软件工程与计算II重点整理

刘屿

【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

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（批量数据处理和事务计算）；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。奠定了软件工程的基础

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；大量面对大众的web产品；追求快速开发、可变更性、用户价值和创新

【第四章 项目启动】

1. 如何管理团队

（1）团队结构：主程序员团队；民主团队；开放团队

（2）团队建设：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

（1）需求开发：需求评审、需求度量

（2）体系结构：体系结构评审、集成测试（持续集成）

（3）详细设计：详细设计评审、设计度量、集成测试（持续集成）

（4）实现（构造）：代码评审、代码度量、测试（测试驱动、持续集成）

（5）测试：测试、测试度量

3. 配置管理有哪些活动

（1）标识配置项（2）版本管理（3）变更控制（4）配置审计（5）状态报告（6）软件发布管理

【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描述

2. 区分需求的三个层次

【题型】给出一个实例，给出三个层次的例子；对给定的需求实例，判断其层次

【示例】

系统使用三个月后，销售额应该提高20%（业务需求）

系统要帮助收银员完成销售处理（系统需求）

收银员输入商品的标识和数量时，系统要显示商品的描述、单价、数量和总价

【描述】

（1）业务需求：抽象层次最高的需求称为业务需求，是系统建立的战略出发点，表现为高层次的目标，描述了组织为什么要开发系统

（2）用户需求：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

（3）系统级需求：用户对系统行为的期望，每个系统及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

3. 掌握需求的类型

【题型】对给定实例，给出其不同类型的需求例子；对给定的需求示例，判断需求类型

【需求分类】项目需求、过程需求和系统需求（软件需求、硬件需求和其他需求）

【软件需求分类】

（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，用户希望系统能够执行的活动，这些活动可以帮助用户完成任务

（2）性能需求：速度、容量、吞吐量、负载、实时性

（3）质量属性：可靠性、可用性、安全性、可维护性、可移植性、易用性

（4）对外接口：系统和环境中其他系统之间需要建立的接口，包括用户界面、硬件接口、软件接口、网络通信接口

（5）约束：进行系统构造时需要遵守的约定，例如编程语言、硬件设施等

（6）数据需求：功能需求的补充，是需要在数据库、文件或者其他介质中存储的数据描述

【第六章 需求分析方法】

1. 建立用例图

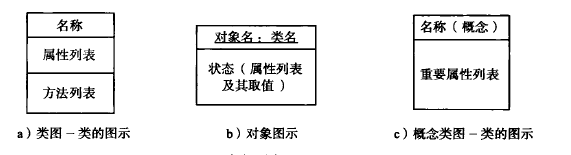
【四要素】用例、参与者、关系、系统边界

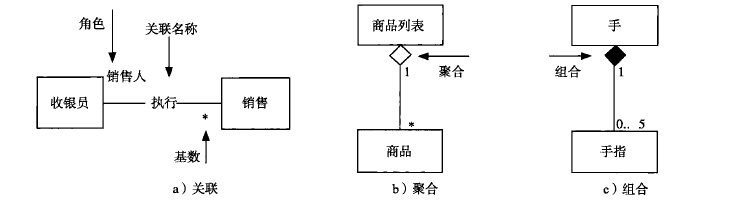
【步骤】（1）进行目标分析与确定解决方向（2）寻找参与者（3）寻找用例（4）细化用例

