|  |  |
| --- | --- |
| 阶段 | 说明 |
| ****可行性分析/项目论证**** | 判断项目是否值得开发，从**技术、经济、时间、人力、社会**等方面评估可行性，形成《可行性研究报告》。 |
| ****需求分析**** | 明确系统要实现的功能、性能及约束条件，形成《软件需求规格说明书（SRS）》 |
| ****软件体系结构设计**** | 确定系统的整体结构和模块划分，定义各模块之间的接口与交互方式，为后续详细设计提供蓝图。 |
| ****用户界面设计**** | 设计用户与系统交互的界面布局、操作流程和视觉风格，确保系统的**可用性和易操作性**。 |
| ****详细设计**** | 对每个模块进行细化，确定数据结构、算法逻辑、数据库设计、接口定义等技术细节。 |
| ****软件实现**** | 按照设计文档编写源代码、调试程序，将设计转化为可运行的软件系统。 |
| ****软件测试**** | 通过**单元测试、集成测试、系统测试和验收测试**等阶段验证软件是否满足需求。 |
| ****软件交付与部署**** | 将软件安装到目标运行环境中，完成系统部署、数据迁移及用户培训等工作。 |
| ****软件维护**** | 软件投入运行后的长期阶段，包括**错误修复、功能改进、性能优化和环境适配**等。 |

**颜色都属于软件设计的一部分**

**软件生命周期，支撑软件，正确性，有效性，安全性，可信性，可靠性，系统规模，代码增长导致什么，软件危机，软件开发，软件过程，软件工程三要素，软件原型开发，模块化，信息隐藏，双向追踪（正向，反向）**

**瀑布模型，原型模型（演化模型），增量模型，迭代模型，螺旋模型（演化模型），基于构建模型，UP模型（统一过程），RUP，喷泉模型**

# 项目管理图

|  |  |  |  |
| --- | --- | --- | --- |
| 图名 | 又称 | 主要用途 | 特点 |
| PERT 图 | 网络图 | 用于管理大型、复杂项目 | 表示任务之间的依赖关系和关键路径 |
| Gantt 图 | 横道图、条状图 | 表示项目进度、评估工作进展 | 横轴：时间； 纵轴：任务； 线条：计划与实际进度； 箭头：依赖关系，必须完成前面的才能进行后面的； 颜色：表示不同阶段 |

**工程网络图包含：PERT图，CPM, AON, AOA**

# **软件/程序设计法**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法 | 阶段 | 核心思想 | 主要模型 | 说明 |
| 结构化设计SD | 概要设计 / 详细设计 | 模块化设计，控制流和数据流优化 | 结构图、**HIPO图**（层次化） |  |
| 结构化分析法SA | 概要设计 / 详细设计 | 面向结构化 | 数据流图（**DFD**）、**ER**图、状态转换图（**STD**），**数据字典**，加工说明 |  |
| Jackson 方法（JSD） | 详细设计 | 面向数据结构 | **Jackson**结构图 |  |
| OOA | 需求分析 | 面向对象，类和交互建模 | **UML** 图 |  |
| 判定表 | 详细设计 | 面向逻辑决策 | **判定表** | 处理多条件、多规则业务逻辑，即支持描述组合条件 |
| 因果图 | 详细设计 | 面向逻辑决策 | **因果图** | 描述事件/条件导致的结果关系，辅助设计复杂逻辑 |
| NS图 / 盒图 | 详细设计 | 面向逻辑决策 | **NS图 / 盒图** | 用嵌套矩形框表示程序逻辑结构；表示顺序、选择、循环控制结构； |
| PAD图 | 详细设计 | 面向程序控制流程 | **PAD图** | 程序逻辑图表示方法；描述程序的控制流程 |
| 程序流程图 | 详细设计 | 面向程序控制流程 | **程序流程图** | 用于表示程序控制流程 |
| 程序构造图（PSC） | 详细设计 | 面向模块化结构和控制流程 | **程序构造图** | 描述模块及调用关系，模块化设计辅助工具 |
| 形式化分析方法 | 详细设计（可辅助需求验证） | 面向形式化逻辑和控制 | 形式化规格说明、公式 | 可验证需求正确性 |

**只有紫色的可以作为需求分析法**

**UML图**

|  |  |  |
| --- | --- | --- |
| UML视点 | 图 | 说明 |
| 结构视图 | 包图 | 从包层面描述系统的静态结构 |
| 类图 | 从类层面描述系统的静态结构 |
| 对象图 | 从对象层面描述系统的静态结构 |
| 构件图 | 描述系统中各构件及其依赖关系，是静态的 |
| 行为/交互 | 状态图 | 描述一个对象或设备在生命周期中可能的状态以及事件触发下的状态变化和行为。是动态的 |
| 活动图 | 描述系统中活动的流程与控制逻辑。是动态的 |
| 通信图/协作图 | 描述对象间的**消息传递**与协作。是动态的 |
| 顺序图 | 描述对象间的**交互顺序**与**时间关系**。是动态的 |
| 部署 | 部署图 | 描述系统中工件在物理运行环境中的部署情况。是静态的 |
| 用例视图 | 用例图 | 从**外部用户角度**描述**系统的功能需求与参与者交互**（不仅仅是单向的）。是动态的。参与者不止有人员 |

**HIPO（Hierarchy plus Input-Process-Output）**

* 用途：需求分析阶段 / 软件设计阶段
* 核心思想：层次化方法，展示模块的输入、处理、输出关系
* 组成：
  + **H图（Hierarchy图）**：描写软件总的模块层次结构
  + **IPO图（Input-Process-Output图）**：描写每个模块的输入、输出、处理功能及模块调用关系，可采用NS图表示模块内部逻辑
  + 输入输出信息来自数据字典

**DFD图**

* 基本原则：数据守恒，子父图平衡，数据流封闭

**面向数据流方法**

* DFD（结构化分析法）、JSD（Jackson）、HIPO、PAD图/程序流程图。
* **信息流 = 变换流 + 事务流**
  + 变换流：数据在系统中流动时发生的变化处理
  + 事务流：事务处理或业务逻辑的执行

