一、软件开发的生命周期

软件开发的生命周期分为需求分析、概要设计、详细设计、编码实现、软件测试、软件部署、软件维护等阶段

1. 传统开发模式--瀑布模型

- 在瀑布模型中，严格的把软件项目的开发分隔成各个阶段：需求分析、要件定义、基本设计、详细设计、编码、单体测试、结合测试、系统测试等。
- 使用里程碑的方式，严格定义了各开发阶段的输入和输出。如果达不到输出的要求，下一阶段的工作就不展开。
- 强调文档，在开发的后期才会看到软件的模样。
- 各阶段的开发人员只能接触到自己工作范围内的东西，从而提高开发效率。
- 既然叫做瀑布，就意味着不应该走回头路，否则如果出现返工，付出的代价会很大。

2. 需求分析

2.1 概述

把软件计划期间建立的软件可行性分析求精和细化，确定对系统的综合要求。分析系统的数据要求，导出系统的逻辑模型，修正系统的开发计划。

2.2 三个阶段

1.需求提出：开发人员和用户确定一个问题领域，并定义一个描述该问题的系统，在用户和开发人员之间充当合同。

2.需求描述：对用户的需求进行鉴别、综合和建模，清除用户需求的歧义性。

3.需求评审：分析人员要在用户和软件设计人员的配合下对需求规格说明书和初步的用户手册进行复核与更正。

2.3 四个需求

1.业务需求：反映了客户对系统高层次的目标要求，他们在项目视图与范围文档中予以说明。

2.用户需求： 文档描述了用户使用产品必须要完成的任务，这在使用实例文档或方案脚本说明中予以说明。

3.功能需求：定义了开发人员必须实现的软件功能，是的用户能够完成他们的任务。

4.非功能需求：包括产品必须遵从的标准；外部界面的具体细节；性能要求等。

