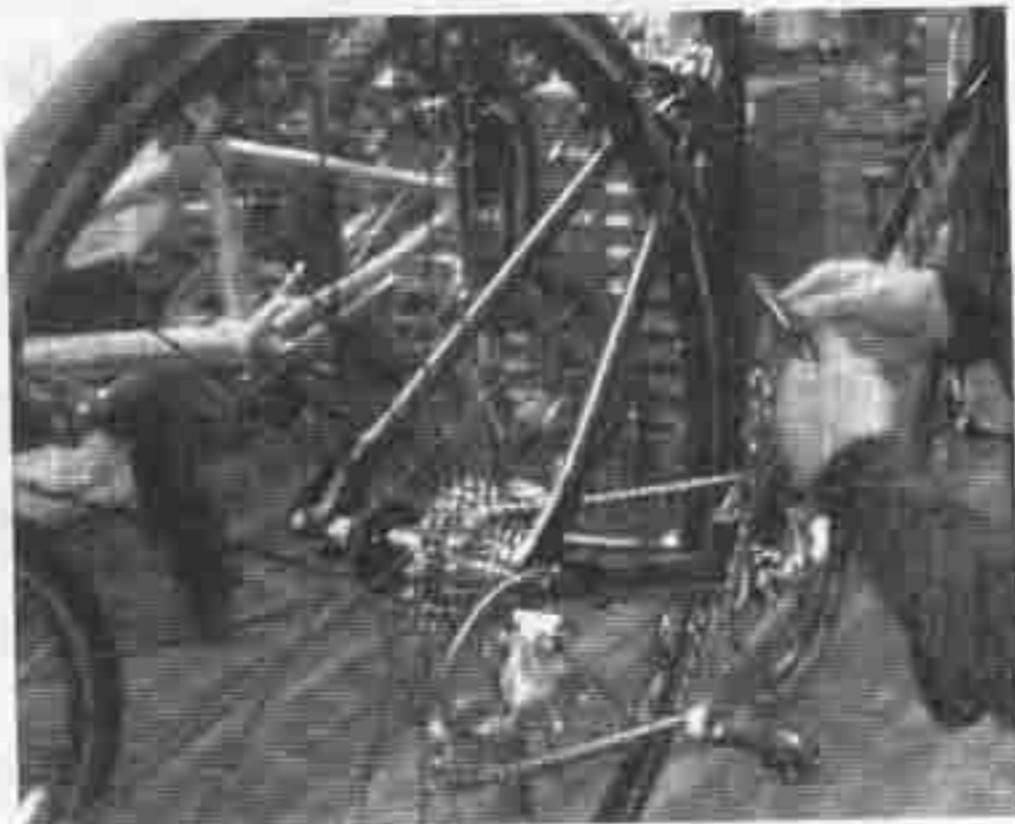


第10章

柔性设计



软件的最终目的是为用户提供服务，但是，它首先必须为开发人员服务。在强调重构的过程中，这一点尤为突出。随着程序的不断完善，每个部分都可能会被重新组织和编写。新创建的领域对象将与原有的领域对象及应用集成起来。甚至在多年以后，系统维护人员都还可能需要对代码进行修改和扩充。但是，他们是否会愿意做这些工作呢？

如果软件的行为非常复杂，但是又缺乏良好的设计，那它就会变得难以重构和组合。由于开发人员不能确切地预知计算的全部内涵，重复问题就产生了。如果设计元素都是整块的、无法重新组合的部分，重复就是不可避免的了。我们可以对那些类和方法进行分解使之便于重用，但这样一来各个小部分分别有什么作用又变得难以跟踪了。一个缺乏清晰设计的软件，单是看到其中的混乱结构就会让开发人员头脑发晕，更不用说对设计进行修改——那只会使设计更加混乱，有时甚至会由于意料之外的依赖关系而导致设计完全无法工作。这种脆弱性导致我们无法构造出行为更加丰富的系统(除非系统的规模相当小)，也阻碍了重构和迭代精化的进行。

随着项目开发的进行，我们应该感到速度越来越快，而不是感到包袱越来越沉。这



就要求我们的设计能让人乐于使用，也能很容易地适应改变。这就是柔性设计。

柔性设计是深层建模的补充。在把隐含概念挖掘出来并将其变成显式之后，我们的原料就算是准备好了。接着，我们通过一个迭代周期把原料锤炼成有用的形状，使模型能够简单明了地抓住问题的关键，同时使我们的设计能够让客户的开发人员真正将模型投入使用。设计和代码的开发又会引导我们对模型概念作进一步精化，使我们进入到下一个迭代周期，将模型重构到更深层的理解。这种迭代会重复很多次。但是，我们最终希望达到什么样的设计呢？在这个过程中我们应该进行哪些尝试？这就是本章将要讨论的内容。

许多过度工程(overengineering)也打着“灵活性”的旗号。但是，多余的抽象和间接层次往往回弄巧成拙。去看看那些真正为人们提供了灵活性的软件设计，我们就会发现其中某些简单的东西。简单并不是唾手可得的。为了使我们创建的元素能组装进一个精巧的系统之中，并且在组装之后仍然能被我们所理解，就必须遵循模型驱动设计的方法，并保持适度严谨的设计风格。要创造或使用这种方法很可能还需要相对精湛的设计技巧。

开发人员充当着两种角色，我们的设计必须为这两种角色服务。这两种角色完全可以由同一个人来充当——甚至在一分钟之内来回切换(但是不同角色和代码的关系是不同的)。一种角色是客户的开发人员，他负责按照设计将领域对象编织到应用或其他领域的代码中。柔性设计能够清晰地展现深层模型，将它的潜力完全发挥出来。客户开发人员可以灵活地使用一个最小的概念集合(这些概念之间都是松散关联的)，来描述领域中的各种场景。设计元素之间的配合非常自然，结果是可预测的，而且可以清晰地刻画出来，同时也非常健壮。

另外一点也同样重要，那就是设计还必须为对它进行修改的开发人员服务。为了对修改开放，设计必须易于理解，把客户开发人员使用的同一个底层模型展现出来。设计必须符合领域深层模型定下的轮廓，这样它才能在灵活点适应改变。代码的作用必须是透明的，这样修改的后果才容易预测。

设计的早期版本通常都是僵硬的。在项目工期和预算的压力下，许多设计甚至从未获得过任何柔性。我也从未见过哪个大程序从开始到结束都一直具有柔性。但是，当复杂性变成了进度的桎梏的时候，我们就必须对那些关键的、复杂的部位进行精心打磨，使之成为柔性设计，这样才能冲破复杂性的限制；否则，项目必将掉进维护旧代码的泥潭。

像这样的软件设计并没有公式可套，但是我还是精选了一系列模式，如图 10-1 所示。以我的经验，适当地使用这些模式能够为我们的设计带来柔性。这些模式和示例可以使我们体验到柔性设计的特点，以及获得柔性设计的思考方式。

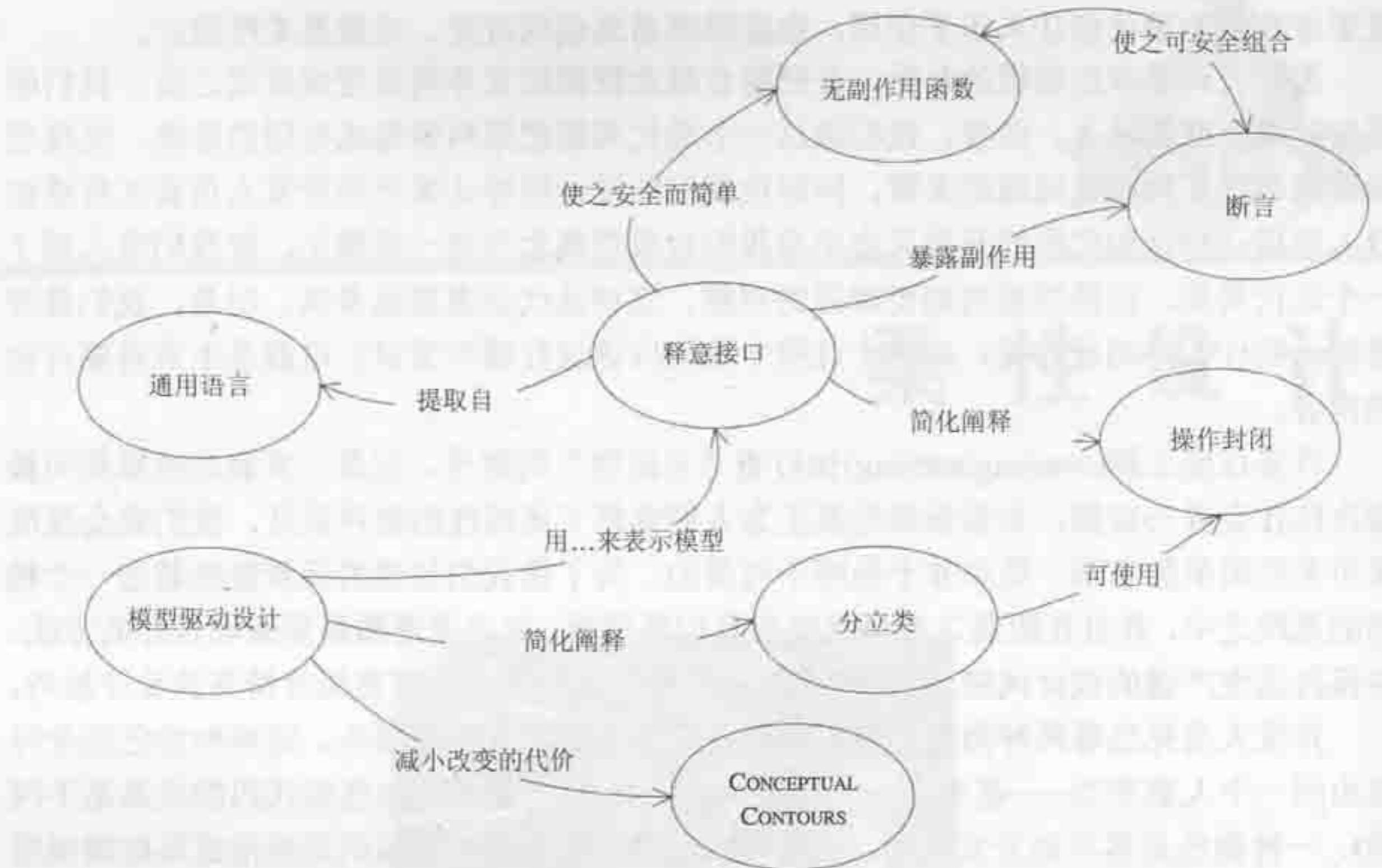


图 10-1 一些有助于获得柔性设计的模式

10.1 释意接口

在领域驱动设计中，我们希望考虑有意义的领域逻辑。如果代码仅仅产生了规则的效果，而没有将规则清晰地陈述出来，那我们就必须一步一步地考虑软件的工作过程。同样，如果函数只是执行代码、返回结果，而不显式地说明其意图，也会造成同样的问题。如果代码与模型缺乏清晰的联系，我们就很难理解代码的作用，也很难预测修改代码的影响。在前一章，我们研究了如何将规则和计算显式地建模出来。为了实现那些规则和算法，我们必须深入地理解其中所包含的详细细节。这些细节可以通过对象封装起来(这正是对象的优美之处)，使得客户代码能够非常简单，并可以用高层概念来理解。

但是，如果一个客户开发人员无法从接口中了解到，需要哪些信息才能有效地使用这个对象，他就必须深入对象的内部去设法理解其中的实现细节。阅读客户代码的人也必须这样做。这样，封装的大部分价值都已经丧失了。我们必须竭力避免“认知过载”：如果客户开发人员的头脑被“那个组件是如何工作的”这种问题纠缠不清的话，他就无



法理清自己的思绪来解决客户设计的复杂问题。这一点甚至在同一个人充当两种角色(使用他自己的代码开发客户程序)的情况下都不会例外，因为虽然他不需要去学习那些细节，但是一个人一次能同时考虑的问题个数是有限的。

如果开发人员必须考虑组件的实现才能使用它，那么封装就失去价值了。如果某个人(除了组件的原作者)必须根据实现来推测一个对象或操作的意图，那么他推测出来的很可能只是那个对象或操作偶然完成的。如果他猜错了意图，那么代码可能会在一段时间内正常工作，但是设计的概念基础已经被破坏了，两个开发人员的工作会变得有些驴唇不对马嘴。

将概念以类或方法的形式显式地建模出来之后，为了真正发挥它们的作用，我们还必须为那些程序元素起一个能反映其概念的名称。类和方法的名称可以极大地增进开发人员之间的交流，同时提高系统的抽象能力。

Kent Beck 提出用 Intention-Revealing Selector(Beck 1997)来构造方法名，使之能表达出方法的目的。设计的所有公共元素一起构成了它的接口，每个元素的名称都是一个解释设计意图的机会。类型名、方法名和参数名一起构成了一个释意接口(Intention-Revealing Interface)。

因此：

类和操作的名称应该表达出其效果和目的，同时又不涉及它们为了达到承诺而使用的手段，这样才能避免客户开发人员需要去理解内部实现。名字必须与通用语言一致，这样小组成员才能迅速理解它们的含义。在创建类和操作之前先编写测试代码，这样能促使您从客户开发人员的角度来考虑问题。

所有复杂的机制都应该用抽象接口封装起来。接口应该说明它们的意图，而不是实现手段。

领域的公共接口可以声明关系和规则，但不要声明这些关系和规则是如何保证的；可以描述事件和活动，但是不要描述这些事件和活动是如何执行的；可以阐明一个方程式，但是不要阐明解方程的方法。提出问题，但是不要给出问题的求解方法。

示例：重构调漆程序

涂料店需要一个程序来让顾客能看到标准涂料的混合结果。图 10-2 所示是最初的设计，它只有一个领域类。

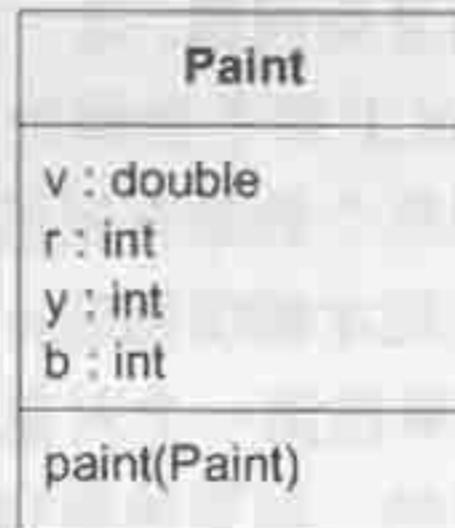


图 10-2 最初的 Paint 类

惟一能够猜出 paint(Paint)方法用处的途径就是阅读代码。

```
public void paint(Paint paint) {  
    v = v + paint.getV(); //After mixing, volume is summed  
    // Omitted many lines of complicated color mixing logic  
    // ending with the assignment of new r, b, and y values.  
}
```

好了，看来这个方法是想将两种涂料混合起来，结果是得到了更大的体积和混合的颜色。

让我们为这个方法编写测试，切换一下视角(下面的代码是基于 JUnit 测试框架编写的)。

```
public void testPaint() {  
    // Create a pure yellow paint with volume=100  
    Paint yellow = new Paint(100.0, 0, 50, 0);  
    // Create a pure blue paint with volume=100  
    Paint blue = new Paint(100.0, 0, 0, 50);  
  
    // Mix the blue into the yellow  
    yellow.paint(blue);  
  
    // Result should be volume of 200.0 of green paint  
    assertEquals(200.0, yellow.getV(), 0.01);  
    assertEquals(25, yellow.getB());  
    assertEquals(25, yellow.getY());  
    assertEquals(0, yellow.getR());  
}
```

通过这个测试只是开了个头。我们不会就此满足，因为测试代码并不能告诉我们它在测试什么。我们可以假设自己正在编写客户应用，然后按我们希望看到的使用 Paint 的方式来重写一个测试。最开始，这个测试会失败——实际上，它连编译都通不过。这



