CS619

> **Tip**
>
> **UCB CS169: software engineering**
>
> 考虑过后还是优先学习CS619内容，相较于MIT相关课程，CS619更贴近软件工程。
>
> 使用`DownSub`进行Youtube字幕下载最终整理得到笔记。忽略语言以及框架的学习，只关注软件工程本身，故省略一些章节的学习。
>
> 学习时间：2025.03.04 - 2025.03.08

- [x] 概述
  - [x] 1.1 Welcome to Agile Software Engineering
  - [x] 1.2 Early Software Development Processes: Plan-and-Document
  - [x] 1.3 Software Development Processes: Agile
  - [x] 1.4 Testing and Software Quality
  - [x] 1.5: Productivity: Conciseness, Synthesis, Reuse, and Tools
  - [x] 1.6-1.7: Deploying SaaS: SOA (Service Oriented Architecture) and Utility Computing
  - [x] 1.8: Deploying SaaS: Browsers & Mobile
- [x] 语言
  - [x] 2.1: Learning to Learn Languages and Frameworks
  - [x] 2.2: Pair Programming
  - [x] 2.7-2.8: Fallacies & Pitfalls, and How Not to Learn a Language By Googling
- [x] SaaS应用架构
  - [x] 3.1: The Web's Client-Server Architecture
  - [x] 3.2: Interlude: Armando's Computer History Minute - Pair Programming the ENIAC in 1945
  - [x] 3.3: HTTP part 1 - Basics and curl demo
  - [x] 3.3: HTTP part 2 - Cookies
  - [x] 3.4: Microservices and Service Oriented Architecture
  - [x] 3.5: REST APIs and JSON
  - [x] 3.6: RESTful URIs API Calls and JSON - TMDb
  - [x] 3.8-3.9: Demo - A Tour Of Sinatra, A Simple SaaS Framework
  - [x] 3.8-3.9: Fallacies, Pitfalls, and Concluding Remarks
- [ ] SaaS框架：MVC
  - [x] 4.1: Part 1 - Context Setting_ From Sinatra to Rails
  - [x] 4.1: Part 2 - The Model-View-Controller (MVC) Architectural Design Pattern
  - [x] 4.1: Part 3 - Rails as an MVC Framework
  - [x] 4.1: Part 4 - Putting it all together A trip through a Rails app
  - [x] 4.2: Overview - Models As Resources

- [ ] 4.2: Active Record (The Glue) Part 1 - Find
- [ ] 4.2: Active Record Part 2 - Create
- [ ] 4.2: Active Record Part 3 - Update and Destroy
- [ ] 4.2: Active Record Summary
- [ ] 4.2: Databases and Migrations
- [x] 4.4: Part 1 - Controllers and Views
- [ ] 4.4: Part 2 - Redirection, the Flash, and the Session
- [ ] 4.4: Part 3 - Redirect or Render
- [ ] 4.5: Rails Routing Basic
- [ ] 4.6: Part 1 - Creating Forms
- [ ] 4.6: Part 2 - Strong Params for Form
- [ ] 4.6: Part 3 - Forms Summary
- [ ] 4.8: When Things Go Wrong_ Debugging

- [ ] SaaS框架：高级抽象

- [ ] 需求
  - [x] 7.1: Intro to Behavior Driven Design and User Stories
  - [x] 7.2: SMART User Stories
  - [x] 7.3: Lo-Fi User Interface Sketches and Storyboards
  - [x] 7.4: Points and Velocity
  - [x] 7.5: Agile Cost Estimation
  - [x] 7.6: Part 1 - From Stories to Acceptance Tests_ Introducing Cucumber
  - [x] 7.6: Part 2 - Introducing Capybara
  - [ ] 7.6: Part 3 - Basic Cucumber & Capybara demo
  - [ ] 7.6: Part 4 - More advanced Cucumber & Capybara demo
- [x] 测试
  - [x] 8.1: Part 1 - The Big Picture_ FIRST, TDD, and Red-Green-Refactor
  - [x] 8.1: Part 2 - Testing Today_ FIRST
  - [x] 8.2: Anatomy of a Test Case - Arrange, Act, Assert
  - [x] 8.3: Part 1 - Isolating Code - Doubles & Seams Intro
  - [x] 8.3: Part 2 - Doubles & Seams In RSpec & Rails
  - [x] 8.3: Part 3 - Method stubs & mock objects
  - [x] 8.4: Stubbing the Internet
  - [x] 8.6: Part 1 - Fixtures & Factories
  - [x] 8.6: Part 2 - When Fixtures vs. When Factories
  - [x] 8.7: Coverage
  - [x] 8.8: Other Testing Concepts
  - [x] 8.118.12: Concluding Remarks_ TDD vs. Debugging
- [x] 软件维护

- [x] 9.1: What Makes Code *Legacy* and How Can Agile Help
- [x] 9.2: Approaching a Legacy Codebase
- [x] 9.3: Establishing Ground Truth With Characterization Tests
- [x] 9.4: Comments & Commits - Tell Me a Story
- [x] 9.5: Beyond Correctness - Smells, Metrics, and SOFA
- [x] 9.6: Part 1 - Intro to Method Level Refactoring
- [x] 9.6: Part 2 - Reflections on Refactoring
- [x] 9.7: The Plan And Document Perspective on Working With Legacy Code
- [x] 9.8-9.9: Refactoring & Legacy Code - Fallacies, Pitfalls, Conclusions

- [x] 敏捷团队
  - [x] 10.1 It Takes a Team Two Pizza and Scrum
  - [x] 10.2: Effective Branching part 1 - Branch Per Feature
  - [x] 10.2: Effective Branching part 2 - Branches & Deploying
  - [x] 10.3: Design Reviews, Code Reviews
  - [x] 10.4: Delivering the Backlog Using Continuous Integration
  - [x] 10.6: Fixing Bugs -The Five R's
  - [x] 10.8-10.9: Fallacies, Pitfalls, and Concluding Remarks on Teams
- [x] 设计模式
  - [x] 11.1: Patterns, Antipatterns, and SOLID Class Architecture
  - [x] 11.2: Just Enough UML
  - [x] 11.3: Single Responsibility Principle
  - [x] 11.4: Open_Closed Principle
  - [x] 11.5: Liskov Substitution Principle
  - [x] 11.6: Part 1 - Dependency Injection
  - [x] 11.6: Part 2 - More Adapter-Like Patterns
  - [x] 11.7: Demeter Principle
  - [x] 11.8: Plan And Document Perspectives on Design Patterns
- [ ] Dev/Ops
  - [x] 12.1: From Development to Deployment
  - [x] 12.2: Three Tier Architecture
  - [x] 12.3: part 1 - Availability & Responsiveness - An Introduction
  - [ ] 12.3: Responsiveness part 1 - Service Level Objectives and Apdex
  - [ ] 12.3: Responsiveness part 2 - Where does the time go
  - [x] 12.4: Part 1 -Continuous Integration & Continuous Deployment
  - [ ] 12.4: Part 2 - Upgrades & Feature Flags
  - [ ] 12.4: Part 3 - Upgrades & Feature Flags, continued
  - [ ] 12.5: Monitoring
  - [ ] 12.6: Part 1 - Caching_ Speeding up database queries

- [ ] 12.6: Part 2 - Caching for speeding up rendering
- [ ] 12.7: Don't Abuse the Database, part 1_ Indices
- [ ] 12.7: Don't Abuse the Database, part 2 - N+1 query problem
- [ ] 12.9: Defending Customer Data part 1_ SSL_TLS
- [ ] 12.9: Defending Customer Data part 2_ SQL Injection
- [ ] 12.9: Defending Customer Data part 3_ Cross Site Request Forgery
- [x] 12.11-12.12: DevOps Fallacies, Pitfalls, and Concluding Remarks

概述

1.1 Welcome to Agile Software Engineering

课程目标

- 培养学生成为优秀的软件工程师，具备团队合作、沟通、决策和技术能力。
- 强调软件工程的复杂性，需要团队合作才能完成大型项目。
- 学习软件工程的过程，而不仅仅是编程。

> **Important**
>
> **软件工程：多人开发多版本程序。**
>
> - **强调合作和软件的不断开发。**

优秀软件工程师的特点

- **热情和投入**: 对软件工程充满热情，并全身心投入其中。
- **团队技能**: 具备良好的团队合作能力，支持团队成员，相信团队成功才能个人成功。
- **决策能力**: 善于根据信息和替代方案做出明智的决策。
- **技术技能**: 具备良好的技术能力，能够看待问题，跳出固有思维模式，寻找解决方案。

避免糟糕体验的建议

- **被动学习**: 跳过阅读和讲座，直接进行编码作业。
- **孤立学习**: 不与他人合作，不参与团队合作项目。
- **抗拒新工具**: 不学习新的编程工具和语言。
- **过度关注分数**: 将分数视为唯一目标，忽视实际技能的学习。
- **多任务处理**: 同时进行多项任务，分散注意力，降低效率。
- **忽视过程**: 认为过程不重要，只关注编码。

学习建议

- **主动学习**: 认真阅读教材和讲座，积极参与课堂讨论。
- **团队合作**: 积极参与团队合作项目，与队友相互学习。
- **学习新工具**: 不断学习新的编程工具和语言，提高效率。
- **关注实际技能**: 将学习重点放在实际技能上，而非分数。
- **专注学习**: 避免多任务处理，专注于编码任务。
- **重视过程**: 理解和遵循软件开发流程，提高开发效率。

远程编程建议

- **加强团队合作**: 与队友保持沟通，进行结对编程。
- **利用论坛**: 在论坛上提问和回答问题，获取反馈和帮助。
- **保持专业**: 在论坛上保持文明和专业，尊重他人。
- **具体提问**: 提问时提供具体信息，以便他人理解并帮助解决问题。
- **自我解决问题**: 尽量自己解决问题，提高独立解决问题的能力。

❗本课程会更多地聚焦于`敏捷`(Agile)这一相对于轻量级的过程。

1.2 Early Software Development Processes: Plan-and-Document

软件工程的起源与阿波罗计划

- 软件工程起源于1968年，当时NATO赞助了一次会议，旨在应对所谓的**"软件危机"**。

计划和文档集方法

- **"计划和文档集"**方法，旨在模仿大型土木工程项目的完成方式。
- 该方法包括项目经理、详细计划、详细文档、定期检查和瀑布生命周期等阶段。
- 第一个P&D过程：瀑布（1970）

计划和文档集方法的局限性

- **软件与硬件的差异**: 软件可以改变，而硬件则相对固定。用户对软件的需求可能随着时间而变化，导致软件需要不断适应和调整。
- **灵感和嘲讽**: 用户对软件的需求可能不够明确，导致开发出的软件不符合用户的实际需求。
- **计划抛弃**: 在软件开发过程中，可能会发现需要重新设计某些组件或子系统，导致原先的计划需要被抛弃。
- **人月神话**: **增加人力并不一定能提高项目进度**，因为沟通成本和培训成本会随之增加。

其他软件生命周期模型

- 为了应对计划和文档集方法的局限性，出现了其他软件生命周期模型。
- 这些模型试图在构建过程中与客户进行更多互动，以更好地适应需求变化。

最大教训

- 软件必须能够**应对变化**。这与建造硬件、桥梁或建筑物有着根本的不同。
- 软件可以不断迭代和改进，以适应不断变化的需求。

1.3 Software Development Processes: Agile

变化是软件开发不可避免的现实

- 软件必须适应不断变化的需求和环境。
- 布鲁克斯定律指出，**增加人力会导致项目延期。**
- 佩雷斯定律强调，变化不是问题，而是需要应对的事实。

敏捷开发方法

- 敏捷宣言提倡个人和互动、工作软件、客户协作和响应变化。

- 敏捷开发强调**持续改进和迭代开发**。
- 客户参与是敏捷开发的核心，通过迭代和用户故事来满足需求。
- 敏捷开发不是摒弃计划和文档，而是采用更轻量级的方法。
    - 专注于每个阶段，不断演化而不是只在生命周期中做一次计划和文档。

极限编程（XP）

- 极限编程是敏捷开发的一种极端形式。
- XP 强调短迭代、简单设计、**测试驱动开发**和持续集成、代码评审。
- XP 旨在最大限度地提高开发效率和软件质量。

敏捷开发的适用性

- 敏捷开发适用于需求变化快、客户参与度高、项目规模适中的项目。
- 对于需求稳定、客户不参与或项目规模庞大的项目，可能需要更传统的计划驱动方法。

总结

- 软件开发需要适应变化，敏捷开发是一种有效的应对方法。
- 敏捷开发强调持续改进、迭代开发和客户参与。
- 极限编程是敏捷开发的一种极端形式，强调短迭代和简单设计。
- 敏捷开发适用于特定类型的项目，需要根据项目特点选择合适的方法。

1.4 Testing and Software Quality

软件质量

- 软件质量的核心是满足客户需求，确保软件能够**解决客户的问题**。
- 质量保证（QA）涉及确保软件满足客户需求的过程和标准。
- 软件质量不仅仅是测试和查找错误，还包括确保软件能够长期满足客户需求，并进行必要的更新和维护。
- 敏捷开发中，质量是**每个人的责任**。

敏捷开发和测试

- 敏捷开发强调持续改进和迭代开发，测试是敏捷开发的重要组成部分。
- 在敏捷开发中，测试不再是独立的阶段，而是贯穿整个开发过程。
- 开发人员负责编写单元测试，确保代码的正确性。
- 测试人员负责编写自动化测试工具和框架，支持开发人员进行测试。

测试类型

- 验证：硬件更加关注，确保符合规范和设计要求。
- 确认：软件更加关注，确保软件满足客户需求，解决客户的问题。
- 单元测试：测试单个函数或方法的行为。
- 模块测试/功能测试：测试一组协同工作的函数或模块的行为。
- 集成测试：测试系统不同组件之间的接口和通信。
- 系统测试/验收测试：测试整个系统的行为，确保其满足客户需求。

测试策略

- 测试需要**分层**进行，避免重复测试。
- **覆盖率**是衡量测试有效性的指标，指测试用例执行了系统行为的百分比。
- 自动化测试是提高测试效率和质量的的有效方法。
- 测试人员应该编写自动化测试脚本，模拟客户行为，验证系统功能。

总结

- 软件测试和软件质量是软件开发过程中至关重要的环节。
- 通过有效的测试策略，我们可以确保软件满足客户需求，提高软件质量和可靠性。

1.5: Productivity: Conciseness, Synthesis, Reuse, and Tools

为什么生产力重要?

