第26章

扩展型模式介绍

在使用 Java 编程时，你并不是从零开始，而是“继承”了 Java 类库的所有功能。通常，还可以使用同事和前人写好的代码。只要你不是在重新组织或改进遗留代码，你就是在对它进行扩展。可以认为：Java 编程本身就是一种扩展。

如果你曾经继承过一个代码库，或许你会抱怨糟糕的代码质量。然而，你添加的新代码质量就一定好吗？答案通常是主观的。不过，本章介绍的面向对象软件开发的几个原则，却可以用来评估你的工作。

除了扩展代码库的一些常见技术外，还可以使用设计模式添加新的行为。在了解了基本的面向对象的设计原则后，本章会回顾前面提到的具有扩展功能的模式，然后介绍其他面向扩展的模式。

面向对象设计的原则

石桥的存在已有悠悠数千年的历史，为了设计它们，我们拥有太多久经考验并得到一致认可的原则。面向对象编程的诞生不过才 50 年左右，因此它的设计无法达到石桥的设计水平，也就不足为奇了。然而，我们已经拥有了许多讨论设计原则的优秀论坛。其中，www.c2.com 的 Portland Pattern Repository 就是其中之一。访问该网站，你会发现在评估面向对象的设计方面，一些原则业已体现了自身价值。在设计时，需要考虑的其中一条原则是 Liskov 替换原则（LSP）。

Liskov 替换原则

子类应该在逻辑性上与其超类保持一致，但是什么才是逻辑性（logical）与一致性（consistent）呢？Java 编译器将会保证一定级别的一致性，但是，任何一个一致性的原则又都会绕过编译器。Liskov 替换原则（由 Liskov 在 1987 年提出）能够帮助我们改善设计，它的主要思想是：一个类的实例应该具有其父类的所有功能。

诸如 Java 这样的面向对象设计语言，已经满足基本的 LSP。例如，如果 `UnloadBuffer` 类是 `Machine` 类的子类，就可以将 `UnloadBuffer` 对象当做 `Machine` 对象来使用：

```
Machine m = new UnloadBuffer(3501);
```

LSP 的某些方面还需要人的参与，因为今天的编译器显得还不够智能。考虑图 26.1 所示的类层次关系。

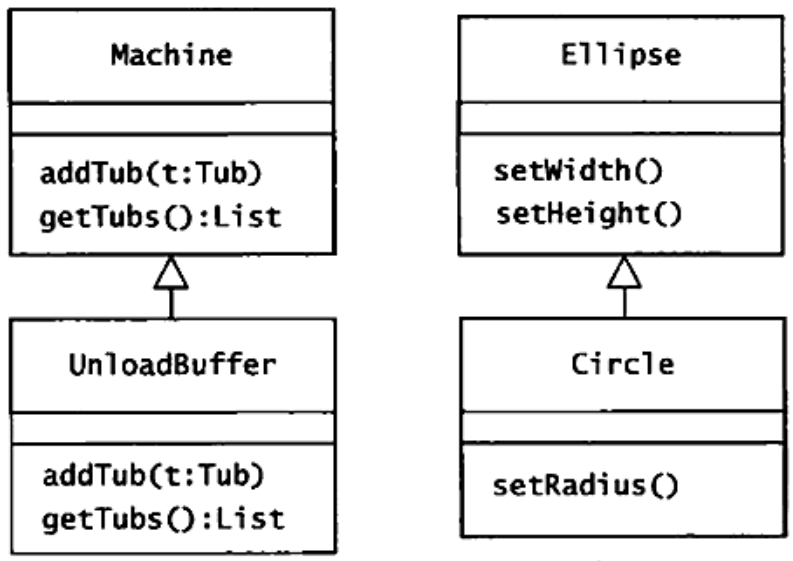

图 26.1　这幅图有两个问题：卸载缓冲池是机器吗？圆是一个椭圆吗？

卸载缓冲池当然是一个机器，但是在类层次中这样建模就会有问题。对于 Oozinoz 公司的机器来说，除了卸载缓冲区，其他的机器都可以接收一桶原料。几乎每台机器都可以接收原料桶，并将其拥有的原料桶集合，报告给附近的机器。将此行为迁移到 `Machine` 类中是很有价值的。但是，如果调用 `UnloadBuffer` 对象的 `addTub()`或者 `getTubs()`方法，就会出现错误。倘若出现了错误，我们在调用方法时是否应该抛出异常呢？

假设其他的开发者编写了一个方法，用来查询车间里所有机器拥有的原料桶的集合。一旦访问到卸载缓冲池，如果 UnloadBuffer 类的 getTubs()方法抛出异常，这段代码就会出现问题。这严重违反了 LSP：当你将 UnloadBuffer 对象当做 Machine 对象来使用时，程序可能会崩溃！假设不抛出异常，我们需要简单地忽略对 UnloadBuffer 类中 getTubs()和 addTub()的调用。这一做法依然违反了 LSP，因为在你给机器添加一个原料桶时，这个原料桶可能会消失！

违反 LSP 并不一定是设计缺陷。针对 Oozinoz 公司的这种情况，需要权衡一下，让 Machine 类拥有大多数机器的行为和违反 LSP 原则，究竟哪个价值更大。重要的一点是意识到 LSP，并且清楚为什么其他设计可能违反了 LSP。

