山东大学软件工程协作开发补充

组员：杨逸，黄德业，刘钊，谢骁飏，闫威

指导老师：余仲星

目录

实验 2

1.

CASE 工具

CASE（Computer-Aided Software Engineering，计算机辅助软件工程）工具调研及应用

小组分工搜索各种主流软件工程技术网站，调研有哪些流行的 CASE 工具 （如教材中提到的甘特图等），分析它们的用途、技术特点。

搜索各种主流软件工程技术网站，调研有哪些流行的 CASE 工具 （如教材中提到的甘特图等），分析它们的用途、技术特点

https://blog.csdn.net/Black_8/article/details/94844429

https://www.cnblogs.com/guojin705/archive/2011/05/22/2053863.html

https://www.tutorialspoint.com/ch/software_engineering/case_tools_overview.htm

实验 3

1.

讨论传统软件开发过程模型与敏捷开发（中几种主要方法）的比较

小组分工讨论传统软件开发过程模型与敏捷开发（中几种主要方法）的比较，分

析各自的优缺点，以及如何应用于自己的项目中？并且分析自己项目中可能存在的风险，细化风险管理（做出风险分级及应对预案）。

传统软件开发过程模型和敏捷开发是两种不同的方法论，各自具有一系列的优点和缺点。下面是它们的比较以及如何应用于人事管理系统项目的分析：

1. 传统软件开发过程模型（如瀑布模型）：
   优点：
   - 清晰的阶段和任务划分，适合大型、复杂项目。
   - 易于管理和跟踪进度。
   - 强调完整的文档和规范，有利于团队合作和知识传承。

   缺点：
   - 刚性的计划和阶段划分，缺乏灵活性。
   - 需求固化后难以变更，容易导致需求脱离实际。
   - 风险管理相对较少，问题可能在后期才被发现。

   在人事管理系统项目中的应用：
   - 需求稳定且明确，适用于传统模型。
   - 可以在项目一开始就进行详细的需求分析和规划。
   - 需要确保项目的需求和范围被准确理解和定义，避免后期变更的风险。

2.
阅读 Scrum 开发方法文档，理解 Scrum 过程工作模型

敏捷开发（如 Scrum 和 XP）：
   优点：
   - 强调迭代和增量开发，适应需求的变化。
   - 更高的灵活性和响应能力，快速交付可用的软件。
   - 鼓励团队协作和客户参与，增加客户满意度。

   缺点：
   - 对团队成员的技能和经验要求较高。
   - 需要更多的沟通和协调工作。
   - 可能会牺牲一些文档和规范的完整性。

   在人事管理系统项目中的应用：
   - 适用于需求不稳定、变化频繁的项目。
   - 可以在迭代中逐步开发和交付系统功能。
   - 需要与客户进行紧密的合作和反馈，确保系统符合其需求和期望。

风险分级和应对预案：

1. 风险分级：

- 高风险：可能对项目进度和成果产生重大影响。
- 中风险：可能对项目进度和成果产生一定影响。
- 低风险：可能对项目进度和成果产生较小影响。

2. 风险管理预案：
- 高风险：对高风险风险进行详细风险分析，制定具体的应对措施，并提前预留充足的缓冲时间和资源。
- 中风险：对中风险风险进行适度的风险分析，制定灵活的应对方案，并与团队共享风险意识和应对策略。
- 低风险：对低风险风险进行简单的风险识别，并建立一些常规性的措施来应对可能出现的问题。

在人事管理系统项目中可能存在的风险包括：
- 需求变更：客户对需求的变更可能导致项目范围扩大或调整，影响项目进度和交付。
- 技术风险：使用新技术或平台可能带来技术上的挑战和不确定性。
- 人员风险：团队成员的离职或能力不足可能影响项目的进度和质量。
- 交付风险：部署和交付过程可能面临问题，如系统配置、数据迁移等。

针对这些风险，可以采取以下措施：
- 与客户保持紧密的沟通和协作，及时识别和处理需求变更。
- 在项目初期进行技术验证和风险评估，寻找解决方案。
- 预留适当的人力资源，确保团队成员的稳定性和能力满足项目需求。
- 进行充分的测试和交付准备，确保系统的稳定性和可靠性。

通过风险管理的实施，可以在项目中及早识别和应对潜在风险，减少其对项目进度和质量的影响。

2.
阅读 Scrum 开发方法文档，理解 Scrum 过程工作模型
Scrum 过程工作模型包括以下步骤：

1. 产品待办清单：定义产品待办清单，即要开发的所有功能和需求列表。
2. 冲刺计划会议：在冲刺计划会议上，团队选择要在下一个冲刺中完成的功能，并制定实现这些功能的计划。
3. 冲刺：在冲刺期间，团队按照计划执行工作，并每天进行短暂的站立会议以检查进度和协调工作。
4. 每日站立会议：每天进行短暂的站立会议以检查进度和协调工作。
5. 冲刺评审会议：在冲刺评审会议上，团队展示他们已经完成的工作，并接受利益相关者的反馈和建议。
6. 冲刺回顾会议：在冲刺回顾会议上，团队回顾他们已经完成的工作并讨论如何改进下一个冲刺。
7. 产品增量：在每个冲刺结束时，团队应该交付一个可用且完整的产品增量。

实验 4

1.

