软件工程

0. 面向考试

各章考点

第一章:

- [ ] SE基础知识
- [ ] （非重点）开发模式/范式
- [ ] 错误、缺陷与失败
- [ ] 软件质量
- [ ] （非重点）软件系统的系统组成
- [ ] 现代软件工程的几个阶段
- [ ] （非重点）现代软件工程实践发生变化的关键因素
- [ ] 软件过程，重要性，各个阶段
- [ ] 重用、抽象等概念

第二章:

- [ ] 软件过程，重要性，生命周期
- [ ] 瀑布模型及各阶段文档，优缺点
- [ ] 原型的概念和用途
- [ ] 分阶段开发模型的含义，基本分类及特点
- [ ] 螺旋模型四个象限的任务及四重循环的含义
- [ ] UP，RUP，进化式开发的含义

- [ ] 敏捷方法及其代表性方法

第三章：

- [ ] 项目进度，活动，里程碑，项目成本
- [ ] 如何计算软件项目活动图的关键路径，冗余时间，最早和最迟开始时间
- [ ] 软件团队人员应该具备的能力
- [ ] 软件项目团队组织的基本结构
- [ ] 专家估算法的大概含义
- [ ] COCOMO模型的三个阶段
- [ ] 软件风险，主要风险活动，降低风险策略

第四章：

- [ ] 需求的含义
- [ ] 需求阶段确定需求的过程
- [ ] 获取需求时，若有冲突发生如何考虑优先级
- [ ] （非重点）如何使需求变的可测试
- [ ] 需求文档的两类
- [ ] 功能性需求和非功能性需求，设计约束，过程约束
- [ ] （非重点）需求的特性
- [ ] DFD数据流图的构成及画法
- [ ] （非重点）抛弃型原型，演化型原型

第五章：

- [ ] 软件体系结构，设计模式，设计公约，设计，概念设计，技术设计
- [ ] 软件设计过程模型的几个阶段
- [ ] （非重点）三种设计层次及其关系
- [ ] （非重点）模块化，抽象

- [ ] 设计用户界面应考虑的问题
- [ ] 模块独立性，耦合与内聚
- [ ] 复审，设计复审

第六章：

- [ ] 面向对象的概念
- [ ] 设计模式
- [ ] OO设计的基本原则
- [ ] OO开发的优势
- [ ] OO开发过程的步骤
- [ ] 用例图的组成和画法
- [ ] 用例模型建模
- [ ] 类图的组成和画法
- [ ] （非重点）状态图的含义及用途

第七章：

- [ ] （非重点）为什么说编码工作是纷繁复杂甚至令人气馁的
- [ ] 一般性的编程原则
- [ ] （非重点）编码阶段实现某种算法所涉及的问题
- [ ] 编写程序内部文档时需要添加的注释信息
- [ ] 敏捷方法的大致思想，极限编程，派对编程

第八章：

- [ ] 软件产生缺陷的原因
- [ ] （非重点）软件缺陷进行分类的理由
- [ ] 几种主要的缺陷类型
- [ ] 正交缺陷分类法

- [ ] 测试的各个阶段及其任务，涉及的文档
- [ ] （非重点）测试的态度问题
- [ ] 掌握测试的方法
- [ ] 单元测试
- [ ] 黑盒测试方法的分类，测试用例的设计方法
- [ ] 白盒方法的分类，覆盖方法
- [ ] 集成测试及其主要方法的分类
- [ ] 传统测试和OO测试的区别
- [ ] （非重点）测试计划设计的步骤

第九章：

- [ ] 系统测试的主要步骤及各自含义
- [ ] （非重点）系统配置，系统配置管理
- [ ] 回归测试
- [ ] 功能测试的含义及其作用
- [ ] 功能测试的基本指导原则
- [ ] 性能测试的含义与作用
- [ ] 性能测试的主要分类
- [ ] （非重点）可靠性、可用性和可维护性
- [ ] 确认测试的概念，分类
- [ ] Alpha测试，Beta测试
- [ ] 安装测试

1. 软件工程简介

软件危机：开发软件所需的高成本和软件的低质量之间有尖锐的矛盾，其主要原因是**软件本身特点、缺乏好的开发方法和手段、开发效率低。**

基础知识

软件工程：用**系统科学**的**工程性方法**解决软件开发时遇到的问题，也就是，将**系统化的、严格约束的、可量化**的方法应用于软件的**开发、运行和维护**。需要着重记忆的部分有：

- 目的：规范软件**开发流程**，推出能解决实际问题的高质量的软件产品。
- 项目管理：将一个大的系统分为小的部分，明确开发过程，控制开发进度，应对种种变化。
- 作用：**降低开发成本，达到要求的软件功能、提高软件性能、提高软件的可移植性、降低维护费用、按时完成工作并交付使用。**

软件异常与质量

> 这部分存在一些争议

- 错误（Error）：在软件开发过程中的错误，停留在**需求工作和编码过程**中。

  > 例如：误解了需求，敲错了代码。

- 缺陷（Fault）：存在于软件（文档、数据、程序）之中的那些**不希望或不可接受的偏差**。由于错误而引起的，存在于某些功能实现处的问题，属于**功能实现**层面。

  > 例如：对需求的误解导致的需求文档错误和与设计意图不相符的设计，敲错代码导致的错误的功能实现 （缺陷是错误的结果/表现）。

  > 软件缺陷的主要特征：
  >
  > - 软件未达到软件产品需求说明书指明的要求。
  > - 软件出现了软件产品需求说明书中指明不应出现的错误。
  > - 软件功能超出软件产品说明书指明的范围。
  > - 软件未达到软件产品说明书未指明但应达到的要求。
  > - 软件测试人员认为难以理解、不易使用、运行速度慢或最终用户认为不好。

- 软件故障：在一个计算机程序中出现的不正确的步骤、过程或数据定义常称为故障。是指**软件运行时丧失了在规定的限度内执行所需功能的能力，执行输出错误结果，导致失效。**
- 软件失效：是指**软件运行时产生的一种不希望或不可接受的外部行为**，偏离了用户需求。

软件错误、缺陷、故障和失效：

- 软件错误是一种人为错误。**一个软件错误必定产生一个或多个软件缺陷。**
- 当一个软件缺陷被激活时，便产生一个软件故障；同一个软件缺陷在不同条件下被激活，可能产生不同的软件故障。
- 软件故障如果没有容错措施加以处理，便不可避免地导致软件失效；同一个软件故障在不同条件下可能产生不同的软件失效。

软件故障和失效的比较：

软件失效	软件故障
面向用户	面向开发者
软件运行时偏离用户需求	程序执行输出错误结果
根据对用户应用的严重性等级分类	根据定位和排除故障的难度分类

软件质量可以从以下三个方面衡量：

1. **产品质量**：从用户看来，高质量软件需要有充实的功能且便于学习上手从开发者来看，高质量软件需要有良好的内部特性，比如缺陷要少。
2. **过程质量**：许多软件开发活动都会引起最终产品质量的变化，所以过程也存在质量指标。我们有量化过程质量的参数，比如CMM。
3. **商业价值**：技术价值以技术指标衡量，如：速度、维护成本等然而软件的技术价值并不一定能转化为商业价值，公司并不一定会认为一个软件与自己的商业战略吻合，商业价值是一个独立的领域。

产品质量的标准包括：

- 用户：功能足够，易于上手。
- 开发者：衡量软件的内部特征，例如缺陷的数量。

过程质量的标准涉及**软件过程中发生的事件**，他们会影响过程的质量，并最终影响软件的质量，量化的方法包括CMM等。

CMM（Capability Maturity Model，能力成熟度模型）是一种用于评估和改进软件工程组织过程质量的方法。CMM定义了五个成熟度级别，每个级别描述了组织在软件过程管理方面的不同能力水平：

1. 初始级别（Level 1 - Initial）：过程是无序的，项目的成功依赖于个别人员的能力和努力。

> 2. 可重复级别（Level 2 - Repeatable）：过程已经被定义，并且在项目中得到了重复使用。组织开始建立基本的项目管理控制。
> 3. 定义级别（Level 3 - Defined）：过程已经被标准化，并在整个组织范围内得到了统一的应用。组织能够根据标准过程执行项目。
> 4. 管理级别（Level 4 - Managed）：过程已经被量化和统计，并进行了过程管理和控制。组织能够通过度量和分析来预测项目的结果。
> 5. 优化级别（Level 5 - Optimizing）：过程已经通过持续的改进得到优化。组织能够根据量化的数据和经验教训来改进过程。