注：

需求并未包括设计细节、实现细节、项目计划信息或测试信息。

2.4 需求说明书

1.引言：编写目的、背景、软件定义、参考资料

2.任务概述：目标、用户的特点、约束（经费限制、开发期限等）

3.需求规定：

对功能的规定：逐项阐述功能要求，说明输入、处理过程、输出等。

对性能的规定

输入输出要求：各输入输出数据类型的格式、数值范围、精度等。

数据管理能力要求：要按可预见的增长对数据及其分量的存储要求做估算。

故障处理要求：列出基本的可能出现的软件、硬件故障、后果和故障处理。

其他专门要求：可维护性、易读性、可靠性、运行环境等。

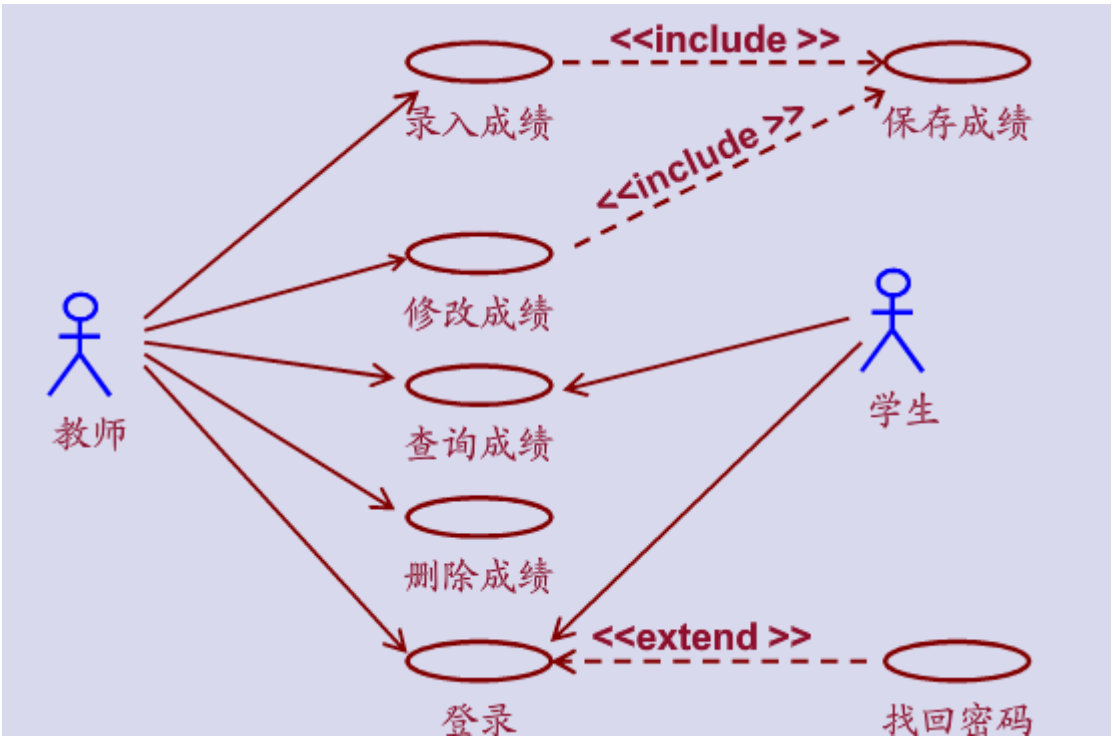
4.运行环境要求：设备、支持软件、接口等。

2.5 UML

2.5.1 用例图

由参与者（Actor）、用例（Use Case）以及它们之间的关系构成的用于描述系统功能的静态视图。

- 参与者：人或其他系统
- 用例：参与者可以感受到的系统服务或功能单元
- 系统边界：系统与系统之间的界限
- 关联：可以在用例之间抽象出以下几种关系：包含（Include）、扩展（Extend）、以及泛化（Generalization）等关系。



2.5.2 活动图

描述活动的顺序，展现从一个活动到另一个活动的控制流。活动图在本质上是一种流程图。

- 用例动作状态：指原子的、不可中断的动作，并在此动作完成后通过完成转换转向另一状态。
- 动作流：用带箭头的直线表示活动图转换
- 动作状态约束：前置条件和后置条件等。
- 关联其他：开始节点、终止节点、数据存储对象、分支合并、分支汇合、异常处理等。