个测试使我们能站在客户开发人员的角度，来考察 Paint 对象的接口设计。

```
public void testPaint() {  
    // Start with a pure yellow paint with volume=100  
    Paint ourPaint = new Paint(100.0, 0, 50, 0);  
    // Take a pure blue paint with volume=100  
    Paint blue = new Paint(100.0, 0, 0, 50);  
  
    // Mix the blue into the yellow  
    ourPaint.mixIn(blue);  
  
    // Result should be volume of 200.0 of green paint  
    assertEquals(200.0, ourPaint.getVolume(), 0.01);  
    assertEquals(25, ourPaint.getBlue());  
    assertEquals(25, ourPaint.getYellow());  
    assertEquals(0, ourPaint.getRed());  
}
```

我们必须花时间来编写测试，借助测试来反映出我们希望看到的使用对象的方式。接下来，我们对 Paint 类进行重构，使之能够通过这个测试。

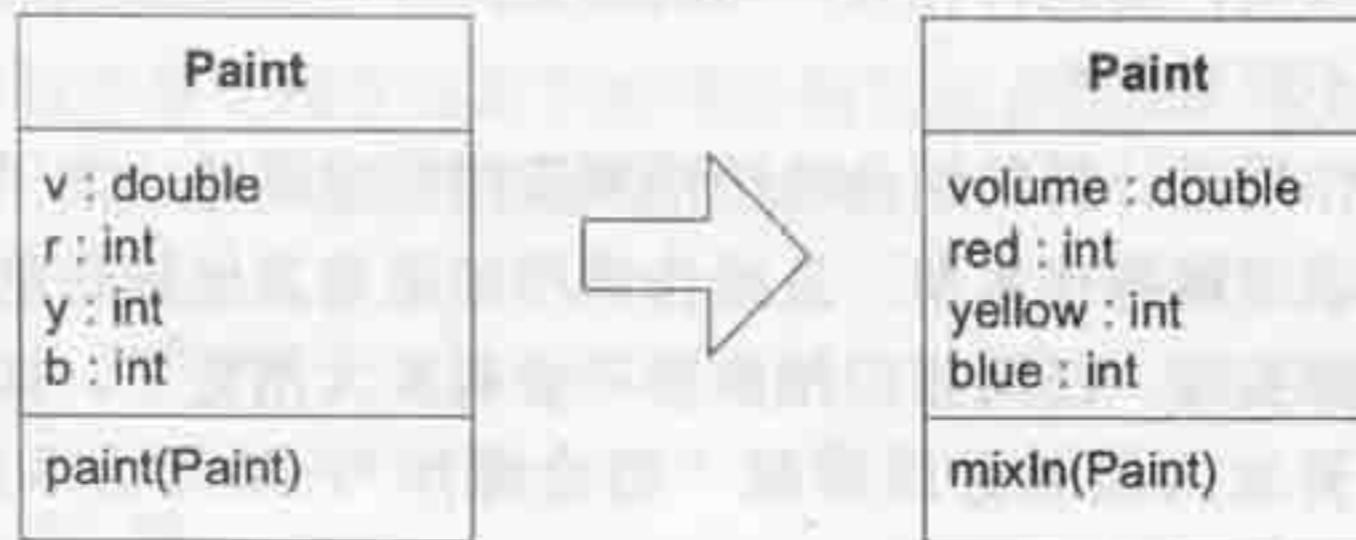


图 10-3 重构 Paint 类

读者也许不能从新的方法名了解到，“mixIn(混合)”另一个 Paint 对象会产生什么效果(要实现这一点我们需要断言，后面就会提到)。但是，这个名称足以引导读者开始使用这个类了，而且测试也为读者提供了使用示例。同时，这个方法名也使得客户代码的读者能够解释客户的意图。在本章的后面几个例子中，我们将对这个类继续进行重构，使之更加清晰。

我们可以把整个子领域切成分离的模块，并将其用释意接口封装起来。这种切割领域的方法可以用来调整项目的焦点和管理大型系统的复杂性。我们将在第 15 章“精炼”中更多地讨论这个话题，包括 Cohesive Mechanism 和 Generic Subdomain。

在接下来的两个模式中，我们将讨论如何才能确切地预测使用一个方法的结果。复杂的逻辑可以用无副作用函数(Side-Effect-Free Function)来安全地实现。改变系统状态的方法可以用断言(Assertion)来刻画。



10.2 无副作用函数

操作可以划分为两个大的类型：命令和查询。“查询”从系统中获取信息，这可能是访问变量中的数据，也可能是根据数据执行某种计算。“命令”又叫修改器(modifier)，它是一种能够使系统发生改变的操作(一个简单的例子是变量赋值)。在标准英语中，“副作用(side effect)”这个词隐含有“无意造成的结果”的意思，但是在计算机科学中，“副作用”意味着对系统状态产生的任何影响。在这里，我们把它的含义缩小为“任何将对后续操作造成影响的系统状态的改变”。

导致系统状态发生改变的操作显然不是“无意”地执行的，但为什么人们会接受并使用“副作用”这个词呢？我认为这种用法来自于复杂系统的经验。大多数操作都会调用其他操作，而那些操作又会调用另外的其他操作。只要存在这种任意深度的嵌套，我们就很难预测调用一个操作究竟会产生哪些结果。客户开发人员也许没有考虑到第二层和第三层上的操作会产生什么影响——这些影响无论怎么看都是一种“副作用”。复杂设计中的元素也要相互作用，而这种相互作用的结果同样可能无法预测。副作用这个词强调了这种相互作用的不可避免性。

多条规则的相互作用或计算的组合将使预测变得极度困难。为了预测调用一个操作的结果，开发人员必须理解操作本身，及操作调用的所有其他操作的实现。如果开发人员被迫去理解那些内部实现，任何接口抽象都不会有多大用处了。如果不能安全地对抽象的结果进行预测，开发人员就必须限制“组合爆炸”，从而为系统行为所能够达到的复杂性人为地设置了一个上限。

返回结果但是不产生副作用的操作称为函数(function)。同一个函数可以被多次调用，每次都会返回同一个结果。函数可以调用其他函数，而无需担心嵌套的深度。函数比其他产生副作用的操作更易于测试。因此，函数的风险更低。

显然，在大多数软件系统中不可避免都要用到命令，有两条途径可以减轻命令带来的问题。首先，我们可以将命令和查询严格地隔离到不同的操作中去，同时确保引起状态改变的方法不返回任何领域数据，并尽可能地简单。在不引起任何可观测副作用的方法中执行所有查询和计算(Meyer 1988)。

其次，我们往往可以换一种方法来构造模型和设计，使之不对已有的对象作任何修改。我们可以创建并返回一个新的值对象来代表计算的结果。这是一种常用的技术，我们将在下面的示例中向读者演示这一点。值对象可以在查询中创建出来，返回给调用者，然后被删除掉——这一点和实体不同，实体的生命周期是受到严格控制的。

值对象具有不变性，这意味着除了在创建过程中调用的初始化器，它们的其他操作



都是函数。值对象和函数一样，使用更安全，测试也更容易。如果一个操作把逻辑(或计算)与状态修改混在一起，那么它应该被重构为两个分离的操作(Fowler 1999, p. 279)。但是按照定义，这种将副作用隔离到简单的命令中的方法只对实体有效。在分离了命令和查询之后，我们可以作进一步重构，将执行复杂计算的职责分离到一个值对象中。创建一个值对象而不是直接修改现有状态，或者将整个职责放入一个值对象——这样往往可以把副作用彻底消除干净。

因此：

尽可能地将程序的逻辑放到函数中，以及只返回结果而不产生可观测副作用操作中。将命令(导致可观测状态产生改变的方法)严格地分离为非常简单的、不返回任何领域信息的操作。如果概念与职责明显匹配的话，我们还可以将复杂逻辑移入值对象，以进一步减少副作用。

无副作用函数，特别是具有不变性的值对象，使我们可以安全地对操作进行组合。如果我们通过释意接口将函数描述出来，那么开发人员无需理解其实现细节就能够使用它。

示例：再次重构调漆应用

涂料店的调漆程序要让顾客能看到标准涂料的混合结果。我们继续对上一个例子进行讨论，图 10-4 所示是我们在前面得到的领域类。

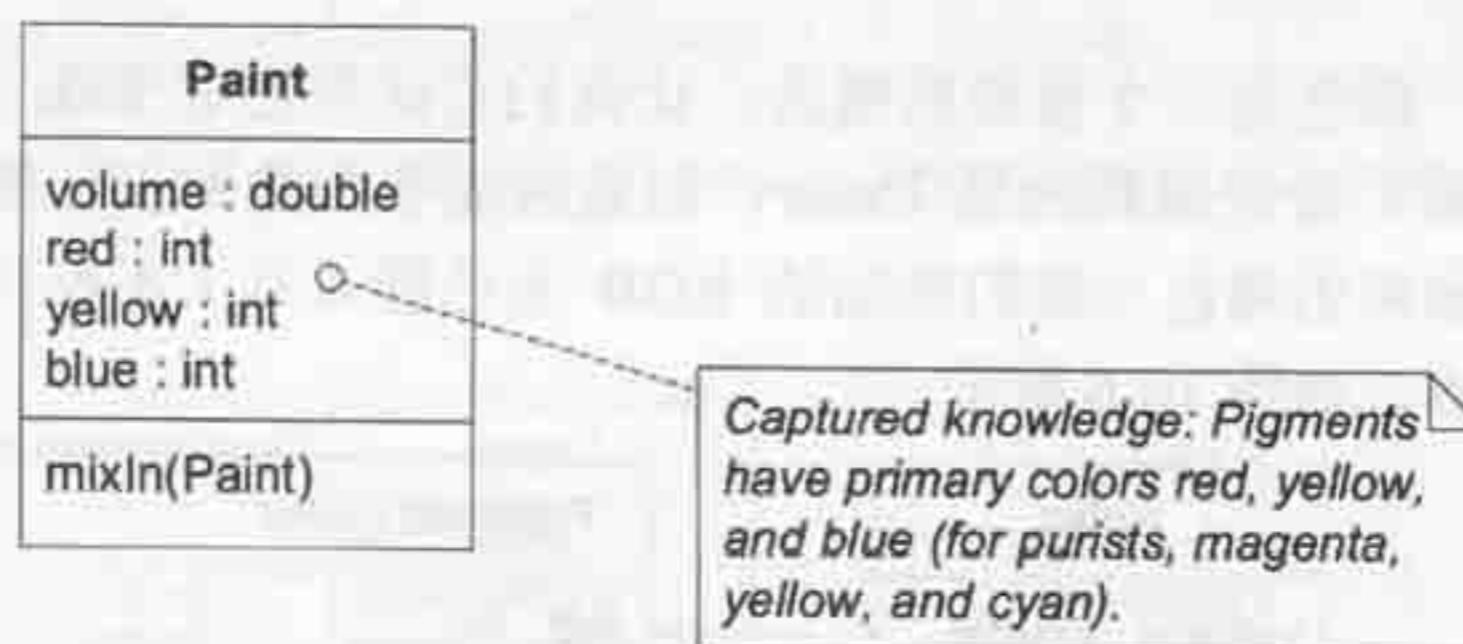


图 10-4 重构的 Paint 类

```
public void mixIn(Paint other) {
    volume = volume.plus(other.getVolume());
    // Many lines of complicated color-mixing logic
    // ending with the assignment of new red, blue,
    // and yellow values.
}
```

mixIn()方法做了许多事情，但是这个设计确实遵循了修改和查询分离的原则。这里有一个问题，那就是 Paint 2 这个对象(它被作为变元传递给 mixIn()方法)的体积被我们忽



略了，如图 10-5 所示。执行 mixIn()方法之后，Paint 2 的体积没有发生变化，这一点从概念模型上下文来说缺乏逻辑。Paint 类的开发人员可能觉得这不是一个问题，因为他对 Paint 2 对象在操作完成之后变成什么样子并不感兴趣。但是，我们很难预测这种副作用（Paint 2 的体积是否发生变化）所产生的后果。在讨论完断言之后，我们再回过头来解决这个问题。现在我们来看看颜色。

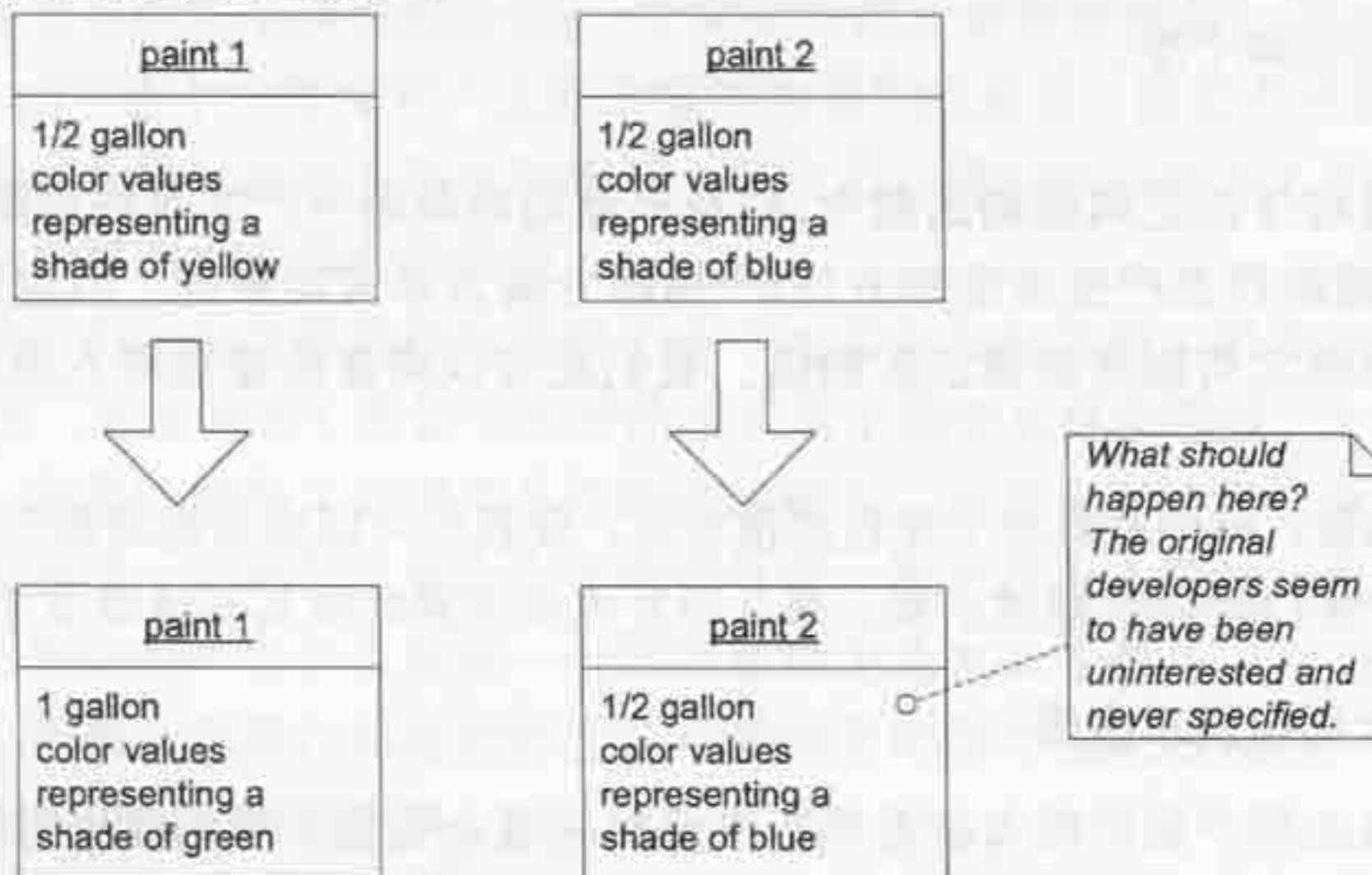


图 10-5 mixIn()方法的副作用

在这个领域中，颜色是一个重要的概念。让我们尝试着把它变成一个显式对象。我们应该怎么称呼它呢？首先想到的是 `Color`，但是前面学到的知识让我们理解了一个重要的事实，那就是涂料的调色与我们熟知的 RGB 呈色模式是大不相同的。因此，它的名称必须反映这一点，如图 10-6 所示。

