第 3 章

CHAPTER 3

设计模式前置知识

在学习设计模式之前，首先需要了解一些基础的设计模式知识。这些知识将帮助读者更好地理解设计模式的背景、作用及它们如何解决软件开发中普遍存在的问题。本章将概述设计模式的起源、常见的设计模式类型，以及设计模式背后的核心原则。通过这些内容，读者能够为接下来的深入学习设计模式奠定坚实的基础。

19min

3.1 设计模式概述

设计模式是解决软件设计问题的一种通用方法，是经验丰富的开发者总结出的一套可复用的解决方案。在软件开发过程中，许多问题是常见的、反复出现的，而设计模式正是为了解决这些问题而设计的。学习并掌握设计模式，不仅有助于提升软件开发的质量和可维护性，也能让开发者在遇到复杂问题时快速地找到合适的解决方案。

3.1.1 设计模式的起源

设计模式的概念最早起源于 20 世纪 90 年代初期的计算机科学领域。在软件工程的发展过程中，随着系统变得越来越复杂，开发者面临着诸多的设计挑战：如何组织代码、如何保持系统的可扩展性、如何避免代码重复等。正是在这样的背景下，设计模式应运而生，成为解决这些常见问题的一种通用方法。

设计模式的真正奠基可以追溯到 GoF(Gang of Four)——他们是四位计算机科学领域的专家：Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides。这四位学者于 1994 年联合出版了《设计模式：可复用面向对象软件的基础》(*Design Patterns：Elements of Reusable Object-Oriented Software*)一书，书中详细描述了 23 种常见的设计模式，并对每种模式进行了深入分析。GoF 的工作标志着设计模式在软件开发领域的正式诞生，也为后来的设计模式研究提供了重要的理论基础。

1. GoF 书籍的影响

《设计模式：可复用面向对象软件的基础》是面向对象设计的经典之作。它不仅描述了

设计模式的概念和分类，还强调了设计模式的应用场景、实现方法及其优缺点。GoF 通过对每种设计模式进行详细说明，展示了如何通过设计模式提高系统的可维护性、可扩展性、灵活性和可重用性。该书深入探讨了以下几个方面。

1）模式的定义

设计模式并非简单的“解决方案”，而是一种经验证的可复用的设计方法。GoF 给出了一种标准的模式描述结构，包含了以下几方面核心内容。

（1）模式名称：一个设计模式的名字，便于开发者理解和沟通。

（2）问题：描述在软件开发中遇到的具体问题。

（3）解决方案：为了解决这些问题，模式给出了的具体设计方案。

（4）结果：使用设计模式后，能够带来的具体好处，如提高可扩展性、可复用性等。

（5）应用场景：说明该模式适用的具体场景。

2）模式的分类

GoF 根据设计模式解决问题的类型，将其分为三大类。

（1）创建型模式：这些模式关注对象的创建方式，旨在提高对象创建的灵活性，例如单例模式、工厂方法模式、建造者模式等。

（2）结构型模式：这些模式关注如何通过组合不同的类和对象来构建更复杂的结构，减少系统的复杂度，例如适配器模式、桥接模式、装饰器模式等。

（3）行为型模式：这些模式关注对象之间的交互，尤其是如何分配和管理职责，促进对象之间的协作，例如观察者模式、策略模式、命令模式等。

3）设计原则的强调

在书中，GoF 不仅讲解了具体的设计模式，还深入地探讨了如何通过遵循一定的设计原则来保证系统的健壮性和灵活性。最重要的原则包括单一职责原则（SRP）、开闭原则（OCP）、里氏替换原则（LSP）等。

2. GoF 书籍的核心贡献

（1）系统化的设计模式体系：GoF 将设计模式从零散的解决方案提升为系统化的设计工具，涵盖了不同种类的设计问题，为开发者提供了全面的理论框架。

（2）规范化的模式描述：通过规范的模式描述方法，GoF 书籍为后来的设计模式研究者提供了统一的语言，使不同设计模式之间的对比和应用变得更加清晰和直观。

（3）实践性和可复用性：GoF 不仅给出了设计模式的定义和理论框架，还通过具体的示例和应用场景，帮助开发者理解如何在实际开发中应用这些设计模式，提高了系统设计的质量和灵活性。

3. GoF 设计模式的普及

尽管《设计模式：可复用面向对象软件的基础》出版已有多年，但它在全球软件开发领域的影响依然深远。书中的 23 种设计模式已成为现代软件开发的基石，在许多经典的开源框架、企业级应用及大型软件系统的设计中能够看到 GoF 模式的身影。对于初学者来讲，

