第一、二章 软件工程概论

1.软件工程（名词解释）

(1)应用系统的、规范的、可量化的方法，来开发、运行和维护软件，即将工程应用到软件。
(2)对(1)中各种方法的研究。

2.从1950s—2000s之间的特点（简答）

1950s：科学计算；以机器为中心进行编程；像生产硬件一样生产软件。
1960s：业务应用（批量数据处理和事物计算）；软件不同于硬件；用软件工艺的方式生产软件。
1970s：结构化方法；瀑布模型；强调规则和纪律。它们奠定了软件工程的基础，是后续年代软件工程发展的支撑。
1980s：追求生产力最大化；现代结构化方法/面向对象编程广泛应用；重视过程的作用。
1990s：企业为中心的大规模软件系统开发；追求快速开发、可变更性和用户价值；web应用出现
2000s：大规模web应用；大量面向大众的web产品；追求快速开发、可变更性、用户价值和创新。

第三、四章 项目启动

1.团队结构：主程序员团队；民主团队；开放团队

如何管理团队：建立团队章程；持续成功；和谐沟通；避免团队杀手

2.质量保障有哪些措施？结合实验进行说明

(1)需求开发——需求评审和需求度量；
(2)体系结构——体系结构评审和集成测试(持续集成)；
(3)详细设计——详细设计评审、设计度量和集成测试(持续集成)；
(4)构造阶段——代码评审、代码度量、测试（测试驱动和持续集成）；
(5)测试阶段——测试、测试度量。

> 要及时的根据保障计划进行质量验证，质量验证的方法主要有评审、测试和质量度量三种。

3.配置管理有哪些活动？实验中是如何进行配置管理的

（1）标识配置项版本管理 （2）变更控制 （3）配置审计 （4）状态报告 （5）软件发布管理

> 在项目实践中，使用配置管理工具对项目进行配置管理，如SVN。

第五章 软件需求基础

1.需求（名词解释）

(1)用户为了解决问题或达到某些目标所需要的条件或能力；
(2)系统或系统部件为了满足合同、标准、规范或其他正式文档所规定的要求而需要具备的条件或能力；
(3)对(1)或(2)中的一个条件或一种能力的一种文档化表述。

2.区分需求的三个层次：业务需求，用户需求，系统级需求

【题型】给出一个实例，给出三个层次的例子；对给定的需求示例，判断其层次
【业务需求】——来自现实：系统建立的战略出发点，表现为高层次的目标（Objective），它描述了组织为什么要开发系统，为了满足用户的业务需求，需求工程师需要描述系统高层次的解决方案，定义系统应该具备的特性（Feature）
【用户需求】——来自现实：执行实际工作的用户对系统所能完成的具体任务的期望，描述了系统能够帮助用户做些什么。对所有的用户需求，都应该有充分的问题域知识作为背景支持
特性：模糊、不清晰； 多特性混杂； 多逻辑混杂
【系统级需求】——来自软件：用户对系统行为的期望，每个系统级需求反映了一次外界与系统的交互行为，或者系统的一个实现细节。描述了开发人员需要实现什么
将用户需求转化为系统需求的过程是一个复杂的过程：首先需要分析问题领域及其特性，从中发现问题域和计算机系统的共享知识，建立系统的知识模型；然后将用户需求部署到系统模型当中，即定义系列的系统行为，让它们联合起来实现用户需求，每一个系统行为即为一个系统需求。该过程就是需求工程当中最为重要的需求分析活动，又称建模与分析活动。

> 【示例】
> R1：在系统使用3个月后，销售额度应该提高20%（业务需求 - 为何开发系统）
> R2：系统要帮助收银员完成销售处理；（用户需求 - 帮助用户做什么）
> 系统特性SF1：管理VIP顾客信息，针对每一个系统特性，都可以建立一组用户需求。例如对SF1，每一条都是用户完成具体任务所需要的功能：
> UR1.1：客户经理可以使用系统添加、修改或者删除会员个人信息。（用户需求）
> UR1.2：收银员使用系统进行销售时会记录会员的购买信息。（用户需求）
> UR1.3：客户经理可以使用系统查看会员的个人信息和购买信息。（用户需求）
> UR1.4：客户经理可以使用系统查看所有会员的统计信息。（用户需求）
>
> R3：收银员输入购买的商品时，系统要显示该商品的描述、单价、数量和总价（系统级需求 - 系统怎么与外界交互）
> 对用户需求UR1.3，可以依据任务中的交互细节将之转化为系统级需求SR1.3.1 ~ SR1.3.4。
> SR1.3.1在接到客户经理的请求后，系统应该为客户经理提供所有会员的个人信息。（系统级需求）
> SR1.3.2在客户经理输入会员的客户编号时，系统要提供该会员的个人信息。（系统级需求）
> SR1.3.3在客户经理选定一个会员并申请查看购买信息时，系统要提供该会员的历史购买记录。
> SR1.3.4经理可以通过键盘输入客户编号，也可以通过读卡器输入客户编号。（系统级需求）
>
> ATM机：问题：营业厅人力成本过高，不吸引客户（业务需求）
> 问题域：存钱、取钱、转账（用户需求）

3.掌握需求的类型

【题型】对给定实例，给出不同类型的需求例子；对给定的需求示例，判定需求类型
【需求】项目需求、过程需求、系统需求（软件需求、硬件需求、其他需求）、不切实际的期望

> （1）项目需求（人的数量，计划书成本、时间）
> R5：项目的成本要控制在60万元人民币以下。
> R6：项目要在6个月内完成。
> （2）过程需求（人的分工、合作、方法、工具）
> R7：在开发中，开发者要提交软件需求规格说明文档、设计描述文档和测试报告。
> R8：项目要使用持续集成方法进行开发。
> （3）其他需求
> R9：系统要购买专用服务器，其规格不低于……
> R10：系统投入使用时，需要对用户进行1个星期的集中培训。
> （4）不切实际的期望
> R11：系统要分析会员的购买记录，预测该会员将来一周和一个月内会购买的商品；（技术上不可行）
> R12：系统要能够对每月的出入库以及销售行为进行标准的财务分析；（在有限的资源条件下不可

> 行）
> R13：在使用系统时，收银员必须要在2个小时内完成一个销售处理的所有操作。（超出了软件所能影响的问题域范围）
> R14：如果一个销售处理任务在2个小时内没有完成，系统要撤销该任务的所有已执行操作（正确）

【软件需求分类】功能需求、性能需求、质量属性、对外接口、约束、数据需求
（1）功能需求：和系统主要工作相关的需求，即在不考虑物理约束的情况下，用户希望系统所能够执行的活动，这些活动可以帮助用户完成任务。功能需求主要表现为系统和环境之间的行为交互。
最常见、最主要和最重要的需求；能够为用户带来业务价值的系统行为；最需要按照三个抽象层次进行展开；软件产品产生价值的基础

（2）性能需求：系统整体或系统组成部分应该拥有的性能特征，例如CPU使用率、内存使用率等。
需要进行专门模拟和测试

> 速度，系统完成任务的时间，例如PR1。PR1：所有的用户查询都必须在10秒内完成。
> 容量，系统所能存储的数据量，例如PR2。PR2：系统应该能够存储至少100万个销售信息。
> 吞吐量，系统在连续的时间内完成的事务数量，例如PR3。PR3：解释器每分钟应该至少解析5000条没有错误的语句。
> 负载，系统可以承载的并发工作量，例如PR4。PR4：系统应该允许50个营业服务器同时从集中服务器上进行数据的上传或下载。
> 实时性，严格的实时要求，例如PR5。PR5：监测到病人异常后，监控器必须在0.5秒内发出警报。
> PR6：98%的查询不能超过10秒。
> PR7：（最低标准）在200个用户并发时，系统不能崩溃；
> （一般标准）在200个用户并发时，系统应该在80%的时间内能正常工作；
> （理想标准）在200个用户并发时，系统应该能保持正常的工作状态。