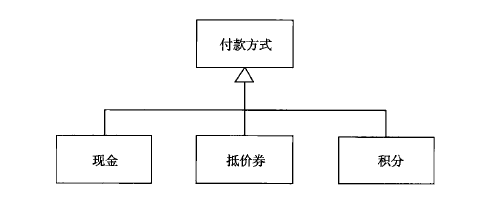
2. 建立分析类图（概念类图、领域模型）

【基本元素】对象、类、链接、关联（聚合）、继承

【图示】







【步骤】

（1）对每个用例文本描述，尤其是场景描述，建立局部的概念类图

（2）将所有用例产生的局部概念类图进行合并，建立软件系统的整体概念类图

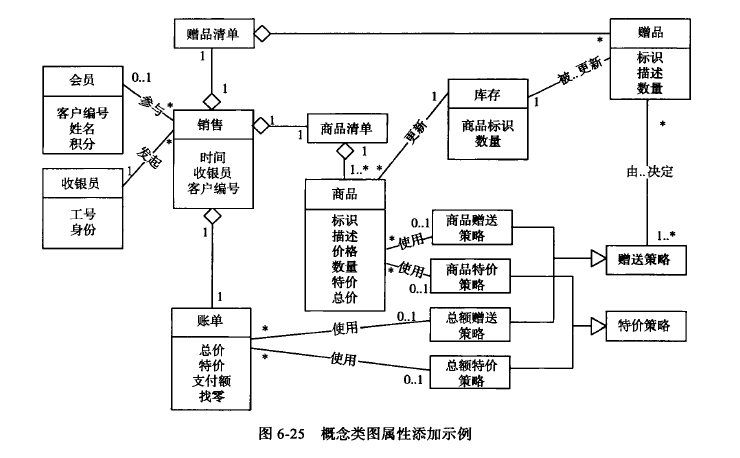
【建立局部概念类图】

识别候选类

确定概念类

识别关联

识别重要属性



3. 建立系统顺序图

系统顺序图是将整个系统看作一个黑箱的对象而描述的简单顺序图形式，他强调外部参与者和系统的交互行为，重点展示系统级事件

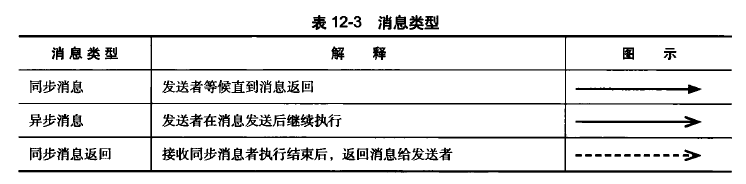
【步骤】

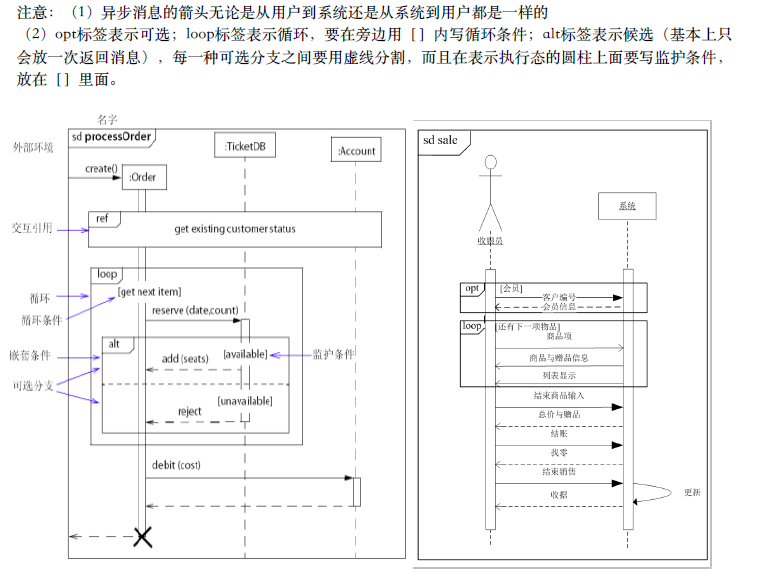
（1）确定上下文环境

（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

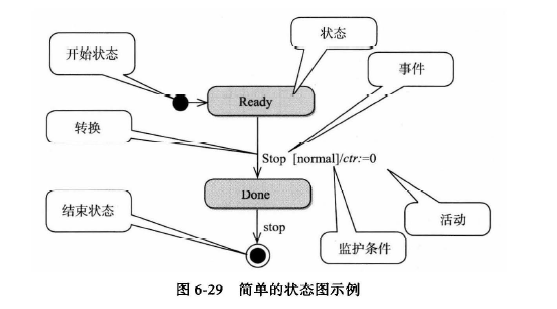
【图例】





4. 建立状态图

【要素】



【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

【第七章 需求文档化与验证】

1. 为什么建立需求规格说明？结合实验进行说明

（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，所以软件系统开发需要编写多种不同类型的文档，每种文档都针对项目中需要进行广泛交流的内容。

（2）软件需求是项目中需要进行广泛交流的内容之一，所以需求开发阶段需要进行需求的文档化

2. 对给定的需求示例，判断并修正其错误（对给定的需求规格说明片段，找出并修正其错误）

【需求文档化要点】

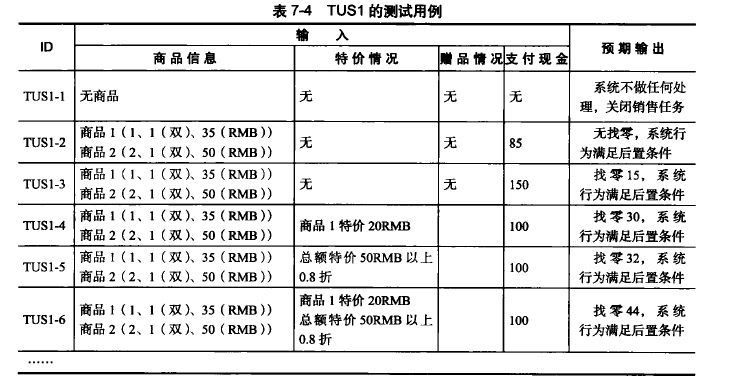
（1）技术文档写作要点：简洁、精确、易读（查询）、易修改

（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、保持所有内容位置得当；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



3. 对给定的需求实例，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）

软件设计是关于软件对象的设计，是一种设计活动，具有设计的普遍特性。

软件设计既指软件对象实现的规格说明，也指产生这个规格说明的过程。

2. 软件设计的核心思想是什么？

抽象和分解

分解是横向上将系统分割为几个相对简单的子系统以及各子系统之间的关系。

分解之后每次只需关注经过抽象的相对简单的子系统及其相互间的关系，从而降低了复杂度。

抽象则是在纵向上聚焦各子系统的接口。抽象可以分离接口与实现，让人更好地关注系统本质，从而降低复杂度。

3. 软件工程设计有哪三个层次？各层的主要思想是什么

高层设计、中层设计、低层设计

【高层设计】

基于反映软件高层抽象的构件层次，描述系统的高层结构、关注点和设计决策

【中层设计】

更加关注组成构件的模块的划分、导入/导出、过程之间调用关系或者类之间的协作

【低层设计】

深入模块和类的内部，关注具体的数据结构、算法、类型、语句和控制结构等

【第九、十章 软件体系结构设计与构建】

1. 体系结构的概念

一个软件系统的体系结构规定了系统的计算部件和部件之间的交互

2. 体系结构的风格的优缺点

【主程序/子程序】

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，这会使系统难以修改和复用

b. 程序调用的连接方式限制了各部件之间的数据交互，可能会使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