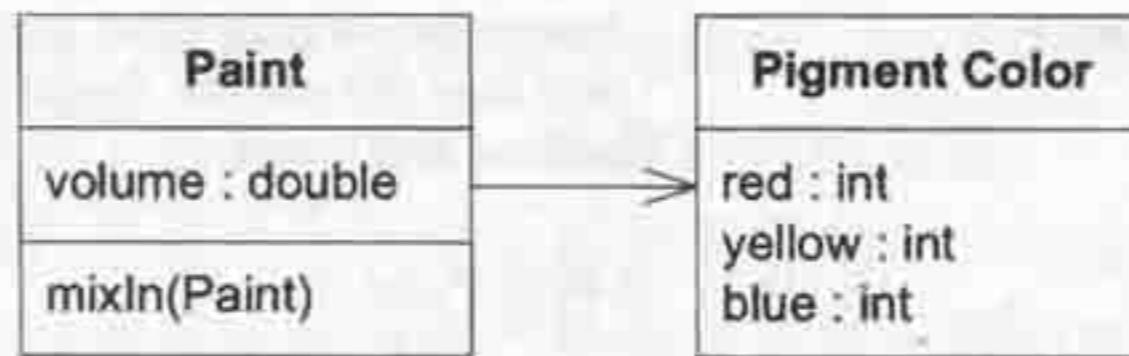


图 10-6 分解出 Pigment Color

`Pigment Color`(颜料颜色)分解出来之后，我们的设计确实比前面的版本更易于交流了。但是，颜色的计算还是留在 `mixIn()` 方法中。既然我们把颜色数据移出来了，那么就应该把它的行为也一起移出来。在移动之前，注意 `Pigment Color` 是一个值对象，因此它应该具有不变性。当我们混合涂料的时候，发生改变的是 `Paint` 对象本身，因此 `Paint` 是一个具有生命周期的实体。而 `Pigment Color`(例如，某种特定色调的黄色)永远保持原来的颜色。混合涂料将产生一个新的 `Pigment Color` 对象，用来表示新的颜色，修改结果如



图 10-7、10-8 所示。

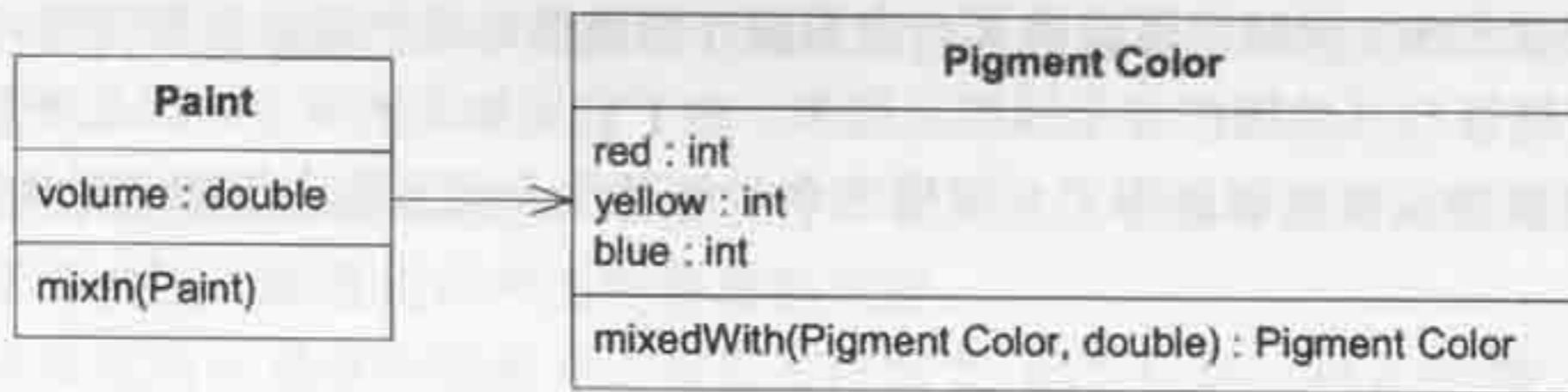


图 10-7 添加行为

```

public class PigmentColor {

    public PigmentColor mixedWith(PigmentColor other,
                                  double ratio) {
        // Many lines of complicated color-mixing logic
        // ending with the creation of a new PigmentColor object
        // with appropriate new red, blue, and yellow values.
    }
}

public class Paint {

    public void mixIn(Paint other) {
        volume = volume + other.getVolume();
        double ratio = other.getVolume() / volume;
        pigmentColor =
            pigmentColor.mixedWith(other.pigmentColor(), ratio);
    }
}

```

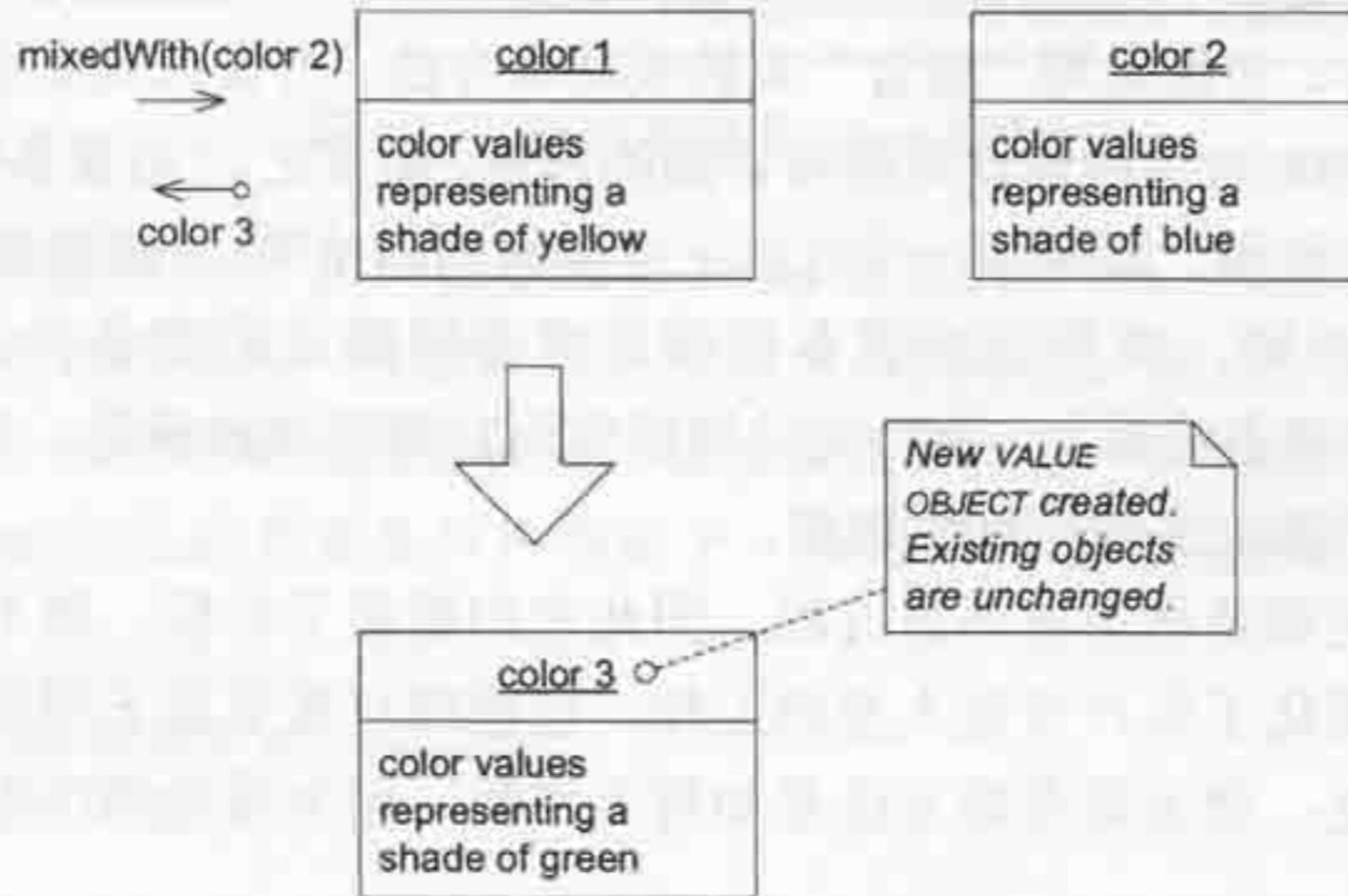


图 10-8 混合产生新的颜色



现在 Paint 中的修改代码已经尽可能简单了。新的 Pigment Color 类捕捉了领域知识并将其显式地描述出来，同时它还提供了一个无副作用函数。这个函数非常好懂，易于测试，使用安全，并能够与其他操作安全地组合起来。由于它是如此的安全，因此我们可以认为调色的复杂逻辑确实被封装起来了。使用这个类的开发人员无需去理解它的实现细节。

10.3 断言

将复杂的计算分离到无副作用函数中可以减小问题的规模，但实体中的命令仍然可以产生副作用，因此要使用这些对象就必须理解那些命令的执行结果。我们可以用断言 (assertion) 来显式地描述副作用，使之更易于处理。

如果命令不包含复杂的计算，通过检查代码来理解它确实很容易。但是，大的设计元素是由小的设计元素构成的，一个命令也可能会调用其他命令。使用高层命令的开发人员必须理解所有底层命令的结果——这使得封装失去了含义。同时，由于对象接口并没有对副作用进行限制，因此实现同一个接口的两个子类可以产生不同的副作用。使用这两个子类的开发人员必须清楚哪个是哪个，才能预测结果——这又使得抽象和多态失去了含义。

如果操作的副作用仅仅由其实现来隐含地定义，而设计中又使用了大量委托，那么设计中的因果关系将会变成一团乱麻。理解这种程序的惟一方法就是去跟踪执行过程中的每个分支。封装变得毫无意义，抽象也由于需要跟踪具体执行而失去作用。

我们需要一种方法来理解设计元素的含义和操作的执行结果，而无需深入研究其内部。释意接口为我们提供了一部分答案，但这种非形式的意图说明还不够。“契约式设计”流派往前更进了一步，它通过用“断言”来约束类和方法，开发人员必须保证这些断言总是满足的。Meyer (1988) 对这种设计风格有详细的讨论。简言之，“后置条件(postcondition)”描述了一个操作的副作用，即调用一个方法一定会得到的结果；“前置条件(precondition)”类似于一份精制的合同，要保证后置条件满足就必须满足前置条件。类不变量(Class invariant) 是关于对象状态的断言，执行完任何操作后它都应该被满足。不变量还可以是针对整个聚合声明的严格定义的完整性规则。

所有断言描述的都是状态而不是过程，因此它们更易于分析。类不变量不仅有助于刻画类的含义，还简化了客户开发人员的工作，它使得对象更易于预测。只要您信任后置条件所作出的保证，就无需考虑方法是如何工作的，因为委托所产生的效果应该已经包含在断言之中了。

因此：



声明操作以及类和聚合不变量的后置条件。如果您所使用的编程语言不能直接编写断言，那就把它们写成自动化的单元测试。如果项目开发过程的风格允许，还可以把它们写入文档或图中。

努力提高模型概念的内聚性，以便引导开发人员从中推导出符合原意的断言。这样可以加快学习曲线，降低代码中出现矛盾的风险。

虽然许多面向对象语言现在并不直接支持断言，但它仍是一种非常强大的设计思路。自动化的单元测试可以从一定程度上弥补语言支持的欠缺。由于断言都是对状态而非过程的描述，因此它可以方便测试的编写。在搭建测试时先保证前置条件被满足，然后在执行完毕时再检查后置条件是否也被满足。

把不变量前置条件和后置条件清晰地声明出来可以帮助开发人员理解一个操作或对象所能产生的结果。理论上，只要断言之间不存在自相矛盾，开发人员就不会出现误解。但人们并不是在自己的头脑里“编译”断言，他们会做出错误的推测，或者篡改模型的概念。因此，力求让模型意义明确并满足应用需要，这一点是非常重要的。

示例：回到调漆程序

记得在前一个例子中，我们注意到了一个问题，那就是 Paint 类的 mixIn(Paint) 操作的变元存在一些不明之处，为了方便起见，图 10-9 再次给出了 Paint 类和 Pigment Color 类。

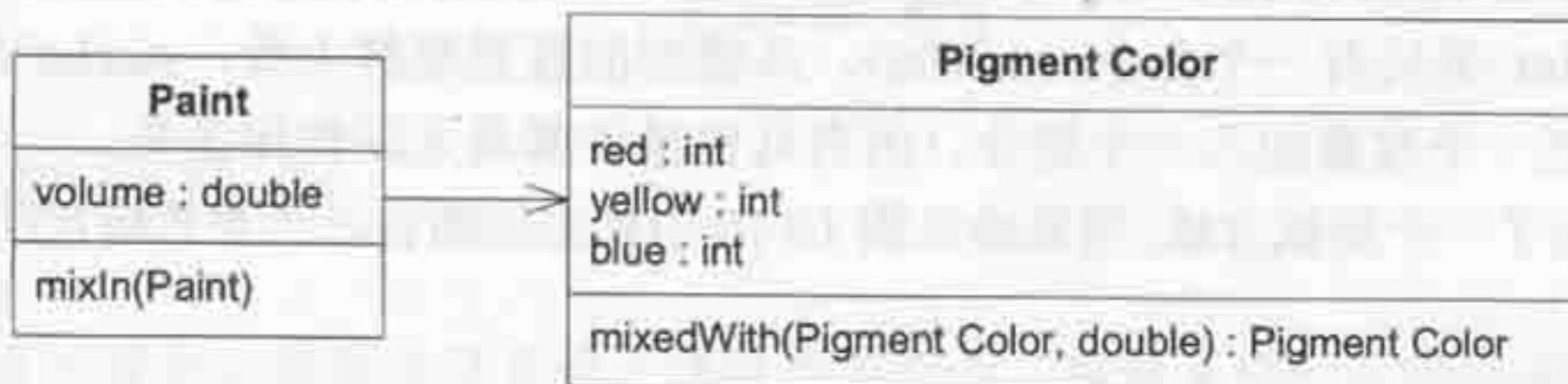


图 10-9 Paint 类和 Pigment Color 类

接收者对象的体积增加了(增加量就是 Paint 变元的体积)。按照我们通常的理解，将两份涂料混合起来应该使另一份涂料减少同样大小的体积，使之体积变为 0 或者完全消失。现在的实现并没有对变元进行修改，而修改变元正是一种特别危险的副作用。

我们把 mixIn() 方法的后置条件声明如下，作为后面讨论的基础：

执行 p1.mixIn(p2) 之后：

p1.volume 增加了 p2.volume
p2.volume 不变

问题在于，这会使开发人员犯错误，因为这种特性与我们希望让开发人员考虑的概念(把 Paint 想象为现实世界中的涂料)不符。我们可以将另一份涂料的体积改为 0，直接把这



个问题解决掉。虽然修改不变量是一种糟糕的行为，但是它非常简单，也符合直觉。我们可以声明一个不变量：

混合涂料不改变涂料的总体积。

但是等一下！当开发人员思考这种选择的时候，他们会突然领悟到，当初我们之所以那样设计，竟然还有非常充分的理由——程序最终必须把混合之前的涂料一一列举出来，毕竟它的最终目的是要告诉用户用哪几种涂料进行混合。

这样，这个体积模型在逻辑上保持一致，却又不能满足其应用需求了。看起来好像是进退两难。我们是不是应该硬着头皮坚持这个古怪的后置条件，然后试着通过沟通来弥补它呢？世间万物并非全都符合直觉。虽然有时直觉是最好的答案，但是在这个例子中，问题似乎表明我们遗漏了某个概念。让我们寻找一个新的模型。

一个更清晰的模型

在寻找一个更好的模型时，我们比最初的设计者多了一个明显的优势，那是因为知识消化和向更深层理解重构在中间阶段发生了。例如，我们用一个值对象上的无副作用函数来计算颜色，这意味着我们可以在任何时候根据需要反复多次执行这种计算。这为我们带来了方便。

我们似乎让 Paint 承担了两种不同的基本职责。让我们试着把它们分割开来。

现在 Paint 类只有一个命令：mixIn()。从模型的直观理解上看，mixIn()的作用很明显，仅仅是把一个对象加入一个集合。所有其他操作都是无副作用函数。

下面给出了一个测试方法，用来确认图 10-10 中列出的断言之一是否满足(用 Junit 测试框架)。

```
public void testMixingVolume {
    PigmentColor yellow = new PigmentColor(0, 50, 0);
    PigmentColor blue = new PigmentColor(0, 0, 50);

    StockPaint paint1 = new StockPaint(1.0, yellow);
    StockPaint paint2 = new StockPaint(1.5, blue);
    MixedPaint mix = new MixedPaint();

    mix.mixIn(paint1);
    mix.mixIn(paint2);
    assertEquals(2.5, mix.getVolume(), 0.01);
}
```

