程序设计中级实践

大作业：重构“饿了么”项目四（SpringBoot）

具体要求

1. 修改教程中一些明显的错误和不规范的问题。

以下是部分问题，不是全部的问题，仅供参考。

(1) 级联修改/删除错误问题。如：修改/删除商品信息、修改送货地址等可能导致已完成的订单发生变化。

(2) 前后端接口没有身份/权限校验，导致用户可以绕过前端，直连后端进行越权操作。

2. 将面向对象设计原则应用于重构过程。

包括但不限于课程理论教学中学过以下设计原则：

(1) 高内聚，低耦合。在一个软件系统中，应当尽量保证模块的独立性，模块实现功能职责单一，越简单越好。这样有利于系统复用，并且大大减少模块之间的依赖，系统稳定性高，更易于维护。

(2) 面向抽象编程，即基于接口而非实现编程原则。参见：依赖倒置原则。

(3) 组合/聚合复用原则（Composite/Aggregation Reuse Principle，CARP）。即多用组合少用继承原则。简述：要尽量使用组合，而不是继承关系达到重用的目的。详细描述：要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次

才考虑使用继承关系来实现。

(4) 单一职责原则（SRP）。简述：设计目的单一的类。详细描述：一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分。

(5) 开闭原则（OCP）。简述：对拓展开放，对修改封闭。详细描述：当应用的需求改变时 ,在不修改软件实体的源代码或者二进制代码的前提下 ,可以扩展模块的功能，使其满足新的需求。

(6) 里氏替换原则（LSP）。简述：子类可以替换父类。详细描述：子类可以扩展父类的功能，但不能改变父类原有的功能。

(7) 接口隔离原则（ISP）。简述：一个类对另一个类的依赖应该建立在最小的接口上。详细描述：要为各个类建立它们需要的专用接口 ,而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

(8) 依赖倒置原则（DIP）。简述：要依赖于抽象，而不是具体实现（针对接口编程，而不是针对实现编程）。详细描述：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。

(9) 迪米特法则（LOD），又称最少知识原则（LKP）。简述：一个对象应当对其他对象有尽可能少的了解。详细描述：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用 ,可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

(10) 其它原则：DRY、KISS、YAGNI、ROT 等。

3. 前后端之间的接口不规范。

建议考虑使用 RESTful 风格重新设计接口。RESTful 参考文献：

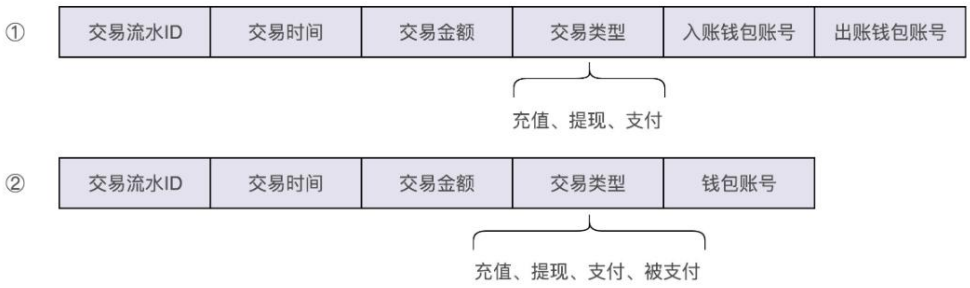
- 《理解 RESTful 架构》
  https://www.ruanyifeng.com/blog/2011/09/restful.html
- 《RESTful API 设计指南》
  https://ruanyifeng.com/blog/2014/05/restful_api.html
- 《RESTful API 最佳实践》
  https://www.ruanyifeng.com/blog/2018/10/restful-api-best-practices.html

4. 在饿了么（项目四）中，增加“虚拟钱包”模块（仅后端）。

(1) 钱包支持充值、提现、支付、查询余额、查询交易流水这五个核心的功能，其他比如冻结、透支、转赠等不常用的功能，我们暂不考虑。

(2) 我们可以把整个钱包系统的业务划分为两部分 ,其中一部分单纯跟应用内的虚拟钱包账户打交道，另一部分单纯跟银行账户打交道。我们基于此进行业务划分，给系统解耦。

