目录

[一：代码块味道 2](#_Toc386030538)

[1.1：重复代码 2](#_Toc386030539)

[1.2：方法过长 3](#_Toc386030540)

[1.3：条件逻辑太复杂 3](#_Toc386030541)

[1.4：基本类型迷恋 4](#_Toc386030542)

[1.5：不恰当的暴露 5](#_Toc386030543)

[1.6：解决方案蔓延 5](#_Toc386030544)

[1.7：异曲同工的类 5](#_Toc386030545)

[1.8：冗赘类 6](#_Toc386030546)

[1.9：类过大 6](#_Toc386030547)

[2.0：分支语句 7](#_Toc386030548)

[2.1：组合爆炸 7](#_Toc386030549)

[2.2：怪异解决方案 8](#_Toc386030550)

[二：重构 8](#_Toc386030551)

[2.1：用Creation Method替换构造函数(6.1)，链构造函数(11.1) 8](#_Toc386030552)

[2.2：用Factory封装类(6.3) 8](#_Toc386030553)

[2.3：用Factory Method引入多态创建(6.4) 8](#_Toc386030554)

[2.4：用Strategy替换条件逻辑(7.2) 8](#_Toc386030555)

[2.5：形成Template Method(8.1) 8](#_Toc386030556)

[2.6：组合方法(7.1) 8](#_Toc386030557)

[2.7：用Composite替换隐含树(7.5) 8](#_Toc386030558)

[2.8：用Builder封装Composite(6.5) 8](#_Toc386030559)

[2.9：将聚集操作搬移到Collecting Parameter(10.1) 8](#_Toc386030560)

[2.10：提取Composite(8.2)，用Composite替换一/多之分(8.3) 8](#_Toc386030561)

[2.11：用Command替换条件调度程序(7.6) 8](#_Toc386030562)

[2.12：提取Adapter(8.6),通过Adapter统一接口(8.5) 8](#_Toc386030563)

[2.13：用类替换类型代码(9.1) 8](#_Toc386030564)

[2.14：用State替换状态改变条件语句(7.4) 8](#_Toc386030565)

[2.15：引入Null Object(9.3) 8](#_Toc386030566)

[2.16：内联Singleton(6.6)，用Singleton限制实例化(9.2) 9](#_Toc386030567)

[2.17：用Observer替换硬编码的通知(8.4) 9](#_Toc386030568)

[2.18：将装饰功能搬移到Decorator(7.3),统一接口(11.2)，提取参数(11.3) 9](#_Toc386030569)

[2.19：将创建知识搬移到Factory(6.2) 9](#_Toc386030570)

[2.20：将聚集操作搬移到Visitor(10.2) 9](#_Toc386030571)

[2.21：用Interpreter替换隐式语言(8.7) 9](#_Toc386030572)

# 一：代码块味道

1.1：重复代码

形成Template Method（8.1 节）

用Factory Method 引入多态创建（6.4 节）

链构造函数（11.1 节）

用Composite 替换一/多之分（8.3 节）

提取Composite（8.2 节）

通过Adapter 统一接口（8.5 节）

引入Null Object（9.3 节）

小结：我们经常可以通过应用形成Template Method（8.1 节）重构，去除类层次

中不同子类里存在的明显和/或微妙的重复。如果不同子类中的方法除了对象创建不足之外其他实现方式都类似，可以应用用Factory Method 引入多态创建（6.4节）重构，为使用Template Method 去除更多重复做好准备。

如果类的构造方法包含重复代码，通常可以通过应用链构造函数（11.1 节）重构去除这种重复。

如果有单独的代码处理一个对象或者一组对象，也许可以通过应用用Composite 替换一/多之分（8.3 节）重构去除重复。

如果类层次的多个子类都实现了自己的Composite，而且实现可能完全相同，这时可以使用提取Composite（8.2 节）重构。

如果对象处理方式的区别仅在于它们的接口不同，应用通过Adapter 统一接口（8.5 节）重构能够为去除重复的处理逻辑做好准备。

如果有条件逻辑用于处理空对象，而且相同的空逻辑在整个系统中都是重复的，那么可以应用引入Null Object（9.3 节），消除重复并简化系统。

1.2：方法过长

组合方法（7.1 节）

将聚集操作搬移到Collecting Parameter（10.1节）

用Command 替换条件调度程序（7.6 节）

将聚集操作搬移到Visitor（10.2 节）

用Strategy 替换条件逻辑（7.2 节）

小结：Fowler 和Beck [F]在对这种坏味的描述中，解释了方法短胜于长的几个充分理由。主要的理由是逻辑的共享。两个很长的方法很可能包含重复代码。如果将这些方法分解为多个小方法，就能发现通常有很多方式能够使它们共享逻辑。

我只要遇到长方法，马上就有应用组合方法（7.1 节）重构将它分解为一个Composed Method [Beck, SBPP] 的冲动。这项工作通常包含提炼方法[F]的应用。如果正在转换为Composed Method 的代码要将某个信息累加到一个公共变量中，可以考虑应用将聚集操作搬移到Collecting Parameter（10.1 节）重构。

如果方法很长，是因为它包含了一个用来分派和处理请求的大switch 语句，可以使用用Command 替换条件调度程序（7.6 节）重构将其缩短。

如果使用switch 语句从接口不同的许多类收集数据，可以通过应用将聚集操作搬移到Visitor（10.2 节）重构缩短方法的长度。

如果方法很长，是因为它包含算法的许多版本和运行时用来选择版本的条件逻辑，那么可以应用用Strategy 替换条件逻辑（7.2 节）重构缩短方法的长度。

1.3：条件逻辑太复杂

用Strategy 替换条件逻辑（7.2 节）

将装饰功能搬移到Decorator（7.3 节）

用State 替换状态改变条件语句（7.4 节）

引入Null Object（9.3 节）

小结：如果条件逻辑很容易理解，而且只包含几行代码，那么其本身是无辜的。不幸的是，它很少能够这样善始善终。例如，实现几个新功能后，条件逻辑就突然变得复杂而又开销高昂。《重构》[F] 一书中和本目录中的几个重构正是用来解决这一问题的。