这个模型捕获和交流了更多的领域知识。不变量和后置条件符合人们的常识，这使得设计更易于维护和使用。



释意接口方便了沟通，而无副作用函数和断言提高了可预测性，二者的结合将使得封装和抽象非常安全。

可重组元素的下一个要素是有效的分解……

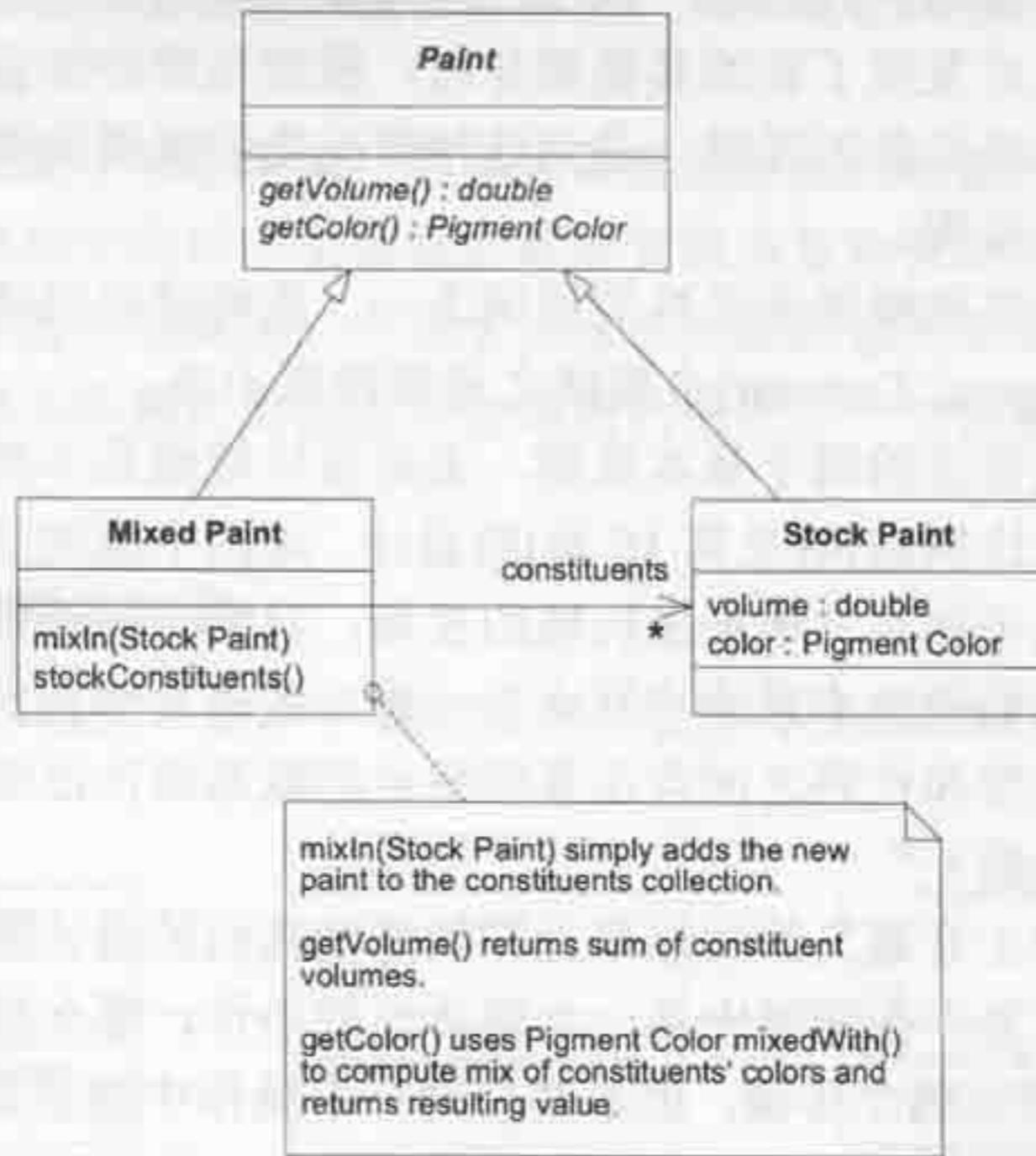


图 10-10 断言

10.4 概念轮廓

我们有时要把功能细分开来使之能灵活地组合，有时又要把功能集中起来以便封装复杂性。有时我们希望找到一种统一的粒度，使得所有类和操作都具有类似的规模。这些做法和想法都把问题看得太简单，不能当作普遍规律来用。但是，它们都来自于对一些基本问题的考虑。

如果把模型或设计中的元素嵌在一个整体构造中，就会造成功能上的重复，因为其外部接口无法提供客户可能会关心的所有信息。我们很难理解这些元素的含义，因为各种不同的概念都混在了一起。

另一方面，把类和方法细分可能会导致客户不必要地复杂化，强迫客户开发人员去理解那些小碎片是如何配合起来的。更糟糕的是，概念可能会完全丢失。半个铀原子就不再是铀了。此外，毋庸置疑，粒度的大小不是惟一的重要信息，还要看粒度是否符合具体的情况。

菜谱式的规则是行不通的。但是，大多数领域都蕴含着某种逻辑上的一致性，否则



它们不可能自成一体。这并不是说领域都是百分之百的一致；人们讨论领域的方式当然也不是一致的。但是，领域中一定有某个地方存在着一种旋律和共鸣，不然的话建模就毫无意义了。由于存在着这种内在的一致性，因此如果我们的模型能够与领域的某个部分发生共振，那么当以后发现了新的其他部分时，模型与那些部分也是一致的可能性就更大。有时候模型难以适应新的发现，遇到这种情况我们就要向更深层理解重构，力图使模型能够适应下一个发现。

这也是反复重构最终能够带来柔性的原因之一。重构使代码适应了新理解的概念和需求，概念轮廓(Conceptual Contour)也就随之逐渐浮现出来。

高内聚、低关联是设计的两个基本原则，无论设计规模是小到单个方法的设计，还是大到类和模块甚至大比例结构(见第 16 章)的设计。这两个原则对于概念和编码具有同样的重要性。为了避免不知不觉地形成机械的视角，我们必须经常用自己的直觉去感触领域的基本实质，让我们的技术思维“回火”一番。在每次作决定时，我们都要自问：

“这是不是由于当前模型和代码之间存在某些特定的联系而作出的权宜之计？它是否反映了潜在领域的某种轮廓？”

只要能找到在概念上有意义的功能单元，就能使我们的设计既灵活又易于理解。例如，如果“添加两个对象”在领域中是一个整体性的动作，那么就按那个含义来实现方法，而不要把 add() 分解成两个步骤，也不要在同一个操作中越位到下一个步骤。从稍大一点的尺度上看，每个对象都应该是完整的一个概念，一个 WHOLE VALUE¹。

同样地，在任何领域中都有一些用户对那些细节不感兴趣的地方。拿我们假想的调漆应用来说，用户们不是加入红颜料或蓝颜料，而是将它们混合成涂料(3 种颜料都包含在涂料之中)。对于那些没有必要进行分解或重排的东西，我们就把它当成一个整体来处理，这样既能避免混乱，也使得真正需要重新组合的东西更容易看清楚。如果用户使用了某种物理设备，使得他们能够单独加入各种颜料，那么领域就发生变化了，我们可能还要对单个颜料进行处理。涂料工程师可能需要更为精细的控制，涉及到一整套的其他分析，也许会产生一个比调漆程序中抽象的 Pigment Color 还要详细得多的涂料构成模型。但是，对于调漆应用项目中的任何人而言，这些都是毫不相干的。

因此：

将设计元素(操作、接口、类和聚合)分解为内聚的单元，同时把您对领域中一些重要部分的直观认识考虑进去。通过连续的重构来观察模型在哪些地方发生改变、哪些地方保持稳定，寻找能够对这种差异作出解释的内在的概念轮廓，尽早使模型与那些使领域自成一体的稳定面紧密地切合起来。

我们的目标就是得到一批简单的接口，这些接口合理地组合在一起，使我们能够用

¹ Whole Value 模式，由 Ward Cunningham 提出。



通用语言清晰易懂地描述出来；同时，那些毫不相干的事情不会使我们分心，也不会造成维护问题。这往往要通过重构才能达到，我们很难一开始取得这种效果。但是，仅仅从技术的角度去重构是永远达不到这个目标的，只有通过向更深层理解重构，概念轮廓才会浮现出来。

即使我们的设计遵循了概念轮廓，也可能需要进行修改和重构。如果连续的重构越来越趋向于局部化，模型中的许多主要概念都没有因为重构而动摇，那就说明模型已经比较贴合领域了。如果我们遇到的一个需求导致了大范围修改和对象及方法的细分，那就是我们对领域的理解尚需精化的标志，它为我们指出了一个使模型更加深化、使设计更加具有柔性的机会。

示例：应计费用的概念轮廓

在第9章中，我们根据对会计学概念的深入理解将贷款跟踪系统重构为如图10-11所示：

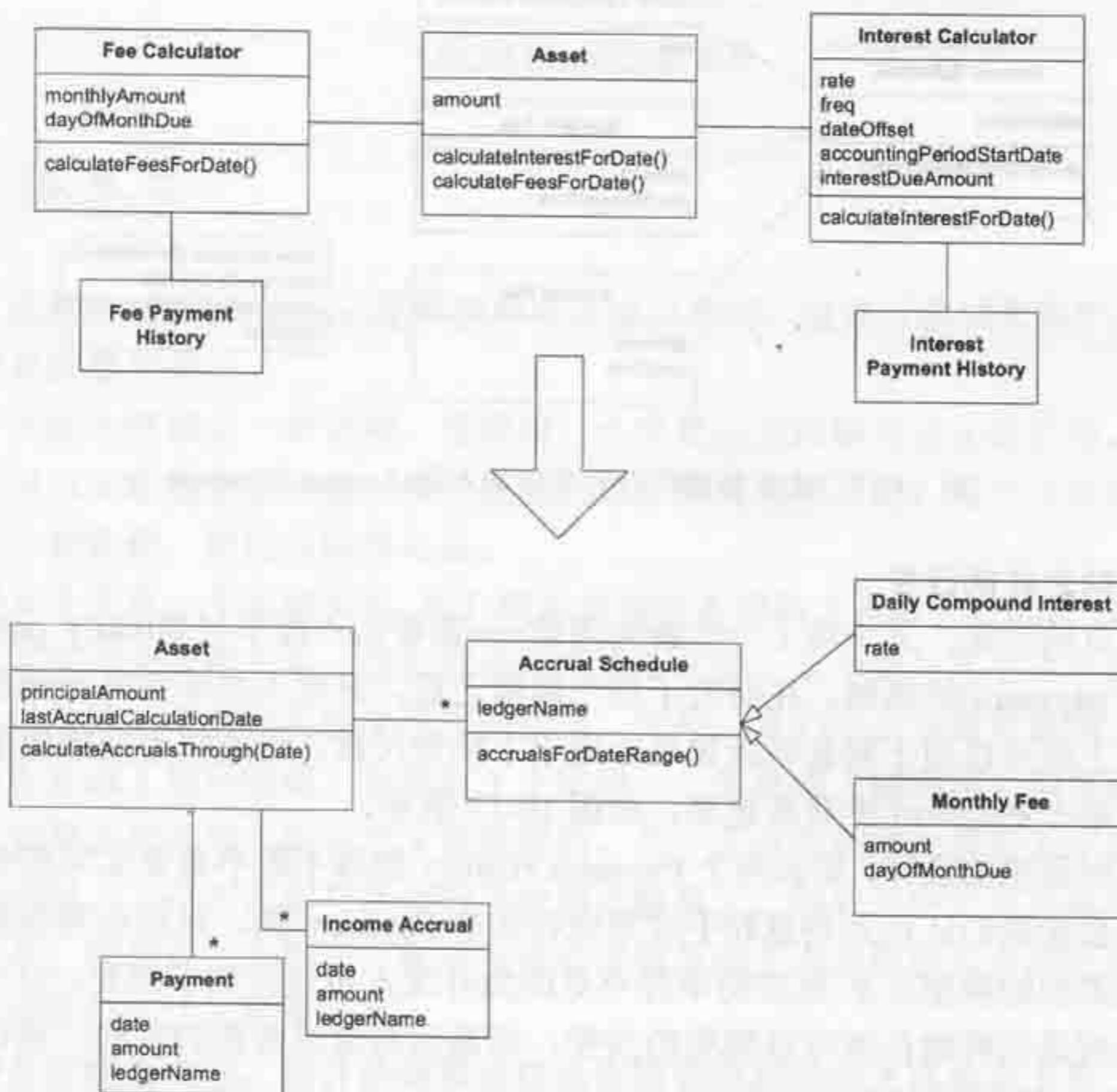


图 10-11 重构的贷款跟踪系统



新模型只比旧模型多包含一个对象，但是职责的分配情况已经大大改变了。

计划(schedule)原来是在 Calculator 类中通过 case 逻辑来实现的，现在已经被分解为多个离散的类，分别用来处理不同类型的手续费和利息。另一方面，手续费和利息的付款以前是分开的，现在合到一起来了。

由于新的显式概念与领域形成了共振，而且 Accrual Schedule 的层状结构具有很好的内聚性，开发人员相信模型更好地符合了领域的某些概念轮廓。

对于加入新的 Accrual Schedule 这种改变，开发人员可以非常放心。模型对那些需求早就张网以待了。这样，开发人员选择的模型不仅使现有的功能更清晰简单，还使之更易于引入新的 Schedule。但是，她是否已经找到了一个概念轮廓，能够帮助领域设计随着应用和业务的发展而改变呢？没有任何方法可以保证一个设计能够处理任何意料之外的改变；但是开发人员感到，她的设计已经更易于容纳那些改变了，如图 10-12 所示。

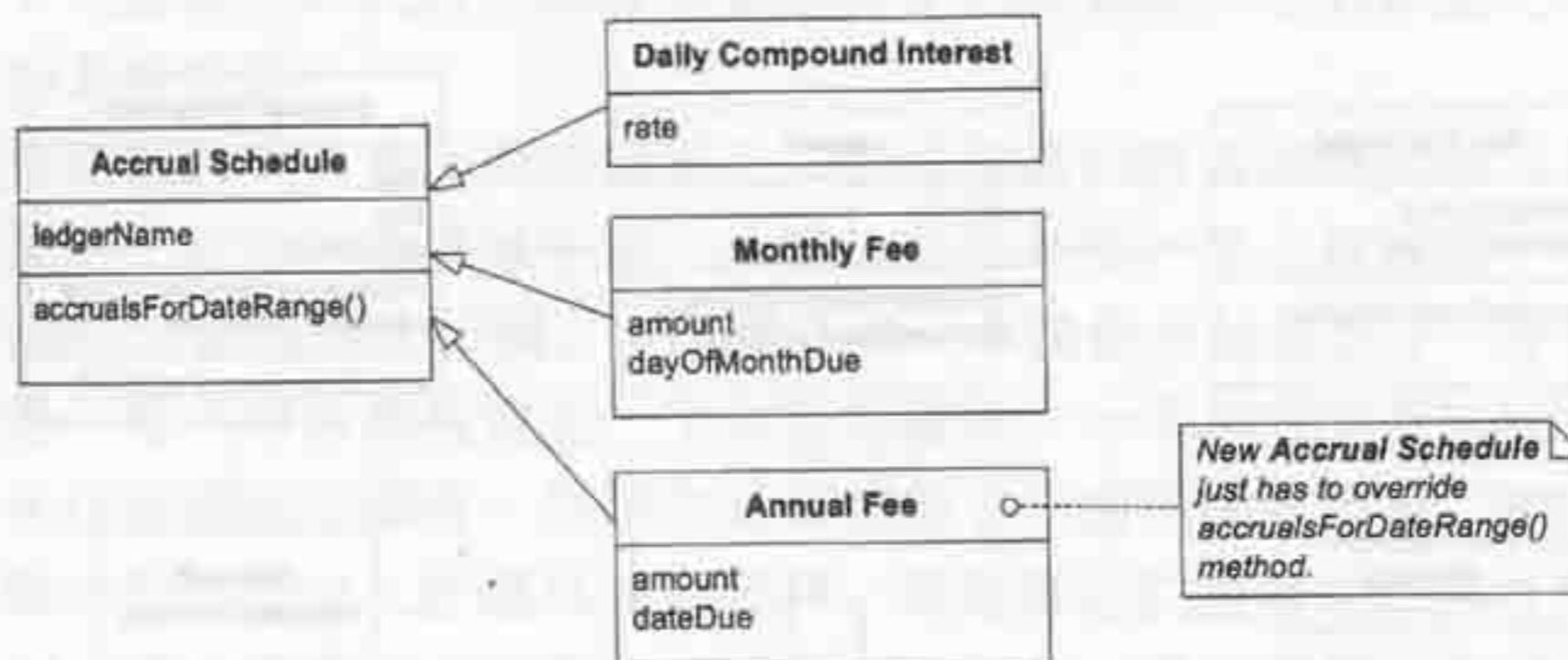


图 10-12 这个模型可以容纳新加入的 Accrual Schedule

一个意料之外的改变

随着项目的进展，又出现了一个新的需求——需要加入提早付款(early payment)和推迟付款(late payment)的规则。在研究了这个问题之后，开发人员非常高兴地看到，同样的规则实际上既可以用于利息付款也可以用于手续费付款。这意味着新的模型元素可以很自然地与单个 Payment 类联系起来，如图 10-13 所示。

