实验十二 设计模块（一 ）

实验目的：

1. 培养设计原则实践的能力

2. 学习依赖注入（dependency injection）

实验内容：

1. 参考教材6.2，结合项目的进程和开发历程，从设计原则的几个方面，组员对负责设计的模块进行评估，思考存在的问题和解决方案。书P207-217

模块化 接口 （刘易）

模块化评估

1. 模块独立性

现状：

系统采用分布式架构，分为客户端（用户端和骑手端）、服务端和管理后台。每个部分都有独立的功能模块，例如用户功能包括注册登录、浏览餐厅和菜品、购物车管理等。

问题：

文档中缺少对模块独立性的具体度量标准，如耦合度和内聚度的详细描述。

例如，用户模块和订单管理模块之间可能存在数据依赖和功能调用，但未详细说明这些依赖关系的性质和程度。

解决方案：

引入耦合度和内聚度的度量工具：

耦合度：度量模块之间的依赖程度，包括内容耦合、公共耦合、控制耦合、标记耦合、数据耦合等。

内聚度：度量模块内部元素的关联程度，包括功能内聚、通信内聚、过程内聚、时态内聚、逻辑内聚、巧合内聚等。

模块划分方法：

使用Dijkstra的关注点分离原则，将系统不相关的功能分离到各自的模块中，如用户管理模块、订单处理模块、支付模块等。

2. 模块边界清晰

现状：

项目文档中定义了系统的主要功能和各用户角色的职责，但缺少详细的模块接口说明。

问题：

模块边界不够明确，容易导致功能重叠和模块间的耦合增加。例如，订单创建支付和订单跟踪可能涉及多个模块，如果边界不清晰，会导致功能实现混乱。

解决方案：

接口规格说明：

为每个模块编写详细的接口文档，定义每个模块提供的服务、依赖的外部服务、输入输出数据格式、调用顺序等。

模块功能定义：

清晰定义每个模块的功能边界，避免功能重叠。例如，订单管理模块仅负责订单的创建、支付和跟踪，不涉及用户注册和登录。

接口评估

1. 接口定义清晰

现状：

项目文档提到系统应提供标准的API接口，但未提供详细的接口定义。

问题：

缺少详细的接口定义，可能导致开发过程中接口使用不当或功能重复实现。例如，不同模块可能重复实现用户身份验证功能。

解决方案：

详细接口文档：

编写详细的API文档，包含操作名称、参数、返回值、前置条件、后置条件等。例如，用户注册接口文档应详细说明需要的参数（用户名、密码、电子邮件等），返回值（注册成功或失败），前置条件（用户名未被注册）和后置条件（用户信息保存到数据库）。

接口设计规范：

制定接口设计标准和规范，确保所有模块接口的一致性。例如，所有接口采用RESTful风格，统一使用HTTP方法（GET、POST、PUT、DELETE）和状态码（200、404、500等）。

2. 接口独立性

现状：

接口设计中强调了数据安全和高并发访问，但未充分考虑接口的独立性。

问题：

如果接口之间存在过多的依赖，可能导致系统扩展性和可维护性降低。例如，支付模块依赖订单模块和用户模块的多个接口，可能会导致接口调用链过长。

解决方案：

降低接口依赖性：

通过设计松散耦合的接口，减少模块之间的依赖。例如，支付模块通过独立的支付网关接口完成支付，不直接依赖订单模块和用户模块。

接口抽象化：

抽象接口功能，提供通用的服务。例如，定义一个统一的用户验证接口，所有模块调用该接口进行用户身份验证，而不是各模块自行实现。

存在的问题及解决方案

1. 模块耦合度高

问题：

模块间存在紧密耦合，影响系统的灵活性和维护成本。例如，用户模块和订单模块之间的紧密耦合可能导致一个模块的变更影响到另一个模块。

解决方案：

重构模块：

采用依赖倒置原则（Dependency Inversion Principle），将高层模块的依赖关系反转为接口依赖。例如，定义用户服务接口和订单服务接口，用户模块和订单模块分别实现各自的接口，通过接口调用彼此的功能，减少直接依赖。

使用中介者模式：

引入中介者模式（Mediator Pattern），通过中介者对象管理模块间的交互，减少模块间的直接依赖。例如，用户模块和订单模块通过中介者对象进行通信，中介者负责协调模块间的交互。