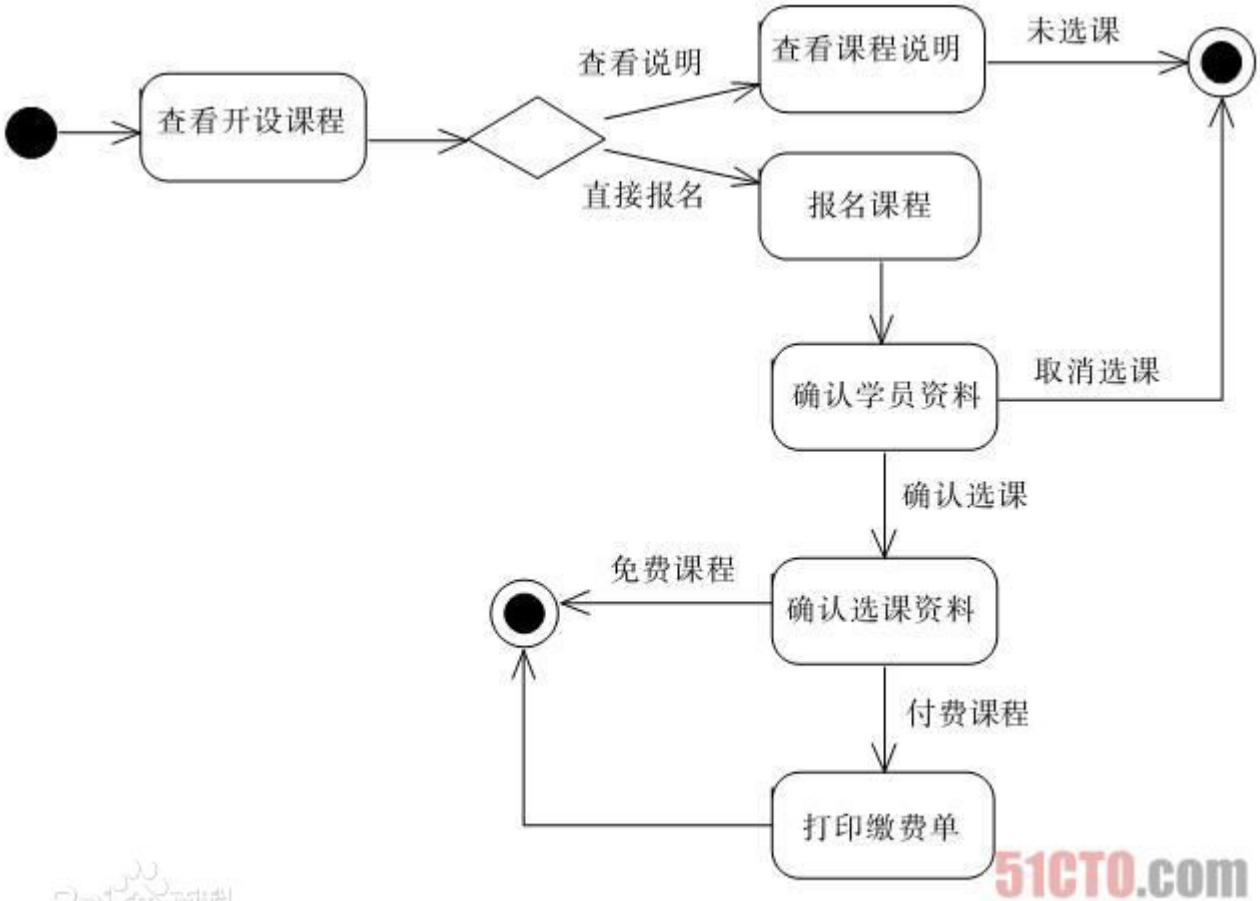


图 1-56　选课流程的活动图

3 概要设计

3.1 概述

根据用户交互过程和用户需求来形成交互框架和视觉框架的过程，其结果往往以反映交互控件布置、界面元素分组以及界面整体板式的页面框架图的形式来呈现。

3.2 具体任务

- 总体设计：说明本系统的基本设计概念和处理流程；用一览表及框图的形式说明本系统的系统元素（各层模块、子程序、公用程序等）的划分；运行环境说明。
- 功能设计：模块划分、建立模块的层次结构及调用关系。
- 接口设计：人机界面；软硬件接口；与其他系统的接口；系统内各元素间接口定义。
- 数据结构设计：包括数据特征的描述、确定数据的结构特性、以及数据库的设计。
- 出错处理：用一览表的方式说明每种可能的出错或故障情况出现时，系统输出信息的形式、含义及处理方法。
- 运行设计：对系统施加不同的外界运行控制时所引起的各种不同的运行模块组合；每种运行模块组合将占用的资源与时间。