- 摩尔定律使得硬件性能不断提升，需要更高效的软件开发方法来充分利用硬件性能。
- 复杂软件系统的开发需要高效的方法和工具。

提高生产力的四种机制:

- **简洁性**:
  - 使用更少的代码实现更多功能，提高抽象层次。
  - 例子：高级编程语言、灵活的语法、代码习语。
  - 好处：减少认知负荷，代码更易读易懂。
- **综合性**:
  - 让代码编写其他代码，例如代码生成器和**模板**。
  - 例子：Rails 框架中的代码生成器。
  - 好处：**自动化重复性工作**，提高开发效率。
- **重用性**:
  - 不要重复代码，将通用功能抽象成函数、库、对象或设计模式。
  - 例子：函数、库、面向对象编程、设计模式。
  - 好处：减少错误，提高代码可维护性。
- **工具和自动化**:
  - 使用工具和自动化技术来提高开发效率。
  - 例子：Make、Ant、持续集成/持续部署 (CI/CD) 工具。
  - 好处：节省时间，提高准确性。

工具的价值:

- 学习使用新工具需要时间，但长期来看可以节省更多时间。
- 选择值得投资的新工具，提高开发效率。

学习机会:

- 课程中将介绍各种提高生产力的工具。
- 学习使用这些工具，将有助于职业生涯发展。

1.6-1.7: Deploying SaaS: SOA (Service Oriented Architecture) and Utility Computing

部署 SaaS、SOA 和公用计算

- **软件即服务 (SaaS) 的优势**:
  - 无需安装，易于访问和使用。网络浏览器即可。
  - 数据存储在云端，更安全可靠。
  - 支持协作，方便多人共享和编辑。
  - 自动更新，无需手动下载和安装。
  - 可扩展性强，可根据需求调整资源。
- 一些其他事：
  - 桌面应用程序转向软件即服务，摆脱先前压缩包的安装方式。
  - 亚马逊平台，内部多个子系统但在外界看来只有API操作列表。
    - 子系统相互独立
    - 相互组合创造新服务
    - 面向服务架构每个小的部件向外提供自己能做什么的API
- SaaS的需求：
  - 交流
  - 可扩展
  - 可靠
- 解决方案：
  - 集群而不是一台超级计算机，只需添加更多PC即可。
  - 公共云计算，按需付费。
- **面向服务架构 (SOA)**:
  - 将应用程序拆分为独立的子系统，每个子系统负责特定的功能。
  - 子系统之间通过 API 进行通信，实现模块化和可重用性。
  - 亚马逊是 SOA 的先驱，其内部架构基于微服务，每个服务都有自己的 API。
- **公用计算/公共云计算**:
  - 按需付费，用户只需支付实际使用的资源。
  - 提高资源利用率，降低成本。
  - 提高可扩展性和灵活性。
  - 亚马逊 AWS 是公共云服务的代表，为用户提供了丰富的计算资源。

学习目标:

- 了解 SaaS 的优势和应用场景。
- 掌握 SOA 的核心概念和设计原则。
- 理解公共云计算的原理和特点。
- 学习如何利用 SaaS、SOA 和公共云计算构建高效的应用程序。

1.8: Deploying SaaS: Browsers & Mobile

客户端部分的重要性

- 软件应用程序分为客户端和云部分。
- 客户端通常是用户交互的界面，例如智能手机上的移动应用程序。
- 移动设备流量已超过传统计算机。
- 应用程序需要考虑移动友好性，即使不是移动优先的。
- 需要考**虑响应式设计和可访问性。**
- 需要考虑不同设备的交互模型。

HTML 的作用

- HTML 是网络的语言，用于描述文本和其他媒体的结构。
- HTML 元素可以嵌套，形成层次结构。
- HTML 元素可以有属性，如 ID 和 class。
- `id`用以特定元素，`class`用在一组元素。
- HTML 用于结构，CSS 用于样式。

CSS 的作用

- CSS 用于控制页面上不同元素的视觉呈现方式。
- CSS 选择器用于引用元素。
- CSS 框架（如 Bootstrap）提供预定义的样式和组件，方便开发。
- 响应式布局可以适应不同屏幕尺寸。
- 没有理由拒绝一个很好的框架帮我们做到很多的事情。

移动应用程序开发

- 渐进式 Web 应用（PWA）使用 HTML、CSS 和 JavaScript 构建，可以离线运行。
- PWA 可以通过应用商店分发，也可以作为网站的一部分。
- 原生应用程序提供更好的性能和硬件访问，但开发和更新成本更高。
- 移动浏览器上的 JavaScript 绑定功能不断改进。

云端部署的优势

- 可以按需付费，提高资源利用率。
- 可以使用相同的技术开发 Web 应用和移动应用。
- 开发和调试工具不断改进。

语言

2.1: Learning to Learn Languages and Frameworks

学习语言和框架的重要性

- 软件开发需要不断学习新的语言和框架。
- 学习语言是为了使用特定的框架或工具。
- 语言和框架的选择取决于应用场景和个人兴趣。

- 命令式、面向对象
    - 前者一步一步做事情。

学习语言的有效方法

- 理解语言之间的**相似之处**，关注差异。
    - 并不会改变你思考的方式。
- 学习框架定义的抽象，理解**框架如何利用语言特性。**
- 学习语言的成语，掌握特定于语言的表达方式。

本课程的学习重点

- Ruby 和 Rails 框架。
- 学习 Ruby 的原因：测试工具、代码质量工具、社区文化。**代码之美**
- 八步计划学习新语言：
    - 类型和命名
    - 变量、字符串操作
    - 方法定义、类和面向对象
    - 抽象
    - 特殊的习语
    - 库和包管理
    - 调试
    - 测试

学习过程中的注意事项

- 避免同时学习多种相似的语言，容易混淆。
- 不要试图逐字翻译代码，要理解语言的特性和成语。
- 关注语言的**独特之处**，避免陷入反模式。

2.2: Pair Programming

- **结对编程的定义和优势**:
    - 两个人一起工作，一个人是“司机”（输入代码），另一个人是“观察员”（检查和思考）。
    - 通过互相检查，可以提高软件质量。
    - 可以减少完成时间，因为一个人犹豫时，另一个人可能能提供解决方案。
    - 促进知识共享和团队凝聚力。
    - 增强自律，因为需要集中注意力。
- **结对编程的实践**:
    - 使用专门的结对编程工作站，避免分心。
    - 每隔10到15分钟更换角色，锻炼不同的技能。
    - 与不同经验水平的人配对，互相学习。
- **研究发现的结对编程效果**:
    - 对于简单任务，结对编程可以更快完成。
    - 对于复杂任务，虽然不一定更快，但可以获得更好的代码和更高的质量。

- **结对编程的注意事项**:
  - 观察员需要专注于任务，不要分心。
  - 经常更换角色，以锻炼不同的技能。
  - 避免同时学习多种相似的语言，容易混淆。
- **学生反馈和建议**:
  - 结对编程可以避免愚蠢的错误，减少调试时间。
  - 在小组项目中频繁更换合作伙伴可以增强团队凝聚力。
  - 如果不熟悉Ruby或Rails，结对编程是快速上手的好方法。

2.7-2.8: Fallacies & Pitfalls, and How Not to Learn a Language By Googling

- **避免复制粘贴代码**:
  - 不要在不理解代码的情况下从 Stack Overflow 等网站复制代码。
  - 确保理解代码的作用和原理，并根据需要调整代码。
- **批判性地使用教程**:
  - 不要盲目地按照教程的步骤操作，要理解每一步的目的和原理。
  - 选择解释为什么需要这样做的教程，而不是仅仅告诉你怎么做。
- **谨慎使用 LLM (大型语言模型)**:
  - LLM 可以帮助你更快地编写代码，但无法取代对代码内容的理解。
  - 如果你是新手，过度依赖 LLM 可能会导致错误和无法修复的问题。
- **不要翻译代码**:
  - 不要将一种语言的代码简单地翻译成另一种语言。
  - 要学会用新语言思考，理解其独特的特性和习语。
- **学习语言的动机**:
  - 学习语言的目的是为了使用特定的框架或工具。
  - 要了解语言的**特点和框架如何利用这些特点**。
- **学习语言的建议**:
  - 理解语言之间的相似之处，关注差异。
  - 学习框架定义的抽象，理解框架如何利用语言特性。
  - 学习语言的成语，掌握特定于语言的表达方式。
- **代码的优雅性**:
  - 优雅的代码更容易理解、维护和扩展。
  - **代码的优雅性**与其**管理复杂性的能力**密切相关。
- **坚持和信念**:
  - 编程是一项具有挑战性的工作，需要坚持和信念。
  - 当遇到困难时，要记住编程的乐趣在于解决智力挑战。

SaaS应用架构

3.1: The Web's Client-Server Architecture

- web是一种客户端-服务器架构：
  - 是一种请求-响应架构，客户端发送请求，服务器响应请求。
  - 服务器通常设计为同时为多个客户端提供服务。
  - 客户端和服务器各自专注于不同的任务，如客户端负责用户界面，服务器负责数据处理。
  - 客户端和服务器可以使用不同的软件框架，如 React、iOS、Rails、Django、Node 等。
- 域名系统（DNS）：
  - 将计算机名称映射到 IP 地址。
  - 是 Web 浏览器进行主机名查找的必要步骤。
  - 通常由 ISP 运营。
- Web 事务的典型流程：
  - 用户通过 Web 浏览器进行主机名查找。
  - 浏览器向服务器请求内容。
  - 服务器响应请求并提供内容。
- **设计模式：**
  - 是组织相关问题的解决方案的方法。
  - **客户端-服务器架构是一种设计模式**，它定义了客户端和服务器各自的职责。
- 点对点系统：
  - 每个实体都可以**发起请求和接收请求。**
  - 与客户端-服务器架构相比，点对点系统没有专门的角色划分。
- 客户端-服务器架构的历史：
  - Sabre：美国航空公司开发的早期软件即服务客户端-服务器协议。
  - FTP：文件传输协议，允许用户从服务器下载或上传文件。
  - Netware：IBM 的 PC 产品，允许多台 PC 共享文件服务器。
  - 邮局协议（POP）：允许用户从服务器下载电子邮件。
  - IMAP：改进的邮局协议，支持多客户端访问。
  - HTTP：网络的基本通信协议，经历了多次修订，增加了性能增强功能。
- 协议标准：
  - 开放标准：如 DNS、FTP、POP、IMAP、HTTP 等。
  - 专有标准：如 Saber 和 Netware，最终转向基于开放协议和标准。

3.2 Interlude: Armando's Computer History Minute - Pair Programming the ENIAC in 1945

- ENIAC（电子数值积分计算机）：
  - 被广泛认为是第一台或最早的**可编程全电子计算机**之一。
  - 由宾夕法尼亚大学在二战期间开发，作为一项机密防御项目。

    - 使用全真空管设计，尽管存在高温和易烧坏的问题，但比机电式计算机快两个数量级。
- ENIAC 的编程团队：
    - 由女性程序员组成，因为男性都去打仗了。
    - 她们拥有数学天赋或高级数学学位，这在当时是最接近编程的工作。
    - 她们的工作为第一台商用计算机提供了信息。
- 结对编程的早期实践：
    - ENIAC 的程序员以两人一组的形式工作，互相检查彼此的工作，确保代码的正确性。
    - 这种实践可以说是现代结对编程的开端。
- 关注细节的重要性：
    - ENIAC 的程序员强调了细节的重要性，即使是大局观很好，但魔鬼往往隐藏在细节中。
    - 他们认为，要让东西发挥作用，需要一定的工艺或工艺，而最后 10% 的功能可能需要 80% 的努力。
- 可测试性的重要性：
    - ENIAC 的程序员开发了测试程序，通过手动模拟计算机的操作来检查中间结果和最终结果。
    - 这种测试方法可以帮助快速缩小问题的范围，并找出需要修复的硬件部分。
    - 他们认为测试就像随时随身携带的手术工具，可以帮助快速解决问题。

3.3: HTTP part 1 - Basics and curl demo

- **HTTP 协议栈基础**：
    - HTTP 基于传输控制协议 (TCP) 和互联网协议 (IP)，统称为 TCP/IP 协议栈。
    - TCP/IP 协议栈是互联网的核心，负责数据传输和路由。
    - IP 负责将数据包从源地址发送到目标地址，而 TCP 负责确保数据包按顺序到达并正确组装。
- **IP 协议**：
    - 目前最广泛使用的 IP 版本是 IPv4，使用 32 位地址。
    - IPv6 是下一代 IP 协议，使用 128 位地址，可以提供更多的地址空间。
    - 特殊的 IP 地址 127.0.0.1 表示本地主机，即当前计算机。
    - IP是尽力而为，并不可靠。
- **TCP 协议**：
    - TCP 在 IP 之上提供可靠的连接，确保数据包按顺序到达并正确组装。
    - TCP 使用端口来区分同一台计算机上的不同应用程序。
    - TCP 连接是双向的，允许数据在两个方向上传输。
- **Route = HTTP method + URI**
- **HTTP 协议**：
    - HTTP 是超文本传输协议，用于在 Web 浏览器和 Web 服务器之间传输数据。
    - HTTP 使用请求-响应模型，客户端发送请求，服务器返回响应。
    - HTTP 请求包含方法（如 GET、POST）、URI、协议版本、头部和可选的正文。
    - HTTP 响应包含状态码、头部和可选的正文。
- **HTTP 方法**：
    - GET: 请求获取资源。

    - POST: 向服务器发送数据，通常用于创建或更新资源。
    - PUT: 更新或替换现有资源。
    - DELETE: 删除资源。
- **URI 和 URL**:
    - URI 是统一资源标识符，用于标识网络上的资源。
    - URL 是统一资源定位符，是 URI 的一个子集，包含资源的地址信息。
- **HTTP 状态码**:
    - 200: 请求成功。
    - 300: 重定向。
    - 400: 请求错误。
    - 500: 服务器错误。
- **HTTP 头部**:
    - HTTP 头部包含关于请求或响应的元数据，例如内容类型、内容长度、缓存控制等。
- **curl 工具**:
    - curl 是一个命令行工具，用于发送 HTTP 请求和查看 HTTP 响应。
    - curl 可以用于测试 Web 服务器、调试网络连接等。
- **学习 HTTP 的重要性**:
    - 理解 HTTP 协议对于开发 Web 应用程序至关重要。
    - HTTP 协议是互联网的基础，几乎所有网络应用程序都使用 HTTP。

