第 1 章

软件工程

(1) 应用系统、规范、可度量的方法来开发、运行和维护软件，即将工程应用到软件。
(2) 对 (1) 中方法的研究。

第 2 章

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 4 章

团队结构

（结合实验采取了哪些办法？有哪些经验？）
1. 主程序员团队
优点：项目规模较小或主程序员的能力非常突出，能够获得很高的工作效率，保证产品不同元素的一致性。
缺点：项目复杂或者主程序员能力不足，其他团队成员无法发挥主动性，降低团队成员的积极性。
2. 民主团队
优点：能充分发挥所有成员的积极性。
缺点：工作效率较低，在统一思想和解决冲突上花时间。
3. 开放团队
优点：为了创新而存在，极大地激励成员的主动性，最大化发挥团队成员的创新能力。
缺点：项目进展没有可视度。

质量保障的措施

（结合实验说明）
质量验证的方法主要有评审、测试和质量度量三种。
需求开发：需求评审、需求度量
体系结构：体系结构评审、集成测试
详细设计：详细设计评审、设计度量和集成测试

实现（构造）：代码评审、代码度量、测试（测试驱动、持续集成）
测试：测试、测试度量

配置管理活动
（结合实验说明如何进行配置管理）
1. 标识配置项：哪些配置项需要被保存和管理
2. 版本管理：为配置项赋予版本号
3. 变更控制：以可控、一致的方式进行变更处理
4. 配置审计：确定一个项目满足需求的功能和物理特征的程度
5. 状态报告：标识、收集和维持演化中的配置状态信息
6. 软件发布管理：将软件配置项发布到开发活动之外

第 5 章

需求
(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。

需求的三个层次 P72
（给实例写三个层次 / 给需求实例判断层次）
1. 业务需求：完成的是某个目标
系统建立的战略出发点
eg. 提供会员服务，增加回头率
2. 用户需求：完成的是某个任务
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
eg. 系统要帮助收银员完成销售处理
3. 系统级需求：完成的是某个系统行为
用户对系统行为的期望，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
eg. 收银员输入购买商品的标识和数量时，系统显示该商品的描述、数量和总价。

需求的类型 P76
（给实例写不同类型的需求 / 给需求实例判断类型）
1. 功能需求：解决用户问题和生产价值的基础
业务、用户、系统需求
2. 性能需求：速度、容量、吞吐量、负载、实时性
速度：所有用户查询都必须在10秒内完成
容量：系统所能存储的数据量至少10万个
吞吐量：解释器每分钟至少应该解析5000条没有错误的语句
负载：系统应该允许50太营业服务器通识从集中服务器上进行数据的上传或下载
实时性：检测到病人的异常后，监控器必须在0.5秒内发出警告

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
可靠性：在客户端和服务器端通信时，如果网络故障，系统不能出现故障。
可用性：系统的可用性要达到98%
安全性：收银员只能查看不能修改、删除会员信息
可维护性：如果系统要增加新的特价类型，要能够在2个月内完成
可移植性：服务器要能够在一个月内从 windows7 操作系统更换到××系统
易用性：使用系统一个月的收银员进行销售处理的效率要达到10商品/min
4. 对外接口：系统和环境中其他系统之间需要建立的接口
系统的网络通信接口为 RS232 接口
5. 约束：进行系统构造时需要遵守的约定，如编程语言、硬件设施等，常见：系统开发及其运行的环境，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
系统使用 Java 语言进行开发
6. 数据需求：明确定义了相关的数据结构
系统需要存储1年内的销售记录和退货记录

第 6 章

为给定描述建立 用例图 分析类图 系统顺序图 状态图 P94
（看复习C5&6&7）

建立用例图
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
5. 注意事项：不要将用例细化为单个操作，不要将同一个业务目标细化为不同用例，不要将没用业务价值的内容作为用例