如果条件逻辑控制的是应该执行一种计算操作几个变形中的某一个，则可以考虑应用用Strategy 替换条件逻辑（7.2 节）重构。

如果条件逻辑控制的是应该执行类的核心行为之外某个特殊行为的若干段中的某一段，则可以使用将装饰功能搬移到Decorator（7.3 节）重构。

如果控制对象状态转换的条件表达式比较复杂，可以考虑通过应用用State替换状态改变条件语句（7.4 节）重构简化逻辑。

处理空操作情形经常需要创建条件逻辑。如果在整个系统中有重复的相同的空条件逻辑，则可以使用引入Null Object（9.3 节）重构进行清理。

1.4：基本类型迷恋

用类替换类型代码（9.1 节）

用State 替换状态改变条件语句（7.4 节）

用Strategy 替换条件逻辑（7.2 节）

用Composite 替换隐含树（7.5 节）

用Interpreter 替换隐式语言（8.7 节）

将装饰功能搬移到Decorator（7.3 节）

用Builder 封装Composite（6.5 节）

小结：如果控制对象的状态转换的是使用基本类型值的复杂条件逻辑，那么可以使用用State 替换状态改变条件语句（7.4 节）重构。其效果是用许多类表示每个状态和简化的状态转换逻辑。

如果控制算法运行的是非常复杂的条件逻辑，而且该逻辑还使用基本类型值，可以考虑应用用Strategy 替换条件逻辑（7.2 节）重构。

如果隐式创建了使用基本类型表示的树结构，比如字符串，代码将非常难用、易错而且充满重复。应用用Composite 替换隐含树（7.5 节）重构将减少这些问题。

如果类有许多方法支持多个基本类型值的组合，那么就可能存在隐式语言。这种情况下，考虑应用用Interpreter 替换隐式语言（8.7 节）重构。

如果类中有基本类型值只是为提供类的核心职责的装饰，可以使用将装饰功能搬移到Decorator（7.3 节）重构。

最后，即使有类，但是它过于原始，对客户代码的用处不大。比如，很难使用的 Composite [DP]实现。可以通过应用用Builder 封装Composite（6.5 节）重构简化客户构建 Composite 的工作。

1.5：不恰当的暴露

用Factory 封装类

小结：用Factory 封装类（6.3 节）重构可以去除这种坏味。并不是所有对客户代码有用的类都需要设为公开的（即有公开构造函数）。有些类应该只通过公共接口引用。如果将类的构造函数设为非公开的，并使用Factory 对象生成实例，就可以做到这一点。

1.6：解决方案蔓延

将创建知识搬移到Factory

1.7：异曲同工的类

通过Adapter 统一接口

小结：这个Fowler 和 Beck [F]书中讲到的编程坏味，是指两个相似类却有不同接口的时候。如果发现两个类很相似，通常可以将它们重构为共享一个公共的接口。

但是，有些时候不能直接修改类的接口，因为对代码没有控制权限。比较典型的例子是使用第三方库的时候。这种情况下，可以应用通过Adapter 统一接口（8.5 节）重构为两个类生成一个公共接口。

1.8：冗赘类

内联Singleton

小结：在描述这种坏味时，Fowler 和 Beck 这样写道：“类如果功能有限，缺乏存在价值，就应该删除。”[F, 83]。经常能够遇到缺乏存在价值的Singleton[DP]。事实上，Singleton 可能使你的设计过分依赖全局数据，致使代价过高。内联Singleton（6.6 节）重构给出了删除Singleton 的一种快速而优雅的过程。

1.9：类过大

用Command 替换条件调度程序（7.6 节）

用State 替换状态改变条件语句（7.4 节）

用Interpreter 替换隐式语言（8.7 节）

小结：Fowler 和 Beck [F] 提到，存在太多的实例变量，往往说明类的职责太多。一般而言，过大的类通常包含过多的职责。提炼类[F]和提炼子类[F]重构是用来处理这种坏味的主要重构，有助于将职责搬移到其他类中。本书中模式导向的重构都将使用这种重构为类减肥。

用Command 替换条件调度程序（7.6 节）重构将行为提取到Command [DP]类中，这样能够极大地减小响应不同请求而执行各种行为的类的体积。

用State 替换状态改变条件语句（7.4 节）重构能够将充满状态转换代码的大类缩减为小类，将职责委托给一族State [DP] 类。

用Interpreter 替换隐式语言（8.7 节）重构能够通过将大量模仿某种语言的代码转换为小的Interpreter [DP]，从而将类大而化小。

2.0：分支语句

用Command 替换条件调度程序（7.6 节）

将聚集操作搬移到Visitor（10.2 节）

小结：switch 语句（或者if...elseif...elseif...结构的等效语句）本身并没有问题。只有在使用这种语句会使设计过度地复杂或者僵硬时，它们才会成为问题。这时，最好将分支语句去除，重构为更基于对象或者多态的解决方案。

用Command 替换条件调度程序（7.6 节）重构描述了如何将大的分支语句分解为一组Command [DP] 对象，每个Command 对象都可以不使用条件逻辑进行查找和调用。

将聚集操作搬移到Visitor（10.2 节）重构描述了这样一个例子，其中用分支语句保存来自具有不同接口的实例的数据。通过将这段代码重构为使用Visitor [DP]，就不需要使用条件逻辑，而且设计也变得更加灵活。