（3）质量属性：系统完成工作的质量，即系统需要在一个“好的程度”上实现功能需求，

> 可靠性：在规格时间间隔内和规定条件下，系统或部件执行所要求能力的能力。
> QA1：在进行数据的下载和上传中，如果网络故障，系统不能出现故障。
> 可用性：软件系统在投入使用时可操作和可访问的程度或能实现其指定系统功能的概率。
> QA2：系统的可用性要达到98%。
> 安全性：软件阻止对其程序和数据进行未授权访问的能力，未授权的访问可能是有意，也可能是无意的。
> QA3：VIP顾客只能查看自己的个人信息和购买记录；收银员只能查看，不能修改、删除VIP顾客的信息。
> 可维护性：软件系统或部件能修改以排除故障、改进性能或其他属性或适应变更了的环境的容易程度，包括可修改性和可扩展性。
> QA4：如果系统要增加新的特价类型，要能够在2个人月内完成。
> 可移植性：系统或部件能从一种硬件或软件环境转换至另外一种环境的特性。
> QA5：集中服务器要能够在1人月内从Window 7操作系统更换到Solaris 10操作系统。
> 易用性：与用户使用软件所花费的努力及其对使用的评价相关的特性。
> QA6：使用系统1个月的收银员进行销售处理的效率要达到10件商品/分钟。

（4）对外接口：系统和环境中其他系统之间需要建立的接口，包括硬件接口、软件接口、数据库接口等
解系统和其他系统之间的软硬件接口
接口的用途，接口的输入输出，数据格式，命令格式，异常处理要求；用户界面

（5）约束：进行系统构造时需要遵守的约束，例如编程语言、硬件设施等，总体上限制了开发人员设计和构建系统时的选择范围

> 系统开发及运行的环境，包括目标机器、操作系统、网络环境、编程语言、数据库管理系统等
> C1：系统要使用Java语言进行开发。
> 问题域内的相关标准，包括法律法规、行业协定、企业规章等。
> 商业规则，用户在任务执行中的一些潜在规则也会限制开发人员设计和构建系统的选择范围

> R1：已过保质期的食品不能销售
> R2：顾客可以使用美元付款

（6）数据需求：
功能需求的补充，如果在功能需求部分明确定义了相关的数据结构，那么就不需要再行定义数据需求
数据需求是需要在数据库、文件或者其他介质中存储的数据描述，通常包括下列内容：各个功能使用的数据信息；使用频率；可访问性要求；数据实体及其关系；完整性约束；数据保持要求。

> 例如，连锁超市销售系统可以使用数据需求DR1和DR2。
> DR1：系统需要存储的数据实体及其关系为图6-14的内容。
> DR2：系统需要存储1年内的销售记录和退货记录。

第六章 需求分析方法

1.建立用例图

参与者是与开发的系统进行交互的用户或其他系统等角色
用例图中一个单一的参与者可以代表多个用户（或系统）
一个单一的用户（或系统）可能扮演多种角色
参与者不一定是人，例如，需要从当前系统获取信息的外部系统也是参与者
步骤：1）目标分析与解决方向确定 2）寻找参与者 3）寻找用例 4）细化用例

> 【示例1】
> ×××连锁商店是一家刚刚发展起来的小型连锁商店，其前身是一家独立的小百货门面店。
> 首先是随着商店规模的扩大，顾客量大幅增长，手工作业销售迟缓，顾客购物排队现象严重，导致流失客源。其次是商店的商品品种增多，无法准确掌握库存，商品积压、缺货和报废的现象上升明显。再次是商店面临的竞争比以前更大，希望在降低成本，吸引顾客，增强竞争力的同时，保持盈利水平。
>
> BR1：在系统使用6个月后，商品积压、缺货和报废的现象要减少50%
> BR2：在系统使用3个月后，销售人员工作效率提高50%
> BR3：在系统使用6个月后，运营成本要降低15%
> 范围：人力成本和库存成本，度量：检查平均员工数量和平均每10,000元销售额的库存成本
> BR4：在系统使用6个月后，销售额度要提高20%，最好情况：40%，最可能情况：20%，最坏情况：10%
>
> SF1：分析商品库存，发现可能的商品积压、缺货和报废现象
> SF2：根据市场变化调整销售的商品
> SF3：制定促销手段，处理积压商品
> SF4：与生产厂家联合进行商品促销
> SF5：制定促销手段进行销售竞争
> SF6：掌握员工变动和授权情况
> SF7：处理商品入库与出库
> SF8：发展会员，提高顾客回头率
> SF9：允许积分兑换商品和赠送吸引会员的礼品，提高会员满意度
> SF10：帮助收银员处理销售与退货任务
>
> 从上述特性可以发现涉及的用户类别：总经理，客户经理，收银员，管理员
> 总经理的目标有：产品调整（增删改产品信息），特价策略制定（增删改特价策略），赠送策略制定（增删改赠送策略），库存分析；（分析可能的商品积压）
> 客户经理的目标有：会员管理；（会员发展、礼品赠送），库存管理；（商品入库、出库和库存分析）
> 收银员的目标有：销售处理（销售），退货；（退货）
> 管理员的目标有：用户管理（增删改用户信息）

如果用例的粒度不合适就需要进行细化和调整。判断标准是：用例描述了为应对一个业务事件，由一个用户发起，并在一个连续时间段内完成，可以增加业务价值的任务。
不要将没有业务价值（而是技术实现需要）的事件作为用例（例如，登录（安全性需求）、输入/输入数据检查（数据需求或者业务规则）、连接数据库、网络传输等等）
不要将没有独立业务价值的单个操作（仅仅是技术实现上独立）作为用例，例如删除、增加、修改、保存

> 总经理的目标有：特价策略制定、赠送策略制定两个用例的业务目的、发起源和过程基本相同，仅仅是业务数据不同，所以可以合并为一个用例销售策略制定。
> 客户经理的目标有：会员管理用例有两个明显不同的业务事件，可以被细化为发展会员和礼品赠送2个更细粒度的用例。
> 客户经理的库存管理用例也有三个不同的业务目标：出库、入库和库存分析，所以也应该细化为三个用例商品出库、商品入库和库存分析，其中库存分析用例与总经理的库存分析用例相同。

【常见错误】
不要将用例细化为单个操作
不要将单个步骤细化为用例
不要将片面的一个方面细化为用例
不要将“登录”、“数据验证”、“连接数据库”等没有业务价值的内容作为用例

> 【示例2】
> 网上书店系统（OBS）是一个基于web的应用程序，允许用户浏览和购买网上产品。该应用程序支持网上购物车的概念，类似于其他网上零售商，如Amazon.com,。该系统的结账功能将集成信用卡交易处理以及内部计费系统。该系统还提供管理员视图，允许授权的员工查看和管理产品、用户和订单。
> 用户：购买、浏览
> 员工：查看用户、订单、产品管理
> 管理员：授权
> 信用卡：结账
> 内部计费：结账
> 注意：内部计费指的是单位内部，而不是系统内部

2.建立分析类图（概念类图 / 领域模型）

注意：与设计类图有所不同，分析类图关注现实世界问题域，而不是软件系统的内部构造机制；
类型、方法、可见性等复杂的软件构造细节不会在概念类图中，不允许出现与现实无关的内容
对象：标识（对象自治、对象请求协作）、状态（存储数据，如密码、名称……）、行为（利用数据做什么）
链接：对象间交互的路径（a拥有b的引用——a拥有b的可见性——a可以导航到b）
类：对象的集合
关联：潜在的链接抽象

要写出分析的步骤：
1）识别候选类（名词分析法）
2）确定概念类 （看是否满足既有状态又有行为）
既需要维持一定的状态，又需要依据状态表现一定的行为——确定为一个概念类
如只需要维护状态，不需要表现行为——其他概念类的属性
不需要维护状态，却需要表现行为——首先重新审视需求是否有遗漏，因为没有状态支持的对象无法表现行为；如果确定没有需求的遗漏，就需要剔除该候选类，并将行为转交给具备状态支持能力的其他概念类
既不需要维护状态，又不需要表现行为——应该被完全剔除
3）识别关联（文本中提取出“名词 + 动词 + 名词”的结构）
第一标准是满足需求的要求，第二标准是现实状况
4）识别重要属性