【面向对象式】

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）

【分层】

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

【MVC】

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

3. 体系结构设计的过程

（1）分析关键需求和项目约束

（2）选择体系结构风格

（3）进行软件体系结构逻辑（抽象）设计

（4）依赖逻辑设计进行软件体系结构物理（实现）设计

（5）完善软件体系结构设计

（6）定义构建接口

（7）迭代过程（3）-（6）

4. 包的原则

（1）重用发布等价原则REP

（不要把很多用例放在一个包里），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

（不要把一个用例拆成多个包），最小化变化给程序员造成的影响，一起变化的类放在一个包里

（3）共同重用原则CRP

根据公共复用给类分组，一个包里的类应当一起被复用，常常导致细分包

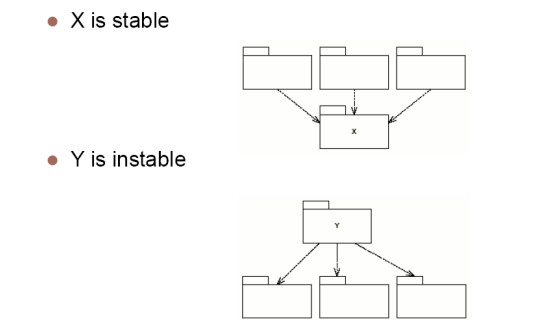
CCP和CRP原则互相排斥：CCP使维护者更简单，CRP使复用者更简单；CCP包更大，CRP包更小

（4）无环依赖原则ACP

包的依赖结构应该是一个有向无环图

（5）稳定依赖原则SDP

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

5. 体系结构构建之间接口的定义

根据分配的需求确定模块对外接口，如逻辑层接口根据页面的需求得到，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（参数，返回值）

（1）通常情况，VIEW的需接口可以直接作为响应LOGIC的供接口

（2）通常情况，LOGIC的需接口需要分解为同层模块和不同DATA的供接口

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

桩（Stub）：自顶向下集成，先继承和测试上层的模块，下层的模块使用伪装的具有相同接口的桩，比真实程序简单得多，是用最简单的逻辑。

（2）编写驱动程序，在桩程序的帮助下进行集成测试

驱动（Driver）：自底向上集成，从底层的模块集成起，测试的时候上层的模块使用伪装的相同接口的驱动

（3）持续集成：提倡尽早集成和频繁集成

尽早集成指开发之初就利用stub集成起来，频繁集成指每次完成一些开发任务，就用开发结果替换stub中相应组件

好处：a. 防止软件开发中出现无法集成与发布的情况。 b. 有利于检查和发现集成缺陷

工具链：利用版本控制工具和持续集成工具

【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。他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，所以易用性不是单维度的质量属性，而是多维度的质量属性，包括易学性、易记性、效率、出错率和主观满意度

2. 能够列出至少5个界面设计的注意事项，并加以解释

【题型】例子\*\*违反了哪几条界面设计原则/很好的运用了哪些原则？

（1）简洁设计：不要使用太大的菜单，不要在一个窗口表现过多的信息类别，不要在一个表单中使用太多的颜色和字体作为线索，等等

（2）一致性设计：遵循用户已有的精神模型，不要出现相似任务具有不同的交互机制（例如按钮位置不一致）

（3）低出错率设计：首先帮助用户避免犯错，设计不让用户犯严重错误的系统，错误出现时，系统要提供简洁、有建设性、具体的指导来帮助用户消除错误

（4）易记性设计：减少用户的记忆负担，方式有：逐层递进展示信息、使用直观的快捷方式、设置有意义的默认值

3. 精神模型、差异性

精神模型：用户进行人机交互时头脑中的任务模型，人机交互设计需要依据精神模型进行隐喻设计

差异性：不同用户群体的任务模型是有差异的，可以划分为新手用户、专家用户、熟练用户

【新手用户】是对业务不熟悉的人，设计系统时要关注易学性，进行业务导航，尽量避免出错

【专家用户】是能熟练操作计算机完成业务的人，设计系统要关注效率

【熟练用户】介于新手和专家之间，要在易学性和效率之间折中

4. 导航、反馈、协作式设计

【导航】

为用户提供一个完成任务的入口，好的导航会让这个入口非常符合人的精神模型

有全局结构和局部结构两种方式

全局结构包括窗口、菜单、列表、快捷方式等，主要以功能分层和任务交互过程为依据

局部结构包括可视化控件布局与组合、按钮设置、文本颜色等，主要以用户关注的任务细节为依据

【反馈】

提示用户交互行为的结果，但不能打断用户工作时的意识流

【协作式设计】

人和计算机是人机交互的两方，其中人的因素是比较固定的，一定时期内不会发生大的变化，所以要让二者交互顺畅，就需要让计算机更多地适应人的因素，这也是人机交互设计以用户为中心的根本原因