Contents

[项目管理图 3](#_Toc211697490)

[**软件/程序设计法** 3](#_Toc211697491)

[第一章：软件工程基本概念 7](#_Toc211697492)

[第二章：软件开发过程管理 12](#_Toc211697493)

[第三章：需求分析 18](#_Toc211697494)

[第四章：系统设计 23](#_Toc211697495)

[第五章：软件开发 30](#_Toc211697496)

[第六章：软件测试 34](#_Toc211697497)

[第七章：软件维护 36](#_Toc211697498)

[错题 41](#_Toc211697499)

## 第一章：软件工程基本概念

**软件**

* **软件 = 文档 + 代码 + 数据（不仅仅是程序）**
  + 软件：一种逻辑产品，不会磨损或老化，是无形的；是现代社会的关键基础设施；复杂性体现在问题的复杂，度量参数为 软件规模和软件难度
  + 文档：记录软件开发各阶段成果及变更信息，如需求文档、设计文档、测试文档、用户手册等
  + 数据：是程序加工处理对象 和 结果
* 软件生命周期：从提出开发开始 到 开发出系统，运行维护以及最终退役的全过程。不同阶段具有不同的特征
* 软件分类：
  + 应用软件：直接面向用户需求（如办公软件、游戏）。
  + 系统软件：支撑硬件和应用运行（如操作系统、编译器）。
  + 支撑软件：辅助开发的软件（如IDE、测试工具）。
* 开源软件：有源代码；Linux，Mysql，Chrome，Eclipse；低成本，质量高，交付快，功能强大
* 闭源软件：软件代码不对用户开放，购买软件时只提供可运行的软件或服务，没有提供源代码。以许可证的方式授权用户；安全性强、维护集中
* **软件质量** = 软件满足需求的程度，是产品的生命线。
  + **功能性：规定系统做什么（如登录、处理支付）**
    - **正确性：输出结果符合预期**
    - **有效性：实现功能目标**
    - **安全性：防止未授权访问与攻击**
    - **可信性：满足人们的信任期望**
  + **非功能性：定义系统应该多好地做这件事（如响应速度、安全性、易用性）**
    - 可靠性：规定时间内完成任务，不故障
    - 健壮性：异常情况下仍能运行
    - 可用性：系统可被正常使用
    - 可维护性：易于修改、扩展和修复
    - 可移植性：能在不同平台上运行
    - 可重用性：模块可重复使用
    - 可理解性：易于阅读和理解
    - 持续性：面对突发情况仍能维持服务
* 软件形态：软件是一个“系统的系统”
  + 运行生态：物理设备，社会系统，OS，运行环境等
  + 演化生态：从初始版本到最终版本
  + 人员生态：使用人，开发者，运维者等
* **系统规模 = 代码数量，数据量，运用人员数**

**软件工程**

* 软件工程：将系统的、规范的、可量化的方法运用于软件开发、运行和维护的全过程。（核心特征：系统化、规范化、可量化）
* 发展思路：抽象层次不断提高，重用粒度不断增大。开发组织从个体 → 团体 → 群体发展。
* 随着软件规模增长：缺陷增多，需求复杂，生产率先升后降，成功率降低
* 软件开发
  + 本质：**软件开发 = 软件创作 + 软件生产**
  + 解决的问题：开发过程，开发方法，开发管理，质量保证
* **软件过程**：针对软件开发的有序开发活动集合，是软件开发活动的整体体现。包括活动、方法、步骤、策略
  + 技术活动：需求分析，系统设计等
  + 管理活动：项目计划与任务分配，质量管理，团队协作与沟通等
* **软件危机**：开发和维护过程中遇到的一系列问题；是**导致软件工程出现的主要原因**
  + 根源：对软件特点，规律，复杂度认识不够
  + 表现：成本高，进度难控，质量难保，维护困难，失败风险大（**风险的特定：不确定性 和 损失**）
  + 常见体现为：关键技术人员流式，有重大技术障碍，软件没有市场要求；**不包含 不适合产品整体战略（这是管理层面的问题）**
* **软件工程三要素**：过程，方法，工具
  + 过程：从管理视角，管理与生命周期
    - 瀑布、迭代、螺旋、增量、配置管理
  + 方法：从技术视角，技术与开发
    - 敏捷、DevOps、面向对象、结构化
  + 工具：从工具辅助视角，简化开发，提高效率，提高质量
    - 文档工具 WPS
    - 建模工具 UML、Rational Rose
    - 原型设计 Axure RP Pro、UIDesigner

在正式开发完整系统之前，快速构建一个可运行的原型系统（Prototype），用于 验证需求、展示功能、收集用户反馈，从而帮助开发人员和用户更好地理解系统需求。

* + - 需求分析 IBM Rational RequisitePro
    - 配置管理 Git、GitHub、PVCS
    - 开发环境 Eclipse、VSCode
* **软件工程原则** 
  + 抽象和建模：通过抽象去除无关细节，建立模型以便理解和分析复杂系统。
  + 模块化：模块之间通过接口连接，实现模块内部高聚合，外部松耦合
  + **软件重用**：在不同系统环境下多次重复使用
  + 信息隐藏：模块内的信息对外不可见，提升模块独立性，支持模块的并行开发
  + 关注点分离：简化开发任务，在不同的开发任务中针对不同的重点
  + 分而治之：降低开发复杂度
  + 双向追踪原则： 该原则保证了需求完整性、设计一致性与变更可控性
    - 正向追踪：确保所有需求均被实现与验证。
    - 反向追踪：确保所有实现都有相应的需求支撑，避免无效开发。
  + 工具辅助