2. 接口一致性差

问题：

接口定义不一致，导致系统集成和维护困难。例如，不同模块的接口返回值格式不一致，可能导致前端处理复杂。

解决方案：

接口设计标准：

制定接口设计标准和规范，确保所有模块接口的一致性。例如，统一接口的返回值格式，所有接口返回统一的JSON格式数据，包括状态码、消息和数据。

接口测试和验证：

对所有接口进行测试和验证，确保接口定义和实现的一致性。例如，使用自动化测试工具对所有API接口进行测试，确保接口的正确性和一致性。

信息隐藏 增量式开发 （王攀）

在评估一个负责设计的模块时，信息隐藏和增量式开发是两个重要的概念。下面，我将分别从这两个角度讨论模块设计中可能存在的问题及相应的解决方案。

一、信息隐藏

可能存在的问题：

1. 接口设计不清晰:

模块的接口如果设计不清晰，容易导致模块的使用者误用或者滥用模块的内部功能，增加了维护难度。

2. 内部实现暴露：

如果模块的内部实现细节暴露给了外部使用者，那么任何对模块内部的修改都可能影响到使用该模块的其他部分，降低了系统的灵活性和可维护性。

3. 耦合度高：

模块之间如果耦合度高，意味着一个模块的变化可能引起其他多个模块的变化，影响系统的稳定性。

解决方案：

1. 明确接口：

设计清晰、简洁且稳定的接口，只暴露必要的功能和数据。使用接口和抽象类来定义模块的功能，并尽可能隐藏实现细节。

2. 封装内部实现：

通过封装，将模块的内部实现细节隐藏起来。可以使用访问修饰符（如private、protected）来限制外部对模块内部的访问。

3. 模块独立性：

尽量降低模块之间的依赖，通过依赖注入、服务定位器等设计模式实现模块的松耦合，提高系统的灵活性和可维护性。

二、增量式开发

可能存在的问题：

1. 需求变更管理困难：

在增量式开发中，需求频繁变更会导致设计和实现需要不断调整，可能出现前后不一致的情况。

2. 版本管理复杂：

多个增量版本并行开发和维护，容易出现版本冲突和集成问题。

3. 测试覆盖不足：

增量开发中，每次的增量功能需要充分测试，如果测试覆盖不足，会导致系统中出现较多的缺陷。

解决方案：

1. 需求管理：

采用敏捷需求管理方法，如用户故事、产品待办列表（Product Backlog），并通过迭代规划和回顾会议不断调整需求，确保每个增量版本都满足当前的优先级需求。

2. 持续集成：

使用持续集成（CI）工具，如Jenkins、Travis CI等，自动化构建和测试每个增量版本，及时发现和解决版本冲突和集成问题。

3. 自动化测试：

设计并实现全面的自动化测试，包括单元测试、集成测试和回归测试，确保每次增量开发的功能都经过充分验证，减少缺陷的引入。

4. 版本控制：

使用版本控制系统（如Git），并采用分支管理策略（如Git Flow、Feature Branch）来管理不同增量版本的开发和集成，确保代码库的稳定性和可追溯性。

总而言之，通过实施信息隐藏，可以减少模块之间的耦合，提高系统的可维护性和灵活性。而通过增量式开发，能够更快地响应需求变化，提高开发效率和产品质量。两者结合使用，可以显著提升系统设计和开发的整体质量。