3.3 概要设计说明书

1.引言

2.总体设计：运行环境、基本设计概念和处理流程

3.接口设计：用户接口、外部接口、内部接口

4.系统数据结构设计：逻辑结构设计要点、物理结构设计要点

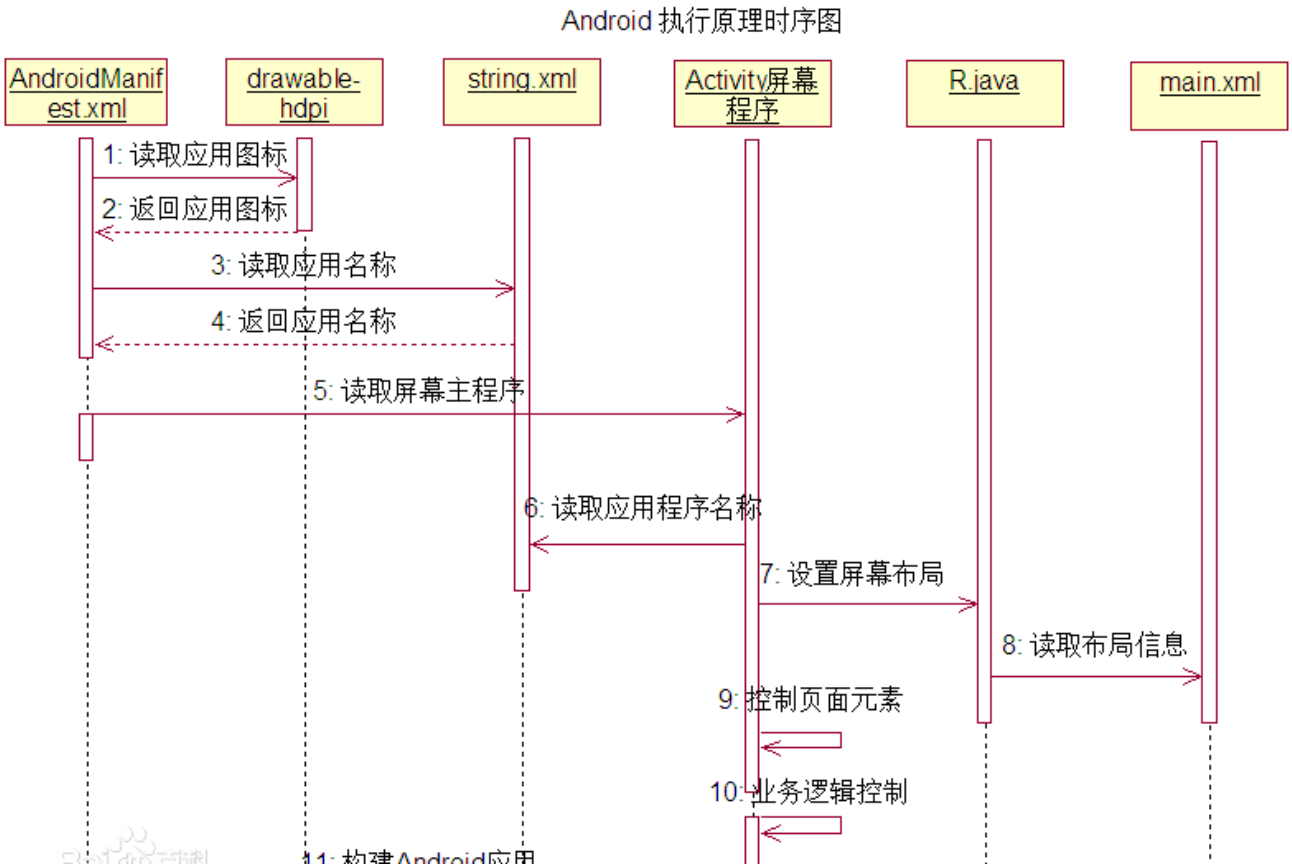
5.系统出错处理设计：出错信息、补救措施、系统维护设计

3.4 UML

3.4.1 时序图

对对象之间传送消息的时间顺序的可视化表示。由对象、生命线（Lifeline）、激活（Activation）、消息、分支与从属流等元素构成的。

- 对象：类的实例。三种状态：激活、运行（存在）和销毁
- 生命线：一条垂直的虚线，用来表示序列图中的对象在一段时间内的存在。
- 激活：表示一个对象直接或通过从属操作完成操作的过程
- 消息：对象间的一种通信机制
- 分支：从同一点出发的多个消息并指向不同的对象
- 从属流：从同一点发出多个消息指向同一个对象的不同生命线。