**软件开发相关人员的素养**

* 软件从业人员需保证其工作内容始终与公主利益保持一致
* 技术组长负责 技术工作 和 非技术事务

## 第二章：软件开发过程管理

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **模型名称** | **关注点** | **适合软件** | **管理难度** | **是否循环** | **关键字** |
| **瀑布模型** | 与软件生命周期一致，可预先定义的应用 | ***需求变动不大***、较为明确、易预先定义 | ⭐ | ❌ 否 | 稳定、明确的项目 |
| **原型模型（演化模型）** | 用户参与、需求澄清 | **需求变动大**、不易导出和获取需求的应用；**用户反馈** | ⭐⭐ | ✅ 是 | 用户需求模糊、用户参与开发；是迭代模型和瀑布模型的组合 |
| **增量模型** | 各增量模块化、阶段交付 | ***需求变动不大***；**快速交付**；需求分阶段实现的系统 | ⭐⭐ | ⚙️ 部分循环 | 需求分阶段实现 |
| **迭代模型** | 每轮迭代都包含分析、设计、实现、测试 | **需求变动大**、难以一次性说明的项目 | ⭐⭐ | ✅ 是 | 需求不确定、持续优化 |
| **螺旋模型（演化模型）** | 风险管理与控制 | **需求变动大**、**开发风险大**、**用户反馈**，需求难以确定的系统 | ⭐⭐⭐ | ✅ 是 | **风险**驱动、结合**原型模型与瀑布模型**；可操作性差 |
| **基于构建模型** | 组件复用、兼容性 | 组件库成熟、可复用资源丰富的系统 | ⭐⭐ | ❌ 否 | 可复用资源丰富 |
| **UP模型（统一过程）** | 面向对象分析与设计 | 中大型面向对象系统 | ⭐⭐⭐ | ✅ 是 | **用例驱动**、迭代式开发、**面向对象** |
| **RUP** | UP的工程化实现 | 面向对象系统 | ⭐⭐⭐ | ✅ 是 | **面向对象**。**解决了螺旋模型的可操作性问题**；**基于UML** |
| **喷泉模型** | 阶段重叠与迭代 | 面向对象、需求变化大 | ⭐⭐⭐ | ✅ 是 | **面向对象**；**无间隙性**；**喷泉模型比迭代模型更适合需求变化大、不确定性高的场景**。 |

|  |  |
| --- | --- |
| 对比对象 | 关系与特点 |
| RUP模型 | 每一次迭代都可以看作一次完整的瀑布流程；RUP是在瀑布模型基础上发展出的迭代、面向对象开发框架。 |
| 增量模型 | 可以看作“分段的瀑布模型”；每个增量包含完整的瀑布开发流程，但只实现部分功能，逐步集成成完整系统。 |
| 快速原型模型 | 在迭代中可能使用瀑布模型；通过原型快速验证需求，原型的开发与改进仍遵循阶段性步骤（需求 → 设计 → 实现 → 测试）。 |
| 螺旋模型 | 是瀑布模型与快速原型模型的结合；每一次**顺时针**旋转都相当于执行一次完整的瀑布流程，同时加入风险分析和用户反馈。 |

**瀑布模型**

* 开发过程严格按阶段顺序推进（线性）：需求 → 设计 → 实现 → 测试 → 维护。
* 特点：
  + 与软件生命周期一致（包含维护）
  + 每个活动结束后需要评审
  + 相邻活动间存在因果关系
* 优点：简单、易于管理。
* 缺点：阶段之间缺乏反馈，变更代价高。
* 适用场景：**需求明确**且**稳定**的项目。
* 详细步骤
  + 需求分析：定义软件需求，包括功能，非功能需求，从用户视角出发
    - 产出：软件需求模型，软件需求文档，**确认测试计划**
  + **概要设计（宏观）**：自顶向下建立总体架构，制定集成测试计划
    - 依据 软件需求文档
    - 产出：软件概要设计模型，软件概要设计文档，**集成测试计划**
  + **详细设计（微观）**：设计模块内部算法
    - 依据 软件需求文档 和 概要设计文档
    - 产出：软件详细设计模型，软件详细设计文档，**单元测试计划**
  + 编程实现：编写代码和调试
    - 依据 软件需求文档 和 概要设计文档 和 **单元测试计划**
    - 产出 软件
  + 集成测试：测试整体功能；组织软件模块，并进行测试；自底向上组装
    - 依据 概要设计文档 和 **集成测试计划**
    - 产出：测试报告
  + 确认测试：测试是否满足符合用户需求
    - 依据 软件需求文档 和 **确认测试计划**
    - 产出：测试报告

**原型模型**

* 通过快速构建原型，帮助用户明确需求，再基于原型进行系统开发
* 特点：
  + 支持用户参与软件开发
  + 持续渐进明确用户需求
* 优点：能早期验证需求，减少误解。
* 缺点：原型若未及时替换，可能导致系统结构不良。
* 适用场景：用户需求模糊或交互性强的系统。

**增量模型**

* 将系统划分为多个功能增量，每次交付一个可运行子系统。
* 特点：
  + 模块化
  + 开发顺序灵活
  + 不需要一次性交付
* 优点：部分功能可提前交付，风险较低。
* 缺点：软件需求可确定，且很少变化
* 适用场景：需求可分阶段实现的系统。

**迭代模型**

* 开发以循环迭代方式进行，每次迭代都包含需求分析、设计、实现和测试。
* 优点：支持需求变更，持续改进产品质量。
* 缺点：版本管理与项目控制较复杂。
* 适用场景：需求变动大或需要持续优化的项目。

**螺旋模型（演化模型）**

* 结合瀑布与原型模型的思想，以**风险分析为驱动**，循环迭代开发。
* 优点：强调风险管理，灵活性高。
* 缺点：成本高，需有风险评估能力，不易于操作
* 适用场景：大型、复杂、风险高的项目。

