重构-改善既有代码的设计：编写代码 22 宗罪（三）

1 Duplicated Code 重复代码

不同的地方出现相同的程序结构:

如果你在一个以上的地点看到相同的程序结构，那么可以肯定：设法将它们和而为一，程序会变得更好。最常见的“重复代码”就是一个类内的两个函数含有相同的表达式。另一种常见情况就是两个互为兄弟的子类内含有相同的表达式。

1）同一个类的 2 个函数含有相同的表达式，这时可以采用 Extract Method(提炼函数）提炼出重复的代码，然后让这 2 个地点都调用被提炼出来的那段代码。

2）两个互为兄弟的子类内含相同表达式，只需对 2 个类都是用 Extract Method(提炼函数），然后对被提炼出来的函数是用 Pull Up Method （方法上移） ，将它推入超类。如果代码之间只是类似， 并非完全相同，那么就得运用 Extract Method(提炼函数 将相似部分和差异部分隔开，构成单独一个的函数。然后你可能发现可以运用 Form Template Method （塑造模板函数）获得一个 Template Method 设计模式。如果有些函数以不同的算法做相同的事，你可以选择其中较清晰地一个，并是用 Substitute Algorithm （替换算法）将其他函数的算法替换掉。

如果 2 个毫不相关的类出现 重复代码，你应该考虑对其中一个运用 Extract Class （提炼类），将重复代码提炼到一个独立类中，然后在另一个类内使用这个新类。但是，重复代码所在的函数可能只应该属于某个类，另一个类只能调用它，抑或这个函数可能属于第三个类，而另 2 个类应该引用这第三个类。你必须决定这个函数放在哪儿最合适，并确保它被安置后就不会再在其他任何地方出现。

2 Long Method 过长函数

函数中的代码行数原则上不要多于 100 行:

我们遵循这样一条原则：每当感觉需要以注释开说明点什么的时候，我们就需要把说明的东西写进一个独立的函数中，并以其用途（而非实现手法）命名。

90%的场合里，要把函数变小，只需使用 Extract Method(提炼函数) ，找到函数中适合集中在一起的部分，将它们提炼出来形成新函数。

如果函数内有大量的参数和临时变量，它们会对你的函数提炼形成障碍。你可以经常运用 Replace Temp with Query （以查询取代临时变量），来消除这些临时元素。Introduce Parameter Object （引入参数对象），Preserve Whole Object （保持对象完整）则可以将过长的参数列变得简洁一些。

如果已经这么做了，仍然有太多的临时变量和参数，就应该使用 Replace Method with Method Object （以函数对象取代函数）。如何确定提炼哪一段代码呢？一个很好的技巧是：寻找注释。它们通常指出代码用途和实现手法之间的语义距离。如果代码前方有一行注释，就是在提醒你：可以将这段代码替换成一个

函数，而且可以在注释的基础上给这个函数命名。就算只有一行代码，如果它需要以注释来说明，那也值得将它提炼到独立函数中。

条件表达式和循环常常也是提炼的信号。可以使用 Decompose Conditional （分解条件表达式）处理条件表达式。至于循环，应该将循环和其内的代码提炼到独立函数中。（间接层所带来的全部利益---解释能力，共享能力，选择能力--都是由小型函数支持）

3 Large Class 过大的类

类不要负责超越本类的职责,即前面提到的单一原则:

如果想利用单个类做太多的事，其内往往就会出现太多实例变量。这样 重复代码（Duplicated Code）就接踵而至了。

运用 Extract Class （提炼类）将几个变量一起提炼到新类里。提炼类时应该选择类内彼此相关的变量，将它们放在一起。通常类内的数个变量有相同的前缀或字尾，这就意味有机会把它们提炼到某个组件内。如果这个组件适合作为一个子类，就可以运用 Extract Subclass （提炼子类）。

有时类并非在所有时刻都使用所有实例变量。可多次使用Extract Class（提炼类）和Extract Subclass（提炼子类）

一个类如果拥有太多代码，往往也适合使用 Extract Class （提炼类）和 Extract Subclass （提炼子类）。技巧：先确定客户端如何使用它们，然后运用 Extract Interface（提炼接口） 为每个使用方式提炼出一个接口。可帮助你看清楚如何分解这个类。