概念类图只需要用到四种关系线：
聚合关系不必可以使用，但是组合关系要适当的使用
继承关系、组合关系、聚合关系、普通关联

> 【示例1】
> 1、如果是会员，收银员输入客户编号（属性、无行为）
> 2、系统显示会员信息（就是会员），包括姓名（属性、无行为）与积分（属性、无行为）
> 3、收银员输入商品标识（商品属性、无行为）
> 4、系统记录并显示商品信息（有状态、有行为），商品信息包括商品标识、描述、数量、价格、特价（如果有商品特价策略的话）和本项商品总价（商品属性）
> 5、系统显示已购入的商品清单（有状态、有行为），商品清单包括商品标识、描述、数量、价格、特价、各项商品总价和所有品总价（商品属性）
> 收银员重复3-5步，直到完成所有商品的输入
> 6、收银员结束输入，系统计算并显示总价（存储在账单中，有行为），计算根据总额特价策略（有状态、有行为）进行
> 7、系统根据商品赠送策略和总额赠送策略计算并显示赠品清单（要），赠品清单包括各项赠品的标识、描述与数量（不要）
> 8、收银员请顾客（就是会员）支付账单（有状态、有行为）
> 9、顾客支付，收银员输入收取的现金数额（有属性、无行为、不要）
> 10、系统给出应找的余额（有属性、无行为、不要），收银员找零
> 11、收银员结束销售，系统记录销售信息（有状态、有行为）、商品清单、赠品清单和账单信息，并更新库存（有状态、有行为）
> 12、系统打印收据（根据需求，如果是一次性就无状态无行为，如果是丢了还可以打印就有状态有行为）
>
> 注意：一切看需求。
> （1）若商品ID必须符合标准，则ID有状态、有行为
> （2）若商品数量单位不同，则单位换算的职责交给数量，有状态、有行为
> （3）若商品价格按照国际汇率有不同定位，则价格有状态、有行为
>
> 【示例2】
> ATM系统通过显示屏，输入键盘（有数字和特殊输入按键），银行卡读卡器，存款插槽，收据打印机等与用户交互。客户使用ATM机存款，取款，余额查询，对账户的更新交由账户系统的一个接口来处理。安全系统将为每个客户分配一个PIN码和安全级别。每次事物执行之前都需要验证PIN码。将来，银行计划使用ATM机支持一些常规操作，例如使用地址和电话号码修改。
> 分析：显示器、按键、读卡器、存款插槽、收据打印机 不属于现实世界
> 客户：属性：PIN、地址、电话号码、安全级别
> 账户：余额
> 交易
>
> 【示例3】
>
> 1. 顾客向系统提起查询请求
> 2. 系统根据请求为顾客提供一个CD的推荐列表
> 3. 顾客在推荐列表中选定一个CD，然后要求查看更详细的信息
> 4. 系统为顾客提供选定CD的详细信息
> 5. 顾客购买选定CD.
> 6. 顾客离开
>
> 分析：
> 查询请求：有状态、有行为
> 顾客和CD：看戏球，不确定是否存储详细信息
> 推荐列表：有状态、有行为（增删改）

3.建立系统顺序图（交互图）

步骤：1）确定上下文环境 2）根据用例描述找到交互对象 3）按照用例描述中的流程顺序逐步添加消息同步消息、异步消息、返回消息
注意：
（1）异步消息的箭头无论是从用户到系统还是从系统到用户都是一样的
（2）opt标签表示可选；loop标签表示循环，要在旁边用［］内写循环条件；alt标签表示候选（基本上只会放一次返回消息），每一种可选分支之间要用虚线分割，而且在表示执行态的圆柱上面要写监护条件，放在［］里面。

4.建立状态图

状态: 一组观察到的情况，在一个给定的时间描述系统行为
状态转换: 从一个状态到另一个状态的转移
事件: 导致系统表现出可预测行为的事件
活动: 作为产生转换的结果而发生的过程
步骤：1）确定上下文环境 2）识别状态 3）建立状态转换 4）补充详细信息，完善状态图
注意：并不是所有的状态图都有开始态和结束态，开始态通常只有一个，结束态可以有多个

> 【示例】明确状态图的主体：用例UC1销售处理。
> 空闲状态（开始状态）：收银员已经登录和获得授权，但并没有请求开始销售工作的状态；
> 销售开始状态：开始一个新销售事务，系统开始执行一个销售任务的状态；
> 会员信息显示状态：输入了客户编号，系统显示该会员信息的状态；
> 商品信息显示状态：刚刚输入了一个物品项，显示该物品（和赠品）描述信息的状态；
> 列表显示状态：以列表方式显示所有已输入物品项（和赠品）信息的状态；
> 错误提示状态：输入信息错误的状态；
> 账单处理状态：输入结束，系统显示账单信息，收银员进行结帐处理的状态。
> 销售结束状态：更新信息，打印收据的状态。

第七章 需求文档化与验证

1.为什么建立需求规格说明？结合试验说明

（1）软件开发过程中，子任务与人员之间存在错综复杂的关系，存在大量的沟通和交流，所以要编写软件开发中要编写不同类型的文档，每种文档都是针对项目中需要广泛交流的内容。因为软件需求需要进行广泛交流，所以要把需求文档化。
（2）需求规格说明是在软件产品的角度以系统级需求列表的方式描述软件系统解决方案
用例侧重于交互流程，规格（解决方案）侧重于独立需求
用例以一次交互为基础，规格需求以一次交互中的软件系统处理细节为基础

2.对给定的需求规格说明示例，判定并修正其错误。

首先了解需求文档化要点：
1）技术文档写作要点（简洁，精确，易读，易修改）；
2）需求书写要点（使用用户术语，可验证，可行性）；
3）需求规格说明文档书写要点（充分利用标准的文档模板，保持所以内容位置得当；保持文档内的需求集具有完备性和一致性；为需求划分优先级）

> 【错误示例】
> 付款过程完成后，相关信息应被添加到日志文件
> 该系统应被构造成将来很容易添加新功能
> 计算购买的汽油的价格：该类汽油每加仑的价格乘以购买的加仑数（使用两位小数点表示加仑小数部分）
> 该系统一周7天，一天24小时都可用

3.给定的需求示例，设计功能测试用例

结合测试方法

第八章 软件设计基础

1.软件设计（名词解释）

a)为使一软件系统满足规定的需求而定义系统或部件的体系结构、部件、接口和其他特征的过程；
b)设计过程的结果。

2.软件设计的核心思想

复杂度控制
分解、抽象、层次性

第九、十章 软件体系结构设计与构建

1.体系结构概念

部件+连接件+配置
部件：承载系统主要功能，包括处理与数据
连接件：定义部件间的交互，是连接的抽象表示
配置：定义了部件和连接件之间的关联方式，将它们组织称系统的总体结构

2.体系结构的风格的优缺点

常见风格模式：

(1）主程序/子程序风格

部件：程序、函数、模块
连接件：它们之间的调用
控制从子程序层次结构顶部开始且向下移动
层次化分解：基于定义使用关系
单线程控制
隐含子系统结构：子程序通常合并成模块
层次化推理：子程序的正确性依赖于它调用的子程序的正确性
优点：流程清晰，易于理解；强控制性。
缺点：a.程序调用是一种强耦合的连接方式，非常依赖交互方的接口规格，使得系统难以修改和复用；
b.基于程序调用（声明 - 使用）关系的连接方式限制了各部件之间的数据交互，可能会使得不同部件使用隐含的共享数据交流，产生不必要的公共耦合，进而破坏它的“正确性”控制能力。
实现：功能分解、集中控制，每个构件一个模块实现，主要是单向依赖。使用utility或tools等基础模块

(2）面向对象式风格

部件: 对象或模块
连接件: 功能或调用（方法）
数据表示对于其他对象是隐藏的
对象负责保持数据表示的完整性（例如一些不变量）
每个对象都是自主代理
优点：内部实现的可修改性；易开发、易理解、易复用的结构组织。
缺点：接口的耦合性；标识的耦合性；副作用（Ex：A和B都修改C）。

适用于核心问题是识别和保护相关机构信息（数据）的应用
数据表示和相关操作封装在抽象数据类型，例: 抽象数据类型
实现：任务分解，（委托式）分散式控制，每个构件一个模块实现，使用接口将双向依赖转换为单向依赖
将每个构件分割为多个模块，以保证单向依赖，使用utility或tools等基础模块

