



Licus7 / FlowerC



&lt;&gt; Code

Issues

Pull requests

Actions

Projects

Wiki

Security

FlowerC / Python学习网站第一版 / FlowerC文档 / Study guide(FlowerC)

/ study guide.md



Licus7 study guide has some changes...

27203f0 · 6 minutes ago



1728 lines (1331 loc) · 50.1 KB

Preview

Code

Blame

Raw



# Study Guide

2025 Flower C

## 第一章：软件问题 - 为什么软件开发这么难？

### 1.1 本章学习目标

学完这一章，你应该能够：

- 理解软件开发面临的主要挑战
- 分析成本、进度、质量之间的平衡关系
- 认识到软件规模和变更带来的复杂性
- 为后续学习软件工程方法打下基础

### 1.2 核心概念详解

#### 1.2.1 软件开发的三大难题

成本问题 - 钱都花到哪里去了？

成本不仅仅是写代码的时间，还包括：

- **直接成本**：开发时间、测试时间、部署时间
- **间接成本**：学习新技术的时间、沟通协调的时间
- **隐藏成本**：调试bug的时间、修改需求的时间、文档编写的时间

比如我们项目计划15周，但实际可能遇到技术难题需要额外时间，这就是成本估算的困难。

**进度问题 - 为什么总是延期？** 我们小组有时遇到这种情况：

- **乐观估计：** 开始觉得2天能完成的功能，实际要5天
- **意外情况：** 组员生病、考试冲突、技术难题

就像我们的项目计划，虽然分了15周，但中间必须留出缓冲时间。

**质量问题 - 什么是好软件？** 好的软件不仅要能运行，还要：

- **正确性：** 功能符合需求（比如我们的代码检测要准确）
- **可靠性：** 稳定不崩溃（网站不能经常打不开）
- **易用性：** 用户操作方便（界面要直观）
- **可维护性：** 后续修改容易（代码要清晰，模块化留有余地）

### 1.2.2 规模和变更的挑战

**规模的影响** 小型项目（比如个人作业）和大型项目（比如我们的网站）完全不同：

- **沟通成本：** 多个人开发需要开会交流进度想法等，一个人开发就不需要
- **复杂度：** 功能越多，代码越复杂，可能的bug越多
- **协调难度：** 每个人负责的模块要能完美对接

**变更的必然性** 我们在开发过程中遇到：

- 教授提出新要求
- 技术方案需要调整
- 用户体验需要优化 变更不是坏事，但需要管理。

## 1.3 学习资源推荐

### 视频资源

- [Software Engineering: Crash Course](#) - 16分钟快速了解软件工程，讲解生动有趣
- [为什么软件项目会失败](#) - 分析真实项目失败案例，很有启发性
- [软件开发生命周期](#) - 9分钟讲清楚整个开发过程

### 实用网站

- [GeeksforGeeks软件工程专题](#) - 概念解释很清晰，有代码示例
- [TutorialsPoint教程](#) - 适合初学者，循序渐进
- [软件工程知识体系SWEBOK](#) - 权威参考

### 重点阅读

- 《人月神话》 - 软件工程的经典，讲了很多真实项目的经验教训
- 教材第一章 - 一定要精读，掌握基本概念

## 1.4 习题

### 选择题

软件危机的主要表现不包括以下哪项？ A. 软件开发成本超出预算 B. 软件产品质量无法保证 C. 软件开发进度难以控制 D. 软件用户数量增长过快

根据教材，软件维护成本通常占整个生命周期成本的： A. 10%-20% B. 30%-40% C. 50%-60% D. 70%-80%

"软件规模"这一概念主要描述的是： A. 软件代码的行数多少 B. 软件项目的复杂程度和大小 C. 软件开发团队的人数规模 D. 软件安装包的大小

软件需求变更难以控制的主要原因是： A. 开发人员技术水平不足 B. 用户需求本身具有模糊性和变化性 C. 项目管理工具不够先进 D. 软件开发方法选择不当

教材中提到的"软件=程序+数据+文档"公式强调了： A. 软件不仅仅是程序代码 B. 数据比程序更重要 C. 文档是软件的核心 D. 三者同等重要

### 判断题

- ( ) 软件危机只存在于大型软件项目中。
- ( ) 软件维护成本可以通过良好的设计来降低。
- ( ) 软件规模的增长会导致开发成本线性增长。
- ( ) 需求变更是软件项目的异常情况，应该尽量避免。

### 简答题

简述软件危机的主要表现及其产生原因。

为什么软件维护成本通常很高？如何降低维护成本？

软件规模对软件开发有哪些影响？

## 答案

选择题：

D - 软件用户数量增长过快（这不是危机表现）

D - 70%-80%（教材强调维护成本很高）

B - 软件项目的复杂程度和大小

B - 用户需求本身具有模糊性和变化性

A - 软件不仅仅是程序代码

判断题：

×（软件危机存在于各种规模的项目中）

√（良好设计可以降低维护成本）

×（规模增长导致成本非线性增长）

×（需求变更是正常现象）

简答题：

软件危机表现：成本超支、进度延误、质量低下、维护困难。原因：软件复杂性、需求变化、管理不当等。

维护成本高原因：软件规模大、结构复杂、文档不全。降低方法：提高软件质量、完善文档、采用良好架构。

规模影响：开发成本增加、进度难以控制、质量保证困难、团队协作复杂。

## 1.5 案例分析：我们的Python学习网站项目

**质量要求的平衡：**既要功能完整又要界面美观，在时间不充裕的情况下我们不得不：

- 优先保证核心功能（学习系统）
- 简化游戏化功能（精灵收集）
- 在质量和进度之间找到平衡点

### 实用建议

给同学们的建议：

1. **成本估算：**在实际时间基础上增加30%缓冲
2. **进度管理：**使用GitHub或gitee，每周检查进度
3. **质量保证：**从开始就注重代码质量和测试
4. **变更处理：**GitHub上有详细的变更记录表

## 第二章：软件过程 - 如何有序地开发软件？

---

### 2.1 本章学习目标

- 掌握各种软件开发过程的优缺点

- 理解敏捷开发的核心思想
- 学会为项目选择合适的过程模型
- 能够在实际项目中应用过程方法

## 2.2 核心概念详解

### 2.2.1 传统过程模型比较

**瀑布模型 - 按部就班** 就像建房子一样，必须按顺序来：

- 优点：计划明确，文档完整，适合需求固定的项目
- 缺点：不能适应变化，后期修改成本高
- 适合：政府项目、银行系统等需求稳定的项目

**增量模型 - 分批交付** 我们项目就用了这种方法：

- 先做核心功能（用户系统+学习功能）
- 再添加增强功能（游戏化系统）
- 优点：早期看到成果，风险分散
- 缺点：需要良好的架构设计

**螺旋模型 - 重视风险** 像蜗牛壳一样一圈圈扩大：

- 每轮都进行风险分析
- 适合大型复杂项目

### 2.2.2 敏捷开发实战理解

**敏捷的核心思想**：快速响应变化，持续交付价值\*\*

**Scrum框架的实用解读** 我们小组实际应用的简化版Scrum：

**产品待办列表** - 把所有要做的功能列出来，按优先级排序

1. 用户注册登录 [高]
2. 章节内容展示 [高]
3. 选择题测试 [中]
4. 代码检测功能 [高]
5. 精灵收集系统 [低]