如果是个 GUI 类，可能需要把数据和行为移到一个独立的领域对象去。可能需要 2 边各保留一些重复数据，并保持 2 边同步。Duplicate Observed Data （复制"被监视数据"）告诉你该这么做。

4. Long Parameter List 过长参数列

过长参数不易理解和维护:

太长的参宿列难以理解，而且会造成前后不一致，不易使用。而且一旦你需要更多数据，就不得不修改它。如果将对象传递给函数，大多数修改都没有必要。例外：不希望造成被调用对象与较大对象间的某种依赖关系。

如果向已有的对象发出 1 条请求就可以取代 1 个参数，那么就运用 Replace Parameter with Methods （以函数取代参数）。在这里，"已有的对象"可能是函数所属类里的 1 个字段，也可能是另一个参数。还可以运用 Preserve Whole Object （保持对象完整）将来自同一对象的一堆数据收集起来，并以该对象替换它们。如果某些数据缺乏合理的对象归属，可使用 Introduce Parameter Object （引入参数对象）为它们制造出一个参数对象。

重要的例外：如果你明显不想造成"被调用对象"与"较大对象"间的某种依赖关系。这时候将数据从对

象中拆解出来单独作为参数，也很合情理。但是请注意其所引发的代价。如果参数列太长或变化太频繁，就需要重新考虑自己的依赖结构了。

5. Divergent Change 发散式变化（相对聚焦式）

修改一个地方需要需要很多处代码.因为修改的代码散布四处:

如果在系统需要修改的时候不能做到只在某一点出做修改，应该意识到代码是否过于紧密耦合的味道了:

例如男人是“聚焦式认知”女人是“发散式认知”.简单点说，聚焦式就是把大面凝聚成一个点，发散式相反它是把一个点扩展成大面

男人可以把几个问题的共性找出来，再再想办法加以解决。男人以解决问题为目的；女人可以把一个问题分拆出好几个方向去诉说，永远不想解决的操作办法。女人以诉说问题为目的。

如果某个类经常因为不同的原因在不同的方向上发生变化，发散式变化（Divergent Change）就出现了。那么此时也许将这个对象分成 2 个会更好，这么一来每个对象就可以只因 1 种变化而需要修改。针对某以外界变化的所有相应修改，都只应该发生在单一类中，而这个新类内的所有内容都应该反应此变化。为此应该找出某特定原因而造成的所有变化，然后运用 Extract Class （提炼类）将它们提炼到另一个类中。

6. Shotgun Surgery 霰弹式修改

仅做一个简单的修改却要求改变多个类。即我们需要修改的代码散布四处:

Shotgun Surgery 霰弹式修改 （仅做一个简单的修改却要求改变多个类。即我们需要修改的代码散布四处，不但很难找到它们，也很容易忘掉某个重要的修改。 ）将多个类相分离是代码的一大职责。有可能缺少一个通晓全盘职责的类 (而大量修改本应针对这个类完成)。另外,也有可能因过度去除发散式改变而招致这个坏味道。

找出一个应对这些修改负责的类。这可能是一个现有的类,也可能需要通过应用抽取类来创建一个新的类。使用搬移字段(Move Field)和搬移方法(Move Method),将功能置于所选的类中。如果未选中类足够简单,则可以使用内联类(Inline Class)将该类去除。

7. Feature Envy 依恋情节

类的方法应该去该去的地方:

函数对某个类的兴趣高过对自己所处的类，通常的焦点就是数据，某个函数为了计算某个值，从另一个对象那儿调用几乎半打的取值函数。这时一个运用 Move Method （搬移函数）把它移到自己该去的地方。有时候函数中只有一部分受这种依恋之苦，这时候使用 Extract Method （提炼函数）把这部分提炼到独立函数中，再使用 Move Method （搬移函数）带它去它的梦中家园。

一个函数往往会用到几个类的功能，那么它究竟该被置于何处呢？我们的原则是：判断哪个类拥有最多被此函数使用的数据，然后就把这个函数和那些数据摆在一起。

8. Data Clumps 数据泥团

众多数据项待在一块:

常常可以在很多地方看到相同的 3、4 项数据：2 个类中相同的字段、许多函数签名中相同的参数。这些总是绑在一起出现的数据真应该拥有属于它们自己的对象。首先找出这些数据以字段形式出现的地方，运用 Extract Class （提炼类）将它们提炼到一个独立对象中。然后将注意力转移到函数签名上，运用 Introduce Parameter Object （引入参数对象）或 Preserve Whole Object （保持对象完整）为它减肥。这么做的直接好处是可以将很多参数列缩短，简化函数调用。不必在意数据泥团（Data Clumps）只用上新对象的一部分字段，只要以新对象取代 2（或更多）个字段，就值得了。

一个好的评判方法是：删除众多数据中的一项。这么做，其他数据有没有因而失去意义？如果它们不再有意义，这就是个明确信号：你应该为它们产生一个新对象。

9. Primitive Obsession 基本类型偏执

喜欢使用基本类型,而不愿运用小对象:

对象的一个极具价值的东西是：它们模糊（甚至打破）了横亘于基本数据和体积较大的类之间的界限。你可以轻松的编写出一些与语言内置（基本）类型无异的小型类。
对象技术的新手通常不愿意在小任务上运用小对象—像是结合数值和币种的 money 类，由一个起始值和结束值组成的 range 类等。你可以使用 Replace Data Value withObject （以对象取代数据值）将原本单独存在的数据值替换为对象。如果想要替换的数据值是类型码，而它并不影响行为，则可以运用 Replace Type Code with Class （以类取代类型码）将它替换掉。如果你有与类型码相关的条件表达式，可运用 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）加以处理。

如果你有一组应该总是被放在一起的字段，可运用 Extract Class （提炼类）。如果你在参数列中看到基本数据类型，不妨试试 Introduce Parameter Object （引入参数对象）。如果你发现自己正从数组中挑选数据，可运用 Replace Array with Object（以对象取代数组）.

10. Switch Statement （switch 惊悚现身）

switch 语句的问题在于重复:

面向对象程序的一个最明显特征就是：少用 switch 或（case）语句。从本质上说，switch 语句的问题在于重复。你常会发现 switch 语句散布于不同地点。如果要为它添加一个新的 case 子句，就必须找到所有 switch 语句并修改它们。面向对象中的多态概念可为此带来优雅的解决办法。

大多数时候，一看到 switch 语句，就应该考虑以多态来替换它。问题是多态该出现在哪？switch 语句常常根据类型码进行选择，你要的是“与该类型码相关的函数或类”，所以应该使用 Extract Method （提炼函数）将 switch 语句提炼到一个独立函数中，再以 Move Method （搬移函数）将它搬移到需要多态性的那个类里。此时你必须决定是否使用 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）。一旦完成这样继承结构后，就可以运用 Replace Conditional with Polymorphism （以多态取代条件表达式）了。

如果你只是在单一函数中有些选择事例，且并不想改动它们，那么多态就不必要了。这时可运用 Replace Parameter with Explicit Methods （以明确函数取代参数）。如果你的选择条件之一是 null，可以试试 Introduce Null Object （引入 Null 对象）.

11. Parallel Inheritance Hierarchies 平衡继承体系

平行继承体系其实是散弹式修改（Shotgun Surgery）的特殊情况:

在这种情况下，每当你为某个类增加 1 个子类，必须也为另一个类相应增加 1 个子类。如果你发现某个继承体系的类名前缀和另一个继承体系的类名前缀完全相同，便是闻到了这种坏味道。

消除这种重复性的一般策略是：让一个继承体系的实例引用另一个继承体系的实例。如果再接再厉运用 Move Method （搬移函数）和 Move Field （搬移字段）,，就可以将引用端的继承体系取消。

12. Lazy Class（冗赘类）

如果一个类的所得不值其身价，它就应该消失:

即如果某些子类没有足够的工作，试试 Collapse Hierarch （折叠继承体系）。对应几乎没用的组件，应该以 Inline Class （将类内联化）对付它们。

13. Speculative Generality（夸夸其谈未来性）

如果用不到的类就不要用:

如果你的某个抽象类其实没有太大作用，请运用 Collapse Hierarch （折叠继承体系）。不必要的委托可运用 Inline Class （将类内联化）除掉。如果函数的某些参数未被用上，可对它实施 Remove Parameter （移除参数）。如果函数名称带有多余的抽象意味，应该对它实施 Rename Method （函数改名）