3.4.2 协作图

显示了一系列的对象和在这些对象之间的联系以及对象间发送和接收的消息。

与时序图的区别：

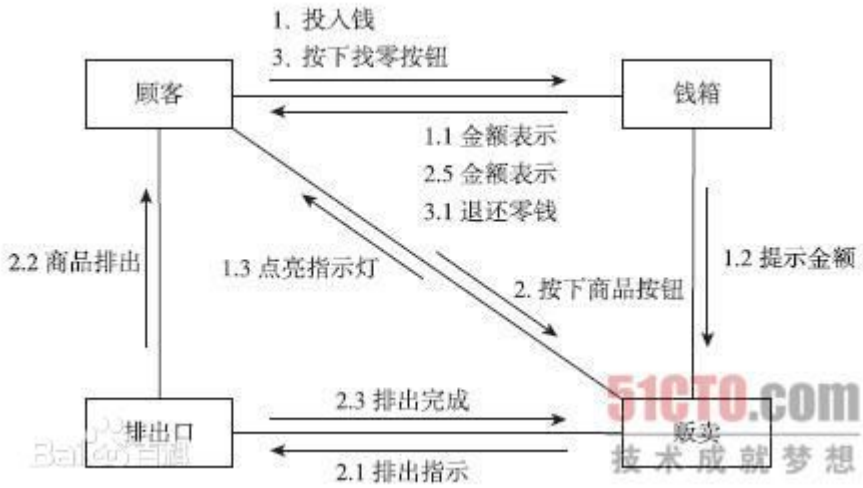
时序图：强调时间和序列

协作图：强调上下文相关

协作图显示对象之间的关系，更有利于理解对给定对象的所有影响

基本组成元素：活动者、对象、连接和消息。在UML中，使用实线标记两个对象之间的连接。

对象与消息：用长方形框表示对象。当两个对象间有消息传递时用带箭头的有向边连接这两个对象。



4 详细设计

4.1 概述

设计每个模块的实现算法、所需的局部数据结构。详细设计的目标有两个：实现模块功能的算法要逻辑上正确和算法描述要简明易懂。

4.2 基本任务

- 为每个模块进行详细的算法设计；为模块内的数据结构进行设计；为数据结构进行物理设计
- 其他设计：代码设计。输入输出格式设计；人机对话设计。
- 编写详细设计说明书

4.3 概要设计和详细设计的区别

概要设计实现软件的总体设计、模块划分、用户界面设计、数据库设计等。

详细设计则根据概要设计所做的模块划分，实现各模块的算法设计，实现用户界面设计、数据结构设计的细化。

4.4 概要设计和详细设计的联系

- 概要设计是详细设计的基础
- 概要设计里的功能应该是重点的功能描述，对需求的解释和整合，整体划分功能模块
- 详细设计重点在描述系统的实现方式，详细说明各模块实现功能所需的类及具体的方法函数。