建立分析类图
概念类图又称为领域模型，关注系统与外界的交互
1. 识别候选类
2. 确定概念类
3. 识别关联
4. 识别重要属性

建立系统顺序图
1. 确定上下文环境
2. 发现交互对象
3. 根据用例描述中的流程，逐步添加信息

建立状态图
1. 确定上下文环境，搞清楚状态的主体

2. 识别状态
3. 建立状态转换
4. 补充详细信息

第 7 章

需求规格说明
（结合实验进行说明）
1. 需求规格说明描述了软件系统的解决方案，书写需求规格说明，可以建立管理控制的基线，方便任务分配，制定工作计划，并进行跟踪，度量。
2. 方便交流，软件需求是项目中需要进行广泛交流的内容之一，文档化有利于沟通交流。用户需要验证文档内描述的需求信息与其最初意图一致；项目管理者基于它进行软件估算，并根据估算数据安排进度和人员分工；设计人员和程序员需要依据需求文档来完成自己的任务，依次判断其工作是否正确；测试人员，根据文档的需求内容进行验收测试，确保产品符合用户要求；文档编写人员依此编写用户手册；维护人员依此理解原有软件。对给定的需求示例，判断并修正其错误。在实验中，以需求为基础进行了系统的功能测试，开发测试用例套件，在系统级需求上达到测试全覆盖。并且后续每个阶段几乎都是以需求规格为依据，尤其是在设计阶段评审的重要内容就是是否符合需求规格。
3. 在实验中，需求规格的重要性不只体现在结果上，还包括中间过程，在书写需求规格过程中，才真正把问题域的问题和分析模型的成果转化为系统级需求，方便小组成员真正明确需求，个人认为在这个阶段包含一部分的需求在发现和完整化。

对给定的需求规格说明片段，找出并纠正其错误 PPT例题
技术文档注意要点：
1. 简洁：动词名词+辅助词，不要使用复杂长句，形容词和副词。
2. 精确：不能产生歧义或者无法理解。
3. 易读（查询）有效使用引言目录和索引；使用系统化的方式组织内容信息，提高文档内容的可读性。
4.易修改：使用相同的语句格式组织相关联或相似的信息；使用列表组织独立、并列的信息；使用编号表达繁杂信息之间的关系。
需求书写要点：
1. 使用用户术语：不要使用“计算机术语”
2. 可验证：不可验证的需求往往是过于抽象或者描述模糊。具体化，避免程度词的使用。
3. 可行性：在运行环境的已知条件和约束下实现。

对给定的需求示例，结合测试方法，设计功能测试用例 PPT
1. 以需求为线索，开发测试用例套件
2. 使用测试技术确定输入输出

第 8 章

软件设计

软件设计是关于整个软件对象的设计。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。

软件设计的核心思想
抽象和分解是软件设计的核心思想。
分解：横向上将系统分解为几个相对简单的子系统以及各子系统之间的关系。
抽象：纵向上聚焦各子系统的接口，可以分离接口和实现，使得人们更好地关注软件系统本质，降低复杂度。

软件工程设计的三个层次
高层设计：基于反映软件高层抽象的构建层次，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、过程之间的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 9 章

体系结构
软件体系结构 = { 部件，连接件，配置 }
部件：软件体系结构的基本组成单位之一，承载系统的主要功能，包括处理与数据。
连接件：是软件体系结构的另一个基本组成单位，定义了部件之间的交互，是链接的抽象表示。
配置：对“形式”的发展，定义了“部件”以及“连接件”之间的关联方式，将他们组织成系统的总体结构。

体系结构的风格的优缺点
1. 主程序/子程序
   优点：
      1. 流程清晰，易于理解（符合分解和分治的思想）
      2. 强控制性（很容易保证正确性）
   缺点：
      1. 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，这会使得系统难以修改和复用
      2. 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
2. 面向对象式
   优点：
      1. 内部实现的可修改性（隐藏内部实现）
      2. 易开发、易理解、易复用的结构组织（契合模块化思想）
   缺点：
      1. 接口的耦合性（由于方法调用机制，接口的耦合性无法消除）
      2. 标识的耦合性（一个对象要和其他对象交互，必须知道标识）
      3. 副作用（难以实现程序的“正确性”）

3. 分层
   优点：
      1. 设计机制清晰，易于理解（抽象层次分离，隔离复杂度）
      2. 支持并行开发（层次之间遵守成熟稳定的接口）
      3. 更好的可复用性与内部可修改性（接口的稳定性，不同层次的部件能够互相替换）

   缺点：
      1. 交互协议难以修改（可能需要改变所有的层次，接口具有强耦合性）
      2. 性能损失（禁止跨层调用）
      3. 难以确定层次数量和粒度
4. MVC
   优点：
      1. 易开发性（分别抽象了业务逻辑，表现和控制机制清晰，易于开发）
      2. 视图和控制的可修改性（一个模型可以同时建立并支持多个视图）
      3. 适宜于网络系统开发的特征
   缺点：
      1. 复杂性（不利于理解任务实现）
      2. 模型修改困难（视图和控制均依赖于模型）