2.1：组合爆炸

用Interpreter 替换隐式语言

小结：这种坏味其实是一种不那么明显的重复。当有许多段代码使用不同种类或数量的数据或对象做同样的事情时，就会出现这种坏味。

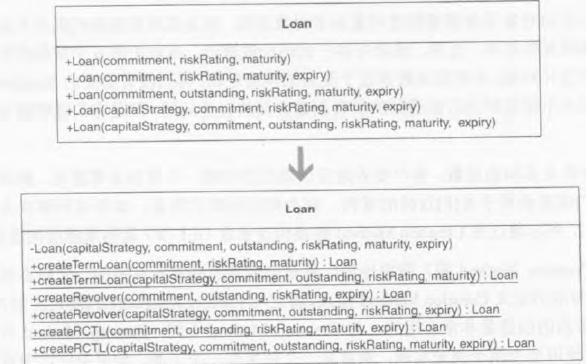
例如，假设你的类中有许多方法执行查询。每个方法都使用特定的条件和数据执行查询。需要支持的特殊查询越多，必须创建的查询方法也就越多。很快，用来处理各种查询方式的方法就会大爆炸。此外，你还使用隐式的查询语言。应用用Interpreter 替换隐式语言（8.7 节）重构，可以删除所有这些方法，同时去除组合爆炸坏味。

2.2：怪异解决方案

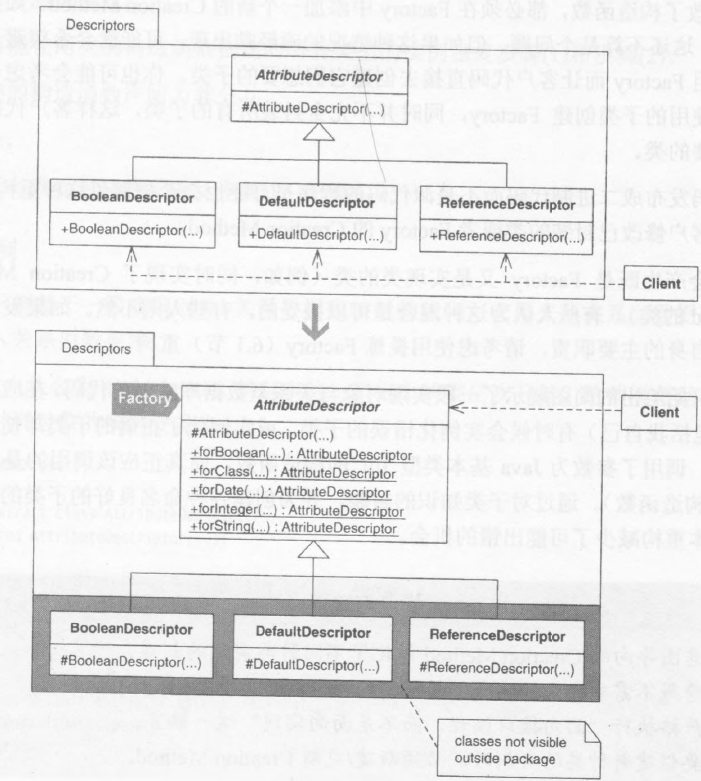
通过Adapter 统一接口

# 二：重构

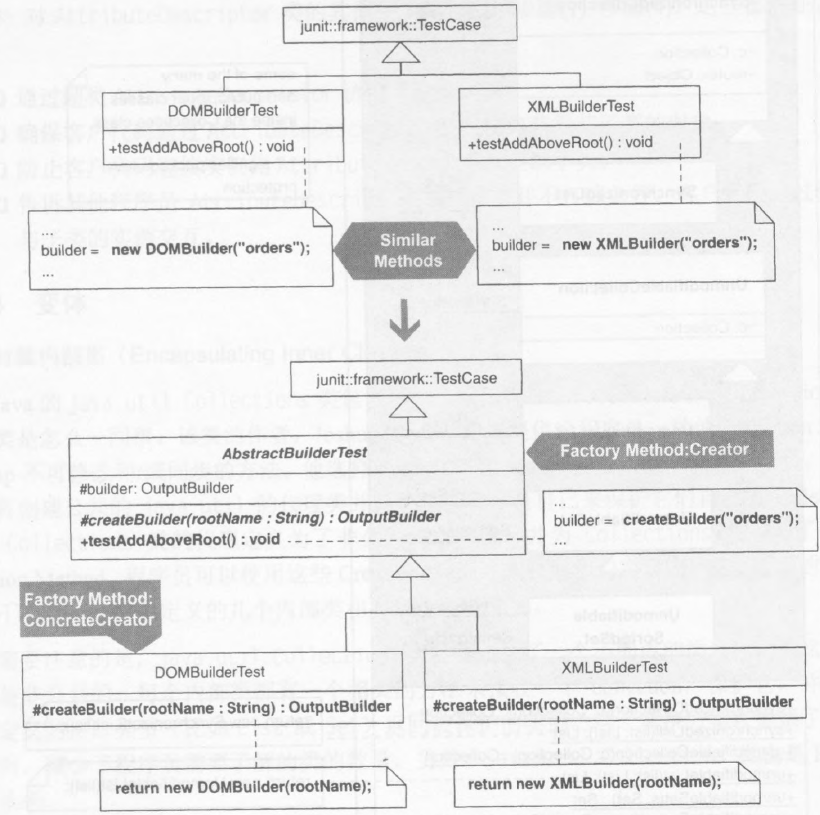
2.1：用Creation Method替换构造函数(6.1)，链构造函数(11.1)



2.2：用Factory封装类(6.3)