GoF 模式为他们提供了一套清晰的解决方案和思维方式，使他们能够在面对设计挑战时，迅速找到适合的模式。

同时，GoF 的设计思想和模式分类在后来的研究中得到了进一步发展和扩展，例如，在并发设计模式、分布式系统设计模式等领域，后来的设计模式专家对 GoF 模式进行了改进和补充，以应对更加复杂和特定的设计需求。

4. GoF 与现代设计模式

尽管 GoF 模式依然是设计模式领域的“经典之作”，但随着软件开发技术的不断发展和变化，现代设计模式的应用也变得越来越丰富。如今，除了 GoF 的 23 种设计模式，业界还提出了许多其他的设计模式，如领域驱动设计（DDD）中的设计模式、微服务架构中的设计模式等，这些模式不断适应新的开发需求。

然而，GoF 的设计模式仍然是每位开发者必备的基础知识，掌握这些模式对于理解和运用现代设计模式、提升系统的灵活性和可扩展性至关重要。

GoF 的贡献不仅在于提出了 23 种经典的设计模式，更在于它为整个软件开发行业提供了一个全新的思维框架，使开发者能够通过系统化的方式解决软件设计中常见的问题。GoF 的书籍作为设计模式的奠基之作，不仅影响了几代开发者的思维方式，也为软件设计的灵活性和可复用性提供了理论支持。学习并理解 GoF 设计模式的核心思想和应用场景，是任何开发者成为设计高手的必经之路。

3.1.2 23 种设计模式概览

设计模式是软件开发中的常用解决方案，帮助开发者应对不同的设计问题。GoF（四人帮）提出的 23 种设计模式可以分为三大类：创建型模式、结构型模式和行为型模式。每种模式都有其独特的应用场景和优势，能够提高代码的可复用性、灵活性和可维护性。以下是对 23 种设计模式的概览。

1. 创建型模式

创建型设计模式的核心目的是处理对象的创建过程，其目的是让对象的创建过程更加灵活，并隐藏具体的创建逻辑。此类模式关注如何实例化对象，尤其是如何从系统中解耦创建细节。常见的创建型模式包括以下几种。