**基于构建/组件的过程模型**

* 通过集成已有的组件构建系统。
* 优点：提高开发效率，增强可重用性。
* 缺点：依赖组件质量与兼容性。
* 适用场景：组件库成熟、可复用资源丰富的开发环境。

**统一过程模型/UP模型**

* 以用例驱动、以架构为中心、迭代式开发为核心思想。
  + 例如：在电商系统中，“用户下单”就是一个用例。
* 主要阶段：初始、细化、构建、移交。
* 优点：过程清晰、可扩展性强、支持面向对象开发。
* 缺点：实施复杂、对团队经验要求高。
* 适用场景：中大型面向对象系统开发。

**RUP**

* RUP 是 统一过程（UP） 的一种工程化实现，支持面向对象开发。
* 解决了螺旋模型的可操作性问题；
* 基于UML

**喷泉模型**

* 面向对象的软件开发过程模型，强调开发阶段的重叠与迭代，各阶段之间像喷泉的水流一样上下贯通（无间隙性），形成连续的开发流程。
* **特点**：
  + 阶段之间无明显界限，**分析、设计、编码、测试可以并行、反复进行**；
  + 强调**面向对象的重用机制**，以类和对象为核心；
  + 适合**不断修改和扩展**的项目。
* **优点**：
  + 支持需求变更；
  + 阶段之间自然过渡，开发效率高；
  + 促进软件复用，适应面向对象开发。
  + **比迭代模型更加适合需求变化大，不确定性高的场景**
* **缺点**：
  + 阶段并行与重叠使得进度控制与管理难度大；
  + 不适合需求和架构完全不确定的项目。
* **适用场景**：面向对象的系统开发、需求易变且强调**类复用与渐进开发**的项目。

## 第三章：需求分析

**可行性分析/项目论证**（在需求分析之前，**对软件质量的影响最大**）

* 分析内容：技术、设备、进度、成本、商业、社会可行性
  + 还需要进行一次精简的需求分析，进行初步判断
* **成本分析主要在可行性阶段完成**；设计与开发阶段主要关注技术实现与性能优化。
* **成本分析方法**
  + 类比估算法
  + 细分估算法
  + 周期估算法
  + **不包含 代码行估算法**

软件需求

* 软件需求描述软件系统应具备的功能、性能和约束条件，是用户、客户、系统和开发者对软件系统期望的明确说明。
  + 领域相关需求：由利益相关者提供（用户、客户、业务部门）
  + 技术相关需求：由软件工程师分析和确定，融合需求与技术约束
* 内容：运行环境、交付进度、功能、性能、质量要求、约束条件等
* 需求类别
  + **功能性需求**：系统必须提供的功能或服务，即“系统应该做什么”
  + **外部质量需求**：用户可感知的特性，如性能、可靠性、可用性，对用户，客户可感知
  + **内部质量需求**：系统内部结构或设计要求，对开发者可见，如模块化、接口规范
  + **开发约束性需求**：技术、工具、标准、法规等方面的限制
* 需求特点
  + **隐式性**：部分需求未被显式提出。
  + **隐晦性**：表达含糊或存在歧义。
  + 多源性：来源多样（用户、市场、法规等）。
  + 易变性：受环境、市场或技术影响。
  + 领域相关性：依赖业务知识。
  + **价值不均性**：关键需求影响系统成败。
* 多视点的需求建模：从不同的角度（视点）对系统进行描述和建模，以便更全面地理解系统需求。
  + 结构视点：系统构成（UML包图、类图、对象图、构件图）
  + 行为视点：系统行为（交互图、状态图、活动图）
  + 部署视点：软件及运行环境（部署图）
  + 用例视点：系统功能（用例图）
* 需求描述方法
  + 自然语言：最常用，但可能不具体、不准确、有二义性
  + 原型描述：直观、可操作，但无法描述所有细节
  + 用例图描述：表示系统边界、外部参与者及其交互（仅行为，不涉及数据结构）

**面向对象需求分析方法**

* 用类与对象描述系统行为
  + **抽象数据类型ADT**不是类，因为ADT是理论上的，而类是ADT的实现
* 概念
  + 消息：对象间通信，通过调用方法实现
    - **同步消息**：请求者等待响应者返回
    - **异步消息**：请求者无需等待
  + 继承：形成层次化类结构
    - **单继承**：每个类只有一个父类
    - **多继承**：每个类有多个父类
  + **多态**：相同接口，不同实现
    - **动态绑定：**指在程序运行时决定调用哪一个方法或函数的具体实现，而不是在编译时就确定。通常发生在**多态**场景中
  + **覆盖**：子类重新定义父类方法
  + **重载**：同类方法名相同，但参数不同
  + 聚合与组合：
    - 聚合：整体与部分弱相关，部分可独立存在
    - 组合：整体与部分强相关，部分不能独立存在
* **分析步骤**
  + 明确问题边界，获取需求，建立用例模型
  + 进行用例分析，形成分析模型
  + 整合模型，撰写并评审需求文档
* **面向对象建模语言**：用图形化方式表示系统中对象、类及其关系。用于抽象建模与沟通交流。
  + 组成：
    - 语法：符号及结构规则（如矩形表示类）
    - 语义：符号含义（如箭头表示继承）
    - 语用：使用指南（需求阶段用用例图，设计阶段用类图）

**软件需求工程**

* 以工程化理念，组织和管理**需求的获取、分析、验证与维护**的全过程。
* 特点
  + 知识密集型
  + 多方参与
  + 多种形式和来源获取需求
  + 持续迭代和推进
* 方法：1970 结构化方法 (DFD, 数据字典) -> 1990 面向对象方法 (UML)
* 包含3部分
  + 获取需求：通过 分析动机、识别利益相关方、参考已有系统、展示原型（早期版本）
  + 需求确认：通过评审、原型确认等方式验证需求
  + 输出：**软件需求模型（抽象表示需求）、软件需求文档（详细记录需求）**、软件原型