这种调整计算机因素以更好地适应并帮助用户地设计方式就成为协作式设计

【第十二章 详细设计概述】

1. 详细设计的出发点

软件详细设计在软件体系结构设计之后进行，以需求开发的结果和软件体系结构的结果为出发点

2. 职责分配

通过职责建立静态模型

【抽象对象的职责】类的职责主要由两部分组成：属性职责和方法职责。属性表示对象的状态，方法表示对象的行为

【抽象类之间的关系】类与类之间存在一定关系。关系表达了相应职责的划分与组合。它们的强弱顺序为：依赖 < 关联 < 聚合 < 组合 < 继承

【添加辅助类】帮助实现所有功能

3. 协作

通过协作建立动态模型

【抽象对象之间的协作】有两种方法

（1）从小到大，将对象的小职责聚合形成大职责

（2）从大到小，将大职责分配给各个小对象

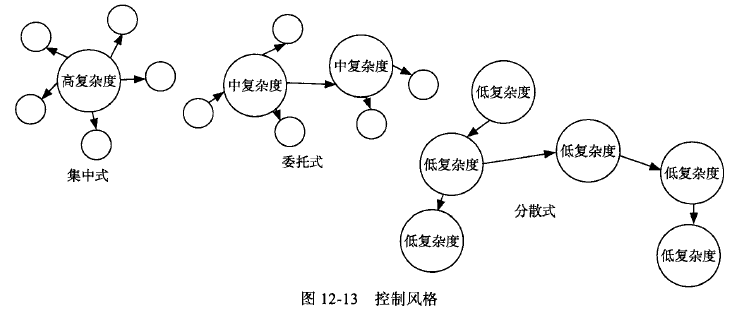
4. 控制风格

为了完成某一个大职责，需要对职责的分配做很多决策。控制风格决定了决策谁来做和怎么做决策。

【集中式】作决策的往往只有一个对象，由这个对象决定怎么来分配职责，怎么实现大的职责

【委托式】做出决策的对象不止一个，分别承担一定的职责，做出一定的决策，从而实现大的职责

【分散式】无法找到明确的控制对象，每个对象只承担一个相对较小的职责，完全由各个对象自治的方式来实现大的职责

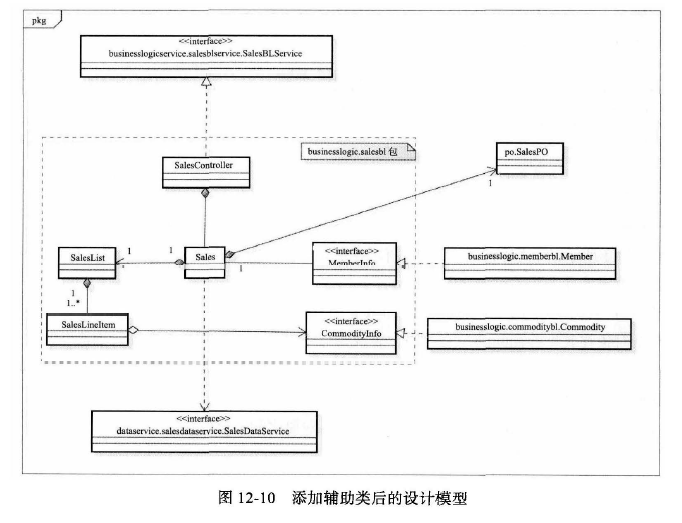


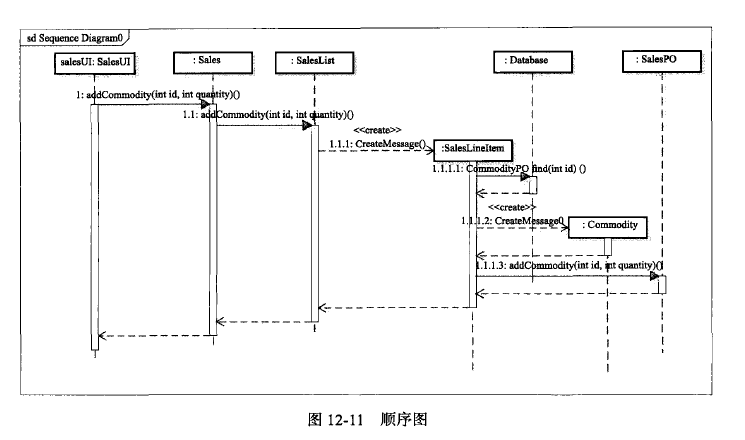
5. 给定分析类图、系统顺序图和设计因素描述

建立设计类图或者详细顺序图

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、数据类、启动类、控制器类等





6. 协作的测试

类间协作的桩程序：Mock Object

【第十三章 详细设计中的模块化与信息隐藏】

1. 耦合与内聚（名词解释）

【耦合】耦合描述的是两个模块之间关系的复杂程度。

【内聚】内聚表达的是一个模块内部的联系的紧密性

2. 对例子\*\*，说明它们之间的耦合程度与内聚，给出理由

【耦合】（高->低）

（1）内容耦合（一个模块直接修改另一模块内容，GOTO语句，共享代码，内部类肯定是内容耦合）