（1）单例模式：确保一个类只有一个实例，并提供全局访问点。它主要用于需要集中管理资源或控制的场景。

（2）工厂方法模式：定义一个用于创建对象的接口，让子类决定实例化哪个类。该模式通过推迟到子类来处理对象的创建，促进了代码的解耦。

（3）抽象工厂模式：提供一个接口，用于创建一组相关或相互依赖的对象，而无须明确指定具体类。适用于需要创建多个产品族的场景。

（4）建造者模式：将一个复杂对象的构建过程抽象化，允许通过相同的构建过程创建不同的表示。它通常用于对象构建的步骤较烦琐且具有可变性的情况。

(5) 原型模式：通过复制现有的对象来创建新的对象，而不需要知道对象的具体类型。适合于对象初始化较复杂、需要频繁复制的场景。

2. 结构型模式

结构型设计模式主要关注如何通过组合不同的类和对象来构建更复杂的结构，其目的是降低系统的复杂度，提高可复用性。常见的结构型模式包括以下几种。

(1) 适配器模式：将一个类的接口转换成客户端期望的接口。它使原本接口不兼容的类能够一起工作，适用于需要适配不同接口的情况。

(2) 桥接模式：将抽象部分与实现部分分离，使它们可以独立地变化。适合于有多个可能变化维度的系统，能够提高系统的可扩展性。

(3) 组合模式：将对象组合成树形结构，以表示“部分-整体”的层次结构。它让客户端以统一的方式处理单个对象和组合对象，适用于树形结构的场景。

(4) 装饰器模式：动态地给一个对象添加额外的职责，而不影响其他对象。它允许在不修改对象的情况下，增加功能或特性，适用于需要动态扩展对象功能的情况。

(5) 外观模式：为子系统中的一组接口提供一个统一的高层接口，使子系统更加容易使用。它通过封装复杂的子系统操作，简化了外部接口的使用。

(6) 享元模式：通过共享对象来支持大量细粒度对象的复用。它适用于需要大量重复对象的场景，例如大规模的文本或图形渲染。

(7) 代理模式：为其他对象提供一个代理，以控制对该对象的访问。它通过控制对目标对象的访问来增加额外的功能，例如延迟加载、访问控制等。

3. 行为型模式

行为型设计模式主要关注对象之间的交互和职责分配，其目的是提高对象之间的协作和通信效率。常见的行为型模式包括以下几种。

(1) 责任链模式：将请求沿着处理链传递，直到找到一个合适的处理者。它通过将请求传递给多个处理对象来减少每个对象的责任，适用于多层次的请求处理场景。

(2) 命令模式：将请求封装成一个对象，从而允许用户将请求的发起者与请求的执行者解耦。它适用于需要支持撤销、重做等操作的场景。

(3) 解释器模式：为语言中的每个句子定义一个解释器，解释器能够将这些句子转换成机器能够理解的格式。适用于构建语言解析器的场景。

(4) 迭代器模式：提供一种方法顺序访问一个集合对象，而不暴露集合对象的内部结构。它使遍历集合对象变得更加统一，适用于需要遍历不同类型集合的情况。

(5) 中介者模式：定义一个中介对象来封装一组对象之间的交互，使对象之间不需要直接交互。它减少了对象之间的耦合，适用于对象间交互复杂的情况。

(6) 备忘录模式：在不暴露对象实现细节的情况下，保存对象的内部状态，以便在需要时恢复。它适用于需要保存和恢复对象状态的场景。

(7) 观察者模式：当一个对象的状态发生变化时，所有依赖于它的对象都会自动收到

通知并更新。它常用于实现事件处理和消息传递。

(8) 状态模式：允许对象在其内部状态改变时改变其行为。它使状态转移变得更加清晰，适用于状态变化频繁且有不同处理方式的场景。

(9) 策略模式：定义一系列算法，将每个算法封装起来，并使它们可以互换。它允许算法的独立变化，适用于需要动态选择不同算法的情况。

(10) 模板方法模式：定义一个操作中的算法骨架，将一些步骤的实现延迟到子类中。它通过继承和重写方法来改变算法的某些步骤，适用于算法框架固定、步骤可变的情况。

(11) 访问者模式：表示一个作用于某对象结构中的各元素的操作，它使在不改变元素类的情况下，定义作用于元素的新的操作。它适用于操作对象结构中的元素，并且操作会经常变化的情况。

3.1.3 设计模式的六大原则

设计模式的六大原则是软件设计中的核心指导思想，它们可以帮助开发者构建出高内聚、低耦合、易于维护和扩展的系统。这些原则不仅是设计模式的基石，也是优秀软件架构的灵魂所在。掌握它们，不仅能提升代码质量，还能让读者的设计更具艺术感与生命力。接下来，笔者将用通俗易懂的语言，结合生活中的比喻，为读者详细解析这六大原则，让读者在轻松愉快的氛围中领悟其精髓。

1. 单一职责原则

单一职责原则(Single Responsibility Principle，SRP)指出，一个类应该只有一个职责，并且这个职责应该完全封装在类内部。换句话说，每个类应该只有一个引起它变化的原因。

通俗解释，想象你是一位厨师，专精于烹饪一道招牌菜。如果餐厅要求你同时负责采购食材、清洗碗碟和招待客人，则你的工作将变得一团糟，工作效率会大打折扣。同样，一个类如果承担了太多职责，当其中的一个职责发生变化时，可能会无意中影响到其他职责，从而导致代码脆弱且难以维护。

1) 生活中的例子

在家庭中，妈妈负责做饭，爸爸负责修理，孩子负责学习。每个人专注于自己的职责，家庭运转井然有序。如果让妈妈同时做饭、修理和辅导功课，则效率和质量都会下降。

2) 应用场景

在软件设计中，单一职责原则鼓励读者将功能模块化，例如，一个用户管理类只负责用户信息的增、删、改、查操作，不应同时处理日志记录或权限控制。这样，当需求发生变化时，只需修改对应的类，而不会波及整个系统。

2. 开放-封闭原则

开放-封闭原则(Open-Closed Principle，OCP)指出，软件实体(类、模块、函数等)应该对扩展开放，而对修改封闭。

通俗解释，这就好比乐高积木：可以通过添加新的积木块来扩展功能，但不应该修改已

有的积木。也就是说，当你需要新功能时，应该通过添加新代码来实现，而不是修改现有的代码。这样可以避免引入新的错误，保持系统的稳定性。

1）生活中的例子

智能手机的设计就是一个很好的例子。手机厂商通过发布新的应用程序来增加功能，而不是频繁地修改手机的操作系统。用户可以自由选择安装新应用，而不必担心手机的基础功能受影响。

2）应用场景

在设计模式中，开放-封闭原则常常通过抽象和多态来实现，例如，读者可以先定义一个抽象基类，然后通过继承或实现接口来扩展新功能，而不改动基类代码。这样，系统的可扩展性便大大增强。