XP 过程工作模型

阅读 XP 开发方法文档，理解 XP 过程工作模型

XP（eXtreme Programming）是一种敏捷软件开发方法，它强调团队合作、迭代开发和持续交付。XP 采用一种称为"XP 过程工作模型"的方法来组织和管理软件开发过程。下面是 XP 过程工作模型的关键要素：

1. 用户故事：
   - XP 使用用户故事来描述系统的功能需求。用户故事是从用户的角度描述系统功能的简短描述，通常以用户目标或需求的方式编写。
2. 计划：
   - XP 的计划是以迭代为基础的。在每个迭代开始之前，团队和客户一起决定要完成的用户故事，并将其放入迭代计划中。迭代通常持续 2-4 周。
3. 设计：
   - XP 鼓励团队进行简单且持续的设计。设计是一个与开发任务同时进行的过程，团队根据当前需要进行设计，并在需要时进行重构。
4. 编码：
   - XP 鼓励团队成员进行配对编程（Pair Programming）。两名开发人员共同参与一个任务的编码过程，一个负责编写代码，另一个负责即时代码审查和提供反馈。
5. 测试：
   - 测试在 XP 中是一个持续且重要的活动。每个迭代中的用户故事都需要进行测试，并且团队会编写自动化测试用例来支持持续集成和回归测试。
6. 集成与交付：
   - XP 倡导频繁的集成和交付。团队通过持续集成实践，将各个开发人员的代码整合到主干代码库中，并使用自动化构建和部署工具来实现持续交付。
7. 反馈与改进：
   - XP 鼓励团队通过持续反馈来改进开发过程。团队会定期回顾迭代结果和过程，以识别问题、改进实践并制定适当的行动计划。

XP 过程工作模型强调团队合作、灵活性和快速交付。它通过迭代开发、持续测试和持续集成等实践，帮助团队在不断变化的需求和环境中快速适应和交付高质量的软件。

2.

