NJUSE

> **Note**
>
> 该课程为本院大二课程，由67老师主讲，传统的软件工程教学方式，中文更加利于上手。
>
> 学习时间：2025.02.05 - 2025.02.20

- [x] 概述
- [x] 软件工程基础
- [x] 软件工程发展
- [x] 项目启动
- [x] 需求基础
- [x] 需求分析方法
- [x] 需求文档化与验证
- [x] 软件设计基础
- [x] 软件体系结构基础
- [x] 软件体系结构设计与构建
- [x] 人机交互
- [x] 详细设计
- [x] 模块化与信息隐藏
- [x] 面向对象的模块化
- [x] 面向对象的信息隐藏
- [x] 设计模式
- [x] 代码设计
- [x] 软件构造
- [x] 软件测试
- [x] 软件开发过程模型

概述

主要内容：

- 概念基础
- 需求分析
- 体系结构
- 详细设计
- 构造测试
- 交付演化

软件工程基础

- 软件
  - 软件发展各个阶段

    - 总结：
        - 软件独立于硬件
        - 软件是工具
        - 软件大于编程，还包括文档、数据、知识
        - 软件发展比起编程更为复杂
        - 应用软件来源于现实但高于现实(解决实际需求)
- **软件工程**
    - **定义：**
        - 运用系统的、**规范的**、**可量化**的方法来开发、运行和维护软件，即将**工程化的方法应用到软件。**
        - 对上述各种方法的研究。
    - 软件工程面临的问题：
        - 现实世界的复杂性；
        - 领域的广泛性；
        - 问题的不明确性；
        - 需要工程化的思维；
    - Engineering = Science + Principle + Art
        - S：需要计算机科学知识作为基础；
        - P：实践的经验应该被有效的分享；
        - A：需要些许创造性设计；
    - 成本效益
        - 选择最具成本效益的方法而不是最先进的方法。
    - 总结：
        - 软件工程是一种**工程活动**
        - 软件工程的动机是**解决实际问题**
        - 软件工程是**科学性、实践性和工艺性并重的**
        - 软件工程**追求足够好**，而不是最好
        - 软件工程产品是基于虚拟计算机的**软件方案**
- 知识领域：参考SWEBOK
    - 软件工程技术知识域
        - 软件需求
        - 软件设计
        - 软件构造
        - 软件测试
        - 软件维护
    - 软件工程管理知识域
        - 软件配置管理
        - 软件工程管理
        - 软件工程过程

        - 软件工程与方法
        - 软件质量
        - 相关学科知识域

❗总而言之，软件工程是一门需要多人协作合作，工程化的软件开发活动，需要掌握繁多的知识。

软件工程发展

❗软件工程发展的动力：**现实需求**

- 1950s之前：软件是硬件的一部分
- 1950s：软件工程就是硬件工程
- 1960s：**软件不再依附于硬件**
    - **人力成本**是软件开发的**主要成本**
    - **软件危机**：针对软件生产中的问题，提出了“软件工程”的方向，用工程的方法生产软件。
- 1970s：**结构化编程**、瀑布模型和形式化方法
    - 结构化编程
        - 组织程序：函数、控制逻辑、格式
        - 有组织的程序：模块化方法
            - 信息隐藏
            - 数据抽象
        - 数据流图、实体关系图和结构图来进行结构化设计
- 1980s：生产力，面向对象、重用、软件过程模型
    - 软件应用成指数型增长：软件复杂度和个人消费市场的规模
    - 软件发展方法：
        - 现代结构化分析和设计
        - **面向对象编程**
            - 结构化编程和面向对象编程对比
                - 结构化
                    - 学术化、严谨和充足的数学支持
                - 面向对象
                    - 商业化、原则的必要性和流派的复杂度
    - 软件过程：
        - 过程模型：
            - 原型、渐进交付、演化式开发、螺旋
            - 过程评价：软件成熟度模型、软件过程评价标准ISO-9001
    - > **Important**
      >
      > **没有银弹**：软件开发的根本困难在于软件产品的内在特性，它们是无法回避的，也不是可以轻易解决的，没有技术能够起到银弹的作用。
      >
      > Brooks提出的主要挑战：
      >
      > - 好的设计者

    - 快速原型
    - 迭代发展
    - 通过重用避免重复
  - 人力仍然是最主要的因素：软件工程依赖于人、为了人
- 1990s：局域网、软件架构、RUP、过程改进
  - 企业为中心，避免信息孤岛
  - **大规模软件系统**
  - 面向对象思想进一步发展，一系列面向对象的分析与设计方法提出
    - UML作为面向对象建模语言被建立和传播
    - 设计模式、面向对象设计原则等实践经验广泛传播
  - 软件体系结构：更进一步的模块化、信息隐藏
  - 软件过程
    - 过程模型：RUP、Agile、产品线
    - 过程改进：CMMI
    -
- 2000s：互联网、敏捷、混合敏捷和计划驱动
  - Web技术发展，大规模的网络应用
  - 特定领域的软件工程方法
  - 敏捷
- 2010s：云原生、大数据和AI

项目启动

> **Important**
>
> 本部分内容更详细的可查看rgp老师主讲的**软件质量。**

项目和项目管理

- 项目是具有下列特征的活动和任务
  - 具有明确目标
  - 有限定开始和结束日期
  - 有成本限制
  - 消耗人力和非人力资源
  - 多工种合作
- 项目管理的目标：
  - 限定时间
  - 一定成本
  - 要求的质量水平之上
  - 高效利用资源
  - 获得客户的认可

团队组织与管理

- 团队结构：
  - 主程序员团队、民主团队、开放团队
- 团队建设：
  - 建立团队章程、持续成功、和谐沟通、**避免团队杀手**

软件质量管理

- 建立质量模型，进行质量保障活动。
- 重点关注**评审**
- 采取质量度量进行量化

软件配置管理

- 软件配置管理的对象：最终产品以及一系列中间制品，包括软件、文档、可执行代码、相关数据等等
- 配置管理活动
  - 主要活动：标识配置项、版本管理、变更控制、配置审计、状态报告、软件发布管理
  - 版本控制：
    - 分支管理常见策略：
      - 主分支、开发分支、临时性分支
      - 善用`git`

