适当的引用对象。它们可能造成内存区域之间错综复杂的关联。在分布式和并发系统中，不可变的值对象特别有用，因为你无需考虑它们的同步问题。

值对象有一个非常重要的特征：它们应该是不可变的。无论何时，只有你调用同一对象的同一个查询函数，都应该得到同样结果。如果保证了这一点，就可以放心地以多个对象

表示同一个事物。如果值对象是可变的，你就必须确保某个对象的修改会自动更新其他“代表相同事物”的对象。与其如此还不如把它变成引用对象。

澄清下“不可变”（immutable）的意思：如果你以 money 类表示“钱’的概念，其中有”币种“和”金额“2 条信息。那么 Money 对象通常是一个不可变的值对象。这并非意味你的薪资不能改变，而是意味：如果要改变你的薪资，就需要使用另一个 Money 对象来取代现有的 Money 对象，而不是在现有的 Money 对象上修改。你和 Money 对象之间的关系可以改变，但 Money 对象自身不能改变。

5.Replace Array with Object 以对象取代数组

你有一个数组，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。

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

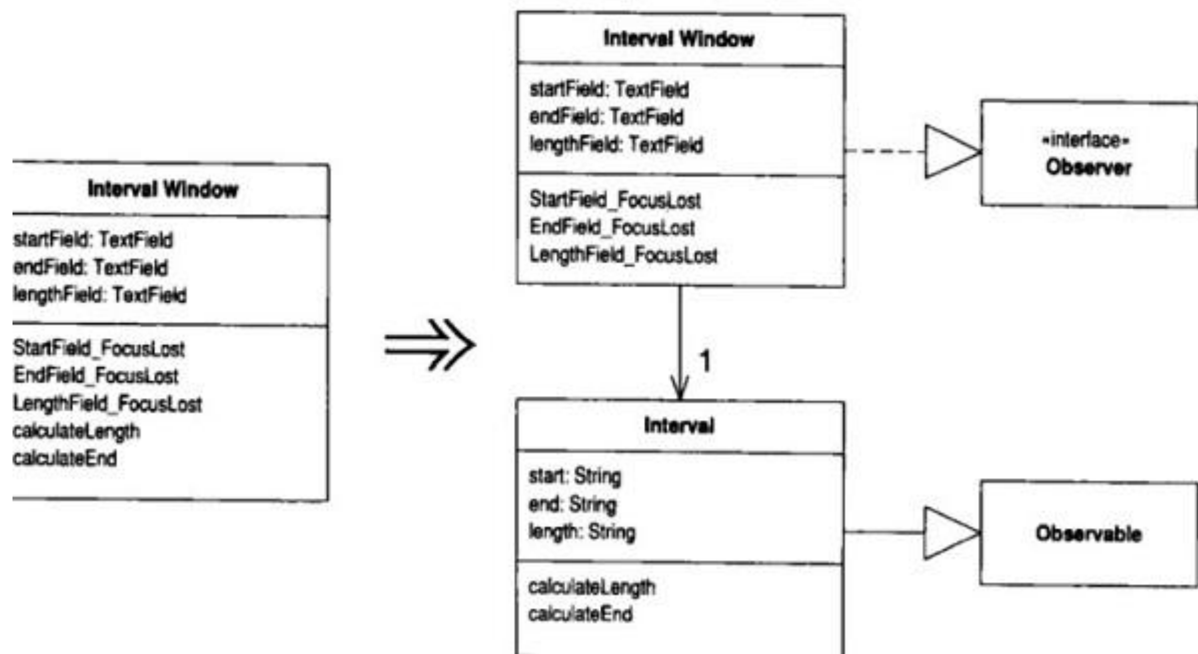
```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

数组时一种常见的用以组织数据的结构。不过，它们应该只用于“以某种顺序容纳一组相似对象”。有时候你会发现，一个数组容纳了多种不同对象，这会给用户带来麻烦，因为他们很难记住像“数组的第一个元素是人名”这样的约定。对象就不同了，你可以运用字段名和

函数名来传达这样的信息，因此你无需死记它，也无需依赖注释。而且如果使用对象，你还可以将信息封装起来。并使用 Move Method （搬移函数）为它加上相关行为。