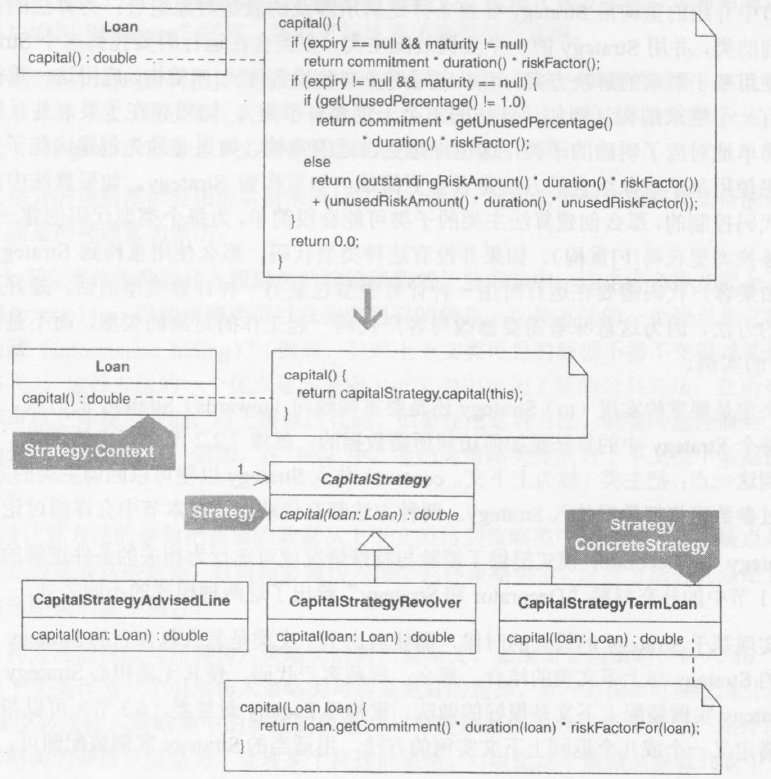


2.3：用Factory Method引入多态创建(6.4)



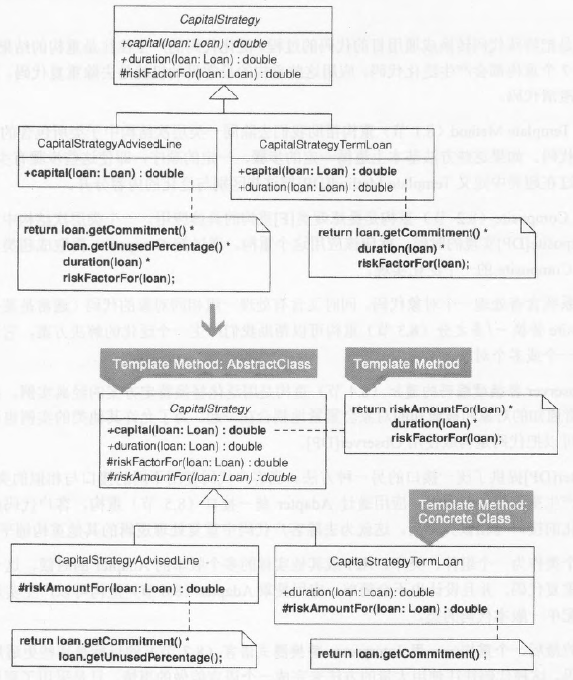
Factory Method是子类必须要实现的方法

2.4：用Strategy替换条件逻辑(7.2)



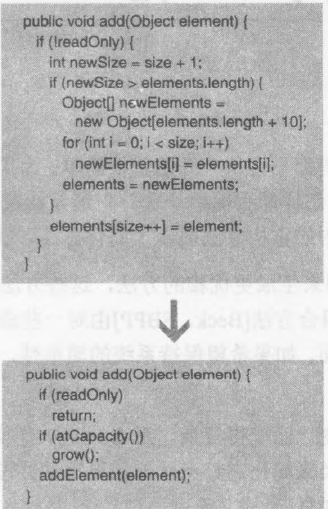
2.5：形成Template Method(8.1)

一次性实现了一个算法的不变部分，并将可变的的行为留给子类来实现

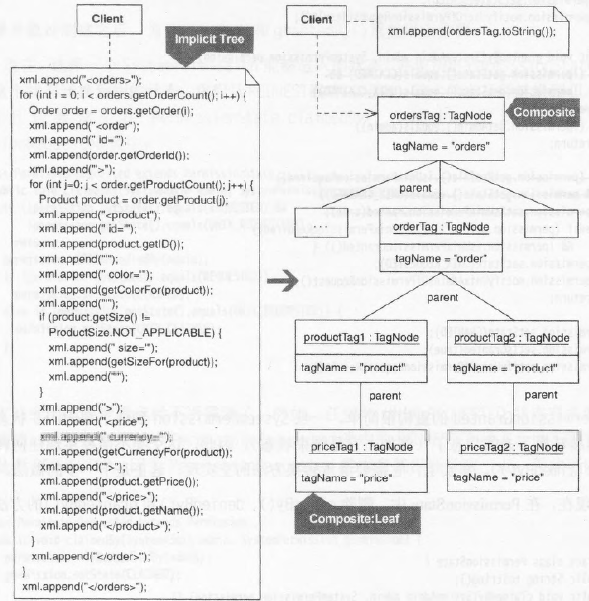


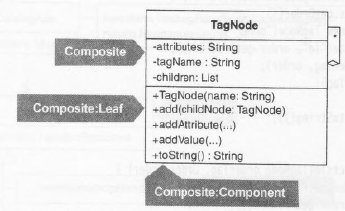
2.6：组合方法(7.1)

把方法的逻辑转换成几个同一细节层面上的，能够说明意图的步骤



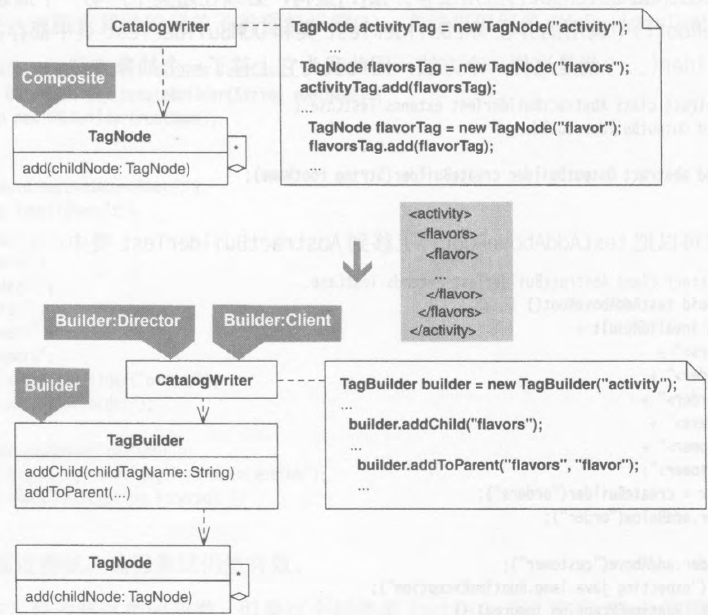
2.7：用Composite替换隐含树(7.5)



