第一章：

在当今软件开发领域，我们经常面临一个问题，那就是如何用更高层次的抽象来描述和设计软件系统。在这个问题的背后，UML（统一建模语言）的出现，为我们提供了一个非常有力的工具。通过阅读这本书，我深刻理解了UML的起源、发展和应用，并对软件工程的设计和开发产生了新的认识。

UML是一种图形化的建模语言，通过它，我们可以更好地描述和设计面向对象的软件系统。这种语言的产生，源于20世纪80年代末和90年代初许多面向对象的图形建模语言的统一。UML的出现，让过去那混乱的图形建模语言变得有序，这让我等许多开发者倍感欣慰。

在阅读这本书的过程中，我发现UML可以用来进行建模的方式有三种：草图、蓝图和编程语言。其中，草图模式最为常见，通过这种方式，开发者可以用UML来表达系统的某些方面，如设计思路、交流观点等。而蓝图模式则注重完整性，设计者可以通过详细的设计来指导程序员编写代码。这种方式借鉴了其他工程领域的经验，如建筑工程等。编程语言模式则意味着UML可以被用作一种编程语言，但这种方式相对较少见。

第二章

在这一章中，作者详细讨论了软件开发过程中的迭代和瀑布模型，以及预测性和自适应性规划。本章深入探讨了这些方法之间的差异，以及它们在实际应用中的优缺点。

首先，作者提到了统一建模语言（UML）的发展。UML的发展源于多种面向对象分析和设计方法的结合。这些方法将图形建模语言与描述如何进行软件开发的过程相结合。然而，各方在建模语言上达成一致的同时，并未就开发过程达成共识。因此，UML仅作为一种建模语言存在。

接下来，作者详细讨论了迭代和瀑布模型。迭代模型将项目按功能子集划分，每次迭代完成一部分功能的分析、设计、编码和测试。而瀑布模型则按活动划分项目，分为需求分析、设计、编码和测试阶段。迭代模型具有较高的灵活性，适应性强，更适合需求变化较大的项目。瀑布模型在实际应用中容易出现进度跟踪困难的问题，作者建议至少采用分阶段交付的方式，而不是纯瀑布模型。

作者还讨论了预测性和自适应性规划的区别。预测性规划试图通过提前进行大量工作来提高项目后期的可预测性，而自适应性规划认为需求变化是不可避免的，因此应使用能够适应变化的规划方法。预测性规划需要精确、准确和稳定的需求，否则可能导致项目偏离预期目标。自适应性规划在需求不确定的情况下更具优势。

通过阅读本章，我深刻理解了迭代和瀑布模型、预测性和自适应性规划的差异。这对于我在实际软件开发项目中选择合适的开发过程具有重要的指导意义。这一章不仅提供了关于软件开发过程的理论知识，还为我提供了一些实践建议，使我能够更好地应对软件开发过程中可能遇到的挑战。

第三章

在学习UML（统一建模语言）时，类图是最常见和重要的一种图表。类图用于描述系统中的对象类型及它们之间的静态关系，包括属性、操作和约束条件。通过学习类图，我们可以更好地理解和探讨业务领域的概念模型。

在类图中，属性可以通过两种表示方法来描述：属性表示法和关联表示法。属性表示法将属性作为类框内的一行文本展示，而关联表示法则用一个实线来连接两个类。两种表示法在图表上看起来截然不同，但实际上表示的是相同的概念。

作者还详细介绍了属性的可见性、名称、类型、多重性等特征。其中，多重性是一个重要概念，用于表示属性可能包含的对象数量。此外，还讨论了如何在类图中表示有序和允许重复的多值属性。

在绘制类图时，我们需要注意不要忽略了对象之间的行为。为了更好地使用类图，建议从简单的概念开始，如类、关联、属性、泛化和约束，逐步引入其他高级概念。同时，不要试图在一张类图中包含所有可用的符号，而是集中精力在关键领域上。此外，在绘制类图时，要与行为建模技术相结合，以便更全面地理解软件系统。