体系结构设计的过程
1. 分析关键需求和项目约束：软件需求规格说明和项目约束
2. 选择体系结构风格
3. 进行软件体系结构逻辑（抽象）设计：产生逻辑包图
4. 依据逻辑设计进行软件体系结构物理（实现）设计：产生物理包图
5. 完善软件体系结构设计：关键类图，持久化数据格式的定义等
6. 定义构件接口：包、重要文件的创建，定义接口。
7. 迭代过程 3 –> 6

包的原则
1. 重用发布等价原则（REP）：重用的粒度就是发布的粒度
2. 共同封闭原则（CCP）：包中所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则对该包中所有类产生影响，而对于其他包不造成任何影响。
3. 共同重用原则（CRP）：一个包中的所有类应该是共同重用的。
4. 无环依赖原则（ADP）：在包的依赖关系图中不能存在环
5. 稳定依赖原则（SDP）：朝着稳定的方向进行依赖
6. 稳定抽象原则（SAP）：包的抽象程度应该和其稳定程度一致

体系结构构建之间接口的定义（能够根据用例写出某层的接口，需求分配的过程，重点）

体系结构开发集成测试用例（Stub和Driver）

第 11 章

可用性

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。

人机交互设计原则

（例子**违反了哪些条界面设计原则）

1. 简洁设计
2. 一致性设计
3. 低出错率设计（用简洁、精炼准确、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
    - 设置有意义的默认值
5. 可视化设计要点
    - 按照任务模型设计界面隐喻，同时不要把软件系统的内部构造机制暴露给用户
    - 可视化设计还应该基于界面隐喻，尽可能地把功能和任务细节表现出来

精神模型

精神模型就是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

对不同用户群体的任务模型是有差异的，所以对他们的人机交互设计也要有差异。按照用户群体自身的特点，可以将其划分为新手用户（易学性）、专家用户（效率）和熟练用户

导航、反馈、协作式设计

软件系统**导航**要能帮助用户找到任务的入口，导航的目的就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。

好的人机交互设计需要对用户行为进行**反馈**，让用户能够意识到行为的结果。反馈的目的是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。对用户思考和反应时间的把握。

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，所以要让两者交互顺畅，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的设计方式被称作为**协作式设计**。

第 12 章

详细设计的出发点

以需求开发的结果（需求规格说明书和需求分析模型）和软件体系结构的结果（软件体系结构设计方案与原型）为出发点。
在详细设计中一般进行中层设计和低层设计，详细设计的目的是实现所有功能性需求和非功能性需求。

职责分配

1. 职责是持有某项数据或者表现某种职责的义务；数据职责主要由属性满足，行为职责主要由方法满足；可能会包含类间协作
2. 职责可以从不同的抽象层次开始；职责是可以被分解的；职责分解可以是分解组件的基础
3. 职责分配可以帮助实现高内聚低耦合，确保不同模块职责没有重合，当且仅当数据和方法有助于实现模块职责时将其添加给模块。
4. GRASP原则（通用职责分配软件模式—将职责分配原则总结为模式）：
    1. Expert专家：将一个职责分配给专家—掌握了履行职责所必需的信息的类。
    2. Creator创建者：
    3. 高内聚：分配一个职责的时候要保持类的高聚合度
    4. 低耦合：分配一个职责是要保持低耦合度
    5. Controller：
        A. 避免大多数消息由一个类发出
        B. 组件相对较小
        C. 行为、职责和数据绑定
        D. 职责单一

协作

根据协作建立动态模型：
1. 从大到小，将对象的小职责聚合形成大职责；
2. 从小到大，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。

控制风格

1. 集中式：做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，怎么来实现大的职责。
2. 委派式：做出决策的对象不只一个，这些对象分别承担一定的职责，做出一定的决策，从而共同实现大的职责。
3. 分散式：无法找到明确的控制对象，每个对象都只承担一个相对较小的职责，完全靠各个对象自治的方式来实现大的职责。