## 2.3 学习资源推荐

### 推荐视频

- [瀑布vs敏捷直观对比](#) - 用动画展示区别，很直观

- [Scrum实战演示](#) - 真实团队如何开Scrum会议
- [敏捷开发误区](#) - 避免把敏捷用成混乱

## 🔧 实用工具

- **Trello** - 免费的看板工具，适合学生项目
- **GitHub Projects** - 和代码仓库集成，很方便
- **微信群** - 我们小组的交流工具

## 📖 延伸阅读

- **教材第二章** - 重点理解各种模型的适用场景
- **Scrum指南** - 只有十几页，但很精华

## 2.4 习题

### 选择题

瀑布模型最适合以下哪种情况？ A. 需求不明确的项目 B. 需要快速开发原型的项目 C. 需求明确且变化较少的项目 D. 高风险的大型项目

敏捷开发方法的核心思想是： A. 详细的文档和严格的流程 B. 快速响应变化和持续交付价值 C. 完整的架构设计 D. 严格的阶段评审

螺旋模型区别于其他模型的特点是： A. 线性开发过程 B. 强调风险分析 C. 不需要用户参与 D. 文档驱动

在增量模型中，系统是： A. 一次性完整交付 B. 分批次逐步交付 C. 永远处于测试状态 D. 不需要集成测试

Scrum框架中的"产品待办列表"包含： A. 已完成的功能列表 B. 所有已知的需求项 C. 技术难题列表 D. 项目风险列表

### 判断题

- ( ) 瀑布模型允许在开发后期进行需求变更。
- ( ) 敏捷开发方法适用于所有类型的软件项目。
- ( ) 螺旋模型在每个迭代中都进行风险分析。
- ( ) 增量开发可以降低项目风险。

### 简答题

比较瀑布模型和敏捷开发方法的优缺点。

为什么说螺旋模型特别适合高风险项目？

在Scrum框架中，冲刺（Sprint）的主要活动有哪些？

## 答案

选择题：

- C - 需求明确且变化较少的项目
- B - 快速响应变化和持续交付价值
- B - 强调风险分析
- B - 分批次逐步交付
- B - 所有已知的需求项

判断题：

- ×（瀑布模型难以适应后期变更）
- ×（不是所有项目都适合敏捷）
- √（风险分析是螺旋模型的核心）
- √（早期交付可以降低风险）

简答题：

瀑布模型优点：过程明确、文档完整。缺点：难以适应变化。敏捷优点：灵活、快速响应变化。缺点：文档相对简单。

螺旋模型适合高风险项目原因：每个迭代都进行风险分析，可以及时调整方向，降低整体风险。

Scrum冲刺活动：冲刺规划、每日站会、开发工作、冲刺评审、冲刺回顾。

## 2.5 案例分析：我们项目的开发过程

### 真实过程分享

我们为什么选择迭代增量模型？

- 需求相对明确
- 但细节会调整（根据开发情况）
- 需要早期看到成果（给教授演示）
- 团队规模小（4人），适合敏捷

实际执行中的调整 第一轮迭代（第1-8周）：

- 计划：完成用户系统+基础框架
- 实际：用户系统完成，但代码编辑器遇到技术难题
- 调整：先跳过代码编辑器，用简单文本框代替

#### 经验教训：

- 计划要灵活：留出20%时间应对意外
- 沟通要频繁：各职责组员间及时同步进度与需要的配合

#### 给同学的建议

1. 从小开始：先尝试2周的迭代周期
2. 重视回顾：每次迭代后总结改进点
3. 工具适宜：选择团队都能熟练使用的工具
4. 持续改进：过程模型不是一成不变的

## 第三章：软件计划 - 详细学习指南

---

### 3.1 本章概述

软件计划是软件工程过程中的关键环节，是项目管理的基础。计划阶段的主要任务包括：确定软件范围、资源需求、成本估算、进度安排、风险分析等。一个好的软件计划是项目成功的保证。

### 3.2 核心概念详解

#### 3.2.1 软件范围定义

##### 目标确定

软件范围的定义是计划的基础，必须明确：

- 软件的功能和性能要求
- 系统的接口特性
- 可靠性要求
- 交付成果和验收标准

##### 资源确定

资源包括三大类：

1. 人力资源：开发人员、测试人员、管理人员等
2. 环境资源：硬件设备、软件工具、开发环境



### 3. 时间资源：项目总体时间框架和里程碑

#### 3.2.2 成本估算方法

##### 自顶向下估算方法

这种方法基于历史项目的总体数据，通过类比的方式进行估算。

##### 特点：

- **适用场景：**项目初期，需求不够明确时
- **优点：**估算快速，不需要详细设计
- **缺点：**精度较低，依赖估算者的经验
- **典型技术：**专家判定法、Delphi技术

##### 自底向上估算方法

将项目分解为各个模块，分别估算每个模块的成本，然后汇总。

##### 特点：

- **适用场景：**需求明确，设计详细时
- **优点：**估算精度高，基于详细分析
- **缺点：**耗时较长，需要完整的工作分解结构
- **典型技术：**代码行技术、功能点技术

#### 3.2.3 具体的估算技术

##### 代码行技术（LOC）

这是一种简单的量化估算方法：

$$\text{工作量} = \text{代码行数} \div \text{生产率}$$

其中生产率基于历史数据，如：500行/人月

##### 优缺点分析：

- **优点：**直观、容易理解
- **缺点：**依赖编程语言、对设计质量敏感

##### 功能点技术（FP）

Albrecht提出，从用户视角度量软件规模：

##### 计算步骤：

###### 1. 计算未调整功能点（UFP）

- 外部输入 (EI)、外部输出 (EO)、外部查询 (EQ)
- 内部逻辑文件 (ILF)、外部接口文件 (EIF)

## 2. 计算值调整因子 (VAF)

- 考虑14个系统特性
- 每个特性评分0-5分

## 3. 计算调整功能点 (AFP)

$$AFP = UFP \times (0.65 + 0.01 \times \sum Fi)$$

## COCOMO模型

由Boehm提出，分为三个层次：

**基本COCOMO 静态单变量模型：**

$$\text{工作量} = a \times (\text{规模})^b$$

$$\text{时间} = c \times (\text{工作量})^d$$

**中间COCOMO** 在基本模型基础上，考虑四类成本驱动属性：

1. **产品属性：**可靠性、数据库规模、复杂性
2. **计算机属性：**执行时间约束、存储约束等
3. **人员属性：**分析员能力、应用经验等
4. **项目属性：**现代编程实践、软件工具使用等