通过阅读本文，我对UML类图的基本概念、使用方法和注意事项有了更深入的了解。在今后的项目中，我将灵活运用这些知识，提高我的系统建模能力。

文还强调了在属性和方法之间保持一致性和同步的重要性，特别是在双向关联中。实现双向关联时，需要确保在主控一端修改属性时，从属一端的属性也得到相应的更新。这种同步机制可能导致一些封装泄露，但可以通过优化代码设计和访问控制来解决。

在操作方面，作者介绍了完整的UML语法，包括可见性、名称、参数列表、返回类型和属性字符串等组成部分。参数列表的形式包括方向、名称、类型和默认值。此外，还解释了输入参数（in）、输出参数（out）和输入输出参数（inout）之间的区别。对于概念模型，作者建议使用操作来表示类的主要职责，而不是详细指定类的接口。

本文还探讨了查询操作和修改操作之间的区别，以及操作和方法之间的区别。查询操作是无副作用的，不会改变系统状态，而修改操作会改变系统状态。遵循命令-查询分离原则，可以提高代码的可读性和可维护性。操作和方法之间的区别在于，操作是在对象上调用的过程声明，而方法是过程的实现。在多态情况下，这两者之间存在差异。虽然人们通常将这两个术语互换使用，但在某些情况下，区分这两者仍具有重要意义。

总之，本文详细讨论了属性在编程中的解释，以及属性、关联、操作等在面向对象编程中的实现。通过遵循一定的编程规范和设计原则，可以提高代码的可读性、可维护性和可扩展性。在项目团队中，确保团队成员之间达成共识并遵循统一的编程规范，有助于提高项目的开发效率和质量。

第四章

作者详细讨论了序列图在面向对象编程中的重要性。序列图的主要优势在于清晰地展示了参与者之间的交互，使得调用关系一目了然，方便理解各个参与者之间的处理过程。序列图不擅长展示算法细节，如循环和条件行为，但可以通过交互帧等表示方法进行一定程度的描述。

作者强调了分布式控制的优势，即将处理分散在多个参与者之间，有助于实现更好的对象设计。分布式控制有利于使用多态性代替条件逻辑，并在面向对象环境中创建更多的重写和变化的插入点。这种风格可能让习惯于使用长过程的人感到困惑，但在经历了面向对象编程的环境后，很多人会突然领悟到分布式控制的优势。

此外，作者介绍了序列图中的创建和删除参与者的表示方法，如何使用交互帧表示循环、条件等控制结构，以及一些非官方的表示方法。尽管序列图中有多种表示条件逻辑的方法，但作者认为这些方法仍不如代码或伪代码更清晰。因此，在实际应用中，开发者可以根据实际情况选择合适的表示方法，以提高代码的可读性和可维护性。

这章主要讨论了序列图中同步和异步调用的表示方法以及何时使用序列图。在UML 2中，实心箭头表示同步消息，而线形箭头表示异步消息。同步消息意味着调用方必须等待消息处理完成，而异步消息则允许调用方在不等待响应的情况下继续处理。异步调用通常出现在多线程应用程序和基于消息的中间件中，它提高了响应速度并降低了时间耦合，但调试起来较困难。

作者建议在需要强调异步消息时使用已过时的半线箭头，以更好地引起注意。在阅读序列图时，除非确信作者有意区分同步和异步，否则不要根据箭头形状做出假设。

序列图适用于观察多个对象在单个用例中的行为。它们擅长展示对象间的协作，但不太适合精确定义行为。如果需要查看单个对象在多个用例中的行为，可以使用状态图；如果需要查看多个用例或多个线程的行为，可以考虑活动图。

此外，这段文字还介绍了CRC卡（类-职责-协作图）的概念和应用。CRC卡是一种用于探索对象交互的有效方法，专注于行为而非数据。通过对各种场景的讨论和模拟，CRC卡有助于团队成员理解每个类的高级行为。虽然CRC卡不是UML的一部分，但在面向对象设计领域中非常受欢迎。