如果使用原来的设计，那么两个 Payment History 类将不得不重复实现那些规则。(这种困难可能会使我们认识到利息和手续费应该共享 Payment 类，从而引导我们顺着另一条途径得到类似的模型)。扩充的简单性不是因为开发人员预计到了改变，也不是因为她的设计通用到足以容纳任何可以想到的改变，而是因为通过前面的重构，设计已经很好地与领域的内在概念贴合起来了。



释意接口使得客户可以把对象描述为有意义的单元，而不仅仅是机制。无副作用函数和断言使我们能够安全地使用这些单元，并对它们进行复杂的组合。概念轮廓的浮现使模型的一些部分更加稳定，同时使那些单元更加符合直觉，更易于使用和组合。

但是，如果模型中存在过多的交叉依赖，我们就会被迫一次性考虑太多的问题，从而可能还是会遇到“概念过载”的问题。

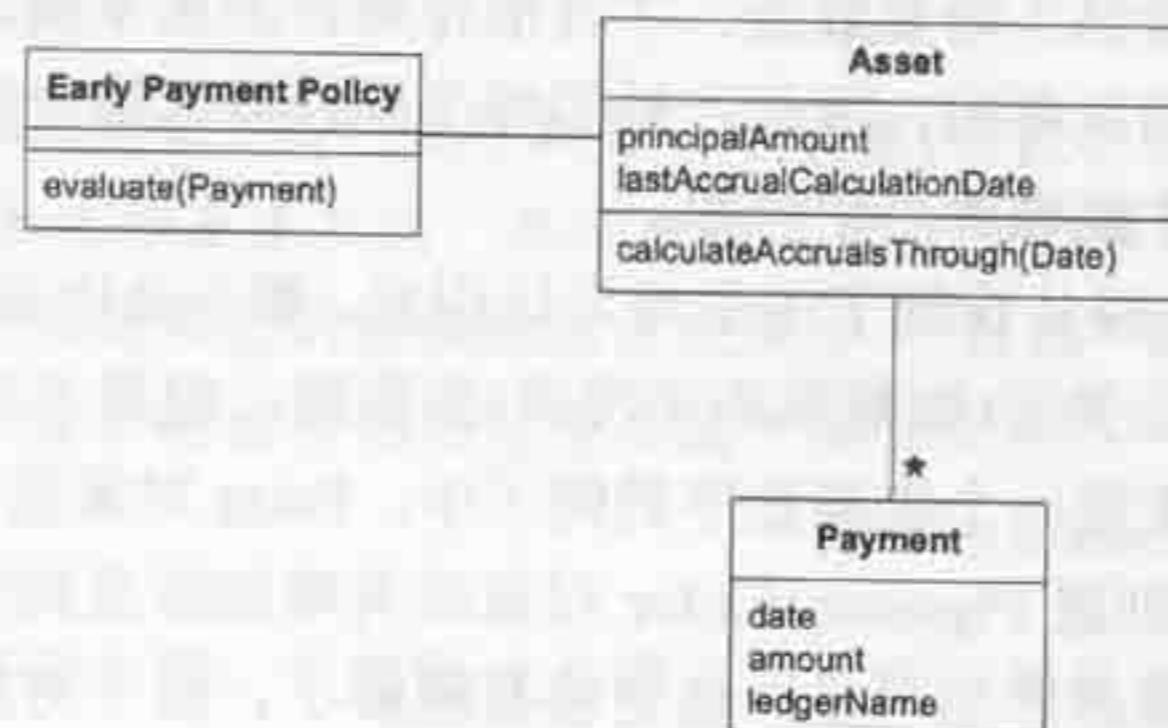


图 10-13 添加新模型

10.5 孤立类

交叉依赖(interdependency)使模型和设计难以理解，也难以测试和维护。而且，交叉依赖很容易积累起来。

每个关联当然都是一种依赖，要理解一个类就必须理解与它关联的类。与一个类关联的类又可以关联到更多的类，那些类我们也必须要理解清楚。每个方法的每个变元的类型也是一种依赖。返回值同样如此。

如果某个类有一个依赖关系，我们就必须同时考虑两个类以及它们之间关系的本质。如果某个类有两个依赖关系，我们就必须考虑 3 个类，这个类与其他两个类之间的关系的本质，以及这 3 个类和其他类存在的联系。此外，我们还必须提高警惕，注意这 3 个类之间是否形成了循环依赖。如果有 3 个依赖……那就成了越滚越大的雪球。

模块和聚合都是用来限制交叉依赖网的。当一个具有高度内聚性的子领域被分离到一个模块中时，一批对象都与系统的其他部分解耦了，因此其中只有有限个相互关联的概念。但是，即便是一个模块，如果不是近乎偏执地去追求控制其内部的依赖的话，要把它理解清楚也会让人煞费心思。

即使是在模块内部，设计的理解难度也会随着依赖的加入而急剧上升。这会进一步加重大脑过载的问题，进而限制设计开发人员能够处理的复杂度。如果存在隐含概念，



那么它造成的大脑负荷比显式的引用还要大。

模型精化可以精炼到的程度是，模型中每个保留下来的连接都代表了概念之间某种根本的东西。在一个重要的子集中，依赖的个数可以减小到 0，从而得到一个能被完全孤立地理解的类，它只引用一些原始类型和基本的类库概念。

每种编程环境都有一些普遍使用的、常驻于大脑之中的基本概念。例如，Java 的原始类型和一些标准类库提供了诸如数值、字符串和集合的基本概念。客观地说，“整数”这种概念是不会增加大脑负荷的。除此之外的每个额外概念都会使大脑过载，因为我们必须将其保留在头脑中才能理解一个对象。

隐含概念，无论是已经认识到了还是尚未认识到，都不会比显式的引用好到哪里去。虽然我们可以忽略对原始类型(如整数和字符串)的依赖，但是不能忽略那些类型的对象所代表的含义。例如，在第一个调漆程序的例子中，Paint 对象包含 3 个公有整数：red、yellow 和 blue 颜色值。创建 Pigment Color 对象没有增加涉及到的概念数或依赖数，但是它使那些已经存在的概念更加明显，也更容易理解了。另一方面，Collection 的 size() 操作返回一个 int 来简单地表示一个合计数，而这是整数的基本含义，因此其中没有隐含新的概念。

每个依赖都是值得怀疑的，除非能证明它是对象背后的概念的根本。从分解模型概念本身开始，就应该对依赖关系作仔细的检查。然后，我们还要注意每个单独的关联和操作。精心选择模型和设计可以使依赖大为减少——通常是减为 0。

低关联是对象设计的基本原则，只要有可能就应该这样做。从图中把所有其他概念全部排除出去，只剩下一个完全自包含的类，这样我们就能单独观察和理解它了。每个这种自包含的类都明显地降低了理解一个模块的难度。

依赖于同一个模块中的其他类比依赖一个外部的类要好。如果两个对象自然地紧密关联在一起，那么会有多个操作把它们同时包含进来，这实际上把二者关系的本质清晰地显露出来了。我们的目标不是消除所有依赖，但是要消除所有非本质性的依赖。每消除一个依赖都进一步解放了开发人员，使之能够把精力集中起来处理剩下的概念性依赖关系。

力图把最复杂的算法分离到孤立类(Standalone Class)中，这也许可以通过建模联系更紧密的类所特有的值对象来做到。

涂料这个概念在本质上是与颜色的概念相关的，但是在考虑颜色(甚至颜料)时无需考虑涂料。通过把这两个概念显式地建模出来并精炼二者的关系，我们得到了一个重要的单向关联，而封装了大部分复杂算法的 Pigment Color 类也能够单独观察和测试了。

低关联是减轻概念过载问题的基本方法。孤立类是低关联的极致。



消除依赖不是说要武断地把所有东西都降解为原始类型，使模型变成一个哑巴。本章的最后一个模式，操作封闭(Closure Of Operation)，就演示了一种在减少依赖的同时又能保持丰富接口的技术。

10.6 操作封闭

如果将两个实数相乘，我们就得到另一个实数[实数包括所有有理数和所有无理数]。因为这一点永远都是正确的，我们就说实数是“在乘法操作下是封闭的”——没有其他的方法可以使乘法的结果脱离实数集合。“封闭”是指如果把集合的任意两个元素组合起来，组合的结果仍然包含在这个集合之中。

—The Math Forum, Drexel University

依赖毫无疑问总是存在的，只要概念从根本上具有这种依赖，那并不是坏事。如果把接口降解到只剩下一些处理原始类型的方法，那么接口的表达能力也就枯竭了。但是，我们往往会在接口中引入许多不必要的依赖甚至整个概念。

大部分我们感兴趣的对像所做的事情不能仅用原始类型来刻画。

精化设计的另一个常用实践是所谓的“操作封闭”。这个名称来自于最精简的概念系统——数学。 $1 + 1 = 2$ 。加法在实数集合中是封闭的。数学家们崇尚不引入多余的概念，而封闭性就为他们提供了这样一种方法，使他们可以在无需引入任何其他概念的情况下定义一个操作。我们对于数学的精练已经习以为常了，以至于可能很难领体会到那些小窍门能有多么强大。但是，封闭性在软件设计中也被广为使用。XSLT 的基本用途是将 XML 文档转换为另一个 XML 文档，这种类型的 XSLT 操作在 XML 文档集合下就是封闭的。封闭性极大地简化了操作的理解，而且考虑把封闭操作串接或者组合起来也不是件难事。

因此：

在适当的情况下，把操作的返回值与其变元定义成相同的类型。如果在计算中要用到实现者(implementer)的状态，那么实现者实际上就是操作的一个变元，即操作的变元和返回值应该与实现者具有同一种类型。这样的操作对于这种类型的实例集合就是封闭的。封闭的操作为我们提供了一个高层接口，同时又无需引入任何对其他概念的依赖。

这个模式更多地应用于值对象的操作中。由于实体的生命周期在领域中具有重要意义，因此随手创建一个实体是行不通的。有的操作可能会在实体类型下封闭，例如我们可以让一个 Employee 对象返回它的主管(也是一个 Employee 对象)。但是，实体这种概念一般来说不大可能成为计算的结果。因此，我们很大程度上都是在值对象中寻找应用



这种模式的机会。

操作还可以在某种抽象类型下封闭，此时操作的变元可以是不同的具体类型。例如，加法在实数集合下封闭，而实数既可以是有理数也可以是无理数。

您可能有过这种经验，在设法减少交叉依赖、增加内聚性的过程中，有时会部分地应用到这种模式。例如，变元和实现者匹配但是返回类型不同，或者返回类型和实现者匹配但是变元不同。这些操作不是封闭的，但是它们确实能带来某些封闭的好处。如果额外的类型是原始类型或者基本类库中的类型，那么这些操作的效果就和封闭相差无几了。

前面的例子中，Pigment Color 的 mixedWith() 操作在 Pigment Color 下是封闭的，在本书中还有许多其他的例子。下面的例子说明，即使操作并不是完全的封闭，也能从封闭性中获得很大的好处。

示例：从集合中筛选

在 Java 中，如果要从一个 Collection 中筛选出一个元素子集，我们首先要得到一个 Iterator(迭代器)，然后用迭代器遍历所有元素，逐个进行测试，把那些符合条件的元素集中到一个新的 Collection 中来。

```
Set employees = (some Set of Employee objects);
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
    Employee anEmployee = (Employee)it.next();
    if (anEmployee.salary() < 40000)
        lowPaidEmployees.add(anEmployee);
}
```

概念上，我们只是筛选了一个集合的子集。那我们需要 Iterator 这个额外的概念(还有它那些复杂的机制)做什么呢？在 Smalltalk 中，我们可以调用 Collection 的 select 操作，将测试作为变元传给它。这个操作的返回值是一个新的 Collection，其中包含了所有通过了那个测试的元素。

```
employees := (some Set of Employee objects).
lowPaidEmployees := employees select:
    [:anEmployee | anEmployee salary < 40000].
```

Smalltalk 的 Collection 还提供了其他一些这样的函数，能够返回不同具体类型的派生 Collection。这些操作不是封闭的，因为它们需要一个“块”作为变元。但是，块是

Smalltalk 中基本的库类型，因此它们不会增加开发人员的大脑负荷。由于返回值与实现者类型一致，因此这些操作可以像一连串的过滤器一样串接起来，非常便于编码和阅读。它们没有引入与筛选子集这个问题无关的额外的概念。

本章中给出的模式演示了一种通用的设计风格，以及一种设计的思维方法。易于理解的、可预测和易于交流的软件将使得抽象和封装更有效率。模型可以分解，使得其中的对象使用和理解起来更简单，但同时仍然具有功能丰富的高层接口。

这些技术要有相当高级的设计技巧才能发挥作用(有时甚至在编写客户时都是如此)。细节设计和实现决策的质量对模型驱动设计能否发挥作用有着明显的影响，实际上，只要有少数几个开发人员犯糊涂就足以使我们的项目偏离目标了。

总之，如果一个小组希望培养自己的建模和设计技巧，那么他们可以使用这些模式及其反映出来的思维方法来进行工作，这样得到的软件可以由开发人员不断改进，从而生产出复杂的软件。

10.7 声明性设计

虽然用断言来进行测试并不是一种严格的形式化方法，它还是能引导我们获得更好的设计。不过，手工编写的软件是没有任何方法能够真正保证其正确性的。例如，只要让代码产生额外的、没有被断言专门排除在外的副作用，我们就绕过了断言。无论我们的设计如何“由模型驱动”，最终我们还是要通过编写过程来产生概念之间相互作用的效果。我们还要花大量时间去编写公式化的代码，那些代码实际上并没有加入什么含义或行为。编写那些代码既枯燥又容易出错，而且它们还会使我们的模型变得含混不清(有些语言可能比其他的要好一些，但是都需要我们做大量的无用功)。本章中提出的释意接口和其他模式会对这种状况有所帮助，但是它们永远无法使传统的面向对象程序获得形式化的严密性。

这些问题都是进行“声明性设计”的内在动机。“声明性设计”这个词对许多人来说都有不同的含义，但通常它表示一种将程序(或程序的某些部分)通过可执行的规格来实现的编程方法。在这种方法下，软件实际上是由精确描述的属性来控制的。声明性设计有不同的形式，在实现上可以使用反射机制，也可以在编译时使用代码生成技术(根据声明来自动产生传统代码)。声明性设计使得其他开发人员可以按代码的字面意思去理解，因为所有声明都得到了绝对的保证。

根据模型中的属性声明来产生运行的程序，是模型驱动设计梦寐以求的目标，但实际上它也有自己的缺陷。例如，下面就是我多次碰到过的两个问题：



- 声明性语言的表达能力不足以支持所有需要，结果变成了一个只能自动化处理部分问题的框架，除此之外软件就非常难以扩充；
- 代码生成技术扰乱了软件开发的迭代周期，没有使用适当的方法把自动生成的代码与手写代码合并起来，结果每次重新生成代码就会造成很大的破坏。