6. Duplicate Observed data 复制被监视数据

你有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。将该数据复制到一个领域对象中。建立一个 Observer 模式，用以同步领域对象和 GUI 对象内的重复数据。



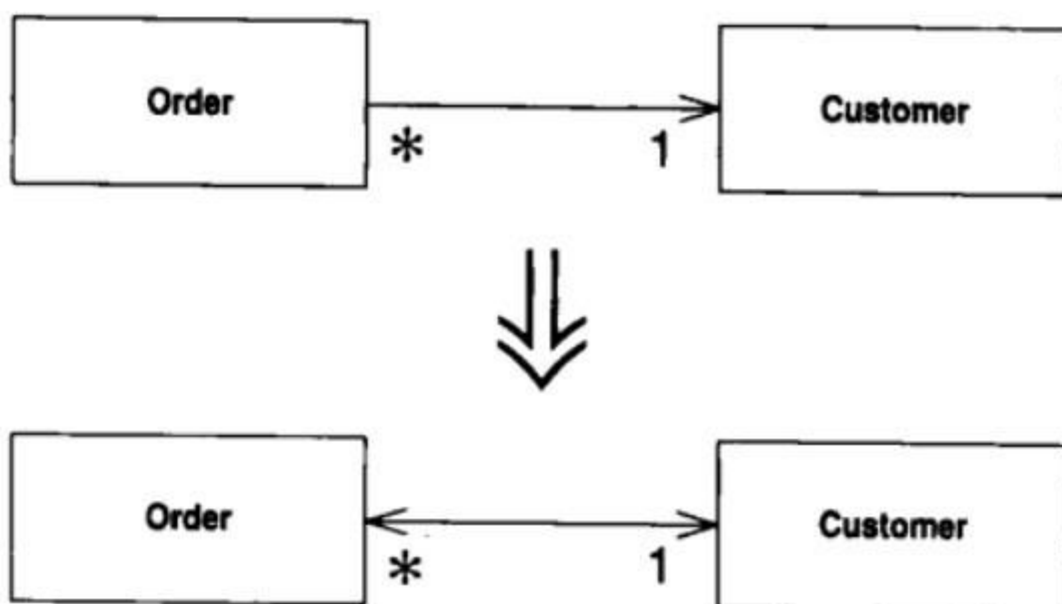
一个分层良好的系统，应该将处理用户界面和处理业务逻辑的代码分开。之所以这样做，原因有以下几点：1）你可能需要使用不同的用户界面来表现相同的业务逻辑，如果同时承担 2 种责任，用户界面会变得过分复杂；2）与 GUI 隔离后，领域对象的维护和演化都会更容易，你甚至可以让不同的开发者负责不同部分的开发。

尽管可以轻松地将“行为”划分到不同部位，“数据”却往往不能如此。同一项数据可能既需要内嵌于 GUI 控件，也需要保存于领域模型里。自从 MVC 模式出现后，用户界面框架都使用多层系统来提供某种机制，使你不但可以提供这类数据，并保持它们同步。

如果你遇到的代码是以 2 层方式开发，业务逻辑被内嵌于用户界面之中，你就有必要将行为分离出来。其中的主要工作就是函数的分解和搬移。但数据就不同了；你不能仅仅只是移动数据，必须将它复制到新的对象中，并提供相应的同步机制。

7.Change Unidirection Association to Bidirectional 将单向关联改为双向关联

两个类都需要使用对方特性，但其间只有一条单向连接。**添加一个反向指针，并使修改函数能够同时更新 2 条连接。**



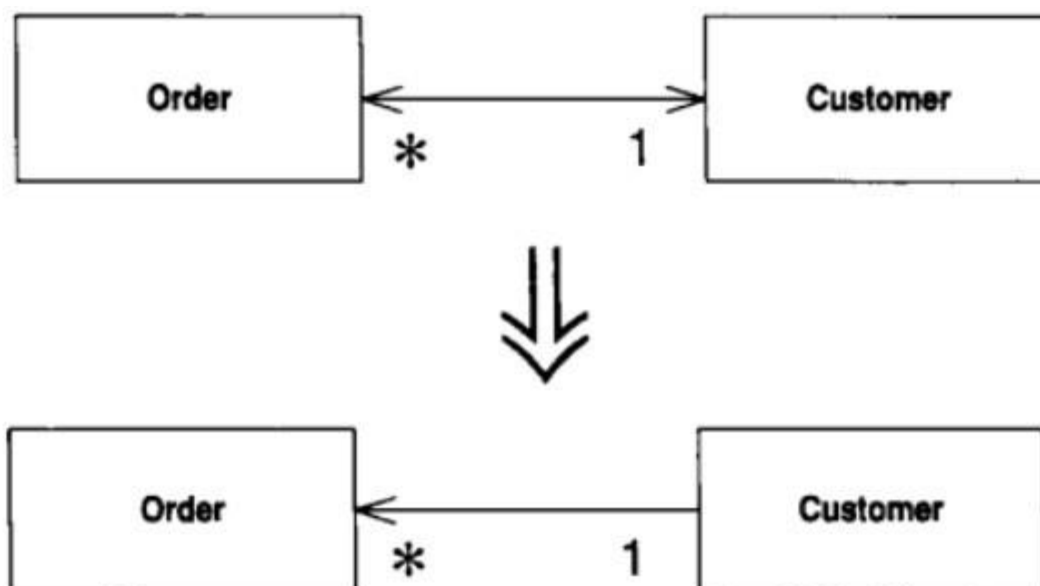