3. 里氏替换原则

里氏替换原则（Liskov Substitution Principle，LSP）指出，子类对象必须能够替换父类对象并且程序功能不受影响。

通俗解释，这就像父母和孩子的关系：孩子应该能够接替父母的工作，而不破坏原有的工作流程。也就是说，子类在继承父类的同时，必须保持与父类行为的一致性，不能违背父类的设计意图。

1）生活中的例子

假设你有一台打印机，无论是黑白打印机还是彩色打印机，它们都应该能够完成基本的打印任务。如果彩色打印机不能打印黑白文档，就违背了里氏替换原则。

2）应用场景

在面向对象设计中，里氏替换原则确保了多态的有效性，例如，当一种方法接受一个基类参数时，传入任何子类对象都应该正常工作，而不需要额外地进行调整。

4. 依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）指出，高层模块不应该依赖低层模块，二者应该依赖于抽象，而且抽象不应该依赖于细节，细节应该依赖于抽象。

通俗解释，这就像插头和插座的关系。插座（高层模块）提供标准的接口（抽象），各种电器（低层模块）只要符合这个接口，就可以插入使用。这样，插座不需要关心电器的具体类型，电器也不需要关心插座的实现细节。

1）生活中的例子

USB 接口就是一个典型的例子。无论是 U 盘、鼠标还是键盘，只要符合 USB 标准都可以连接到计算机上。计算机（高层模块）依赖于 USB 接口（抽象），而各种设备（低层模块）则实现这个接口。

2）应用场景

在软件设计中，依赖倒置原则通过接口或抽象类来实现模块间的解耦，例如，一个业务逻辑层不直接依赖于具体的数据访问类，而是依赖于数据访问接口。这样，数据访问的实现

可以灵活替换，而不影响业务逻辑。

5. 接口隔离原则

接口隔离原则(Interface Segregation Principle，ISP)指出，客户端不应该被迫依赖它不需要的接口。

通俗解释，这就像餐厅的菜单：你不应该被迫点一整套菜品，而应该能够选择你需要的菜品。同样，接口不应该包含太多不相关的方法，否则实现接口的类可能被迫实现不需要的方法，增加了不必要的负担。

1) 生活中的例子

在快餐店，可以单独点汉堡、薯条或饮料，而不是必须点一个包含所有物品的套餐。这样，可以根据自己的需求进行选择，避免浪费。

2) 应用场景

在设计 API 时，接口隔离原则建议将大接口拆分为多个小接口，例如，一个通用的 Animal 接口可能包含 eat、sleep、fly 等方法，但对于不会飞的动物来讲，fly 方法是多余的，因此更好的做法是将 fly 方法独立为一个 Flyable 接口。

6. 迪米特法则

迪米特法则(Law of Demeter，LoD)并不是 SOLID 原则的一部分，但它是软件设计中的一个重要的指导原则，它指出，一个对象应该尽量少了解其他对象的内部实现，应该与其直接的依赖进行交互。

通俗解释，这就像社交网络中的“朋友的朋友”：你不应该直接与朋友的朋友交流，而是通过你的朋友来传递信息。也就是说，一个类不应该直接访问其他类的内部细节，而是通过接口或方法来交互，减少耦合。

1) 生活中的例子

在公司中，你通常只与你的直属上司或下属沟通，而不是越级与其他人直接交流。这样可以保持信息的有序传递，避免混乱。

2) 应用场景

在软件设计中，迪米特法则鼓励读者通过封装和接口来隐藏内部实现，例如，一个订单处理类不应该直接访问数据库层的具体实现，而是通过数据访问层提供的接口来获取数据。这样，即使数据库层发生了变化，订单处理类也不受影响。

这六大原则是设计模式的灵魂，它们如同一组和谐的音符，共同奏响了软件设计的华美乐章。掌握它们，将能够使读者的软件系统具有以下特点。

(1) SRP：让读者的类专注而高效，如同专精一技的匠人。

(2) OCP：让读者的系统可以灵活扩展，如同乐高积木般自由组合。

(3) LSP：让读者的继承体系稳健可靠，如同家族传承般井然有序。

(4) DIP：让读者的模块解耦，如同插头与插座的完美适配。

(5) ISP：让读者的接口精简实用，如同菜单上的精选菜品。

(6) LoD：让读者的类间通信简洁明了，如同社交中的礼仪规范。

设计模式的六大原则是软件设计中的核心指导思想，它们帮助开发者创建出松耦合、高内聚、易于维护和扩展的软件系统。通过遵循这些原则，读者可以避免常见的代码设计问题，提高代码质量和开发效率。