(3）分层风格

上层调用下层，禁止逆向调用，跨层调用
部件: 通常是程序或对象的集合
连接件: 通常是有限可见度下的程序调用或方法调用
优点：设计机制清晰，易于理解；支持并行开发；更好的可复用性与内部可修改性。
缺点：交互协议难以修改；性能损失；难以确定层次数量和粒度。
应用：适用于包含不同类服务的应用，而且这些服务能够分层组织
尤其当应用可能在某些层改变，例如交互，通信，硬件，平台等
例子:分层通信协议，每一层在某种程度的抽象上提供了通信的基底，低层定义了低层次的交互最底层是硬件连接（物理层）
实现：关注点分离（每层逐次抽象），层间接口使用固定协议（固定控制），每层一或多个模块实现，单向依赖，层间数据传递建立专门模块，使用utility或tools等基础模块

(4）MVC风格

模型——视图——控制器
Model子系统被设计成不依赖任何View或Controller子系统
它们状态的改变会传播到View子系统
部件：Model部件负责维护领域知识和通知视图变化；View部件负责给用户显示信息和将用户手势发送给控制器；Controller改变模型的状态、将用户操作映射到模型更新、选择视图进行响应
连接件: 程序调用，消息，事件
优点：易开发性；视图和控制的可修改性；适宜于网络系统开发的特征。
缺点：复杂性；模型修改困难。
适用于以下应用:在运行时，用户界面的改变很容易且是可能的
用户界面的调整或移植不会影响该应用功能部分的设计和编码
例:Web 应用
实现：特定技术，通常专用于WEB：Model与Controller单向，Controller与View双向，Model与View双向
典型实现：View：JSP，HTML；Controller：Servlet；Model：JavaBean

3.体系结构设计的过程

分析关键需求和限制条件；
通过选择体系结构风格，确定顶层架构；
实现功能需求的逻辑映射；
通过构件的逐层设计从逻辑视角向物理视角的转化；
添加辅助构件；
完善构件之间接口的定义；
完善数据的定义。
迭代过程3-8

> 安全与网络分布：方案比较
> 方案一：哪个接口更多、更复杂？ 将RMI置于更少、更简单的层间接口
> 方案二：哪两层之间更内聚？将RMI置于更不内聚的层间接口
> 方案后果：更简洁、更易于修改
> 方案一：Mainui只包含登录

> 方案二：Mainui包含登录和主界面
> 关键区分：是否需要共同的主界面、界面一样

4.包的原则

最高原则：包与包之间不能有重复和冗余、复用发布等价原则、共同封闭原则、共同复用原则、无环依赖原则、稳定依赖原则、稳定抽象原则

（1）复用发布等价原则REP

（不要把很多用例放在一个包）为复用者聚合构件（类），单一的类通常不可复用，几个协作类组成一个包
一个包中的类应该形成一个可复用和可发布的模块，模块提供一致的功能，减少复用者的工作

（2）共同封闭原则CCP

最小化变化给程序员造成的影响
（不要把一个用例拆成多个包）当一个变化是必要的，如果该变化影响尽可能少的包，则对程序员是有利的，因为需要时间编译和链接以及重新生效
把具有相似闭包的类聚合在一起，包闭合了预期的变化
限制对一些包的修改，减少包的发布频率，减少程序员的工作

（3）共同复用原则CRP

根据公共复用给类分组，避免给用户带来不必要的依赖
常常导致细分包，获得更多更小更集中的包，减少复用者的工作
CCP和CRP原则相互排斥，例如它们不能同时被满足：CRP使得复用者的工作更简单，而CCP使维护者的生活更简单；CCP努力让包尽可能大，而CRP设法让包很小。项目的早期，架构师可能以CCP为主导来辅助开发和维护。架构稳定后，架构师可能重构包结构，为了外部复用者而最大化CRP。

（4）无环依赖原则ACP

包的依赖结构应是一个有向无环图
一块块地稳固和发布项目，以自上而下的层次结构组织包依赖
解决：分解包 / 抽象依赖

（5）稳定依赖原则SDP

例如：向下接口的依赖是不稳定依赖

（6）稳定抽象原则SAP

稳定包应是抽象包，不稳定包应是具体包
稳定包包含高层设计，抽象它们使得对扩展开发对修改关闭（OCP）。一些灵活性留在稳定的难以改变的包中。
对象创建：方案一: Spring方案,需要事先明确依赖关系
在程序启动时,第三方(初始化程序)一次性创建完成对象实例并set给上一层
Service：View——Logic；DAO: Logic——Data
方案二: 也适用于非层间的包依赖解除
构造一个第三方(Factory)负责创建具体对象实例
上层和下层都依赖于第三方，但第三方可以视作例外（即不影响层间独立关系）

5.体系结构构件之间接口的定义

根据分配的需求确定模块对外接口，如逻辑层接口根据界面的需求得到，数据层接口根据逻辑层调用得到
根据刺激与响应确定接口，依据详细规格明确接口内容（数据、返回值）
（1）通常情况下，VIEW的required接口可以直接作为相应Logic的Provided
（2）通常情况下，LOGIC的required接口需要分解为同层模块和不同Data的Provided
（3）Data一般没有层间依赖，接口通常来自于上一层的相应模块

> 【示例】saleLogic的其他接口来源：
> （1）其他被归类为saleLogic模块承担的需求，如退货用例
> （2）同层模块间的依赖
> 解除双向依赖：需要被抽象为一个独立接口和独立包，saleLogic再导入上面的包
> 独立接口和独立包也属于saleLogic模块最终承担的Provide接口，也需要转化为对Data的Required接口
> 在分析其他模块时逐步完善，例如进行礼品赠送时查看特定客户的销售记录、库存分析时查看一定时间段内的销售记录或特定商品的销售记录
> 在独立接口和独立包中定义有限制的领域类供外界使用，例如对象限制为查询接口，限制数据种类等等

6.体系结构开发集成测试用例：Stub和Driver

（1）依据模块接口建立桩程序Stub——为了完成程序的编译和连接而使用的暂时代码，对外模拟和代替承担模块接口的关键类，比真实程序简单的多，使用最为简单的逻辑

> 【示例】P177 SalesBLService_Stub

（2）编写驱动程序，在桩程序帮助下进行集成测试
View的测试比较特殊，其他层都需要增加Driver进行测试
可以基于Junit编写Driver；基于接口规格设计测试用例
开发View层时：需要logic的stub
开发logic层时：需要模仿view的driver，需要data的stub，需要模拟同层调用的driver和stub
开发data层时：需要模拟logic的driver

（3）持续集成：逐步编写各个模块内部程序，替换相应的桩程序
真实程序不仅实现业务逻辑，而且会使用其他模块的接口程序（真实程序或者桩程序）
开始：客户端: view driver, logic stub, data stub；服务器端: logic driver, data stub
进展：客户端: (逐步替换)driver,逐步替换logic stub, data stub；服务器端: logic driver, 逐步替换 data stub
最后联调：使用真实的客户端和服务器

第十一章 人机交互设计

1.可用性（名词解释）

易用性是人机交互中一个既重要又复杂的概念。它不仅关注人使用系统的过程，同时还关注系统对使用它的人所产生的作用，因为比较复杂，所以易用性不是一个单一的质量维度，而是多维度的质量属性。从易于度量的角度讲，易用性的常用维度包括：易学性、易记性、有效率、低出错率和主观满意度。

2.界面设计的注意事项及解释（至少5个）

【题型】例子违反了哪条界面设计原则
（1）不要暴露内部结构
例子：该设计明显暴露了软件结构，三个独立软件过程：创建、更新、解除
（2）展示细节——所见即所得
（3）常见界面类型：软件系统通常同时使用多种界面类型，以适应差异性的用户和任务。