3.3: HTTP part 2 - Cookies

- HTTP 协议的**无状态特性**:
    - 每个请求都是独立的，没有上下文信息。
    - 即使是对同一服务器的连续请求，服务器也无法区分它们是否来自同一用户。
- Cookies 的作用:
    - 用于在无状态的 HTTP 协议中实现状态保持。
    - 服务器通过设置 Cookie 来存储信息在客户端。
    - 客户端在后续的请求中携带这些 Cookie，**以供服务器识别用户**。
- Cookie 的工作原理:
    - 服务器在响应中设置 `Set-Cookie` 头部，包含 Cookie 的名称、值和其他属性。
    - 浏览器存储这些 Cookie。
    - 在后续的请求中，浏览器在请求头部中包含 `Cookie` 字段，发送这些 Cookie 给服务器。
- Cookie 的生命周期:
    - 可以设置过期时间，过期后 Cookie 会被删除。
    - 在隐私模式下，Cookie 会在会话结束时被删除。
- Cookie 的安全性:
    - Cookie 可以被篡改，因此不应存储敏感信息。
    - 可以使用 HTTPS 来加密 Cookie 的传输过程。
- Cookie 的起源:

    - Cookie 的名称来源于早期文本冒险游戏中的“魔术饼干”，它是一种必须保留以便在游戏中使用的物品。
    - Netscape Navigator 的开发者将这种概念应用到 HTTP 中，用于实现会话跟踪。
- Cookie 的使用场景：
    - 用户身份验证和会话管理。
    - 跟踪用户行为和个性化内容。
    - 保存用户的偏好设置。

3.4: 🚨 Microservices and Service Oriented Architecture

- **Web 1.0 时代**：
    - 网站主要提供静态内容，如文字和图片。
    - 用户通过浏览器向服务器发送 HTTP 请求获取信息，通常是 **HTML 文档**。
    - 每次请求都会导致加载**一个全新的页面**，用户体验较差。
- **Web 2.0 时代的改进**：
    - 网站逐渐转变为动态服务，更像是**在线运行的程序**。
    - JavaScript 的出现允许页面与服务器进行交互，无需重新加载整个页面。
        - 返回一个数据交换结构。
    - 异步 JavaScript 和 XML (Ajax) 技术实现了页面内容的动态更新，提升了用户体验。
- **微服务的概念**：
    - 微服务是支持基于消息通信的小型、独立可部署的服务。
    - 它们通常不是完整的网站，而是**提供特定功能的服务**。
    - 微服务可以独立开发和部署，由小型团队管理，响应速度快。
- **微服务的优势**：
    - 高度专业化，可以专注于**狭窄的任务**。
    - 小型团队可以快速响应客户需求，进行迭代开发。
    - 开发人员与客户距离近，可以快速进行增强。
- **微服务的缺点**：
    - 页面加载时间可能受**最慢**组件的影响。
    - 需要考虑部分故障的处理方式。
    - 需要管理更多接口和契约。
    - 开发人员需要了解 DevOps 知识。
- **RESTful API 的作用**：
    - RESTful API 是一种设计原则，用于构建可访问的、可维护的 API。
    - 它有助于解决微服务架构中接口管理的问题。
- **DevOps 文化**：
    - DevOps 文化鼓励开发人员和运维人员合作，共同保证服务的稳定运行。
    - 这有助于**提高团队对服务故障的调试能力**。

3.5: REST APIs and JSON

- RESTful APIs 的核心概念：
  - REST 是一种组织原则，用于构建可访问的、可维护的 API。
  - RESTful API 基于资源概念，**服务器管理的一切都被视为资源。**
  - API 调用影响资源，例如获取、创建、更新或删除资源。
- RESTful API 的操作：
  - 创建资源：POST 请求。
  - 读取资源：GET 请求。
  - 更新资源：PUT 请求。
  - 删除资源：DELETE 请求。
  - 列出资源集合：GET 请求，通常带有过滤器。
- RESTful API 的资源标识：
  - 使用 URI 指定资源，包括主机名、路径和可选的查询参数。
  - 路径部分通常指定操作和资源类型。
- RESTful API 的参数传递：
  - 参数可以是 URI 路径的一部分。
  - 参数可以是查询字符串的一部分。
  - 参数可以是 JSON 或 XML 有效负载。
- RESTful API 的返回值：
  - 绝大多数返回 JSON 数据结构。
  - 可以是 XML 或其他格式，但 JSON 最常见。
  - 返回值可以是资源副本、操作结果或错误信息。
- RESTful API 的错误处理：
  - HTTP 状态码指示操作是否成功。
  - 状态码 4xx 表示客户端错误。
  - 状态码 5xx 表示服务器错误。
  - 返回的数据结构可能包含错误描述。
- JSON 数据结构：
  - JSON 代表**JavaScript 对象表示法。**
  - 类似于 Python 字典或 Ruby 哈希。
  - 可以包含字符串、数字、数组和其他嵌套对象。
- RESTful API 设计原则：
  - API 应该是无状态的，每个请求都应该包含所有必要的信息。
  - API 应该使用 HTTP 方法来表示操作。
  - API 应该使用 URI 来标识资源。
  - API 应该使用 JSON 或 XML 格式来表示数据。
- RESTful API 的优势：
  - 简单易用，易于理解和实现。

    - 可伸缩性好，可以轻松扩展和维护。
    - 与语言无关，可以使用任何编程语言实现。

3.6: RESTful URIs API Calls and JSON - TMDB

- 使用 RESTful API 的步骤：
    - 阅读和理解 API 文档。
    - 根据文档构建请求 URI 和参数。
    - 设置必要的请求头，例如授权标头。
    - 发送请求并解析返回的 JSON 数据。
    - 处理可能的错误和异常。
- 工具和资源：
    - 使用开发者网站提供的代码示例和工具。
    - 利用ChatGPT 和 StackOverflow 等工具解决文档中的疑问。
    - 将常用的 API 请求保存为脚本或函数，以便重复使用。

3.8-3.9: Demo - A Tour Of Sinatra, A Simple SaaS Framework

- Sinatra 简介：
    - Sinatra 是一个轻量级的 Ruby Web 框架，提供简洁的 API 来构建 Web 应用程序。
    - Sinatra 基于 Rack，一个 Ruby Web 服务器接口。
- Sinatra 应用程序结构：
    - Sinatra 应用程序是一个继承自 `Sinatra::Base` 的类。
    - 使用 `get`、`post`、`put` 和 `delete` 方法定义路由和对应的处理函数。
- 路由参数：
    - 使用 `:symbol` 的形式在路由中定义参数，参数值会自动填充到 `params` 哈希中。
- 视图和模板：
    - 使用 `.erb` 文件定义 HTML 视图，可以使用嵌入式 Ruby 表达式。
    - `erb` 文件可以包含在布局文件中，布局文件会自动包裹每个视图。
- 会话管理：
    - 使用 `session` 哈希存储跨请求的状态。
    - `session` 哈希的内容会被序列化并存储在 Cookie 中。
- Sinatra 优势：
    - 简洁易用，易于理解和实现。
    - 适合小型 Web 应用程序和微服务。
    - 可以与其他 Ruby 库和工具集成。
- Sinatra 与 Rails 的关系：
    - Sinatra 和 Rails 都是基于 Rack 的 Web 框架。
    - Rails 提供更多功能和灵活性，适合大型 Web 应用程序。
    - Sinatra 的概念和机制在 Rails 中也有体现。

- 学习 Sinatra 的意义：
  - 理解 Sinatra 有助于更好地理解 Rails 和其他 Web 框架。
  - Sinatra 可以用于构建小型 Web 应用程序和微服务。

3.8-3.9: Fallacies, Pitfalls, and Concluding Remarks

- 使用 API 的技能：
  - 阅读和理解 API 文档。
  - 学会逐步完成 API 的使用。
  - 了解如何进行身份验证，必要时使用 `curl` 手动尝试。
- API 页面的工具和资源：
  - API 页面通常提供代码示例和工具。
  - 可以使用 `curl` 或不同语言的 HTTP 库调用 API。
  - 使用 `jq` 来打印和解析结果。
- 本地语言绑定：
  - API 包装器库提供了类和方法集合，简化了 API 使用。
  - 包装器将 API 资源封装为易于使用的对象。
  - 了解底层 HTTP 调用原理有助于调试。
- 资源的概念：
  - 资源是应用程序的组织原则，代表应用程序中的内容。
  - 资源优先的设计方式有助于清晰管理应用程序状态。
  - 资源的操作包括创建、读取、更新、删除和索引。
- 资源优先的设计：
  - 围绕应用程序中的内容及其操作来设计资源。
  - 使用资源关联来管理应用程序状态。
  - 资源优先的设计使流程的每个步骤清晰，便于状态管理。

SaaS框架：MVC

4.1: Part 1 - Context Setting_ From Sinatra to Rails

- 从 Sinatra 到 Rails 的过渡：
  - 介绍 Rails 之前，先介绍 Sinatra，以展示 Web 应用程序的基本结构。
  - Sinatra 是一个简单框架，Rails 是一个更强大的框架，提供更多复杂性和灵活性。
  - Rails 注重应用程序结构，如 MVC 模式和 RESTful 资源。
- Rails 应用程序的结构：
  - Rails 应用程序由模型、视图和控制器组成。
  - 模型负责数据交互，视图负责显示数据，控制器负责处理请求和响应用户。
  - Rails 提供了路由机制，将 HTTP 请求映射到控制器动作。
- 活动记录（Active Record）：
  - 活动记录是 Rails 中的模型部分，用于实例化模型并处理数据库操作。

    - 它简化了数据操作，如查询、更新和删除。
- 概念基础的重要性：
    - 讲座主要介绍概念基础，帮助理解 Rails 的工作原理。
    - 目标是提供足够的概念词汇，以便更好地理解 Rails 文档和示例。
- 使用大型语言模型（LLM）学习 Rails：
    - LLM 可以生成 Rails 应用程序的示例代码，帮助理解编程模式。
    - 例如，可以生成一个具有 CRUD 操作的 Rails 应用程序示例。
    - LLM 生成的代码可以作为学习工具，但不应替代实际学习和实践。
- 脚手架（Scaffolding）：
    - Rails 提供脚手架工具，可以快速生成应用程序的基本结构。
    - 通过阅读脚手架代码，可以学习 Rails 的最佳实践和模式。
    - 脚手架是一个有用的工具，但应谨慎使用，不应替代实际开发经验。

4.1: 🚨 Part 2 - The Model-View-Controller (MVC) Architectural Design Pattern

- 模型-视图-控制器（MVC）架构设计模式：
    - MVC 是一种设计模式，用于组织应用程序的结构。
    - 它包括三个主要组件：控制器、模型和视图。
    - 控制器负责处理用户输入，更新模型状态，并更新视图。
    - 模型负责存储应用程序的数据和逻辑。
    - 视图负责显示数据并允许用户与数据交互。
- MVC 的优点：
    - 将应用程序的不同部分分开，提高了代码的可维护性和可测试性。
    - 使得不同组件可以独立更新和替换。
    - 提高了代码的重用性。
- MVC 的应用场景：
    - 适用于复杂的应用程序，尤其是那些需要处理大量数据和用户交互的应用程序。
    - 适用于需要频繁更新和修改的应用程序。
    - 适用于需要跨平台或跨设备运行的应用程序。

4.1: 🚨 Part 3 - Rails as an MVC Framework

- Rails 作为 MVC 框架：
    - Rails 是一个遵循 MVC 设计模式的框架。
    - MVC 模式将应用程序分为模型、视图和控制器三个部分。
    - 模型负责数据管理，视图负责显示数据，控制器负责处理用户输入。
- Rails 的 MVC 实现：
    - Rails 应用程序中的每个资源通常对应一个控制器、一组视图和一个模型。
    - 控制器负责处理请求并调用相应的模型和视图。
    - 视图负责将数据呈现给用户。

    - 模型负责数据的存储和逻辑处理。
- Rails 的路由系统：
    - Rails 有一个独立的路由系统，用于**将请求映射到相应的控制器和动作**。
    - 路由系统使得应用程序的结构更加清晰和灵活。
- Rails 的模型和数据库：
    - Rails **模型通常与关系数据库中的表对应**。
    - Rails 使用 ActiveRecord 作为对象关系映射（ORM）工具，简化了数据库操作。
    - 模型与数据库表之间的关系通过外键等机制来维护。
- Rails 的视图和模板：
    - Rails 视图通常使用 ERB 模板系统，允许在 HTML 中嵌入 Ruby 代码。
    - Rails 视图可以输出多种格式，如 HTML、JSON 和 XML。
- Rails 的部署和环境：
    - Rails 应用程序可以在不同的环境中运行，如开发环境、测试环境和生产环境。
    - Rails 具有内置的环境管理机制，可以根据不同的环境配置应用程序。
    - Rails 应用程序可以使用多种数据库和应用服务器，如 SQLite、PostgreSQL。
- Rails 的优势：
    - Rails 提供了丰富的抽象和工具，简化了应用程序的开发。
    - Rails 的 MVC 架构有助于提高代码的可维护性和可测试性。
    - Rails 的社区活跃，提供了大量的资源和帮助。

4.1: Part 4 - Putting it all together A trip through a Rails app

- Rails 应用程序的结构：
    - Rails 是一个遵循 MVC 设计模式的框架。
    - MVC 模式将应用程序分为模型、视图和控制器三个部分。
    - 模型负责数据管理，视图负责显示数据，控制器负责处理用户输入。
- Rails 的路由系统：
    - Rails 有一个独立的路由系统，用于将请求映射到相应的控制器和动作。
    - 路由系统使得应用程序的结构更加清晰和灵活。
    - 路由映射在 `routes.rb` 文件中定义。
- Rails 的控制器和动作：
    - 控制器负责处理请求并调用相应的模型和视图。
    - 控制器中的每个动作都对应一个 HTTP 方法（如 GET、POST、PUT、DELETE）和一个资源（如用户、电影等）。
    - 控制器动作可以设置实例变量，这些变量在渲染视图时可见。
- Rails 的视图和模板：
    - Rails 视图通常使用 ERB 模板系统，允许在 HTML 中嵌入 Ruby 代码。
    - 视图与控制器动作名称相匹配，位于 `app/views` 目录下。
- Rails 的模型和数据库：
    - Rails 模型通常与关系数据库中的表对应。