小结

设计模式的六大原则为开发者提供了构建高质量软件系统的核心指导思想。这些原则不仅可以帮助开发者确保代码的高内聚、低耦合，还可以让系统变得更加易于维护和扩展。通过单一职责原则(SRP)，能够确保每个类专注于一个单一任务，从而提高代码的可读性和可维护性；开放-封闭原则(OCP)和依赖倒置原则(DIP)则可以帮助开发者在系统扩展时，尽量避免修改现有代码，从而保持系统的稳定性和灵活性；里氏替换原则(LSP)保证了继承结构的正确性，避免了子类与父类不兼容的问题；接口隔离原则(ISP)通过拆分接口，使客户端只依赖其实际需要的接口，避免不必要的负担；最后，迪米特法则(LoD)则鼓励开发者减少类之间的直接依赖，通过间接的接口进行交互，从而降低耦合。

掌握这些设计原则，不仅能帮助开发者在面对复杂的系统设计时做出更清晰、更高效的决策，还能在实际开发中提高系统的可扩展性、灵活性和可维护性。通过遵循这些原则，读者将能够设计出更加稳健且易于维护的软件架构，提升软件的质量和开发效率。

3.2　先实现后重构


20min

在软件开发过程中，许多开发者和团队会面对一个常见的困惑：是先设计好系统的结构再开始实现，还是边实现边优化、边重构？在实际开发过程中，“先实现后重构”是一种非常有效且有意义的工作方法，它鼓励开发者在初期快速实现功能，并在后续迭代中不断对代码进行重构和优化设计。接下来，本节将深入探讨这种方法的意义、优势、挑战和实践步骤，同时通过实际的代码示例展示这一过程的实施方式。

3.2.1　先实现后重构的概念

“先实现后重构”指的是在初期阶段，开发者专注于功能的实现和需求的完成，而不必过度担心代码的完美结构或最优设计。初期的代码可以是简单、直接且不完美的。只要代码能够正常地运行，开发者就可以继续推进开发，等到系统初步完成后，再对代码进行重构和优化。

这种方法的核心思想是，实现优先于设计，重构优于完美设计。也就是说，可以通过先实现功能来快速验证需求、收集反馈，并根据实际的需求和使用情况来进行设计上的改进。

1. 为什么选择先实现后重构

“先实现后重构”方法并不是随意选择的，它有以下几个非常重要的理由。

（1）减少分析过度的风险：在许多项目中，过度分析和设计往往会导致“分析瘫痪”，即花费大量时间在需求和设计阶段，却没有实际的代码实现。开发者可能会因为对系统架构、设计模式等进行过度思考而拖延开发进度。通过先实现功能，开发者可以更早地看到系统的实际效果，避免过度分析和设计带来的负面影响。

（2）适应需求的变化：软件需求常常随着项目的进行而发生变化，即使开发者在开始时已经做了精心的设计和规划，最终的需求也可能会有所调整或重新定义。通过“先实现后重构”，开发者能够在功能实现之后，根据实际的需求变化来推动设计的调整。这样可以避免在需求不明确的阶段投入过多的设计工作。

（3）验证和改进设计：“先实现后重构”鼓励开发者通过实际的代码来验证设计的有效性。很多时候，理论上的设计方案可能在实际使用中并不那么适用，或者在代码实现时会出现未曾预料到的问题。通过先实现并运行系统，开发者能够收集反馈，识别出潜在的设计缺陷或优化之处，从而在重构时做出更加精确的决策。

（4）迭代式优化：采用“先实现后重构”的方法，系统的功能和架构会随着开发的推进逐步得到改善。在每次功能迭代和版本发布后，开发者可以通过重构来优化系统，解决性能瓶颈、代码重复等问题。通过这种持续的优化，系统会逐渐变得更加稳定、灵活和高效。

2. 先实现后重构的实践步骤

实施“先实现后重构”时，通常会遵循以下几个步骤。

（1）快速实现功能：在开发初期，专注于快速实现需求，而不是过多关注代码的架构或设计。目标是确保功能能够按预期工作，并满足最基本的需求。这个阶段的代码可能比较简单或不完美，甚至可能存在重复和冗余的地方，但最重要的是功能要尽快落地。

（2）收集反馈并验证需求：在实现了基本功能后，通过测试、用户反馈或内部评审等方式，收集关于代码和功能的反馈。验证这些功能是否可以满足需求，并确定是否存在设计缺陷或优化的空间。

（3）进行重构和优化：在收集反馈后，开始对代码进行重构和优化。重构不仅是为了提高代码的质量，也可能包括对设计模式的应用、架构的调整及性能优化。这个阶段的重构应该遵循设计模式和架构原则，确保代码更具可维护性和可扩展性。