DevOps

阅读 DevOps 文档，了解 DevOps

DevOps 是一种软件开发和运维的方法论，旨在通过加强开发团队和运维团队之间的合作与沟通，实现软件开发、测试、交付和运维的协同化和自动化。以下是

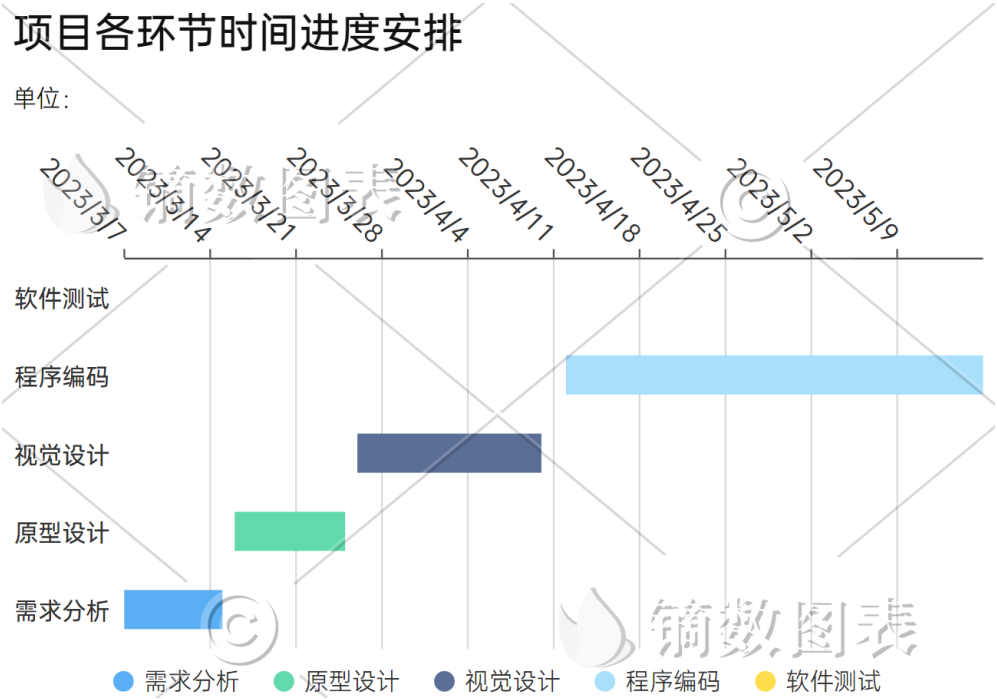
DevOps 的关键概念和实践：

1. 文化：
   - DevOps 强调开发团队和运维团队之间的协作和共享责任。它倡导打破组织内部的壁垒，营造一种文化氛围，促进团队成员之间的合作、沟通和信任。
2. 流程：
   - DevOps 通过优化软件开发和交付流程，提高效率和质量。它鼓励采用持续集成、持续交付和持续部署的实践，使软件的开发、测试和部署过程更加自动化和可靠。
3. 自动化：
   - 自动化是 DevOps 的核心原则之一。通过自动化工具和脚本，可以实现代码构建、测试、部署和配置的自动化，减少人工操作，提高效率和可靠性。
4. 软件质量：
   - DevOps 关注软件的质量和稳定性。通过持续集成和自动化测试，可以及早发现和解决问题，确保交付的软件具有高质量和可靠性。
5. 监控与反馈：
   - DevOps 强调对应用程序和基础设施进行实时监控，并及时收集反馈信息。通过监控和日志分析，可以及时发现和解决问题，并优化系统性能。
6. 基础设施即代码：
   - DevOps 鼓励使用基础设施即代码（Infrastructure as Code）的概念。通过使用代码来定义和管理基础设施资源，可以实现基础设施的版本控制、自动化和可重复性。
7. 容器和微服务：
   - DevOps 推崇使用容器化和微服务架构来实现应用程序的灵活性和可伸缩性。容器化可以实现应用程序的快速部署和隔离，而微服务架构可以使应用程序更加模块化和可扩展。

DevOps 的目标是加强开发和运维团队之间的合作，促进软件交付的速度、质量和稳定性。它通过文化、流程、自动化和工具的综合应用，使软件开发和运维变得更加协同和高效。

实验 5

一个简单的甘特图例子
练习项目跟踪工具的使用，如用甘特图记录跟踪项目过程。



实验 6

1.
工作量估算：
ch3 习题 12
生产率的测量是一个复杂的问题，而仅仅使用代码行数来衡量生产率并不一定合适。

1. 不同编程语言的影响：不同编程语言在表达同样的逻辑和实现相同功能时，所需的代码行数可能会有很大差异。因此，将代码行数作为唯一的生产率度量标准可能会忽略了不同编程语言的特性和差异。
2. 测量前的设计阶段：在软件实现开始之前，通常需要进行设计阶段。在这个阶段，开发团队需要进行需求分析、架构设计等工作，这些工作并不能直接通过代码行数来衡量。因此，在实现开始之前，使用基于代码行的生产率进行测量并不可行。
3. 堆积代码的问题：使用代码行数作为生产率的唯一指标可能会导致程序员为了达到生产率目标而堆积代码，编写冗长而复杂的代码。这种做法可能会降低代码的可读性、可维护性和质量，从而对项目产生负面影响。

综上所述，单纯地使用代码行数来测量生产率并不是一个全面和准确的方法。生产率的测量应综合考虑其他因素，如功能完成度、质量指标、交付时间等。此外，团队应该注重代码的质量、可读性和可维护性，而不仅仅关注代码行数的增加。

2.
风险管理
ch3 习题 11

对于一个学生软件开发项目，以下是一些可能的风险因素和相应的风险暴露以及缓解技术：

时间管理风险：
风险暴露：项目可能无法按时完成，导致延期交付。
缓解技术：制定详细的项目计划，包括任务分解、里程碑和时间估算。使用项目管理工具来跟踪进度和管理任务。

资源限制风险：
风险暴露：缺乏足够的人力资源、技术设备或开发工具。
缓解技术：合理评估项目所需资源，并确保资源的可用性。合理规划和分配团队成员的工作负载。
需求变更风险：
风险暴露：需求在项目过程中频繁变更，导致范围蔓延和进度延误。
缓解技术：进行充分的需求分析和规划，确保对需求的理解和共识。实施变更控制机制，确保任何需求变更都经过适当的评估和批准。

技术挑战风险：
风险暴露：遇到技术复杂性、缺乏经验或新技术的挑战，导致开发困难。
缓解技术：进行技术评估和风险分析，识别潜在的技术难题。提前学习和研究相关技术，寻求专家的指导和建议。

沟通和协作风险：
风险暴露：团队成员之间的沟通不畅，协作不协调，导致误解和冲突。
缓解技术：建立清晰的沟通渠道，定期召开会议和讨论，确保团队成员之间的有效沟通。使用协同工具和版本控制系统来促进协作和文档管理。