如果函数或类的唯一用户是测试用例，这就飘出了坏味道 夸夸其谈未来性（Speculative

Generality)。 如果有这样的函数或类，请把它们连同其测试用例一并删除。但如果它们的用途是帮助测试用例检测正当功能，则不能删除。

14. Temporary Field（令人迷惑的暂时值域）

对象的暂时性属性经常让人迷惑：

有时你会看到这样的对象：其内某个实例变量仅为某种特定情况而设。这样的代码让人不易理解，因为你通常认为对象在所有时候都需要它的所有变量。在变量未被使用的情况下猜测当初设置目的，会让你发疯。

请使用 Extract Class （提炼类）给这些变量创造一个家，然后把所有和这些变量相关的代码都放进这个新家，也许你还可以使用 Introduce Null Object （引入 Null 对象）在变量不合法的情况下创建一个 null 对象，从而避免写出条件式代码。

如果类中有一个复杂算法，需要好几个变量，往往就可能导致坏味道令人迷惑的临时字段（Temporary Field）出现。由于实现者不希望传递一长串参数，所以他把这些参数都放进字段。但是这些字段只在使用该算法时才有效，其他情况下只会让人迷惑。这时可以利用 Extract Class （提炼类）把这些变量和其相关函数提炼到一个独立的类中。提炼后的新对象将是一个函数对象。

15. Message Chains（过度耦合的消息链）

A 对象请求 B 对象，B 对象请求 C 对象...:

如果你看到用户向一个对象请求另一个对象，然后再向后者请求另一个对象，然后再请求另一个对象……这就是消息链。实际代码中你看到的可能是一长串 getThis（）或一长串临时变量。采取这种方式，意味客户代码将与查找过程中的导航紧密耦合。一旦对象间关系发生任何变化，客户端就不得不做出相应的修改。

这时候应该使用 Hide Delegate （隐藏委托关系）。你可以在消息链的不同位置进行这种重构。理论上可以重构消息链上任何对象，但这么做往往会把一系列对象都变成 Middle Man（中间人）。通常更好的选择是：先观察消息链最终得到的对象是用来干什么的，看看能否以 Extract Method （提炼函数）把使用该对象的代码提炼到一个独立函数中，再运用 Move Method （搬移函数）把这个函数推入消息链。

16. Middle Man（中间转手人）

过度运用委托:

对象的基本特征之一就是封装：对外部世界隐藏其内部细节。封装往往伴随委托。比如说你问你主管是否有时间参加一个会议，他就把这个消息“委托”给他的记事簿，然后才

能回答你。你没必要知道这位主管到底使用传统记事簿或电子记事簿或秘书来记录自己的约会。

但是人们可能过度运用委托。你也许会看到某个类有一半的函数都委托给其他类，这样就是过度运用。这时应该使用 Remove Middle Man （移除中间人），直接和真正负责的对象打交道。如果不干实事的函数只有少数几个，可以运用 Inline Method （内联函数）把它们放进调用端。如果这些中间人还有其他行为，可以运用 Replace Delegation with Inheritance （以继承取代委托）把它们变成实责对象的子类，这样你即可以扩展原对象的行为，又不必负担那么多的委托动作。

17. Inappropriate Intimacy（狎昵关系）

2 个类过于亲密:

有时候你会看到 2 个类过于亲密，花费太多时间起探究彼此的 private 成分。你可以采用 Move Method （搬移函数）和 Move Field （搬移字段）帮他们划清界限。你也可以看看是否可以运用 Change Bidirectional Association to Unidirectional （将双向关联改为单向关联）让其中一个类对另一个斩断情丝。如果 2 个类实在是情投意合，可以运用 Extract Class （提炼类）把 2 者共同点提炼到一个安全地点，让它们坦荡的使用这个新类。或者可以尝试运用 Hide Delegate （隐藏委托关系）让另一个类来为它们传递相思情。

继承往往造成过度亲密，因为子类对超类的了解总是超过后者的主观愿望，如果你觉得该让这个孩子独自生活了，请运用 Replace Inheritance with Delegation （以委托取代继承）让它离开继承关系。