给定分析类图、系统顺序图和设计因素描述
建立设计类图 P205
或者详细顺序图 P207

协作的测试

第 13 章

耦合和内聚

耦合

描述的是两个模块之间关系的复杂程度

耦合根据其耦合性的高低分为：内容耦合、共享耦合、控制耦合、印记耦合、数据耦合。

高–>低

内容耦合：一个模块直接修改或者依赖于另一个模块的内容

公共耦合：模块之间共享全局的数据

重复耦合：模块之间有同样逻辑的重复代码

控制耦合：一个模块给另一个模块传递控制信息，传递模块和接收模块必须共享一个共同的内部结构和逻辑

印记耦合：共享一个数据结构，但是却只用了其中一部分

数据耦合：模块和模块之间通过参数传递，只共享对方需要的数据

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。

内聚

表达的是一个模块内部的联系的紧密性

内聚根据其内聚性的高低分为：信息内聚、功能内聚、通信内聚、过程内聚、时间内聚、逻辑内聚、偶然内聚。

低–>高

偶然内聚：模块执行多个完全不相关的操作

逻辑内聚：模块执行一系列相关操作，每个操作的调用由其他模块来决定，逻辑上相似但没有直接关联的操作

时间内聚：模块执行一系列与时间有关的操作，是模块中执行一系列操作，但是这些操作在同一时间段内发生

过程内聚：模块执行一些与步骤顺序有关的操作，是模块执行的多个操作，是解决同一个问题的不同步骤

通信内聚：模块执行一系列与步骤有关的操作，这些操作在相同的数据上进行

功能内聚：模块只执行一个操作或达到一个单一目的

信息内聚：模块进行许多操作，各个都有各自的入口点，每个操作的代码相对独立，而且所有操作都在相同的数据结构上完成

内聚性越高越好，越低越不易实现变更和重用。

（对例子**，说明它们之间的耦合程度与内聚，给出理由）

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，对外表现一定的契约，并且在这份契约下隐藏着只有这个模块知道的设计决策或者秘密，决策实现的细节（特别是容易改变的细节）只有该模块自己知道。

（复习知识点）

两种常见的信息隐藏决策

一是根据需求分配的职责，因为实践表明，需求是经常变化的，频率和幅度都很大。
二是内部实现机制，常见的变化主题包括硬件依赖，输入输出形式，非标准语言特征和库，负责的设计和实现，复杂的数据结构，复杂的逻辑，全局变量，数据大小限制等。
（对例子**，说明其信息隐藏程度好坏）

第 14 章

模块化设计原则
1. 《Global Variables consider Harmful》 不要使用全局变量
2. 《To be Explicit》 内容明确
3. 《Do not Repeat》 不要有代码重复
4. Programming to Interface(Design by Contract) 针对接口编程
5. The Law of Demeter 迪米特法则(每个单元对于其他单元只能拥有有限的知识；每个单元只能和它的朋友交谈；只和自己直接的朋友交谈
6. Interface Segregation Principle(ISP) 接口分离原则
7. Liskov Substitution Principle(LSP) Liskov替换原则(子类型必须能够替换掉他们的基类型)
8. Favor Composition Over Inheritance 使用组合代替继承
9. Single Responsibility Principle 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、9针对降低继承耦合。

第 15 章

信息隐藏的含义
1. 封装类的职责，隐藏职责的实现
2. 预计将会发生的变更，抽象它的接口，隐藏内部实现机制

封装
分离接口与实现
1. 将数据和行为同时包含在类中
2. 分离对外接口和内部实现
实现细节：
1. 封装数据与行为
2. 封装内部结构
3. 封装其他对象的引用
4. 封装类型信息
5. 封装潜在变更

OCP（Open Close Principle）
开闭原则
1. 好的设计应该对“扩展”开放
2. 好的设计应该对“修改”关闭
在发生变更时，好的设计只需要添加新的代码而不需要修改原有的代码，就能够实现变更

DIP（Dependency Inversion Principle）

依赖倒置原则

1. 抽象不应该依赖于细节，细节应该依赖于抽象。因为抽象是稳定的，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。因为抽象是稳定的，而高层模块和低层模块都可能是不稳定的