管理实践

- 经济为本、分工协作、目标驱动、常来常往、有张有弛、**不断**总结
  - 分工协作：项目管理中角色、建议的团队构成

需求基础

需求工程

- **概念**: 所有需求处理活动的总和，包括收集信息、分析问题、整合观点、记录需求并验证其正确性。
- **三个主要任务**:
  - 说明软件系统将被应用的应用环境及其目标，以及用来达成这些目标的软件功能。
  - 将目标和功能反映到软件系统当中，映射为可行的软件行为，并对软件行为进行准确的规格说明。
  - 妥善处理目标和功能随着时间演化的变动情况。

需求开发过程模型

- **需求获取**: 从人、文档或者环境当中获取需求的过程，利用各种方法和技术来“发现”需求。
- **需求分析**: 通过建模来整合各种信息，以使得人们更好的理解问题，为问题定义出一个需求集合，并检查需求当中存在的错误、遗漏、不一致等各种缺陷，并加以修正。
  - ❗**边界分析**
    - 定义项目的范围
    - 系统边界的定义要保证能够和周围环境形成有效互动
    - 系统**用例图**常被用来定义系统的边界
  - **需求建模**

        - 类图、顺序图、状态图等建模技术。
- **需求规格说明**: 在系统用户之间交流需求信息，要简洁、精确、一致和易于理解。
- **需求验证**: 验证需求规格说明文档是否正确、准确、完整、一致、可读和可修改。
    - 同级评审、原型、模拟
- **需求管理**: 保证需求作用的持续、稳定和有效发挥，进行变更控制，纳入和实现合理的变更请求，拒绝不合理的变更请求，控制变更的成本和影响范围。

需求基础

- **需求定义**: 用户为了**解决问题或达到某些目标**所需要的条件或能力；系统或系统部件为了满足合同、标准、**规范或其它正式文档所规定的要求**而需要具备的条件或能力；对上述条件或能力的一种文档化表述。
- ❗ **需求是一种解决问题后所能达到的期望**
- **需求分类**:
    - **功能需求**: 和系统主要工作相关的需求，即用户希望系统所能够执行的活动。
    - **性能需求**: 系统整体或系统组成部分应该拥有的性能特征，例如 CPU 使用率、内存使用率等。
    - **质量属性**: 系统完成工作的质量，例如可靠性程度、可维护性程度等。
    - **对外接口**: 系统和环境中其他系统之间需要建立的接口，包括硬件接口、软件接口、数据库接口等等。
    - **约束**: 进行系统构造时需要遵守的约束，例如编程语言、硬件设施等。
    - **数据需求**: 功能需求的补充，需要在数据库、文件或者其他介质中存储的数据描述。
- **需求层次性**:
    - **业务需求**: 系统建立的战略出发点，描述了组织为什么要开发系统。**系统特性**
    - **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。
    - **系统需求**: 用户对系统行为的期望，每个系统级需求反映了外界与系统的交互行为，或者系统的一个实现细节。**交互细节**

需求分析方法

需求分析基础

- **为什么要需求分析**?
    - 建立分析模型，达成开发者和用户对需求信息的**共同理解**。
    - 依据共同的理解，发挥创造性，创建软件系统解决方案。
- **需求分析模型与建模**
    - 模型是对事物的抽象，帮助人们在创建一个事物之前可以有更好的理解。
    - 建模是对系统进行思考和推理的一种方式，目标是建立系统的表示，以精确一致的方式描述系统，使系统的使用更加容易。
    - 常用建模手段：抽象、分解
    - 常见分析模型：用例图、概念类图、顺序图、状态图、数据流图、实体关系图

面向对象分析

- **需求与用例**
    - 用例描述了在不同条件下系统对某一用户的请求的响应。

    - 一个用例是多个场景的集合，场景描述了系统是如何被使用的。
- **用例图基本元素**
    - 参与者 (Actor)：与系统交互的用户或其他系统。
    - 用例 (Use Case)：描述系统对参与者请求的响应。
    - 系统边界：定义系统范围，明确哪些内容需要详细描述，哪些不需要。
    - 关系：参与者与用例之间的关系，例如参与者使用用例。
- **用例图的建立**
    1. 目标分析与解决方案的确定
    2. 寻找参与者
    3. 寻找用例
    4. 细化用例
- **用例细化**
    - 根据业务事件和粒度进行细化，确保每个用例都描述了一个有价值的任务。
    - 避免将用例细化为单个操作、将同一业务目标细化为不同用例、将没有业务价值的内容作为用例。
        - 例如细分为增加、修改和删除
- **用例模板**
    - ID
    - 名称
    - 参与者
    - 触发条件
    - 前置条件
    - 后置条件
    - 正常流程
    - 扩展流程
    - 特殊需求

概念类图（Conceptual Class Diagram）

- 描述类（对象）和这些类（对象）之间的关系。
- 基本元素：对象、标识符、状态、行为、类、链接、关联、聚合与组合、继承
- 关联与依赖：描述对象之间的协作关系，例如对象之间的物理或业务联系。
- 继承：组织领域对象类成层次结构，类从超类继承属性和服务。
- 建立概念类图：
    1. 对每个用例文本描述，建立局部的概念类图。
    2. 根据用例的文本描述，识别候选类。
        1. 名词分析
    3. 筛选候选类，确定概念类。
        1. 原则：依据系统的需求、该类的对象实例的状态与行为是否完全必要
    4. 识别关联。

    1. 概念类之间的协作
    2. 概念类之间的整体部分关系
    3. 去除冗余关联和导出关联
5. 识别重要属性。
6. 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。

顺序图 (Sequence Diagram)

- 显示对象之间为完成某个用例而进行的交互行为。
- 基本元素：参与者、对象、消息、激活条、控制结构

状态图 (State Chart)

- 显示对象在不同状态之间的转换，以及触发这些转换的事件和动作。
- 基本元素：状态、转换、事件、动作、监护条件

结构化方法

- 结构化分析 (Structured Analysis) 思想：
    - 自顶向下分解
    - 图形化表示：数据流图 (DFD)、实体关系图 (ERD)
- 数据流图 (DFD)
    - 将系统看做是过程的集合，过程就是对数据的处理。
    - 基本元素：外部实体、过程、数据流、数据存储
    - 分层结构：上下文图、0层图、N层图
    - 需要注意数据流必须和过程产生关联，要么是过程的数据输入，要么是过程的数据输出
- 实体关系图 (ERD)
    - 考察数据对象，独立于处理过程。
    - 基本元素：实体、属性、关系
    - 关系类型：一对一、一对多、多对多
    - 键 (Key)：唯一确定和标识实体实例的属性或属性组合。