3.精神模型，差异性

（1）精神模型就是用户进行人机交互时头脑中的任务模型
依据精神模型可以进行隐喻（Metaphor）设计：隐喻又被称为视觉隐喻，是视觉上的图像，但会被用户映射为业务事物。用户在识别图像时，会依据隐喻将控件功能与已知的熟悉事物联系起来，形成任务模型；隐喻本质上是在用户已有知识的基础上建立一组新的知识，实现界面视觉提示和系统功能之间的知觉联系。
（2）用户希望看到的+希望用户看到的：识别并添加哪些能够帮助用户完成任务的功能，任务的频率也要纳入考虑
常见错误：加入一些容易加入（例如正好是一个独立的软件过程）的功能，这会扰乱用户的精神模型，影响使用过程的顺利性
（3）新手用户：关注易学性，进行业务导航，尽量避免出错
专家用户：关注效率
熟练用户：在易学性和效率之间进行折中
好的人机交互应该为不同的用户群体提供差异化的交互机制。
既为新手用户提供易学性高的人机交互机制（图形界面）
又为专家用户提供效率高的人机交互机制（命令行、快捷方式、热键）

4.导航、反馈、协作式设计

（1）导航
主动将自己的产品和服务简明扼要地告诉用户，目的是为用户提供一个很好的完成任务的入口，好的导航会让这个入口非常符合人的精神模型。
全局结构按照任务模型将软件产品的功能组织起来，并区分不同的重要性和主题提供给不同的用户。全局结构常用的导航控件包括窗口、菜单、列表、快捷方式、热键等等。全局结构的设计主要以功能分层和任务交互过程为主要依据。
局部结构通过安排界面布局细节，制造视觉上的线索来给用户提供导航。局部结构常用的导航控件包括可视化控件布局与组合、按钮设置、文本颜色或字体大小等等。局部结构的设计主要以用户关注的任务细节为主要依据。

（2）反馈
让用户能够意识到行为的结果，但不能打断用户工作时的意识流
用户喜欢较短的响应时间；较长的响应时间（>15秒）具有破坏性；
用户会根据响应时间的变化调整自己的工作方式；
较短的响应时间导致了较短的用户思考时间；较快的节奏可能会提高效率，但也会增加出错率；
根据任务选择适当的响应时间：打字、光标移动、鼠标定位：50～150毫秒；简单频繁的任务：1秒；普通的任务：2～4秒；复杂的任务：8～12秒
响应时间适度的变化是可接受的；意外延迟可能具有破坏性；经验测试有助于设置适当的响应时间。

（3）协作式设计
A.简洁设计（摘要图片优于文字描述）
列举、隐藏、赋予（标签、图标等线索暗示）
B.一致性设计（确认与删除键相对位置不一致）
C.低出错率设计（用具体的指导来提示用户出错）
限制输入：列表、可选框等选择性组件代替输入框；按钮代替命令行；限定输入：类型，范围，格式...
限制范围：简单化单步操作
辅助：事前提示；事后检查；随时可以撤销
D.易记性设计
减少短期记忆负担；使用逐层递进的方式展示信息；使用直观的快捷方式。重新认知比记忆更容易；设置有意义的缺省值，可以帮助用户减少记忆负担。

第12章 详细设计概述

中层设计＋低层设计：实现所有功能性＋非功能性需求

1.详细设计的出发点

需求开发的结果（需求规格说明和需求分析模型）和软件体系结构的结果（软件体系结构设计方案与原型）

> 明确职责建立静态模型（设计类图），明确协作建立动态模型（详细顺序图）
> GRASP（General Responsibility Assignment Software Patterns）
>
> ## （1）信息专家模式
>
> 基本的职责分配原则之一，把职责分配给具有完成该职责所需信息的那个类
> 例如：总价（委托）——数目——单价——促销策略：耦合没有增加
> 总价（得到）——数目、单价、促销策略：增加了耦合（总价知道太多）
> 优点：促进低耦合、高内聚、维护封装
>
> ## （2）控制器
>
> 处理外部事件（用户和系统时钟发生的外部交互）
> 核心思想：解耦
> 不要界面直接调用代码，也不要代码直接调用界面，把系统 / 人 / 用例作为controller
>
> ## （3）创建者模式
>
> 根据潜在创建者类和被示例话类之间的关系决定哪个类创建实例
> 当满足以下的条件时，B创建A：B“包含”A，或者B组合A；B直接使用A；B拥有A的初始化数据；B记录A
>
> ## （4）纯虚构
>
> 软件特有的ui、DAO、RMI、文件读写、复杂行为、设计模式、复杂数据结构、界面与逻辑层的model，不属于现实世界
> 作用：保持代码可复用性，高内聚、低耦合

2.职责分配

通过职责建立静态模型：面向对象分解中，系统是由很多对象组成的。对象各自完成相应的职责，从而协作完成一个大的职责。类的职责主要有两部分构成：属性职责和方法职责。类与类之间也不是孤立存在的，它们之间存在一定的关系。关系表达了相应职责的划分和组合。它们的强弱顺序为：依赖<关联<聚合<组合<继承。

3.协作

根据协作建立动态模型：（1）从小到大，将对象的小职责聚合形成大职责；
（2）大到小，将大职责分配给各个小对象。通过这两种方法，共同完成对协作的抽象。

4.控制风格

为了完成某一个大的职责，需要对职责的分配做很多决策。控制风格决定了决策由谁来做和怎么做决策
分散式：所有系统行为在对象网络中广泛传播
集中式：少数控制器记录所有系统行为的逻辑
委托式（授权式）：决策分布在对象网络中，一些控制器作主要决策

5.建立设计类图或详细顺序图

抽象类的职责->抽象类之间的关系->添加辅助类
辅助类：接口类、记录类（数据类）、 启动类、控制器类、实现数据类型的类、容器类

6.协作的测试

MockObject

第13章 详细设计中的模块化与信息隐藏

1.耦合与内聚（名词解释）

（1）耦合

描述的是两个模块之间的关系的复杂程度。
耦合根据其耦合性由高到低分为几个级别：模块耦合性越高，模块的划分越差，越不利于软件的变更和重用。1、2、3不可接受，4、5可以被接受，6最理想
（1）内容耦合（一个模块直接修改另一个模块的内容，如GOTO语句；某些语言机制支持直接更改另一个模块的代码；改变另一个模块的内部数据）
（2）公共耦合（全局变量，模块间共享全局数据，例子：全局变量）
（3）重复耦合（模块间有重复代码）
（4）控制耦合（一个模块给另一个模块传递控制信息，如“显式星期天”）
（5）印记耦合（共享数据结构却只是用了一个部分）
（6）数据耦合（模块间传参只传需要的数据，最理想）

（2）内聚

表达的是一个模块内部的联系的紧密性。
内聚性由高到低分为：内聚性越高越好，越低越不易实现变更和重用，3、4、5等价，1、2最好，6、7不能接受。
（1）信息内聚（模块进行许多操作，各自有各自的入口点，每个操作代码相对独立，而且所有操作都在相同的数据结构上进行：如栈）
（2）功能内聚（只执行一个操作或达到一个目的）
（3）通信内聚（对相同数据执行不同的操作：查书名、查书作者、查书出版商）
（4）过程内聚（与步骤有关：守门员传球给后卫、后卫传给中场球员、中场球员传给前锋）
（5）时间内聚（与时间有关：起床、刷牙、洗脸、吃早餐）
（6）逻辑内聚（一系列可替换的操作：如坐车、坐飞机）
（7）偶然内聚（多个不相关的操作：修车、烤面包、遛狗、看电影）

> 【题型】对实例，说明它们之间的耦合程度与内聚，给出理由。书上P226习题

2.信息隐藏基本思想

每个模块都隐藏一个重要的设计决策——职责。职责体现为模块对外的一份契约，并且在这份契约之下隐藏的只有这个模块知道的决策或者说秘密，决策实现的细节仅自己知道。
模块的信息隐藏：模块的秘密（容易变更的地方）：根据需求分配的职责、内部实现机制。

> 【题型】对实例，说明其信息隐藏程度好坏。教材222页
>
> 【项目实践】耦合的处理?
> 分层风格：仅程序调用与简单数据传递
> 包设计：消除重复
> 分包：接口最小化
> 创建者模式：不增加新的耦合
> 控制者模式：解除View与Logical的直接耦合

> 内聚的处理?
> 分层：层间职责分配高内聚，层内分包高内聚
> 信息专家模式
> 控制器与委托式控制风格
>
> 信息隐藏处理?
> 分层与分包：消除职责重复、最小化接口：View独立？数据库连接独立?
> 模块信息隐藏：模块需求分配与接口定义
> 类信息隐藏：协作设计，接口定义
> 变化设计？分层风格、RMI