第 16 章

如何实现可修改性、可扩展性、灵活性

通过接口与实现的分离来实现：

1. 通过接口和实现该接口的类完成接口与实现的分离
2. 通过子类继承父类，将父类的接口和子类的实现相分离

（给定场景，应用设计模式并写出代码 / 给出代码，要求用设计模式改写）

策略模式 P266

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。

图 16-8　策略模式

抽象工厂模式 P272

为应对灵活性要求，我们提供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

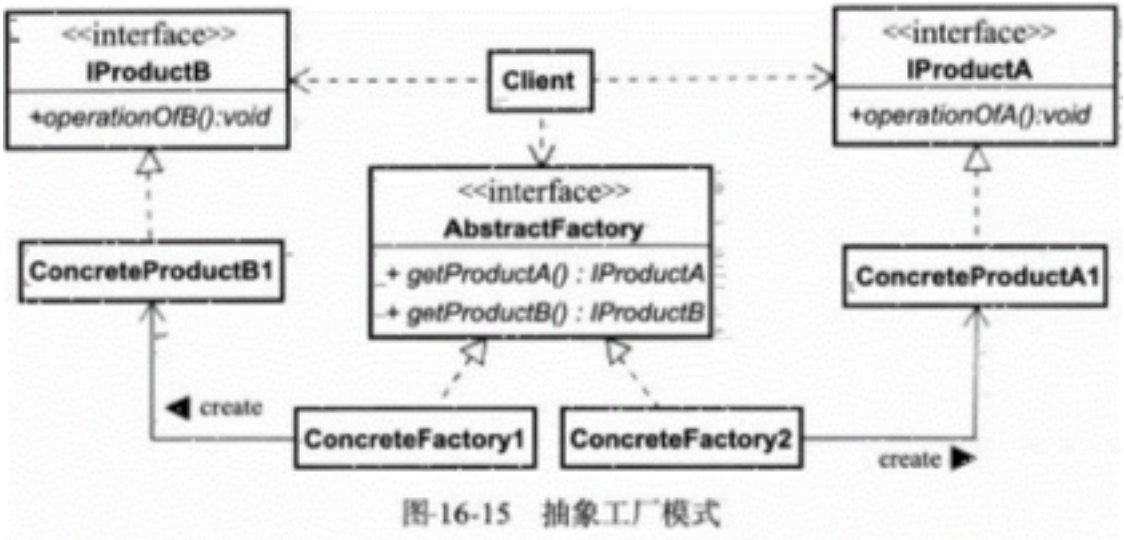
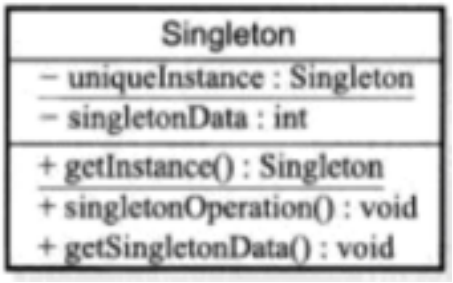


图-16-15 抽象工厂模式

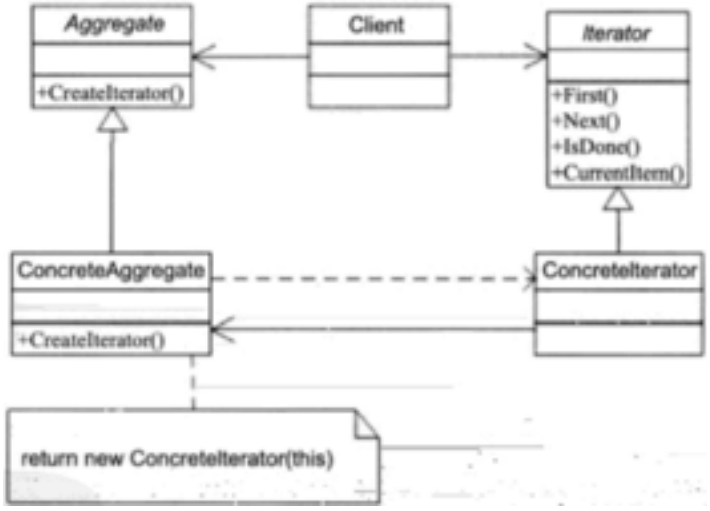
单件模式 P276

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。



迭代器模式 P279

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，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。



第 17 章

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，构造与管理。

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内部结构。不改变代码的外部表现是指不改么软件系统的功能，改进代码的内部结构是指提升详细设计结构的质量，使其能够进一步演化。

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。并在编程中重复执行测试代码，以验证程序的正确性。

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。

第 18 章

给定代码段示例，对其进行改进或者发现其中的问题
简洁性/可维护性
使用数据结构消减复杂判定
控制结构
变量使用
语句处理
How to write unmaintainable code
防御和错误处理

单元测试用例的设计
1. 基于规格的测试技术开发测试用例，等价类划分和边界值分析。
2. 基于代码的测试技术开发测试用例，关键复杂代码：路径覆盖；复杂代码：分支覆盖；简单代码：语句覆盖。