使用需求分析方法细化和明确需求

- **为什么要细化**?
    - 用户需求的描述的模糊性与系统设计所需的严谨性之间的矛盾。
- **如何细化**?
    - 需求分析建模，发现遗漏、冲突、冗余和错误。
    - 迭代：获取、分析、获取、分析...
- **系统顺序图有助于发现**
    - 交互性的缺失
- **概念类图有助于发现**
    - 部分信息的使用不准确
    - 部分信息不明确
    - 遗漏了重要内容

- **状态图有助于发现**
  - 界面的跳转

建立系统需求

- 8种规格说明
- 不同的分析方法适合不同的规格说明

需求文档化与验证

需求文档化

- **为什么文档化需求**?
  - 团队协作和沟通：文档化需求可以确保团队成员对需求的理解一致，并方便后续的开发、测试和维护工作。
  - 项目管理：需求文档是项目计划和管理的重要依据。
- **用例文档**：
  - 从用户角度描述软件系统与外界的交互，以用例文本为主。
  - 主要职责是将**问题域信息和需求**传达给软件系统解决方案的设计者。
  - 示例：电梯乘坐用例，包括场景描述、异常情况等。
- **软件需求规格说明文档 (SRS)**：
  - 从软件产品的角度，以系统级需求列表的方式描述软件系统解决方案。
  - 示例：电梯系统规格说明，包括功能需求、相关功能需求等。
- **文档化需求的注意事项**：
  - **技术文档写作要点**：简洁、精确、易读、易修改。
  - **系统化的方式**：使用相同的语句格式、列表或表格、编号等。
  - **避免歧义词汇**：例如“可接受的”、“有效的”、“灵活的”等，需要明确其具体含义。
  - **需求书写要点**：使用用户术语、可验证、可行性。
  - **需求规格说明文档书写要点**：充分利用标准模板、保持完备性和一致性、划分优先级。

需求验证

- **验证需求的方法**：
  - 评审：重视需求评审，保证用户与客户参与，使用检查列表。
  - 开发系统测试用例：基于用例描述，设计测试场景的输入与输出数据。
- **测试用例套件**：基于用例描述，确定测试用例套件，并建立测试用例对需求的覆盖情况。

需求度量

- **度量需求**：用例数量、平均每个用例中的场景数量、平均用例行数、软件需求数量、非功能需求数量、功能点数量。
- **度量的意义**：评估需求的完整性、粒度、遗漏等。
- **功能点度量**：
  - 用于估算和度量软件系统规模与复杂度的抽象单位。
  - 计算公式：功能点 = (输入数量 × 加权因子) + (输出数量 × 加权因子) + … + (对外接口数量 × 加权因子) × (0.65 + 0.01 × 修正因子)

    - 修正因子考虑系统的复杂度、性能、易修改性等因素。

软件设计基础

什么是软件设计?

- 软件设计是将需求转换为软件解决方案的过程，它包括规划、思考、建模、交互调整和重新设计等环节。
- 软件设计的目标是创建有用、可靠、高效、易维护和可复用的软件产品。
- 软件设计是**工程**和**艺术**的结合，需要考虑审美、功能、用户关系、过程关系、大规模软件设计、体系结构设计、复用、产品线、框架、构件和设计模式等因素。

为什么要做设计?

- 事物的复杂性 vs 思维的有限性：为了应对**软件的复杂性**，需要通过设计将问题分解和抽象，以便更好地理解和解决。
- 关注点分离与层次性：设计可以帮助我们将不同的关注点分离，并以层次化的方式组织软件系统，从而降低复杂性。

软件设计思想的发展

- 从程序设计到软件设计：早期软件开发主要关注程序设计，随着软件规模和复杂性的增加，软件设计的重要性逐渐凸显。
- 从结构化设计到面向对象设计：结构化设计方法强调模块化和信息隐藏，面向对象设计方法则更加强调对象、类和继承等概念。
- 从低层设计到高层设计：软件设计可以分为低层设计（代码设计）、中层设计（模块设计）和高层设计（体系结构设计）等层次。

软件设计的核心思想

- **分解与抽象**： 将复杂问题分解成更小的子问题，并通过抽象隐藏细节，从而降低复杂性。
- **关注点分离**： 将不同的关注点分离，并以模块化的方式组织软件系统。
- **信息隐藏**： 隐藏模块内部的实现细节，只暴露必要的接口。
- **抽象数据类型**： 定义数据类型及其操作，隐藏数据的具体表示和操作细节。
- **封装**： 将数据和操作封装在一起，形成一个独立的单元。

软件设计的分层

- **低层设计**： 关注代码层面的设计，例如数据结构、算法、代码风格等。
    - 本质：屏蔽程序中复杂数据结构与算法的实现细节。
- **中层设计**： 关注模块层面的设计，例如模块划分、接口设计、信息隐藏等。
    - 模块划分隐藏一些程序片段的细节，暴露接口于外界。
- **高层设计**： 关注系统层面的设计，例如系统架构、系统行为、质量属性等。

软件设计过程、方法和模型

- **软件设计过程的主要活动**： 分析设计需求、建立设计模型、生成候选方案、评审等。
- **软件设计的方法**： 结构化设计方法、面向对象设计方法、数据为中心设计方法、基于构件的设计方法、形式化方法设计等。
- **软件设计的模型**： 静态模型（实体关系图、设计类图等）和动态模型（数据流图、状态图等）。

软件设计描述

- **软件设计描述规范**： IEEE 1016-1998 和 IEEE 1016-2009 标准规定了软件设计文档的规范。
- **设计视角**： 根据不同的利益相关者（例如顾客、架构师、设计师、开发人员、测试人员、维护人员）的需求，提供不同的设计视图。
- **设计理由**： 解释设计决策的原因和依据。
- **设计文档书写要点**： 充分利用标准的文档模板、使用体系结构风格的图、利用完整的接口规格说明、从多视角出发、体现对于变更的灵活性等。

体系结构基础

> **Important**
>
> 本部分详细可参考zh老师主讲的软件架构

软件体系结构的发展

- **1969年**: 第一次出现“软件体系结构”一词。
- **20世纪80年代**: 开始关注软件系统的**组织结构**，代表人物包括 Brooks、Lampson、Parnas 和 Mills。
- **1992年**: Perry 和 Wolf 提出“软件体系结构 = {元素, 形式, 原因}”的公式，并加入“约束”因素。
- **1995年**: IEEE 发布 IEEE 1471-2000 标准，指导软件密集型系统的体系结构描述。