抽象 通用性 （丁弘扬）

在评估外卖系统的设计模块时，从抽象和通用性这两个设计原则的角度出发，可以进行以下评估、思考存在的问题以及提出相应的解决方案。

抽象评估

评估内容：

模块化设计：系统是否将功能合理地划分为独立的、高内聚、低耦合的模块。

接口定义：模块间的接口是否清晰、简洁，且隐藏了不必要的实现细节。

抽象层次：系统是否提供了适当的抽象层次，使得业务逻辑与具体实现分离。

存在的问题：

耦合度过高：模块间依赖关系复杂，导致修改一个模块可能影响多个其他模块。

抽象层次不足：业务逻辑与具体实现紧密耦合，缺乏必要的抽象层，使得系统难以扩展和维护。

接口设计不合理：接口定义过于复杂或过于简单，无法有效支持模块间的通信和协作。

解决方案：

明确模块职责：重新划分模块，确保每个模块只负责一个特定的功能或业务逻辑。

引入设计模式：使用设计模式来优化接口设计，降低模块间的耦合度。

增加抽象层：通过引入中间层或适配器模式等方式，将业务逻辑与具体实现分离，提高系统的抽象层次。

通用性评估

评估内容：

可扩展性：系统是否能够方便地添加新功能或修改现有功能，而不需要对整体架构进行大规模修改。

可配置性：系统是否提供了灵活的配置选项，以满足不同业务场景的需求。

跨平台兼容性：系统是否能够在不同的操作系统、设备和网络环境下稳定运行。

存在的问题：

硬编码限制：系统中存在硬编码的业务规则或配置，导致系统难以适应新的业务场景或变化。

缺乏灵活性：系统配置选项有限，无法满足所有业务场景的需求。

平台依赖性强：系统过度依赖特定的技术栈或平台，导致跨平台兼容性差。

解决方案：

引入配置管理：使用配置文件、数据库或外部服务来管理系统的业务规则和配置选项，提高系统的可配置性和灵活性。

采用微服务架构：将系统拆分为多个独立的微服务，每个微服务负责一个特定的业务功能，并通过统一的接口进行通信。这样可以降低系统的耦合度，提高可扩展性。

使用跨平台技术：选择跨平台的技术栈和框架来构建系统，确保系统能够在不同的操作系统和设备上稳定运行。同时，使用标准的网络通信协议来确保系统的网络兼容性。

通过优化模块设计、引入设计模式、增加抽象层、引入配置管理、采用微服务架构和使用跨平台技术等方式，可以提高系统的可维护性、可扩展性、可配置性和跨平台兼容性，从而确保系统能够满足不断变化的业务需求。

2. 阅读下面DI资料（或查阅其它相关资料），学习依赖注入技术。（ 张勇）

什么是依赖注入（DI）？

依赖注入是一种设计模式，允许在运行时（而不是编译时）选择组件，从而实现松耦合代码。这意味着代码只关心抽象而不是具体的实现，便于在运行时替换组件而无需修改代码。

依赖注入的好处

可扩展性：轻松添加新功能，而不需要在代码库的多个部分进行更新。

后期绑定：在运行时选择使用哪些组件。

并行开发：多个团队可以独立在不同的层次工作。

可维护性：组件独立，功能隔离，便于定位问题和调整功能。

可测试性：便于对代码的小部分进行隔离测试。

S.O.L.I.D. 原则

依赖注入与S.O.L.I.D.原则密切相关，这些是面向对象设计（OOD）的五个原则，由Robert C. Martin（即Uncle Bob）提出。

依赖注入模式

构造函数注入：通过构造函数参数传递依赖。

属性注入：设置对象的属性以注入依赖。

方法注入：通过方法调用传递依赖。

环境上下文：依赖存储在环境上下文中，组件从环境中检索它们。

服务定位器：通过服务定位器来解析依赖。

示例应用分析

文档通过一个非DI的示例应用开始，逐步展示如何在应用中添加DI以增加其可扩展性和可测试性。示例应用使用WCF服务从存储库获取数据，并使用MVVM模式。

依赖注入实现

构造函数注入：通过修改ViewModel的构造函数来注入仓库接口，而不是直接实例化具体仓库类。

视图与视图模型的解耦：通过构造函数注入，视图不再负责创建视图模型的实例，而是通过外部注入。

组合依赖：在应用程序启动时，通过一个组合根（Composition Root）来创建对象并注入依赖。

依赖注入容器

使用第三方依赖注入容器（如Unity和Ninject）可以简化依赖注入过程，提供生命周期管理、配置和依赖解析等功能。

单元测试

依赖注入使得单元测试变得更加容易，可以通过注入模拟（mock）依赖来隔离测试代码。

总结

依赖注入是一组强大的模式，它不是万能的，但在许多情况下都非常有用。通过依赖注入，我们可以创建更灵活、更易于维护和测试的代码。

[Dependency injection - Wikipedia](https://en.wikipedia.org/wiki/Dependency_injection" \l ":~:text=In%20software%20engineering%2C%20dependency%20injection,object%20is%20called%20a%20service.)

Dependency Injection-A Practical Introduction.pdf

项目跟踪，建立能反映项目及小组每个人工作的进度、里程碑、工作量的跟踪图或表，将其保存到每个小组选定的协作开发平台上，每周更新。