**详细COCOMO** 包括中间模型的所有特性，再加上对软件工程过程中各个步骤的影响分析。

## 3.2.4 进度安排

### 甘特图

**特点：**

- 水平条形图显示任务时间安排
- 直观展示任务开始和结束时间
- 便于进度跟踪和控制

**局限性：** 不能很好显示任务之间的依赖关系

### 工程网络技术

#### 关键路径法 (CPM)

- 确定项目中最长的路径
- 计算最早开始时间（ES）、最晚开始时间（LS）
- 计算时差（浮动时间）
- 识别关键任务

计划评审技术（PERT）

- 考虑时间估计的不确定性
- 使用三种时间估计：乐观时间、最可能时间、悲观时间
- 计算期望时间和方差

3.2.5 风险分析

风险识别

识别可能影响项目的各种风险因素：

风险类型	具体表现
技术风险	技术不成熟、技术难度大
管理风险	资源不足、进度安排不当
商业风险	市场变化、竞争对手行动

风险估计

定量分析风险：

- 发生概率：风险事件发生的可能性
- 影响程度：风险发生后的损失大小
- 风险暴露量 = 概率 × 影响

风险应对策略

策略类型	实施方法	适用场景
风险规避	改变计划消除风险	高风险、高影响
风险转移	将风险转嫁给第三方	可外包的风险
风险缓解	采取措施减轻风险影响	大多数风险
风险接受	准备应急计划接受风险	低概率、低影响

3.3 软件计划的主要内容

### 3.3.1 项目目标与范围

- 明确软件要解决的问题
- 定义软件的功能边界
- 确定交付物和验收标准

### 3.3.2 资源计划

人力资源计划：

- 项目经理：1人（全程）
- 开发工程师：3人（编码阶段）
- 测试工程师：1人（测试阶段）
- 文档工程师：1人（文档阶段）



环境资源计划：

- 开发工具：IDE、版本控制系统
- 测试环境：测试服务器、测试数据
- 生产环境：服务器配置、网络要求

### 3.3.3 成本估算报告

应包括：

- 估算方法说明
- 估算结果明细
- 估算假设和约束
- 风险储备金计算

### 3.3.4 进度计划表

使用甘特图或网络图展示：

- 任务分解结构
- 任务依赖关系
- 关键路径分析
- 里程碑设置

### 3.3.5 质量计划

质量目标：

- 功能完整性
- 性能指标要求

- 可靠性目标
- 可用性标准

#### 质量保证措施：

- 评审计划
- 测试策略
- 质量度量

### 3.3.6 风险应对计划

#### 风险登记册内容：

- 风险描述
- 概率影响评估
- 应对策略
- 责任人分配

## 3.4 软件计划的制定过程

### 3.4.1 信息收集阶段

- 收集用户需求
- 分析技术可行性
- 调研市场环境
- 评估组织能力

### 3.4.2 方案设计阶段

- 确定技术方案
- 设计系统架构
- 制定开发策略
- 选择过程模型

### 3.4.3 计划编制阶段

- 编制详细计划
- 进行风险评估
- 制定应急计划
- 准备计划文档

### 3.4.4 评审确认阶段

- 内部评审

- 用户确认
- 管理层批准
- 基线化建立

## 3.5 软件计划的管理与控制

### 3.5.1 计划跟踪机制

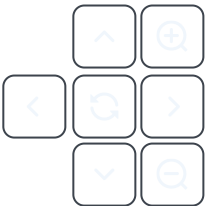
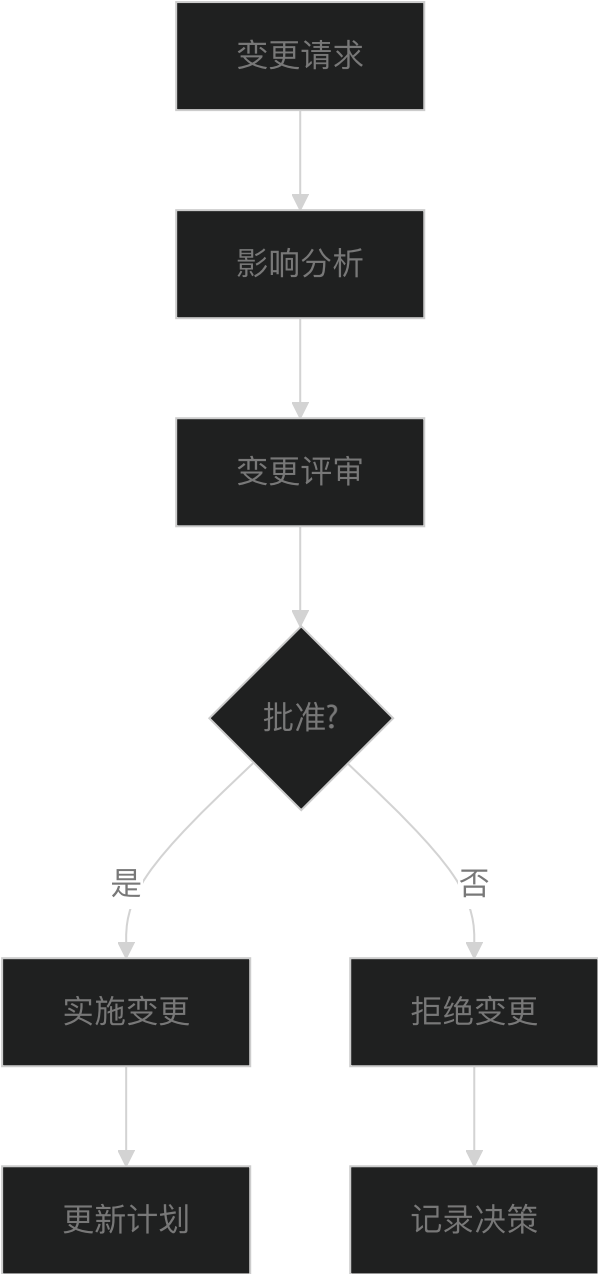
进度跟踪：

- 每周进度报告
- 里程碑检查
- 偏差分析

成本控制：

- 实际成本跟踪
- 成本绩效分析
- 预算调整

### 3.5.2 变更管理流程



3.5.3 风险监控活动

- 定期风险评审
- 新风险识别
- 应对措施跟踪
- 风险状态报告

3.6 案例分析：Python学习网站项目计划

3.6.1 项目范围定义

**项目目标：** 开发一个面向Python初学者的互动学习平台，包含完整的教学、练习、评估功能。

**范围边界：**

- 包含：12章教学内容、代码练习系统、用户进度跟踪
- 不包含：移动端APP、社交功能、在线编译器

3.6.2 成本估算实践

采用自底向上估算

**模块分解：**

模块名称	工作量(人天)	说明
用户管理	12	注册、登录、权限管理
内容管理	24	章节内容、进度跟踪
练习系统	20	题目管理、自动评分
游戏化功能	16	金币、成就系统
系统集成	8	模块整合、测试
总计	80人天	

