第 1 章 介绍

编写计算机软件是人类历史上最纯粹的创作活动之一。程序员不受诸如物理定律等实际限制的约束。我们可以用现实世界中永远不会存在的行为创建令人兴奋的虚拟世界。编程不需要很高的身体技能或协调能力，例如芭蕾或篮球。所有编程都需要具有创造力的头脑和组织思想的能力。如果您能够将一个系统具象化，就可以在计算机程序中将它实现。

这意味着编写软件的最大限制是我们理解所创建系统的能力。随着程序的演进和更多功能的加入，它变得越来越复杂，其组件之间会具有微妙的依赖性。随着时间的流逝，复杂性不断累积，程序员在修改系统时将所有相关因素牢记在心中变得越来越难。这会减慢开发速度并导致代码缺陷，从而进一步拖慢开发速度并增加成本。在任何程序的生命周期中，复杂性都会不可避免地增加。程序越大，工作的人越多，管理复杂性就越困难。

好的开发工具可以帮助我们应对复杂性，许多出色的工具已经在过去的几十年中被创建出来。但是，仅凭工具我们能做的事情仍然有限制。如果我们想简化编写软件的过程，从而可以用更低的成本构建功能更强大的系统，则必须找到简化软件的方法。尽管我们尽了最大努力，复杂性仍会随着时间的推移而增加，但是更简单的设计使我们能够在复杂性取得压倒性优势之前构建出更大、功能也更强大的系统。

有两种解决复杂性的通用方法，这两种方法都将在本书中进行讨论。第一种方法是通过使代码更简单和更易理解来消除复杂性。例如，可以通过消除特殊情况或以一致的方式使用标识符来降低复杂性。

解决复杂性的第二种方法是封装它，以便程序员可以在系统上工作而不会立即暴露在所有复杂性的面前。这种方法称为模块化设计。在模块化设计中，软件系统分为模块，例如面向对象语言中的类。这些模块被设计为彼此相对独立（低耦合），以便程序员可以在一个模块上工作而不必了解其他模块的细节。

由于软件具有很好的延展性，软件设计是一个贯穿软件系统整个生命周期的连续过程。这使得软件设计与诸如建筑物、船舶或桥梁的物理系统的设计不同。但是，软件设计并非总是以这种方式被看待。在编程历史的早期阶段，设计往往都集中在项目的开始，就像其他工程学科一样。这种方法的极端称为瀑布式模型，该模型将项目划分为离散的阶段，例如需求定义、设计、编码、测试和维护。在瀑布式模型中，每个阶段都在下一阶段开始之前完成；在许多情况下，每个阶段都由不同的人负责。在设计阶段，立即设计整个系统。在设计阶段结束时冻结设计，而后续阶段的作用是充实和实现这个设计。

不幸的是，瀑布式模型很少适用于软件。软件系统本质上比物理系统更为复杂。在构建任何东西之前，不可能充分具象化出大型软件系统的设计，以了解其所有含义。结果，初始设计将有许多问题。在实现到一定程度之前，问题不会变得明显。但是，瀑布式模型此时无法适应主要的设计变更（例如，设计师可能已转移到其他项目）。因此，开发人员尝试在不改变整体设计的情况下解决问题。这导致了复杂性的爆炸式增长。

由于这些问题，当今大多数软件开发项目都使用诸如敏捷开发之类的增量方法，其中初始设计仅着重于整体功能的一小部分。这一小部分将被设计、实现和评估，然后发现和纠正原始设计中的问题，然后再设计、实现和评估更多功能。每次迭代都会暴露现有设计的问题，这些问题在设计下一组功能之前就已得到解决。通过以这种方式扩展设计，可以在系

统仍然很小的情况下解决掉初始设计的问题。较新的功能受益于较早的功能在实现过程中获得的经验，因此问题也会较少。

增量方法适用于软件，因为软件具有足够的延展性，可以在实施过程中进行重大的设计变更。相比之下，对物理系统而言，主要的设计变更更具挑战性：例如，在建筑过程中更改支撑桥梁的塔架数量是不切实际的。

增量开发意味着永远不会完成软件设计。设计在系统的整个生命周期中不断发生：开发人员应始终在思考设计问题。增量开发还意味着不断的重新设计。系统或组件的初始设计几乎从来都不是最好的。随着经验累积，不可避免地会产生更好的做事方式。作为软件开发人员，您应该始终在寻找机会来改进正在开发的系统的设计，并且应该计划将部分时间花费在设计改进上。

如果软件开发人员应始终考虑设计问题，而降低复杂性是软件设计中最重要的要素，则软件开发人员应始终考虑复杂性。这本书就是关于如何使用复杂性来指导软件设计的整个生命周期。

这本书有两个总体目标。首先是描述软件复杂性的本质："复杂性"是什么意思、为什么它很重要、以及当程序具有不必要的复杂性时如何识别？本书的第二个也是更具挑战性的目标是介绍可在软件开发过程中使用的技术，以最大程度地减少复杂性。不幸的是，没有简单的方法可以保证出色的软件设计。取而代之的是，我将提出一些与哲学相关的更高层级的概念，例如"类应该是深的"或"通过定义来规避错误"。这些概念可能不会立即确定最佳设计，但您可以使用它们来比较设计备选方案并引导您探索设计空间。

1.1 如何使用这本书

这里描述的许多设计原则有些抽象，因此如果不看实际的代码，可能很难理解它们。找到足够小的示例以包含在书中，但是又足够大以说明真实系统的问题是一个挑战（如果遇到好的示例，请发给我）。因此，这本书可能不足以让您学习如何应用这些原则。

使用本书的最佳方法是与代码审查结合使用。阅读其他人的代码时，请考虑它是否符合此处讨论的概念，以及它与代码的复杂性之间的关系。在别人的代码中比在您的代码中更容易看到设计问题。您可以使用书里描述的危险信号来发现问题并提出改进建议。审查代码还将使您接触到新的设计方法和编程技术。

改善设计技能的最好方法之一就是学会识别危险信号：危险信号表明一段代码可能比需要的复杂。在本书的过程中，我将指出一些危险信号，这些危险信号与一些主要的设计问题相关，其中最重要的内容总结在本书的最后。您可以在编码时使用它们：当看到危险信号时，停下来寻找可消除问题的替代设计。当您第一次尝试这种方法时，您可能必须尝试几种设计替代方案，然后才能找到消除危险信号的方案。不要轻易放弃：解决问题之前尝试的替代方法越多，您就会学到更多。随着时间的流逝，您会发现代码中的危险信号越来越少，并且您的设计越来越清晰。您自己的经验可能也涉及到一些其它的可用于识别设计问题的危险信号（我很乐意听到这些）。

在应用本书中的思想时，务必要节制和谨慎。每条规则都有例外，每条原则都有其局限性。如果您将任何设计创意都发挥到极致，那么您可能会陷入困境。精美的设计反映了相互竞争的思想和方法之间的平衡。有几个章节的标题为“做过头了”，它们描述了如何识别自己是否正在把事情做得过头了。

本书中几乎所有示例都是使用 Java 或 C++ 语言编写的，并且大部分讨论都是针对以面向对象的语言设计类的。但是，这些想法也适用于其他领域。几乎所有与方法有关的思想也可以应用于没有面向对象特性的语言中的功能，例如 C 语言。设计思想还适用于除类之外的模块，例如子系统或网络服务。

在这种背景下，让我们详细讨论导致复杂性的原因以及如何简化软件系统。