（4）迭代改进：通过不断迭代开发和重构，系统会逐步完善和优化。每次添加新功能时都应该通过重构和优化来提升系统的整体质量。每次迭代都是一个不断优化的过程，使代码更加健壮、灵活和高效。

3. 优势

先实现后重构至少有以下 4 点优势。

（1）加速开发进度：开发者不需要在一开始就完美地设计整个系统，可以尽早开始开发，快速完成初步版本。

（2）提高灵活性：通过实际的开发和使用情况来调整设计，能够更好地适应需求变化。

（3）减少风险：在早期阶段验证设计的可行性，而不是在设计阶段过度投入。

（4）可持续优化：通过不断地进行重构和优化，系统能够逐步改进，避免过早的设计锁死。

4. 挑战

除了优势，先实现后重构同样需要面临以下 3 点挑战。

（1）代码质量可能不高：在初期快速实现时，可能会出现不符合最佳实践的代码，必须通过后期重构进行修正。

（2）需要良好的重构实践：重构本身也需要技巧和经验，重构过程中的不当操作可能会导致出现问题。

（3）项目管理难度增加：如果没有有效的重构和优化计划，则代码可能会积累大量的技术债务，影响后续的开发。

3.2.2　图书管理系统的实现与重构

通过以下简单的图书管理系统示例，展示从初版实现到重构优化的过程。本节将使用 C++和 C＃分别呈现两种语言的实现。

1. 初版实现

初版本是大多数初学者面向单一的点需求开发时下意识写出来的版本，这样的代码往往可以满足特定需求，但是对拓展不友好。

C++示例代码如下：

```
//Chapter-03/Section-02/Cpp/3-2-1.cpp
#include <iostream>
#include <string>
#include <vector>

class Library {
    std::vector<std::string> books;
public:
    void addBook(std::string book) {
        books.push_back(book);
        std::cout << "Added: " << book << std::endl;
    }
    void findBook(std::string book) {
        for (const auto& b : books) {
            if (b == book) {
                std::cout << "Found: " << book << std::endl;
                return;
            }
        }
        std::cout << book << " not found." << std::endl;
    }
};

int main() {
```

```
    Library lib;
    lib.addBook("Design Patterns");
    lib.findBook("Design Patterns");
    return 0;
}
```

C#示例代码如下：

```
//Chapter-03/Section-02/Cs/3-2-1.cs
using System;
using System.Collections.Generic;

class Library {
    private List<string> books = new List<string>();
    public void AddBook(string book) {
        books.Add(book);
        Console.WriteLine( $ "Added: {book}");
    }
    public void FindBook(string book) {
        if (books.Contains(book)) {
            Console.WriteLine( $ "Found: {book}");
        } else {
            Console.WriteLine( $ "{book} not found.");
        }
    }
}

class Program {
    static void Main() {
        Library lib = new Library();
        lib.AddBook("Design Patterns");
        lib.FindBook("Design Patterns");
    }
}
```

这个版本简单粗暴，Library 类既管理书籍列表，又处理输入/输出，违反了单一职责原则(SRP)。C++和C#实现类似，但C++使用vector和手动循环，C#则利用List的Contains方法，语法更简洁。

2. 重构后的版本

这里的重构也只是一个简单的示例，在实际开发中，随着不同种类需求的增加，代码一定会更加丰富。

C++示例代码如下：

```
//Chapter-03/Section-02/Cpp/3-2-2.cpp
#include<iostream>
#include<string>
#include<vector>

class BookManager {
    std::vector<std::string> books;
```

```
public:
    void addBook(std::string book) {
        books.push_back(book);
    }
    bool containsBook(std::string book) const {
        for (const auto& b : books) {
            if (b == book) return true;
        }
        return false;
    }
};

class LibraryUI {
    BookManager manager;
public:
    void addBook(std::string book) {
        manager.addBook(book);
        std::cout << "Added: " << book << std::endl;
    }
    void findBook(std::string book) {
        if (manager.containsBook(book)) {
            std::cout << "Found: " << book << std::endl;
        } else {
            std::cout << book << " not found." << std::endl;
        }
    }
};

int main() {
    LibraryUI lib;
    lib.addBook("Design Patterns");
    lib.findBook("Design Patterns");
    return 0;
}
```