使用COCOMO模型验证

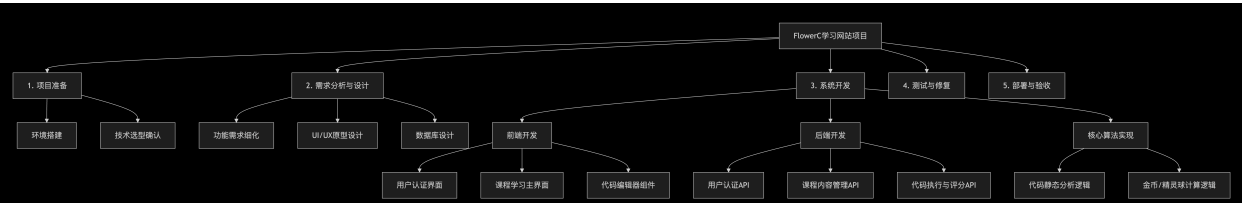
基于中间COCOMO模型，考虑调整因子：

- 产品复杂性：高 (1.15)
- 团队经验：中等 (1.00)
- 开发环境：良好 (0.90)

调整后工作量 = 80×1.15×1.00×0.90 = 82.8人天

3.6.3 进度安排设计

工作分解结构（WBS）



关键路径分析



**关键路径：** 需求分析 → 系统设计 → 内容系统开发 → 系统测试

**总工期：** 15周 **关键路径时长：** 2 + 2 + 3 + 2 = 9周

3.6.4 风险管理计划

风险识别与评估

风险项	概率	影响	风险值	等级
技术难题	0.7	0.8	0.56	高
进度延误	0.6	0.7	0.42	中
需求变更	0.5	0.6	0.30	中
人员变动	0.3	0.8	0.24	低

风险应对措施

技术风险应对：

- 提前技术调研和原型验证
- 准备技术备选方案
- 安排技术培训

进度风险应对：

- 制定详细的WBS
- 设置里程碑检查点
- 预留10%的缓冲时间

3.7 软件计划的质量保证

3.7.1 计划评审要点

完整性检查：

- 是否覆盖所有重要方面
- 是否有遗漏的任务
- 资源分配是否合理

一致性检查：

- 成本、进度、资源是否匹配
- 任务依赖关系是否正确
- 风险应对措施是否可行

### 3.7.2 计划优化方法

#### 资源平衡：

- 解决资源冲突
- 优化资源利用率
- 平衡工作负荷

#### 时间优化：

- 关键路径压缩
- 并行任务安排
- 浮动时间利用

## 3.8 习题

### 一、选择题

#### 单项选择题

1. 软件计划阶段的主要任务不包括： A. 确定项目范围 B. 编写程序代码 C. 估算项目成本 D. 制定进度计划
2. 自顶向下估算方法的主要特点是： A. 先估算模块成本再汇总 B. 从系统整体出发进行估算 C. 需要详细的设计文档 D. 估算精度很高
3. 下列哪项不是功能点技术考虑的功能组件： A. 外部输入 B. 内部逻辑文件 C. 代码行数 D. 外部查询
4. 在COCOMO模型中，考虑产品、硬件、人员、项目属性的是： A. 基本COCOMO B. 中间COCOMO C. 详细COCOMO D. 扩展COCOMO
5. 关键路径法的主要作用是： A. 计算项目成本 B. 确定最短项目工期 C. 分配人力资源 D. 评估代码质量
6. 风险暴露量的计算公式是： A. 风险概率 + 风险影响 B. 风险概率 × 风险影响 C. 风险概率 / 风险影响 D. 风险影响 - 风险概率
7. 甘特图的主要优点是： A. 能显示任务依赖关系 B. 能计算关键路径 C. 直观显示时间进度 D. 能自动优化资源分配
8. 自底向上估算方法最适合在哪个阶段使用： A. 项目立项阶段 B. 需求分析阶段 C. 详细设计阶段 D. 测试阶段

#### 多项选择题

9. 软件计划包括的主要内容有： A. 成本估算 B. 进度安排 C. 风险分析 D. 代码编写 E. 资源分配
10. 下列属于风险应对策略的有： A. 风险规避 B. 风险转移 C. 风险忽略 D. 风险缓解 E. 风险接受
11. COCOMO模型的三个层次包括： A. 基本COCOMO B. 中间COCOMO C. 详细COCOMO D. 简化COCOMO E. 扩展COCOMO
12. 工程网络技术包括： A. 甘特图法 B. 关键路径法（CPM） C. 计划评审技术（PERT） D. 代码行技术 E. 功能点技术

## 二、判断题

1. ( ) 软件计划只需要在项目开始时制定一次。
2. ( ) 自底向上估算方法适用于项目初期阶段。
3. ( ) 代码行技术需要依赖历史项目数据。
4. ( ) 关键路径上的任务可以有时差。
5. ( ) 甘特图能够清晰显示任务之间的依赖关系。
6. ( ) 风险缓解是消除风险的策略。
7. ( ) 功能点技术是从用户角度度量软件规模的方法。
8. ( ) 软件计划的质量直接影响项目成功率。

## 三、简答题

1. 比较自顶向下和自底向上两种估算方法的优缺点及适用场景。
2. 风险管理的四个主要步骤是什么？每个步骤的重点工作有哪些？
3. 为什么说软件计划是一个迭代的过程？在项目中应该如何进行计划的调整？
4. 结合Python学习网站项目，说明如何制定一个完整的软件计划。

## 四、案例分析题

某高校要开发一个"在线考试系统"，请根据以下需求制定软件计划：

系统需求：

- 支持5000名学生同时在线考试
- 包含题库管理、试卷生成、在线考试、自动评分功能
- 6个月内完成开发
- 开发团队8人

请回答：

1. 采用什么估算方法比较合适？为什么？
2. 如何制定进度计划？使用什么工具？
3. 识别至少3个主要风险，并制定应对策略。
4. 如何保证软件计划的质量？

## 答案

### 一、选择题答案

#### 单项选择题答案

1. **B** - 编写程序代码（属于实现阶段）
2. **B** - 从系统整体出发进行估算
3. **C** - 代码行数（功能点技术的五个组件是：外部输入、外部输出、外部查询、内部逻辑文件、外部接口文件）
4. **B** - 中间COCOMO
5. **B** - 确定最短项目工期
6. **B** - 风险概率 × 风险影响
7. **C** - 直观显示时间进度
8. **C** - 详细设计阶段

#### 多项选择题答案

9. **ABCE** - 代码编写不属于计划内容
10. **ABDE** - 风险忽略不是正式应对策略
11. **ABC** - COCOMO模型的三个标准层次
12. **BC** - 关键路径法和计划评审技术属于工程网络技术

### 二、判断题答案

1. **×** - 软件计划需要随项目进展不断调整
2. **×** - 自底向上需要详细设计，更适合项目中后期
3. **√** - 代码行技术需要历史生产率数据
4. **×** - 关键路径上的任务没有时差
5. **×** - 甘特图不能很好显示任务依赖关系
6. **×** - 风险缓解是减轻风险影响，不是消除
7. **√** - 功能点技术从用户角度度量
8. **√** - 好的计划是项目成功的基础