测试和质量风险：
风险暴露：不充分的测试和质量保证，导致软件存在缺陷和稳定性问题。
缓解技术：制定全面的测试计划和策略，包括单元测试、集成测试和系统测试。进行代码评审和质量检查，确保高质量的交付物。

实验 7

1.
搜集“软件需求规格说明 SRS”编写案例
搜集“软件需求规格说明 SRS”编写案例

https://software-system-analysis-and-design.github.io/SE-308/docs/Software_Requirements_Specification.html
https://github.com/sysu-abi/docs/blob/master/%E8%BD%AF%E4%BB%B6%E9%9C%80%E6%B1%82%E8%A7%84%E6

%A0%BC%E8%AF%B4%E6%98%8E%E4%B9%A6%EF%BC%88SRS%EF%BC%89.md

实验 8

1.
阅读“SYSTEM MODELLING WITH PETRI NETS”，进一步学习 Petri 网知识，了解如何应用 Petri 网对系统进行建模

Petri 网是一种图形化的数学工具，用于建模和分析并发系统、并行系统和分布式系统等。它由卡尔·亨利克·彼得里（Carl Adam Petri）于 20 世纪 60 年代提出。Petri 网具有以下基本元素：
1. 位置（Place）：表示系统中的状态或条件。
2. 过渡（Transition）：表示系统中的事件或活动。
3. 弧（Arc）：连接位置和过渡，表示条件和活动之间的关系。弧分为输入弧和输出弧。
4. 标记（Marking）：指示位置中所包含的标记数量，表示系统的状态。

Petri 网的建模过程通常包括以下步骤：
1. 确定系统的组成部分和交互方式：确定系统中的位置、过渡以及它们之间的关系。
2. 绘制 Petri 网：使用图形工具绘制 Petri 网，将位置、过渡和弧表示为适当的图形元素。
3. 添加标记：根据系统的初始状态，将适当数量的标记分配给位置。
4. 定义弧和过渡的触发条件：将弧和过渡之间的关系定义为输入弧和输出弧，并指定触发条件。
5. 分析和验证：使用 Petri 网模型进行分析和验证，如死锁分析、活动性分析、性能分析等。

实验 9

1.
UML 对系统进行建模步骤
阅读“The Unified Modeling Language Reference Manual”，进一步学习 UML 知识，理解如何应用 UML 对系统进行建模

1. 确定建模目的：明确建模的目标和目的。确定需要建模的系统的范围和边界，

以及所关注的关键方面。
2. 选择适当的 UML 图表：根据建模目的选择合适的 UML 图表。UML 提供了多种图表，包括用例图、类图、时序图、活动图等，每种图表都有不同的应用场景和表示能力。
3. 定义系统的结构：使用类图和对象图描述系统的静态结构。识别系统中的主要类、接口、关系和属性，并进行适当的分类和组织。使用类图表示类之间的关系，对象图表示类的实例和实例之间的关系。
4. 描述系统的行为：使用活动图、时序图和状态图等来描述系统的动态行为。活动图描述业务流程和操作的流程，时序图描述对象之间的交互顺序，状态图描述对象的状态和状态转换。
5. 识别系统的用例和功能：使用用例图和活动图来识别系统的用例和功能。用例图描述系统的功能需求和用户角色，活动图描述用例的执行流程和操作步骤。
6. 建立交互和通信：使用通信图、序列图和协作图等来描述系统中的交互和通信。这些图表用于表示对象之间的消息传递和协作。
7. 精化和细化模型：逐步完善和细化模型，添加更多的细节和约束。确保模型准确、一致和可理解。
8. 验证和验证模型：通过与相关利益相关者进行交流和讨论，验证模型的准确性和有效性。根据反馈和需求变更进行适当的修改和调整。

通过以上步骤，应用 UML 对系统进行建模可以帮助分析、设计和沟通软件系统的不同方面，促进团队的理解和合作，提高软件开发过程的可靠性和效率。

2.
浏览“LOGIC IN COMPUTER SCIENCE--Modelling and Reasoning about Systems”，了解常用逻辑及其在计算机学科中的应用

"LOGIC IN COMPUTER SCIENCE--Modelling and Reasoning about Systems" 是一本关于计算机科学中逻辑建模和推理的书籍。它介绍了常用的逻辑和它们在计算机学科中的应用。以下是一些常用的逻辑以及它们的应用领域：