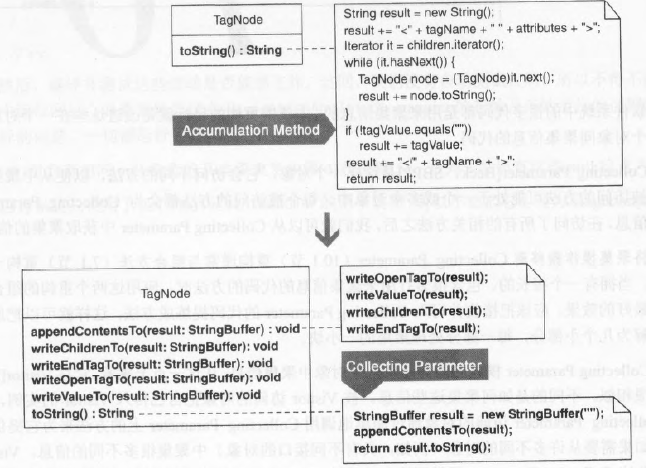


2.8：用Builder封装Composite(6.5)

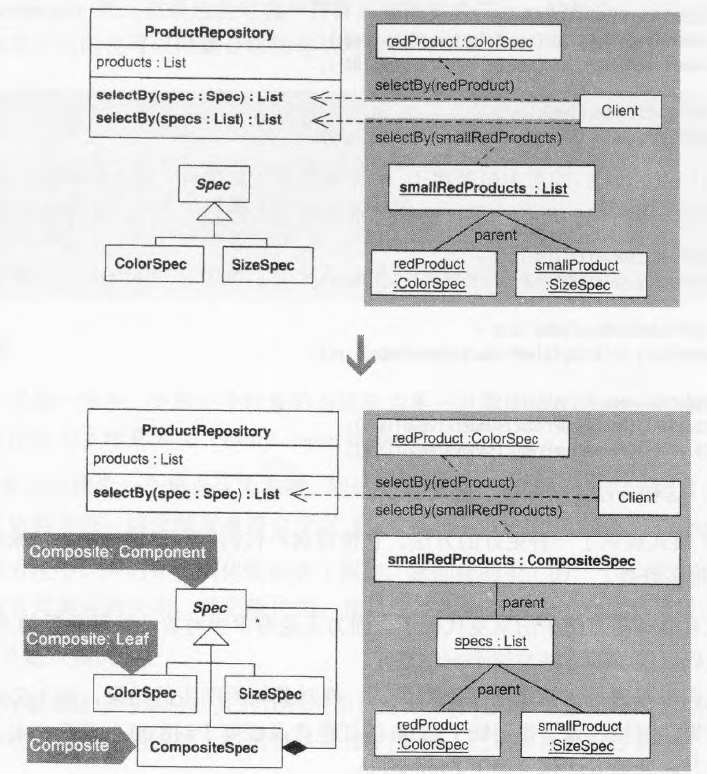
设计模式对Builder的描述：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示



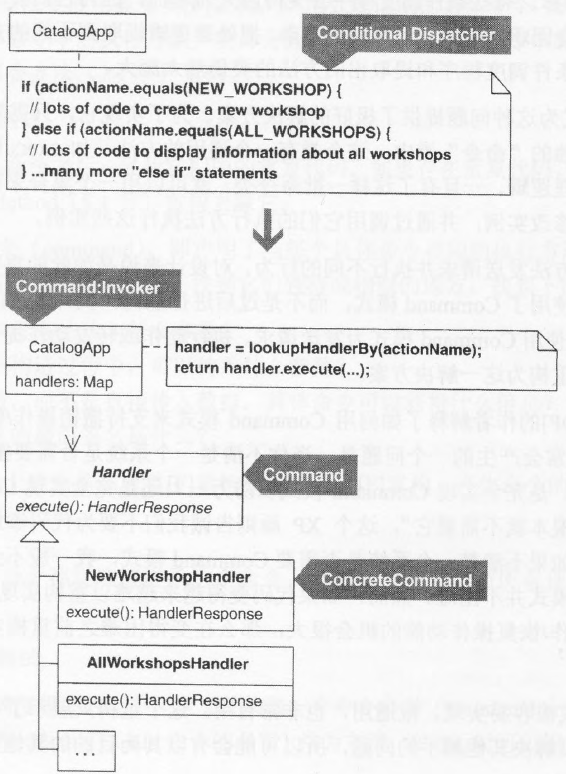
2.9：将聚集操作搬移到Collecting Parameter(10.1)