理解软件体系结构

- **定义**: 软件体系结构是指软件系统的结构、组织方式、组件间交互以及这些交互背后的原因和约束。
- 简洁定义：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**。
- **重要性**:
  - 促进沟通
  - 早期设计决策
  - 系统的可移植性
- **高层抽象**:
  - **组件 (Component)**: 封装处理和数据，提供特定服务。
  - **连接件 (Connector)**: 定义组件间的交互方式。
  - **配置 (Configuration)**: 定义组件和连接件的组织方式，形成系统整体结构。

体系结构风格初步

- **主程序/子程序风格**: 基于程序调用关系建立连接件，形成层次结构，适用于功能可分解为顺序步骤的系统。
- **面向对象风格**: 基于数据信息和操作封装建立对象组件，通过方法调用建立连接件，适用于数据信息相关的系统。
- **分层风格**: 组件抽象层次逐级提升，下层为上层提供服务，形成层次结构，适用于可分解为不同抽象层次任务的系统。
- **模型-视图-控制器 (MVC) 风格**: 分离业务逻辑、表现和控制，适用于用户界面需要灵活修改的系统。

设计决策与约束

每种体系结构风格都有其设计决策和约束，例如：

- **主程序/子程序**: 单向依赖、层次分解。
- **面向对象**: 数据封装、信息隐藏、方法调用。
- **分层**: 层次划分、单向依赖、标准接口。
- **MVC**: 模型、视图和控制**分离**，单向依赖。

软件体系结构设计与构建

> **Important**
>
> 本部分更加详细同样可参考zh老师主讲的软件体系结构

体系结构设计

- **设计过程**:
    1. 分析关键需求和项目约束
    2. 选择体系结构风格
    3. 逻辑设计（抽象设计）
    4. 依赖逻辑设计（实现设计）
    5. 完善设计
    6. 添加构件接口
    7. 迭代 (3-7)
- **需求类型**:
    - 功能需求
    - 非功能性需求 (质量、性能、接口、约束)
    - 项目约束 (开发团队、市场大小、预算、进度、风险、开发环境、技术)
- **实践案例**:
    - 使用用例模型 (UCM) 分析需求
    - 将需求分配到子系统和模块 (考虑功能和可复用性)
    - 选择合适的体系结构风格 (如分层风格)
    - 评估和改进设计 (使用非功能性需求和项目约束)

体系结构构建

- **物理设计**:
    - 包设计原则 (REP, CCP, CRP, ADP, SDP, SAP)
    - 包设计过程 (迭代，先 CCP，再 CRP/REP，最后 ADP/SDP/SAP)
    - 初始物理包设计
    - 细节考虑 (RMI, 数据持久化, 图形界面, 接口包, 数据对象, 循环依赖)
- **4+1 视图模型**:
    - 逻辑视图 (功能)
    - 进程视图 (动态行为)
    - 开发视图 (程序员视角)
    - 物理视图 (部署)
    - 场景视图 (用例)

- **构件设计**:
  - 确定模块对外接口
  - 编写接口规范
  - 关键需求的实现

体系结构集成与测试

- **集成策略**:
  - 大爆炸式
  - 增量式 (自顶向下、自底向上、三明治式)
  - 持续集成
- **桩程序和驱动程序**:
  - 桩程序模拟未完成的模块
  - 驱动程序调用模块接口

体系结构文档化

- **文档模型 (IEEE 1471-2000)**
- **体系结构评审**:
  - 评审角度 (正确性、合理性、明确性)
  - 评审方法 (Checklist, ATAM)

人机交互

> **Important**
>
> 本部分更详细的可参照fgh老师主讲的**人机交互**课程

❗人机交户设计**以用户为中心**

HCI 设计目标：可用性

- **易学性**: 用户可以轻松快速地学习如何使用系统。
- **效率**: 熟练用户可以高效地完成任务。需注意**易学性和效率是存在冲突的。**
- **可记忆性**: 中断用户可以轻松恢复操作，无需重新学习。
- **错误率低**: 用户犯错少，且可以快速恢复。
- **满意度**: 用户对系统感到满意。

HCI 三大要素

- **人**: 用户是使用计算机技术完成任务的人，他们有不同的技能、经验和期望。
  - 好的人机交互应该为不同的用户群体提供差异化的交互机制。
- **计算机**: 计算机设备包括输入设备 (键盘、鼠标等) 和输出设备 (显示屏、打印机等)。
  - 可视化设计的要点：
    - 按照任务模型设计界面隐喻，不要将软件内部构造暴露给用户
    - 可视化设计还应该基于界面隐喻，尽可能把功能和任务细节表现出来
- **交互**: 人机交互是**双向的**，用户向系统提出请求，系统给予响应，并提供信息。
  - 交互形式：GUI、菜单、表格、命令行、自然语言.....

    - 导航：需得符合**人的精神模型**
    - 反馈：根据任务选择适当的响应时间

HCI 设计过程

- **定义使用环境和用户需求**: 了解用户是谁，他们如何使用系统，以及他们需要完成哪些任务。
- **设计解决方案**: 根据用户需求设计系统界面和功能。
    - **原型**
- **评估设计**: 测试设计是否满足用户需求，并根据反馈进行调整。

GUI 设计原则

- **用户控制**: 用户应该能够控制他们的操作，并了解他们的操作结果。
- **减少记忆负担**: 系统应该帮助用户减少记忆负担，例如使用直观的快捷方式、设置有意义的默认值等。
- **一致性**: 系统应该保持一致性，例如使用相同的术语、图标和交互方式。
- **反馈**: 系统应该提供及时、清晰、有用的反馈，让用户了解他们的操作结果。
- **简洁性**: 系统应该简洁明了，避免使用不必要的复杂功能。
- **易记性**: 系统应该易于记忆，例如使用用户熟悉的语言和概念。
- **帮助和文档**: 系统应该提供帮助和文档，帮助用户了解如何使用系统。