    - Rails 使用 ActiveRecord 作为对象关系映射（ORM）工具，简化了数据库操作。
- **Rails 的约定优于配置**：
    - Rails 通过命名约定来减少配置文件的需求。
    - 例如，**控制器和视图的名称通常与资源名称相匹配。**
- Rails 的 DRY（Don't Repeat Yourself）原则：
    - **应用程序的重要信息应该放在一个地方，以避免重复。**
    - 例如，路由信息集中在一个文件中，方便管理和修改。
- **Rails 的哲学**：
    - 约定优于配置：通过遵循命名约定，Rails 可以自动推断许多配置。
    - DRY 原则：避免代码重复，提高可维护性。

4.2: Overview - Models As Resources

- 模型在 MVC 设计模式中的作用：
    - 模型是应用程序的核心，负责数据管理和业务逻辑。
    - 应用程序的状态和操作都集中在模型中。
    - 模型操作包括创建、读取、更新和删除资源。
- RESTful 资源与模型的关系：
    - RESTful API 将应用程序视为资源的集合。
    - 模型可以被视为应用程序操作的执行者，负责对资源进行操作。
    - 模型中的方法对应于对资源执行的操作。
- Rails 应用程序中的模型与数据库：
    - Rails **模型通常对应于数据库表。**
    - 模型的**属性对应于数据库表的列。**
    - 模型的**实例对应于数据库表中的行。**
- Rails 的数据库抽象：
    - Rails 提供了 ActiveRecord ORM，用于抽象数据库操作。
    - ActiveRecord **自动生成 SQL 语句**，避免了手动编写 SQL 的风险和复杂性。
        - 对实例操作而不是直接对数据库对象操作。
    - ActiveRecord 提供了安全性，防止 SQL 注入等安全问题。
- Rails 的命名约定：
    - Rails 使用命名约定来确定数据库表名和主键名称。
    - 模型类的复数形式通常用作数据库表名。
    - 主键通常命名为 `id`，并自动递增。
- Rails 中的模型操作：
    - Rails 提供了创建、读取、更新和删除操作的方法。
    - 这些操作对应于数据库的增删改查操作。
    - Rails 使用 ActiveRecord 提供的方法来执行这些操作。

4.4: Part 1 - Controllers and Views

- 控制器和视图的职责划分：
  - 模型负责**数据管理和业务逻辑**。
  - 控制器负责**从模型中获取数据，或将数据添加到模型中**，并在必要时**将这些信息提供给视图**。
  - 视图负责**格式化数据并展示给用户**。
- 路由的作用：
  - 路由是连接模型、控制器和视图的粘合剂。
  - 路由定义了用户请求的 URL 与控制器动作之间的映射关系。
- 控制器方法与 RESTful API 的对应关系：
  - 控制器方法对应于对资源的操作，如创建、读取、更新和删除。
  - 例如，`show` 方法对应于读取操作。
- 路由参数：
  - 引用特定资源的路由通常包含一个 ID 参数。
  - Rails 会自动从 URL 中提取参数并放入 `params` 哈希中。
- 表单数据：
  - 当用户提交表单时，表单字段的名称和值会被放入 `params` 哈希中。
- 实例变量：
  - 控制器可以使用 `params` 哈希中的数据来设置实例变量。
  - 视图可以访问这些实例变量来展示数据。
- 视图功能：
  - 视图可以格式化数据并展示给用户。
  - 视图可以提供操作资源的链接或表单，如更新或删除资源。

4.8: When Things Go Wrong--Debugging

- 调试软件作为服务的挑战：
  - 多个组件参与请求处理（路由、控制器、模型、视图）。
  - 错误可能发生在请求处理的任何阶段。
  - 错误可能不一致，影响特定路由或用户。
- 调试工具和方法：
  - `printf` 调试（在关键位置打印变量值）。
  - 日志记录：使用 Rails 的日志抽象记录不同级别的消息。
  - 调试器：在开发环境中进行交互式调试。
  - **日志即服务**（如 Heroku 提供的服务）：将日志导出到外部工具，便于搜索和警报设置。
- **调试策略（RASP）：**
  - 读取（Read）：仔细阅读错误消息。
  - 询问（Ask）：向有经验的人提问。
  - 搜索（Search）：在 Google 和 Stack Overflow 上搜索解决方案。
  - 发布（Publish）：在 Stack Overflow 上发布问题，提供最小可重现的例子。

- 错误处理和调试技巧：
  - 分析错误堆栈跟踪，找到与应用程序相关的部分。
  - 注意 `nil` 错误，检查变量是否为 `nil`。
  - 使用 `debug` 和 `inspect` 方法在视图中打印变量值。
  - 使用日志记录功能记录信息，便于后续分析和问题追踪。
- 效率和学习的建议：
  - 复用现有工作，避免重复造轮子。
  - 有效利用 Google 和 Stack Overflow。
  - 编写简洁、可读性强的代码。
  - 不要害怕搜索和提问，这是提高效率和学习的关键。
- 调试技能的重要性：
  - 调试能力是成为优秀 Rails 和 SaaS 程序员的关键。
  - 调试技能可迁移到其他框架和技术。

需求

🚨 7.1: Intro to Behavior Driven Design and User Stories

- **行为驱动设计（BDD）**
  - 目的：**避免自上而下计划和文档方法的失败记录，通过让利益相关者持续参与来确保项目满足需求。**
  - 方法：使用有效的产品类型，逐步部署新功能，并收集反馈。
    - 不断进行迭代。
  - 核心概念：通过持续的对话，特别是与客户，来**减少沟通不畅的风险。**
- 用户故事：用日常语言描述应用程序的特定行为，**关注商业价值而非实现细节。**
  - 用户关注的是能否满足他们想要的功能。
  - 用户故事格式：通常包含三个部分：**谁（利益相关者）、他们的长期目标是什么、这个故事中描述的任务是什么。**
  - 用户故事的特点：通俗易懂、足够小可以放在索引卡上、描述应用程序的特定行为、简短。
  - 用户故事的价值：捕捉功能的商业价值，**便于利益相关者参与头脑风暴**，灵活地确定优先级。
  - 用户故事的多样性：相同的功能对**不同利益相关者**可能具有**不同的商业价值。**
  - 用户故事的重要性：**保持简短、非技术性描述**，以便**所有利益相关者都能参与对话。**

7.2: SMART User Stories

- SMART 用户故事：
  - 具体（Specific）：足够详细和具体地描述了一种行为。避免主观的问题。
  - 可衡量的（Measurable）：需要有一种方法来明确地知道用户故事何时已成功实施。
    - 先决条件下某些事是否发生。
  - 可实现的（Achievable）：理想情况下，可以在一次迭代中完成。
    - 进行分解，逐步推进功能。
  - 相关的（Relevant）：有商业价值，值得开发人员花时间来做。

        - Why?
    - 有时间限制的（Timeboxed）：知道何时放弃。
        - 超出预期而拖延了，可能需要重新绘制草图了。
- 用户故事的示例：
    - 用户可以按标题搜索电影。not Smart，不够具体
    - 用户可以使用免费电影通行证兑换一部符合条件的电影。no，什么是符合条件的？
    - 当用户执行广告电影工作流程时，该工作流程的 99% 的页面加载应在三秒内出现。
- 类责任和协作者卡：
    - 总结设计新类时需要了解的信息。
    - 类负责了解哪些信息，它跟踪哪些信息。
    - 类知道如何做什么，以及它必须与哪些其他类协作。

7.3: Lo-Fi User Interface Sketches and Storyboards

- 低保真用户界面草图和故事板的重要性：
    - 避免误解客户需求。
    - 让利益相关者持续参与。
    - 逐步部署新功能，并收集反馈。
    - 减少沟通不畅的风险。
- 低保真用户界面草图和故事板的优点：
    - 易于理解和实现。
    - 可伸缩性好，可以轻松扩展和维护。
    - 与语言无关，可以使用任何编程语言实现。
- 低保真用户界面草图和故事板的制作方法：
    - 使用笔和纸或铅笔和纸画一个东西。
    - 绘制每个页面的小草图，然后告诉用户，好的，告诉我您要单击页面上的哪个位置。
    - 手动绘制和动画化与我们的工作流相对应的事件序列。
- 低保真用户界面草图和故事板的优点：
    - 非技术型客户来说，这样就不会那么吓人，所以更容易与他们进行讨论。
    - 速度也很快，因为你可以在白板前，你可以一边聊天一边画草图。
    - 事实证明，当你画草图时，客户更有可能提出修改建议。
    - **让他们专注于交互，这是行为驱动设计的行为部分。**

7.4: Points and Velocity

用户故事

- 用户故事是敏捷开发中描述用户需求的一种方式，通常以“作为…，我想要…，以便…”的格式表达。
- 用户故事应该尽可能简洁，并具有明确的商业价值。
- 用户故事可以**分解成更小的子故事**，以便**更好地进行估算和管理。**

故事点

- 故事点是衡量用户故事难度的标准，通常使用斐波那契数列（1, 2, 3, 5, 8, 13…）来表示。

- 故事点的分配应该**由整个团队共同决定**，并基于团队成员的经验和对故事的理解。
- 故事点并不是衡量开发时间的标准，而是**衡量工作量的标准**。

速度

- 速度是指团队每次迭代交付的故事点数，通常使用指数加权移动平均数来计算。
- 速度可以帮助团队预测未来的工作量和交付时间。
- **速度是团队内部存在的**，不同的团队可能具有不同的速度。

Pivotal Tracker

- Pivotal Tracker 是一个敏捷项目管理工具，可以帮助团队跟踪用户故事、任务、迭代和速度。
- Pivotal Tracker 提供了多种功能，例如：
    - 故事板：可以查看用户故事的当前状态，例如待办、进行中、完成等。
    - 任务板：可以跟踪每个用户故事的任务，并记录任务的完成情况。
    - 速度图表：可以查看团队的迭代速度，并预测未来的工作量和交付时间。
    - 故事搜索：可以根据关键词、标签等条件搜索用户故事。
    - 故事分析：可以分析用户故事的完成情况，并找出潜在的问题。

其他概念

- 史诗：史诗是指一系列相关的用户故事，通常代表一个大的功能或特性。
- 冰箱故事：冰箱故事是指尚未被团队讨论的用户故事，通常代表一些潜在的需求。
- 尖峰：尖峰是指探索新技术或新功能的故事，通常不需要编写实际的代码。
- Bug：Bug 是指软件中的缺陷，通常需要修复。
- 杂务：杂务是指团队必须完成的一些非功能性的工作，例如代码重构、升级依赖库等。
    - Bug以及杂事不应该有价值和分数。

7.5: Agile Cost Estimation

- **Pivotal Labs 的成本估算方法**
    - 避免固定交付日期和功能
        - **不承诺在特定日期前交付特定功能**，而是投入资源以最有效的方式工作。
    - 迭代式开发
        - 每次迭代分配固定数量的开发小时。
        - 客户参与优先级定义和进度调整。
    - 范围界定会议
        - 经验丰富的 Pivotal 工程师与客户共同讨论项目范围。
        - 客户提供设计草图、文档等信息。
        - 工程师识别估算中的**不确定性因素**。
    - 时间窗口和协议
        - 根据会议结果，确定完成工作的时间窗口。
        - 协议基于时间和材料成本。
        - 如果出现不确定性，进度可能调整。
    - **客户参与**

        - 客户参与整个开发过程。
        - 定期审查进展和功能原型。
        - 客户是参与者和合作伙伴。
- **与传统项目管理的区别**
    - **灵活性**： 避免固定承诺，允许根据实际情况调整进度。
    - **持续合作**： 与客户保持密切沟通，共同解决问题。
    - **透明度**： 客户了解项目的进展和潜在风险。
    - **共同目标**： 客户和开发团队共同致力于项目的成功。

7.6: Part 1 - From Stories to Acceptance Tests： Introducing Cucumber

- Cucumber 是一种行为驱动开发 (BDD) 工具，它使用接近自然语言的方式来描述测试用例，并将其与代码关联起来，以便进行自动化测试。
- Cucumber 的核心思想是**用户故事和验收测试之间的对应关系。**
- 用户故事是用客户友好的语言描述的，可以转化为可执行的测试用例。
- **验收测试是验证软件行为是否符合用户预期。**
- Cucumber 测试用例是用接近自然语言编写的，但需要定义应用程序上下文中有意义的词汇表。
- Cucumber 使用五个关键字来描述不同的步骤：Given, When, Then, And, But。
- 每个功能可以包含多个场景，每个场景描述用户与软件交互的具体步骤。
- 场景中的步骤描述用户在应用程序前会做什么。
- Cucumber 通过**正则表达式**将场景中的步骤与步骤定义进行匹配。
- 步骤定义是实际导致步骤描述发生的方法。
- 关键字 Given, When, Then, And, But 是相同方法的别名，**用于提高代码的可读性。**
- Cucumber Rails Training Wheels 提供了一些基本的步骤定义，用于常见的操作，如点击链接、提交表单等。

7.6: Part 2 - Introducing Capybara

- Cucumber 是一个 BDD 工具，它使用自然语言来描述测试用例，并将其与代码关联起来。
- Capybara 是一个与 Cucumber 配合使用的库，它模拟用户操作，通过自动执行浏览器操作来进行测试。
- Capybara 可以向应用中的任何路由创建 HTTP 请求，提交表单，与 UI 元素交互，接收页面，检查响应内容等。
- Capybara 使用 Given, When, Then, And, But 等关键字来描述测试步骤。
- Capybara 可以通过标签或 CSS 选择器来定位页面元素。
- Capybara 通过 Rack Test 适配器与 Rails 应用交互，并通过无头浏览器或 Web 驱动程序模拟用户操作。
- Capybara 可以模拟包含 JavaScript 的应用程序的行为。
- Capybara 可以直接控制应用，例如设置数据库中的内容，或者通过用户界面与独立部署的应用进行交互。
- Capybara 的要点是模拟浏览器操作，并允许以编程方式检查结果。