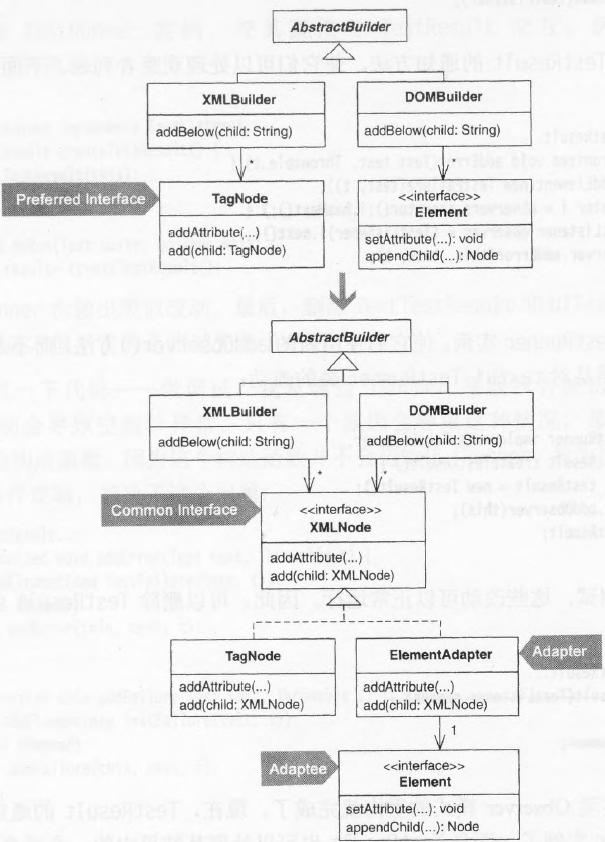
2.10：提取Composite(8.2)，用Composite替换一/多之分(8.3)



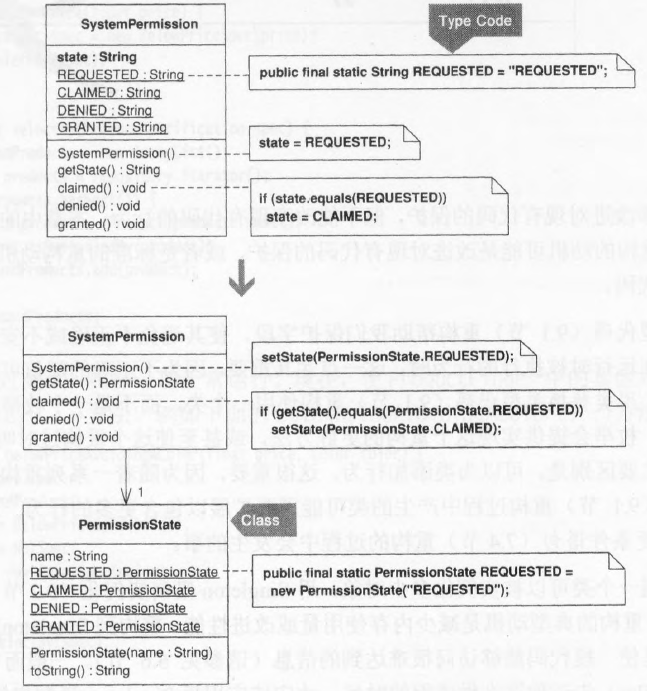
2.11：用Command替换条件调度程序(7.6)



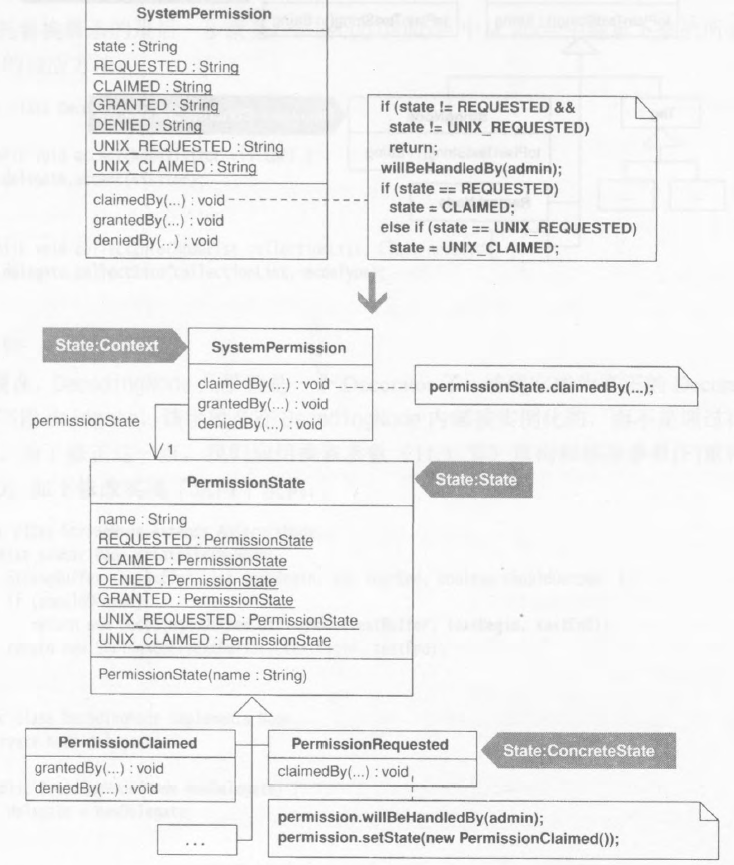
2.12：提取Adapter(8.6),通过Adapter统一接口(8.5)



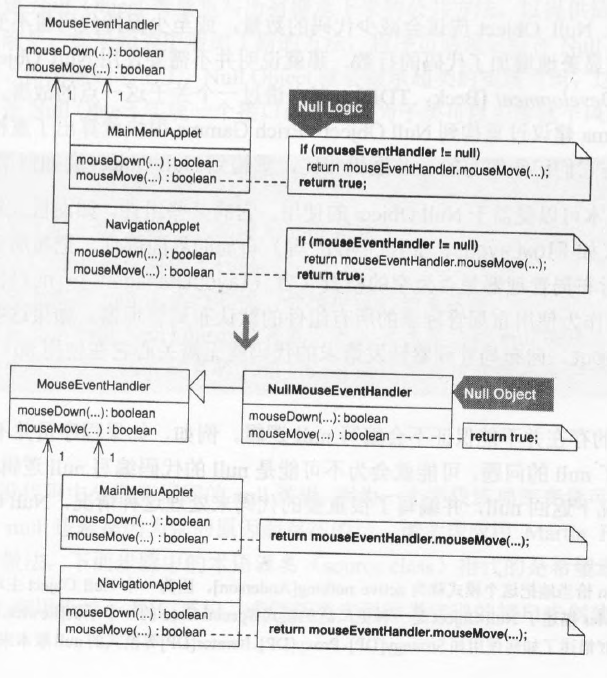
2.13：用类替换类型代码(9.1)