许多声明性设计的尝试无意之中造成了糟糕的后果，导致模型和系统受制于框架的局限性，开发人员也被它束缚住了手脚，不得不采取一些应急手段来完成工作。

基于规则的方法使用推理引擎和规则库进行工作，这是另一种有希望实现声明性设计的方法。遗憾的是，一些微妙的问题可能会破坏这个目标。

虽然基于规则的程序原则上是声明性的，但大多数系统都加入了“控制性谓词”来优化性能。这种控制性代码会引进副作用，因此系统的行为不能够完全由声明的规则来决定了。增减规则或者对规则重新排序都可能会导致意料之外的错误结果。因此，逻辑程序员必须小心地保证代码的效果明显，就象面向对象程序员所做的那样。

许多声明性方法都会被开发人员有意无意地绕过而遭到破坏。难以使用或者限制太多的系统更容易出现这种问题。每个人都必须遵循框架的规则，这样才能获得声明性编程的好处。

当框架的应用范围很窄，被用来专门自动处理一些特别单调、易错的设计方面(如持久性和对象-关系映射)时，声明性设计通常能够表现出最大的价值。好的声明性设计能够减少开发人员的重复工作，同时又不造成任何额外的限制。

领域特定语言

领域特定语言(Domain-Specific Language)是一种有意思的方法，它有时也是声明性的。在这种风格下，编写客户代码的编程语言是针对于某个特定领域的特定模型而剪裁的。例如，运输系统的语言可能会包含类似于 cargo 和 route 这样的词，以及相关的语法。在编译这种程序时，通常会把它转换为传统的面向对象语言，由一个类库用该语言提供那些术语的实现。

在这种语言中，程序可以具有最强的表达力，并与通用语言最紧密结合。这是一个令人兴奋的概念，但是从我所见到的基于面向对象技术的一些方法来看，领域特定语言也存在一些缺点。

为了精化模型，开发人员必须能够对语言进行修改。这可能涉及到修改语法声明和其他语言解释特性，以及修改底层类库。虽然我自己非常喜欢学习高级技术和设计概念，但是我们必须冷静地评估一个特定的开发团队(以及将来的维护团队)所具有的技巧。当



然，用同一种语言来实现模型和应用，能够使它们无缝集成，这一点非常有价值。另一个缺点在于我们很难对客户代码进行重构，使之符合修改后的模型及其相关的领域特定语言。当然，重构问题可以通过技术上的修正来解决。

自底向上

还有一种不同的范型可能比对象要适合于实现领域特定语言。Scheme 编程语言(函数式编程(functional programming)家族的一种典型代表)与声明性设计非常相似，但是又在某种程度上具有标准的编程风格。用这种语言可以构造出领域特定语言的表达能力，同时又无需把系统割裂开来。

当模型非常成熟时，领域特定语言的使用效果可能会最好。此时客户代码也许将由另一个团队来编写。一般而言，这会导致开发团队被分成两个部分：一个负责技术含量高的框架开发，另一个负责技术含量低的应用程序开发。但是如果有必要这样做的话，分裂将给开发团队带来危害。

10.8 一个声明性风格的设计

一旦我们在设计中用到了释意接口、无副作用函数和断言，我们就能逐渐进入声明性的王国了。通过把相互交流的元素组合起来，并使之具有明显的特征或效果(或者根本没有可观测的效果)，我们就可以获得声明性设计所带来的诸多好处了。

柔性设计使得客户代码使用声明风格的设计成为可能。为了演示这一点，我们将在下一节把本章中的一些模式联合起来使用，使得规格更加具有柔性和声明性。

用声明性风格扩充规格

第9章讨论了规格的基本概念，它在程序中可以充当的角色，以及实现这个模式时需要考虑的一些问题。现在让我们来看看一些更高级的特性，在规则复杂的某些情况下，它们可能会非常有用。

规格是由“谓词”这个概念改编而成的。谓词还有一些其他的有用的特性，可供我们有选择地加以利用。

1. 使用逻辑运算符组合规格

在使用规格时，我们很快就会遇到一些情况，需要把多个规格组合起来。如前所述，规格就是一种谓词，而谓词是可以用运算符 AND、OR 和 NOT 来组合和修改的。这些



第Ⅲ部分 面向更深层理解的重构

逻辑运算对于谓词是封闭的，因此规格的组合将表现为一种操作封闭。

随着一些重要的通用能力的加入，创建一个能表示各种规格的抽象类或接口就变得越来越有用了。这意味着把我们必须把变元类型定义为某种高层的抽象类(这里是Object)。

```
public interface Specification {  
    boolean isSatisfiedBy(Object candidate);  
}
```

按照这种抽象调用，在方法开始的地方应该有一个警戒子句，但是如果没有也不影响功能。例如，Container Specification(见第9章的例子)可能会修改成这样：

```
public class ContainerSpecification implements Specification {  
    private ContainerFeature requiredFeature;  
  
    public ContainerSpecification(ContainerFeature required) {  
        requiredFeature = required;  
    }  
  
    boolean isSatisfiedBy(Object candidate) {  
        if (!candidate instanceof Container) return false;  
  
        return  
            (Container)aContainer.getFeatures().contains(requiredFeature);  
    }  
}
```

现在，我们对 Specification 接口进行扩展，加入 3 个新的操作：

```
public interface Specification {  
    boolean isSatisfiedBy(Object candidate);  
  
    Specification and(Specification other);  
    Specification or(Specification other);  
    Specification not();  
}
```

有些 Container Specification 的配置需要 Container 具有通风性，其他的需要防爆性。如果某种化学品既要求通风又要求防爆，那么它应该就需要这两种规格。这个不难用新方法实现：

```
Specification ventilated = new ContainerSpecification(VENTILATED);  
Specification armored = new ContainerSpecification(ARMORED);
```



```
Specification both = ventilated.and(armored);
```

以上声明定义了一个新的具有期望特性的 Specification 对象。这种组合可能需要我们针对各种特殊的情况定义更复杂的 Container Specification。

假设我们有多种通风的 Container。对于某些化学品，打包到什么容器中可能关系不大，随便哪种都行。

```
Specification ventilatedType1 =
    new ContainerSpecification(VENTILATED_TYPE_1);
Specification ventilatedType2 =
    new ContainerSpecification(VENTILATED_TYPE_2);

Specification either = ventilatedType1.or(ventilatedType2);
```

如果觉得把盐放到特殊容器中有些浪费，那么我们可以定义一个规格来为盐指定更加“廉价”（即没有特殊特性）的容器，来防止这种浪费。

```
Specification cheap = (ventilated.not()).and(armored.not());
```

这个约束可以阻止某些不优化的打包行为（像第 9 章中讨论的仓库打包机原型所做的那样）。

这种从简单元素中构造复杂规格的能力提升了代码的表达力。上面的组合就是用声明性风格来编写的。

根据规格实现方法的不同，提供运算符的难度也可能不同。下面是一种非常简单的实现，它在某些情况下可能很高效，在其他一些情况下却十分实用。我们只把它看成一个说明性的例子。和其他模式一样，实现的方法总是多种多样的。

```
public abstract class AbstractSpecification implements
    Specification {
    public Specification and(Specification other) {
        return new AndSpecification(this, other);
    }
    public Specification or(Specification other) {
        return new OrSpecification(this, other);
    }
    public Specification not() {
        return new NotSpecification(this);
    }
}

public class AndSpecification extends AbstractSpecification {
```



第Ⅲ部分 面向更深层理解的重构

```
Specification one;
Specification other;
public AndSpecification(Specification x, Specification y) {
    one = x;
    other = y;
}
public boolean isSatisfiedBy(Object candidate) {
    return one.isSatisfiedBy(candidate) &&
        other.isSatisfiedBy(candidate);
}
public class OrSpecification extends AbstractSpecification {
    Specification one;
    Specification other;
    public OrSpecification(Specification x, Specification y) {
        one = x;
        other = y;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return one.isSatisfiedBy(candidate) ||
            other.isSatisfiedBy(candidate);
    }
}
public class NotSpecification extends AbstractSpecification {
    Specification wrapped;
    public NotSpecification(Specification x) {
        wrapped = x;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return !wrapped.isSatisfiedBy(candidate);
    }
}
```

上面的代码编写得尽可能简单，以方便读者理解。我说过，这种实现的效率在某些情况下可能很低。也许存在其他的实现选择，能将对象的个数降到最低，或者使执行速度加快，或者与项目中的某种特定技术相吻合。这里重要的是我们的模型捕捉到了领域中的关键概念，并提供了一个忠实地反映该模型的实现，如图 10-14 所示。至于性能问题则还有很大的解决空间。

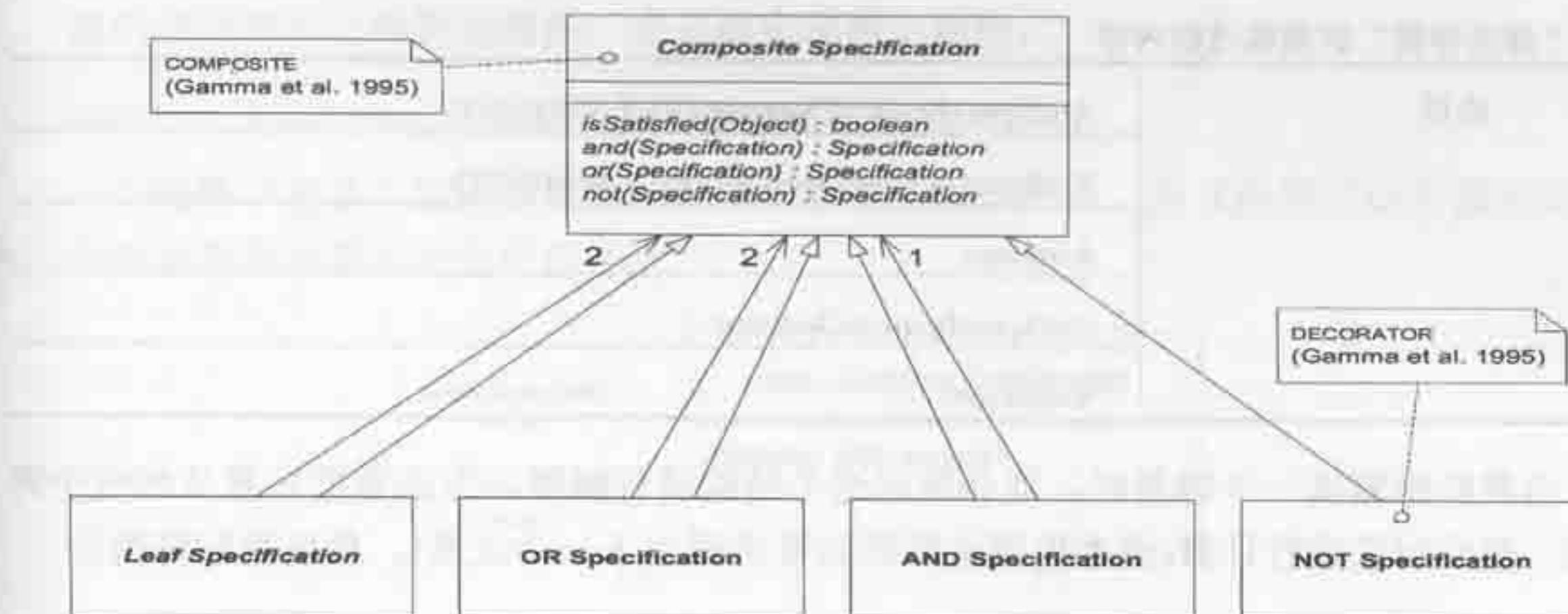


图 10-14 规格的组合设计

此外，在许多情况下并不需要像这样的完全通用性。特别是 AND 比其他逻辑运算往往用得更多一些，而且在实现上也更简单。如果需要的只是 AND，那么不用害怕，仅仅实现 AND 就够了。

回顾一下第 2 章的例子，开发人员显然没有实现其规格的“satisfied by”行为。那时的规格还只是仅仅用来作按需创建。即便如此，抽象依然完整，加入功能也相对容易。使用一个模式并不意味着构造不需要的特性，这些特性可以在以后再添加进来，只要概念没有发生混乱就行。

示例：组合规格的另一种实现

有些实现环境要求对象的粒度不能太小。我曾经碰到过一个项目，在该项目中，我们使用了一个对象数据库，它给每个对象都加上一个 ID，并对这些对象进行跟踪。每个对象都有很大的内存空间和性能开销，而地址空间又是一个限制因素。我在领域设计中的一些重点地方使用了规格模式，并认为那些决定都还不错。但是我使用的实现方法有些过于精细，结果它产生了成千上万个粒度非常小的对象，把整个系统都拖跨了。因此，那种实现方法是个绝对错误的选择。

下面的例子给出了另一种实现，它把组合规格编码为一个字符串，或者说把逻辑表达式编码为一个数组，然后在运行时对它进行解释。

(如果您不能理解为什么要这样实现，也不必担心。重要的是，您得知道用逻辑运算符实现规格的方法有许多种，因此当简单的实现满足不了您的需要时，您还有其他的选择。)



“廉价容器”的规格栈的内容

栈顶	AndSpecificationOperator (FLY WEIGHT)
	NotSpecificationOperator (FLY WEIGHT)
	Armored
	NotSpecificationOperator
	Ventilated

当我们想测试一个容器时，就必须对这个结构进行解释，方法是把元素从堆栈中弹出来，然后对它进行计算(或者根据运算符的要求弹出下一个元素)。最后我们将得到：

```
and(not(armored),not(ventilated))
```

这个设计有一些优点和缺点：

优点：对象个数少

优点：内存使用效率高

缺点：需要水平更高的开发人员

您必须根据具体情况来进行权衡，找出满足要求的实现。根据同一个模式和模型可以构造出非常不同的实现来。

2. 包含

包含(subsumption)这个特性有时是并不需要的，也很难实现，但是它往往能解决一种相当困难的问题。它还能阐明一个规格的含义。

重新考虑一下化学品仓库打包机的例子。每个 Chemical 都有一个 Container Specification，而 Packer 服务保证当把 Drum 指派到 Container 时，所有这些规格都能被满足。一切正常——直到有人改变了规则。

由于每隔几个月都会发布一些新的规则(指容器规格)，因此我们的用户希望能够生成一个列表，把那些存储要求变得更加严格的化学品类型列举出来。

当然，我们可以用新的规格对存货清单中的每种 Drum 进行验证，找出所有不再满足规格的产品，从而得到一个不完整的列表(它可能也是用户想要的)。这个结果可以告诉用户，按当前的存货清单他们需要移动哪些 Drum。

但是，用户要求的是把那些存储要求变得比原来更严格的化学品列举出来。也许有些化学品尚未到货，也许它们正好被打包到了一个更严格的容器中。在这两种情况下，我们刚才提到的报表就不会把它们列出来了。