测试

8.1: Part 1 - The Big Picture_ FIRST, TDD, and Red-Green-Refactor

- **测试驱动开发（TDD）**：
  - TDD 是一种开发方法，强调先编写测试再编写代码。
  - 目标是确保代码的功能性和可测试性。
  - TDD 过程包括编写测试、运行测试（期望测试失败）、编写代码使测试通过、重构代码。
- **Rspec**：
  - Rspec 是一个测试框架，使用 Ruby 语言编写。
  - 它提供了一套 DSL（领域特定语言），用于编写测试用例。
  - Rspec Rails 是 Rspec 的一个扩展，用于测试 Web 应用程序。
- **Rspec 测试类型**：
  - **单元测试**： 测试单一方法或类的功能。
  - **功能测试**： 测试多个方法或类的功能。
  - **集成测试**： 测试多个组件或模块的集成。
- **测试的重要性**：
  - 测试可以提高软件质量和开发效率。
  - 测试可以帮助开发团队更好地理解用户需求。
  - 测试可以自动化测试用例，节省时间和精力。
- **调试**：
  - 调试是测试的一个重要组成部分。
  - 调试的目的是找到并修复代码中的错误。
- **测试的局限性**：
  - 测试不能证明软件中没有错误，只能证明它们存在。
  - 测试可以帮助发现错误，但不能保证软件的完美性。

8.1: Part 2 - Testing Today: FIRST

- 测试的历史：
  - 过去，**测试的重点是调试**，开发人员编写代码后进行调试。
  - QA 团队负责进行真正的测试，并将错误报告给开发人员。
  - QA 人员进行手动测试，寻找应用未按预期运行的地方。
- 现代测试：
  - 测试更侧重于可维护性和验证。
  - **测试是敏捷迭代的一部分，开发人员需要自己完成大部分测试。**
  - 测试流程更加自动化，开发人员更接近自己的代码。
  - QA 小组不再专注于测试，**而是专注于创建工具和流程来改进测试。**
- 测试的哲学：

    - 测试应该快速、独立、可重复、自检和及时。
    - 快速**F**ast：测试应该快速运行，以便在开发过程中持续运行。
    - 独立**I**ndependent：测试之间不应相互依赖，可以按任何顺序运行。
    - 可重复**R**epeatable：测试应该在相同的条件下产生相同的结果。
    - 自检**S**elf-checking：测试可以自动检测问题，无需人工查看输出。
    - 及时**T**imely：测试应该在编写代码的同时编写。
- 测试的挑战：
    - 可重复性：测试时间相关的行为可能存在颤动测试。
    - 独立性：测试之间可能存在依赖关系。
    - 控制外部条件：测试代码可能依赖于随机性或时间等外部条件。
- 测试工具：
    - Cucumber：使用自然语言描述测试用例的 BDD 工具。
    - Capybara：模拟用户操作进行自动化测试的库。
    - Rspec：使用 Ruby 语言编写的测试框架。
    - Rspec Rails：Rspec 的扩展，用于测试 Web 应用程序。
- 测试的重要性：
    - 测试可以提高软件质量和开发效率。
    - 测试可以帮助开发团队更好地理解用户需求。
    - 测试可以自动化测试用例，节省时间和精力。

8.2: Anatomy of a Test Case - Arrange, Act, Assert

- 测试用例结构 (AAA 模式)
    - Arrange: 设置测试环境，包括初始化数据、配置对象状态等。
    - Act: 执行测试操作，如调用方法、触发事件等。
    - Assert: 验证结果是否符合预期。
- Arrange
    - 非叶方法测试
        - **非叶方法依赖于其他方法完成工作。**
        - 测试时需要控制所有依赖项，确保测试独立性。
        - 存根和模拟技术用于模拟外部服务和依赖项。
    - 测试独立性
        - 测试之间不应相互依赖，可以独立运行。
        - 避免测试依赖于外部状态，如时间、网络服务等。
    - 控制外部条件
        - 测试应控制所有外部条件，避免随机性和不可预测性。
- 测试模型
    - 模型测试通常涉及调用方法并验证输出和副作用。
    - 可以使用 RSpec 和类似工具进行模型测试。
- 测试控制器

    - 控制器测试需要模拟路由系统和请求参数。
    - 断言控制器操作是否正确执行，如渲染视图、重定向等。
    - 至于视图的测试就不是控制器测试该管的事情了。
- 测试视图
    - 视图测试通常由 Cucumber 场景负责，确保用户界面正确性。
    - 可以使用 Capybara 等工具进行视图测试。
- **断言和匹配器**
    - 断言用于验证测试结果是否符合预期。
    - 匹配器是断言的条件类型，如等于、包含、为空等。
    - 负期望用于断言某些条件不成立。
- 安全地处理异常
    - 使用匿名 Lambda 和大括号来安全地处理异常。
    - 避免异常导致测试提前终止。
- 测试哲学
    - 测试应快速、独立、可重复、自检和及时。
    - 测试是敏捷迭代的一部分，应与代码开发同步进行。

8.3: Part 1 - Isolating Code - Doubles & Seams Intro

- **隔离代码的重要性**： 为了更好地测试 Web 应用程序的各个部分，我们需要将代码隔离，以便单独测试每个组件。
- **TMDB 搜索电影功能的示例**： 通过一个具体的例子，解释了如何将逻辑分配到控制器、模型和视图。
- **控制器测试的边界**： 控制器测试应该关注其行为，例如调用模型方法、传递正确的参数、处理结果等，而不应关注底层实现，例如 TMDB 搜索是否有效。
- **用户故事和组件设计**：通过分析用户故事，确定了需要创建的组件，例如搜索表单、控制器操作、模型方法和视图。

8.3: Part 2 - Doubles & Seams In RSpec & Rails

RSpec Rails 的功能

- 提供模拟 HTTP 请求的方法 (get, post, put 等)
- 提供访问响应对象的功能
- 提供断言匹配器，例如检查视图渲染和实例变量分配
- 提供创建 Doubles 的支持

测试示例：添加 TMDB 搜索电影功能

- 用户故事和组件设计
- 创建路由和视图模板
- 编写控制器操作的占位符
- 测试控制器方法调用模型方法
    - 使用 `expect` 和 `receive` 方法模拟方法调用

    - 使用 `expect_any_instance_of` 方法模拟实例方法调用
- 测试控制器选择正确的视图进行渲染
    - 使用 `render_template` 匹配器检查视图渲染
    - 使用 `expect_any_instance_of` 方法模拟方法调用
- 测试控制器将搜索结果传递给视图
    - 使用 `assigns` 方法检查实例变量分配
    - 使用 `expect_any_instance_of` 方法模拟方法调用

接缝的概念

- 定义接缝：在不更改应用程序源代码的情况下修改应用程序行为的地方
- 举例说明接缝的使用：控制模型方法的返回值
- 强调接缝在测试中的重要性：隔离测试和模拟行为

总结

- **重申控制器测试的重点：行为而非底层实现**
- 强调测试的细粒度：每个测试只关注一个行为
- 鼓励使用 RSpec Rails 的功能和 Doubles 进行测试

8.3: Part 3 - Method stubs & mock objects

1. 引言

- 回顾测试控制器操作：确保调用正确的模型方法并传递正确参数。
- 引入新的关注点：返回值和传递给视图的信息。

2. 方法子集和模拟对象

- 两种基本的接缝技术，用于测试代码的特定部分。
- 例子：测试控制器操作是否将返回值传递给视图。

3. 控制器操作和实例变量

- Rails 中控制器传递信息给视图的方式：设置实例变量。
- 使用 `assigns` 方法获取控制器操作中设置的实例变量。

4. 使用模拟对象 (Double)

- 模拟对象在测试中的作用：代替真实对象，仅执行测试所需的最小行为。
- 创建模拟对象：`double` 方法。
- 设置模拟对象的行为：`allow` 和 `expect` 方法。
- 例子：创建模拟电影对象，设置其 `title` 方法返回特定值。

5. 模拟对象的应用场景

- 创建虚假的结果对象，用于测试控制器操作。
- 使用 `allow` 方法设置模拟对象的行为，而不是 `expect` 方法。
- 例子：测试控制器操作是否将模拟电影对象传递给视图。

6. 测试的专注性

- 每个测试都应该**专注于一个行为**，排除无关因素。
- **确保测试失败时，可以明确知道失败原因。**

7. 总结

- 方法子集和模拟对象是测试代码的重要工具。
- 通过模拟对象，可以控制测试环境，专注于测试特定行为。
- 测试应该专注于单一行为，避免无关因素的干扰。

8.4: Stubbing the Internet

1. 面向服务架构的测试挑战

- 介绍面向服务架构的动机：模块化、可组合性。
- 引出测试挑战：如何测试服务间的调用。

2. 测试服务调用的策略

- **方法子集和模拟对象**: 两种基本的接缝技术，用于测试代码的特定部分。
- **控制器操作和实例变量**: Rails 中控制器传递信息给视图的方式。
- **使用模拟对象 (Double)**: 创建模拟对象，设置其行为，代替真实对象进行测试。
- **模拟对象的应用场景**: 创建虚假的结果对象，用于测试控制器操作。

3. 测试的专注性

- 每个测试都应该专注于一个行为，排除无关因素。
- 确保测试失败时，可以明确知道失败原因。

4. 存根不同级别的服务调用

- **电影类级别**: 假设 tmdb API 调用有效，专注于电影类的行为。
- **API 级别**: 返回看起来像 API 将返回的东西，例如使用 movie DB gem。
- **HTTP 级别**: 返回 JSON blob，例如使用 webmock gem。
- **使用 VCR 记录和回放 API 调用**: 第一次运行测试时记录 API 调用和响应，之后使用预装的对象进行测试。

5. 存根的利弊

- **优点**: 隔离测试，专注于特定行为，运行速度快。
- **缺点**: 越远越现实，依赖性越强，运行速度越慢。

6. 不同场景下的存根策略

- **集成测试**: 使用沙盒服务或测试模式进行测试交易。
- **功能测试**: 使用 webmock 确保应用程序对 API 响应的正确性。
- **其他测试**: 覆盖整个支付概念或工作流程。

7. 其他模拟工具

- **Timecop**: 存根系统时间，用于测试与时间相关的行为。
- **Cucumber 步骤定义**: 使用 Timecop 实现时间旅行，模拟特定时间点的行为。
- **选择性存根**: 只对真正需要的测试用例进行存根或伪造。

8. 总结

- 存根是测试服务调用的重要工具。
- **选择合适的存根级别取决于测试目的和场景。**
- 保持测试的专注性，确保测试的独立性和可维护性。

8.6: Part 1 - Fixtures & Factories

引言

- 伪造对象和伪造方法的价值：隔离测试代码。
- 有时需要真实对象：例如，测试格式化方法。

Fixture 的概念

- Fixture 的定义：预先加载到数据库中的数据。
- Fixture 的用途：在测试数据库非空状态下进行测试。
- Fixture 的风险：可能引入对特定数据的依赖。

Factory 的概念

- Factory 的定义：方便地创建特定测试所需的真实对象副本。
- Factory 的优点：**快速构建具有默认值的对象。**
- Factory 的使用场景：对象创建过程复杂，需要关联。

Factory Bot 简介

- Factory Bot 的作用：简化对象创建过程。
- Factory Bot 的使用方法：定义工厂规范，实例化对象。
- Factory Bot 的优势：**提供默认值，可按需覆盖。**

关联工厂

- 关联工厂的用途：处理对象间的关联关系。
- 关联工厂的使用方法：定义关联对象，创建关联关系。
- 关联工厂的优势：简化测试，提高可读性。

Faker 简介

- Faker 的作用：提供各种虚假数据。
- Faker 的使用方法：生成虚假的名称、地址、信用卡号等。
- Faker 的优势：简化测试，提高可读性。

子工厂和工厂变体

- 子工厂的用途：定义具有不同默认属性的对象。
- 工厂变体的用途：返回对象的预定义变体。
- 子工厂和工厂变体的优势：提高测试的灵活性和可读性。

总结

- Fixture 和 Factory 的区别：Fixture 预先加载数据，Factory 创建对象副本。
- Fixture 和 Factory 的适用场景：**Fixture 适用于静态数据**，**Factory 适用于动态数据。**

- Fixture 和 Factory 的优势：简化测试，提高可读性，提高测试的灵活性和可维护性。

8.6: Part 2 - When Fixtures vs. When Factories

- Fixtures 的适用场景：
  - 应用程序运行过程中**不变或很少变化**的数据，例如：
    - TMDB API 密钥
    - 应用程序时区设置
    - 特殊用户账户（例如管理员账户）
  - 需要保持一致性的数据，例如：
    - 模型测试用例中的数据
  - 考虑因素：
    - 测试套件的性质
    - 数据使用的频率
    - 现实世界的约束
- Factories 的适用场景：
  - 测试用例中需要灵活创建和修改的数据
  - 复杂的对象网络
- 避免模拟火车失事：
  - 连续多个点操作模拟对象，导致**对象链**（例如朋友的朋友调用该方法）过长，代码结构混乱
  - 代码设计问题：对象之间耦合度过高，违反封装原则
  - 解决方法：使用设计模式（例如委托）来抽象对象之间的交互

8.7: Coverage

覆盖率的定义和意义

- 测试套件的好坏如何衡量？
- 过去：为了按时发布，追求足够的测试数量。
- 现在：静态测量，例如测试行数除以代码行数。
- 目标：测试行数大于等于代码行数，理想情况下为1.5倍。
- 任务关键型应用程序可能需要更高的测试覆盖率。

测试彻底性的衡量

- **正式方法：分析代码哪些部分可能被覆盖。**更加关注于测试的彻底性而不是多少。
- 覆盖率测量：关注重点。

覆盖率级别

- 100% 覆盖率的含义不明确，需要进一步分析。
- S0：调用每个方法。
- S1：从每个调用站点调用每个方法。
- C0：至少运行每一行代码。
- C1：每个分支在两个方向上执行。

- C1+决策覆盖率：条件中的每个子表达式都被评估为真和假。
- C2：通过代码的每一条可能路径。

不同类型测试的优缺点

- 单元测试：快速，适合处理代码中的小角落情况。
- 功能测试：例如请求规范，需要接触多个组件。
- 集成测试：例如Cucumber场景，涉及多个请求。
- 平衡不同类型的测试用例，避免走极端。