4.5 详细设计说明书

1.引言

2.程序系统的结构

3.程序X的设计说明：从本章开始逐个地给出各个层次中的每个程序的设计。

程序描述

功能

性能

输入项/输出项

算法

流程逻辑

接口

存储分配

注释设计

测试计划

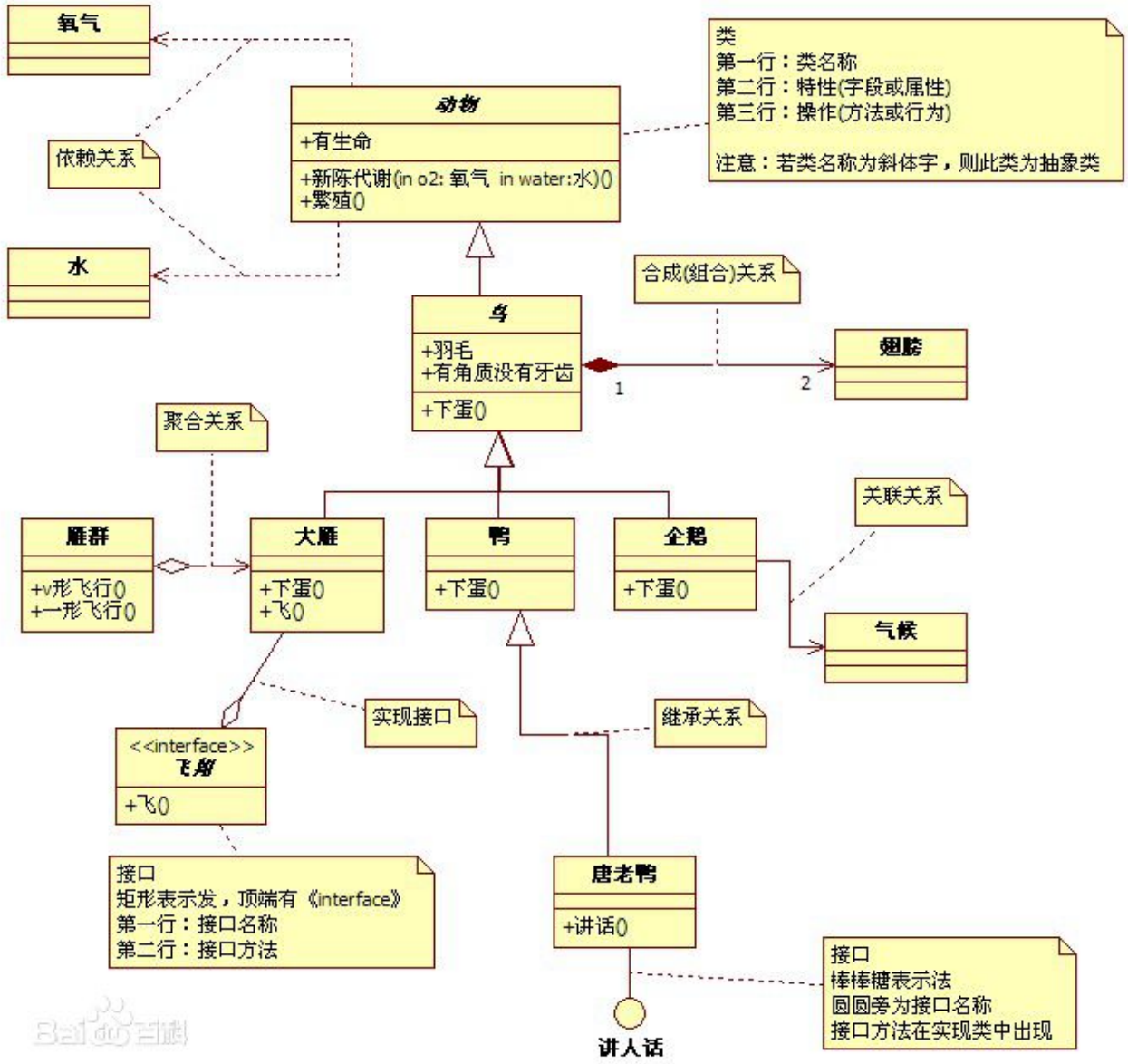
尚未解决的问题

4.6 UML

4.6.1 类图

显示了模型的静态结构，模型中存在的类、类的内部结构以及它们与其他类的关系等。类图不显示暂时性信息。

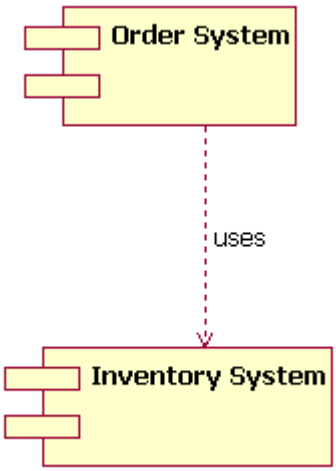
- 类：一般包含3各组成部分。第一个是类名；第二个是属性；第三个是该类提供的方法。此外，类的性质可以放在第四部分；如果类中含有内部类，则会出现第五个组成部分。
- 包：UML中的一个包直接对应于Java中的一个包。在Java中，一个包可能含有其他包、类或者同时含有这两者。
- 关系：常见的关系有：继承关系、关联关系、聚合关系、复合关系、依赖关系。



4.6.2 组件图

组件是系统中可替换的物理部分，它包装了实现而且遵从并提供一组接口的实现。组件图即为对该实现的描述。它包含组件、接口、实现和依赖四部分。

- 组件：系统的一个可执行程序，一个库，一个web程序等。
- 接口：组件所提供服务，可以理解为一个方法，一个webservice，一个WCF，或者一个UI界面，接口可以有多个，但至少有一个。
- 实现：组件与接口元之间的连线，代表谁现实了这个接口。
- 依赖：指组件使用了另一个组件的接口，依赖与另一个接口的存在。