我们可以引入一种新的操作，来直接比较两个规格：

```
boolean subsumes(Specification other);
```

一个规格“包含”比它更宽松的规格，如图 10-15 所示。包含操作可以直接添加进来，不会对前面的需求产生任何影响。

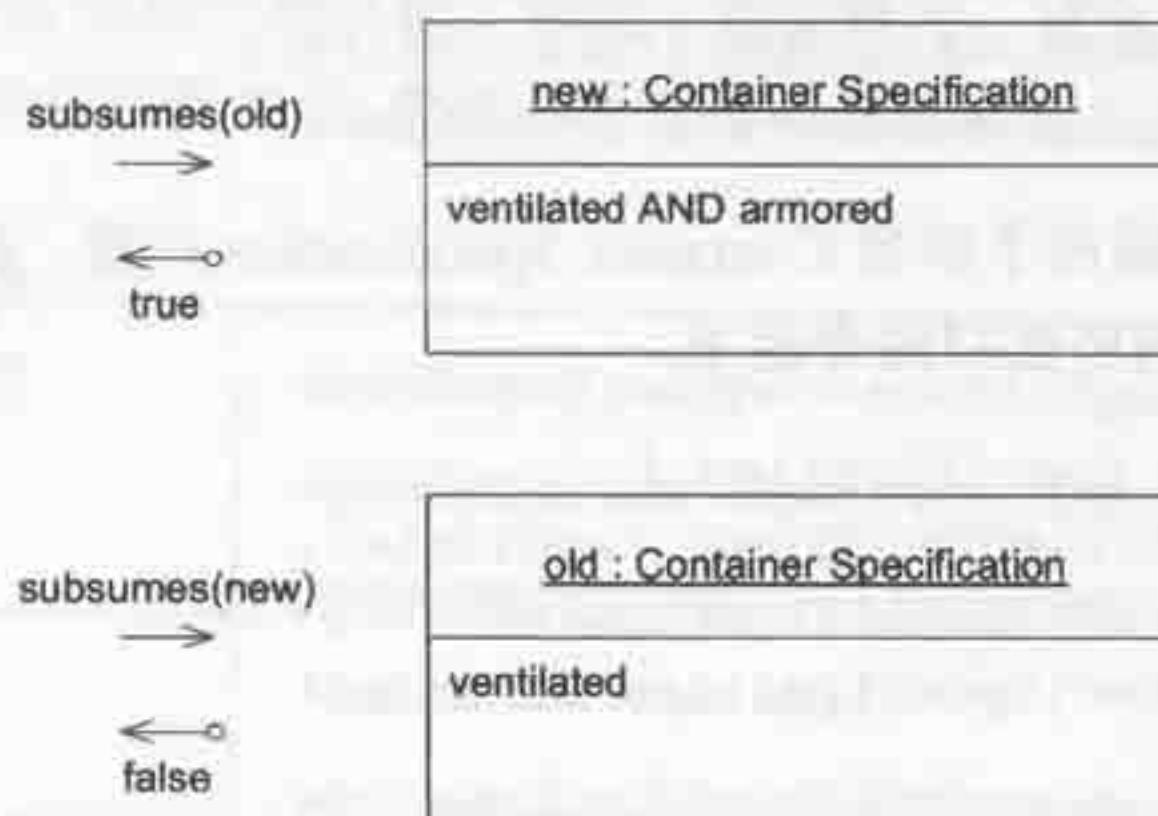


图 10-15 汽油容器的规格得到了加强

用规格的语言来说，我们可以说新规格包含了老规格，因为任何满足新规格的容器都可以满足老规格。

如果把每个这些规格都视为一个谓词，那么包含就等价于逻辑蕴涵。用习惯的表示法， $A \rightarrow B$ 意味着声明 A 蕴涵声明 B，因此若 A 为真，则 B 也为真。

我们可以用这种逻辑操作来解决容器匹配的问题。当一个规格发生变化时，我们希望知道新提出的规格是否能满足原规格的所有条件。

New Spec \rightarrow Old Spec

也就是说，如果新规格为真，那么老规格也为真。用通用的方法来证明逻辑蕴涵是非常困难的，但是在特殊情况下就非常简单。例如，特定的参数化的规格可以定义它们自己的包含规则。

```
public class MinimumAgeSpecification {
    int threshold;

    public boolean isSatisfiedBy(Person candidate) {
        return candidate.getAge() >= threshold;
    }

    public boolean subsumes(MinimumAgeSpecification other) {
```



```
        return threshold >= other.getThreshold();
    }
}
```

在 JUnit 测试中可以包含下面的代码：

```
drivingAge = new MinimumAgeSpecification(16);
votingAge = new MinimumAgeSpecification(18);
assertTrue(votingAge.subsumes(drivingAge));
```

还有一种特殊情况适用于解决 Container Specification 问题，那就是把包含与仅有一个逻辑运算符 AND 的规格接口结合起来。

```
public interface Specification {
    boolean isSatisfiedBy(Object candidate);
    Specification and(Specification other);
    boolean subsumes(Specification other);
}
```

证明只包含 AND 运算符的蕴涵非常简单：

$$A \text{ AND } B \rightarrow A$$

或者在更复杂的情况下：

$$A \text{ AND } B \text{ AND } C \rightarrow A \text{ AND } B$$

因此，如果 Composite Specification 能够把所有用 AND 连接起来的叶子(leaf)规格收集起来的话，我们就只需检查新规格是否含有被蕴涵规格所含的所有叶子(当然也可以包含更多叶子)——也就是说，其所含的叶子是另一个规格所含叶子的超集。

```
public boolean subsumes(Specification other) {
    if (other instanceof CompositeSpecification) {
        Collection otherLeaves =
            (CompositeSpecification) other.leafSpecifications();
        Iterator it = otherLeaves.iterator();
        while (it.hasNext()) {
            if (!leafSpecifications().contains(it.next()))
                return false;
        }
    } else {
        if (!leafSpecifications().contains(other))
            return false;
    }
}
```



```
return true;
}
```

我们还可以改进这种交互，精心选择参数化的叶子规格来进行比较(或者利用其他更复杂的特殊情况)。遗憾的是，当包括 OR 和 NOT 运算时，蕴涵的证明将会复杂得多。在大多数情况下，最好能作一些权衡，放弃一些运算符，或者放弃蕴涵操作，以便避免这种复杂性。如果二者都需要的话，那就必须慎重考虑实现这么困难的功能到底值不值得。

规格下的苏格拉底

所有人都是要死的	Specification manSpec = new ManSpecification(); Specification mortalSpec = new MortalSpecification(); assert manSpec.subsumes(mortalSpec);
苏格拉底是人	Man socrates = new Man(); assert manSpec.isSatisfiedBy(socrates);
所以苏格拉底是要死的	assert mortalSpec.isSatisfiedBy(socrates);

10.9 攻击角度

本章给出了多种技术来澄清代码的意图、阐明其执行结果，并对模型元素进行解耦。即使有这些技术，柔性设计还是非常困难的。我们不能看着一个巨大的系统，然后说“让我们来让它具有柔性”。我们必须选择目标。下面将介绍几种具有一般性的方法，然后给出一个扩充的例子来演示如何把这些模式组合起来使用，以应付更大的设计。

10.9.1 切分子领域

我们不能一次就把整个设计问题解决掉，而只能各个击破。如果您对系统的某些方面有了解决的线索，那么就把它分离出来单独研究。如果您发现模型的某个部分可以被视为某种专门数学，也分离之。如果您的应用需要用一些复杂的规则来约束状态改变，那么把它们提取到一个单独的模型中，或者实现一个简单的框架，然后在这个框架中把那些规则声明出来。这些步骤不仅能使我们获得清晰的新模型，也使得旧模型只留下一些更小、更清晰的部分。旧模型的剩余部分将具有声明性的风格——这种声明性可以借助专门数学、验证框架，或者子领域所提供的任何其他形式来实现。

大刀阔斧地对领域进行切分、使设计的一部分真正获得柔性，比分散精力四面出击要有用得多。第 15 章更深入地讨论了如何选择和管理子领域的问题。

10.9.2 尽可能利用现成的形式

从头开始创建一个严密的概念框架很不容易，不是每天都能做到的。在项目的开发进程中，我们有时需要发现并精化一个这样的模型。但是，我们常常可以借用和改造现成的概念系统。这些系统在我们的领域或者其他项目之中可能早已创建好了，有的甚至已经精化提炼几个世纪了。例如，许多商业应用中都涉及到会计学，会计学中定义了一系列非常成熟的实体和规则，可以非常容易地融合到深层模型和柔性设计中。

像这样的形式化概念框架有许多种，但我个人特别喜欢的一种概念框架是数学。数学的用处令人称奇，它用基本的算术就能把某些错综复杂的概念提取出来。许多领域都在某些地方包含数学。寻找它，挖掘它。专门数学非常清晰，能用清晰的规则进行组合，也很容易理解。一个例子就是前面说过的“股份数学”，我们将用它来结束本章的讨论。

示例：模式的整合：股份数学

第8章我们讲述了一个模型突破的故事，那是一个构建银团贷款系统的项目。现在我们探讨这个例子的细节，只关注设计的一个特征，并与原来的设计进行对比。

在那个应用中，有一个需求是当借款人偿付本金时，本金在默认情况下是按照放贷人的贷款股份来分配的。

1. 付款分配的最初设计

随着重构的进行，下面的代码会更容易理解，因此不要被图 10-16 所示的这个版本迷惑了。

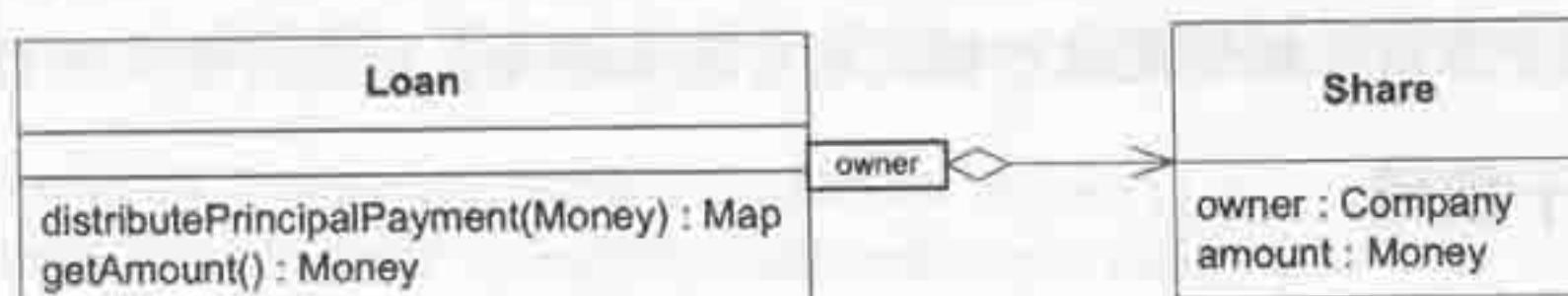


图 10-16 最初模型

```
public class Loan {
    private Map shares;

    //Accessors, constructors, and very simple methods are excluded

    public Map distributePrincipalPayment(double paymentAmount) {
        Map paymentShares = new HashMap();
        Map loanShares = getShares();
        double total = getAmount();
```



```

Iterator it = loanShares.keySet().iterator();
while(it.hasNext()) {
    Object owner = it.next();
    double initialLoanShareAmount = getShareAmount(owner);
    double paymentShareAmount =
        initialLoanShareAmount / total * paymentAmount;
    Share paymentShare =
        new Share(owner, paymentShareAmount);
    paymentShares.put(owner, paymentShare);

    double newLoanShareAmount =
        initialLoanShareAmount - paymentShareAmount;
    Share newLoanShare =
        new Share(owner, newLoanShareAmount);
    loanShares.put(owner, newLoanShare);
}
return paymentShares;
}

public double getAmount() {
    Map loanShares = getShares();
    double total = 0.0;
    Iterator it = loanShares.keySet().iterator();
    while(it.hasNext()) {
        Share loanShare = (Share) loanShares.get(it.next());
        total = total + loanShare.getAmount();
    }
    return total;
}
}

```

2. 分离命令和无副作用函数

这个设计已经具有了释意接口。但是 `distributePaymentPrincipal()` 方法做了一件危险的事情：它除了计算分配的股份，还修改了 `Loan`。让我们把这个查询从修改器中分离出来，如图 10-17 所示。

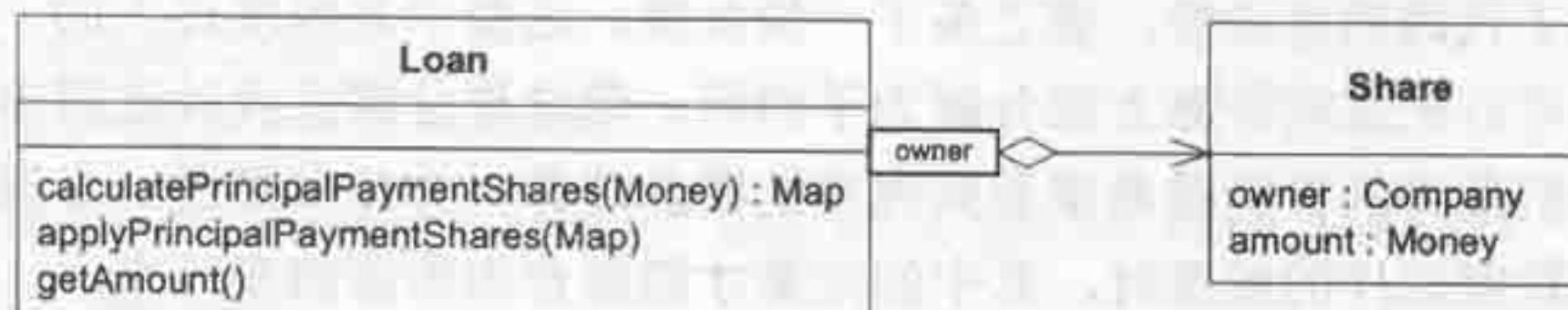


图 10-17 分离查询



```
public void applyPrincipalPaymentShares(Map paymentShares) {
    Map loanShares = getShares();
    Iterator it = paymentShares.keySet().iterator();
    while(it.hasNext()) {
        Object lender = it.next();
        Share paymentShare = (Share) paymentShares.get(lender);
        Share loanShare = (Share) loanShares.get(lender);
        double newLoanShareAmount = loanShare.getAmount() -
            paymentShare.getAmount();
        Share newLoanShare = new Share(lender, newLoanShareAmount);
        loanShares.put(lender, newLoanShare);
    }
}

public Map calculatePrincipalPaymentShares(double paymentAmount) {
    Map paymentShares = new HashMap();
    Map loanShares = getShares();
    double total = getAmount();
    Iterator it = loanShares.keySet().iterator();
    while(it.hasNext()) {
        Object lender = it.next();
        Share loanShare = (Share) loanShares.get(lender);
        double paymentShareAmount =
            loanShare.getAmount() / total * paymentAmount;
        Share paymentShare = new Share(lender, paymentShareAmount);
        paymentShares.put(lender, paymentShare);
    }
    return paymentShares;
}
```

客户代码现在看起来像这样：

```
Map distribution =
    aLoan.calculatePrincipalPaymentShares(paymentAmount);
aLoan.applyPrincipalPaymentShares(distribution);
```

还好，我们已经用函数封装了释意接口后面的许多复杂性。但是随着我们加入`applyDrawdown()`、`calculateFeePaymentShares()`等这些方法，代码就开始成倍增长了。每次扩充都增加了代码的复杂性，使之多了一份负担。这似乎是粒度过大的一种信号。按照习惯，我们可以把这些计算方法分解为子例程。像这样分解在设计过程中可能是一步好棋，但是我们最终的目的是希望看到内在的概念边界，并使模型进一步深化。只有当设计达到概念轮廓这样的粒度时，其中的元素才能组合出所需的变体来。



3. 把隐式概念变成显式

现在我们有足够的线索来寻求新的模型了。Share 对象在这个实现中是被动的，用复杂、低层次的方法操纵。这是因为与股份有关的大部分规则和计算都不是应用在单个股份，而是一组股份。这实际上就漏掉了一个概念：股份之间是有关联的，继而构成整体。把这个概念变成显式，使我们能够更加简洁地表达这些规则和计算，如图 10-18 所示。

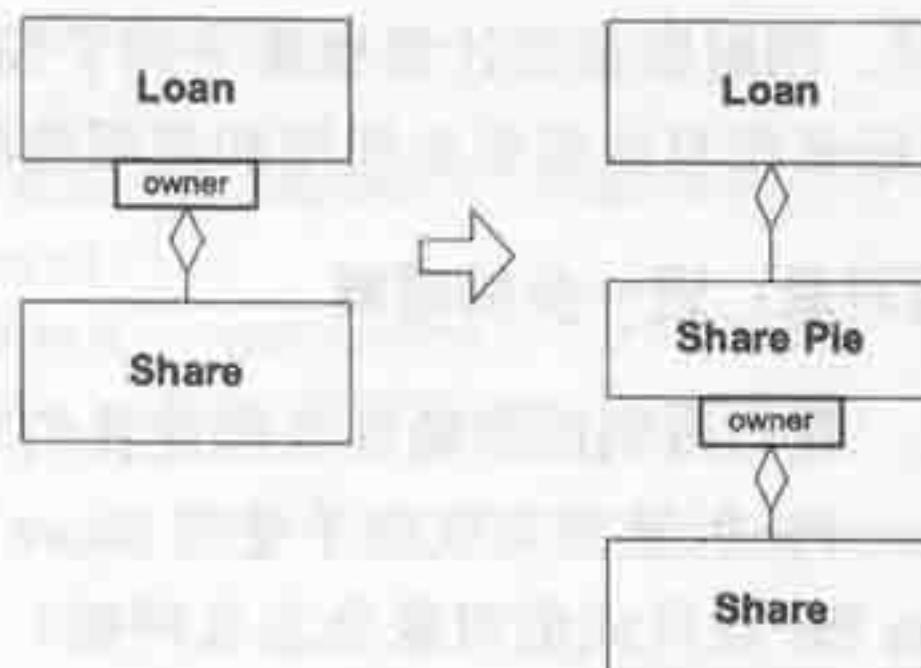


图 10-18 股份间的关联

Share Pie 描述了一个特定 Loan 上总的股份分布情况。它是一个实体，其标识在 Loan 的聚合中是局部性的。实际的分配计算可以委托给 Share Pie 来执行，如图 10-19 所示。