1. 命题逻辑（Propositional Logic）：命题逻辑是一种最基本的逻辑形式，用于描述命题之间的真假关系。在计算机科学中，命题逻辑被广泛应用于逻辑推理、布尔代数、自动推理和形式化验证等领域。
2. 一阶逻辑（First-Order Logic）：一阶逻辑引入了谓词和量词的概念，可以描述对象和它们之间的关系。一阶逻辑在计算机科学中的应用包括形式化验证、程序验证、人工智能和数据库查询等。
3. 时序逻辑（Temporal Logic）：时序逻辑用于描述系统的时间和顺序性质。它可以表达关于系统行为的性质，如安全性、活性、并发性等。时序逻辑在硬件设计、并发算法、模型检测和实时系统验证等领域中发挥重要作用。
4. 模态逻辑（Modal Logic）：模态逻辑扩展了命题逻辑，引入了模态词（如"必须"、"可能"、"知道"）来描述命题的特定性质。模态逻辑在计算机科学中的应用包括形式化规范和验证、多代理系统、知识表示和推理等。
5. 高阶逻辑（Higher-Order Logic）：高阶逻辑允许对函数和谓词进行抽象和推理，可以描述更复杂的系统和属性。它在形式化验证、证明助理、编程语言语义

和类型理论等领域中得到应用。

阅读这本书可以深入了解逻辑在计算机科学中的应用，以及如何使用逻辑进行系统建模和推理。

3.
On-the-Criteria-To-Be-Used-in-Decomposing-Systems-into-Modules.pdf
分工协作，学习、检索研究经典软件体系结构案例。
On-the-Criteria-To-Be-Used-in-Decomposing-Systems-into-Modules.pdf

这篇论文是由 David L. Parnas 于 1972 年发表的经典论文，对软件系统的模块化和模块划分提出了一些重要的准则和原则。

在这篇论文中，Parnas 讨论了软件系统的模块划分问题，并提出了一些指导原则和标准，以帮助软件工程师进行合理的模块化设计。论文强调了模块划分的重要性，并提出了以下几个关键准则：
1. Information Hiding（信息隐藏）：通过隐藏模块的内部实现细节，使模块之间的耦合最小化。每个模块应该定义一个接口，只暴露必要的信息给其他模块，并将其内部实现细节封装起来。

2. Modular Decomposability（模块可分解性）：系统的模块划分应该是可分解的，即可以将系统分解为相互独立的、具有清晰职责的模块。这样可以简化系统的设计、测试和维护。
3. Modular Composition（模块组合）：模块之间的组合应该是简单、灵活和可靠的。模块之间的接口应该定义清晰，允许模块的替换、组合和重用。
4. Development Tolerance（开发容忍度）：模块应该独立于开发团队的特定技术和工具，以便在需要时能够进行更改、扩展和重构。

这些准则提供了一些原则和指导方针，帮助软件工程师进行模块化设计，提高软件系统的可维护性、可测试性和可扩展性。论文对软件工程领域的模块化设计产生了深远的影响，并被广泛引用和研究。

实验 10

1.
关于 MVC 视图和 Kruchten 4+1 视图
对比书上各种软件体系结构风格和视图特点，思考自己项目属于哪种设计风格？网上搜索最新的软件体系结构资料，如 MVC、Kruchten 4+1 视图等。

MVC（Model-View-Controller）是一种软件设计模式，用于将应用程序的逻辑、数据和用户界面分离。它包含以下三个核心组件：

1. Model（模型）：代表应用程序的数据和业务逻辑。它负责管理数据的状态、验证和更新，并为 View 和 Controller 提供接口。
2. View（视图）：负责展示数据和呈现用户界面。它从 Model 中获取数据并将其以用户可以理解的方式进行显示。View 可以是图形界面、命令行界面或其他形式。
3. Controller（控制器）：接收来自用户的输入并将其转发给 Model 或 View。它处理用户操作，更新 Model 的状态，并将更新的数据发送给 View 进行显示。

MVC 模式的优势包括提高代码的可重用性、可维护性和可测试性，以及促进不同组件的解耦和独立开发。

Kruchten 4+1 视图是一种软件体系结构视图模型，由 Philippe Kruchten 提出。它提供了多个视图来描述软件系统的不同方面，以便满足不同利益相关者的需求。这些视图包括：
1. 逻辑视图（Logical View）：描述系统的静态结构，包括组件、模块、类和它们之间的关系。它关注系统的功能、子系统和模块之间的依赖关系。
2. 开发视图（Development View）：关注软件系统的组织结构和开发过程，包括代码库、构建过程、模块化和团队组织等方面。
3. 进程视图（Process View）：描述软件系统的并发性、通信和执行流程。它关注系统的运行时行为、任务分配和并发处理等方面。
4. 物理视图（Physical View）：描述软件系统的部署和物理环境，包括服务器、网络拓扑、硬件设备等。
5. 场景视图（Scenarios View）：描述系统在特定场景下的使用和行为，例如用例、用户故事或场景描述。