5 编码规范

5.1 命名规范

- 采用大小写混合使名字可读
- 尽量少用缩写，但如果用了，要明智地使用，且在整个工程中统一
- 避免使用长的名字（一般小于15个字母）
- 避免使用类似的名字，或者仅仅是大小写不同的名字
- 避免使用下划线（除静态常量等）
- 包：由小写字母组成
- 类：首字母大写
- 方法：以小写字母开头，中间单词的首字母大写，无下划线。方法的名字建议动词短语
- 类变量、字段：完整的英文描述，首字母大写，中间单词的首字母大写
- 静态常量字段：全大写字母，单词间用下划线分隔
- 缩进规范：行的缩进要求死四个空格。不推荐使用Tab，这是由于在使用不同的源代码管理工具时，Tab字符将因为用户设置的不同而扩展为不同的宽度。建议将Tab字符扩展为4个空格。

6 软件测试

6.1 概述

在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。

6.2 测试目的

- 测试是为了发现程序中的错误而执行程序的过程
- 成功的测试是发现了至今为止尚未发现的错误的测试
- 测试并不仅仅是为了找出错误，通过分析错误产生的原因和错误的发送趋势，可以帮助及时改进系统性能，及改善测试效率和有消息
- 没有发现错误的测试也是有价值的，完整的测试是评定软件质量的一种方法

6.3 测试原则

- 测试应尽早进行，最好在需求阶段就开始介入

- 软件测试的实施与结果确认应该由第三方来负责，避免由程序员检查自己的程序
- 设计测试用例时应考虑到合法与不合法的输入，及各种边界条件，可适当制造极端状态，如网络异常中断，电源断电等。
- 制定严格的测试计划。并妥善保存测试计划、测试用例、出错统计和最终分析报告，为维护提供方便。

6.4 测试划分

测试阶段划分：单元测试、集成测试、确认测试、系统测试、验收测试、回归测试、Alpha测试、Beta测试

角度划分：白盒测试、黑盒测试、灰盒测试

由是否执行划分：静态测试、动态测试

6.5 单元测试

对软件中的最小可测试单元进行检查和验证。所谓最小可测试单元可指Java中的类、方法、模块、用户界面中的窗口等人为规定的最小的被测功能模块。单元测试是由程序员自己来完成。

6.5.1 测试时机

单元测试越早越好，可适当采取测试驱动开发方式：先编写测试代码，在进行开发。在实际的工作中，可先编写产品函数的框架，然后编写测试函数，针对产品函数的功能编写测试用例，然后编写产品函数的代码，每写一个功能点都运行测试，随时补充测试用例。

6.5.2 桩模块

桩代码就是用来代替某些代码的代码，可用于实现测试隔离。但单元测试应避免编写桩代码。实际工作中，可先开发底层的代码并先测试。这样做的好处有：减少工作量；测试上层函数时，也是对下层函数的间接测试；当下层函数修改时，通过回归测试可以确认修改是否导致上层函数产生错误。

6.5.3 相关开发活动

单元测试联系起来的另外一些相关开发活动包括代码浏览、静态分析和动态分析。

- 静态分析：对软件的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和执行
- 动态分析：通过观察软件运行时的动作，来提供执行跟踪，时间分析，以及测试覆盖度方面的信息。

6.5.4 常见误解

“单元测试是浪费时间”或者“集成测试将会抓住所有的Bug”都是开发人员常有的误解，但这些观念都不正确。实践表明，严格的单元测试可以提早跟踪并纠正Bug，有效地缩短项目的总开发时间，并保证高质量的软件交付。

6.6 集成测试

也叫组装测试或联合测试。在单元测试的基础上，将所有模块按照设计要求（如根据结构图）组装成为子系统或系统，进行集成测试。实践表明，一些模块在某些局部反映不出来的问题，在全局上很可能暴露出来，影响功能的实现。

6.6.1 简单的形式

把两个已经测试过的单元组合成一个组件，测试它们之间的接口。它要求在组合单元前每个单元都得到了有效地测试并通过测试，从而得知所发现的任何错误都很可能与单元之间的接口有关。