覆盖率工具

- 介于C0和C1之间，通常为C0或C1。
- 查看所有类型测试的覆盖率信息，包括Cucumber场景和RSpec测试。

覆盖率的价值

- 覆盖率是识别代码中未经测试部分的好方法。
- 随着时间的推移，**一些单元测试可能过时**。
- **不要过分关注单元测试或集成测试**，每种测试都可以发现其他测试遗漏的错误。
- 良好的覆盖率信息可以作为指导，告诉您应用程序的哪些部分正在执行。
- 拥有一个好的测试套件是一件令人高兴和美丽的事情。

8.8: Other Testing Concepts

I. 测试概述

- 软件测试的重要性
- 本文目的：介绍测试相关术语，帮助了解测试的整体情况

II. 突变测试

- 定义：在代码中**故意插入错误**，观察测试是否失败
- 作用：发现代码漏洞，即使覆盖率看起来不错
- 应用：芯片 8.5 自动评分器

III. 模糊测试

- 定义：在代码中**随机**输入，观察程序反应
- 作用：测试程序对**非预期输入**的处理能力
- 应用：Web 应用测试，确保程序不会崩溃

IV. DU 覆盖率

- 定义：测试代码中定义define和使用use对的比例
- 作用：评估测试的全面性
- 适用场景：安全性要求高的软件

V. 黑盒测试 vs 白盒测试

- 黑盒测试：不了解内部实现，只关注功能
- 白盒测试：了解内部实现，关注代码逻辑

- 应用：根据测试目的选择合适的测试方法

VI. 测试实践建议

- 红绿重构：TDD 开发方法
- 接缝：隔离代码，方便测试
- 占位符：方便编写测试用例
- 覆盖率：评估测试全面性
- 综合测试套件：多种测试方法结合，提高测试效果

8.11-8.12: Concluding Remarks: TDD vs. Debugging

1. TDD 与传统调试的区别

- TDD **更细粒度，关注特定方法实现**，可以更精确地控制代码路径和测试行为。
- TDD 提供存根、模拟、方法存根等工具，**方便隔离测试。**
- TDD 自动化测试过程，提高效率，获得更好的测试覆盖率。

2. TDD 与传统调试的相似之处

- 需要相同的技能，例如分析代码、理解变量值、确定代码路径等。
- 都是为了发现和修复代码中的错误。
- TDD可以做调试能做的所有事并且更加高效。

3. TDD 的优势

- 快速发现错误：测试套件会随着时间增长，一旦破坏代码，测试套件会立即报告。
- 帮助组织代码和想法：迫使开发者思考如何执行代码，提高代码的可测试性和可维护性。
- 增强团队合作：自动化测试和代码审查机制，确保代码质量。

4. TDD 的挑战

- 初始学习曲线：一开始可能感觉陌生，需要更多时间来编写测试。
- 需要改变习惯：从先写代码后写测试，转变为先写测试后写代码。

5. 测试覆盖率和代码质量

- 测试覆盖率：衡量测试用例覆盖代码的程度，高覆盖率意味着更低的代码破坏风险。
- 代码质量：使用工具如 code Climate 检测代码设计问题和代码异味，提高代码可维护性。

6. 团队合作和组织

- 持续集成：代码推送时自动运行测试，确保代码质量。
- 代码审查：多人审核代码，确保代码符合团队标准。
- 自动化工具：测量测试覆盖率和代码质量，帮助团队保持代码质量。

7. 总结

- TDD 需要信心和耐心，但带来的好处会很快显现。
- TDD 是一种更高效、更可靠的软件开发方法。
- TDD 有助于提高代码质量，增强团队合作。

软件维护

9.1: What Makes Code *Legacy* and How Can Agile Help

I. 遗留代码的价值和重要性

- 遗留代码的定义和误解
- 60%的成本用以维护软件且其中的60%用以给其扩充新功能。
- 遗留代码的价值：软件维护成本和功能需求
- 遗留代码处理技能的重要性

II. 遗留代码的特点

- 仍然有用户需求
- 缺乏文档和测试
- 缺乏测试的危害：修改风险和代码理解困难
    - 测试作为文档和保障

III. 处理遗留代码的方法

- 编辑和**祈祷**：盲目修改的风险
- 覆盖和修改：测试覆盖率的重要性

IV. 敏捷开发在处理遗留代码中的应用

- 确定变化点：设计模式的应用
- **重构：改变代码结构而不改变行为**
- 重构的类型和时机
- 重构的好处：提高测试性和维护性

V. 敏捷开发对遗留代码的价值

- 添加测试：提高测试覆盖率
- 代码重构：使代码更易于测试和维护
- 拥抱变化：适应新功能和新需求

VI. 总结

- 遗留代码的价值和重要性
- 敏捷开发在处理遗留代码中的作用
- 持续改进代码的目标

9.2: Approaching a Legacy Codebase

I. 遗留代码库的挑战

- 面对一个不熟悉的代码库，需要进行功能增强。
- 需要利用敏捷开发的知识来应对挑战。

II. 让代码运行起来

- 使用版本控制创建临时分支，避免对主代码库造成影响。

- 进行必要的修改以适应开发环境，例如配置文件、数据库等。

III. 学习和理解代码库

- **用户故事和文档**：
  - 查看用户故事和集成测试，了解功能需求。
  - 与客户交流，观察使用流程，**创建自己的文档。**
- **应用程序的 BDD 层次**：
  - 了解应用程序的行为和架构之间的关系。
- **领域模型**：
  - 发现应用程序的领域模型，理解实体之间的关系。
  - 使用数据库模式创建实体关系图。
  - 分析类之间的关系和职责。
- **代码和文档**：
  - 运行测试套件，了解功能实现和代码结构。
  - 查看代码与测试的比率，评估测试基础。
  - 收集测试覆盖率信息，关注关键区域的覆盖率。
  - 寻找非正式设计文档，例如图片、线框图等。
  - 查看项目管理板的历史任务和设计文档。
  - 查找嵌入的文档，例如 JavaDoc。

IV. 缺乏测试覆盖时的应对策略

- 讨论如何处理没有足够测试覆盖的代码区域。

9.3: Establishing Ground Truth With Characterization Tests

核心问题： 遗留代码通常缺乏测试，导致难以安全修改和缺乏文档。
解决方案： 通过**特性测试建立应用程序当前工作方式的基线**，从而提高测试覆盖率。
特性测试的意义：

- **建立基线**： 捕获应用程序当前行为，以便在修改时确保不破坏功能。
- **提高覆盖率**： 为应用程序可能需要修改的部分提供测试，降低风险。
- **学习代码**： 通过编写测试来了解代码的内部工作方式。

特性测试的类型：

- **集成测试**： 在应用程序的集成级别上模拟用户行为，例如使用 Cucumber 等工具。
- **单元测试**： 在类级别或单个方法级别上测试代码，需要了解代码的内部细节。

特性测试的步骤：

1. **创建模拟对象**： 使用模拟对象（double）代替真实的对象，以便控制测试环境。
2. **运行测试**： 运行测试并观察应用程序的行为。
3. **分析结果**： 根据测试结果调整模拟对象的行为，并修改测试期望值，直到测试通过。
4. **重复迭代**： 重复以上步骤，直到捕获应用程序的所有关键行为。如何可以捕获当前代码的特性以便在后续确保原先的功能。

注意事项：

- **不要试图改进代码**： 特性测试的目标是建立基线，而不是改进代码。
- **测试风格**： 初期可以采用冗长的命令式场景，后期再进行改进。
- **测试覆盖率**： 特性测试只是提高测试覆盖率的第一步，还需要编写更详细的单元测试。

9.4: Comments & Commits - Tell Me a Story

- 注释的意义：
  - **记录代码中不明显的信息，而非显而易见的内容。**
  - 为经验丰富的程序员提供帮助，假设他们熟悉使用的框架。
  - 解释代码中不变的东西，例如输入数据的有效性假设。
  - **解释不寻常的实现或错误解决方法，以及背后的原因。**
  - 提醒未来开发者注意代码中的脆弱之处或潜在问题。
- 注释的抽象级别：
  - 注释的抽象级别应高于代码本身，关注“是什么”而非“如何做”。
  - 避免冗余注释，例如解释锁的作用或描述方法的功能。
- 提交信息的重要性：
  - 提交信息应解释已修复的错误或所做的更改，以便其他开发者理解代码的演变。
  - 对于解释性注释或错误解决方法，提交信息同样重要，因为它们会影响未来开发者的工作。
- 代码审查与信息获取：
  - 代码审查是获取代码信息的重要途径，包括代码功能和潜在问题。
  - 查看拉取请求的讨论和提交消息可以了解代码的背景和演变过程。
- 信息放置的判断标准：
  - 信息的放置取决于其对当前处理代码的开发者的重要性。
  - 如果信息对理解代码或进行决策至关重要，则应**放入注释或提交信息中**。
  - 一旦问题得到解决或不再需要，应及时清理相关注释和代码。

9.5: Beyond Correctness - Smells, Metrics, and SOFA

主题：

- 代码可维护性和美观性的评估与改进

主要内容：

- **重构的动机**： 不仅仅是为了代码的正确性，还包括提高可维护性和美观性，使代码透明、美观，并揭示设计意图。
- **如何评估代码**： 探讨代码的定性方面（如代码异味）和定量方面（如代码复杂度指标）。
- **代码异味（Smells）**：
  - **SOFA 指导方针**： **简短（Short）、单一职责（One Responsibility）、较少参数（Fewer Parameters）、一致抽象级别（Same Abstraction Level）。**
  - **案例分析**： 通过重构示例展示如何将代码异味转化为清晰、简洁的代码结构。
  - **工具**： 介绍 Reek 工具，用于检测代码异味。
- **代码复杂度指标**：
  - **ABC 复杂度**： 赋值、分支和条件的加权复杂度，用于评估单个方法的复杂度。

    - **圈复杂度**： 代码块中的不同路径数量，用于评估代码的复杂度。
    - **案例分析**： 通过重构示例展示如何降低 ABC 复杂度和圈复杂度。
    - **工具**： 介绍 MetricFu gem 和 Code Climate 服务，用于收集和分析代码复杂度指标。
- **总结**： 通过多种方法评估代码的可维护性和美观性，并利用工具和最佳实践进行改进，以提高代码质量和可维护性。

核心思想：

- 代码异味和代码复杂度指标是评估代码可维护性和美观性的重要工具。
- 重构是提高代码质量的关键，可以帮助我们消除代码异味，降低代码复杂度，并使代码更易于理解和维护。
- 利用工具和最佳实践可以帮助我们更有效地进行代码评估和改进。

9.6: Part 1 - Intro to Method Level Refactoring

主题： 方法级别重构的介绍与实践
主要内容：

- **重构的动机：**
    - 改善代码的指标，如可维护性和美观性。
    - 使代码更透明、美观，并揭示设计意图。
- **评估代码的方法：**
    - 定性方面：代码异味（Smells）。
    - 定量方面：代码复杂度指标，如 ABC 复杂度和圈复杂度。
- **重构示例：微软 Zune 相机代码重构**
    - 问题背景：早期 Zune 模型存在固件错误，导致设备变砖。
    - 重构过程：
        - 重命名变量，提高代码可读性。
        - 提取辅助方法，如 `leap`，用于判断闰年。
        - 提取类，封装日期转换逻辑。
        - 提取更多辅助方法，如 `add_leap_year` 和 `add_common_year`，简化主方法。
        - **编写单元测试，验证每个辅助方法的功能。**
        - 一个 `=` 变砖的故事。
- **重构的益处：**
    - 提高代码可读性和可维护性。
    - 降低代码复杂度，减少潜在错误。
    - 帮助开发者理解代码和设计意图。
- **总结：**
    - **重构是提高代码质量的重要手段**，可以帮助我们消除代码异味，降低代码复杂度，并使代码更易于理解和维护。
    - 利用工具和最佳实践可以帮助我们更有效地进行代码评估和改进。

9.6: Part 2 - Reflections on Refactoring

- 重构的动机：
  - 基于真实代码错误案例，强调重构的重要性。
  - 使用白盒测试（玻璃盒测试）理解方法内部工作原理，识别临界值和非临界区域。
- 重构的目标：
  - 提高代码可读性和可维护性。
  - 降低代码复杂度。
- 重构的益处：
  - 提高代码可读性和可维护性。
  - 降低代码复杂度，减少潜在错误。
  - 帮助开发者理解代码和设计意图。
- 重构的实践：
  - 使用白盒测试**识别临界值和非临界区域。**
  - 重构示例：通过重构示例展示如何消除代码异味，降低代码复杂度，并使代码更易于理解和维护。
  - 编写单元测试，验证每个辅助方法的功能。
- 重构的反思：
  - 重构可以降低复杂性，但**并不一定减少代码的大小。**
  - **更易读、更简单的代码几乎总是好的。**
  - 重构会改变方法的实现，可能导致现有测试失败。
  - 重构过程中需要更新测试套件，确保测试通过。

9.7: The Plan And Document Perspective on Working With Legacy Code

- 软件维护的计划和文档视角
  - 重构作为敏捷开发的一部分
  - 传统自上而下的计划和文档方法中的维护
  - 维护与开发成本相似，但维护通常用于增强而非修复错误
  - **维护人员与开发人员可能分离**，维护团队可能独立于开发团队
  - 维护经理的角色与开发经理类似，但需要维护计划和文档
  - 维护过程中的客户协作和变更控制
  - 客户支持小组和文档团队的角色
  - 紧急变更请求的处理
  - 重构与重新设计的区别
  - 不同的时间尺度上处理变化的重要性
  - 计划和文档方法在特定环境中的适用性
- 维护处理方式的差异
  - 计划和文档方法中的变更请求与敏捷中的用户故事（短）

- P&D中**维护经理与变更控制委员会的角色**
- **敏捷中的团队协作**和故事优先排序
- 敏捷中保持顾客的参与，许多角色由开发团队替代。
- 敏捷维护方法的总结
  - 总是快速、逐步响应
  - 更小粒度的。
  - 随时用于重构、持续部署。