开发初期，你可能会在 2 个类之间建立 1 条单向连接，使其中一个类可以使用另一个类。随着时间推移，你可能发现被引用类需要得到其引用者以便进行某些处理。也就是说它需要一个反向指针。但指针是一种单向连接，你不可能反向操作它。通常你可以绕道而行，虽然会耗费一些计算时间，成本还算合理，然后你可以在被引用类中建立一个函数专门负责此行为。但是，有时候想绕过这个问题并不容易，此时就需要建立双向引用关系，或称为反向指针。如果使用不当，反向指针和很容易造成混乱；但只要你习惯了这种手法，它们其实并不太复杂。

“反向指针”手法有点棘手，所以在你能够自如运用之前，应该有相应的测试。通常不用花心思去测试访问函数，因为普通访问函数的风险没有高到需要测试的地步，但本重构要求测试访问函数，所以它是极少数需要添加测试的重构手法之一。

本重构运用反向指针实现双向关联，其他技术需要其他重构手法。

8.Change Bidirectional Association to Unidirection 将双向关联改为单向关联

两个类之间有双向关联，但其中一个类如今不再需要另一个类的特性。**去除不必要的关联。**



双向关联很有用，但你必须为它付出代价，那就是维护双向连接、确保对象被正确创建和删除而增加的复杂度。而且，由于很多程序员并不习惯使用双向关联，它往往成为错误之源。

大量的双向连接也很容易造成“僵尸对象”：某个对象本来应该死亡了，却仍然保留在系统中，因为对它的引用没有完全清除。

此外，双向关联也迫使 2 个类之间有了依赖：对其中任一个类的任何修改，都可能引发另一个类的变化。如果这 2 个类位于不同的程序集，这种依赖就是程序集之间的相依。过多的跨程序集依赖会造就紧耦合的系统，使得任何一点小小改动就可能造成许多无法预知的后果。

只有在真正需要双向关联的时候，才该使用它。如果发现双向关联不再有存在价值，就应该去掉不必要的一条关联。

9.Replace Magic Number with Symbolic Constant 字面常量取代魔法数

你有一个字面数值，带有特别含义。创建一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