（2）公共耦合（模块间共享全局变量，文件，设备等）

（3）重复耦合（两个模块有相同的业务逻辑代码）

（4）控制耦合（一个模块给另一个模块传递控制信息，共享逻辑）

（5）印记耦合（传递的数据结构只使用了一部分）

（6）数据耦合（只共享对方需要的数据）

【内聚】（高->低）

（1）信息内聚（模块每个操作都有各自入口点，相对独立，所有操作在相同数据结构上完成，如：栈）

（2）功能内聚（模块只执行一个操作或达成单一目的，如：计算平方根，决定最短路径）

（3）通信内聚（对相同数据执行不同操作：查书名、查作者、查出版商）

（4）过程内聚（与步骤有关：守门员传后卫，后卫传中场，中场传前锋）

（5）时间内聚（一系列与时间有关的操作：起床、刷牙、吃早点）

（6）逻辑内聚（一系列可替换的操作：坐车去、坐飞机去、坐船去，switch）

（7）偶然内聚（多个不相关操作）

3. 信息隐藏

【基本思想】每个模块都隐藏一个重要的设计决策。

每个模块都承担一定的职责，对外表现为一份契约，并且在这份契约之下隐藏着只有这个模块知道的设计决策或者秘密，决策的实现细节只有模块自己知道

【两种常见的信息隐藏决策】

根据需求分配的职责、内部实现机制

【题型】对例子\*\*，说明其信息隐藏程度好坏

【第十四章 详细设计中面向对象方法下的模块化】

1. 模块化的原则

【降低隐式耦合】

（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）

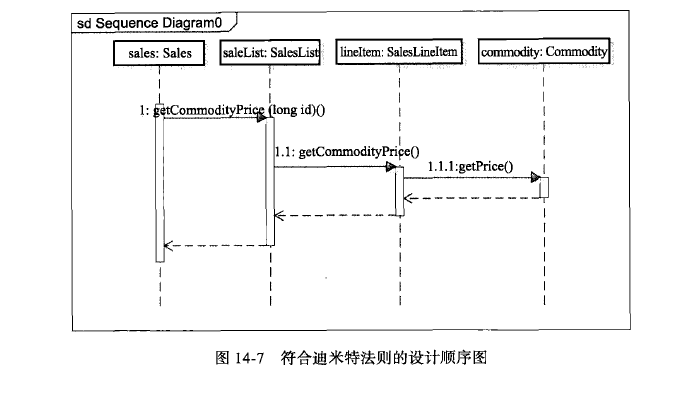
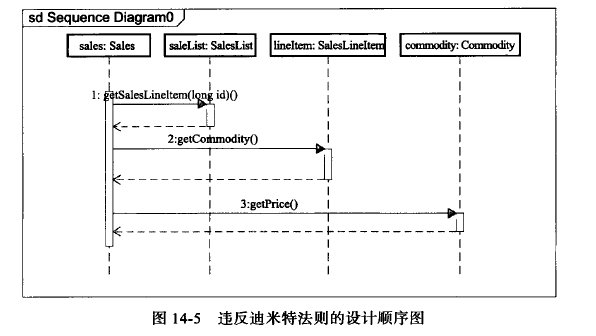
（3）不要有代码重复

【降低访问耦合】

（4）针对接口编程：供接口、需接口

（5）接口分离原则（ISP）：将一个统一的接口匹配为多个更独立的接口

（6）迪米特法则：不要出现a.getB().methodB()



【降低继承耦合】

（7）里氏替换原则（LSP）：只要调用一个接口，不论是哪种子类或者父类自身，都可以完成相同语义的工作（因为接口没有被改变），（子类的前置条件必须与超类前置条件相同或更少，子类方法后置条件必须与超类方法后置条件相同会更多）