其他重要概念

- **认知模型**: 用户对系统工作方式的假设。
- **导航**: 帮助用户找到完成任务入口的机制。
- **响应时间**: 系统对用户操作的响应时间。
- **协作式设计**: 让计算机适应人的因素，以实现更顺畅的人机交互。
- **易用性测试**: 通过用户测试评估系统的可用性。
- **可访问性**: 确保系统对所有用户都可用，包括残障人士。

详细设计

详细设计基础

- **定义**: 详细设计是介于软件架构和代码实现之间的一个中间阶段，它将软件架构中定义的模块进行更细致的设计，并落实到具体的类和对象上。
- **输入**: 需求规格说明书 (SRS)、软件体系结构设计文档。
- **输出**: 详细设计文档，包括类图、顺序图、状态图等。
- **出发点**:
    - **需求分析**: 需求分析的结果，如用例图、领域模型、顺序图、状态图等，会直接影响详细设计的内容。
    - **软件体系结构**: 软件体系结构定义了模块之间的接口和关系，详细设计需要遵循体系结构的设计原则。
- **上下文**: 详细设计需要考虑模块的输入输出接口、模块内部的职责分配以及模块之间的协作关系。

面向对象详细设计

- **思想**:

    - **职责 (Responsibility)**: 每个类都应该承担一定的职责，包括操作职责和数据职责。职责的分配应该遵循高**内聚低耦合**的原则。
    - **协作 (Collaboration)**: 对象之间需要相互协作才能完成复杂的任务。协作的方式可以是分散式、集中式或委托式。
- **过程**:
    1. **设计模型建立**:
        - 通过职责建立静态设计模型：确定类的职责，并建立类之间的关系。
        - 通过协作建立动态设计模型：确定对象之间的协作关系，并选择合适的控制风格。
    2. **设计模型重构**: 根据模块化和信息隐藏的原则，对设计模型进行重构，以提高系统的可维护性和可扩展性。
    3. **利用设计模式**: 使用设计模式可以提高代码的可重用性和可维护性。
- **GRASP 原则**:
    - **信息专家 (Information Expert)**: 将**职责分配给拥有完成该职责所需信息的类。**
    - **创建者 (Creator)**: 将创建对象的职责分配给最了解该对象用途的类。
    - **控制器 (Controller)**: 将处理系统事件的职责分配给控制器类。

为类间协作开发集成测试用例

- **集成测试**: 集成测试是测试**模块之间协作**的有效方法，它可以发现模块之间的接口问题和协作问题。
- **Mock Object**: Mock Object 是一种模拟对象，它可以模拟其他对象的的行为，以便进行单元测试和集成测试。

结构化详细设计

- **思想**: 结构化设计是一种自上而下的设计方法，它将系统分解成一系列相互关联的过程。
- **工具**: 数据流图 (DFD) 和结构图 (SC) 是结构化设计的主要工具。

详细设计文档描述和评审

- **文档内容**: 详细设计文档应该包括模块的静态结构、动态行为、接口规范、实现注解和设计原理等内容。
- **评审**: 详细设计文档需要进行评审，以确保其质量。

模块化与信息隐藏

> **Important**
>
> **模块化与信息隐藏**是软件工程中重要的设计原则，旨在构建易于理解、维护和扩展的系统。

动机

- **设计良好的软件**:
    - **管理**: 将开发工作分解，实现“分而治之”。
    - **演化**: 解耦系统组件，使修改局部化，避免牵一发而动全身。
    - **理解**: 将系统分解成易于理解的模块，降低复杂性。

发展历程

- **萌芽阶段 (1970年代)**: Wirth 提出逐步求精方法，Parnas 提出信息隐藏原则。

- **形成阶段 (1970年代-1980年代)**: Stevens 提出“结构化设计”方法，进一步发展模块化和信息隐藏的概念。
- **发展阶段 (1990年代)**: Eder 和 Hitz 等人将模块化和信息隐藏原则应用于面向对象系统。
- **反思阶段 (1990年代至今)**: McConnell 和 Demarco 等人对模块化和信息隐藏原则进行反思和评估。

模块化

- **模块**: 具有明确定义接口和功能的代码单元，可以独立开发、测试和部署。
- **模块化**: 将系统分解成多个模块的过程。分而治之，便于管理、演进、理解。
- 关键问题：模块化使用什么标准？信息隐藏。

信息隐藏

- **信息**: 模块内部的实现细节，例如数据结构、算法等。
    - 信息？秘密！
        - 什么是秘密？会产生变化的部分。
- **隐藏**: 将信息封装在模块内部，避免外部访问和修改。
- **原则**:
    - **局部化变化**: 将可能变化的实现细节隐藏起来，减少修改的影响范围。
    - **隔离不同变化速率的部分**: 将变化速率不同的部分分离，避免相互影响。
    - **暴露稳定的假设**: 在接口中暴露稳定的假设，避免不必要的修改。

耦合与内聚

- **耦合**: 模块之间的依赖程度。
- **内聚**: 模块内部元素之间的关联程度。
- **目标**: 高内聚，低耦合。

耦合类型

- **数据耦合**: 模块之间通过参数传递数据。
- **控制耦合**: 模块之间传递控制信息。
- **公共耦合**: 模块之间共享全局变量或环境。
- **内容耦合**: 一个模块修改另一个模块的代码。
- **印记耦合**: 模块之间传递过多的数据。

内聚类型

- **偶然内聚**: 模块内部元素之间没有关联。
- **逻辑内聚**: 模块内部元素完成逻辑上相关的功能。
- **时间内聚**: 模块内部元素在特定时间执行。
- **过程内聚**: 模块内部元素按特定顺序执行。
- **通信内聚**: 模块内部元素访问相同的数据。
- **功能内聚**: 模块内部元素完成单一功能。
- **信息内聚**: 模块内部元素访问相同的信息。

模块指南

- **主要秘密**: 模块要实现的功能需求。
- **次要秘密**: 模块实现的细节，例如数据结构、算法等。
- **角色**: 模块在系统中的功能和作用。
- **接口**: 模块提供的功能和服务。

信息隐藏的应用

- **分层设计**: 将系统分解成多个层次，降低耦合。
- **物理包设计**: 将模块分组，消除重复，降低耦合。
- **控制风格**: 将控制逻辑与业务逻辑分离，提高内聚。