契约式设计 P309
契约式设计（Design by Contract）又称断言式设计
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，那么这个函数就是正确、可靠的。
有2种常见的编程方式：
异常方式：throw Exception
断言方式：assert Expression 1: (Expression 2)

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。
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（不仅仅是前置条件所包含的），不检查输出的后置条件，因为他们的使用者会自行检查。

表驱动 P307
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。

第 19 章

黑盒白盒测试
黑盒测试：
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
1. 等价类划分：把所有可能的输入数据，即程序的输入域划分为若干部分（子集），从每个子集中选取少量具有代表性的数据作为测试用例。
2. 边界值分析：对定价类划分的补充，错误容易发生在各个等价类的边界上，而不是等价类的内部，因此针对边界情况设计测试用例，可以发现更多的缺陷。
3. 决策表：为复杂逻辑判断设计测试用例的技术。
4. 状态转换：针对复杂测试对象的测试技术。该类复杂测试对象对输入数据的反应是多样的，还需要依赖自身的状态才能决定。需要建立状态图，状态装换表。

白盒测试：
将测试对象看做是透明的，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
1. 语句覆盖：确保被测试对象的每一行代码都至少执行一遍。（相对而言，比后两种覆盖率弱）
2. 条件覆盖：确保程序中没个判断的结果都至少满足一次。（保证每个条件都被覆盖了，比语句覆盖覆盖程度强，但不能保证所有执行路径）
3. 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。（覆盖度最强，代价大）

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
给出功能需求，则要求写功能测试用例
给出设计图，则要求写集成测试用例，Stub and Driver
给出方法的描述，则要求写单元测试用例，Mock Object
JUnit基本使用方法