Kruchten 4+1 视图通过提供不同的视角和关注点，帮助开发团队和利益相关者更好地理解和沟通软件系统的不同方面，促进系统的设计和开发。

我们小组的项目属于 MVC 视图。

2.
关于经典软件体系结构案例 KWIC

参阅课本和网上资料，研究经典软件体系结构案例 KWIC。

KWIC（Key Word In Context）是一个经典的软件体系结构案例，它用于对文本中的关键词进行索引和查找。以下是 KWIC 系统的基本架构描述：

1. 数据：KWIC 系统的输入是一组文本行，每行由一系列单词组成。这些文本行可以是文章、新闻标题、书籍章节等。
2. 索引生成：KWIC 系统的主要功能是生成关键词索引，使用户能够根据关键词快速查找相关文本行。索引生成包括以下步骤：
   - 输入处理：将输入的文本行进行预处理，如去除停用词（如“a”、“an”、“the”等）和标点符号。

- 循环移位：对于每个文本行，生成所有可能的循环移位，即将每个单词移到行的开头。
- 排序：将生成的循环移位按关键词进行排序，以构建关键词索引。
- 输出：将排序后的索引结果呈现给用户，供其进行关键词搜索。

3. 用户界面：KWIC 系统还包括一个用户界面，用于接收用户输入的关键词，并显示与关键词相关的文本行。

KWIC 系统的体系结构特点：
- 数据流体系结构：KWIC 系统采用数据流体系结构，其中文本行作为输入通过不同的处理阶段，并最终生成关键词索引作为输出。
- 分层架构：KWIC 系统可以采用分层架构，将输入处理、循环移位、排序和用户界面等功能划分为不同的层次。
- 模块化和可重用性：KWIC 系统的不同功能模块可以独立开发和测试，提高代码的可重用性和维护性。

实验 12

论述利斯科夫替换原则（里氏代换原则）、单一职责原则、开闭原则、德（迪）米特法则、依赖倒转原则、合成复用原则，结合自己的实践项目举例说明如何应用

1.利斯科夫替换原则（Liskov Substitution Principle，LSP）：
LSP 是指任何基类可以出现的地方，子类一定可以出现。即，在不改变原程序语义和功能的前提下，子类可以扩展父类的功能。这个原则的核心思想是：使用派生类型的客户端代码应该能够在无需修改的情况下使用基类对象。
在人事管理系统中，我们通过使用接口或抽象类定义通用的员工信息、薪水等属性和方法，以便将来扩展和修改。

2.单一职责原则（Single Responsibility Principle，SRP）：
SRP 是指一个类只负责完成一个职责或者功能。这个原则的优点是可以提高系统的可维护性、扩展性以及可测试性，降低了重构的成本。
在人事管理系统中，我们为每个类指定尽可能明确的职责，并避免将过多的职责耦合在同一个类中。

3.开闭原则（Open-Closed Principle，OCP）：
OCP 是指一个软件实体(类、模块、函数等)应对扩展开放，对修改关闭。这个原则的目的是通过添加新的功能来改进设计，而不是通过修改已经存在的代码来实现这些变化。
在人事管理系统中，我们为数据访问层、业务逻辑层和 UI 控制层之间添加抽象层，以便在未来实现新的功能或修改现有功能时，不会对整体架构造成影响。

4.德米特法则（最少知识原则）(Law of Demeter，LoD)：
LoD 也称为迪米特法则，是指一个对象应当对其他对象有尽可能少的了解，不直接与其他对象进行相互作用。这个原则试图减少四乱问题(依赖、耦合、全局数据引用和顺序敏感)。它是面向对象设计的基本原则之一，具有较强的可维护性和可扩展性。
在人事管理系统中，我们将职责划分明确，将每个类之间的依赖关系降到最低，以便让其更加解耦合。

5.依赖倒转原则（Dependency Inversion Principle，DIP)：
DIP 是指高层模块不应该依赖于底层模块，而是应该二者都依赖于抽象接口。这个原则目的是降低高层模块与低层模块之间的耦合，提高代码的灵活性、可扩展性和可维护性。
在人事管理系统中，我们使用 IoC 容器如 Spring 来控制对象的创建和管理，并使用依赖注入（DI）的方式来遵循 DIP 原则。

6.合成复用原则（Composite Reuse Principle，CRP)：
CRP 规定尽量使用对象组合，而不是继承来达到复用的目的。这个原则的核心思想是,将多种已有的对象组合在一个类中,作为这个类的对象,来实现新的功能。它提供了更灵活、更可扩展、更易于维护和测试的方式来创建软件系统。
在人事管理系统中，我们实现了多种员工属性的复用，比方说保存与调整员工薪资时利用通用性高的方法，而不是重复编写相同或相似的代码。