（8）使用组合代替继承

【提高内聚】

（9）单一职责原则

（10）集中信息与行为

2. 对给定的实例，发现其违反的原则并进行修正

【第十五章 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

一个模块应该通过稳定的接口对外表现其所承载的需求，而隐藏它对需求的内部实现细节。

2. 封装

（1）将数据和行为同时包含在类中

（2）分离对外接口与内部实现

封装实现细节：封装数据和行为；封装内部结构；封装其他对象的引用；封装类型信息；封装潜在变更

3. OCP

开闭原则OCP：对扩展开放，对修改封闭，即发生变更时，只需要添加新的代码而不需要修改原有的代码

违反OCP原则的典型标志：switch或if-else

可以使用多态来保证

4. DIP

依赖倒置原则DIP（与工厂结合紧密，解决new的创建问题）

（1）抽象不应该依赖于细节，细节应该依赖于抽象。（抽象是稳定的，细节是不稳定的）

（2）高层模块不应该依赖于低层模块，而是双方都依赖于抽象。（抽象是稳定的，高低层模块都可能是不稳定的）

为具体类建立抽象接口并分离该接口是实现DIP的基本手段

【第十六章 设计模式】

1. 如何实现可修改性、可扩展性、灵活性

通过接口与实现分离，Java中有两种方式

（1）通过接口和实现该接口的类

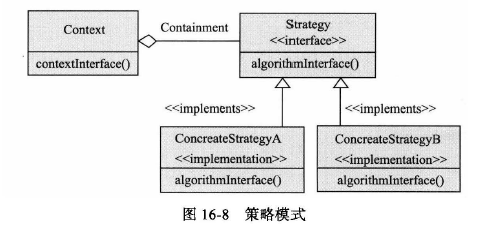
（2）通过子类继承父类

对于实现的可修改性，两种方式都没有任何耦合性

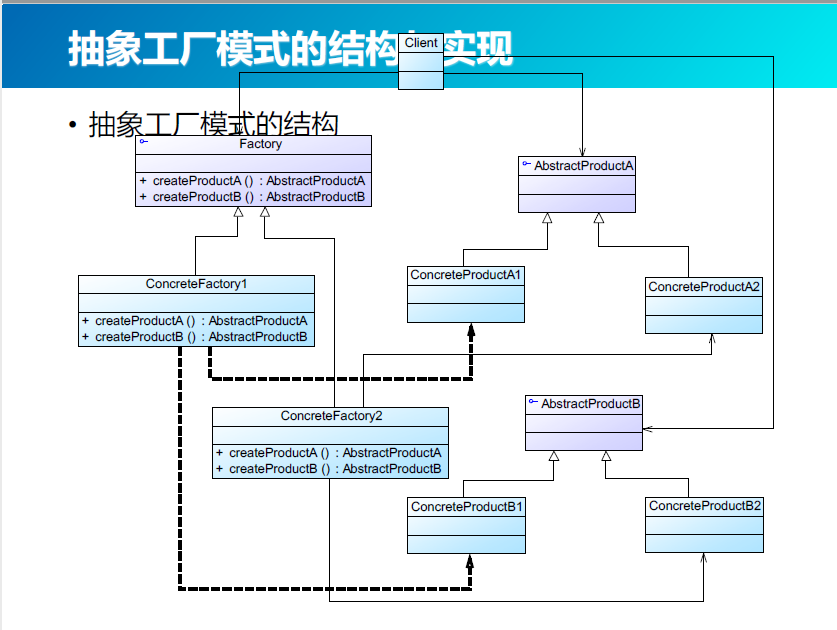
对于实现的可扩展性，可通过新的子类的创建来实现

对于实现的灵活性，可以通过多态的方式来实现

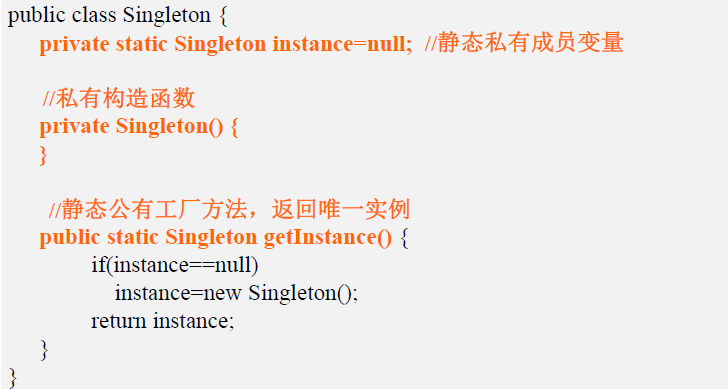
2. 策略模式



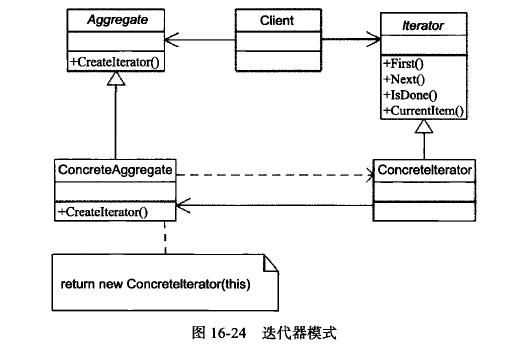
3. 抽象工厂模式



4. 单件模式



5. 迭代器模式



6. 给定场景，应用设计模式并写出代码

7. 给出代码，要求用设计模式改写

【第十七、十八章 软件构造与代码设计】

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. 重构、测试驱动开发、结对编程（名词解释）

【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

【测试驱动开发】编写一段代码前，优先完成该段代码的测试代码