```
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}
```

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

在计算科学中，魔法数是历史悠久的不良现象之一。所谓魔法数是指拥有特殊意义，却又不能明确表现出这种意义的数字。如果你需要在不同的地点引用同一个逻辑数，魔法数会让你烦恼不已，因为一旦这些数发生变化，你就必须在程序中找到所有魔法数，并将它们全部修改一遍。就算你不需要修改，要准确指出每个魔法数的用途，也会让你颇费脑筋。

许多语言都允许声明常量。常量不会造成任何性能开销，却可以大大提高代码的可读性。

进行本项重构之前，你应该先寻找其他替代方案。你应该观察魔法数任何被使用，而后你往往会发现一种更好的使用方式。如果这个魔法数是个类型码，请考虑使用 Replace Type Code with Class （以类取代类型码）；如果这个魔法数代表一个数组的长度，请在遍历数组时，改用数组.length。

10.Encapsulate Field 封装字段

你的类中存在一个 public 字段。将它声明为 private，并且提供相应的访问函数。

```
public String _name;
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

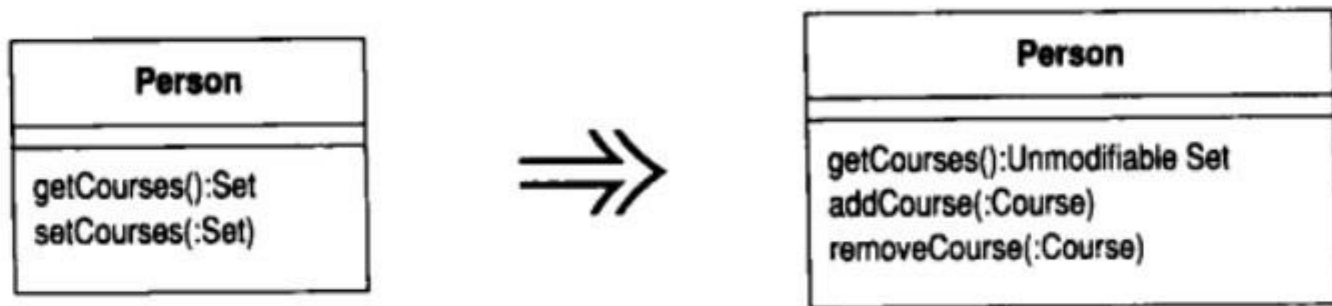
面向对象的首要原则之一就是封装，或者称为“数据隐藏”。按此原则，你绝不应该将数据声明为 public，否则其他对象就有可能访问甚至修改这项数据，而拥有该数据的对象却毫无察觉。于是，数据和行为就被分开了。

数据声明为 public 被看做是一种不好的做法，因为这样会降低程序的模块化程度。数据和使用该数据的行为如果集中在一起，一旦情况发生变化，代码的修改就会比较简单，因为需要修改的代码都集中于同一块地方，而不是星罗棋布地散落在整个程序中。

Encapsulated Field （封装字段）是封装过程的第一步，通过这项重构手法，你可以将数据隐藏起来，并通过相应的访问函数。但它毕竟只是第一步。如果一个类除了访问函数外不能提供其他行为，它终究只是一个哑巴类。这样的；类并不能享受对象技术带来的好处。实施 Encapsulated Field （封装字段）之后，尝试寻找用到新建访问函数的代码，看看是否可以通过简单的 Move Method(搬移函数)将它们移到新对象去。

11.Encapsulate Coolection 封装集合

有一个函数返回一个集合。让这个函数返回该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。



我们常常会在一个类中使用集合来保存一组实例。这样的类通常也会提供针对该集合的取值/设值函数。

但是，集合的处理方式应该和其他种类的数据略有不同。取值函数不该返回集合自身，因为这会让用户得以修改集合内容而集合拥有者却一无所悉。也会对用户暴露过多对象内部数据结构信息。如果一个取值函数确实需要返回多个值，它应该避免用户直接操作对象内所保存的集合。并隐藏对象内与用户无关的数据结构。