商业价值的标准包括：

- 技术价值：各种技术指标。
- 商业价值：机构对于软件是否与其战略利益相吻合的一种战略评估。
- 目标：将技术价值和商业价值进行统一。

现代软件工程的构成

现代软件工程大致包含如下阶段（共九个）：

1. **需求定义与分析**：进行问题定义与可行性分析，得出文档：SRS（软件需求规格说明书）。
2. **系统设计**：设计用户界面，进行顶层设计，得出文档：SAD（软件体系结构图）。
3. **程序设计**：描述模块功能算法与数据描述，处于伪代码阶段，得出文档：模块功能与数据描述文档。
4. **程序实现**：进行编程与调试，得出文档：源代码、注释、源代码文档。
5. **单元测试**：按程序设计阶段的文档（模块功能与数据描述文档）进行测试。
6. **集成测试**：按SAD进行测试。
7. **系统测试**：按SRS进行测试，得出文档：以上三个测试得出各自阶段的测试报告。
8. **系统提交**：用户交付，必要时指导用户如何使用，得出文档：用户手册、操作手册。
9. **维护**：修改软件的过程，不断进行改错和满足新需求，得出文档：维护记录。

软件工程涉及的其他主要概念：

1. **重用**：重复采用以前开发的软件系统中具有的**共性部件**，用到新的开发项目中去。
2. **抽象**：基于**某种层次归纳水平的问题描述**。它使我们将注意力集中在问题的关键方面而非细节。
3. 分析、设计方法和符号描述系统：使用标准来对程序进行描述，利于交流和建模并检查其完整性和一致性，利于重用。
4. **用户界面原型化**：建立系统的小型版，通常具有有限的关键功能，以利于用户评价和选择，帮助我们确认关键需求，证明设计或方法的可行性。

5. 测度和度量：通用的评价方法和体系，有助于使过程和产品的特定特性更加可见，包括量化描述系统、量化审核系统。
6. 工具和集成环境：通过框架比较软件工程环境提供的服务，以决定其好坏，并且将各个阶段使用的工具集成起来。工具成必须处理一下五个问题：
    - 平台集成：工具必须在异构的网络中能相互操作相互配合
    - 表示集成：这些工具应当有统一的用户界面
    - 过程集成：工具与开发过程间要有联系
    - 数据集成：工具间要有共享数据的方式
    - 控制集成：工具间要有彼此交流，彼此通知对方并启动另一个工具的动作的能力

2. 项目过程建模与生命周期

软件过程是软件开发活动中**产生某种期望结果**的一系列**有序任务**，涉及**活动、约束和资源。**他的重要性体现在：

1. **通用性**：软件过程可以让一系列**开发活动保持一致性和结构性。**
2. **指导性**：软件过程使我们可以分析、检查、理解、控制和改善软件开发活动。
3. 可以把获得的**经验传递给其他人。**

> **软件生命周期是软件开发过程的别称。**它包括的就是这九个阶段；而每个阶段本身又是一个过程，由一系列活动构成，每个活动又涉及输入、子活动、约束、资源、输出。

开发模式

常用的软件开发模式有四种：1、瀑布开发模式；2、迭代式开发模式；3、螺旋开发模式；4、敏捷开发模式；

瀑布开发模式

在瀑布模型中，软件开发的**各项活动严格按照线性方式**进行，当前活动接受上一项活动的工作结果，实施完成所需的工作内容。当前活动的工作结果需要进行验证，如验证通过，则该结果作为下一项活动的输入，继续进行下一项活动，否则返回修改。

其**特点**在于：

- 阶段间的**依赖性和连续性。**
- 尽可能**推迟程序的物理实现。**
- 每个阶段**必须完成规定的文档。**

- 从**较高层次审视**开发活动。

瀑布模型的**优点**：

- 每个过程有里程碑，有提交物，项目经理可以评价。
- **简单性**：过程简单，容易对用户解释。
- **基础性**：是其他复杂模型的基础。

瀑布模型的**缺点**：

- 面临软件变动时，该模型无法处理**实际过程中的重复开发问题。**
- 文档转换有困难。

瀑布开发模式**各阶段及产生的文档**：

阶段	产生的文档
需求分析	SRS（软件需求规格说明书）
系统设计	SAD（系统设计文档）
程序设计	模块功能算法和数据描述文档
编码	源程序和注释
单元测试和集成测试	单元测试报告
系统测试	系统测试报告
验收测试	验收测试报告
运行与维护	维护报告

原型化开发模式

原型的定义：一种**部分开发**的产品，用来让开发者和用户**共同研究**，提出意见，为最终产品定型。其用途在于：

- 将**需求或设计**原型化，允许我们进行**改进。**
- 将**可替代的方案**原型化，允许我们进行**选择。**

> 原型化是**对瀑布开发模式的一定改进。**

V模式

V模式基于瀑布模型，让迭代更明显。

瀑布模型强调文档与提交产物，V模型强调**开发活动及正确性**，允许各种重复活动。（增加了各种针对性的措施）

分阶段开发模式

分阶段开发模型的定义：系统被设计成**部分提交**，用户每次都只能**得到部分功能**，而其他部分仍处于开发中。这种开发模型使得**每个软件版本的周期变短。**

根据发布形式的不同，分阶段开发模型有如下三种：

- 增量式开发：系统按需求分成**若干子系统**，一开始先建造出较小的、只具有部分功能的系统，后续版本逐步**添加包含新功能的子系统**，最终获得一个包含全部功能的子系统集合。
- 迭代式开发：一开始就提供**整体功能框架**，各种功能都可以用但是比较弱，后续版本**陆续增强各个子系统**，最终获得一个各个子系统功能达到最强的满血版。
- 增量-迭代融合：两者结合使用，新版本既有新功能，也完善旧有功能。

分阶段开发模型的特点（主打一个快）：

- 可以更快开始**训练用户**使用。
- 更快**投产**，开拓市场。
- 在开发过程整体的**早期就及时发现问题**并解决。
- 可以通过合理安排，让开发团队在**各个版本的开发中专注于某一个技术难点。**