【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，对其进行改进或者发现其中的问题

【易读性】

格式（使用缩进与对齐，将相关逻辑组织在一起，使用空行分隔逻辑，语句分行）

命名

注释（文档注释、内部注释）

【易维护性】

小型任务（分解为多个高内聚、低耦合的小任务）

复杂决策（使用新的布尔变量来简化，使用有意义的名称封装复杂决策，表驱动编程）

数据使用（不要将变量用于与命名不符的目的，不要将单个变量用于多个目的，限制全局变量，不要使用突兀的数字与字符）

明确依赖关系

【可靠性】

契约式设计

防御式编程

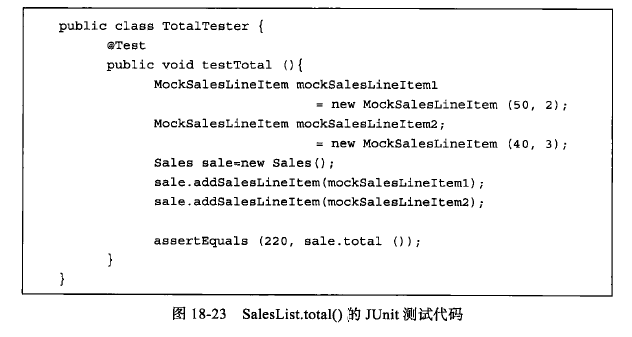
【使用模型辅助设计复杂代码】

决策表、伪代码、程序流程图

4. 单元测试用例的设计

（1）为方法开发测试用例（根据方法的规格、方法代码的逻辑结构）

（2）使用Mock Object测试类方法

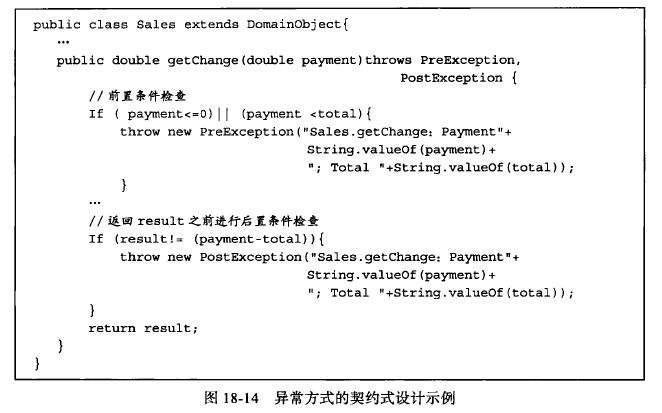
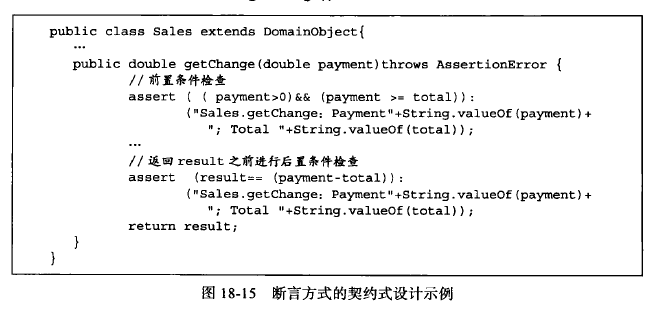


（3）为类开发测试用例

5. 契约式设计

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，那么这个函数或方法就是正确、可靠的。

有异常与断言两种方式

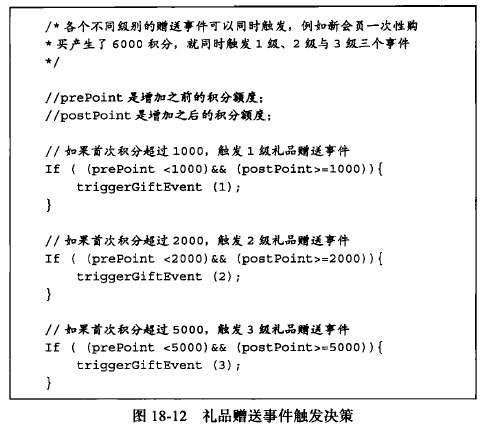
6. 防御式编程

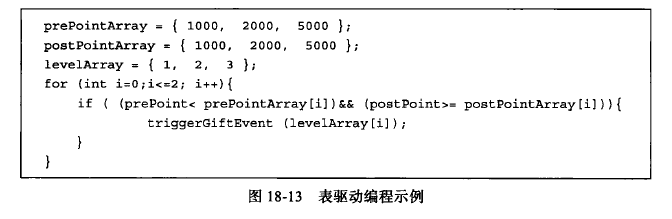
在一个方法与其他方法、操作系统、硬件等外界环境交互时，不能确保外界都是正确的，所以要在外界发生错误时，保护方法内部不受损害

会增加整体代码复杂度，降低易读性和性能，但显著提高程序的可靠性

7. 表驱动

复杂决策包装成决策表





8. 代码复杂度度量

圈复杂度：从1开始，每次遇到if/while/for/repeat就加1，case语句每种情况都加1

【第十九章 软件测试】

1. 掌握白盒测试和黑盒测试的常见方法，并能够进行优缺点比较

【黑盒测试】

基于规格的技术，黑盒测试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

常见方法：等价类划分、边界值分析、决策表、状态转换

【白盒测试】

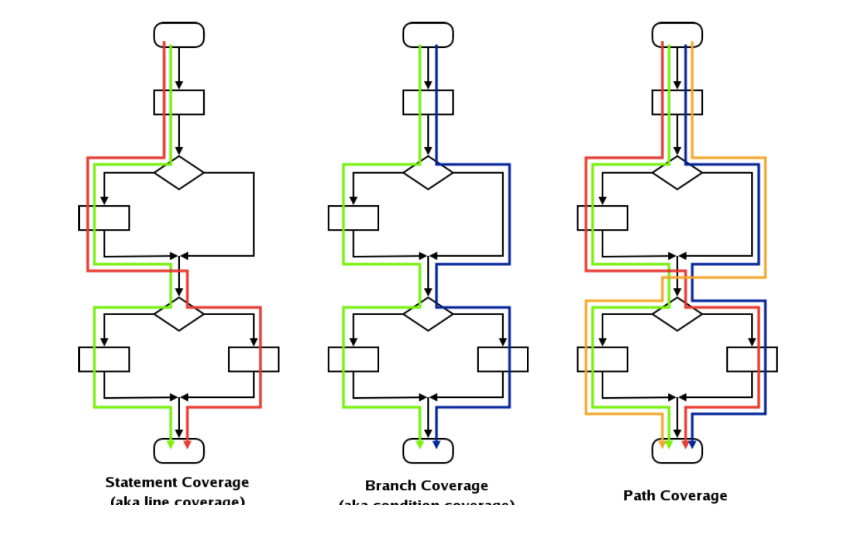
基于代码的技术，白盒测试将测试对象看作透明的，不关心测试对象的规格，而是按照测试对象内部的程序结构来设计测试用例进行测试工作