另外，不应该为这整个集合提供设值函数，但应该提供用以为集合添加/移除元素的函数。这样，集合拥有者就可以控制集合元素的添加和移除。

如果你做到以上几点，集合就被很好地封装起来，这便可以降低集合拥有者和用户之间的耦合度。

12.Replace Record with Data Class 以数据类取代记录

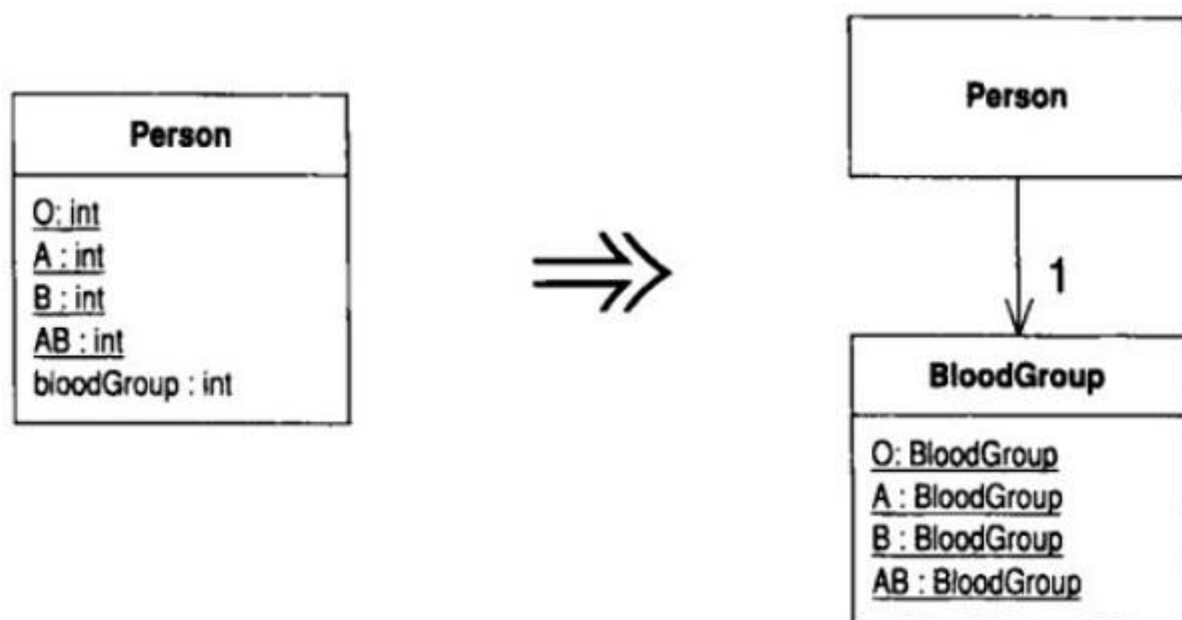
你需要面对传统编程环境中的记录结构。为该记录创建一个“哑”数据对象。

记录型结构是许多编程环境的共同性质。有一些理由使它们被带进面向对象程序之中：你可能面对的是一个遗留程序，也可能需要通过一个传统 API 来与记录结构交流，或是处理从数据库读出的记录。这些时候你就有必要创建一个接口类，用以处理这些外来数据。最简单的做法就是先建立一个看起来类似外部记录的类，以便日后将某些字段和函数搬移到

这个类中。一个不太常见但非常令人注目的情况是：数组中的每个位置上的元素都有特定含义，这种情况下应该使用 Replace Array with Object （以对象取代数组）。

13. Replace Type Code with Class 以类来取代类型码

类之中有一个数值类型码，但它并不影响类的行为。以一个新的类替换该数值类型码。



在以 C 为基础的编程语言中，类型码或枚举值很常见。如果带着一个有意义的符号名，类型码的可读性还不错。问题在于，符号名终究只是个别名，编译器看见的、进行类型检验的，还是背后那个数值。任何接受类型码作为参数的函数，所期望的实际上是一个数值，无法强制使用符号名。这会大大降低代码的可读性，从而成为 bug 之源。

如果把那样的数值换成一个类，编译器就可以对这个类进行类型检验。只要为这个类提供工厂函数，你就可以始终保证只有合法的实例才会被常见出来，而且他们都会被传递给正确的宿主对象。

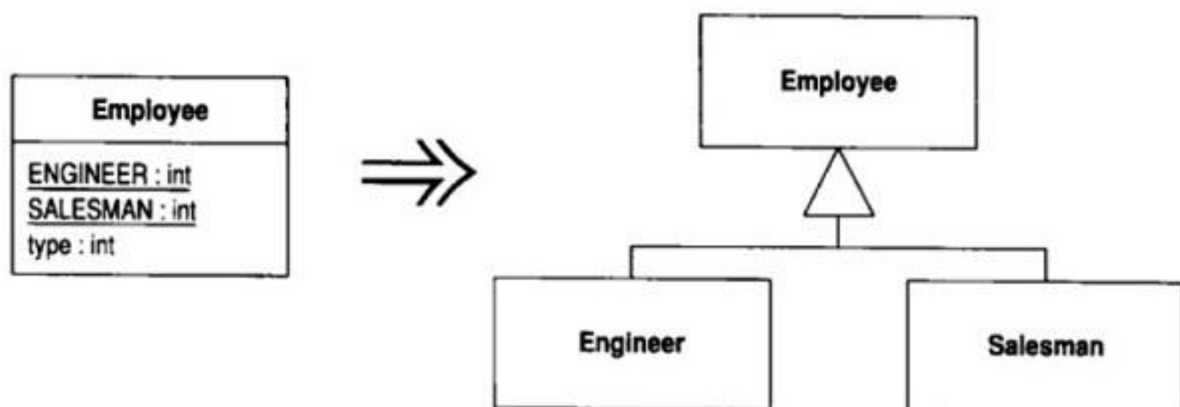
但是，在使用 Replace Type Code with Class （以类取代类型码）之前，你应该先考虑类型码的其他替换方式。只有当类型码是纯粹数据时（也就是类型码不会在 switch 语句