第14章 详细设计中面向对象方法下的模块化

模块化的原则，教材229页

1)全局变量是有害的
2)显式（代码清晰，不进行不必要的隐藏，但可能与可修改性冲突）
3)不要有代码重复
4)针对接口编程
5)迪米特法则（不要跟陌生人说话）
6)接口分离原则（ISP）
7)Liskov替换原则(LSP:子类型可替换掉基类型，一棵满足ISP的继承树耦合度是1)
8)使用组合代替继承（适用于不符合LSP但想要进行代码复用）
9)单一职责原则
其中1、2、3针对降低隐式耦合，4、5、6针对降低访问耦合，7、8针对降低继承耦合，9和信息专家原则用于提高继承内聚，亦可以看作是降低继承耦合之用。
注1：级联调用的问题在：难以应对变更 + 代码难理解。应对的方法：采用委托式调用。
注2：何时接口分离：出现于有一个负责各种交互的公共接口与各个具体类直接存在使用关系，使得权限混乱，可能造成不必要的麻烦。解决方案：保留大接口并提取出几个小借口，通过具体类使用小接口，再用大接口实现小接口即可。

> 【题型】对给定的示例，发现其所违反的原则，并进行修正。
> 练习题 / 教材243页习题 / 教材258页习题

第15章 详细设计中面向对象方法下的信息隐藏

1.信息隐藏的含义

需要隐藏的两种常见设计决策
需求（模块说明的主要秘密）与实现——暴露外部表现，封装内部结构
实现机制变更（模块说明的次要秘密）——暴露稳定抽象接口，封装具体实现细节
面向对象机制
封装：封装类的职责，隐藏职责的实现 + 预计将要发生的变更
抽象类（接口）/继承（实现）：抽象它的接口，并隐藏其内部实现

2.封装

（1）含义1：数据与行为集中在一起
含义2：接口与实现相分离：一个类可以分成两个部分，接口和实现
接口是类的外部表现，对外公开；实现是类的内部结构，内部隐藏
（2）封装的初始观点：把数据（内部结构）隐藏在抽象数据类型内

新观点（信息隐藏）：隐藏任何东西：数据与行为、复杂内部结构、其他对象、子类型信息、潜在变更
（3）封装数据与行为：除非（直接或间接）为满足需求（类型需要），不要将操作设置为public。类型需要的操作：为了满足用户任务而需要对象在对外协作中公开的方法，例如下图的4个操作（属于后一个对象，为满足计算商品总价的任务）
除非（直接或间接）为满足需求（类型需要），不要为属性定义getX方法和setX方法，更不要将其定义为public。例如上一示例中的getPrice()
（4）封装结构：不要暴露内部的复杂数据结构，经验表明复杂数据结构是易于发生修改的。例如暴露了内部使用List数据结构。
改进：Iterator模式（所有涉及到集合类型的操作都可能会出现此问题）
（5）封装其他对象：委托而不是提供自己拥有的其他对象的引用
除非Client对象已经拥有了该其他对象的引用，这时返回其引用不会增加总体设计的复杂度
可以保证Sales只需要关联SalesList，不需要关联SalesLineItem和Commodity；从整个设计来讲，Sales不需要知道SalesList里面存储的是SalesLineItem的集合，更不需要知道SalesLineItem使用了Commodity类型
（6）封装子类（LSP：子类必须能够替换他们的基类）
（7）封装潜在变更：识别应用中可能发生变化的部分，将其与不变的内容分离开来
封装独立出来的潜在变化部分，这样就可以在不影响不变部分的情况下进行修改或扩展（ DIP 和OCP）

3.开闭原则OCP

对扩展开放，对修改封闭
违反了OCP原则的典型标志：出现了switch或者if-else
分支让程序增加复杂度，修改时容易产生新错误（特例：创建）

4.依赖倒置原则DIP

（与工厂结合紧密，解决new的创建问题）
I. 高层模块不应依赖底层模块，两者都应依赖抽象
II. 抽象不应依赖细节，细节应依赖抽象
使用抽象类（继承）机制倒置依赖

> 示例：A依赖于B：B不是抽象类，所以A依赖于具体，而不是抽象，如果需要变更B的行为，就会影响到A
> 添加抽象类BI，让 B实现（继承）BI：A依赖于BI，B依赖于BI，BI是抽象类，所以是依赖于抽象，BI比较稳定，如果B发生变更，可以通过为BI扩展新的实现（子类型）来满足

第16章 设计模式

1.如何实现可修改性、可扩展性、灵活性

教材263页
需要进行接口和实现的分离：通过接口和实现该接口的类；通过子类继承父类
注意：继承关系（A + B）可能使得灵活性下降，因为父类接口的变化会影响子类，这时可以通过组合关系来解决。

利用抽象类机制实现可修改性和可扩展性：只要方法的接口保持不变，方法的实现代码是比较容易修改的，不会产生连锁反应。通过简单修改创建新类的代码，就可以相当容易地做到扩展新的需求（不用修改大量与类方法调用相关的代码。
利用委托机制实现灵活性：继承的缺陷：一旦一个对象被创建完成，它的类型就无法改变，这使得单纯利用继承机制无法实现灵活性（类型的动态改变）。利用组合（委托）机制可以解决这个问题

2.策略模式

设计分析：
1）首先，可以把上下文和策略分割为不同的类实现不同的职责。上下文Context类负责通过执行策略实现自己职责；而策略类Strategy只负责复杂策略的实现。
2）其次，上下文类和策略类之间的关系是用组合代替继承。
3）最后，各种策略则在具体策略类(ConcreteStrategy)中提供，而向上下文类提供统一的策略接口。（客户通常会创建一个ConcreteStrategy对象，然后传递给Context来灵活配置Strategy接口的具体实现）
应用场景：
需要定义同一个行为的多种变体（可以是一个类的行为，也可是多个类的行为）
一个类定义了很多行为，而且通过switch语句进行选择。
相关原则：SRP，OCP+LSP+DIP，使用组合
缺点：代码难以理解（违反显式原则）；需要考虑concreteStrategy由谁来创建的问题

可修改性、可扩展性
符合DIP原则的抽象类（继承）机制：通过定义抽象类与继承（实现）它的子类强制性地做到：接口与实现的分离，进而实现上述质量；强制性地使用抽象类起到接口的作用，强制性地使用子类起到实现的作用
灵活性：组合（委托）机制，动态调整所委托的类，实现灵活性

3.抽象工厂模式

学会区分简单工厂，工厂方法，和抽象工厂
简单工厂：使用一个工厂对象，通过if - else等方式实例化需要的对象
工厂方法：一个抽象方法creator（可以在原来的类中），使用子类继承creator所在的类通过实现creator方法来实例化需要的对象（实例化推迟到子类）。
抽象工厂：为应对灵活性要求，提供2套接口：一是表现出稳定的工厂行为的工厂接口，二是表现出稳定产品行为的产品接口。从而，实现了工厂多态和产品多态。
需要的是产品组合。有一个抽象工厂，该抽象工厂有所有种类产品的create()，
不同的产品组合拥有不同的具体工厂（继承抽象工厂，实现所有的create()）.
（1）相关原则：不要重复，封装，OCP + LSP+DIP。可能违法了显式原则。
（2）应用场景：抽象工厂模式可以帮助系统独立于如何对产品的创建、构成、表现。
抽象工厂模式可以让系统灵活配置拥有某多个产品族中的某一个。
一个产品族的产品应该被一起使用，抽象工厂模式可以强调这个限制。
如果你想提供一个产品的库，抽象工厂模式可以帮助暴露该库的接口，而不是实现。
（3）应用注意点：隔离了客户和具体实现。客户可见的都是抽象的接口。
使得对产品的配置变得更加灵活。
可以使得产品之间有一定一致性。同一类产品可以很容易一起使用。
但是限制是对于新的产品的类型的支持是比较困难。抽象工厂的接口一旦定义好，就不容易变更了。
而这个场景的“代价”，或者是“限制”，是一个工厂中具体产品的种类是稳定的。