## 第四章：系统设计

**软件设计基本概念**

* 针对软件需求，综合考虑各种制约因素，探究软件实现的解决方案
* 高内聚（模块的功能强度）（从低到高）
  + **偶然内聚**：模块中的元素毫无关联，仅因方便放在一起❌ 不推荐
  + **逻辑内聚**：模块执行一类功能，但由外部条件选择具体操作
  + **时间内聚**：模块中元素在相同时间执行，但功能不一定相关
  + **过程内聚**：模块执行一系列相关步骤，但步骤间关联不紧密
  + **通讯内聚**：模块中的元素处理相同的数据集
  + **顺序内聚**：模块中的元素按顺序处理数据，一个活动输出是下一个活动的输入
  + **功能性内聚**：模块中所有元素都为了完成单一功能而存在✅ 最理想的内聚类型
* 低耦合（模块间的联系）（高到低）
  + **内容耦合**：一个模块直接访问或修改另一个模块的内部数据或流程
  + **公共耦合**：多个模块共享全局数据
  + **外部耦合**：模块依赖外部系统或设备接口
  + **控制耦合**：一个模块通过传递控制信息（如标志、参数）影响另一个模块的行为
  + **标记耦合**：模块通过传递复杂数据结构或对象，使另一模块只使用其中一部分数据
  + **数据耦合**：模块之间仅通过参数传递所需数据
  + **无耦合**：模块完全独立，理想状态，极少实际应用
* 软件设计基本流程：可行性分析→需求获取与分析→软件体系结构设计 / 概要设计→用户界面设计→详细设计→编码与测试
* **子系统**：它通常具有明确的职责和边界，可以单独开发、测试和维护，但又与其他子系统协作来实现整个系统的功能。**包可以视作是一个子系统**。
* **构件**：构件是系统中可独立部署、可替换的功能单元，用于封装实现特定功能的代码和资源。每个构件提供一组功能（供给接口）；同时可能依赖其他构件（需求接口）；可能由多个模块组成；**子系统包括构件，构件不包括子系统**
* **模块作用域**：指模块能够直接访问或操作的数据和资源的范围。
  + **模块控制域**：指模块内部元素
* **系统体系结构**：从**整体系统角度**对软件和硬件的设计蓝图。

**软件体系结构设计/ 概要设计（宏观）**

* 设计对象（绿色是概要设计独有的，蓝色是和详细设计一样的）
  + **子系统**：概要设计中划分的高层模块
  + **构件/组件**：可复用的软件单元
  + **连接件：**描述模块间的通信方式
  + **接口：**确定模块/构件之间的交互方式
* 输出：UML包图， 构件图，部署图，软件文档（**规格说明书**），**指定软件开发工具和平台**
* 组成：构件，连接件，约束
  + 构件
    - **连接子**：表示构件之间的连接和交互关系
    - 交互方式：过程调用、远程过程调用、消息传递、事件通知、广播、主题订阅等
  + 连接件
    - 接口：供给接口、需求接口
    - **端口**：绑定一组供给和需求接口，构件通过端口交流
  + 约束
    - 高层元素只能向低层元素发请求
    - 每个元素在适当层次
    - 层次可替换
* 视图种类：从不同角度观察和描述系统结构
  + 逻辑视图 功能结构、模块和类
    - **UML包图**
  + 运行视图 系统运行时行为、线程、进程
    - **UML逻辑部署图**
  + 开发视图 软件模块和代码组织
    - **UML构件图**
  + 物理视图 部署和硬件分布
    - **UML物理部署图**
* **精化体系结构**：在概要设计基础上，把体系结构进一步细化和落实的步骤
  + 确定公共基础设施及服务
  + 设计所需基础服务
  + 确定设计元素，包括接口
* **最高层抽象：不用过早考虑具体如何实现**
* **抽象设计**：**体系结构风格**（面向软件系统，高层抽象） > **构件设计模式**（面向子系统或构件，中层抽象） > **实现设计模式**（针对子系统或构件中的特定问题，底层抽象）

**体系结构风格**

* 分层
  + 抽象层次逐步递增或递减
  + 各层松耦合
  + 高层依赖低层接口，便于替换和复用
* 管道/过滤器
  + 系统由多个过滤器（Filter）组成，每个过滤器独立完成处理功能
  + 管道（Pipe）：连接过滤器，传递数据
  + **主动过滤器**：主动读取输入并发送输出，管道负责同步
  + **被动过滤器**：等待输入数据触发处理
* 黑板风格（健壮性高）
  + **黑板**：共享全局数据结构
  + **知识源**：写入黑板，执行特定功能
  + **控制器**：监控黑板状态，调度知识源活动
  + 健壮性高，适合复杂问题求解
* MVC（Model-View-Controller）
  + **模型（Model）**：存储业务数据，提供业务逻辑
  + **视图（View）**：展示模型数据，不直接修改数据
  + **控制器（Controller）**：协调模型和视图交互，处理用户输入
  + 分离关注点，易于维护和扩展
* SOA（Service-Oriented Architecture）
  + 功能封装为**独立服务**
  + 服务通过标准接口通信
  + 服务提供方在注册中心注册并发布服务，
  + 客户端通过注册中心查询服务，再从服务提供方获取服务
  + 构件可重用，适合分布式和异构系统
* 消息总线（Message Bus）
  + 构件之间通过统一**总线**通信
  + 适用于消息通信密集、构件异构的系统

**用户界面设计（自顶向下）**

* 设计对象：
  + **关键设计类**：由界面设计和功能需求推导出来的核心类，详细设计阶段具体化这个类
  + **界面类**：窗口、表单、按钮、菜单等界面元素
* 界面元素
  + 静态元素：与运行状态无关
  + 动态元素：与运行状态相关，不允许用户修改
  + 用户输入：编辑文本
  + 用户命令：按钮、菜单、超链接
* **用UML类图展示**
  + 窗口/对话框 → 对象类
    - 静态/动态元素 → 类属性
    - 输入元素 → 类属性
    - 命令元素 → 类方法（比如按钮用一个function实现）
* 设计过程（自顶向下）
  + 初步界面设计：确定主要窗口、界面布局
  + 建立界面跳转关系：定义各界面间导航和交互
  + 精细化界面设计：完善界面元素、风格和细节
  + 评审：验证设计合理性与可用性
* 影响设计的因素
  + 屏幕分辨率：分辨率越大，图标越小
  + 用户相关因素：用户年龄
  + 系统与技术因素：不同平台，比如IOS和安卓
  + 业务与功能需求：操作流程长短、步骤复杂性
  + 品牌统一度