中引起行为变化时），你才能以类来取代它。更重要的是：任何 switch 语句都应该运用 Replace Conditional with Polymorphism （以多态取代条件表达式）去掉。为了进行那样的重构，你首先必须运用 Replace Type Code with Subclass （以子类取代类型码）或 Replace Type Code with State/Strategy （以状态策略取代类型码），把类型码处理掉。

即使一个类型码不会因其数值的不同而引起行为上的差异，宿主类中的某些行为还是可能更适合放置于类型码类中，因此你还应该留意是否有必要使用 Move Method（搬移函数）将一两个函数搬过去。

14.Replace Type Code with Subclasses 以子类来取代类型码

你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。



如果你面对的类型码不会影响宿主类的行为，可以使用 Replace Type Code with Class（以类取代类型码）来处理它们。但如果类型码会影响宿主类的行为，那么最后的办法就是借助多态来处理变化行为。

一般来说，这种情况的标志就是像 switch 这样的条件表达式。这种条件表达式可能有 2 种表现形式：switch 语句或者 if –then-else 结构。不论哪种形式，它们都是检查类型码

值，并根据不同的值执行不同的动作。这种情况下，你应该以 Replace Conditional with Polymorphism （以多态取代条件表达式）进行重构。但为了那个顺利进行那样的重构，首先应该将类型码替换为可拥有多态行为的继承体系。这样一个继承体系应该以类型码宿主类为基类，并针对每一种类型码建立一个子类。

但是以下 2 种情况你不能那么做 1）类型码值在对象创建之后发生了变化；2）由于某种原因，类型码宿主类已经有了子类。如果你恰好面临这 2 种情况之一，就需要运用 Replace Type Code with StateStrategy （以状态策略取代类型码）。

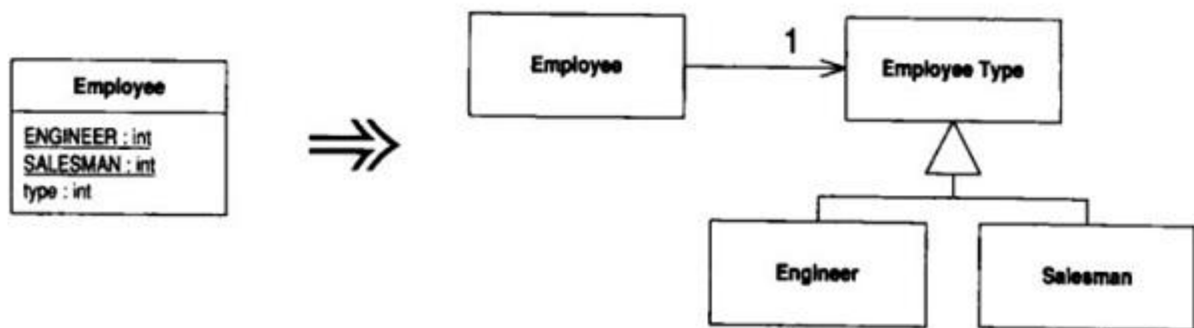
Replace Type Code with Subclass （以子类取代类型码）的主要作用其实是搭建一个舞台，让 Replace Conditional with Polymorphism （以多态取代条件表达式）得以一展身手。如果宿主类中并没有出现条件表达式，那么 Replace Type Code with Class （以类取代类型码）更合适，风险也较低。

使用 Replace Type Code with Subclass （以子类取代类型码）的另一个原因是，宿主类中出现了"只与具备特定类型码之对象相关"的特性。完成本项重构后，你可以使用 push down Method （函数下移）和 push down field （字段下移）将这些特性推到合适的子类中去，以彰显它们只与特定情况相关这一事实。

Replace Type Code with Subclass （以子类取代类型码）的好处在于：它把"对不同行为的了解"从类用户那转移到了类自身。如果需要再加入新的行为变化，只需要添加一个子类就行了。如果没有多态机制，就必须找到所有条件表达式，并逐一修改它们。因此，如果为了还有可能加入新行为，这项重构将特别有价值。