挑战 26.1

圆当然是椭圆的一种特殊形态，不是吗？请说明图 26.1 中 Ellipse 类和 Circle 类的关系是否违背了 LSP。

答案参见第 358 页

迪米特法则

在 20 世纪 80 年代后期，美国东北大学 Demeter Project 的成员尝试编辑了一些规则，用于确保健康的面向对象编程。项目团队将这些规则称为迪米特法则（Law of Demeter，LoD）。Karl Lieberherr 和 Ian Holland 在 *Assuring Good Style for Object-Oriented Programs* 一文中全面地总结了这一规则。声明认为：非正式地说，迪米特法则要求每个方法只能给有限的对象发送消息，包括参数对象、[this]伪变量，以及[this]的直接子部分。文章随后给出了该法则的正式定义。相对于完全理解迪米特法则的意图，识别设计是否违反该法则更加容易。

假设你有一个 MaterialManager 对象，该对象有一个方法接收一个 Tub 对象作为参数。Tub 对象有一个 Location 属性，该属性返回一个 Machine 对象，用以表示桶被放在哪个位置。假设在 MaterialManager 的方法中，想要知道机器是否可用，你可能会写如下代码：

```
if (tub.getLocation().isUp()) {
    //...
}
```

这段代码违反了迪米特法则，因为它调用了一个方法，将消息发送给 tub.getLocation()。而 tub.getLocation()对象不是一个参数，不是 this——MaterialManager 对象正在执行的方法，也不是 this 的一个属性。

挑战 26.2

请解释为何 tub.getLocation().isUp()表达式不合适？

答案参见第 358 页

如果这个挑战仅仅让你觉得形如 a.b.c 的表达式是错误的，那就降低了迪米特法则的价值。事实上，Lieberherr 和 Holland 希望迪米特法则能够进一步确定无误地回答这样的问题：是否可以遵循某种公式或法则来写出更好的面向对象程序？这篇阐释迪米特法则的早期论文非常值得我们拜读。就像 Liskov 替换原则一样，如果知道并遵循这些规则，它就会帮助你写出更好的代码，一旦你的设计违背了这些原则，就能够及时获知。

你会发现，遵循这些指导原则，扩展性就能够自然而然产生好的代码。但是，对于很多开发者而言，面向对象的开发依然是一门艺术。对代码库的艺术扩展源于艺术家们的实践，而这些大师们依然在不断地改进他们的艺术。重构就是诸多技艺中的一种工具，它可以在不改变既有功能的前提下，改善代码的质量。

消除代码的坏味道

你可能寄希望于 Liskov 替换原则与迪米特法则，能够永远地防止你写出拙劣的代码。不过，更实际的做法是运用这些准则来帮助发现代码的坏味道，然后修复它。这是一种通用的实践：先写出可工作的代码，然后找出代码的问题，并且修复它，以提升代码质量。但是该如何准确地识别问题呢？答案就是找到代码的坏味道。在 *Refactoring:Improving the Design of Existing Code*（由 Fowler 等人在 1999 年编写）一书中描述了 22 种坏味道，并给出了相应的重构手法。

本书在运用模式时，多次使用重构来重新组织和改善已有的代码。但是在重构时，并不需要总是运用设计模式。任何时候，只要发现代码的坏味道，就值得去重构。

挑战 26.3

请提供一个有坏味道的方法，它需改进质量，改进时不能违反 LSP 或 LoD。

答案参见第 358 页

超越常规的扩展

很多设计模式，包括本书已经讲到的很多模式，都与扩展行为有关。面向扩展的模式通常涉及两种开发角色。例如在适配器模式中，一个开发者通过接口为对象提供服务，而另一个开发者则使用该服务。

挑战 26.4

请填充表 26.1 的空白处，它们都使用了设计模式来扩展类或者对象的行为。

表 26.1 使用设计模式扩展类或对象的行为

例　子	使用的模式
焰火模拟器的开发者创建了一个接口，该接口定义了对象为了参与模拟所必须处理的行为	适配器模式
一种工具，可以让你在运行时组合可执行的对象	?
?	模板方法模式
?	命令模式
一个代码生成器，通过插入行为提供一种假象，使得运行在其他机器上的对象就像在本地执行一样	?
?	观察者模式
该设计允许定义一个抽象的操作，该操作取决于一个定义良好的接口，并且允许添加新的驱动器实现该接口	?

答案参见第 359 页

除了已经介绍的模式，还剩下三个模式主要用于扩展，如表 26.2 所示。

表 26.2　用于扩展的模式

如果你的意图是	使用该模式
让开发者动态组合对象的行为	装饰器模式
提供一个方法来顺序访问集合中的元素	迭代器模式
允许开发者定义一个新的操作，而无须改变分层体系中的类	访问者模式

小结

编写代码的主要目的是扩展新的功能，这需要重新组织代码，改善代码的质量。世上并不存在完整而客观评估代码质量的方法，但是，面向对象的一些好的设计原则业已提出。

Liskov 替换原则要求子类的实例应该具有父类的全部功能。你应该能够识别和评估违反该原则的代码。迪米特法则是一组规则，可以降低类之间的依赖，使代码变得更加整洁。

Martin Fowler 等人在 1999 年归纳了代码坏味道的一组例子。每种坏味道都可以通过重构来消除，一些坏味道可以重构为设计模式。很多设计模式都可以来用来标识、简化或者帮助扩展。