C#示例代码如下：

```
//Chapter-03/Section-02/Cs/3-2-2.cs
using System;
using System.Collections.Generic;

class BookManager {
    private List<string> books = new List<string>();
    public void AddBook(string book) {
        books.Add(book);
    }
    public bool ContainsBook(string book) {
        return books.Contains(book);
    }
}

class LibraryUI {
    private BookManager manager = new BookManager();
```

```
        public void AddBook(string book) {
            manager.AddBook(book);
            Console.WriteLine( $ "Added: {book}");
        }
        public void FindBook(string book) {
            if (manager.ContainsBook(book)) {
                Console.WriteLine( $ "Found: {book}");
            } else {
                Console.WriteLine( $ "{book} not found.");
            }
        }
    }

    class Program {
        static void Main() {
            LibraryUI lib = new LibraryUI();
            lib.AddBook("Design Patterns");
            lib.FindBook("Design Patterns");
        }
    }
```

重构后，职责被清晰分离：BookManager 负责书籍的核心管理，LibraryUI 负责处理用户界面逻辑，符合 SRP。C++ 和 C# 版本的差异依然体现在容器与方法上，但设计思想一致。这样的结构更容易扩展，例如添加数据库支持时只需修改 BookManager，而不影响 UI 层。

小结

“先实现后重构”是一种务实且灵活的开发方法，能够帮助开发者快速实现功能，并在后续的开发迭代中不断地优化系统。通过实现初期功能并验证需求，开发者可以在实际使用中发现潜在的问题并进行改进。重构不仅让代码更加整洁和高效，还能够在需求变化时提供更高的适应性。掌握这一方法后，读者将能够在紧张的开发周期中迅速推进项目，并在冷静的反思和迭代中优化设计，最终交付一个高质量的软件系统。


10min

3.3 需求频繁变动对设计模式的挑战

在软件开发的真实环境中，需求的变动几乎是不可避免的。特别是在快速发展的市场、不断变化的客户反馈及技术的快速演进的背景下，软件系统必须具备足够的灵活性来应对这些变化。设计模式，作为提升软件质量、可扩展性和可维护性的工具，其应用通常假设需求相对稳定，并侧重于为系统提供长期的可扩展和可维护架构，然而，当需求频繁变动时，这些设计模式的有效性和实用性将面临诸多挑战，甚至可能会导致一些预期外的问题，如代码膨胀、重构成本增加、技术债务累积等。

本节将深入探讨需求频繁变动对设计模式的影响，并提供如何应对这种挑战的策略和

方法，尤其从代码膨胀、重构成本、技术债务等角度，分析需求变动带来的深层次问题。

1. 需求变动对设计模式的深层挑战

需求频繁变化，特别是在大规模、复杂系统的开发过程中，往往会对设计模式的实施带来直接的负面影响。设计模式本质上是为了解决系统中普遍存在的问题，但这些模式常常假设需求在较长时间内保持稳定，而当需求不断变动时，设计模式的应用可能会导致一些不易察觉的副作用，增加系统的复杂性、导致代码膨胀，甚至影响系统的长期可维护性。

1）代码膨胀与复杂性增加

设计模式往往要求系统具备较高的抽象性和灵活性，在满足当前需求的同时，为未来的扩展预留空间。在理想情况下，这样的设计会提高代码的可复用性和可维护性，然而，需求的频繁变化可能会使这些抽象的设计变得不必要，甚至过于复杂。为了应对不断变化的需求，开发者在设计时往往会提前引入额外的类、接口和方法，以应对潜在的需求。最终，这些设计模式的实施可能会导致代码膨胀，使系统的复杂度大幅增加，导致开发和维护的难度增大，例如以下两种场景：

（1）在采用策略模式时，开发者通常会为不同的业务逻辑创建多个策略类，并为每个场景设计一种策略接口。初期看似能够清晰地解决问题，但随着需求的不断变动，新的策略类会层出不穷，策略接口的数目会不断增加，最终造成类和接口数量膨胀。

（2）工厂模式是另一种常见的设计模式，它集中管理对象的创建，然而，在需求变动频繁的场景中，每次新增或修改一个功能都可能需要对现有的工厂类进行修改和扩展，导致类的数目激增，最终使代码变得复杂且难以维护。

随着这些模式的实施，系统会变得更加复杂，维护的成本和调试的难度也会逐渐增加，从而影响开发团队的效率。

2）重构成本的成倍增加

设计模式的核心目的是通过抽象和结构化设计来提高代码的可扩展性、可维护性。理论上，设计模式能够在需求发生变化时提供灵活的调整空间，然而，当需求变动时，设计模式的高度抽象可能会导致重构成本的成倍增加。

例如，设计模式中常常通过抽象类、接口和继承来提供灵活性，预防重复的代码。当需求发生变化时，这种高度抽象的结构往往需要进行大规模修改。

（1）单例模式被广泛地应用于系统中需要共享资源的场景，但当需求变化时，单例模式可能会引入不必要的全局状态或难以测试的情形，导致它变得不适用。此时，开发者可能需要将系统中的单例模式替换为工厂模式、依赖注入等方式，这不仅会导致大量的重构工作，而且可能会影响现有系统中的多个部分。