15.Replace Type Code with State/Strategy 以状态/策略取代类型码

你有一个类型码，它会影响类的行为，但你无法提供继承手法消除它。以状态对象取代类型码。

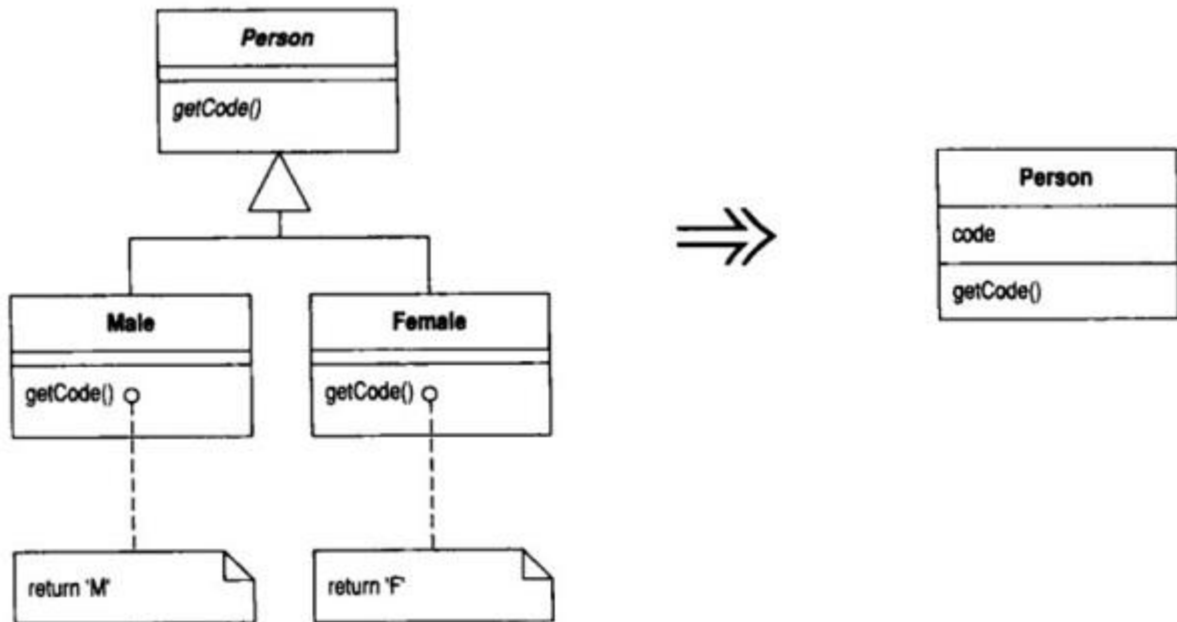


本项重构和 Replace Type Code with Subclass （以子类取代类型码）类似，但如果“类型码在对象生命期中发生变化”或“其他原因使得宿主类不能被继承”，你也可以使用本重构。本重构使用 State 模式和 Strategy 模式。

State 模式和 Strategy 模式非常相似，因此无论你选择其中哪一个，重构过程都是一样的。“选择哪个模式”并非问题的关键所在，你只需要选择更合适特定情境的模式就行了。如果你打算在完成本重构后再以 Replace Conditional with Polymorphism （以多态取代条件表达式）简化一个算法，那么选择 Strategy 模式较合适；如果你打算搬移状态相关的数据，而且你把新建对象视为一种变迁状态，就应该选择 State 模式。

16.Replace Subclass with Fieldls 以字段取代子类

你的各个子类的唯一差别只在“返回常量数据”的函数身上。修改这些函数，使它们返回超类中的某个（新增）只读，然后销毁子类。



建立子类的目的，是为了增加新特性或变化其行为。有一种变化行为被称为"常量函数"，它们会返回一个硬编码的值。这东西有其用途：你可以让不同的子类中的同一个访问函数返回不同的值。你可以在超类中将访问函数声明为抽象函数，并在不同子类中让它返回不同的值。

尽管常量函数有其用途，但若与子类中只有常量函数，实在没有足够的存在价值。你可以在超类中设计一个与常量函数返回值相应的字段，从而完全除去这样的子类。如此一来就可以避免因继承而带来的额外复杂性。