4.单件模式

为了实现只创建一个对象，首先要让类的构造方法变为私有的；然后只能通过getInstance方法获得Singleton类型的对象的引用。
相关原则：封装，而且使用了全局变量

5.迭代器模式

通过Aggregate创建Iterator
对于遍历这件事情，主要有2个行为：1）是否有下一个元素；2）得到下一个元素。所以，我们设计迭代器接口hasNext()和next()，分别对应前面2个行为。有了这两个接口，就可以完成遍历操作。迭代器提供的方法只提供了对集合的访问的方法，却屏蔽了对集合修改的方法，这样就对我们把集合作为参数可以做到对集合的“值传递”的效果。

体现的原则：针对接口编程；SRP; ISP（接口分离）；OCP + LSP + DIP
应用：在同一个集合上进行多种方式的遍历

> 【题型】给出场景，应用设计模式写出代码
> 【题型】给出代码，要求用设计模式改写
>
> 信息隐藏——策略模式、单件模式
> 对象创建——抽象工厂、单件模式
> 接口编程——迭代器模式
>
> 策略模式——减少耦合、依赖倒置
> 抽象工厂——职责抽象、接口重用
> 单件模式——职责抽象
> 迭代器模式——减少耦合、依赖倒置

第17、18章 软件构造和代码设计

1.构造包含的活动

详细设计；编程；测试；调试；代码评审；集成与构建；构造管理。

2.名词解释

（1）重构：修改软件系统的严谨方法，它在不改变代码外部表现的情况下改进其内部结构
（2）测试驱动开发：测试驱动开发要求程序员在编写一段代码之前，优先完成该段代码的测试代码。测试代码之后，程序员再编写程序代码，并在编程中重复执行测试代码，以验证程序代码的正确性。
（3）结对编程：两个程序员挨着坐在一起，共同协作进行软件构造活动

3.给定代码段示例，对其进行改进或发现其中的问题

复杂决策：使用有意义的名称封装复杂决策，例如对于决策“ If((id>0) && (id<=MAX_ID))”，可以封装为“If (isIdValid(id))”，方法isIdValid(id)的内容为 “return ((id>0) && (id<=MAX_ID))”
数据使用
（1）不要将变量应用于与命名不相符的目的。例如使用变量total表示销售的总价，而不是临时客串for循环的计数器。
（2）不要将单个变量用于多个目的。在代码的前半部分使用total表示销售总价，在代码后半部分不再需要“销售总价”信息时再用total客串for循环的计数器也是不允许的。
（3）限制全局变量的使用，如果不得不使用全局变量，就明确注释全局变量的声明和使用处。
（4）不要使用突兀的数字与字符，例如15（天）、“MALE”等，要将它们定义为常量或变量后使用。
明确依赖关系：类之间模糊的依赖关系会影响到代码的理解与修改，非常容易导致修改时产生未预期的连锁反应。对于这些模糊的依赖关系，需要进行明确的注释

4.契约式设计

契约式设计又称断言式设计，它的基本思想是：如果一个函数或方法，在前置条件满足的情况下开始执行，完成后能够满足后置条件，那么这个函数就是正确、可靠的。见书上示例P308
（1）异常
（2）断言：assert Expression1(: Expression2)
Expression1 是一个布尔表达式，在契约式设计中可以将其设置为前置条件或者后置条件；Expression2是一个值，各种常见类型都可以；
如果Expression1 为true，断言不影响程序执行；如果Expression1 为false，断言抛出AssertionError异常，如果存在Expression2 就使用它作为参数构造AssertionError。

5.防御式编程

优点：显著提高可靠性
防御式编程的基本思想是：在一个方法与其他方法、操作系统、硬件等外界环境交互时，不能确保外界都是正确的，所以要在外界发生错误时，保护方法内部不受损害。与契约式设计有共同点，又有很大的差异。共同点是他们都要检查输入参数的有效性。差异性是防御式编程将所有与外界的交互都纳入防御范围，如用户输入的有效性、待读写文件的有效性、调用的其他方法返回值的有效值……
编程方式：异常与断言。

> 输入参数是否合法?
> 用户输入是否有效?
> 外部文件是否存在?
> 对其他对象的引用是否为NULL?
> 其他对象是否已初始化?
> 其他对象的某个方法是否已执行?
> 其他对象的返回值是否正确?
> 数据库系统连接是否正常?
> 网络连接是否正常?
> 网络接收的信息是否有效?

6.表驱动

对于特别复杂的决策，可以将其包装为决策表，然后用使用表驱动编程的方式加以解决。示例见书上P307

7.单元测试用例的设计

MockObject创建桩程序

第19章 软件测试

1.白盒测试和黑盒测试的常见方法，并比较优缺点【必考】

（1）黑盒测试：基于规格的技术，是把测试对象看做一个黑盒子，完全基于输入和输出数据来判定测试对象的正确性，测试使用测试对象的规格说明来设计输入和输出数据。
常见方法：等价类划分（把输入按照等价类划分，包括有效和无效）；边界值分析；决策表；状态转换。
（2）白盒测试：基于代码的技术，将测试对象看成透明的，不关心测试对象的规格，而是按照测试对象内部的程序结构来设计测试用例进行测试工作。

2.能解释并区别白盒测试三种不同的方法【必考】

语句覆盖、分支覆盖和路径覆盖。教材329页
语句覆盖：确保被测试对象的每一行程序代码都至少执行一次
条件覆盖：确保程序中每个判断的每个结果都至少满足一次
路径覆盖：确保程序中每条独立的执行路径都至少执行一次

> 【题型】给出一个场景，判断应该使用哪种测试方法，如何去写（JUnit基本使用方法）
> 给出功能需求——写功能测试用例
> 给出设计图——写集成测试用例，Stub和Driver
> 给出方法描述——写单元测试用例，MockObject

第20、21章 软件交付、软件维护与演化

1.软件维护的重要性

（1）由于会出现新的需求，如不维护软件将减小甚至失去服务用户的作用。
（2）随着软件产品的生命周期越来越长，在软件生存期内外界环境发生变化的可能性越来越大，因此，软件经常需要修改以适应外界环境的改变
（3）软件产品或多或少的会有缺陷，当缺陷暴露出来时，必须予以及时的解决

2.开发可维护软件的方法

（1）考虑软件的可变性：分析需求易变性、为变更进行设计
（2）为降低维护困难而开发：编写详细的技术文档并保持及时更新、保证代码可读性、维护需求跟踪链、维护回归测试基线

3.演化式生命周期模型

初步开发—演化—服务—逐步淘汰—停止

（1）初步开发

初始开发阶段按照传统的软件开发方式完成第一个版本的软件产品开发。第一版的软件产品可以实现全部需求，也可以（通常是）只包含部分需求——对用户来说非常重要和紧急的最高优先级需求。

（2）演化

可能会有预先安排的需求增量，也可能完全是对变更请求的处理，它们的共同点都是保持软件产品的持续增值，让软件产品能够满足用户越来越多的需要，实现更大的业务价值。
总的来说，该阶段可能的演化增量有：预先安排的需求增量；因为问题变化或者环境变化产生的变更请求；修正已有的缺陷；随着用户与开发者之间越来越相互熟悉对方领域而新增加的需求。
演化阶段的软件产品要具备两个特征：(1) 软件产品具有较好的可演化性。一个软件产品在演化过程中复杂性会逐渐增高，可演化性会逐渐降低直至无法继续演化。演化阶段的软件产品虽然其可演化性低于初始开发阶段的软件产品，但是还没有到达无法演化的地步，还具有较好的可演化性。(2) 软件产品能够帮助用户实现较好的业务价值。只有这样，用户才会继续需要该产品，并持续提供资金支持。
如果在演化过程中，一个软件产品开始不满足第（2）条特征，那么该产品就会提前进入停止阶段。如果软件产品满足第（2）条的同时不满足第（1）条特征，那么该产品就会进入服务阶段。如果开发团队因为竞争产品的出现或者其他市场考虑，也可以让同时满足上面两条特征的软件产品提前进入服务阶段。

（3）服务

服务阶段的软件产品不再持续的增加自己的价值，而只是周期性的修正已有的缺陷。
服务阶段的产品还仍然被用户使用，因为它仍然能够给用户提供一定的业务价值，所以开发团队仍然需要修正已有缺陷或者进行一些低程度的需求增量，保证用户的正常使用。