总结：模块化和信息隐藏是构建高质量软件的重要原则，可以帮助我们设计易于理解、维护和扩展的系统。通过理解这些原则，我们可以更好地进行软件设计和开发。

面向对象的模块化

耦合和内聚

- **耦合**指的是模块之间互相依赖的程度。耦合越高，模块之间的关联越紧密，修改一个模块可能需要修改其他模块，导致系统维护困难。
- **内聚**指的是模块内部元素的关联程度。内聚越高，模块内部的元素越相关，模块的功能越单一，模块的维护越容易。

降低耦合的设计原则

- **避免全局变量**: 全局变量会增加模块之间的耦合，应该尽量避免使用。
- **明确声明**: 明确声明模块之间的依赖关系，避免隐式的耦合。
- **避免重复**: 避免重复代码，可以使用函数或类来复用代码。
- **面向接口编程**: 通过接口来定义模块之间的交互，而不是直接依赖具体的实现类。
- **迪米特法则**: 一个对象应该尽可能地少了解其他对象的信息，**只与直接相关的对象进行交互**。
- **接口隔离原则**: 将不同客户端需要的接口分离，避免客户端依赖不需要的接口。
- **里氏替换原则**: 子类应该能够替换父类，保证程序的语义一致性。
- **优先组合而非继承**: 组合可以避免继承带来的耦合问题，并且更加灵活。

单一职责原则

- **单一职责原则**: 一个类应该只有一个改变的理由，即只有一个职责。

耦合和内聚的度量

- **类间耦合度**: 度量一个类与其他类的耦合程度。
- **数据抽象耦合度**: 度量一个类对其他类的数据抽象类型的依赖程度。
- **出向耦合和入向耦合**: 度量一个类与其他类的依赖关系。
- **继承树深度**: 度量一个类在继承树中的位置。
- **子类数量**: 度量一个类的子类数量。
- **类内耦合度**: 度量一个类内部方法的耦合程度。

面向对象的信息隐藏

核心概念:

- **信息隐藏**：每个模块都隐藏了重要的设计决策（秘密），以便只有该模块的组成部分才知道细节。这包括：
  - **主要秘密**：职责变化，即软件设计师在软件需求规格说明书 (SRS) 中指定的隐藏信息。
  - **次要秘密**：实现变化，即设计者在实现模块以隐藏主要秘密时做出的实现决策。
- **职责**：类或对象维护一定的状态信息，并基于状态履行行为职能的能力。
- **封装**：将数据和操作组合在一起，并隐藏实现细节，只暴露必要的接口。
  - **接口**：对象之间交互的消息（方法名）、消息中的所有参数、消息返回结果的类型、与状态无关的不变量、需要处理的异常。
  - **实现**：数据、结构、其他对象的引用、类型信息、潜在变更等。
- **抽象**：关注对象的视图，将对象的行为与其实现分离。
- **多态**：允许以相同的方式处理不同类型的对象。

设计原则：

- **最小化类和成员的可访问性**：只有必要时才暴露类的成员，以减少对外部的影响。
- **开闭原则 (OCP)**：软件实体应该对扩展开放，对修改封闭。这意味着我们应该使用抽象和接口来设计类，以便在不修改现有代码的情况下添加新功能。
- **依赖倒置原则 (DIP)**：高级模块不应该依赖于低级模块，两者都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。这意味着我们应该使用接口和抽象类来定义类之间的关系，以便在不修改现有代码的情况下更改实现。

如何应对变化：

- 识别应用程序中可能变化的方面，并将它们与不变的方面分离。
- 将可变的方面封装起来，以便在不影响其他部分的情况下更改或扩展它们。
- 使用抽象和接口来设计类，以便在不修改现有代码的情况下添加新功能。
- 使用接口和抽象类来定义类之间的关系，以便在不修改现有代码的情况下更改实现。

设计模式

> **Important**
>
> 本部分更详细的可参考pmx老师主讲的**软件系统设计**

可修改性

- 软件设计中，实现的可修改性至关重要，它包括：
  - **实现的可修改性 (M)**：修改已有实现，例如修改现有促销策略。
  - **实现的可扩展性 (E)**：扩展新的实现，例如增加新的促销策略。
  - **实现的灵活性 (C)**：动态配置实现，例如动态修改商品对应的促销策略。
- **如何实现可修改性**?
  - 通过接口与实现的分离，降低代码耦合，提高可修改性。
  - **接口** 定义规约，**实现类** 实现规约。
  - **继承** 也可以实现接口与实现的分离，还可使子类继承父类的实现，但存在局限性，例如子类与父类耦合度高，无法动态配置实现。
  - **组合** 通过接口的组成关系，实现更好的灵活性，前端类和后端类是组合关系，前端类重用后端类代码。

设计模式

- 设计模式是**抽象的、可复用**的解决方案，用于解决软件设计中常见问题。
- 设计模式包含：
  - **典型问题**：描述了设计模式要解决的问题。
  - **设计分析**：分析问题，并提出解决方案的思路。
  - **解决方案**：描述解决方案的组成、协作方式、应用场景和使用注意点。
  - **案例**：展示设计模式的应用实例。

典型设计模式

- **策略模式**：定义算法族，将算法封装起来，使算法的变化独立于使用算法的客户。
  - **参与者**：上下文 (Context)、策略 (Strategy)、具体策略 (ConcreteStrategy)。
  - **协作**：上下文配置具体策略，将请求转发给策略类实现。
  - **应用场景**：实现行为的多态，消除分支选择语句，隐藏复杂算法。
- **抽象工厂模式**：定义创建对象的接口，由子类决定要实例化哪一个类。
  - **参与者**：抽象工厂 (AbstractFactory)、具体工厂 (ConcreteFactory)、抽象产品 (AbstractProduct)、具体产品 (ConcreteProduct)。
  - **协作**：具体工厂创建具体产品，客户使用抽象工厂和抽象产品接口创建产品。
  - **应用场景**：独立于产品的创建、配置产品族、强调产品族的一致性。
- **单例模式**：确保一个类只有一个实例，并提供一个全局访问点。
  - **设计原则**：将构造方法私有化，通过静态方法 getInstance() 获取唯一实例。
- **迭代器模式**：提供一种顺序访问聚合对象各个元素的方式，而不暴露其内部表示。
  - **参与者**：迭代器 (Iterator)、具体迭代器 (ConcreteIterator)、聚合 (Aggregate)、具体聚合 (ConcreteAggregate)。
  - **协作**：具体迭代器跟踪聚合中的当前对象，并提供遍历方法。
  - **应用场景**：访问聚合对象内容，支持多种遍历方式，为不同聚合结构提供统一接口。