9.8-9.9: Refactoring & Legacy Code - Fallacies, Pitfalls, Conclusions

- 遗留代码和重构的关系：遗留代码和重构是同一枚硬币的两面，代码在三个月内可能成为遗留代码，需要重构以适应新特性。
- **重构的误区**：
  - 重新开始并不总是最佳选择，因为丢弃代码会失去积累的知识和智慧。
  - 重新开始的前50%功能可能不需要花费太多时间，但接下来的40%和最后的10%可能需要更长的时间。
- 重构的谨慎性：
  - 重构应该是唯一要做的事情，**不要添加新功能或特性。**
  - **使用指标作为指导，而不是作为真理。**
- 重构的时间尺度：
  - 做一个大重构会破坏很多测试，导致分支偏离主分支，合并回来很痛苦。
  - 做**少量重构，逐步改进，避免积累不可持续的技术债务。**
- 重构的价值：
  - 确保代码在长期内能够有效地增强，避免积累大量技术债务。
  - 使用好的注释和提交，长期保持代码的良好状态。
- 重构的长期影响：
  - 代码的价值可能在未来的长期使用中体现。
  - 重构可以确保代码的长期可维护性和可读性。
  - 当你离开项目时，良好的代码可以让其他人继续维护和增强。

敏捷团队

10.1 It Takes a Team Two Pizza and Scrum

- **团队工作的重要性：**
  - 软件工程是一项团队运动，团队合作、沟通和协作能力与编程能力同样重要。
  - **团队动力和流程改善的重要性。**
  - **失败的队伍里没有赢家，成功的队伍中没有失败者。**
- **团队规模和动态：**
  - 团队规模在过去几十年中**显著增长**，需要考虑如何组织团队以有效工作。
    - 仅靠交谈不能解决冲突，需要团队管理方法。

        - P&D取决于一位有经验的经理，敏捷?
    - 小团队（两个披萨团队）**更易于管理，拥有更多自主权**，可以更紧密地合作。
    - Scrum 团队模式：
        - 两个披萨团队模型，理想人数为 4-9 人。
        - Scrum 会议（每日站立会议）：
            - 三个简单问题：昨天做了什么，今天打算做什么，有什么障碍?
            - 目标是促进沟通，解决问题，确保团队成员了解彼此的工作。
        - Scrum Master：
            - 团队与外部世界的缓冲，**负责跟踪进度，解决问题，促进团队协作。**
            - **与团队成员一起工作**，建立信任和融洽关系。
        - 产品负责人：
            - **代表客户的声音，确定用户故事优先级。**
            - 与 Scrum Master 独立，**避免利益冲突。**
- **冲突解决：**
    - 原则：确定共同目标，理解对方观点，避免混淆。
    - 英特尔策略：
        - 建设性对抗：**有义务提出技术分歧**，帮助团队避免潜在问题。
        - 不同意和承诺：**即使不同意，也要接受决定**，并推动团队向前发展。
- **Scrum 团队的优势：**
    - 赋予团队权力，**自我组织**，拥有一定结构。
    - 迭代过程，定期 Sprint，确保项目进度。
- **建议：**
    - **定期更换 Scrum Master 和产品负责人**，促进团队成长和多样性。
    - 将 Scrum 原则应用于课堂项目和长期运行的团队。

10.2: Effective Branching part 1 - Branch Per Feature

- **分支的重要性**：
    - 分支使团队合作更简单方便。
    - 每个分支代表一组相关的提交，便于追踪和管理。
    - 分支可以隔离功能开发，避免影响主分支。
- **分支的使用**：
    - 每个功能创建一个新分支。
    - 功能完成后，将分支合并回主分支并删除。
    - 分支可以用于错误修复、重构等。
    - 分支**应尽量保持独立**，避免过多冲突。
- **分支的日常操作**：
    - 创建、切换、编辑、提交、推送分支。
    - 从远程仓库拉取分支。
    - 合并分支。

- **Rebase 和 Pull Request**:
  - Rebase 用于调整分支提交顺序，解决冲突。
  - Pull Request 用于请求代码审查和合并。
  - PR 应包含详细描述和代码更改。
  - PR 应保持较小规模，便于审查。
- **代码审查**:
  - 代码审查是重要的质量控制环节。
  - PR 是启动代码审查过程的好方法。
  - PR 应保持较小规模，便于审查。
  - 代码审查时间随着 PR 规模的增长而增长。
  - PR 应保持较小规模，以便获得更有效的审查。
- **GitHub 和 Git**:
  - Pull Request 是 GitHub 的功能，用于管理代码仓库。
  - GitHub 允许团队协作，进行代码审查和合并。
  - GitHub 代码审查流程与 Google 类似。
  - GitHub 作为版本管理服务后端，提供了代码审查功能。

10.2: Effective Branching part 2 - Branches & Deploying

- **分支策略**
  - 短暂的功能分支
  - 使用 `git rebase` 合并更改
  - 使用 `git cherry pick` 合并特定提交
  - 功能标志工具
- **部署策略**
  - 从主分支自动部署
  - 每个发布一个分支
  - 自动化部署流程
- **处理错误修复**
  - 使用 `git cherry pick` 挑选错误修复
  - 阅读相关书籍获取详细指导
- **GitHub 工作流程**
  - Fork 存储库
  - 在 Fork 上创建分支
  - 完成工作后提交 PR
  - PR 讨论
  - PR 记录设计讨论和推理
- **Git 命令**
  - `git status` 查看仓库状态

    - `git checkout` 切换分支
    - `git reset` 撤销更改
    - `git diff` 查看差异
    - `git blame` 查看文件历史
    - `git log` 查看提交记录
    - 提交哈希和引用
- **Git 的发展**
    - 从个人代码库到团队协作
    - 分布式版本控制系统
    - GitHub 的功能和优势
    - 版本控制系统的演变
- **Git 的价值**
    - 错误处理
    - 软件开发协作
    - 个人项目管理

10.3: Design Reviews, Code Reviews

- **设计评审和代码评审的区别：**
    - 设计评审关注架构和设计模式，确保项目方向正确。
    - 代码评审关注具体实现，确保代码质量。
- **评审的重要性：**
    - 避免错误和遗漏。
    - 提高代码质量。
    - 促进团队协作和知识共享。
- **评审的流程：**
    - 制定议程，明确目标和讨论内容。
    - 提前准备材料，例如文档、代码和测试计划。
    - 记录会议纪要，确保信息可追溯。
    - 确定行动项和责任人，跟踪进度。
    - 安排下次会议时间。
- **评审的类型：**
    - 小型设计评审：频繁进行，讨论解决方案和潜在问题。
    - 正式设计评审：在项目关键阶段进行，确保设计合理。
    - 代码评审：持续进行，确保代码质量。
- **评审的挑战：**
    - 大型评审难以进行，容易造成沟通障碍。
    - **定量指标无法全面评估代码质量。**
- **评审的解决方案：**
    - 采用小型、频繁的评审方式。

    - 结合定量指标和定性评估。
    - 鼓励团队协作和知识共享。结对编程。
- **评审工具：**
    - GitHub Pull Requests：用于代码评审，支持代码注释和建议。
    - 其他工具：根据团队需求选择合适的工具。
- **评审的最终目标：**
    - 构建高质量、可维护的软件。

10.4: Delivering the Backlog Using Continuous Integration

- **持续集成的概念**：
    - 通过工具和流程，帮助团队更快地交付代码。
    - 将开发的功能推送到 GitHub 存储库，并**触发持续集成工具运行测试。**
    - 利用持续集成工具进行测试、代码覆盖率分析、截图等，帮助发现问题。
    - 将代码推送到暂存服务器，供客户测试并提供反馈。
    - 将经过测试和反馈的功能部署到生产环境。
- **持续集成的流程**：
    1. **团队仓库**：托管在 GitHub 或其他版本控制系统。
    2. **本地工作副本**：开发者从团队仓库克隆代码到本地。
    3. **功能分支**：为每个功能创建独立的分支进行开发。
    4. **提交和推送**：将本地分支的代码提交并推送到 GitHub。
    5. **持续集成**：GitHub 触发持续集成工具运行测试。
    6. **代码审查**：团队成员审查代码并提出反馈。
    7. **暂存测试**：将代码推送到暂存服务器进行用户验收测试。
    8. **生产部署**：将经过测试和验收的功能部署到生产环境。
- **持续集成与故事状态**：
    - **故事开始**：创建功能分支。
    - **故事进行中**：进行开发并推送代码到 GitHub。
    - **故事完成**：完成代码审查并获得团队成员的认可。
    - **故事交付**：将代码推送到暂存服务器。
    - **故事接受**：客户验收并同意功能。
- **持续集成的最佳实践**：
    - **小故事**：使分支更简单、更易于审查和部署。
    - **尊重团队优先级**：确保功能以正确的顺序开发。
    - **一次处理一个故事**：避免分支切换和冲突。
    - **可持续的工作速度**：避免临近截止日期才进行大量工作。
- **GitHub Fork 模型**：
    - 开发者可以从团队仓库 Fork 出个人副本进行开发。
    - 将个人分支合并到团队 Fork 中，并在团队 Fork 的暂存环境中进行测试。
    - 将最终代码合并回原始 Fork。

10.6: Fixing Bugs -The Five R's

- **错误处理流程概述**
  - 错误是代码库中不可避免的现象。
  - 修复错误的第一步是处理错误报告，可能来自客户或自动化工具。
  - **重要的是要能够重现或重新分类错误。**
  - 重新分类错误可能意味着它实际上是一个功能增强请求。
  - 如果确认是错误，需要进行回归测试。
  - 修复错误后，需要再次进行回归测试以确保问题已解决。
  - 发布修复后的代码。
- **错误报告和敏捷开发**
  - 错误报告通常需要将外部报告转换为进入跟踪系统的形式。
  - 在敏捷开发中，错误报告不会对故事跟踪计分，以避免重复计算工作量。
  - 应该使用自动化工具来简化错误报告和处理流程。
- **错误跟踪工具**
  - 可以使用 Bugzilla 等专门的 Bug 跟踪软件来管理错误。
  - 应该选择最适合团队的工具和流程，避免过度复杂化。
- **错误分类和修复**
  - 有些错误可能不会被修复，例如因为时间优先级、修复难度或潜在的风险。
  - 应该与客户沟通不修复错误的原因。
  - 最难的部分是用最简单的步骤重现错误。
  - 在复杂系统中，可能需要深入分析堆栈的不同层级。
  - 闰日、闰秒等时间相关的错误可能需要特殊的处理方法。
  - 设置测试用例来重现错误，并确保在修复后测试用例能够通过。
- **自动化和持续集成**
  - 使用自动化工具来简化错误处理流程。
  - 持续集成可以确保代码更改不会引入新的错误。

10.8-10.9: Fallacies, Pitfalls, and Concluding Remarks on Teams

- **谬误和陷阱**
  - **按软件堆栈划分工作**： 在小团队中，全栈方法更有效，**减少协调成本。**
  - **意外破坏更改**： 使用 Git 等工具，养成良好习惯，如及时提交、更新代码库等。
  - **在主分支上直接更改**： 创建功能分支，避免直接在主分支上操作。
- **关于团队**
  - **两个披萨团队**： 保持团队规模小，便于沟通和协作。
  - **角色和责任**： 团队中仍需有人负责确定优先级、解决冲突等。
- **关于敏捷开发**
  - **故事点、速度和故事**： 用于评估工作量和预测进度，但不应作为成功或失败的指标。

    - **迭代评估**： 评估项目进展，记录经验教训，不断改进。
- **关于代码和测试**
    - **测试用例**： 确保测试用例有效，避免无效测试。
    - **代码指标**： 选择合适的指标，如代码复杂度、分支寿命等，并制定相应的规则。
    - **沟通和协作**： 及时沟通，寻求帮助，避免让队友失望。

设计模式

11.1: Patterns, Antipatterns, and SOLID Class Architecture

- **设计模式的目标和起源**
    - **促进重用**，解决应用程序中的挑战
    - 来自架构设计模式，历史悠久
    - 模式语言或集合，帮助解决类似问题，**起点和模板。**
- **设计模式的种类**
    - 架构模式 (宏观模式)：HTTP 堆栈、分层模型等
    - 特定模式 (微观模式)：快速傅里叶变换、线性代数等
    - 软件设计模式 (面向对象代码)
        - **四人帮** (GoF) 书籍：结构化、创建性、行为性
        - 23 种设计模式：工厂、适配器、观察者等
- **设计模式的特点**
    - **模式不是实现，而是指导**
    - 模式不特定于语言
    - 模式动物园或分组
- **设计模式的类别**
    - 行为模式：观察者、中介者等
    - 创建模式：工厂、抽象工厂、构建器等
    - 结构模式：适配器、代理、装饰器等
- **设计模式的理念**
    - 将变化与不变分开
    - **程序到接口，而非实现**
    - **组合优于继承**
- **反模式**
    - 接近但不完全的设计模式
    - 不必要的复杂性
    - 技术债务的积累
- **面向对象原则 (SOLID)**
    - 单一责任原则
    - 开放/封闭原则
    - 里氏替换原则

    - 接口隔离原则
    - 依赖倒置原则
- **设计模式的应用**
    - 意识到设计模式的存在
    - 在代码挑战中参考设计模式
    - 使用工具解决问题

11.2: Just Enough UML

- **UML 介绍**
    - UML 作为面向对象系统建模语言
    - UML 用于描述类之间的关系
    - UML 图类型概述
    - 真正要了解的是类之间的关系，所以不需要将所有属性列出。
- **UML 图示例**
    - 汽车和发动机的类关系
    - Armando 受众优先应用程序的类关系
    - 类关系图元素解析：空心圆 (聚合), 菱形 (组合), 箭头 (继承)
    - 关系数量表示：零或一, 多个
- **UML 图工具**
    - 自动生成 UML 图的工具
    - 类责任协作者卡 (CRC 卡) 的作用
- **CRC 卡示例**
    - 电影放映类与订单、票务模型的关系
    - CRC 卡用于规划模型关系和用户故事
- **用户故事示例**
    - 将电影票添加到订单的用户故事
    - 从用户故事中识别模型、对象和操作
    - CRC 卡和用户故事帮助指导应用程序设计

11.3: Single Responsibility Principle

- **单一职责原则 (SRP) 简介**
    - SRP 是 SOLID 原则之一，关注如何设计类。
    - **一个类应该只有一个改变的原因，即只有一个职责。**
    - 类的职责可以用简短的描述来概括。
    - 与 SRP 相反的是“上帝对象”，拥有过多职责的类。
- **评估类职责的工具**
    - **方法缺乏凝聚力 (Lack of Cohesion of Methods, LCOM)**
        - 通过比较实例方法数量、实例变量数量和变量调用次数来评估类凝聚力。
        - LCOM 值越高，类职责越分散。