螺旋开发模式

螺旋模型的**每一圈是一次迭代**，**每一个象限代表一重迭代中的某一部分任务。**

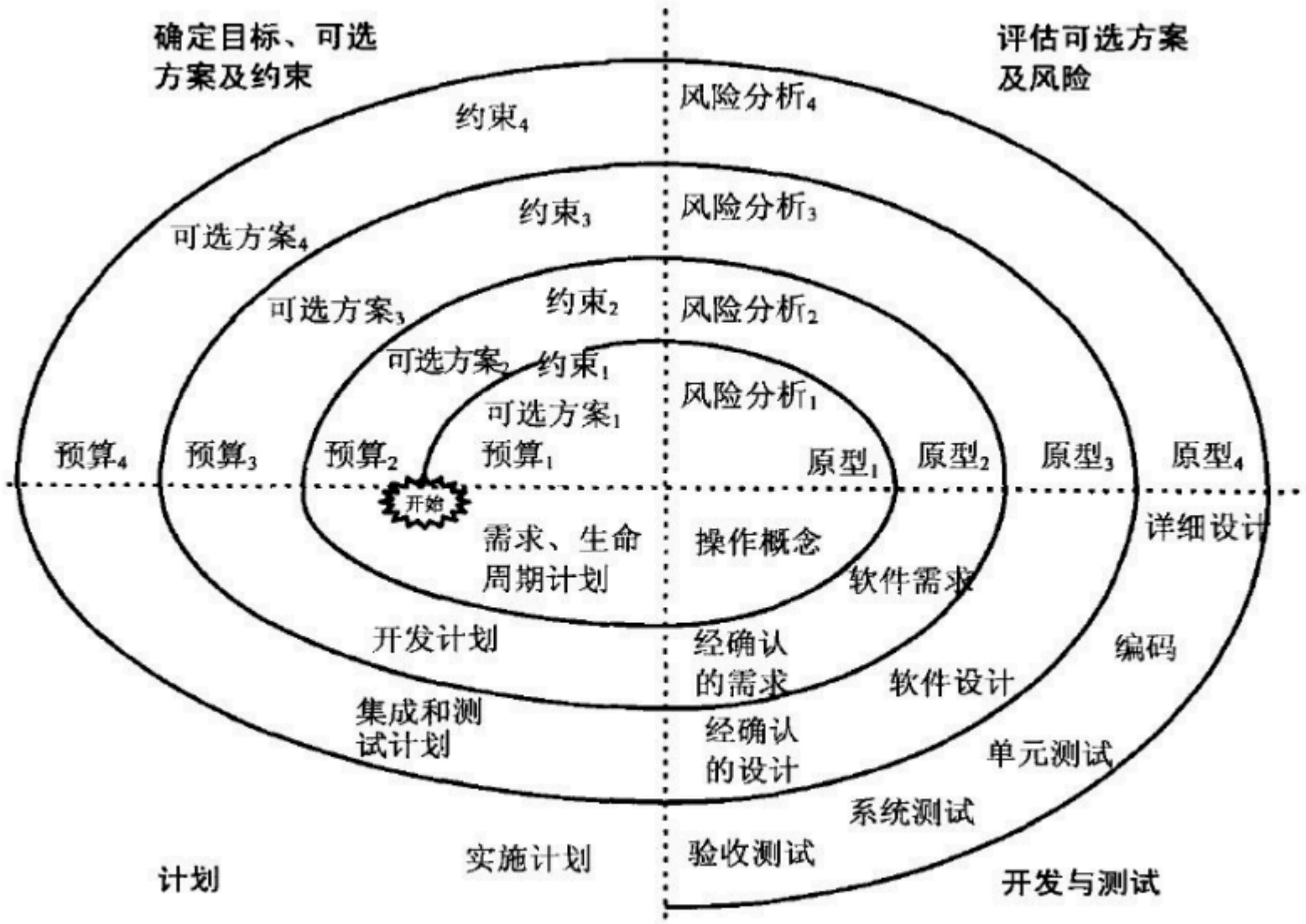


图2-10　螺旋模型

注意看，**每一个循环对应的内容存在差异。**特别是开发与测试和计划两个象限。

螺旋开发模式的四个象限有各自的含义：

- **计划**
- 确定**目标/可选方法**
- **风险评估**
- **开发与测试**

四重循环的含义：

- 获得**操作概念**
- 获得**软件需求**
- 获得**软件设计**
- 完成**开发、测试和验收**

UP/RUP

UP模型即统一过程模型，是一种**用例驱动**的，以**基础架构为中心**的，**迭代式、增量式**的软件开发模型。

> RUP模型是IBM提出的**RUP模型是IBM提出的提供支持和包装的UP模型。**

UP模型包含**四个阶段**：

- **开始阶段**：大体上的构想，业务案例，范围和模糊评估。定义**系统的业务模型，确定系统的范围。**完成后建立目标里程碑。
- **确立阶段**：已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围以及进行更为实际的评估。**完成系统的体系结构设计，完成系统开发计划。**建立结构里程碑。
- **构建阶段**：对遗留下的风险较低和比较简单的元素进行迭代实现，准备部署。构造产品，并继续演进需求、体系结构和计划，直到产品完成。
- **移交阶段**：进行系统部署，系统测试，最终移交给用户。最后建立发布里程碑。

UP模型的**六个核心工序**：

1. **业务模型工序**：通过业务模型获取相关知识以理解需要**系统自动完成的业务**（简单时称**问题定义或领域知识**）。
2. **需求工序**：通过用例模型获取相关知识以**理解自动完成业务的系统需求。**
3. **分析设计工序**，通过分析/设计模型以分析需求，设计系统结构。

4. **实现工序**：基于实现模型实现系统。
5. **测试工序**：通过测试模型进行针对需求的系统测试。
6. **部署工序**：通过部署模型部署系统。

UP模型的**三个支持工序**：

1. **配置变更管理工序**：用来管理系统和需求变更的配置。
2. **项目管理工序**：用来管理项目。
3. **环境配置工序**：用来配置项目的环境，包括所涉及到的过程和工具

进化式迭代开发是RUP的关键实践：

- 开发被组织成一系列固定的**短期小项目。**
- 每次选代都**产生经过测试、集成并可执行的局部系统。**
- 每次迭代都具有**各自的需求分析、设计、实现和测试。**
- 随着时间和一次次迭代，**系统增量式完善。**

敏捷开发模式

敏捷方法：一种较新型软件开发方法。不要求遵循传统的软件开发流程，强调**快速开发**和有效适应需求变化。

敏捷开发的**四个基本原则**：

- 个体和交互的价值胜过过程和工具（个人的卓越创意）。
- 可以工作的软件胜过面面俱到的文档（文档非软件）。
- 客户合作胜过合同谈判。
- 响应变化胜过遵循计划。

敏捷方法**强调**：人与人之间的交互是复杂的，并且其效果从来都是难以预期的，但却是工作中最重要的方面。其**目标**在于：**尽可能早的，持续的交付有价值的软件系统**，以客户满意为最终目标。

敏捷方法的最佳实践包括XP、结对编程、测试驱动开发等。

极限编程（XP）是一种轻量级的软件开发方法，属于敏捷开发方法。它将复杂的开发过程分解为一个个相对比较简单的小周期，通过交流、反馈等方法，开发人员和客户可以非常清楚开发进度、变化、待解决的问题和潜在的困难等，并根据实际情况及时地调整开发过程。

XP的**四个变量**：成本、时间、质量和范围。

XP的四个基本原则：

- **沟通**：客户与开发者之间持续的交流意见。
- **简单性**：鼓励开发者选择最简单的设计或实现来应对客户的需求。
- **反馈**：指在软件开发过程中的各个活动中，包含的各种反馈循环工作。
- **勇气**：指尽早的和经常性的交付软件功能的承诺。

派对编程/结对编程：两个程序员共同开发程序，且角色**分工明确。**一个负责编写程序，另一个负责复审与测试，并且两人**定期交换角色。**

3. 计划与实施项目

项目进度

项目进度：项目进度是对**特定项目的软件开发周期的刻画。**这包括对项目阶段、步骤、**活动**的分解，对各个离散活动交互关系的描述，对**各个活动的完成时间**及整个项目完成时间的初步估算。其中相关名词如下：

- **活动**：项目的一部分，占用项目进度计划中的一部分时间。
- **里程碑**：指标志着**活动结束的特定时刻**，通常伴随着某些**提交物**作为标志。
- **项目成本**：购买软件和工具来支持开发，以及工作量/工资。

软件项目活动图

> AOE网，必须会

妈的，自己学吧。

软件项目团队组织

软件项目团队有两种基本结构：

- **主程序员负责制**：**主程序员负责所有决策**，同时分配任务并监督成员。其余人向他汇报，并由主程序员进行最终决策并为此负责。副主程序员是二号人物，在必要时替代主程序员。资料员负责维护所有的项目文档，编译和链接代码，并对提交的所有模块进行初步测试。

  > 优势：**交流最小化，迅速做出决定，效率高。**

> 劣势：**主观性强，对主程序员要求高。**

- 忘我制：**去中心化**，全员参与决策、共担责任。每个成员平等的承担责任，**批评和表扬只针对产品，不针对个人。**

队伍**结构性的特点**：队伍的结构性越强，越能按时完成复杂任务，但也更加循规蹈矩，能给出普通但功能完备的成品，适合**规模大的团队解决稳定的任务**；队伍的结构性越弱，创造性越强，越能对问题给出创造性的解决方案，但是不稳定，容易无法按时限完成任务，适合解决**含有大量不确定因素的问题。**

COCOMO模型针对项目开发的**不同阶段来设置工作量的衡量标准**，逐步细化，逐渐准确（就是为了方便给程序员算工资）：

- **计划阶段**：项目通过**构建原型**来解决用户界面、软件、交互、性能、技术成熟度等方面的高风险问题；在这一步，使用**应用点AP**来进行规模测量，比如估算屏幕数、报表数、组件数等。
- **早期设计阶段**：已经决定将项目开发向前推进，但是设计人员必须研究几种可选的体系结构和操作的概念；在这一步，使用需求文档中的**功能点FP**来进行规模测量。
- **后体系结构阶段**：开发已经开始，而且已经知道了更多的信息。在这个阶段，可以根据**功能点或代码行**来进行规模估算，而且可以较为轻松地估算很多成本因素（开发已经开始，软件已经被部分构造出来）。

软件风险

软件风险：软件生产过程中不希望看到的，**有负面结果的事件。**软件的风险有两方面：**风险影响（即风险发生会造成的损失）和风险概率（即风险发生的概率）。**

风险管理活动包括两方面：

- 风险评估：包括风险识别，风险分析，风险优先级分配。
- 风险控制：包括**风险降低，风险管理计划，风险化解。**

降低风险的策略有三种：

- 避开风险：改变功能和性能需求，避开可能的风险。

> 例如：不用 C 改用 Java 来避免内存泄漏。

- 转移风险：设法将风险转移到其他系统中，或者买个保险来补偿风险发生时造成的损失（毕竟风险不是100%发生）。
- 假设风险：总假设风险会发生，接受它，并使用资源来控制风险的后果。

4. 软件需求

需求是对**来自用户的关于对软件系统的期望行为**（做什么）的综合描述，涉及**对象、状态、约束、功能**等。

需求阶段作为一个工程，**确定需求的过程**如下：

- **原始需求获取**：从用户处确定系统应该干什么。
- **问题分析**：通过分析，更好地理解需求，并通过模型或原型进行描述。
- **规格说明草稿**：利用符号描述系统，是定义规范化的描述（利用符号描述系统，将定义规范化表示）。
- **需求核准**：开发人员和用户共同对需求进行核准，有时使用到原型进行测试。
- **最终输出**：输出正式的软件需求规格说明书SRS。