**详细设计**

* 对体系结构设计和用户界面设计成果进行优化，**基本最终决定了软件的质量**
  + 详细设计必须遵从体系结构设计（接口与约束）
* 包括：数据设计、用例设计、子系统设计、接口设计、类设计、算法设计
* 设计对象：（蓝色是概要设计，紫色是用户界面设计）
  + **子系统**：概要设计中划分的高层模块
  + **构件/组件**：可复用的软件单元
  + **关键设计类**：由功能需求和界面设计推导出的核心类，这里是具体实现
  + **界面类**：界面元素类，如窗口、表单、按钮、菜单等
* 输入：软件体系结构设计、用户界面设计、软件需求
* 输出：UML类图、构件图、包图、状态图、顺序图，设计文档
* 设计过程
  + 用例设计：实现方案 → 顺序图、类图 → 优化与评审
  + 子系统设计：从封装和重用角度 → 包图、构件图、顺序图、活动图、类图
  + 类设计：精化到可实现软件 → 类图、状态图、活动图
  + 数据设计：产出类图、活动图
* 用UML活动图展示控制流程和对象的活动顺序
  + **泳道**：每个区域由对象或控制线程负责
* **Jackson 方法** 
  + 纯面向数据结构的软件设计方法
  + 核心思想：根据输入数据和输出数据的结构设计程序逻辑
  + 通过分析数据逻辑结构确定程序模块划分和处理流程

**其它知识点**

* 软件设计需要避免冗余
* 模块数量合理：模块越多，接口开销越大，因此需要选择最小成本区域
* 软件重用技术手段：封装、接口、继承、多态，文档重用（不包含覆盖，重载）
* 不同子系统职责正交，独立
* 软件体系结构设计为详细设计提供指导
* 详细设计针对体系结构局部要素

## 第五章：软件开发

**软件实现**

* 编写出目标软件系统的代码，并进行测试，并将可运行的代码部署到目标计算机上运行；兼具创作和生产
* **软件实现 = 编码，单元测试，调试**
  + 测试只负责查出错误，具体修复错误由程序开发员进行
  + 测试不止针对代码
* 输入：软件体系结构设计模型，用户界面设计模型，软件详细设计模型
* 输出：高质量的程序代码
* **程序设计语言**分类：不同领域依据“算法特性“选择不同语言，比如工程用Fortran，属性用MATLAB
  + **机器语言**：由0和1构成，执行效率高，不需要编译
  + **汇编语言**：低级语言，执行效率高，但编写效率低
  + **结构化程序设计语言**：如C、Fortran、MATLAB，高级语言，**模块粒度小**，**抽象层次低，以过程和函数为单元**
  + **面向对象程序设计语言**：如Java、C++、Python，**模块粒度大，抽象层次高，以类为基本单元**，**继承是它独有的**
  + **描述性程序设计语言**：如Prolog、Lisp、ML Meta Language，描述程序解决问题的性质

**软件编码**

* 基本概念
  + **软件编码**：将设计模型转化为可执行程序代码
  + **包**：可以视为一种子系统
* 编码准则
  + 易读性：代码清晰，便于理解
  + 降低代码复杂度：控制逻辑层次，减少嵌套
  + 软件重用：尽可能复用已有代码，编写可复用模块
  + 异常处理与容错性：保证程序健壮
  + 一致性：代码与模型、文档保持一致
* 编码风格
  + 布局格式化：统一缩进、结构清晰
  + 简洁明了：使用简单数据结构，避免变量名重复
  + 合理注释：说明逻辑，避免过多注释
  + 结构化组织：加强代码模块化，减少深层嵌套

**软件缺陷与解决方法**

* **软件缺陷**：软件制品中存在的不正确描述或实现
  + 不仅存在于代码，也存在于模型和文档
  + **软件缺陷不可避免**
  + 流程：**缺陷 → 错误 → 失效**
* 错误与失效：
  + 错误：缺陷的内部表现
  + 失效：缺陷的外在表现
* 标识：每个缺陷都有唯一标识符
* 类型
  + 需求缺陷：需求分析阶段的错误或遗漏
  + 设计缺陷：体系结构、详细设计存在问题
  + 代码缺陷：程序实现中的错误
    - 逻辑缺陷
    - 计算缺陷
    - 判断缺陷
* 修复优先级：非常紧迫→紧迫→一般→不紧迫
* 缺陷应对方法
  + 预防缺陷
    - 工程技术、方法、管理手段
    - **结对编程**、CASE工具、技术培训
      * **结对编程：**一种软件开发实践，属于极限编程方法中的核心技术；指两名程序员在同一台电脑上共同完成编码工作
  + **容忍缺陷**
    - 通过程序健壮性处理异常
  + 发现缺陷
    - 评审（设计评审、代码评审等）
  + 修复缺陷
    - 定位 + 修复
* 缺陷状态：尚未确认，有效，无效，重复，已分配，已修复，信息不完整，已解决，已关闭

## 第六章：软件测试

**软件测试**

* 与代码走查、静态分析不同，软件测试**只负责发现缺陷，不负责修复**
* 占软件开发总量的 **40% 以上**。本质是**对数据的处理与验证**。
* 关键 是 设计**有效的测试用例**
* **测试：发现缺陷 → 测试人员**
* **调试：定位缺陷 → 开发人员**
* **排错：纠正缺陷 → 开发人员**
* **Pareto测试原则**
  + **80%的错误集中在20%的模块中**
  + **测试从微观转向宏观**