### 三、简答题答案

#### 1. 估算方法比较

### 自顶向下估算：

- 优点：快速、整体性强、适合初期
- 缺点：精度低、依赖经验
- 适用场景：项目立项、可行性分析阶段

### 自底向上估算：

- 优点：精度高、基于详细分析
- 缺点：耗时、需要完整设计
- 适用场景：详细设计完成后

## 2. 风险管理步骤

### 四个步骤：

1. 风险识别：发现可能的风险因素
2. 风险估计：评估概率和影响程度
3. 风险评价：确定风险优先级排序
4. 风险应对：制定应对策略和计划

## 3. 计划的迭代性

### 迭代原因：

- 需求会变化和明确
- 技术方案可能调整
- 风险情况发生变化
- 资源可用性变化

### 调整方法：

- 定期评审计划（每月/每阶段）
- 根据实际情况更新估算
- 调整进度和资源分配
- 更新风险应对计划

## 4. Python项目计划实例

### 计划制定步骤：

1. 范围定义：明确12章教学内容边界
2. 成本估算：采用自底向上，模块分解估算
3. 进度安排：15周总工期，关键路径分析
4. 资源分配：4人团队，按技能分配任务

## 5. 风险管理：识别技术、进度、资源风险

### 四、案例分析题答案

#### 1. 估算方法选择

选择中间COCOMO模型，因为：

- 项目规模中等，需求相对明确
- 有历史数据可以参考
- 需要较高的估算精度

#### 2. 进度计划制定

工具选择：

- 甘特图：直观显示时间安排
- 关键路径法：识别关键任务
- 网络计划技术：优化资源分配

制定步骤：

1. 工作分解结构（WBS）
2. 任务工期估算
3. 依赖关系分析
4. 关键路径计算

#### 3. 风险识别与应对

主要风险：

1. 性能风险（5000人并发）
  - 应对：压力测试、性能优化、负载均衡
2. 安全风险（考试系统安全性）
  - 应对：安全审计、加密传输、防作弊机制
3. 进度风险（6个月工期紧张）
  - 应对：增量开发、优先级排序、预留缓冲

#### 4. 计划质量保证

质量保证措施：

- 多方评审：技术评审、管理评审、用户评审

- **经验借鉴**：参考类似项目历史数据
- **持续改进**：根据实际情况调整优化
- **文档规范**：完整的计划文档和变更记录

## 第四章：需求分析 - 如何搞清楚用户想要什么？

---

### 4.1 本章学习目标

学完这一章，你应该能够：

- 掌握需求获取的各种技术方法
- 学会编写清晰的需求规格说明
- 理解需求验证和管理的重要性
- 在我们的Python学习网站项目中实践需求工程

### 4.2 核心概念详解

#### 4.2.1 需求工程过程

**需求获取 - 学会"听"的艺术** 就像我们项目开始时，要搞清楚用户到底想要什么：

- **访谈技巧**：和组员聊天，了解他们的真实需求
- **观察法**：看教授发的资料别人如何制作，内化于心
- **原型演示**：先做个简单版本，收集反馈再改进

**需求分析 - 从混乱到有序** 我们面对一堆需求时要做：

- **分类整理**：哪些是必须的，哪些是锦上添花
- **优先级排序**：先做核心功能，再做高级功能
- **解决冲突**：当不同用户需求矛盾时，要找到平衡点

**需求规约 - 写清楚的"说明书"** 就像我们写需求文档：

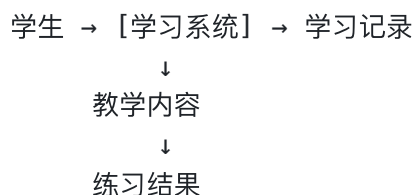
- 要用准确的语言描述功能
- 让开发和测试都能看懂
- 避免模糊的说法，比如"系统要快"要说"页面加载不超过3秒"（可验证性！）

**需求验证 - 确认没理解错** 做完需求文档后要：

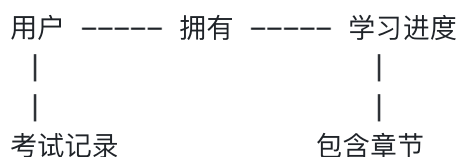
- 找教授评审
- 用原型演示确认
- 提前想好测试用例

## 4.2.2 需求建模技术

**数据流图 - 看清信息的流动** 就像我们的学习网站：



**实体关系图 - 理清数据关系** 我们的用户和进度关系：



**用例图 - 描述系统功能** 比如"学生参加章节测试"就是一个典型用例

## 4.2.3 需求管理

**需求跟踪** 我们要建立需求与代码的对应关系，确保每个需求都实现了

**变更控制** 当教授提出新需求时，我们要：

1. 记录变更请求
2. 分析对进度的影响
3. 团队讨论决定
4. 更新相关文档

## 4.3 学习资源推荐

### 视频资源

- [需求工程概述](#) - 系统讲解需求分析全过程
- [用户故事编写技巧](#) - 实战技巧，很实用
- [用例图绘制教程](#) - 手把手教学，适合初学者

### 实用网站

- [Draw.io在线绘图](#) - 免费画数据流图、用例图
- [需求管理工具介绍](#) - 了解专业需求管理
- [敏捷需求实践](#) - 用户故事编写指南

### 重点阅读



- 教材第三章 - 精读需求分析方法和工具
- 《掌握需求过程》 - 需求工程的实用指南
- IEEE需求标准 - 了解行业标准格式

## 4.4 习题

### 选择题

1. 需求工程的首要任务是： A. 编写详细的需求文档 B. 获取和理解用户需求 C. 设计系统架构 D. 制定项目计划
2. 功能需求主要描述的是： A. 系统的性能指标 B. 系统必须完成的功能 C. 系统的界面设计 D. 系统的安全要求
3. 非功能需求不包括： A. 系统响应时间要求 B. 用户界面美观要求 C. 数据存储容量要求 D. 用户权限管理功能
4. 用例图主要用于描述： A. 系统的数据结构 B. 系统与用户的交互 C. 系统的处理流程 D. 系统的网络拓扑
5. 需求验证的主要目的是： A. 确认需求符合用户真实需要 B. 估算项目成本 C. 制定开发计划 D. 设计系统架构

### 判断题

1. ( ) 需求工程只在项目开始阶段进行。
2. ( ) 功能需求比非功能需求更重要。
3. ( ) 用例图可以描述系统的功能需求。
4. ( ) 需求变更应该完全避免。

### 简答题

1. 简述需求工程包括哪些主要活动？
2. 功能需求和非功能需求有什么区别？各举两个例子。
3. 为什么需要进行需求验证？有哪些常用的验证方法？

## 4.5 答案

### 选择题答案：

1. B - 获取和理解用户需求（这是需求工程的基础）
2. B - 系统必须完成的功能（功能需求描述"做什么"）
3. D - 用户权限管理功能（这是功能需求）
4. B - 系统与用户的交互（用例图展示系统功能边界）
5. A - 确认需求符合用户真实需要（避免开发偏差）

## 判断题答案：

1. × - 需求工程贯穿项目全过程，需要持续调整
2. × - 两者都重要，非功能需求影响用户体验
3. √ - 用例图是描述功能需求的有效工具
4. × - 需求变更需要管理，而不是完全避免

