第 19 章 软件发展趋势

为了说明本书中讨论的原则，本章考虑了过去几十年来在软件开发中流行的几种趋势和模式。对于每种趋势，我将描述该趋势与本书中的原则之间的关系，并使用这些原则来评估该趋势是否提供了对抗软件复杂性的手段。

19.1 面向对象编程和继承

在过去的三四十年间，面向对象编程是软件开发中最重要的新思想之一。它引入了诸如类、继承、私有方法和实例变量之类的概念。如果谨慎使用，这些机制可以帮助产生更好的软件设计。例如，私有方法和变量可用于确保信息隐藏：类外部的任何代码都不能调用私有方法或访问私有变量，所以没有任何对它们的外部依赖。

面向对象编程的关键要素之一是继承。继承有两种形式，它们对软件复杂性有不同的影响。继承的第一种形式是接口继承，其中父类定义一个或多个方法的签名，但不实现这些方法。每个子类都必须实现签名，但是不同的子类可以以不同的方式实现相同的方法。例如，某个接口可能定义用于执行 I/O 的方法。一个子类可能对磁盘文件实现 I/O 操作，而另一个子类可能对网络套接字实现相同的操作。

接口继承通过将同一接口用于多种用途，从而提供了对抗复杂性的手段。它使得从解决一个问题中获得的知识（例如如何使用 I/O 接口读取和写入磁盘文件）可以用于解决其他问题（例如通过网络套接字进行通信）。另一种思考方式是从深浅的角度：某个接口的不同实现越多，这个接口就越深。为了让一个接口可以有很多实现，它必须拥有所有底层实现的本质特性，同时又不会涉及到任何不同实现之间的具体差异。这个概念是抽象的核心所在。

继承的第二种形式是实现继承。以这种形式，父类不仅定义了一个或多个方法的签名，而且还定义了默认实现。子类可以选择继承方法的父类实现，也可以通过定义具有相同签名的新方法来覆盖它。如果没有实现继承，则可能需要在几个子类中重复相同的方法实现，这将在这些子类之间创建依赖关系（需要在方法的所有副本中复制修改）。因此，实现继承减少了随着系统的演进而需要修改的代码量。换句话说，它减少了第 2 章中描述的变更放大问题。

但是，实现继承会在父类及其每个子类之间创建依赖关系。父类中的类实例变量经常被父类和子类访问。这导致了继承层次中的类之间的信息泄露，并且使我们在修改继承层次中的一个类时很难不用考虑其他类。例如，对父类进行修改的开发人员可能需要检查所有子类，以确保所做的修改不会破坏任何内容。同样，如果子类覆盖了父类中的方法，则子类的开发人员可能需要检查父类中的实现。在最坏的情况下，程序员将需要完全了解父类下的整个类层次结构，以便对任何类进行更改。广泛使用实现继承的类层次结构往往具有很高的复杂性。

因此，应谨慎使用实现继承。在使用实现继承之前，请考虑基于组合的方法是否可以提供相同的好处。例如，可以使用小型辅助类来实现共享功能。与其从父类中继承功能，原始类可以各自建立在辅助类的功能之上。

如果没有实现继承的可行替代方案，请尝试将父类管理的状态与子类管理的状态分开。一种方法是让某些实例变量完全由父类中的方法管理，子类仅以只读方式或通过父类中的其他方法使用它们。这适用于类层次结构中的信息隐藏，以减少依赖性。

尽管面向对象编程提供的机制可能有助于实现整洁的设计，但是它们本身不能保证良好的设计。例如，如果类很浅，或者具有复杂的接口，或者允许从外部访问其内部状态，那么它们仍将导致很高的复杂性。

19.2 敏捷开发

敏捷开发是 20 世纪 90 年代末出现的一种软件开发方法，是关于如何使软件开发更加轻量、灵活和增量的一系列想法。它是在 2001 年的一次从业者会议上正式定义的。敏捷开发主要是关于软件开发的过程（团队组织、进度管理，单元测试的角色、与客户的交互等），而不是软件设计本身。但是，它与本书中的一些设计原则有关。