获取需求时，如果有冲突，应该按照以下顺序划分**优先级**：

1. **绝对要满足**的需求。

   > 例如：信用卡转账记录必须被持久化保存，并且可以列出，信用卡账单不保存是不可能的。

2. **非常值得要但并非必要**的需求。

   > 例如：信用卡转账时要能实时发送给用户，这能极大的方便用户，但是非要说的话没有这个功能信用卡也能用。

3. **可要可不要**的需求。

   > 例如：账单根据金额不同用不同的颜色显示。

需求文档分为两种：

- 需求定义：完整罗列**用户客户期望中系统要做的事。**
- 需求规格说明（SRS）：将需求定义**用技术术语和符号进行重述**，描述为系统将如何运转的说明，设计者可以以此展开设计。

比较蛋疼的一部分，首先需求分为功能性需求和非功能性（质量）需求：

- **功能性需求**：描述系统**内部功能或系统与外部环境的交互功能**，涉及系统的输入应对、实体状态变化、结果输出、设计约束、过程约束等。
  功能性需求的特征是：它针对的是**解决问题的方案和可选方案的边界。**

- **非功能性（质量）需求**：描述软件方案必须具有的**质量特征**，比如响应时间、安全性、易用性等。
  非功能性需求的特征是：它们不是用户想让系统完成的事，而是用户希望完成的有多好。

其次是需求会因为一些原因受到约束，也就是一些限制，可以分为两种约束：

- **设计约束**：**已经做出的设计决策**或**限制问题解决方案集合的设计决策。**设计约束会让我们无法使用一些设计方案，设计约束是**技术层面**的。

  > 设计约束包括：物理环境限制（如设备能力上限）、接口限制（如预定的输入输出格式）、用户限制（甲方有限制情况）。

- **过程约束**：对于**构建系统的技术和资源的限制**，这个“技术”也包括软件开发模式等。

  > 过程约束包括：资源限制（如人员、材料、资金）、文档限制（如文档量有要求）等，**过程约束是外部要求。**

5. 软件系统设计

软件设计

四个比较蛋疼的概念，多品品（应该从下往上看比较好，先了解设计的定义）：

- **软件体系结构**：一种软件解决方案，用于解释**如何将软件系统分解为单元，单元之间如何相互关联，单元的所有外部特性。**
- **设计模式**：一种针对**单个或少量软件模块给出的一般性软件解决方案**，这种设计决策层次低于软件体系结构。
- **设计公约**：一系列**设计决策和建议的集合**，用于提高系统某一方面的设计。当一种设计公约发展成熟时，将会被封装成设计模式或体系结构风格，最后可能被内嵌封入程序语言结构。

  > 对象就是一个例子。对象、模板等都是**编程语言支持的设计和编程公约。**

- **设计**：将**需求中的问题转换为软件解决方案的创造性过程。**概念设计和技术设计是设计的两面：
  - **概念设计**：面向**用户**的，告诉用户**系统会做什么**，即**软件的架构与功能**。
  - **技术设计**：面向**程序员**的，告诉程序员**系统将会怎么做来完成任务**，即**程序员参考文档。**

软件设计过程模型的五个阶段：

1. **初始建模**：尝试对系统进行分解，根据需求描述的系统关键特性确定软件体系结构风格。
2. **分析**：分析软件系统的功能和质量属性、各种约束等。

3. **文档化**：确定各个不同的模型视图，进行文档化。
4. **复审**：检查文档是否满足所有功能及质量需求。
5. **最终输出**：软件体系结构图SAD。

用户界面设计

涉及用户界面时需要考虑这些问题：

- 设计界面时的**要素**（界面寓意、思维模型、导航、外观、给人的感觉）。
- **文化差异**问题。
- **用户爱好**问题。

其中，要素包括：

- 隐喻：可识别和学习的基本术语、图像和概念等。
- 思维模型：数据、功能、任务的组织与表示。
- 模型的导航规则：怎样在数据、功能、活动和角色中移动及切换。
- 外观：系统向用户传输信息的外观特征。
- 感觉：向用户提供有吸引力的体验的交互技术。

文化差异包括用户的信仰、价值观、道德规范和传统等因素，因此需要：

1. 使用**国际设计/无偏见设计**，排除特定的文化参考或偏见。
2. 采用**定制界面**，使不同用户看到不同的界面。

用户爱好问题可以为具有不同偏好的人选择备选界面。

耦合与内聚

模块独立性：**模块之间彼此独立的程度。**模块独立性取决于内聚和耦合程度，我们追求的是**高内聚、低耦合。**

耦合：两个软件部件之间的相互关联程度。根据耦合程度可以划分为**紧密耦合**、**松散耦合**、**低耦合**三种，具体来讲有六种：

- **非直接耦合**：模块之间没有信息交换。
- **数据耦合**：模块之间**传递数据，但不限定数据结构。**

> 例如：传递水费、电费。

- **特征耦合**：模块之间传递的是**数据结构（这代表两者传递信息是使用了相同的数据结构）**。

> 例如：传递水电账单。

- **控制耦合**：模块间**传递的是控制量**，控制量由一个模块传出，作为另一个模块完成功能的必要条件，控制另一个模块的活动。

> 例如：一个模块传出一个 `flag` 给别的模块，别的模块根据 `flag` 执行不同操作。

- **公共耦合**：不同模块**访问公共的数据**。

> 例如：大家一起访问全局变量。

- **内容耦合**：一个模块**直接修改另一个模块**（直接调用另一个模块的私有数据，或者一个模块在另一个模块中）。

> 例如：A里直接有一个B对象。

内聚：模块内部各组成成分的关联程度。可以按层次划分为高内聚、低内聚。具体来讲有七种：

- **偶然性内聚**：不相关的功能、过程、数据出现在同一个部件中。

> 例如：数据预处理，不同的模块在不同的步骤位置恰巧都有数据预处理，所以合并为一个模块。

- **逻辑性内聚**：**逻辑上相似或相关的功能或数据**放置在同一个模块内。

> 例如：计算平均分和最高分都可以归纳为读数据、算、输出，那么读数据、输出就可以放进同一个模块里。虽然读和输出不能说有必然联系，但在系统中也可以理解为有逻辑相关性。逻辑性内聚是几个功能整体上在几个流程的相似位置出现，从而有了逻辑的相关性，而不是偶然性内聚中表现的，完全不同的流程中的随机位置恰好有相似的部分。

- **时间性内聚**：模块中各部分要求在**同一时间内完成。**

  > 例如：一个初始化模块中有为变量赋初始值， 打开文件等功能，这些功能不一定说谁是谁的前提，但是他们都必须在“初始化”这一段时间内完成。

- **过程性内聚**：模块中的**各部分有先后顺序。**

  > 例如：数据分析模块中有输入数据、检查数据、分析数据的过程，它们必须按顺序执行才行，前面不执行后面就不能执行。

- **通讯性内聚**：模块中**各个部分共享数据。**

  > 例如：一个传感器读取模块，里面的数据来自于各个传感器，彼此不一定相关，但是对模块内的各部分共享。

- **顺序性内聚**：模块中的各个部分的**输入与输出相连。**
- **功能性内聚**：模块中的各个部分只构成了一个单一功能。

软件复审

设计复审：检查我们的软件设计、软件体系结构图**是否满足了所有功能与质量需求。**其**重要性**在于复审中批评和讨论是“忘我”的，能将开发人员更好地团结在一起，提倡并增强了成员之间的交流在评审过程中故障的改正还比较容易，成本还不高，在这时候发现故障和问题会使每一个人受益。复审可以分为两类：

- **概念设计复审**：与客户和用户一起检查概念设计。
- **技术/程序设计复审**：让程序员可以参与复审，在程序实现之前获得本阶段的反馈。

6. 面向对象

面向对象的设计原则：

- 总原则-开闭原则：对扩展开放，对修改关闭。
- 单一职责原则：一个类应该有且仅有一个引起它变化的原因，一个职责，否则类应当被拆分。
- 重用原则：尽可能不重复已写代码，尽最大可能复用。
- 里氏替换原则：继承必须确保超类所拥有的性质在子类中仍然成立。
- 依赖倒置原则：面向接口编程，依赖于抽象而不依赖于具体。
- 接口隔离原则：每个接口中不存在子类用不到却必须实现的方法，否则就应该拆分接口。

- 迪米特法则：一个类对自己依赖的类知道的越少越好（一个对象应该对其他对象保持最少的了解）。
- 合成复用原则：尽量首先使用合成聚合的方式，而不是使用继承。

OO开发过程的五个步骤：

1. 面向对象需求分析。
2. 面向对象高层设计。
3. 面向对象底层设计。
4. 面向对象编程。
5. 面向对象测试。

UML

各种UML图：

- 用例图：描述系统必须执行的一般过程
- UML类图：描述对象之间的静态关系
- UML活动图（程序框图）：描述业务活动的工作流模型，显示对象的值更改时系统中可能发生的所有活动
- UML状态图：展现一个对象所具有的所有可能的状态，并且它们在接收到什么信息时会进行怎样的转化
- UML包图：类进行打包，使设计更加层次化，易于理解，包图就是这个层次的图
- UML顺序图（时序图）：展示活动或行为发生的顺序
- UML通信图/ML 协作图：使用对象与对象之间的连接来描述对象间的消息顺序（与时序图相似，但顺序图强调时间顺序，通信图强调空间顺序）
- UML构件图：说明运行时的构件以及它们之间的交互
- UML部署图：描述如何为构件分配计算资源