## 简答题答案：

### 1. 需求工程主要活动：

- 需求获取：收集用户需求
- 需求分析：整理和优先级排序
- 需求规约：编写需求文档
- 需求验证：确认需求正确性
- 需求管理：跟踪和变更控制

### 2. 功能需求 vs 非功能需求：

- 功能需求：系统做什么（如用户登录、数据查询）
- 非功能需求：系统做得怎么样（如性能、安全、可用性）
- 例子：功能需求是"用户能搜索产品"，非功能需求是"搜索响应时间小于2秒"

### 3. 需求验证的必要性和方法：

- 必要性：确保需求正确、完整、一致，避免开发偏差
- 验证方法：需求评审、原型验证、测试用例设计、用户确认

## 4.6 案例分析：我们的Python学习网站需求分析

### 真实经验分享

**需求获取的挑战：** 开始我们以为需求很明确，实际发现：

- 组员间说的需求和彼此真正想要的有差异
- 技术诸多限制影响需求实现

### 我们的解决方案：

1. 多次确认：和组员开会，用示例原型确认理解
2. 优先级排序：明确什么必须做，什么可以简化

**需求文档的演变：** 第一版需求文档很简略，后来发现不够用：

- 增加了详细的用例描述
- 补充了非功能需求（可用性）

- 添加了界面原型截图
- 明确了验收标准

## 实用工具和模板

示例用的需求跟踪表：

需求ID	需求描述	优先级	状态	负责人	验收标准
REQ-001	用户注册功能	Must	完成	张三	能成功创建账户，用户名唯一性检查
REQ-002	代码检测功能	Must	开发中	李四	准确识别语法错误，响应<5秒

变更请求记录：

变更ID	提出人	变更内容	影响分析	决策	状态
CR-001	PM	增加游戏性玩法	增加一轮开发	接受	已完成

## 给同学的建议

1. 尽早开始需求分析：越早发现问题，修改成本越低
2. 多方求证：从不同角度理解需求，避免片面
3. 文档化一切：好记性不如烂笔头，记录所有需求决策
4. 保持沟通：需求不是一次性的，要持续与用户沟通

# 第五章：软件体系结构 - 系统的"骨架"设计

## 5.1 本章学习目标

学完这一章，你应该能够：

- 理解软件体系结构的重要作用
- 掌握不同的体系结构视图
- 识别构件和连接件的概念
- 应用常见的体系结构模式
- 为我们的Python学习网站选择合适的架构

## 5.2 核心概念详解

### 5.2.1 软件体系结构的作用

什么是软件体系结构？就像建房子要先画结构图一样，软件体系结构就是软件的"骨架"：

- 宏观规划：决定系统如何组织
- 沟通工具：让团队成员对系统有共同理解
- 质量基础：好的架构决定软件的可维护性、可扩展性

在我们项目中的作用：

- 帮助我们决定用分层架构
- 让4个人能分工合作不冲突
- 保证网站能方便地添加新功能

### 5.2.2 体系结构视图

不同的"视角"看系统：

- 构件和连接件视图：系统由哪些部件组成，怎么连接
- 模块视图：代码如何组织成模块
- 分配视图：软件如何部署到硬件

我们的项目实例：

构件视图：用户模块 + 学习模块 + 游戏模块

连接方式：通过JavaScript函数调用连接

部署视图：全部部署在用户浏览器中



### 5.2.3 构件和连接件视图

构件 - 系统的"积木块" 就像我们项目的各个模块：

```
// 构件例子
class UserManager {} // 用户管理构件
class LearningEngine {} // 学习引擎构件
class GameSystem {} // 游戏系统构件
```



每个构件都有：

- 接口：提供什么功能，需要什么输入
- 功能：具体做什么事情
- 状态：保存什么数据

连接件 - 构件间的"胶水" 在我们的项目中包括：

- **函数调用**：一个模块调用另一个模块的函数
- **事件传递**：用户登录时触发其他模块更新
- **数据共享**：通过IndexedDB共享数据

#### 5.2.4 体系结构模式

**管道-过滤器模式** 像流水线一样处理数据：

原始代码 → [语法检查] → [执行测试] → [结果分析] → 最终反馈



**适合**：数据处理管道

**不适合**：需要共享状态的系统

**共享数据模式** 多个构件操作同一份数据：

用户界面 → 共享数据 ← 业务逻辑  
                  ↑  
          数据持久化



**适合**：需要数据共享的系统

**不适合**：数据一致性要求高的场景

**客户端-服务器模式** 我们没选这个，因为：

- **优点**：逻辑集中，易于维护
- **缺点**：需要服务器，我们做纯前端项目

**我们选择的模式：分层架构**

表示层 → 业务层 → 数据层



为什么选这个：

- 适合中等规模项目
- 团队成员熟悉
- 便于分工开发

### 5.3 学习资源推荐

#### 视频资源

- [软件架构模式详解](#) - 各种架构模式对比
- [分层架构实战](#) - 分层架构的具体实现
- [架构设计原则](#) - 好的架构设计思路

## 实用网站

- [架构模式目录](#) - 各种架构模式介绍
- [UML图教程](#) - 学习架构图绘制
- [软件架构基础](#) - 架构设计电子书

## 重点阅读

- 教材第五章 - 精读体系结构概念和模式
- 《软件体系结构实践》 - 实战经验分享
- 《企业应用架构模式》 - 经典架构指南

## 5.4 习题

### 选择题

1. 软件体系结构的主要作用不包括： A. 定义系统的总体结构 B. 指导具体的代码编写 C. 促进团队沟通和理解 D. 支持系统质量属性
2. 在构件和连接件视图中，连接件的作用是： A. 实现具体的业务逻辑 B. 在构件之间传递信息 C. 存储系统数据 D. 提供用户界面
3. 管道-过滤器模式最适合： A. 需要共享状态的系统 B. 数据流水线处理系统 C. 实时交互系统 D. 分布式计算系统
4. 我们项目选择分层架构的主要原因是： A. 性能最高 B. 最适合前端项目 C. 团队熟悉且适合项目规模 D. 最容易实现
5. 体系结构视图的主要目的是： A. 替代详细设计 B. 从不同角度描述系统 C. 减少代码量 D. 提高运行速度

### 判断题

1. ( ) 软件体系结构只在项目开始时设计一次。
2. ( ) 构件应该尽可能大，减少构件数量。
3. ( ) 分层架构中，上层可以直接访问所有下层。
4. ( ) 好的体系结构应该能够适应需求变化。

### 简答题

1. 为什么软件体系结构对项目成功很重要？
2. 比较管道-过滤器模式和共享数据模式的适用场景。
3. 在我们的Python学习网站中，如何体现构件和连接件的概念？

## 5.5 答案

### 选择题答案：

1. **B** - 指导具体的代码编写（这是详细设计的任务）
2. **B** - 在构件之间传递信息（连接件的核心作用）
3. **B** - 数据流水线处理系统（管道-过滤器的典型场景）
4. **C** - 团队熟悉且适合项目规模（架构选择的重要考虑）
5. **B** - 从不同角度描述系统（多视图的价值）