代码设计

设计易读的代码

- **代码规范**：
  - **格式**： 使用缩进和对其表达逻辑结构，将相关逻辑组织在一起，使用空行分割逻辑，长句断行。
  - **命名**： 使用有意义的名称进行命名，例如使用 `Sales` 而不是 `ClassA`。
  - **注释**：
    - 使用不同类型的注释，例如语句注释、标准注释和文档注释。
    - 使用 Javadoc 工具生成代码文档。
    - 注释要有意义，不要简单重复代码的含义，重视对数据类型和复杂控制结构的注释。

设计易维护的代码

- **小型任务**： 将复杂功能分解为多个小型、高内聚、低耦合的任务。
- **复杂决策**：

    - 使用新的布尔变量简化复杂决策。
    - 使用有意义的名称封装复杂决策。
    - 使用表驱动编程。
- **数据使用**：
    - 不要将变量应用于与命名不相符的目的。
    - 不要将单个变量用于多个目的。
    - 限制全局变量的使用。
    - 不要使用突兀的数字与字符，将其定义为常量或变量后使用。
- **明确依赖关系**： 类之间模糊的依赖关系会影响到代码的理解与修改，容易导致修改时产生未预期的连锁反应。

设计可靠的代码

- **契约式设计**： 使用前置条件和后置条件定义函数或方法的输入和输出。
    - **异常方式**： 使用异常处理机制来处理错误情况。
    - **断言方式**： 使用断言语句来验证代码的假设。
- **防御式编程**： 在方法与其他外界环境交互时，保护方法内部不受损害。

使用模型辅助设计复杂代码

- **决策表**： 将复杂决策逻辑表示为表格形式。
- **伪代码**： 使用自然语言描述算法。
- **程序流程图**： 使用图形表示程序的执行流程。

为代码开发单元测试用例

- **方法测试**： 根据方法的规格和代码的逻辑结构开发测试用例。
- **类测试**： 测试类中不同方法之间的互相影响情况。
- **Mock Object**： 使用 Mock Object 来模拟外部依赖。
- **JUnit 测试代码**： 使用 JUnit 框架编写测试代码。

代码复杂度度量

- **圈复杂度**： 计算程序中独立路径的最大数量。
- **类的复杂度**： 基于所拥有方法的代码复杂度定义类的复杂度。

代码大全

- **变量**：
    - **定义**： 关闭隐式声明，声明全部的变量，遵循某种命名规则，检查变量名。
    - **初始化**： 在声明变量的时候初始化，在靠近变量第一次使用的位置初始化，特别注意计数器和累加器，在类的构造函数里初始化该类的数据成员。
    - **作用域**： 使变量应用局部化，变量跨度尽可能小，尽可能缩短变量的存活时间，变量存活时间尽可能小。
    - **持续性**： 在程序中加上调试代码或者断言来检查那些关键变量的合理取值，准备抛弃变量时给它们赋上“不合理的值”，养成在所有数据之前声明和初始化的习惯。
    - **为变量制定单一用途**： 避免让代码具有隐含意义，避免让代码具有隐含意义。

    - **避免使用“神秘数值”**： 如果需要，可以使用硬编码的 0 和 1，预防除 0 的错误，使类型转换变得明显，避免混合类型的比较，注意编译器的警告。
    - **整数**： 检查整数除法，检查整数溢出，检查中间结果溢出。
    - **浮点数**： 避免数量级相差巨大的数之间的加减运算，避免等量判断，处理舍入误差，检查语言和函数库对特定数据类型的支持。
- **子程序**：
    - **创建子程序的正当理由**： 降低复杂度，引入中间、易懂的抽象，避免代码重复，支持子类化，隐藏顺序，隐藏指针操作，提高可移植性，简化复杂的布尔判断，改善性能。
    - **好的子程序名字**： 描述子程序所做的所有事情，避免使用无意义的、模糊或者表述不清的动词，不要同多数字来形成不同的子程序名字，根据需要确定子程序名字的长度，给函数命名时要对返回值有所描述，给过程起名时使用语气强烈的动词加宾语的形式，准确使用对仗词，为常用操作确立命名规则。
- **算法的设计**： 在执行时间与设计质量、标准、和客户需求之间平衡考虑。
- **一般控制问题**：
    - **布尔表达式**：
        - 用 true 和 false 做布尔判断，不要用 0 和 1
        - 简化复杂的表达式，拆分复杂的判断并引入新的布尔变量，把复杂的表达式做成布尔函数，用决策表代替复杂的条件
        - 编写肯定形式的布尔表达式，用 DeMorgan 定律简化否定的布尔判断，用括号使布尔表达式更清晰，短路求值
        - 按照数轴的顺序编写数值表达式
        - 与 0 比较
            - 隐式地比较逻辑变量
            - 把数与 0 相比较
            - 在 C 中显示地比较字符和零终止符（'\0'）
            - 把指针与 NULL 相比较。
    - **复合语句**： 把括号对一起写出，用括号被条件表达清楚。
    - **空语句**： 可以通过加 {} 等来强调空语句，或者为创建一个 DoNothing() 预处理宏或者内联函数，更加清晰的非空循环体。
    - **驯服危险的深层嵌套**：
        - 通过重复检测条件中的某一部分来简化嵌套的 if 语句
        - 用 break 块来简化嵌套 if
        - 把嵌套 if 转换成一组 if-then-else 语句
        - 把嵌套 if 转换成 case 语句
        - 把深层嵌套的代码抽取出来放进单独的子程序
        - 使用面向对象的方法。

软件构造

软件构造概述：

- **软件构造** 是通过编码、验证、单元测试、集成测试和调试等工作，将详细设计转化为可运行软件的过程。

- **区别于实现**：软件构造不仅仅是编程，还包括详细设计、单元测试、集成测试、调试、代码评审、集成与构建以及构造管理等活动。

软件构造活动：

- **详细设计**：根据编程语言的约束调整详细设计方案。
- **编程**：产生可读、易维护、可靠、高性能、安全的程序代码。
- **测试**：包括单元测试和集成测试，确保程序的正确性。
- **调试**：定位并修复程序错误，常见错误包括内存泄漏、多线程错误、逻辑错误等。
- **代码评审**：通过同行评审发现并修正代码错误，提高软件质量和开发者技巧。
- **集成与构建**：将分散的代码单元集成和构建为可运行软件。
- **构造管理**：包括构造计划、度量和配置管理。