(3) 交易流水如何记录？两种方案，推荐使用第一种方案。



(4) 传统的后端项目分为 Repository 层、Service 层、Controller 层。其中 ,Repository 层负责数据访问，Service 层负责业务逻辑，Controller 层负责暴露接口。本次我们需要在 Service 层中实现充血模型。实现中可以包括 VirtualWalletService 类、VirtualWallet 类和 VirtualWalletTransaction 类，分别代表服务类、钱包类和交易流水类。其它层仍然以贫血模型实现即可。

5. 重构“积分系统”模块。

(1) 需求分析

为提高用户粘性，电商平台一般都会有积分系统，请为电商平台设计一个积分系统，主要考虑的需求包括：

- 积分赚取和兑换规则
- 积分消费和兑换规则
- 积分及其明细查询

(2) 系统设计

系统设计主要有三方面的工作要做：接口设计、数据库设计和业务模型设计（也就是业务逻辑）。

(3) 功能模块（纵向切分）

电商平台中，可能会包括以下几个与积分相关的模块：

- 营销系统，负责各种促销活动。
- 业务系统，如订单系统、评论系统等，可以有机会获得积分。
- 换购商城、支付系统等，可以消费积分。

模块之间功能的划分方法主要有三种情况：

第一种划分方式是：积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护（增删改查），不划分到积分系统中，而是放到更上层的营销系统中。这样积分系统就会变得非常简单，只需要负责增加积分、减少积分、查询积分、查询积分明细等这几个工作。

比如，用户通过下订单赚取积分。订单系统通过异步发送消息或者同步调用接口的方式，告

知营销系统订单交易成功。营销系统根据拿到的订单信息 ,查询订单对应的积分兑换规则(兑换比例、有效期等），计算得到订单可兑换的积分数量，然后调用积分系统的接口给用户增加积分。

第二种划分方式是：积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护，分散在各个相关业务系统中，比如订单系统、评论系统、签到系统、换购商城、优惠券系统等。还是刚刚那个下订单赚取积分的例子，在这种情况下，用户下订单成功之后，订单系统根据商品对应的积分兑换比例，计算所能兑换的积分数量，然后直接调用积分系统给用户增加积分。

第三种划分方式是：所有的功能都划分到积分系统中，包括积分赚取渠道及兑换规则、消费渠道及兑换规则的管理和维护。还是同样的例子，用户下订单成功之后，订单系统直接告知积分系统订单交易成功，积分系统根据订单信息查询积分兑换规则，给用户增加积分。

(4) 模块与模块之间的交互关系

常见的系统之间的交互方式有两种：一种是同步接口调用，另一种是利用消息中间件异步调用。第一种方式简单直接，第二种方式的解耦效果更好。上下层系统之间的调用倾向于通过同步接口，同层之间的调用倾向于异步消息调用。比如，营销系统和积分系统是上下层关系，它们之间就比较推荐使用同步接口调用。

(5) 代码分层（横向切分）

大部分业务系统的开发都可以分为三层：Contoller 层、Service 层、Repository 层。其中 Controller 用来与前端用户界面交互，Service 层用来处理业务逻辑，Repository 层用来与数据库交互。这样做的好处是：

1. 分层能起到代码复用的作用

2. 分层能起到隔离变化的作用

3. 分层能起到隔离关注点的作用

4. 分层能提高代码的可测试性

5. 分层能应对系统的复杂性

(6) **数据对象 BO、VO、Entity**

Controller、Service、Repository 三层，每层都会定义相应的数据对象，它们分别是 VO（View Object）、BO（Business Object）、Entity，例如 UserVo、UserBo、UserEntity。VO、BO、Entity 三个类虽然代码重复，但功能语义不重复。可以使用继承或组合，解决代码重复问题。

6. 修订《软件需求规格说明书》

建议参考《UML 大战需求分析》，尤其是第 10 章和附录 1，使用正确规范的 UML 图（包括但不限于用例图、类图、包图、顺序图、活动图、状态机图等）进行面向对象分析（OOA）。

7. 根据新修订的《软件需求规格说明书》，设计单元测试。

要求支持自动化测试，根据接口准备测试数据。

提交材料

- 实验报告。包括更新后的项目文档和上述具体要求的实现情况。
- 项目代码。