### 判断题答案：

1. **×** - 体系结构需要随项目演进调整
2. **×** - 构件应该适中，过大过小都不好
3. **×** - 分层架构应该遵循依赖方向，不能随意跨层访问
4. **√** - 好的架构具有灵活性和适应性

### 简答题答案：

#### 1. 体系结构的重要性：

- 为系统提供整体蓝图
- 影响软件质量属性（性能、可维护性等）
- 指导团队分工协作
- 降低开发风险和成本

#### 2. 架构模式比较：

- **管道-过滤器**：适合数据转换流水线，如代码检测流程
- **共享数据**：适合需要数据共享的系统，如多人编辑文档
- 选择依据：数据处理方式、性能要求、团队技能

#### 3. 我们项目的构件连接件：

- **构件**：UserManager、LearningEngine、GameSystem
- **连接件**：函数调用、事件监听、数据存储接口
- **体现**：模块间通过明确定义的接口协作

## 5.6 案例分析：我们的Python学习网站架构

### 架构设计决策

#### 为什么选择分层架构：

// 分层架构在我们的体现

表示层：HTML/CSS

业务层：userManager, LearningEngine, GameSystem



数据层：IndexedDB操作封装

持久层：浏览器本地存储

## 构件设计考虑：

```
// 每个构件的职责
userManager：只负责用户相关功能
LearningEngine：只负责学习逻辑
GameSystem：只负责游戏化功能
// 构件间通过清晰接口通信，内部实现隐藏
```



## 架构评估

### 我们的架构优点：

- 可维护性：每层职责清晰，修改影响小
- 可测试性：可以分层测试
- 团队协作：4个人可以分层次发

### 可能的改进点：

- 灵活性：严格的层次可能限制某些优化

## 实用建议

1. 先规划后编码：花时间设计好架构，编码会更顺利
2. 考虑变化：设计时要想到需求可能变化
3. 团队统一理解：确保所有人都理解架构设计
4. 文档化决策：记录为什么选择这个架构

# 第六章：设计 - 从架构到详细实现

## 6.1 本章学习目标

学完这一章，你应该能够：

- 掌握耦合、内聚等设计基本概念
- 理解面向功能设计和面向对象设计
- 学会使用UML进行设计建模
- 掌握详细设计的方法
- 在我们的Python学习网站中应用设计原则



## 6.2 核心概念详解

### 6.2.1 设计的基本概念

**耦合 - 模块间的"亲密程度"** 就像我们项目中模块的关系：

- **低耦合**：用户模块修改不影响学习模块
- **高耦合**：改一个地方要改很多地方

**耦合控制：**

```
// 低耦合的例子 - 通过接口交互
class UserManager {
    async login(credentials) {
        // 只通过参数接收数据
        // 返回明确的结果
    }
}

// 高耦合的反例 - 直接操作内部数据
// 其他模块直接修改userManager.internalData
```



**内聚 - 模块内的"团结程度"** 一个模块应该只做一件事：

- **高内聚**：UserManager只处理用户相关
- **低内聚**：一个模块既管用户又管学习进度

**我们的内聚实践：**

```
// 高内聚 - 功能集中
class LearningEngine {
    // 所有方法都围绕学习功能
    loadChapter() { ... }
    updateProgress() { ... }
    takeTest() { ... }
}

// 低内聚反例
class MiscManager {
    // 什么功能都塞进来
    handleUser() { ... }
    processLearning() { ... }
    manageGame() { ... }
}
```



**开闭原则 - 对扩展开放，对修改关闭** 设计时要考虑未来：

- **开放**：能方便地加新功能

- 关闭：加功能时尽量不改现有代码

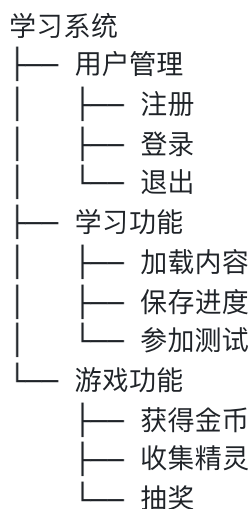
## 我们的应用：

```
// 开闭原则的例子
class RewardSystem {
    // 可以方便添加新的奖励类型
    // 不需要修改现有奖励逻辑
    addRewardType(newType) {
        this.rewardTypes.push(newType);
    }
}
```



### 6.2.2 面向功能设计

结构图 - 功能的层次分解 就像把大任务拆成小任务：



结构化设计方法 自顶向下，逐步细化：

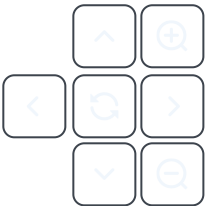
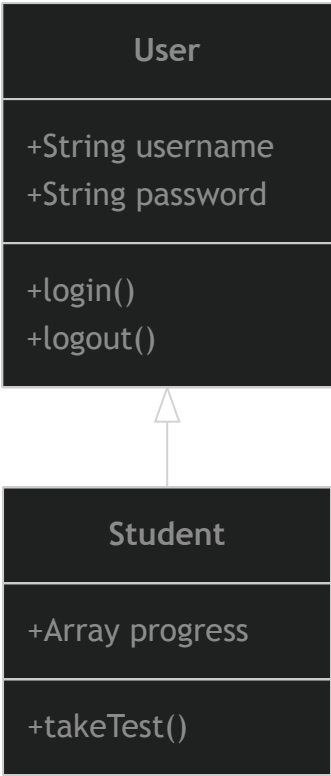
1. 先确定系统要做什么
2. 拆分成主要功能模块
3. 每个模块继续拆分，直到可以编码

### 6.2.3 面向对象设计

#### 面向对象基本概念

- 类：用户的模板（User）
- 对象：具体的用户（张三用户）
- 继承：学生用户继承自用户
- 多态：不同用户类型有不同的行为

UML设计建模 用图形化方式表达设计：



设计方法论 从需求到设计的步骤：

- 1. 识别类和对象
- 2. 定义类的属性和方法
- 3. 建立类之间的关系
- 4. 优化设计结构

6.2.4 详细设计

逻辑/算法设计 具体到每个方法怎么实现：

```
/**
 * 用户登录算法设计
 * 输入：用户名、密码
 * 输出：登录是否成功
 * 步骤：
 * 1. 验证输入不为空
 * 2. 查询用户是否存在
 * 3. 验证密码是否正确
 * 4. 创建用户会话
 * 5. 返回结果
 */
```



类状态模型 对象在不同状态下的行为：

```
class UserSession {
    states: ['未登录', '已登录', '已过期']
    transitions: {
```



```
'登录成功': '未登录' → '已登录',  
'退出登录': '已登录' → '未登录',  
'会话过期': '已登录' → '已过期'  
}  
}
```

## 6.3 学习资源推荐

### 视频资源

- [面向对象设计原则](#) - SOLID原则通俗讲解
- [UML图实战教程](#) - 手把手画UML图
- [设计模式详解](#) - 常用设计模式讲解