**测试用例**

* 定义：为测试而设计的数据集合
* 结构（**四元偶**）：**输入数据、前置条件、测试步骤、预期输出**
* **设计时机**：应在代码编写前设计
* **可复用性**：
  + **黑盒测试** → 可复用
    - **等价划分，错误推测，边界值分析**
  + **白盒测试** → 不可复用
    - **逻辑覆盖，循环覆盖，基本路径测试（最强）**
    - 逻辑覆盖包括：**语句覆盖<判定覆盖<条件覆盖<条件组合覆盖**

**测试种类**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 测试类型 | 设计阶段 | 测试对象 | 测试方法 | 测试内容 / 目标 | 承担人员 | 备注 |
| 单元测试 | 详细设计阶段 | 模块、过程、函数、方法、类 | 白盒 | 验证数据结构、模块边界、运行环境等 | 开发人员 | 测试粒度最小 |
| 集成测试 | 概要设计阶段 | 模块间**接口**、过程调用、消息传递 | 黑盒 | 检查接口交互、消息传递是否正确 | 非开发组人员 | 策略：自顶向下（以**主控模块**为驱动，**桩模块**逐步替换为实际模块） ； 自底向上 |
| 确认测试 (验收测试) | 需求分析阶段 | **整个软件系统** | 黑盒 | 验证是否满足用户需求 | 用户 + 测试团队 | **Alpha 内测**（内部人员模拟用户操作） **/ Beta 公测**（有限的外部用户或公众进行测试） |
| 面向对象测试 | 各阶段均可 | 类、对象、继承结构 | 白盒 + 黑盒 | 测试单个类的功能、对象交互、继承关系 | 开发人员 / 测试人员 | 包括**类测试**、**交互测试**（测试对象之间消息传递）、**继承测试**（验证子类对父类功能的继承与覆盖） |
| 非功能性测试 | 系统完成后 | **系统整体** | 黑盒 | 测试性能、强度、安全性、UI、本地化、安装等 | 测试团队 | 关注系统“怎么运行”而非“做什么”；性能和强度可以同时测量 |
| 回归测试 | 每次修改后 | 受影响的模块或系统 | 白盒或黑盒 | 验证修改是否引入新缺陷、功能是否保持稳定 | 测试团队 | 维护软件稳定性与一致性 |
| 生产测试 | 系统部署后 | 生产环境中的软件系统 | 黑盒 / 灰盒 | 验证系统在真实生产环境下的非功能性需求 | 测试团队 / 运维人员 | 用于发现部署问题、硬件兼容性、实际负载下的表现 |

## 第七章：软件维护

**软件维护概念**

* 软件交付使用后，由于 **需求变化、环境变化** 或 **自身缺陷**，对系统进行改造和调整的过程。
* **关键点**：文档是影响软件可维护性的核心因素。
* 很多时候，维护人员需要为开发人员在开发阶段留下的错误或设计缺陷承担额外工作，导致维护难度增加。
* 维护副作用
  + **代码副作用**：尽量不要修改代码，因为修改可能引发连锁错误，应谨慎变更
  + **数据副作用**：数据结构或格式变动导致兼容性问题
  + **文档和模型副作用**：文档代码的一致性

**维护类型**

* 纠正性维护：纠正软件中的缺陷和错误
* **改善性维护（占比最高）**：增加新的功能，或优化
* **适应性维护**：改造软件适应新环境
  + 比如银行更新存款利率
* 预防性维护：提高软件的**可靠性和可维护性**

**维护特点**

* **同步性**：维护与运行并行进行
* 周期长：长期性活动
* **费用高**：维护费用可占总成本 **80%+**，约为开发费用的 **3倍以上**
* 难度大：系统结构复杂、依赖关系多

**软件演化**

* 对软件进行大规模功能增强和结构调整，以适应变化的需求或提高系统质量。
* 特点：功能增强粒度大，主动应对变更，持续性，引发版本变化
* **Lehman 软件演化法则**

| **法则** | **内容** |
| --- | --- |
| **持续变化法则** | 系统不修改，则越来越不实用 |
| **复杂性增加法则** | 除非有额外的工作来明显降低软件系统的复杂性，否则系统会越来越复杂 |
| **自我调节法则** | 产品与过程度量符合正态分布 |
| **组织稳定性守恒法则** | 每个版本的工作量大致相同 |
| **熟悉度守恒法则** | 平均功能增长率趋于稳定 |
| **功能持续增长法则** | 功能需持续扩展以保持用户满意度 |
| **质量衰减法则** | 代码质量会随着变更不断下降 |
| **反馈系统法则** | 演化过程具有多层次、多回路反馈结构 |

**软件老化（Software Aging）**

* 在维护与演化过程中，软件质量下降、满意度降低、变更成本上升的现象。
* 原因
* 缺乏变更：无法适应新环境（如硬件升级）
* 负面变更：修改引入错误或负面影响（如编码错误）
* 表现
* 设计恶化
  + 设计僵化（难以修改）
  + 设计脆弱（改一点出错多）
  + 模块间耦合过高
  + 结构晦涩
* 成本上升

**逆向工程**

* 基于低抽象层次软件制品，产生高抽象层次的软件制品 ；来源于硬件世界
* 用于代码重构，理解老旧系统，进而优化老化软件

**设计重构**：逆向工程的具体表现

* 软件文档缺失，则可以通过设计重构来理解软件，可以恢复设计文档

**再工程**：通过重新设计与实现优化老化系统；**再工程 = 逆向工程 + 正向工程**

**软件环境与依赖关系**

* **软件环境**：包括硬件和软件支持系统
* **软件间关系**：
  + **纵向关系**：依赖（上层依赖下层）
  + **横向关系**：交互（同层系统间通信）