实验 13

给出 4 种设计模式的例子（语言不限，以组为单位），并总结其特点

1.工厂模式 Factory Pattern:属于创建型设计模式。一个对象(通常称为工厂)负责创建其他对象。它通过使用工厂方法,抽象出实际工厂类繁琐的精细化操作，对客户端提供接口，使得客户端可以从这个通用接口中创建所需产品。工厂模式不仅提高了系统的灵活性和可维护性，还能降低了代码耦合度。

2.单例模式 Singleton Pattern：属于创建型设计模式。确保类只有一个实例存在，并提供全局访问点以便于访问该实例。典型的应用场景是日志记录、缓存或数据库连接对象这种资源密集型对象。但它也常常被滥用和过度使用，容易造成思想定式和破坏代码结构。

3.观察者模式 Observer Pattern：属于行为型设计模式。在对象间建立一对多的依赖关系，当一个对象的状态发生改变时，其所有依赖对象都会收到通知并自动更新。常用于事件处理、GUI 系统或机器人。

4.装饰器模式 Decorator Pattern：属于结构型设计模式。允许将新功能添加到现有对象上，而不影响其他对象。常用于图形用户界面框架中的高级主题支持，或是避免覆盖写一个大类的最多子类。

5.适配器模式 Adapter Pattern：属于结构型设计模式。将两个不兼容的接口转换成兼容的接口。能够让不兼容的代码协同工作。

6.策略模式 Strategy Pattern：属于行为型设计模式。定义一组算法，把它们封装起来，并使它们互相替换。常用于游戏中的 AI 策略、商业智能等复杂系统中。

人事管理系统用到了以下设计模式：

1.MVC 模式：MVC 是 Model-View-Controller 的缩写，是一种常见的 Web 开发模式。在人事管理系统中，Model 负责业务逻辑和数据存储，View 负责呈现页面并与用户交互，而 Controller 则负责处理用户请求和调度 Model 和 View，从而实现业务流程的控制。

2.工厂模式：在人事管理系统中，利用工厂模式来创建用户、部门、职位等对象，从而使代码更加松耦合。

3.单例模式：人事管理系统中存在一些资源密集型对象，如数据库连接池、缓存管理器等，这些对象只需要创建一次就可以充分利用系统资源。因此，可以使用单例模式确保这些对象只有一个实例存在，并提供全局访问点以便于访问该实例。

4.迭代器模式：在人事管理系统中，需要对各种集合进行操作，例如，遍历员工列表、查询部门信息等。迭代器模式可以提供一种简单的方式来访问、遍历和操作集合，同时使代码更加清晰易懂。

5.状态模式：在人事管理系统中，员工的状态可能会发生变化，例如，新员工、试用期员工、正式员工、离职员工等。状态模式可以提供一种简单的方式来管理不同状态下的员工信息，并根据状态的变化自动更新相应的业务逻辑。

6.观察者模式：人事管理系统中存在许多事件需要通知其他对象进行处理，例如，新员工入职、员工离职、部门调整等。观察者模式可以实现对象间的松耦合，当一个事件发生时，它可以将通知传递给所有已注册的观察者，由观察者负责进行相应的处理。

实验 14

1.

软件测试的基本概念、主要技术和重要挑战：

基本概念：

1. 测试目标：确定测试的目的和期望的结果。
2. 测试计划：制定测试策略、资源分配和时间表。

3. 测试用例：编写测试用例，描述预期输入、操作和预期输出。
4. 缺陷管理：记录和跟踪发现的缺陷，追踪其解决过程。
5. 回归测试：在进行更改或修复后，重新运行先前通过的测试用例，以确保新更改不会导致其他问题。

主要技术：
1. 黑盒测试：基于功能需求和规格说明，独立于内部实现的测试方法。
2. 白盒测试：检查软件内部结构、代码和逻辑的测试方法。
3. 自动化测试:使用自动化工具和脚本执行测试用例,提高测试效率和准确性。
4. 性能测试：评估软件在不同负载和压力条件下的性能和响应能力。
5. 安全测试：评估软件系统的安全性和防御能力，以保护数据和用户隐私。

重要挑战：
1. 自动化测试：随着软件系统的复杂性和规模的增加，手动测试变得越来越困难和耗时。因此，自动化测试已成为一种必要的选择。然而，自动化测试也面临着许多挑战，例如如何选择适当的工具、如何编写有效的测试脚本等。