（2）模板方法模式通常为固定的业务流程提供通用的框架，但如果业务流程频繁变化，则模板方法中的某些步骤可能需要不断调整。这种不断重构和修改的工作会导致大量的代码更改，从而增加重构成本。

这种重构的代价往往是成倍增加的。为了使系统能够适应需求的频繁变化，开发者往往需要重新设计系统的架构，重构整个系统的设计，导致工作量和开发成本急剧上升。

3）技术债务的累积

随着需求变化的不断推进，开发团队往往为满足目前的需求做出妥协，牺牲了系统设计的部分质量。这种为了赶进度而积累的“技术债务”在长期内可能会变得越来越严重，从而影响整个系统的健康。特别是在需求变化频繁的项目中，开发团队可能会频繁地使用设计模式，但由于未能预见到需求的未来变化，过度的设计模式应用可能最终会成为技术债务的根源。

例如，策略模式和工厂模式等设计模式在需求变化频繁的情况下，可能需要频繁地修改和调整。当这些模式在系统中滞留太久，导致代码变得冗余，维护和扩展变得困难，技术债务也就开始积累。团队可能因为过早设计复杂的架构而没有对需求变动进行充分预估，从而使技术债务不可避免地增加。

随着时间的推移，这些技术债务的累积会逐渐影响系统的性能、可维护性和可扩展性，从而导致系统的整体质量降低。

2. 灵活的设计与重构策略

尽管需求频繁变动给设计模式带来了诸多挑战，但这并不意味着设计模式无法适应这种变化。实际上，设计模式本身提供的灵活性和可扩展性可以帮助开发者应对这些挑战。以下是一些应对需求变动挑战的策略。

1）灵活应用设计模式

设计模式并不是解决所有问题的万能钥匙，开发者在使用设计模式时，应该灵活调整其应用。避免在需求不明确时过度设计，通过合理选用设计模式来应对不断变化的需求。

（1）策略模式：当业务逻辑频繁变化时，使用策略模式将不同的算法封装到独立的策略类中，可以在不修改客户端代码的情况下动态地替换策略。

（2）观察者模式：对于频繁变动的事件驱动需求，使用观察者模式来实现对象间的解耦，使在某个对象状态发生变化时，能够灵活地通知其他对象，而无须修改原有对象的代码。

灵活应用设计模式可以使系统具有较高的适应性，避免过度设计和代码膨胀。

2）逐步重构与渐进式设计

面对需求变化，逐步重构而非一次性大规模重构能有效地减轻开发压力。逐步重构意味着在每个迭代周期中，适应需求变化时进行小范围修改，而不是一次性改变整个系统架构。通过渐进式设计，可以在保证当前需求得到满足的前提下，逐步改善代码质量，避免系统过度复杂化。

例如，在初期不急于使用复杂的设计模式，先实现最基础的功能，逐步通过重构引入适合的设计模式来满足日益增长的需求变化。

3）强调简洁的设计

为了应对需求变动带来的复杂性，开发者应强调简洁设计，避免在需求不明确时就过早引入复杂的设计模式。采用 KISS 原则（Keep It Simple, Stupid），即保持设计简单而易于理解，尽量减少不必要的抽象和设计模式。

简洁设计能够减少不必要的功能冗余，使系统在需求发生变化时能够更灵活地调整。

这样可以避免过度设计造成的代码膨胀，同时为后期的优化和重构留出空间。

4）持续重构与技术债务管理

当需求发生变化时，重构是不可避免的。为了避免技术债务的积累，开发团队需要保持持续的重构意识。技术债务管理不仅需要定期地进行代码审查和重构，更需要在开发过程中进行快速反馈和持续改进。

通过持续集成、自动化测试及定期重构，开发者能够在每次需求发生变化时，以及时调整系统设计，确保系统在不断变化的环境中保持灵活性。

小结

需求频繁变动对设计模式的应用确实带来了一些挑战，如代码膨胀、重构成本增加和技术债务累积，然而，设计模式本身为开发者提供了灵活性、可扩展性和高效的架构解决方案。通过灵活应用设计模式、逐步重构、保持简洁的设计及持续的技术债务管理，开发团队可以有效地应对需求变化带来的挑战，使系统能够在变化中保持高效和灵活。最终，设计模式将帮助开发者构建更加稳健、可维护和可扩展的软件系统，适应复杂且动态的需求环境。