**软件部署**

* 将包括软构件，配置文件，用户手册，帮助文档等进行收集，打包，安装，配置和发布到运行环境的过程。
* 工作内容包括**安装和配置 “运行环境”和“软件系统”**
* 部署原则
  + 最小化：仅部署必要组件
  + 相关性原则：相关组件同时部署
  + 适应性原则：适应多种环境配置
* 部署方式
  + **单机部署**：不需要网络通讯，适合独立运行系统
  + **分布式部署**：C/S部署方式；通过网络通信实现多节点协作
* **部署方法**
  + 基于操作系统（如Windows Installer）
  + 基于开发框架（如Spring Boot打包）
  + 基于容器和镜像（如Docker）：不需要依赖环境，即**环境一致性**
  + 基于部署图（如UML Deployment Diagram）
* 部署CASE工具
  + **Fat Jar**：Java专用打包工具（Eclipse插件）
  + **Installer** **Projects**：Microsoft Visual Studio，图形化
  + **Jenkins**：允许其它工具以插件的方式加入，通过插件实现构建、测试、部署全过程自动化

## 错题

1.下列属于软件开发项目风险的是（多）

A.关键技术人员流失

B.有重大技术障碍

C.软件没有市场需求

D.不适合产品整体战略

ABC

6.原型化开发方法包括生成原型和实现原型两个步骤。

错，因为还包括用户反馈

10.软件过程就是软件产品的加工过程。

错

【例3-4-2】画分层DFD图的基本原则不包括

A.数据守恒原则

B分解的可靠性原则

C.子、父图平衡的原则

D.数据流封闭的原则

B

1.为了提高模块的独立性,模块之间最好是

A.公共环境耦合

B.控制耦合

C.数据耦合

D.特征耦合

C

9.为高质量地开发软件项目,在软件结构设计时,必须遵循( )原则。

A.信息隐藏

B.质量控制

C.程序优化

D.数据共享

A

17.运行时动态地、而不是编译时静态地激活正确的“方法”,这种机制称为()。

A.面向对象

B.继承

C.数据封装

D.动态绑定

D

2.以下说法正确的包括（多）

A.面向对象语言支持数据封装与信息隐藏

B类是一种抽象数据类型

C.抽象数据类型就是类

D抽象数据类型既支持数据抽象,也支持过程抽象

ABD

1.系统体系结构的最佳表现性是一个可执行的软件原型。

错误，因为它是从**整体系统角度**对软件和硬件的设计蓝图。

3.在软件开发管理中,良好的( )可用来提高程序员的实现效率。

A.程序集成开发环境

B.操作系统的并行管理

C.编译程序的优化功能

D.提供并发计算的服务器

A

1.成功的测试是指运行测试用例后,

A.发现了程序错误

B.未发现程序错误

C.证明程序正确

D.改正了程序错误

A

1.以下关于软件测试的说法,正确的包括（多）

A.软件测试对保证软件质量很重要

B.软件测试应该是自发的,不需要管理

C.软件测试需要事先做好计划

D.软伴测试与软件开发一样,也要认真做好相关文档,并妥善保存

ACD

8.单元测试通常应该先进行“人工走查”,再以白盒法为主,辅以黑盒法进行动态测试。

错误

14.某银行的定期存款利率发生了变化,那么需要对该银行管理信息系统进行

A.纠错性维护

B.完善性维护

C.适应性维护

D.哪个也不是

C

5.下列哪种软件过程模型更适用于需求变化较大、不确定性较高的项目(

A.瀑布模型

B.螺旋模型

C.迭代模型

D.喷泉模型

D

6.在面向对象的需求分析中,用于描述系统功能的视图是。

A.用例视图

B.逻辑视图

C.开发视图

D.物理视图

A

9.可行性分析是要进行一次

A.精简的

B.复杂的

C.全面的

D.详细的

A

16.在软件工程中,哪个阶段对软件质量的影响最大

A.需求分析阶段

B.设计阶段

C.编码阶段

D.测试阶段

A如果需求分析不准确或不完整，后续的设计、编码、测试都会基于错误的需求进行，导致软件质量无法保证。详细设计是基本决定最终软件质量

25.在软件工程的发展历程中,哪个阶段提出了结构化编程的思想

A.初始阶段

B.成熟阶段

C.过渡阶段

D.变革阶段

B

35.在软件需求验证过程中,以下哪个方法主要用于检查需求的一致性和完整性(

A.原型法

B.场景法

C.评审法

D.测试法

C

41.UML 用例图表示

A.对象-类之间的关系

B.参与系统的角色之间的关系

C.系统状态之间的转换

D.系统功能与参与角色构成的场景

D

43.为了描述和理解系统中的控制机制,描述一个设备控制器在不同情况下索要完成的动作,宜采用

A.类图

B.状态图

C.活动图

D.协作图

B

55.项目进度安排是项目按期完成的重要保证, ()就是软件进度的管理工具。

A.软件结构图

B.DFD

C.工程网络图

D.程序流程图

C

57.软件重用是指软件模块或设计模式(

A.被多次使用

B.被循坏使用

C.被其他软件反复调用

D.在不同系统环境下多次重复使用

D

11.关于软件逻辑老化,以下哪些描述是正确的(多选)

A.软件逻辑老化是由于软件设计缺陷或逻辑错误导致的性能下降

B.软件逻辑老化是不可逆的,只能通过重新设计来解决

C.随着软件使用时间的增长,软件逻辑老化问题可能逐渐暴露

D.软件逻辑老化只影响软件的运行速度,不影响其功能

AC

14.以下哪些是关于开源软件许可证的描述（多选）

A.规定了软件的使用、修改和分发权利

B保护了开发者的知识产权

C.确保了用户的隐私权

D.鼓励用户贡献代码

E.限制了软件的商业应用

ABD

15.软件复杂性的度量参数,包括(多选）

A.软件规模

B.软件成本

C.软件文档

D.软件难度

AD

16.在软件项目管理中,哪些是项目收尾的关键活动(多选）

A.产品验收

B.项目复盘

C.文档归档

D.项目评审

ACD