敏捷开发中最重要的元素之一是开发应该是增量和迭代的概念。在敏捷方法中，软件系统是通过一系列迭代开发的，每个迭代都添加并评估了一些新的功能。每个迭代都包括设计、测试和客户的反馈，这是类似于本书里提倡的增量方法的。如第 1 章所述，在项目开始时就不可能对复杂的系统进行充分的具象以决定最佳的设计。最终获得良好设计的最佳方法是增量地开发一个系统，其中每个增量都会添加一些新的抽象，并根据经验重构现有的抽象。这就类似于敏捷的开发方法。

敏捷开发的风险之一是它可能导致战术式的编程。敏捷开发倾向于将开发人员的注意力集中在功能上，而不是在抽象上，它鼓励开发人员推迟设计决策，以便尽快产出可以工作的软件。例如，一些敏捷的实践者认为，您不应该太早实现通用机制；应该先实现一个最小的专用机制，然后在确定需要的时候再重构为更为通用的机制。尽管这些论点在一定程度上是合理的，但他们反对投资思维，并鼓励采用更具战术式的编程风格。这可能会导致复杂性的快速累积。

增量式开发通常是一个好主意，但是 **软件开发的增量应该是抽象而不是功能。**可以推迟对特定抽象的所有想法，直到有功能需要它为止。一旦需要抽象，就要花一些时间进行简洁的设计，遵循第 6 章的建议并使其具有通用性。

19.3 单元测试

过去，开发人员很少编写测试。就算有测试一般也是由一个独立的 QA 团队编写的。然而，敏捷开发的原则之一是测试应该与开发紧密集成，程序员应该为他们自己的代码编写测试。这种做法现在已经很普遍了。测试通常分为两类：单元测试和系统测试。单元测试是开发人员最常编写的测试。它们很小，而且重点突出：每个测试通常验证单个方法中的一小段代码。单元测试可以独立运行，而不需要为系统设置生产环境。单元测试通常与测试覆盖工具一起运行，以确保应用程序中的每一行代码都经过了测试。每当开发人员编写新代码或修改现有代码时，他们都要负责更新单元测试以保持适当的测试覆盖率。

第二种测试包括系统测试（有时称为集成测试），这些测试可确保应用程序的不同部分能正常协同工作。它们通常涉及在生产等同环境中运行整个应用程序。系统测试更有可能由独立的 QA 或测试小组编写。

测试，尤其是单元测试，在软件设计中起着重要作用，因为它们有助于重构。没有一个好的测试套件，对系统进行重大的结构更改是很危险的。没有简单的方法可以找到代码缺陷，因此在部署新代码之前，很可能将无法检测到这些缺陷，这时再去查找和修复它们的成本要高得多。结果，在没有良好测试套件的系统中，开发人员往往会避免进行重构。他们尽量将每个新功能或缺陷修复的代码变更数量降至最低，这意味着复杂性会累积，而设计错误也得不到纠正。如果有一套很好的测试，开发人员可以在重构时更有信心，因为测试套件将发现大多数新引入的代码缺陷。这鼓励开发人员对系统进行结构改进，从而获得更好的设计。单元测试特别有价值：与系统测试相比，它们提供更高的代码覆盖率，因此它们更有可能发现任何代码缺陷。

例如，在开发 Tcl 脚本语言期间，我们决定通过将 Tcl 的解释器替换为字节码编译器来提高性能。这是一个巨大的变化，几乎影响了核心 Tcl 引擎的每个部分。幸运的是，Tcl 有一个出色的单元测试套件，我们在新的字节码引擎上运行了该套件。现有测试在发现新引擎中的错误方面是如此有效，以至于在字节码编译器的 alpha 版本发布之后仅出现了一个缺陷。

19.4 测试驱动开发

测试驱动开发是一种软件开发方法，程序员可以在编写代码之前先编写单元测试。创建一个新的类时，开发人员首先根据其预期行为为该类编写单元测试。因为该类还没有代码，没有一个测试能通过。然后，开发人员一次处理一个测试，编写足够的代码以使该测试通过。所有测试通过后，这个类的功能就完成了。