### 实用网站

- [UML图教程](#) - 各种UML图的画法
- [设计模式指南](#) - 设计模式图文详解
- [面向对象设计原则](#) - OO设计原则和模式

### 重点阅读

- 教材第六章 - 精读设计概念和方法
- 《设计模式》 - 经典的设计模式书籍
- 《UML精粹》 - UML快速入门指南

## 6.4 习题

### 选择题

1. 低耦合的设计目标是： A. 增加模块间的依赖 B. 减少模块间的依赖 C. 让所有模块共享数据 D. 提高代码执行速度
2. 高内聚意味着： A. 模块功能多样 B. 模块功能集中单一 C. 模块代码行数多 D. 模块接口复杂
3. 开闭原则的主要目的是： A. 让系统完全封闭不能修改 B. 支持扩展而避免修改现有代码 C. 开放所有源代码 D. 关闭不必要的功能
4. 在面向对象设计中，继承用于表示： A. 对象间的组合关系 B. "是一个"的关系 C. 对象间的依赖关系 D. 模块间的调用关系
5. UML类图中，关联关系表示： A. 类之间的使用关系 B. 类之间的继承关系 C. 类之间的实现关系 D. 类之间的包含关系

### 判断题

1. ( ) 耦合度越低越好，应该追求零耦合。
2. ( ) 一个类应该只有一个引起变化的原因。
3. ( ) 面向功能设计比面向对象设计更先进。
4. ( ) 详细设计应该描述每个方法的具体实现逻辑。

### 简答题

1. 为什么在软件设计中要追求"高内聚、低耦合"?
2. 比较面向功能设计和面向对象设计的主要区别。
3. 在我们的Python学习网站中，如何应用开闭原则?

## 6.5 答案

### 选择题答案：

1. B - 减少模块间的依赖（低耦合的核心）
2. B - 模块功能集中单一（高内聚的定义）
3. B - 支持扩展而避免修改现有代码（开闭原则的精髓）
4. B - "是一个"的关系（继承的语义）
5. A - 类之间的使用关系（关联关系的含义）

### 判断题答案：

1. × - 适度耦合是必要的，零耦合可能无法协作
2. √ - 这是单一职责原则的表述
3. × - 各有适用场景，没有绝对的优劣
4. √ - 详细设计要具体到实现逻辑

### 简答题答案：

#### 1. 高内聚低耦合的重要性：

- 提高可维护性：修改影响范围小
- 增强可复用性：模块可以独立使用
- 便于测试：可以单独测试每个模块
- 支持团队协作：不同人开发不同模块

#### 2. 设计方法比较：

- 面向功能：以功能为中心，适合算法密集型系统
- 面向对象：以对象为中心，适合业务模型复杂的系统
- 我们项目选择面向对象，因为用户、课程、精灵都是天然的对象

#### 3. 开闭原则应用：



```
// 奖励系统设计
class RewardSystem {
    constructor() {
        this.rewardStrategies = new Map();
    }

    // 可以添加新奖励类型而不修改现有代码
    addStrategy(type, strategy) {
        this.rewardStrategies.set(type, strategy);
    }
}
```

## 6.6 案例分析：我们的Python学习网站设计

### 设计原则应用

#### 内聚性保证：



```
// 每个类职责单一
class ProgressTracker {
    // 只负责进度跟踪
    updateProgress() { ... }
    getProgress() { ... }
    // 不处理用户认证或界面显示
}
```

### 面向对象设计

#### 类设计实例：



```
class User {
    constructor(username, email) {
        this.username = username;
        this.email = email;
        this.progress = new Progress();
    }

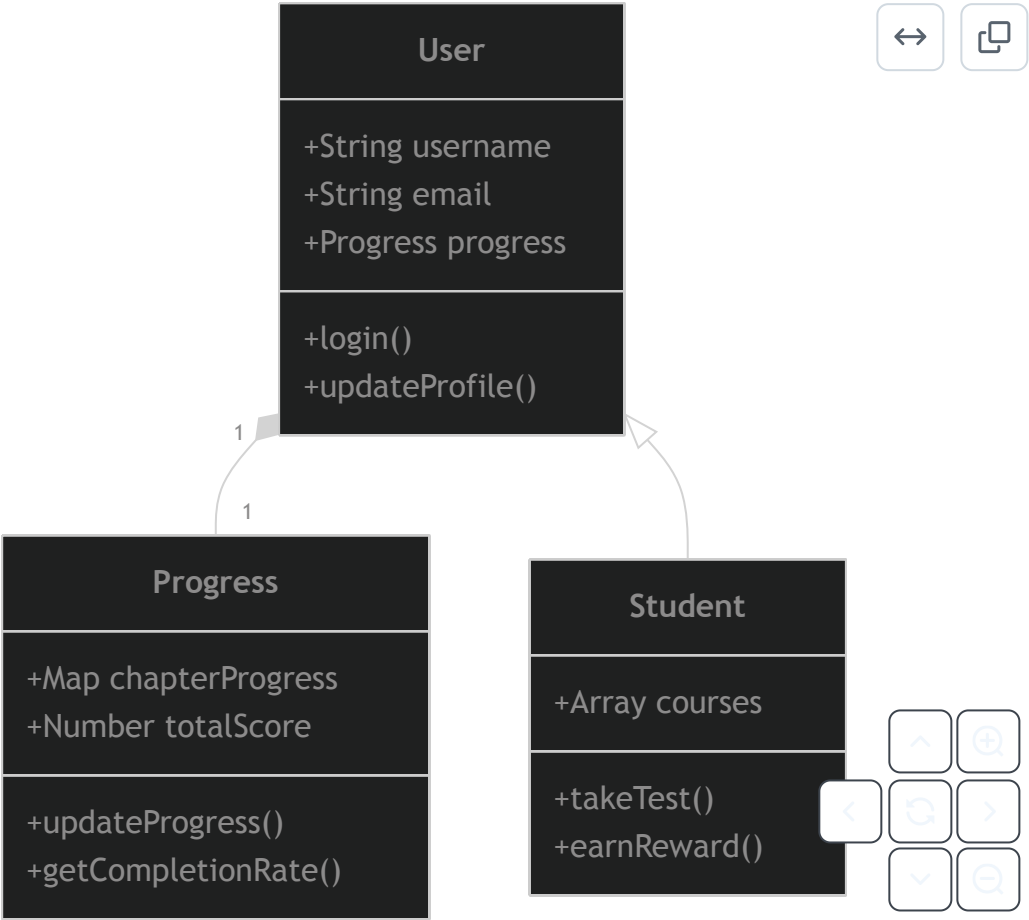
    // 用户基本行为
    updateProfile() { ... }
}

class Student extends User {
    constructor(username, email) {
        super(username, email);
        this.courses = [];
    }

    // 学生特有行为
}
```

```
takeTest() { ... }
earnReward() { ... }
}
```

UML建模应用：



实用建议

- 1. 先设计后编码：好的设计节省编码时间
- 2. 遵循设计原则：高内聚低耦合是基础
- 3. 使用建模工具：UML图帮助理清思路
- 4. 团队设计评审：集体讨论改进设计
- 5. 持续重构：随着理解深入优化设计