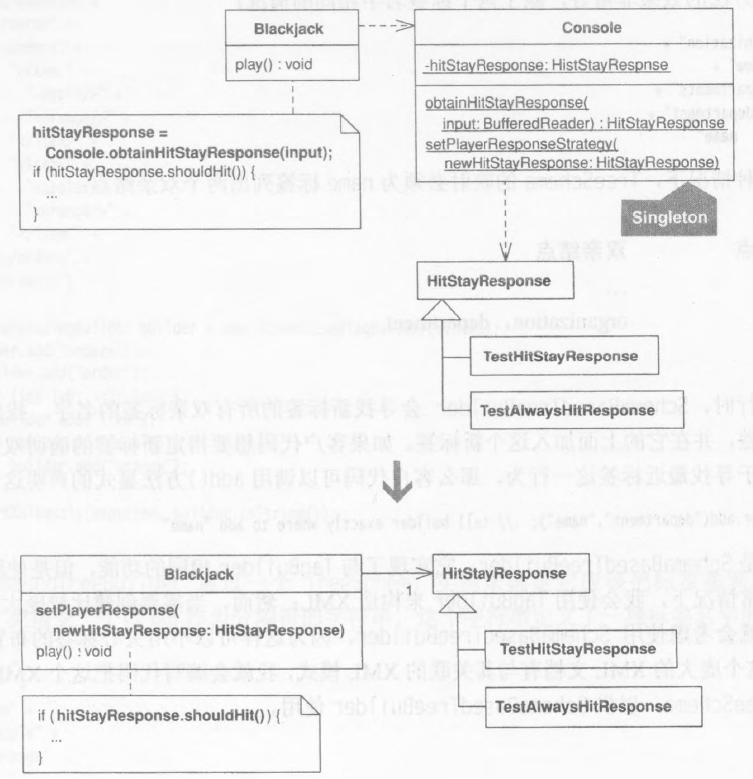
2.14：用State替换状态改变条件语句(7.4)

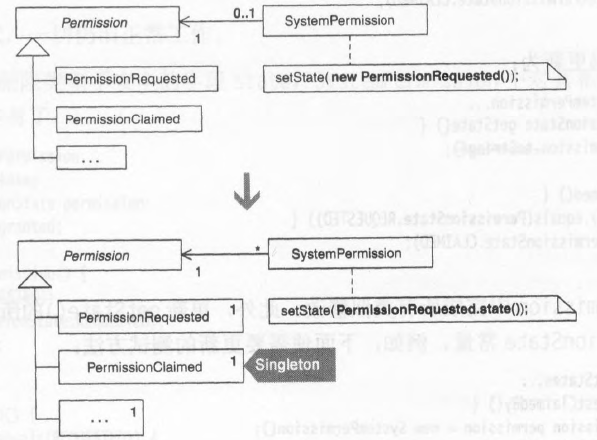


2.15：引入Null Object(9.3)

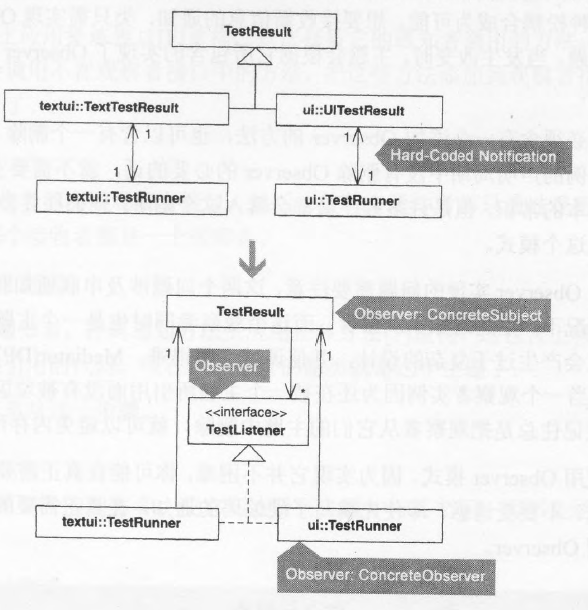


2.16：内联Singleton(6.6)，用Singleton限制实例化(9.2)