尽管我是单元测试的坚决拥护者，但我并不热衷测试驱动开发。**测试驱动开发的问题在于，它将注意力集中在让特定功能正常工作，而不是寻找最佳设计。** 这是纯粹的战术式编程，有其所有的弊端。测试驱动开发过于增量：在任何时间点，都在忙于完成一个功能并让测试通过。没有明显的时间来做设计，因此很容易搞得一团糟。

如第 19.2 节所述，增量开发的单元应该是抽象，而不是功能。一旦发现了对某个抽象的需求，就不要零散地去创建它，而应该一次性的完成其设计（或至少能提供一组合理且全面的核心功能）。这样更有可能产生整洁的设计，能使各个部分很好地契合在一起。

有一个地方先编写测试是有意义的，那就是修复代码缺陷的时候。在修复一个缺陷之前，请编写一个会由于该缺陷而失败的单元测试，然后修复该缺陷并确保相应的单元测试可以通过。这是确保您已真正修复该缺陷的最佳方法。如果您在编写测试之前就已修复了该缺陷，则新的单元测试有可能实际上并不会触发该缺陷，在这种情况下，它也无法告诉您是否真的修复了该问题。

19.5 设计模式

设计模式是解决特定类型问题（例如迭代器或观察者）的常用方法。设计模式的概念在 Gamma、Helm、Johnson 和 Vlissides 合著的《设计模式：可复用的面向对象软件的基础》一书中提及而普及，现在设计模式已广泛用于面向对象的软件开发中。

设计模式代表了另一种做设计的选择：与其从头设计新机制，不如应用一种众所周知的设计模式。在大多数情况下，这是很好的：设计模式的出现是因为它们解决了常见的问题，并且因为它们被普遍认为提供整洁的解决方案。如果设计模式在特定情况下运作良好，那么您可能很难想出另一种更好的方法。

设计模式的最大风险是过度使用。不是每个问题都可以用现有的设计模式来解决。当自定义的方法更加简洁时，请勿尝试将问题强加到设计模式中。使用设计模式并不能自动改善软件系统，只有在设计模式合适的情况下才会如此。与软件设计中的许多想法一样，设计模式是良好的并不一定意味着使用更多的设计模式也一定会更好。

19.6 Getters 和 Setters

在 Java 编程社区中，Getter 和 Setter 方法是一种流行的设计模式。Getter 和 Setter 与一个类的实例变量相关联。它们具有类似 getFoo 和 setFoo 的名称，其中 Foo 是变量的名称。Getter 方法返回变量的当前值，Setter 方法修改该值。

由于实例变量可以是公有的，不一定必须使用 Getter 和 Setter 方法。Getter 和 Setter 的作用是它们允许在获取和设置时执行额外的功能，例如在变量更改时更新相关的值、将变化通知到监听器或者对值实施约束。即使最初不需要这些功能，以后也可以在不更改接口的情况下添加它们。

虽然在你必须公开实例变量的情况下，使用 Getter 和 Setter 方法是有意义的，但最好不要先决定需要公开实例变量。公开的实例变量意味着类的实现的一部分在外部是可见的，这违反了信息隐藏的思想，并增加了类接口的复杂性。Getter 和 Setter 是浅方法（通常只有一行），因此它们在不提供太多功能的情况下使类的接口变得混乱。最好避免使用 Getter 和 Setter（或任何公开的实现层面的数据）。

建立一个设计模式的风险之一是一旦开发人员认为该模式是好的，就会试图尽可能多地使用它。这导致了 Java 中的 Getter 和 Setter 的过度使用。

19.7 结论

每当您遇到有关软件开发范式的新提案时，就必须从复杂性的角度对其进行考察：该提案确实有助于最大程度地降低大型软件系统的复杂性吗？许多提案表面上听起来不错，但是如果您深入研究，您会发现其中一些会使复杂性恶化，而不是更好。