第 20 章

用户文档、系统文档

用户文档是指为用户编写参考指南或者操作教程，常见的有用户使用手册、联机帮助文档等。与用户文档注重系统使用细节不同，系统管理员文档更注重系统维护方面的内容。

第 21 章

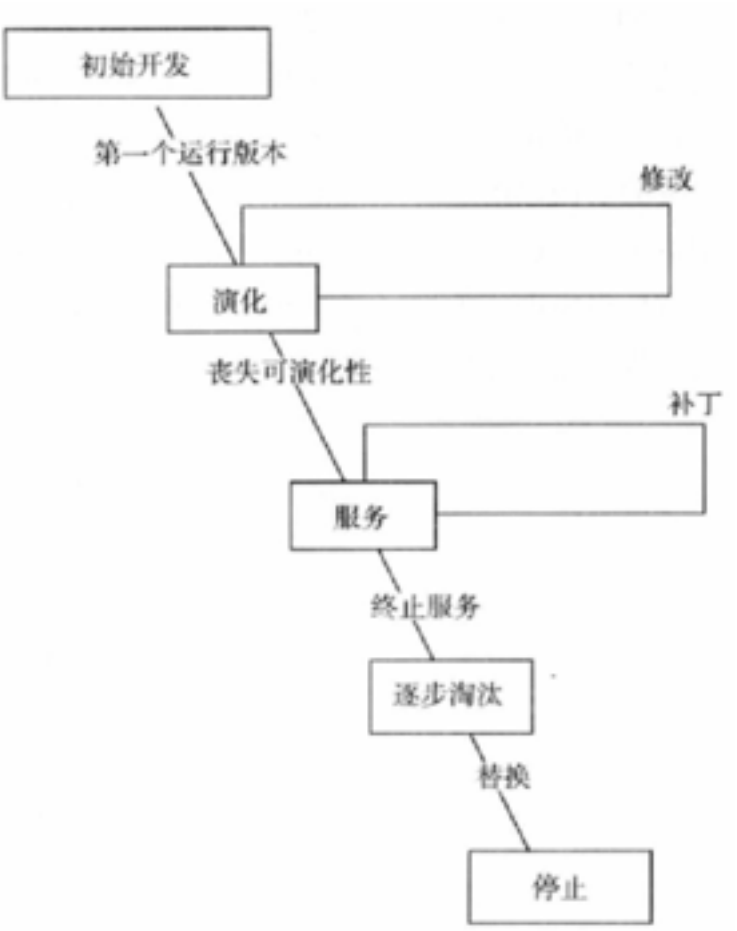
软件维护的重要性

人们需要经常“修改”软件，修改软件的代价是非常高的，这使得软件维护将其工作重点放在了软件修改和变更上。

开发可维护软件的方法

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

演化式生命周期模型

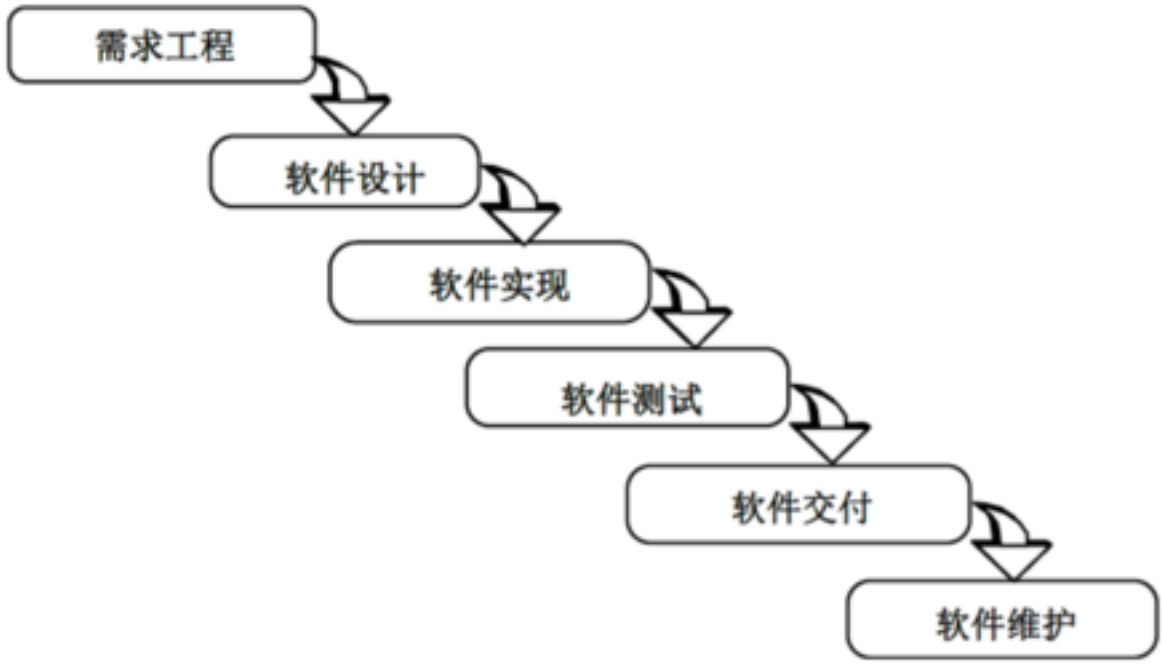


1. 初始开发：按照传统的软件开发方式完成第一个版本的软件产品开发。
2. 演化：保持软件产品的持续增值，让软件产品能够满足用户越来越多的需要，实现更大的业务价值。
3. 服务：不再持续地增加自己的价值，只是周期性地修正已有的缺陷。
4. 逐步淘汰：不再提供软件产品的任何服务，即不再继续维护该软件。
5. 停止：一个软件正式退出使用状态之后就进行停止状态。

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，并且使用其他形式或者更高层的抽象创建系统表现的过程。
逆向工程的主要关注点是理解软件，但并不修改软件。
再工程：关注如何修改软件，不会花费很大气力理解软件。再工程的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。

第 22 章

软件生命周期模型



解释与比较不同模型（要求、特征描述、优点、缺点）
构建–修复模型：
最早最自然产生的软件开发模型。没有规划组织,完全依靠开发人员的个人能力。
缺点：
1. 没有分工，随着软件系统复杂度上升，开发活动超出个人控制能力。
2. 没有分析需求的真实性，给软件开发带来很大的风险。
3. 软件结构的质量随着修改越来越差。

4. 没有考虑可维护性。

使用范围：

1. 软件规模很小，只有几百行程序。
2. 对软件质量要求不高，即使出错也无所谓。
3. 只关注开发，对后期维护的要求不高。

瀑布模型：

按照软件生命周期模型将软件开发活动组织为需求开发、软件设计、软件实现、软件测试、软件交付和软件维护等基本活动，并且规定了它们自上而下、相互邻接的次序。允许反复和迭代，重点在于每个活动必须进行验证，文档驱动的。

缺点：

1. 对文档期待过高。
2. 对开发活动的线性顺序假设是不符合实际情况的。
3. 客户用户参与不够，只有在需求一个阶段用户参与。
4. 里程碑粒度过粗，现在软件系统复杂，每个阶段持续周期较长。