（4）逐步淘汰

在逐步淘汰阶段，开发者已经不再提供软件产品的任何服务，也即不再继续维护该软件。虽然在开发者看来软件的生命周期已经结束，但是用户可能会继续使用处于该阶段的软件产品，因为它们仍然能够帮助用户实现一定的业务价值。只是用户在使用软件时必须要容忍软件产品中的各种不便，包括仍然存在的缺陷和对新环境的不适应。对于该阶段的产品，开发者需要考虑该产品是否可以作为有用的遗留资源用于新软件的开发，用户需要考虑如何更换新的软件产品并转移已有的业务数据。

（5）停止

一个软件正式退出使用状态之后就进行停止状态。开发者不再进行维护，用户也不再使用。

4.用户文档、系统文档

（1）用户文档：这里的文档支持是指为用户编写参考指南或者操作教程，常见的如用户使用手册、联机帮助文档等，统称为用户文档。
文档内容的组织应当支持其使用模式，常见的是指导模式和参考模式两种。指导模式根据用户的任务组织程序规程，相关的软件任务组织在相同的章节或主题。指导模式要先描述简单的、共性的任务，然后再以其为基础组织更加复杂的任务描述。
参考模式按照方便随机访问独立信息单元的方式组织内容。例如，按字母顺序排列软件的命令或错误消息列表。如果文档需要同时包含两种模式，就需要将其清楚地区分成不同的章节或主题，或者在同一个章节或主题内区分为不同的格式。

（2）系统文档：更注重系统维护方面的内容，例如系统性能调整、访问权限控制、常见故障解决等等。因此，系统管理员文档需要详细介绍软硬件的配置方式、网络连接方式、安全验证与访问授权方法、备份与容灾方法、部件替换方法等等。

5.逆向工程、再工程

（1）逆向工程：分析目标系统，标识系统的部件及其交互关系，并且使用其它形式或者更高层的抽象创建系统表现的过程。逆向工程的基本原理是抽取软件系统的需求与设计而隐藏实现细节，然后在需求与设计的层次上描述软件系统，以建立对系统更加准确和清晰的理解。
（2）再工程：检查和改造一个目标系统，用新的模式式及其实现复原该目标系统。目的是对遗留软件系统进行分析和重新开发，以便进一步利用新技术来改善系统或促进现存系统的再利用。
主要包括：改进人们对软件的理解；改进软件自身，通常是提高其可维护性、可复用性和可演化性。
常见具体活动有：重新文档化；重组系统的结构；将系统转换为更新的编程语言；修改数据的结构组织。

第22、23章 软件开发过程模型

1.软件生命周期模型

需求工程—软件设计—软件实现—软件测试—软件交付—软件维护

2.解释与比较不同的软件过程模型

【题型】对给定场景，判断适用的开发过程模型（要求、特征描述、优点、缺点）

（1）构建-修复模型

特征：a.没有对开发过程进行规范和组织，因此一旦开发过程超出个人控制能力，就会导致开发过程无法有效进行而失败。
b.对需求的真实性没有进行分析
c.没有考虑软件结构的质量，导致结构在修改中越来越糟，直至无法修改
d.没有考虑测试和程序的可维护性，也没有任何文档，导致难以维护
适用场景：软件规模很小，只需要几百行程序，其开发复杂度是个人能力能够胜任的；软件对质量的要求不高，即使出错也无所谓；只关注开发活动，对后期维护的要求不高，甚至不需要进行维护。

（2）瀑布模型（文档驱动）

特征：a.对于软件开发活动有明确阶段划分
b.每个阶段的结果都必须验证，使得该模型是“文档驱动”
优点：清晰的阶段划分有利于开发者以关注点分离的方式更好的进行复杂软件项目的开发。
缺点：a.对文档期望过高；
b.对开发活动的线性顺序假设（线性顺序与迭代相反）；
c.客户、用户参与度不够（需求被限制在一个时间段）
d.里程碑粒度过粗（软件复杂使得每个阶段时间长，无法尽早发现缺陷）

适用：需求非常成熟、稳定，没有不确定的内容，也不会发生变化；所需的技术成熟、可靠，没有不确定的技术难点，也没有开发人员不熟悉的技术问题；复杂度适中，不至于产生太大的文档负担和过粗的里程碑。

（3）增量迭代模型（需求驱动）与演化模型互为补充

特征：项目开始时，通过系统需求开发和核心体系结构设计活动完成项目的前景范围的界定，然后将后续开发活动组织为多个迭代、并行的瀑布式开发活动。第一个迭代完成的往往是核心工作，满足基本需求，后续迭代完成附带特性。
优点：a.更符合软件开发实践
b.并行开发可以缩短产品开发时间
c.渐进交付可以加强用户反馈，降低开发风险。
缺点：a.要求高：软件需要有开放式的体系结构
b.不确定性太多，导致难以在项目开始就确定前景和范围
适用：比较成熟和稳定的领域

（4）演化模型（需求驱动）

特点：更好地应对需求变更，更适用于需求变更比较频繁或不确定性较多的领域。将软件开发组织为多个迭代、并行的瀑布式开发活动。是迭代 + 并行 + 渐进
优点：a.适用于需求变更频繁或者不确定性多的系统的开发
b.并行开发可以缩短开发时间
c.加强用户反馈，降低开发风险
缺点：a.无法在项目早起确定项目范围
b.后续迭代活动以前导迭代为基础，容易使后续迭代忽略分析与设计。退化为构建 - 修复方式。
适用：不稳定领域的大规模软件系统

（5）原型模型（需求驱动）

特点：大量使用抛弃式原型（抛弃式原型：通过模拟“未来”的产品，将“未来”的知识置于“现在”进行推敲，解决不确定性）。
优点：a.加强与客户、用户的交流，更好的产品满意度
b.适用于不确定性大的新颖领域
缺点：a.成本较高，有耗尽时间和费用的风险
b.有人不舍得抛弃“抛弃式原型”，使得较差代码进入产品，降低质量
适用：有着大量不确定性的新颖领域

（6）螺旋模型（风险驱动）

特点：基本思想是尽早解决比较高的风险，如果有些问题实在无法解决，那么早发现比项目结束时再发现要好，至少损失要小得多。风险是指因为不确定性（对未来知识了解有限）而可能给项目带来损失的情况，原型能够澄清不确定性，所以原型能够解决风险。
迭代与瀑布的结合：开发阶段是瀑布式的，风险分析是迭代的
优点：降低风险，减少因风险造成的损失
缺点：a.使用原型方法，为自身带来风险
b.模型过于复杂，不利于管理者依据其组织软件开发活动。
适用性：高风险的大规模软件系统开发

3.软件工程知识体系的知识域

需求、设计、构造、测试、维护、配置管理、工程管理、工程过程、工程工具和方法、质量

设计模式

可修改性

- 实现的可修改性：对已有实现的拓展
- 实现的可拓展性：对新的实现的拓展
- 实现的灵活性：对实现的动态配置

如何实现可修改性?

接口与实现的分离

如何将接口与实现分离?

- 通过**接口与实现该接口的类**，将接口与实现分离
- 子类**继承**父类，将父类的接口与子类的实现分离

使用interface，生成的对象只能看到interface里方法的声明，具体的实现是在class里实现的

```
public class Client{
    psvm{
        Interface_A a = new A1();
        a.method();
    }
}
//动态绑定
public interface Interface_A(){
    public void method();
}

public class A1 inplements Interface_A(){
    public void method(){
        sout("method");
    }
}
```

```
public class Client{
    psvm{
        A a = new superA();
        a.method();
    }
}

public class superA(){
    public void method(){
        sout("this is father");
    };
}

public class A extends superA(){
    public void method(){
        sout("this is son");
    }
}
```

以上两种方法对于Client没有任何的耦合性

继承

优点

缺点

- 父类和子类存在共有接口的耦合，当父类的接口需要改变，子类的接口也需要改变
- 子类对象创建后，不好动态改变接口

接口

优点

- 前后端分离，并不关心对方具体内容是什么，通过组合隔离变化，降低耦合性
- 后端类的实现可以动态创建，实现

工厂模式