用例图

类图

7. 编写程序

一般性的**编程原则**应该从以下三个方面考虑：

- **控制结构**：要让程序设计反映出在体系结构和设计中的各种控制结构。
- **算法**：程序设计通常会制定一类算法，用于编写组件。

- **数据结构**：编写程序时，应该安排数据的格式并进行存储，让数据结构决定项目结构，并尽可能以此简化程序。

程序注释

首先是HCB，即**头部注释版块**，是总览性的信息，用于**标识程序、描述数据结构、算法、控制结构。**具体信息包括：

- 名称
- 编写者
- 在系统中的位置
- 编写时间
- 如何使用的数据结构、算法、控制

除此以外，还需要添加：

- 版本注释：随着时间进行修改的记录。
- 解释性注释：解释源代码在干啥。
- 分解性注释：通过注释将代码分解成多个段落。

与此同时，编写内部文档时还要注意：

- 分段注释。
- 注释和代码要一并更改。
- 注释要有意义。
- 一边写代码一边写注释，不要写完代码回过头来添加注释。

8. 测试程序

软件缺陷

软件产生缺陷的原因：

- 系统本身有太复杂的部分
- 客户不清晰的需求
- 设计阶段就存在缺陷
- 其他因素（规模、参与者过多）

- 不是客户真正想要的软件，**错误的需求。**
- 缺失若干事件处理逻辑。
- 软件需求无法实现。
- 曲解需求、设计本身问题等。
- 代码在某些条件下有运行隐患。
- 将设计予以实现时方法不当。

为何要**对软件缺陷进行分类**：系统中不存在明显的故障时，我们就需要对程序进行测试，创造一些条件以期让代码不能像计划那样做出反应，看看有没有更多故障。而为了更容易发现这些故障，为故障进行分类时很重要的。

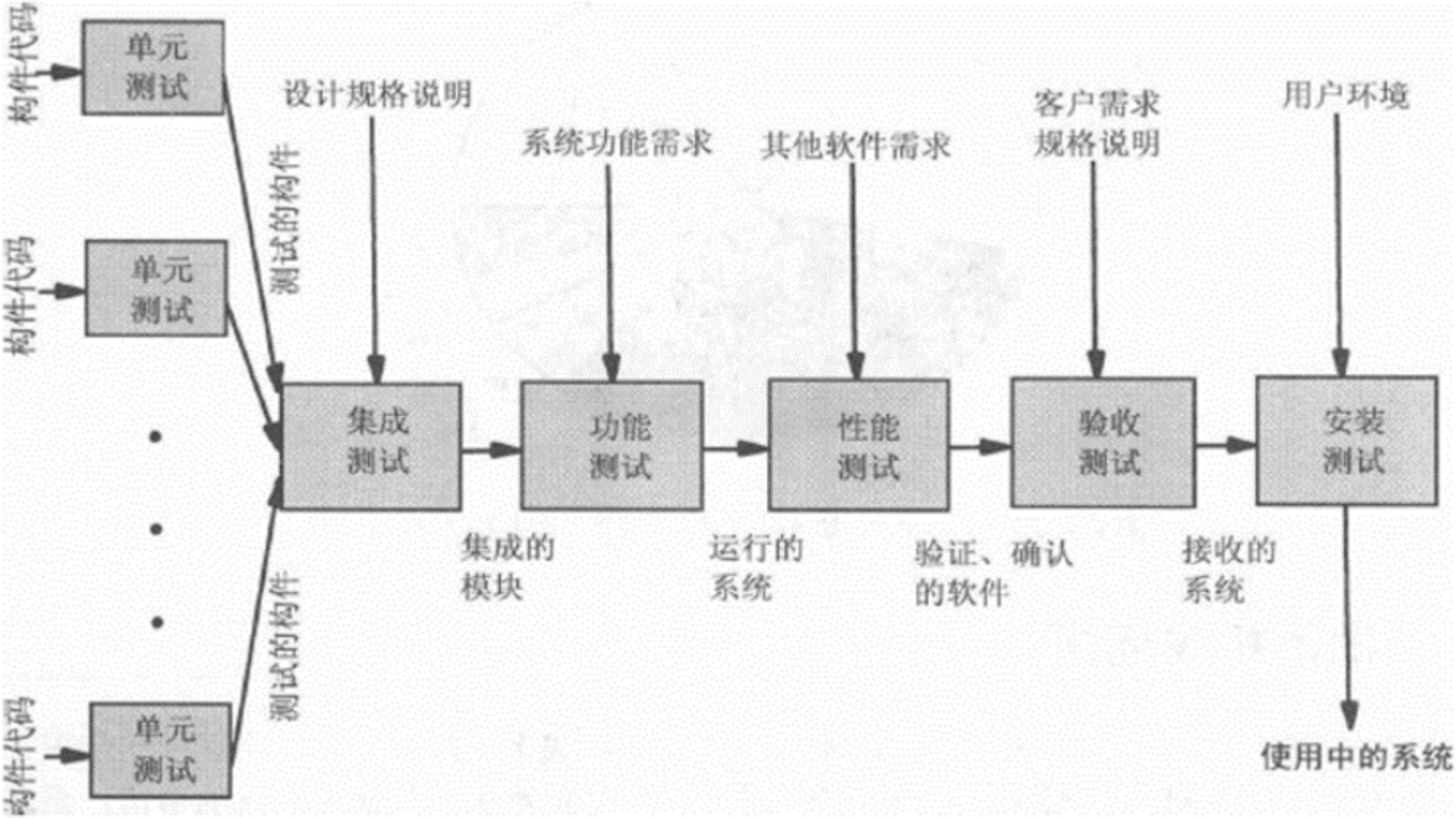
主要的缺陷类型（10种）：

- 算法缺陷：算法某些处理步骤或逻辑有问题，导致软件的部件**对于输入数据不能给出正确的输出**；
- 计算和精度缺陷：算法或公式在编程实现时逻辑没错，但是**计算过程出现了错误或者精度达不到要求**，从而无法获取正确输出；
- 过载缺陷（压力缺陷）：程序运行时**，数据填充量会超过数据结构的规定容量**引起的缺陷；
- 能力缺陷（边界缺陷）：**系统活动量达到系统极限时，系统性能变的不可接受**，称为能力缺陷；
- 性能缺陷（吞吐量缺陷）：系统在**常规状态下就不能以需求规定的速度执行**；
- 时序性缺陷（协调缺陷）：几个**同时或有严格执行顺序的进程协调**出现问题；
- 文档缺陷：**文档描述与程序实际行为**不符；
- 恢复性缺陷：系统失效时，**程序无法再恢复**也是一种缺陷；
- 硬件和系统软件缺陷：作为**底层支持的硬件和系统软件没有按照文档中的操作条件和步骤运作**时，也可引起软件的问题；
- 标准和规格缺陷：代码**没有遵守组织机构的标准和过程。**这个缺陷最大的影响在于：不按照标准的代码可能在测试和修改时让人不好理解，引起问题。

正交缺陷分类：使任意**一个缺陷只属于一个类别**的缺陷分类方案称为正交缺陷分类。

> 如果故障属于不止一个类，则失去了度量的意义。

测试过程



测试的各个阶段如上图所示，具体包括：

1. 单元测试：验证组件的功能。依据文档：程序代码与配套文档。
2. 集成测试：验证系统组件是否能正确的协同工作。依据文档：系统体系结构文档SAD、程序设计规格说明。
3. 功能测试：验证系统是否能执行需求规格说明中描述的功能。依据文档：软件需求规格说明书SRS。
4. 性能测试：验证系统的软硬件表现和性能是否符合需求规格说明文档。这一步之后，软件系统应当能在客户的实际工作环境中成功执行，这时我们说**产生了一个被确认的系统。**依据文档：软件需求规格说明书SRS。
5. 验收测试：验证系统是否满足了客户的需求定义（**需求定义和需求规格说明是有区别的**）。依据文档：客户需求定义。
6. 安装测试：验证系统能否在用户使用的真实环境中安装并正常运行。依据文档：用户环境的说明。