2. 能解释并区分白盒测试三种不同的方法

【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



3. 给出一个场景，判断应该使用哪种测试方法，如何去写

对给定的场景和要求的测试方法，设计测试用例

（1）给出功能需求，则要求写功能测试用例

（2）给出设计图，则要求写集成测试用例，Stub and Driver

（3）给出方法描述，则要求写单元测试用例，Mock Object

（4）JUnit基本使用方法

【第二十、二十一章 软件交付、软件维护与演化】

1. 如何理解软件维护的重要性

（1）出现新的需求，如果不维护，软件将减少甚至失去服务用户的使用

（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，软件需要经常修改以适应外界变化

（3）软件产品中存在缺陷，当这些缺陷在使用中暴露出来时，必须予以及时的解决

2. 开发可维护软件的方法

（1）考虑软件的可变更性：分析需求的易变性，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，保证代码的可读性，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

初始开发->演化->服务->逐步淘汰->停止

【初始开发】可以实现全部需求，也可以只包含部分需求—对用户来说非常重要和紧急的最高优先级需求

【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，保持软件产品的持续增值。

【服务】不再持续地增加自己的价值，而只是周期性地修正已有的缺陷

【逐步淘汰】不再提供软件产品的任何服务，即不再继续维护该软件

【停止】正式退出使用状态。开发者不再维护，用户也不再使用

4. 用户文档、系统文档

用户文档：为用户编写参考指南或者操作教程，常见有用户使用手册、联机帮助文档

系统文档：更注重系统维护方面的内容

5. 逆向工程、再工程

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

再工程：检查和改造一个目标系统，用新的模式及其实现复原该目标系统

【第二十二、二十三章 软件开发过程模型与职业基础】

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. 解释与比较不同过程模型（要求、特征描述、优点、缺点）

【构建-修复模型】

（1）特征：

对软件开发活动没有任何规划和组织，是完全依靠开发人员个人能力进行软件开发的方式

（2）缺点：

a. 没有对开发工作进行规范和组织，一旦开发活动超过个人的控制能力，开发活动就无法有效进行

b. 没有分析需求的真实性，给软件开发带来很大的风险

c. 没有考虑软件结构的质量，使得软件结构在不断地修改中越来越差，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，软件的维护十分困难

（3）适用范围：

a. 软件规模很小，只需要几百行代码，开发复杂度个人能够胜任

b. 软件对质量的要求不高，即使出错也无所谓

c. 只关注开发活动，对后期维护的要求不高，甚至不需要维护

【瀑布模型】

（1）特征：

要求每个活动的结果必须要进行验证，并且只有在经过验证之后才能作为后续开发活动的基础，被看作“文档驱动”

（2）优点：

为软件开发活动定义了清晰的阶段划分，让开发者能够以关注点分离的方式更好地进行那些复杂的软件项目的开发活动

（3）缺点：

a. 对文档的过高期望

b. 对开发活动的线性顺序假设

c. 客户、用户参与不够

d. 里程碑粒度过粗

（4）适用范围：

a. 需求非常成熟、稳定、没有不确定的内容，也不会发生变化

b. 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

c. 复杂度适中，不至于产生太大的文档负担和过粗的里程碑

【增量迭代模型】

（1）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对前景和范围的界定，然后再将后续开发活动组织为多个迭代、并行的瀑布式开发活动。（需求驱动）

（2）优点：

a. 使用迭代式开发，具有更好的适用性

b. 并行开发可以帮助缩短软件产品的开发时间

c. 渐进交付可以加强用户反馈，降低开发风险

（3）缺点：

a. 由于各个构件是逐渐并入已有的软件体系结构中的，所以加入构件必须不破坏已构造好的系统部分，这需要软件具备开放式的体系结构

b. 增量迭代模型需要一个完备、清晰的项目前景和范围以进行并行开发规划，但是在一些不稳定的领域，不确定性太多或者需求变化非常频繁，很难在项目开始就确定前景和范围

（4）适用范围：

大规模软件系统开发

【演化模型】

（1）特征：

可以更好地应对需求变更，将软件开发活动组织为多个迭代、并行的瀑布式开发活动

（2）优点：

a. 使用迭代式开发，具有更好的适用性，尤其适用于需求变更频繁或者不确定性较多的软件系统开发

b. 并行开发可以帮助缩短软件产品的开发时间

c. 渐进交付可以加强用户反馈，降低开发风险

（3）缺点：

a. 无法在项目早期阶段确定项目范围，所以项目的整体计划、进度调度，尤其是商务协商事宜无法准确把握

b. 后续迭代的开发活动是在前导迭代的基础上进行修改和扩展的，这容易让后续迭代忽略分析与设计工作，蜕变为构建-修复方式

（4）适用范围

不稳定领域的大规模软件系统开发

【原型模型】

（1）特征：

注重使用抛弃式原型，而不是演化式原型

（2）优点：

a. 对原型方法的使用加强了与客户、用户的交流，可以让最终产品取得更好的满意度

b. 适用于非常新颖的领域，这些领域因为新颖所以有着大量的不确定性

（3）缺点：

a. 原型方法能够解决风险，但自身也能带来新的风险

b. 实践中，很多项目负责人不舍得抛弃“抛弃式原型”，使得质量较差的代码进入了最终产品，导致了最终产品的低质量

（4）适用范围：

有着大量不确定性的新颖领域进行开发活动组织

【螺旋模型】

（1）特征：

尽早解决比较高的风险，风险驱动

（2）优点：

可以降低风险，减少项目因风险造成的损失

（3）缺点：

a. 风险解决需要使用原型手段，也就会存在原型自身带来的风险

b. 模型过于复杂，不利于管理者依据其组织软件开发活动  
（4）适用范围：

高风险的大规模软件系统开发

3. 对给定的场景，判定适用的开发过程模型

4. 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