6.6.2 常用测试手段

- 功能性测试，使用黑盒测试技术针对被测模块的接口规格说明进行测试
- 非功能性测试，对模块的性能或可靠性进行测试。

6.6.3 常用方案

自底向上集成测试，自顶向下集成测试、Big-Bang集成测试、三明治集成测试、核心集成测试、分层集成测试、基于使用的集成测试等。

6.6.4 完成标准

- 成功地执行了测试计划中规定的所有集成测试
- 修正了所发生的错误
- 测试结果通过了专门小组的评审

6.7 系统测试

将已经确认的软件、计算机软件、外设、网络等其他元素结合在一起，进行信息系统的各种组装测试和确认测试。

6.7.1 目的

验证系统是否满足了需求规格的定义，找出与需求规格不符合或与之矛盾的地方，从而提出更加完善的方案。

6.7.2 测试范围

基于系统整体需求说明书的黑盒类测试，应覆盖系统所有联合的部件。对象不仅仅包括需测试的软件，还要包含软件所依赖的硬件、外设甚至包括某些数据、某些支持软件及接口等。

6.7.3 分类

- 恢复测试：关注导致软件运行失败的各种条件，并验证其恢复过程是否正确执行。在特定情况下，系统需具备容错能力
- 安全测试：验证系统内部的保护机制，以防止非法侵入。准则是使侵入系统所需的代价更加昂贵。
- 压力测试：压力测试是指在正常资源下使用异常的访问量、频率或数据量来执行系统

6.7.4 缺陷管理与改错

发现任何缺陷时都必须使用指定的“缺陷管理工具”。该工具记录所有缺陷的状态信息，并可以自动产生缺陷管理报告。

- 开发人员及时消除已经发现的缺陷
- 开发人员消除缺陷之后应当马上进行回归测试，以确保不会引入新的缺陷

6.8 确认测试

在模拟环境下，运用黑盒测试的方法，验证被测试软件是否满足需求规格说明书列出的需求。

6.8.1 测试内容

安装测试、功能测试、可靠性测试、安全性测试、时间及空间性能测试、易用性测试、可移植性测试、可维护性测试、文档测试。

6.8.2 基本方法

基于系统整体需求说明书的黑盒类测试，应覆盖系统所有联合的部件。对象不仅仅包括被测试的软件，还要包括软件所依赖的硬件、外设甚至包括某些数据、某些支持软件及接口等。

- 确认测试标准：
  - 考虑软件是否满足合同规定的所有功能和性能，文档完整、准确人机界面和其他方面（可维护性等）是否令用户满意
  - 这个阶段才发现的严重错误一般很难在预定的工期内改正，因此必须与用户协商，寻求一个妥善解决的方法
- 配置复审：保证软件配置齐全、分类有序，并且包括软件维护所必须的细节。

6.8.3 测试类别

- α测试：软件开发公司组织内部人员模拟各类用户进行对即将面市的软件产品（称为α版本）进行测试，试图发现错误并纠正
- β测试：软件开发公司组织各方面的典型用户在日常工作中实际使用β版本，并要求用户报告异常情况、提出批评意见。

6.9 回归测试

修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误

6.9.1 目的

自动回归测试将大幅降低系统测试、维护升级等阶段的成本。软件开发的各个阶段都会进行多次回归测试。在极端的编程方法中，更是要求每天都进行若干次回归测试。

6.9.2 观念

- 以关键性模组为核心，重复以前的全部或部分的相同测试
- 新加入测试的模组，可能对其他模组产生副作用，故须进行某些程度的回归测试

6.9.3 测试策略

- 测试用例库的维护：删除过时、冗余的测试用例；改进不受控制的测试用例；增添新的测试用例。
- 回归测试包的选择：再测试全部的用例；或者基于风险选择测试；或者再测试修改的部分

6.9.4 测试过程

1. 识别出软件中被修改的部分
2. 从原基线测试用例库T中，排除所有不再适用的测试用例，并建立一个新的基线测试用例库T0
3. 依据一定的策略从T0中选择测试用例测试被修改的软件
4. 如果必要，生成新的测试用例集T1，用于测试T0无法充分测试的软件部分
5. 用T1执行修改后的软件