实践方法：

- **重构**： 在不改变代码外部表现的情况下改进其内部结构，消除代码坏味道。
- **测试驱动开发**： 在编写代码之前先编写测试代码，确保代码的正确性。
- **结对编程**： 两个程序员协同工作，共同完成设计、算法、代码或测试。

软件构造理念：

- **设计层次提升**： 编程语言能力提升，能够创建更大的代码单元。
- **每日构建和冒烟测试**： 实现增量式集成，减少集成问题。
- **标准库**： 提供丰富的功能，简化开发过程。
- **开源软件**： 促进代码共享和学习。
- **Web 资源**： 提供丰富的学习资源和交流平台。
- **增量式开发**： 将大型项目分解为多个小型项目，逐步完成。
- **测试驱动开发**： 提高代码质量和开发效率。
- **重构**： 保持代码的整洁和可维护性。
- **快速计算机**： 改变优化和编程语言的选择。

软件构造的十大现实：

- **软件构造是一个合法主题。**
- **个人差异显著。**
- **个人纪律很重要。**
- **关注简洁性比关注复杂性更有效。**
- **缺陷成本随时间增加。**
- **设计很重要。**
- **技术浪潮影响构造实践。**
- **增量式方法最有效。**
- **工具箱比喻仍然具有启发性。**
- **软件存在基本张力。**

20 世纪 90 年代和 21 世纪初最糟糕的构造理念：

- **无设计。**

- **全面设计。**
- **无文档。**
- **全面文档。**
- **纯迭代开发。**
- **纯顺序开发。**
- **无结构。**
- **无创造性。**
- **优化过早。**
- **过度工程化。**

软件测试

> **Important**
>
> 本部分更详细的可参考fcr老师主讲的**软件测试**课程

软件测试的重要性：

- 历史案例（千年虫、辐射机器、电话服务故障等），软件测试极其重要，应当避免软件缺陷带来的严重后果。
- 软件测试的目的是验证软件是否正确（Verification）以及是否满足用户需求（Validation）。

软件测试的分类：

- **静态分析技术**：基于代码、文档等进行分析，例如**代码审查**、软件走查、算法分析等。
  - 需求、设计以及开发阶段
- **动态测试技术**：通过执行代码来发现缺陷，例如单元测试、集成测试、系统测试等。

软件测试技术：

- **测试用例选择**：根据测试目标选择合适的测试用例，以最小成本发现最多缺陷。
  - 桩与驱动
    - 桩程序：被测试部件的交互环境，通常直接返回固定数据或者按照固定规则返回数据。
    - 驱动程序：被测试部件的执行环境，驱动和监测被测试用例的过程，判定测试用例的执行结果。
    - 换言之启动驱动程序开始测试，由桩程序提供模拟数据，故桩程序位于下侧。
  - 单元测试、集成测试（包括自底向上以及自顶向下）：更加关注技术上的正确性，重在发现设计缺陷和代码缺陷。
  - 系统测试：功能测试、非功能测试……
    - 系统测试关注整个系统的行为。
- **随机测试**：基于测试人员经验选择测试用例。
- **黑盒测试方法**：**不关注**代码内部结构，例如等价类划分、边界值分析、决策表、状态转换等。
- **白盒测试方法**：**关注**代码内部结构，例如语句覆盖、条件覆盖、路径覆盖等。
- **特点测试技术**：针对特定领域或技术的测试，例如面向对象测试、GUI测试、Web测试等。

软件测试活动：

- **测试计划**：确定测试目标、范围、资源、时间等。

- **测试设计**：设计测试用例和测试脚本。
- **测试执行**：执行测试用例并记录结果。
- **测试评估**：分析测试结果并评估软件质量。

软件测试度量：

- **缺陷数据**：记录缺陷的数量、类型、严重程度等信息。
- **测试覆盖率**：评估测试用例对需求的覆盖程度。
- **需求覆盖率、模块覆盖率、代码覆盖率**：分别表示测试用例对需求、模块、代码的覆盖程度。

软件开发过程模型

软件开发过程模型

- 主要描述了软件开发过程中各个阶段的执行方式和组织方式。它涵盖了从需求分析到最终交付产品的整个生命周期。是在生命周期模型的基础上进一步。

主要模型：

- **构建-修复模型**：code and fix
- **瀑布模型**： 将软件开发过程分为需求分析、设计、编码、测试和维护等阶段，每个阶段必须在前一阶段完成后才能开始。
  - 允许活动出现反复和迭代
  - 重点在于要求每个活动的结果必须进行验证
- **快速原型模型**： 通过快速构建原型来验证**需求**，并根据用户反馈进行迭代开发。
- **增量模型**： 将系统分解为多个增量，每个增量都包含部分功能，并逐步开发、交付和集成。
- **演化模型：**也是迭代、并行开发和渐进交付的，但演化模型能够更好地应对需求变更。
- **螺旋模型：**基本思想为尽早解决较高风险，是风险驱动的。
- **敏捷模型**： 强调灵活性、快速响应和协作，通过短周期的迭代开发来交付产品。
- RUP

模型选择：

- **项目规模和复杂性**： 大型复杂项目可能需要更严格的过程模型，而小型简单项目可以选择更灵活的模型。
- **需求稳定性**： 如果需求不稳定，则选择敏捷模型或原型模型可能更合适。
- **团队经验和技能**： 团队的经验和技能水平也会影响模型的选择。

统一软件开发过程 (RUP)：

- 一种迭代增量模型，将软件开发过程分为多个周期，每个周期都包含需求分析、设计、编码和测试等阶段。
- 强调用例驱动和迭代开发。
- 使用 UML 作为建模语言。

敏捷方法：

- 一系列轻量级软件开发方法，强调灵活性、快速响应和协作。
- 主要方法包括：
  - **极限编程 (XP)**： 强调测试驱动开发、持续集成和结对编程。

- **Scrum**： 强调短周期的迭代开发和团队自组织。
- **特征驱动开发 (FDD)**： 强调基于特征的迭代开发和设计。
- **自适应软件开发 (ASD)**： 强调灵活性和适应性。
- **动态系统开发方法 (DSDM)**： 强调快速开发和用户参与。

❗**软件开发过程模型的选择取决于项目的具体需求和团队的情况。** 选择合适的模型可以提高开发效率，并确保项目成功。