优点：

1. 为软件开发活动定义了清晰的阶段划分（包括了输入/输出、主要工作及其关注点），降低了复杂度。

使用范围

1. 需求非常成熟、稳定。
2. 所需的技术成熟、可靠。
3. 复杂度适中，不至于产生太大的文档负担和过粗的里程碑。

Note：周期过长和渐进演化。时间压力和并行开发

增量迭代模型：

项目开始时，通过系统需求的开发和核心体系结构设计活动完成项目前景和范围的商定，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。需求驱动的。

优点：

1. 迭代式开发符合软件开发的实际情况，更好的适用性。
2. 并行开发可以缩短产品的开发时间。
3. 渐进交付可以加强用户反馈，降低开发风险。

缺点：

1. 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，要求具有开放式的体系结构。
2. 增量交付模型需要一个完备、清晰的项目前景和范围以进行并发开发规划，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。

适用范围:

1. 大规模软件开发，比较成熟和稳定的领域。

演化模型：

演化模型将软件开发活动组织为多个迭代、并行的瀑布式开发活动。

优点：

1. 迭代开发，尤其适用于需求变更频繁或不确定性较高的系统。
2. 并行开发
3. 渐进交付

缺点：

1. 无法在项目早期阶段确定项目范围，项目的整体计划，商务协商难以把握。
2. 体系结构难做，体系结构几乎确定会发生变更。
3. 容易退化为构建修复方式。

使用范围：

1. 不稳定领域的大规模软件系统。

原型模型：

在整体安排迭代的情况下，强调“抛弃式原型”的演化模型。抛弃式原型解决对未来知识的局限性产生的不确定性，将未来置于现在进行推敲。

优点：

1. 加强了和用户、客户的交流，提高满意度
2. 适用于新颖的领域。

缺点：

1. 原型开发带来风险：如成本，耗尽项目时间资金
2. 不舍得抛弃原型，使得低质量的代码进入最终产品导致产品低质量。

使用范围：

1. 大量不确定的新颖领域。

螺旋模型：

螺旋模型将软件开发活动组织为风险解决的迭代。

优点：

1. 降低风险

缺点：

1. 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
2. 模型过于复杂，不利于管理。

使用范围：

1. 高风险的大规模软件系统开发。

Rational统一过程：

RUP总结和借鉴传统上的各种有效经验，建立最佳实践方法的集合，并提供有效的过程定制手段，允许开发者根据特定的需要定制一个有效的过程模型。

核心思想：

1. 迭代式开发，这是过去被反复证明的最佳实践方法；
2. 管理需求，重视需求工程中除了需求开发之外的需求管理活动；
3. 使用基于组件的体系结构，它帮助建立一个可维护、易开发、易复用的软件体系结构；
4. 可视化建模，利用UML进行建模；

5. 验证软件质量，尽早和持续地开展验证，以尽早发现缺陷，降低风险和成本；

6. 控制软件变更，适应1990s以后需求变更越来越重要的事实。

优点：

1. 吸收和借鉴了传统上的最佳实践方法，尤其是其核心的6个实践方法，能够保证软件开发过程的组织是基本有效和合理的。

2. RUP依据其定制机制的不同，可以适用于小型项目，也可以适用于大型项目的开发，适用面广泛。

3. RUP有一套软件工程工具的支持，这可以帮助RUP的有效实施。

缺点：

1. 没有考虑交付之后的软件维护问题；

2. 裁剪和配置工作不是一个简单的任务，无法保证每个项目都能定制一个有效的RUP过程

适用范围：

1. RUP是重量级过程，能够胜任大型软件团队开发大型项目时的活动组织。但RUP经过裁剪和定制，也可以变为轻量级过程，也能够胜任小团队的开发活动组织。

敏捷过程：

针对传统过程模型的缺陷和新的形势，人们总结实践中的经验和最佳实践方法，尝试建立轻量级过程方法。

适用性：从敏捷联盟声明的思想和原则来看，它们反映了1990s之后软件工程的发展趋势，所以得到了广泛的应用，尤其是能够适应于快速变化或者时间压力较大的项目。

（对给定的场景，判定适用的开发过程模型）

软件工程知识体系的知识域

软件需求、软件设计、软件构造、软件测试、软件维护、软件配置管理、软件工程管理、软件工程过程、软件工程工具和方法、软件质量、软件工程职业实践、工程经济学基础、计算基础、数学基础、工程基础。