> 系统测试：**功能测试、性能测试、验收测试和安装测试统称为系统测试。**

黑盒测试与白盒测试

黑盒测试：人员在**完全不了解程序内部的逻辑结构和内部特性**的情况下，只依据程序的需求规格及设计说明，检查程序的功能是否符合它的功能说明。其**原则是依据系统需求文档、系统设计文档、程序设计文档进行测试**，正确的结果是系统完成了所有该做的，拒绝了一切不该做的。

- 优点：测试人员不受程序所带来的束缚，测试更具有客观性。
- 缺点：有时会无法进行完备的测试，在不知道程序内部逻辑的情况下，设计测试时就可能无法面面俱到。

白盒测试：人员拥有全套文档，**以程序内部结构为基本依据**，手动或自动进行测试。

- 优点：有助于进行更细致、切中要害的测试。
- 缺点：全路径、极度细致的测试不现实。

黑盒测试的分类方法包括如下四种：

- **等价分类法**：输入域划分为若干等价类，并且从每个等价类里选择有代表性的少量用例代表其余所有情况。其根本逻辑在于：**如果这些代表性用例没有出现问题，那么其他的一般也没有问题。**
- **边界值分析法**：在等价分类法的基础上，把测试值选在等价类的边界上，经验告诉我们这往往有更好的效果。
- **错误猜测法**：根据测试人员的经验，猜测程序中哪些地方容易出错，并补充出用例；此方法适合作为其他方法的补充。
- **因果图法**：适用于被测试程序有很多输入条件，程序的输出又依赖输入条件的各种组合的情况。

等价分类法中，等价类可以分为**有效等价类**和**无效等价类**两种，前者是正常输入，后者是不应当输出正确结果的无效输入。在划分等价类时，需要尽可能“密铺”式划分，并且尽可能不对测试的实体整体添加限制条件以提升可扩展性。

对有效等价类的用例，**尽量用一个用例覆盖尽可能多的等价类**，这是因为任意一个有效等价类的处理出现问题都会产生故障，被我们发现，我们可以以此减少测试次数；对无效等价类，必须**为每一个无效等价类都设计一个专内验证该点的用例**，这是因为所有无效等价类的表现都是不正常返回，不进行共用以避免多个错误一起发生导致漏过一些错误。用例可以进一步区分为如下三种：

- **弱一般等价类**：测试每个一般等价类都至少出现在用例中一次即可。

  > 基于单缺陷假设：**失效很少因为两个或更多缺陷同时共同引发。**

- **弱健壮等价类**：考虑有效等价类之外，划分出的无效等价类，并且每个等价类在用例中至少出现一次即可；仍基于单缺陷假设。

- **强一般等价类**：测试用例应当遍历等价类的所有笛卡尔积组合；基于多缺陷假设。

白盒测试的方法包括两种：

- 逻辑覆盖法。
- 路经测试法。

逻辑覆盖法是一组逻辑覆盖方法的统称，按照程序逻辑覆盖程度分为：

- **语句覆盖**：每条语句至少执行一次。
- **判定覆盖（分支覆盖）**：每一分支至少执行一次，又称分支覆盖。
- **条件覆盖**：每个条件均按“真”和“假”两种结果至少执行一次。
- **条件组合覆盖**：某个分支的虽然只有一种结果，但可能由多个条件组合而成。条件覆盖只要求单个条件一次真一次假即可，条件组合覆盖要求覆盖所有组合，即使有些组合最终结果一样。

> 满足条件覆盖不一定满足分支覆盖，例如 `a && b`，当用例包括 `a=false, b=true` 和 `a=true, b=false` 时就满足条件覆盖，但显然不满足分支覆盖。

路经测试法借助**程序图**设计测试用例，包括四种：

- 结点覆盖：经过所有结点，相当于逻辑覆盖中的语句覆盖。
- 边覆盖：覆盖所有边，相当于逻辑覆盖中的判定覆盖。
- 完全覆盖：同时满足结点覆盖和边覆盖，也即走过所有位置，这是**测试简单程序的最低标准。**
- 路径覆盖：程序图中每条路径都至少经过一次。

> 路经覆盖法与穷举测试有所不同，路径覆盖法并不关注循环次数，某个循环语句循环1次和n次对于路径覆盖法是一样的，但是穷举测试则认为它们是不同的。

单元测试

单元测试：将每个**程序构件与系统中其他构件隔离**，对其单独进行测试。

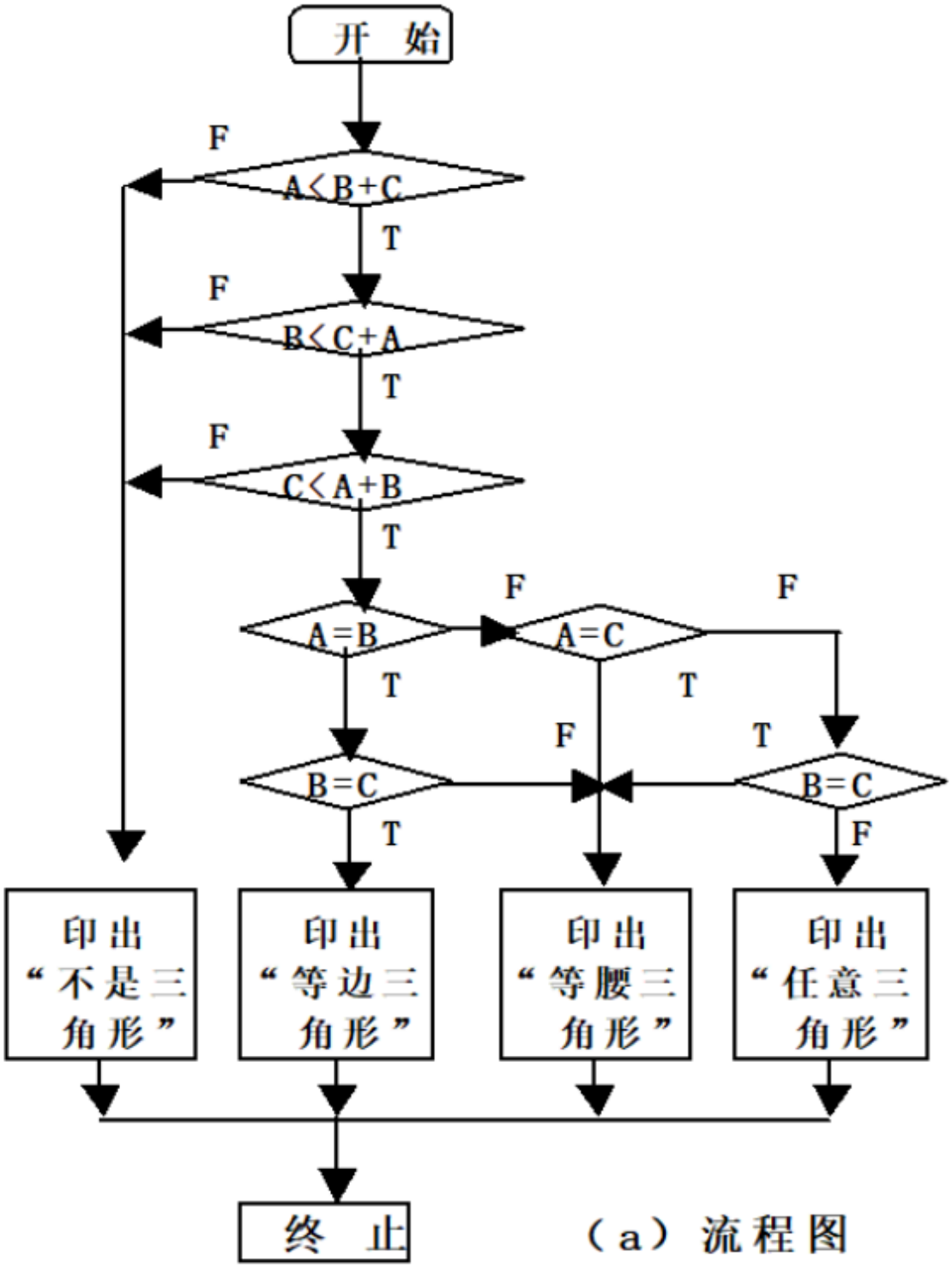
在**集成测试及以后的阶段**，除去很小的程序，都应当使用黑盒原则进行测试。这一具体流程包括：

- 使用**边值分析法**或**等价分类法**提出基本的测试用例。
- 使用**猜错法**补充一些测试用例。
- 如果在程序的功能说明中含有输入条件的组合，宜在一开始就用因果图法，然后再按以上两步进行。