18. Alternative Classes with Different Interfaces（异曲同工的类）

两个函数做同一件事：

如果两个函数做同一件事，却有着不同的签名，请运用 Rename Method （函数改名）根据它们的用途重新命名。但这往往不够，请反复运用 Move Method （搬移函数）将某些行为移入类，直到 2 者的协议一致为止。如果你必须反复而赘余的移入代码才能完成这些，或许可运用 Extract Superclass （提炼超类）。

19. Incomplete Library Class（不完善的库类）

复用常被视为对象的终极目的。不过复用的意义常被高估：大多数对象只要够用就好。但是无可否认，许多编程技术都建立在程序库的基础上。库类的构筑者没用未卜先知的能力，不能因此责怪他们。麻烦的是库往往构造的不够好，而且往往不可能让我们修改其中的类使它们完成我们希望完成的工作。这是否意味着那些经过实践检验的技术，如今都派不上用场了？

幸好我们有 2 个专门应付这种情况的工具。如果你只想修改类库的一两个函数，可以运用 Introduce Foreign Method （引入外加函数）；如果想要添加一大堆额外行为，就得运用 Introduce Local Extension （引入本地扩展）。

20. Data Class（纯稚的数据类）

所谓的 Data Class 是指：它们拥有一些字段，以及用于访问这些字段的函数，除此之外一无长物。这样的类只是不会说话的数据容器，它们几乎一定被其他类过分细琐的操控着。这些类早期可能拥有 public 字段，果真如此你应该在别人注意到它们之前，立刻运用 Encapsulated Field （封装字段）将它们封装起来。如果这些类含容器类的字段，你应该检查它们是不是得到了恰当的封装；如果没有，就运用 Encapsulated Collection （封装集合）把它们封装起来。对于那些不该被其他类修改的字段，请运用 Remove Setting Method （移除设置函数）。

然后，找出这些取值/设值函数被其他类运用的地点。尝试以 Move Method （搬移函数）把那些调用行为搬移到 Data Class 来。如果无法搬移这个函数，就运用 Extract Method （提炼函数）产生一个可搬移的函数。不久之后就可以运用 Hide Method （隐藏函数）把这些取值/设值函数隐藏起来了。

21 Refused Bequest（被拒绝的遗赠）

子类应该继承超类的函数和数据。但如果它们不想或不需要继承，又该这么办呢？它们得到所有礼物，却只从中挑选几样来玩。

按传统说法，这就意味着继承体系设计误。你需要为这个子类新建一个兄弟类，再次运用 push down Method （函数下移）和 push down field （字段下移）把所有用不到的函数下推给那个兄弟。这样一来，超类就只持有所有子类共享的东西。你常常会听到这样的建议：所有超类都应该是抽象的。既然使用“传统说法”这个略带贬义的词，你就可以猜到，我们不建议你这么做，起码不建议你每次都这么做。我们继承利用继承来复用一些行为，并发现可以很好的应用于日常工作。这也是一种坏味道，我们不否认，但气味通常并不强烈。所以我们说：如果 Refused Bequest 引起困惑和问题，请遵循传统忠告。

但不必认为你每次都得那么做。十有八九这种坏味道很淡，不值得理睬。如果子类复用了超类的行为，却有不愿支持超类的接口，Refused Bequest 的坏味道就会变得浓烈。拒绝继承超类的实现， 这点我们不介意；但如果拒绝继承超类的接口，我们不以为然。不过即使你不愿意继承接口，也不要胡乱修改继承体系，应用运用 Replace Inheritance with Delegation （以委托取代继承）来达到目的。

22 Comments（过多的注释）

我们之所以在这里提到注释，是因为人们常把它当做除臭剂来使用。常常会有这样的情况：你看到一段代码有着长长的注释，然后发现，这些注释之所以存在是因为代码很糟糕。注释可以带我们找到代码中的坏味道。找到坏味道后，我们首先应该以各种重构手法把坏味道去除。完成之后我们常常会发现：注释已经变得多余了，因为代码已经清楚说明了一切。

如果你需要注释来解释一块代码做了说明，试试 Extract Method （提炼函数）；如果函数已经提炼出来，但还是需要注释来解释其行为，试试 Rename Method （函数改名）；如果你需要注释说明某些系统的需求规格，试试 Introduce Assertion （引入断言）。