    - **类图连接组件**
        - 通过分析类中方法之间的依赖关系来评估类凝聚力。
        - 连接组件越多，类职责越分散。
- **Active Record 模型与 SRP**
    - Active Record 模型通常包含多个职责，例如数据管理、关系管理、验证等。
    - Active Record 通过模块将部分行为封装起来。
    - 在 Rails 应用中，关键是如何合理地使用模型。
- **在 Rails 中提取类**
    - **基于关系提取**
        - 例如，从 `MovieViewer` 中提取 `Address` 类。
    - **评估是否需要新的数据库模型**
        - 代码重构需要考虑实际意义。
    - **创建不依赖数据库的模型**
        - 用于封装业务逻辑，例如权限管理。
    - **关注测试的复杂性**
        - 如果测试需要模拟大量对象，可能需要提取类以降低耦合度。
- **总结**
    - 意识到类可能变得过大是重要的。
    - 合理使用工具评估类职责。
    - **根据具体情况提取类，降低耦合度，提高代码可维护性。**

11.4: Open/Closed Principle

- **目标**： 寻找模式和大创意，而**不是记住所有术语和设计模式。**
- **开放和封闭原则 (OCP)**: 类应该**对扩展开放，但对修改关闭。**
- **场景**： 需要生成不同格式的报告 (HTML, CSV, PDF, JSON)。
- **问题**： 如何在**不修改现有代码的情况下添加新的报告格式**?
- **解决方案**:
    - **抽象工厂模式**： 在运行时**动态查找不同类型的类。**
    - **模板方法模式**： 一组相同的步骤，**但每个步骤的实现不同。**
    - **策略模式**： 相同的任务，但有多种可能的实现。
- **Ruby 中的例子**:
    - **抽象工厂**： 使用 `constantize` 方法动态创建类实例。
    - **模板方法**： 报告生成示例，每种报告类型实现相同的步骤。继承，调用特定子类的方法。
    - **策略模式**： OmniAuth gem 中的身份验证策略。实现，调用特定实现的方法。
- **核心思想**:
    - **组合优于继承**： 使用小类组合构建更大的功能。
    - **装饰器模式**： **通过包装对象来扩展类的功能。**组合的代表。
    - **作用域**： 共享查询，无需在模型中重新实现。
- **实践建议**:

    - 考虑未来可能需要的更改，但不要过度设计。
    - 识别需要修改的代码，并创建接口以便扩展。
    - 逐步重构，将现有类分解为更小的、可扩展的类。

11.5: Liskov Substitution Principle

- **Liskov 替换原则简介**
    - 定义：子类应该能够替代父类在系统中使用，而不影响系统的行为。
    - 意义：避免过度使用继承，防止子类破坏父类的预期行为，不要过分覆盖父类。
- **Liskov 替换原则的实践意义**
    - 避免创建违反原则的子类，导致系统行为异常。
    - 使得子类可以更容易地替换父类，提高系统的可扩展性和可维护性。
- **违反 Liskov 替换原则的例子：正方形和矩形**
    - 正方形是矩形的子类？将宽度设置为高度两倍就不再是正方形。
    - 正方形是特殊的矩形，但某些操作（如将宽度设置为高度的两倍）对正方形不适用。
    - 解决方法：使用组合代替继承，将共通的功能抽象成独立的类，供子类使用。
- **其他违反 Liskov 替换原则的情况**
    - 子类改变了父类方法的预期行为。
    - 子类对父类方法的输入或输出进行了限制。
    - 子类抛出了父类方法未定义的异常。
- **避免违反 Liskov 替换原则的工具**
    - **设计模式**：例如，使用组合代替继承，使用策略模式等。
    - **接口和抽象类**：定义清晰的接口和抽象类，避免子类破坏父类的约定。
    - **单元测试**：通过单元测试验证子类是否满足父类的预期行为。

11.6: Part 1 - Dependency Injection

- **依赖注入 (DI)**：
    - **依赖倒转**的概念
    - 类 A 依赖类 B，但 B 的实现可能变化，如何处理？
    - 例子：活动记录与不同数据库的交互
    - 解决方案：**注入抽象接口**，**A 和 B 都依赖于底层类**
    - Ruby 中的模块提取共享行为
    - Rails 中的会话存储机制：cookie、数据库等
- **适配器模式**：
    - 类似于策略模式，但更通用
    - 将不同类型的接口整合成一致的 API
    - 例子：Active Record 中的数据库适配器
    - 优点：
        - 处理多种类型的数据库，继承。
        - 应对外部服务变化

        - 利用现有代码
- **外观模式**：
    - **简化复杂的 API**，提供通用功能
    - 例子：Active Record 中的适配器
    - 优点：
        - 简化接口
        - 适应服务变化
        - 处理常见功能

11.6: Part 2 - More Adapter-Like Patterns

- **介绍**
    - 本文将介绍一些"适配器-like"的设计模式，它们有助于解决设计原则或挑战，但可能不会直接出现在固定首字母缩略词中。
- **空对象模式**
    - 定义：创建一个代表空实例的类，以简化应用程序设计。
    - 应用场景：Rails 应用程序中的当前用户，代表已注销用户。
    - 优点：减少了对空对象检查的需要，简化了代码。
- **单例模式**
    - 定义：保证一个类只有一个实例，并提供一个全局访问点。
    - 实现：Ruby 提供了多种方法实现单例模式，例如使用类方法。
    - 作用：组织功能，确保类只有一个实例。
- **代理模式**
    - 定义：控制对对象的访问，并提供额外的功能。
    - 与适配器模式的区别：代理模式**更直接地修改给定接口的行为**，而适配器模式则**转换接口**。
- **外观模式**
    - 定义：简化对复杂系统的访问。
    - 与代理模式的区别：外观模式主要用于简化，而代理模式则控制访问并提供额外功能。
- **总结**
    - 这些"适配器-like"模式可以帮助我们解决设计原则或挑战，并简化应用程序设计。

11.7: Demeter Principle

- **原则概述**
    - 原则思想：只与你的朋友交谈，不要与陌生人交谈。
    - 应用：调用直接属于自己类的方法或实例方法，避免调用相关对象的方法。
    - 滥用：Ruby 等语言中容易滥用，导致**长方法链**。
    - 解决方案：定义具体接口，使用委托方法、访问者模式、观察者模式等。
- **委托方法**
    - 通过组合分离接口，定义单一方法作为接口。
    - 例子：电影观众类委托钱包类进行信用余额操作。

- **行为角度思考**
  - 不仅从代码角度，更要从完成任务的角度思考。
  - 定义更好的接口，例如钱包类的存款和取款方法。
- **观察者模式**
  - 允许类观察其他类的变化并执行操作。
  - 应用场景：发送邮件、记录事件、安全相关任务等。
  - 优势：避免污染模型，提供通用接口。
  - 实现：Rails 提供多种钩子实现观察者模式。

11.8: Plan And Document Perspectives on Design Patterns

- **计划和文档体现设计模式概念**
  - 预先设计：在需求明确的情况下，设计模式可以提前规划，例如使用软件需求规范来确定类和接口，以便应用特定设计模式。
  - 持续重构：在敏捷开发中，设计模式可能在编码过程中逐渐发展。通过持续重构，代码可以不断优化，以利用有意义的设计模式。
- **计划和文档方法与敏捷方法**
  - 传统方法：预先进行详细的规划和文档，有助于提前识别设计模式并应用。
  - 敏捷方法：鼓励在编码过程中不断探索和重构，设计模式可能随着项目发展而逐渐显现。
  - 平衡：在敏捷开发中，可以根据经验和对项目需求的了解，提前规划一些设计模式，但不必过度规划。
- **设计模式的适用性**
  - 模式抽象性：设计模式往往更抽象，并非所有模式都直接适用于特定领域或语言。
  - 框架启示：框架可以提供对应用程序中可能出现问题的不同思考方式，并启发设计模式的应用。
  - 模式选择：根据项目需求和挑战，选择合适的模式，例如身份验证、数据库存储、扩展性等。
- **扩展和 DevOps**
  - 预见性：考虑应用程序可能的扩展需求，并提前准备相应的模式。
  - DevOps 实践：第 12 章将讨论扩展和实现 DevOps 的方法，以支持应用程序的长期发展。

Dev/Ops

12.1: From Development to Deployment

- **开发与部署的差异**：
  - 用户行为：普通用户会以开发人员意想不到的方式使用软件，导致稳定性问题和漏洞暴露。
  - 环境差异：开发环境和生产环境可能存在规模、软件类型等方面的差异，导致潜在问题。
  - 非预期使用：应用程序可能会被用作恶意软件载体或参与拒绝服务攻击，需要防范。
  - 用户误操作：用户可能会输入错误数据或进行误操作，需要考虑容错和异常处理。
- **部署方式的演变**：
  - 早期：需要配置和管理自己的虚拟专用服务器，**安装和配置所有软件**，处理安全问题，自动化水平扩展等。

    - 现代：平台即服务 (PaaS) 如 Heroku 简化了部署过程，提供了基础设施管理、安全性、性能调优等功能，**让开发者可以专注于应用程序本身。**
- **平台即服务的优势**：
    - 简化部署：无需担心基础设施配置和管理，只需关注应用程序开发。
    - 提高效率：PaaS 提供了自动化和优化，让开发者可以更高效地利用硬件资源。
    - 降低成本：对于小型应用程序，PaaS 可以提供经济高效的部署方案。
- **性能和安全性**：
    - 可用性/正常运行时间：应用程序可访问的时间百分比。
    - 响应性：用户操作后获得响应的时间。
    - 可扩展性：随着用户数量增加，能否保持响应水平。
    - 隐私：数据访问权限的控制。
    - 身份验证：验证用户身份，确保用户权限的正确性。
    - 数据完整性：防止数据被篡改或损坏。
- **PaaS 与自建部署**：
    - PaaS 提供了许多基础设施级别的功能，但开发者仍需理解相关概念和技术。
    - 即使使用 PaaS，开发者也需要关注应用程序层面的性能和安全性。

12.2: Three Tier Architecture

- **三层架构概述**
    - 动态内容生成和软件即服务
    - 从**静态网页**到**动态应用程序**的转变
    - 架构模式的形成：观察成功系统，提取共同点
- **三层架构的组成**
    - **表示层**：Web 服务器，负责接受请求和发送响应
    - **应用程序层**：应用服务器，运行实际应用程序逻辑
    - **存储层**：数据库，存储数据并提供持久化
- **部署示例**
    - **本地开发环境**：所有层运行在同一台机器上
    - **中等规模部署**：三列结构，Web 服务器、应用服务器和数据库分离
    - **大规模部署**：更专业化的组件，例如 Heroku 部署
- **数据库扩展性**
    - **分片**：将数据划分到不同的数据库中，扩展性好但跨数据库操作困难
    - **复制**：在多个数据库副本之间复制数据，提高冗余和可用性
- **架构模式的灵活性**
    - 不同规模的部署可以采用不同的架构
    - 数据库扩展需要考虑分区、复制等策略
    - 许多技术可以帮助提高部署的投资回报率

12.3: part 1 - Availability & Responsiveness - An Introduction

- 可用性的定义和衡量
  - 正常运行时间百分比
  - 固定电话时代的可用性标准
  - Web 应用程序的可用性挑战
  - 可用性的不同评估方法
- 响应时间的重要性
  - 亚马逊、雅虎和谷歌的实验
  - 用户对响应时间的感知
  - 100 毫秒的瞬时响应阈值
  - 7-8 秒的放弃时间阈值
  - 谷歌的“速度是一项功能”理念
  - 谷歌开发者网站上的速度分析工具
- 响应时间分布
  - 非正态分布的响应时间
  - 平均值和中位数与 95% 百分位数的差异
  - 关注大多数用户的体验

12.4: Part 1 -Continuous Integration & Continuous Deployment

- 软件发布方式的转变：从盛大的活动到频繁的更新
- 持续集成 (CI) 和持续部署 (CD) 的兴起
- CI 和 CD 的优势：
  - 降低风险，快速发现和修复问题
  - **自动化流程，提高效率**
  - 确保代码质量
- CI 的核心：
  - 集成测试，验证代码集成后是否正常工作
  - 自动化测试，减少手动测试工作量
  - 测试环境与生产环境相似，提高测试准确性
  - 跨浏览器测试，确保应用在不同浏览器中表现一致
  - 针对服务沙箱的测试，模拟真实环境进行测试
  - 安全测试，评估应用的安全性
  - 压力测试，评估应用在高负载下的表现
- CD 的核心：
  - 自动化部署，无需人工干预
  - 快速发布新功能，缩短上市时间
  - 降低发布风险，确保应用稳定运行

- 功能发布的概念：
  - 客户可见的重大更新
  - 需要仔细规划和沟通
  - 可以使用 Git tag 等工具进行版本管理
- 测试是流程的一部分，如果对测试有信心，部署也会是流程的一部分。

12.11-12.12: DevOps Fallacies, Pitfalls, and Concluding Remarks

DevOps 的起源与核心概念

- 开发与运维的交叉与融合
- 平台即服务 (PaaS) 的影响
- DevOps 作为开发与运维结合的边界

性能优化误区与正确做法

- 过早或不明智的优化
- 监控的重要性，利用监控找出如何获得最大的利益。
- **水平扩展**与单台机器性能
- 设计避免性能瓶颈，**高性能的硬件无法拯救糟糕的设计。**
- 数据库优化策略，善用监控。
- 最大限度利用现有资源，比如PaaS，不要花时间在那些不会增加软件价值的事情上。

安全性误区与最佳实践

- 安全性并非无关紧要
- 黑客攻击的目标
- 事后增加安全性的困难
- 使用最佳实践和工具
- 灾难恢复计划与演练

平台即服务的优势

- 资源利用最大化
- 避免管理基础设施
- 数据库友好的抽象
- 保持最新状态
- 灾难恢复

总结

- DevOps 的核心在于开发与运维的融合
- 性能优化应避免过早进行，并注重设计
- 安全性是网站运营的重要方面
- 平台即服务提供许多优势