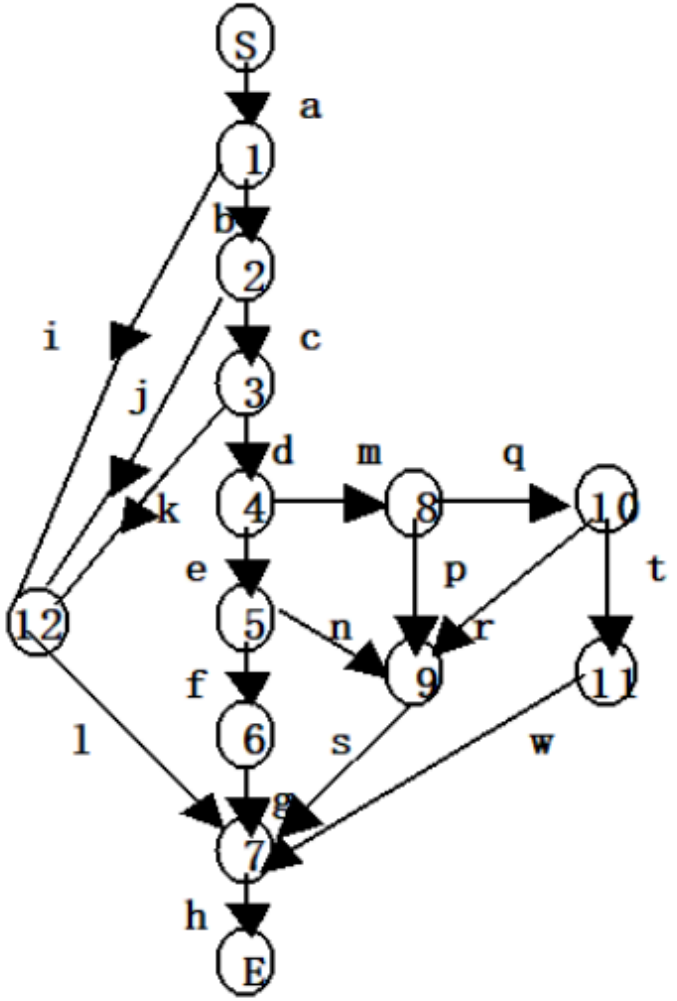
对于**单元测试**，我们往往可以直接参考模块的源代码，并且工作量可以承受，所以宜采用**黑盒法白盒法结合运用**——先使用黑盒法设计测试用例，然后使用白盒法进行补充，达到我们期望的覆盖标准。

下面举一个测试的例子，基本流程如下：

1. 使用等价分类法划分等价类，包括有效等价类和无效等价类。
2. 使用边值法和猜错法补充等价类。
3. 选择测试数据，构建测试用例。


开 始
A<B+C
B<C+A
C<A+B
A=B
A=C
B=C
B=C
F
T
印出“不是三角形”
印出“等边三角形”
印出“等腰三角形”
印出“任意三角形”
终 止

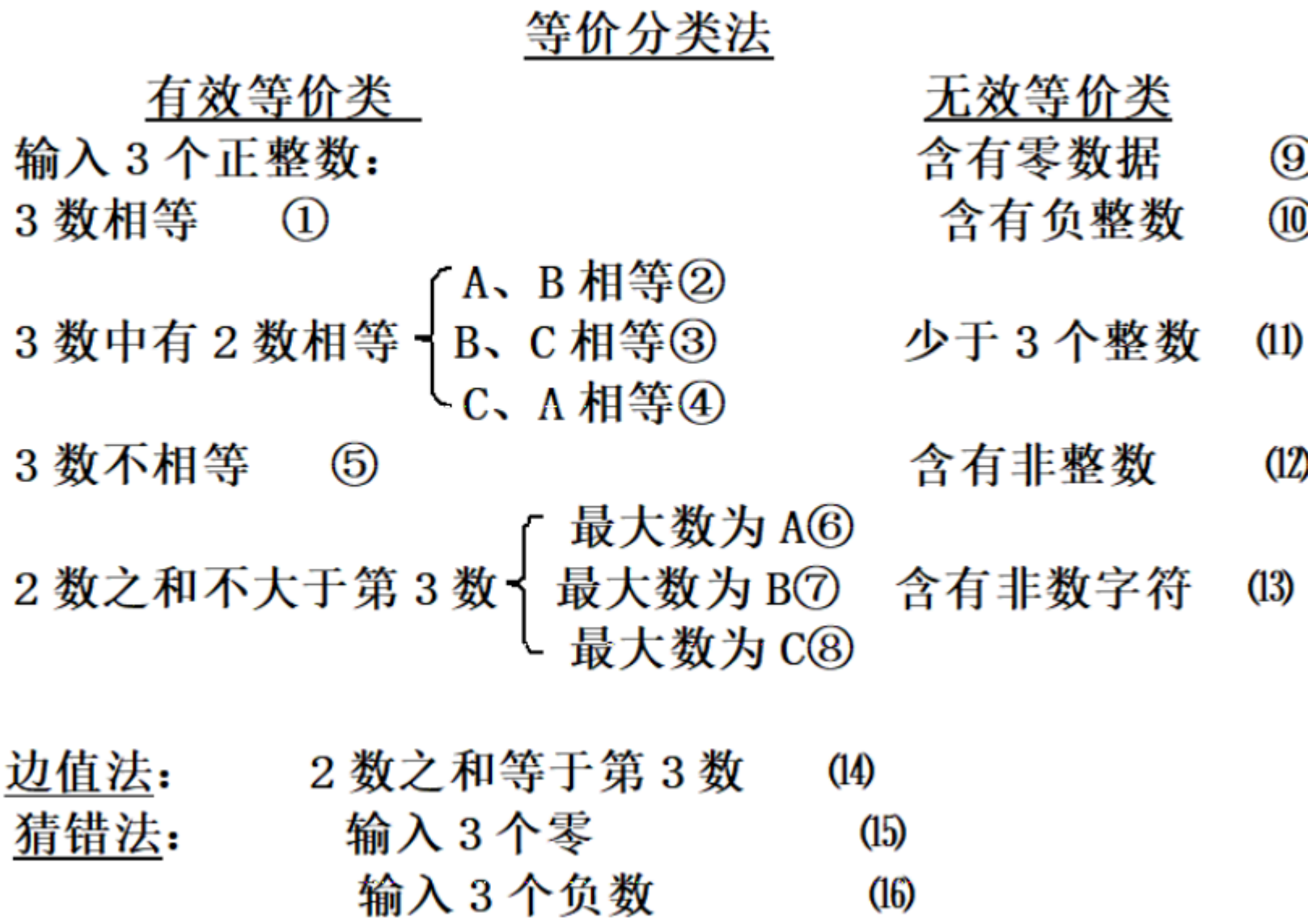
（a）流程图


S
a
1
b
2
c
3
d
m
q
i
j
k
4
8
10
e
p
t
12
5
n
r
9
11
f
l
s
w
6
g
7
h
E

（b）程序图

> 本例先用黑盒法设计测试用例，然后用白盒法进行检验与补充。

首先使用等价分类法和边值法、猜错法进行分析：



根据等价类设计测试用例：

序号	测试内容	测试数据			期望结果
		a	B	C	
1	等边	5，5，5			等边三角形
2	等腰	4，4，5	5，4，4	4，5，4	等腰三角形
3	任意	3，4，5			任意三角形
4	非三角形	9，4，4	4，9，4	4，4，9	
5	退化三角形	8，4，4	4，8，4	4，4，8	
6	零数据	0，4，5	4，0，5	4，5，0	不是三角形
7		0，0，0			
8	负数据	-3，4，5	3，-4，5	3，4，-5	
9		-3，-4，-5			
10	遗漏数据	3，4，—			
11	非整数	3.3，4，5			运行出错
12	非数字符	A，4，5			（类型不符）

用白盒法检验测试用例，可以发现前几个就能完成边覆盖，因此不需要补充更多测试用例：

序号	测试数据	覆盖结点	覆盖的边
1	5，5，5	1，2，3，4，5，6，7	abcdefgh
2a	4，4，5	1，2，3，4，5，9，7	abcdensh
2b	5，4，4	1，2，3，4，8，10，9，7	abcdmqrsh
2c	4，5，4	1，2，3，4，8，9，7	abcdmpsh
3	3，4，5	1，2，3，4，8，10，11，7	abcdmqtwh
4a	9，4，4	1，12，7	ailh
4b	4，9，4	1，2，12，7	abjlh
4c	4，4，9	1，2，3，12，7	abcklh

如上，完成一个测试的设计。

集成测试

集成测试验证系统组件**是否能正确的协同工作**，选择策略要兼顾系统特性和客户需求。

- 驱动模块：**代替上级模块传递测试用例**的程序（出现在**自底而上**集成测试中）。
- 桩模块：**代替下级模块的仿真程序**（出现在**自顶向下**）。

集成测试的方法有四种：

- **自底向上的集成测试**：从模块结构图的最底层开始，**由下而上按调用关系逐步添加新模块**，组成子系统分别测试，直到全部组装完毕。典型特征是：添加的新模块调用的下层模块都必须被全部测试完毕、 **使用驱动模块。**