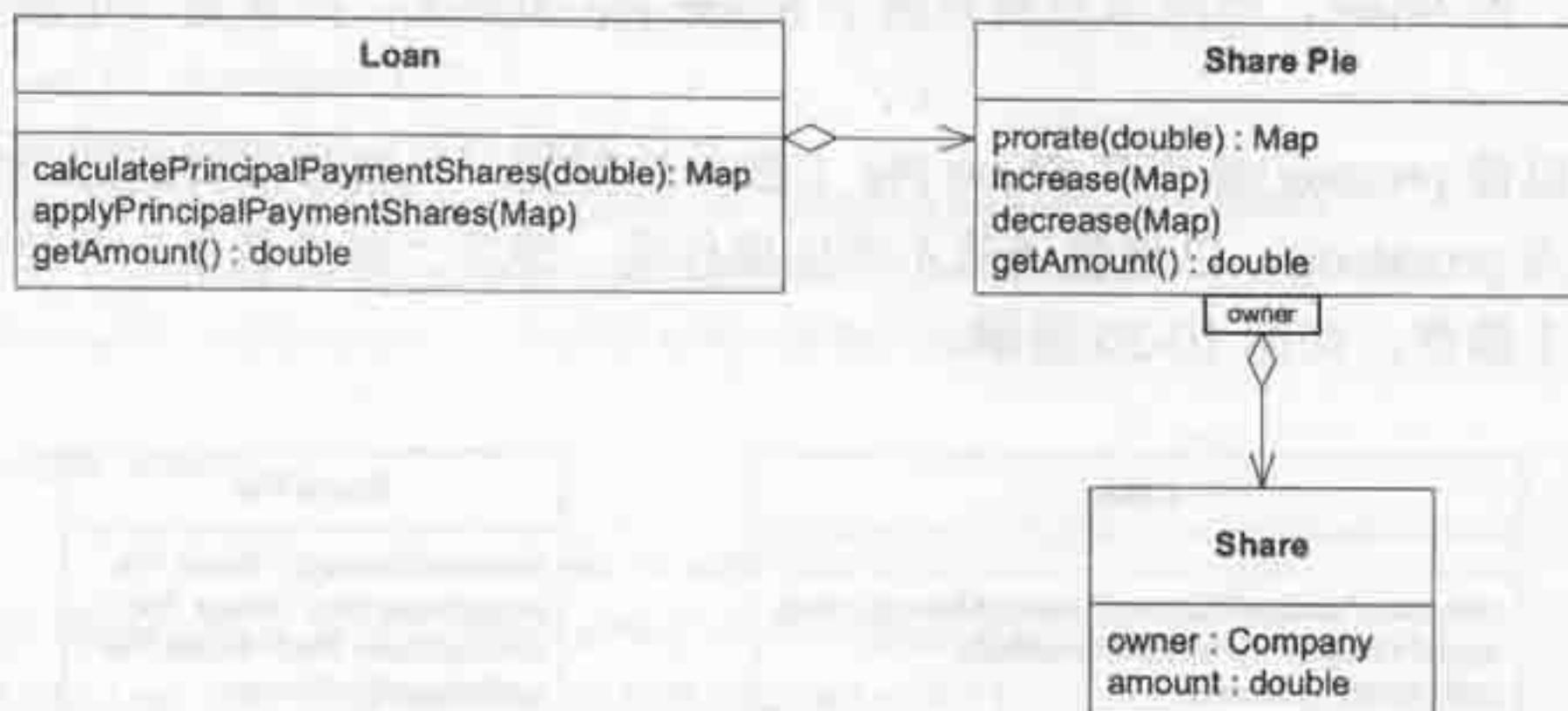


图 10-19 Share Pie 实体

```

public class Loan {
    private SharePie shares;

    //Accessors, constructors, and straightforward methods
    //are omitted

    public Map calculatePrincipalPaymentDistribution()
  
```



```
        double paymentAmount) {  
            return getShares().prorated(paymentAmount);  
        }  
        public void applyPrincipalPayment(Map<String, Double> paymentShares) {  
            shares.decrease(paymentShares);  
        }  
    }  
}
```

这样 Loan 类得到了简化，同时股份的计算也集中到了 Share 中，由这个值对象来专门担负其职责。但是，算法还是没有达到更加通用和易用的程度。

4. 把 Share Pie 变成值对象：进一步的理解

在实现一个新的设计时，我们得到的经验往往能使我们对模型本身获得新的理解。在这个例子中，Loan 和 Share Pie 的紧密关联似乎使得 Share Pie 和 Share 之间的关系变得模糊了。如果我们把 Share Pie 设计成值对象会怎么样呢？

这就意味着 increase(Map) 和 decrease(Map) 这两个方法不能再使用了，因为 Share Pie 必须具有不变性。为了修改 Share Pie 的值，整个对象都应该被替换掉。因此我们可以使用像 addShares(Map) 这样的操作来返回一个全新的、更大的 Share Pie 对象。

让我们再接再厉，把操作封闭也考虑进来。我们不应该说“增加” Share Pie 或者向其中加入一些 Share，而应该直接把两个 Share Pie 加起来：结果是一个新的、更大的 Share Pie。

我们可以使 prorate() 操作在 Share Pie 上部分地封闭，只要修改其返回类型即可。把它重新命名为 prorated()，以便强调其不产生副作用。现在“股份数学”开始现形了，最开始它有 4 个操作，如图 10-20 所示。

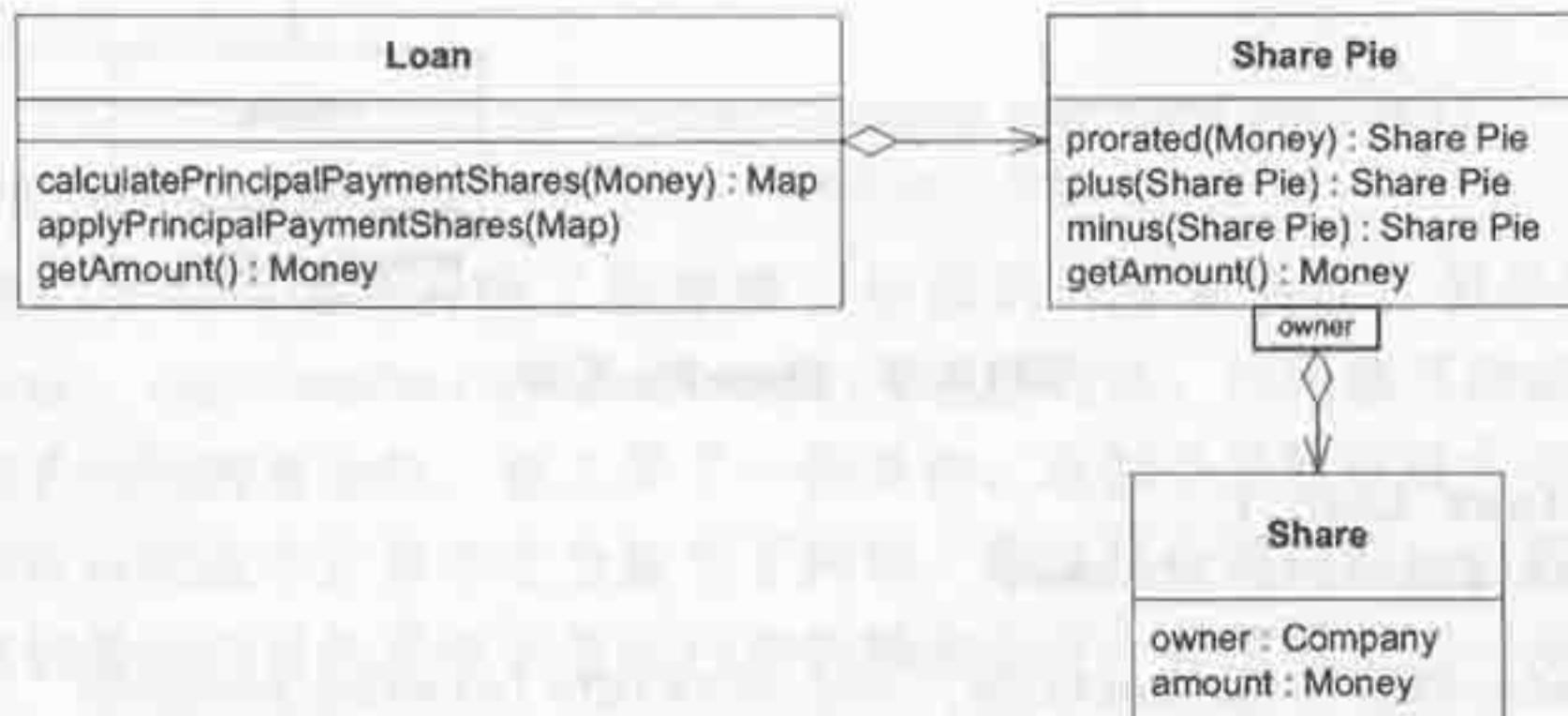


图 10-20 将 Share Pie 修改成值对象



我们可以为新的值对象 Share Pie 创建一些定义良好的断言。每个方法都具有一些含义。

```
public class SharePie {  
    private Map<Object, Share> shares = new HashMap<>();  
  
    //Accessors and other straightforward methods are omitted  
  
    public double getAmount() {  
        double total = 0.0;  
        Iterator<Object> it = shares.keySet().iterator();  
        while(it.hasNext()) {  
            Share loanShare = getShare(it.next());  
            total = total + loanShare.getAmount();  
        }  
        return total;  
    }  
  
    public SharePie minus(SharePie otherShares) {  
        SharePie result = new SharePie();  
        Set<Object> owners = new HashSet<>();  
        owners.addAll(getOwners());  
        owners.addAll(otherShares.getOwners());  
        Iterator<Object> it = owners.iterator();  
        while(it.hasNext()) {  
            Object owner = it.next();  
            double resultShareAmount = getShareAmount(owner) -  
                otherShares.getShareAmount(owner);  
            result.add(owner, resultShareAmount);  
        }  
        return result;  
    }  
  
    public SharePie plus(SharePie otherShares) {  
        //Similar to implementation of minus()  
    }  
  
    public SharePie prorated(double amountToProrate) {  
        SharePie proration = new SharePie();  
        double basis = getAmount();  
        Iterator<Object> it = shares.keySet().iterator();  
        while(it.hasNext()) {  
            Object owner = it.next();  
            Share share = getShare(owner);  
            double shareAmount = share.getAmount();  
            double proratedShareAmount = shareAmount * amountToProrate / basis;  
            proration.add(owner, proratedShareAmount);  
        }  
        return proration;  
    }  
}
```



```
        double proratedShareAmount =
            share.getAmount() / basis * amountToProrate;
        proration.add(owner, proratedShareAmount);
    }
    return proration;
}
}
```

5. 新设计的柔性

至此，最重要的 Loan 类中的方法就相当简单了，如下所示：

```
public class Loan {
    private SharePie shares;

    //Accessors, constructors, and straightforward methods
    //are omitted

    public SharePie calculatePrincipalPaymentDistribution(
        double paymentAmount) {
        return shares.prorated(paymentAmount);
    }

    public void applyPrincipalPayment(SharePie paymentShares) {
        setShares(shares.minus(paymentShares));
    }
}
```

这些简短的方法每个都有其含义。偿付本金意味着按照股份分别从贷款中减去付款额，本金按照各个股东所占股份的比例进行分配。Share Pie 的设计允许我们用一种声明性的风格来编写 Loan，使其代码读起来就像是一个业务交易的概念定义，而不是一种计算。

其他交易类型(以前由于太复杂而没有列出来)现在也可以很容易地声明出来了。例如，支取贷款是按照融通股份(在 Facility 中所占的股份)来分配的。新支取出来的贷款加入未偿贷款中。用我们的新领域语言可以这样描述：

```
public class Facility {
    private SharePie shares;
    . .
    public SharePie calculateDrawdownDefaultDistribution(
        double drawdownAmount) {
        return shares.prorated(drawdownAmount);
    }
}
```



```
public class Loan {  
    . . .  
    public void applyDrawdown(SharePie drawdownShares) {  
        setShares(shares.plus(drawdownShares));  
    }  
}
```

要查看每个借款人所占股份与其协定股份的差额，我们只需先计算出未偿贷款的理论分配值，然后从贷款的实际股份中减去这个值即可：

```
SharePie originalAgreement =  
    aFacility.getShares().prorated(aLoan.getAmount());  
SharePie actual = aLoan.getShares();  
SharePie deviation = actual.minus(originalAgreement);
```

Share Pie 的设计具有一些很好的特性，这使得代码的重新组合非常容易，同时表达也更加清楚。

- 复杂的逻辑通过无副作用函数封装到了特化的值对象中。大部分复杂的逻辑都封装在这些具有不变性的对象中。由于 Share Pie 是值对象，因此其数学运算能够产生新的实例，这样我们就可以用新实例直接将过时的实例替换掉。
- Share Pie 的所有方法都不会导致任何已有对象发生任何改变。这使得我们可以在计算过程中自由地使用 plus()、minus() 和 prorated() 方法，对它们进行组合，期望它们产生名字所说明的效果，而且仅仅只产生这些效果。我们还可以通过这些方法来实现分析特性(以前那些方法只能在真正进行分配的时候才能调用，因为每次调用之后数据就改变了)。
- 状态修改操作非常简单而且可以用断言来刻画。股份数学的高度抽象性使得我们可以用声明性的风格将交易中的不变量精确地编写出来。例如，差额是用实际的股份额减去按融通股份分配的贷款额得到的。
- 模型概念被解耦：操作中涉及的其他类型也达到最少。Share Pie 中的某些方法表现出操作封闭(加减方法在 Share Pie 下是封闭的)。其他方法用简单的总额作为变元或者返回值，虽然不是封闭的，但是只增加很少的概念负荷。Share Pie 只与一个类(Share)紧密交互，因此它是自包含的，易于理解、易于测试，也易于与其他类组合将业务交易声明性地描述出来。这些特性都是从数学形式中获得的。
- 熟悉的形式使得协议更易于把握。我们可以根据金融术语创造一种全新的操纵股份的协议，原则上它的设计也可能会具有柔性。但是这种方法有两个不足。首先，



第Ⅲ部分 面向更深层理解的重构

我们必须发明这种协议——这是个困难而且不确定的任务。其次，使用这种协议的每个人都必须学习它。人们看到股份数学时，会认为这是一个自己早就知道了的系统。同时这个设计还小心地与算术规则保持着一致，因此也不会造成误导。

在上面，我们将问题中具有数学形式的部分提取出来，获得了一个 Share 的柔性设计，并进一步对核心的 Loan 和 Facility 的方法进行了精炼(见第 15 章对 Core Domain 的讨论)。

柔性设计对于软件应付改变和复杂性的能力具有深远的影响。正如本章的例子所示，这种能力往往是以相当细节的建模和设计决定为转移的。柔性设计的影响可以超出某个特定的建模和设计问题。第 15 章将讨论柔性设计的战略价值，我们将把柔性设计作为一种精炼领域模型、使大型的复杂项目更加易于驾驭的工具。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。

如果从一个更高的角度看，我们发现，通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。通过将数学形式从设计中分离出来，我们已经大大降低了设计的复杂性。我们不再需要处理数学公理，而是处理一个相对简单的协议，这个协议是完全可验证的，而且是完全可预测的。