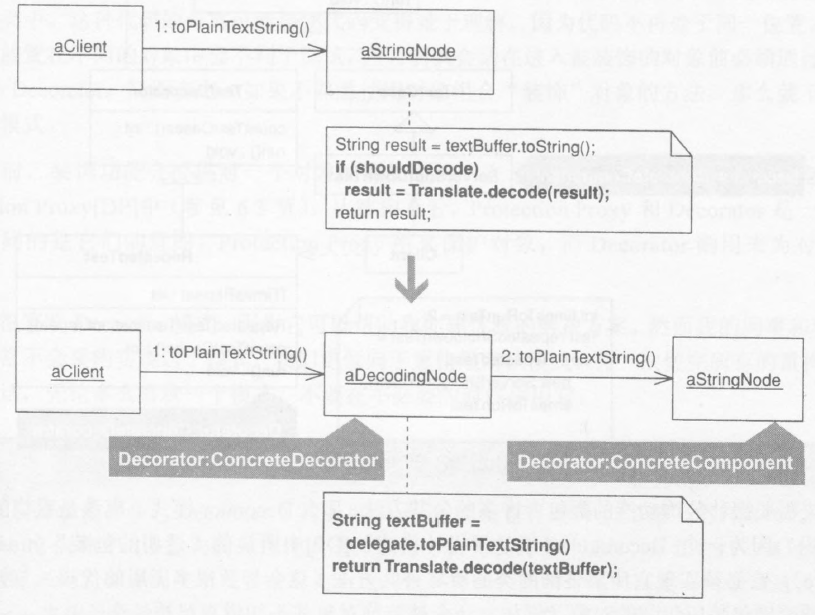


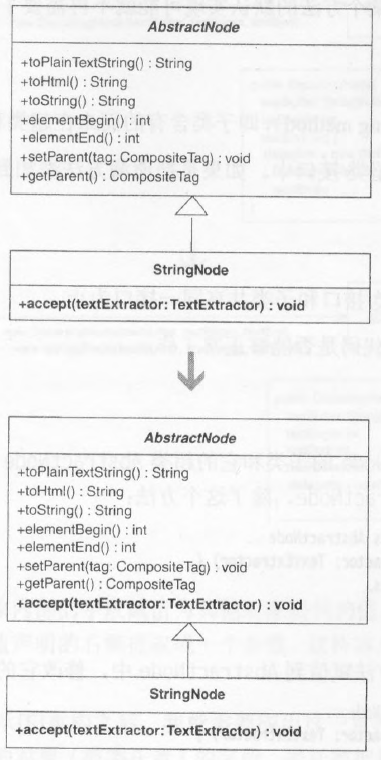


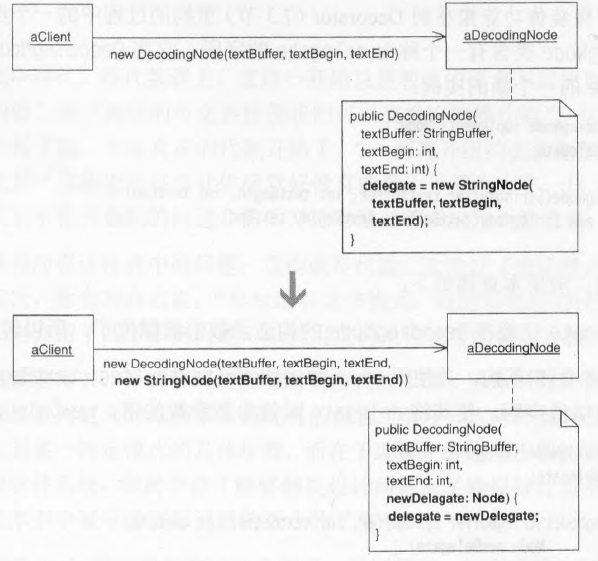
2.17：用Observer替换硬编码的通知(8.4)



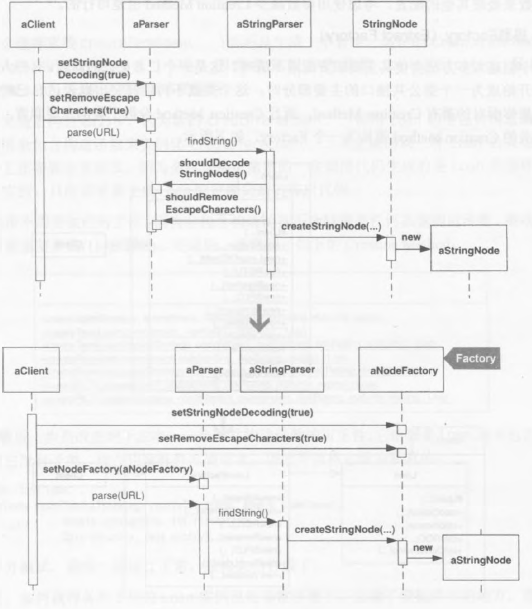
2.18：将装饰功能搬移到Decorator(7.3),统一接口(11.2)，提取参数(11.3)



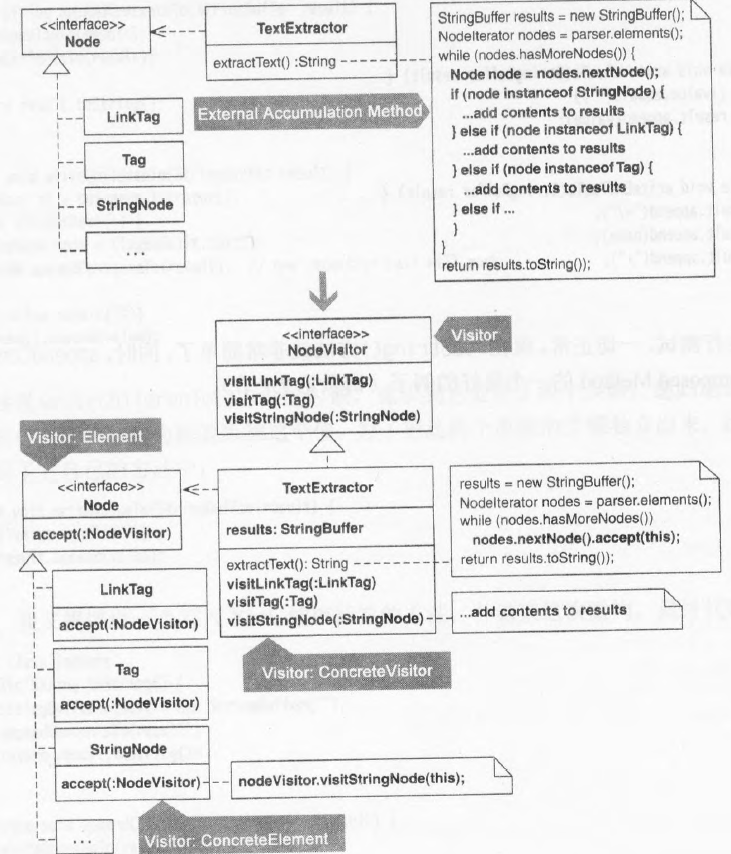




2.19：将创建知识搬移到Factory(6.2)



2.20：将聚集操作搬移到Visitor(10.2)



2.21：用Interpreter替换隐式语言(8.7)