  其优点在于：

> - 容易生成测试用例（因为底层都是真实模块）。
> - 适合面向对象方法（每次加入的是经过测试的对象，也符合消息的传递方式）。
> - 当许多低级组件经常在各个地方被调用时，这种方法十分适用
>
> 缺点是许多存在于高层模块中的关键错误无法被及时发现。

- **自顶向下的集成测试**：从顶层控制组件开始进行测试，然后**逐步将调用的下级组件组合起来**，再对更大的子系统测试，直到全部组装完毕；典型特征是：添加的新模块，调用它的上层模块必须被测试过、 使用桩模块。

> 其优点在于上层的问题常常是影响更大的，自顶向下更利于发现这些关键问题。
>
> 缺点：
>
> - 生成测试用例更难。
> - 可能需要很多桩。

- **莽撞/一次性测试**：先测试每一个模块，之后将所有模块一并集成。
- **混合方式/三明治方式测试**：从上到下将模块分为三层：上层、目标层、下层。上层自顶向下，下层自底向上，中层直接使用**驱动模块+桩模块**独立测试，最后集成三层，测试集中于目标层。

OO测试

传统测试与OO测试的区别：

1. 传统测试：当系统改变时，需要**新老测试用例（回归测试）**，OO只需要推出新的测试用例即可。
2. OO测试：**必须对重载的子类进行测试**，可能会使用不同的测试用例。
3. OO测试，在单元测试中更加简单（对象的粒度更小），但是**集成测试更难**（设计接口、继承、多态等）。

OO测试面临如下四个困难：

1. 需求验证**缺乏工具支持**（很多时候依赖人工）。
2. 测试工具生成的**测试用例**，处理OO模型中的对象和方法时，其针对性不强（某些OO关系是测试工具本身搞不清楚其内在逻辑关系的）。
3. 传统的测试方法（如环路复杂度等）在**评价**OO系统的**规模和复杂性**时，还不是很有效。
4. **对象的交互**是OO系统复杂性的根源，传统的测试方法和根据作用有限。

9. 测试项目

系统测试

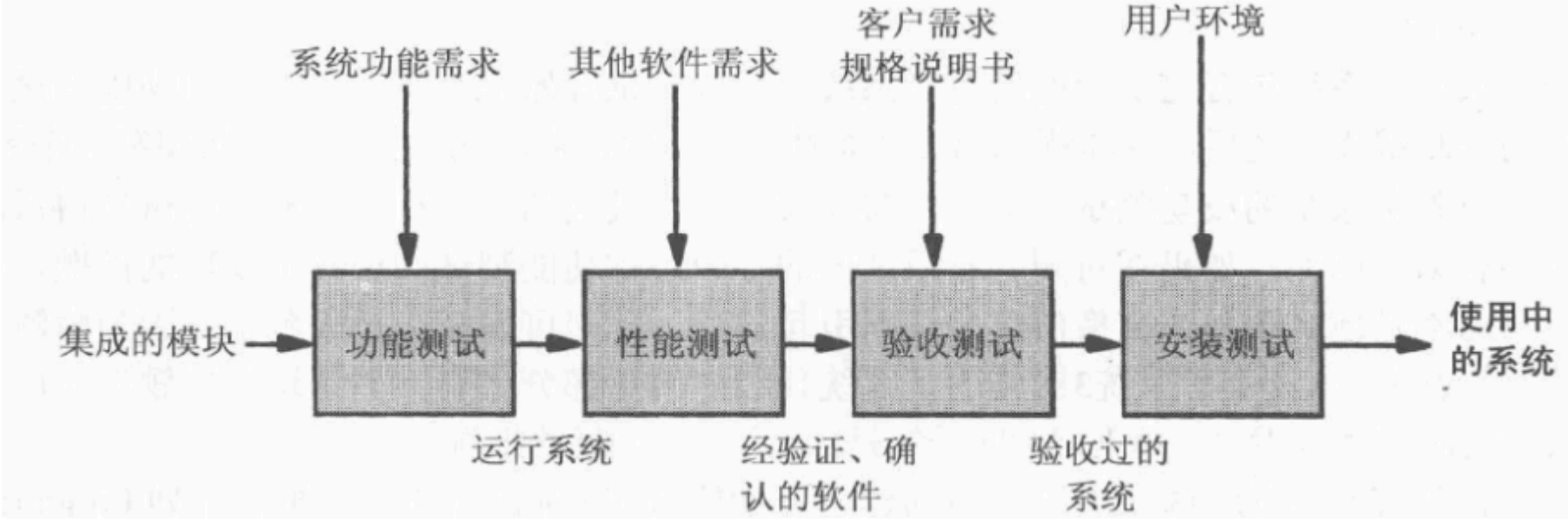


图9-2　测试过程的步骤

系统测试的过程如上图所示，具体步骤包括：

- **功能测试**：验证系统是否能执行需求规格说明SRS中描述的功能；依据文档：软件需求规格说明书SRS。
- **性能测试（非功能性需求）**：验证系统的软硬件表现和性能是否符合需求规格说明文档这一步之后，软件系统应当能在客户的实际工作环境中成功执行，这时我们说**产生了一个被确认的系统**；依据文档：软件需求规格说明书SRS。
- **验收测试**：验证系统是否满足了客户的需求定义（需求定义和需求规格说明是有区别的）；依据文档：客户需求定义。
- **安装测试**：主要解决开发环境和用户环境的不同导致的问题，验证系统能否在真实环境中安装并正常运行；依据文档：用户环境的说明。

回归测试是**用于新版本的一种测试**，验证它与旧版本相比，是否仍以相同的方式执行着相同的功能。

功能测试

功能测试的**作用**：以**高检测率发现缺陷**（因为一项功能测试只面向一小组组件，不容易导致多个缺陷彼此掩盖）。

功能测试的**基本指导原则**：

- 要具有**较高的查错率。**
- 使用**独立的测试团队。**
- 了解**预期的输出结果。**
- 对**合法和非法输入**都进行测试。
- 不能为了测试方便而去修改系统。
- 制定**测试停止的标准。**

性能测试

性能测试的**作用**：确保系统的**可靠性、可用性和可维护性。**

性能测试的主要分类包括如下13种：

1. **压力/强度测试**：短时间内加载极限负荷，验证系统能力，对经常产生负荷高峰的系统很有意义。
2. **容量/巨额数据测试**：验证系统处理巨量数据的能力。
3. **计时测试**：评估涉及对用户的响应时间以及功能执行耗时的相关需求。
4. **配置测试**：对系统**软硬件的各种配置**进行测试。
5. **兼容性测试**：测试其接口在与其他系统互动时能否正常运作。
6. **环境测试**：测试系统在安装场所的执行能力，这里指的是**外部的物理条件**，比如高温、潮湿。
7. **回归测试**：验证软件的新版本与旧版本相比，是否仍能以相同的方式执行着相同的功能。
8. **安全性测试**：确保安全性需求得到满足。
9. **质量测试**：评估系统的**可靠性、可维护性和可用性。**
10. **恢复测试**：检验系统**是否能在故障或丢失电源、数据、设备时自我恢复。**
11. **维护测试**：核验一些诊断工具和过程是否能正常运行，如：诊断程序、事务跟踪、辅助工具。
12. **文档测试**：确保编写了必要的文档。
13. **人为因素/可用性测试**：检查设计系统用户界面的需求。

测试需要考虑的三个性能：

- **可靠性**：软件系统在**给定的时间范围和条件**下运行成功的概率。

- **可用性**：软件系统在**给定的时间点成功运行的概率。**

> 可用性强调**某一时刻**系统正常，系统可能在相当长一段时间内都可用，保持了可靠性，但在不能使用（例如检修）的那一刻，它失去了可用性。

- **可维护性**：是指在给定的使用条件（包括：预定的时间间隔、可用的维护程序、可用的维护资源之下进行维护）下，**维护活动能被执行的概率。**

验收测试

验收测试由**客户检查软件系统是否满足了他们的需求定义**，主导者是客户，开发者只负责解答一些必要的问题。

有三种分类：

- **基准测试**：先由用户准备测试用例，然后在实验环境中安装系统，最后由用户进行评估。
- **引导测试**：先将系统安装在实验环境中，然后在假设系统正式安装的前提下，由测试者在测试系统上进行日常工作，而不是依赖于测试用例；α测试和β测试都属于引导测试。

> α测试：由开发者自己组织人员或委托专业团队来进行小规模测试；β测试：由客户实际进行小规模测试

- **并行测试**：当软件的一个旧版本正在使用，并且要测试一个新版本时使用；新旧版本并行运转，来自用户的操作会同时在新旧系统上执行，旧系统实际工作，新系统进行测试，使用户逐渐习惯新系统。

安装测试

安装测试：验证系统能否**在用户使用的真实环境中安装并正常运行**，以发现并解决因开发环境和用户环境不同所引起的问题。