2. 大规模软件系统的测试：现代软件系统通常非常大且复杂，由许多不同的组件和模块组成。因此,在这些系统中进行全面和有效的测试是非常具有挑战性的。此外，这些系统还可能涉及分布式计算、云计算等技术，这使得测试更加困难。

3. 安全性和隐私保护：现代软件系统需要保护用户数据和敏感信息免受黑客攻击和数据泄露等威胁。因此，在软件开发过程中需要进行安全性和隐私保护方面的测试。然而，这种类型的测试也很具有挑战性，因为攻击者可以使用各种不同的技术来绕过安全措施。

总之，软件测试是一个复杂而重要的领域，需要不断地面对各种挑战和问题。

2.
白盒测试，黑盒测试
阅读下面白盒测试和黑盒测试相关资料（或查阅其它相关资料），深入理解白盒测试和黑盒测试，总结其特点
WhiteBox.pdf
BlackBox.pdf

白盒测试是一种软件工程师可以使用的验证技术,用于检查他们的代码是否按预期工作。它考虑了系统或组件的内部机制，也称为结构测试、透明盒测试和玻璃盒测试。

白盒测试的特点包括：
1.可以检查代码中的路径是否按预期执行。
2.可以使用等价类划分和边界值分析来管理需要编写的测试用例数量,并检查容易出错或极端“角落”测试用例。

3.可以测量测试用例对代码的覆盖程度。
4.是一种受控制的验证和验证技术，可以通过运行测试用例来发现问题(故障)，并且可以根据失败的测试用例确定需要修复问题的代码行。
5.与黑盒测试相比，白盒测试更加经济高效，因为它可以在开发周期早期就发现缺陷。

要编写有效的白盒测试用例，需要考虑以下几个方面：
1.确定要覆盖哪些路径和条件。
2.使用等价类划分和边界值分析来确定输入值范围。
3.编写能够覆盖所有路径和条件的最小集合的测试用例。
4.测试每个条件是否满足，并检查输出是否正确。

黑盒测试同样也是一种软件测试技术，它的特点是在不考虑程序内部结构和实现的情况下，仅通过输入和输出来评估软件系统的正确性和质量。以下是黑盒测试的一些常见技术和特点：
1. 等价类划分：将输入数据分为不同的等价类，每个等价类都有相同的功能和行为。这样可以减少测试用例数量并提高测试效率。
2. 边界值分析：测试边界值情况，例如最大值、最小值、临界值等。这些边界情况通常容易出现错误。
3. 决策表：使用决策表来确定输入条件和输出结果之间的关系。这可以帮助识别可能存在的缺陷。
4. 因果图：因果图是一种图形化工具，用于描述系统中各个组成部分之间的关系。它可以帮助识别潜在问题，并确定哪些部分需要进行更详细的测试。
5. 随机测试：随机选择输入数据进行测试，以发现未知错误或异常情况。

3.
符号测试的基本概念、主要技术和重要挑战
阅读下面符号测试 （Symbolic Testing）相关资料（或查阅其它相关资料），了解符号测试的基本概念、主要技术、重要挑战等

A Survey of Symbolic Execution Techniques.pdf
Symbolic Execution and Program Testing.pdf
Symbolic Execution for Software Testing-Three Decades Later.pdf

符号测试是软件测试中的一种技术方法，它基于对软件系统的规格说明和源代码进行分析，利用符号执行和符号约束求解技术来生成和执行测试用例。下面是符号测试的基本概念、主要技术和重要挑战：

基本概念：
1. 符号执行：使用符号值代替具体输入值来执行程序，从而生成涵盖不同输入路径和条件的符号执行路径。
2. 符号约束：通过收集符号执行路径上的约束条件，将其转化为逻辑表达式，

形成符号约束。
3. 符号执行路径：由符号执行生成的测试路径，其中包含符号值的符号变量。
4. 符号化输入：将具体输入值替换为符号值，以生成不同的输入路径。

主要技术：
1. 符号生成：使用符号执行技术生成具有不同路径的测试用例，以实现高覆盖率和更全面的测试。
2. 符号约束求解：通过符号约束求解器解决符号约束，找到满足约束条件的具体输入值，从而生成具体的测试用例。
3. 符号化执行环境：利用符号执行技术，对程序的执行环境进行符号化，包括文件系统、网络、内存等。

重要挑战：
1. 符号执行路径爆炸：由于程序的复杂性，符号执行可能会生成大量的路径，导致路径爆炸问题，影响测试效率。
2. 符号约束求解效率：符号约束求解过程可能需要消耗大量时间和计算资源，特别是对于复杂的约束条件。
3. 动态内存管理：符号执行对动态内存分配和释放的处理比较困难，需要考虑内存溢出和内存泄漏等问题。
4. 复杂的程序语言特性：某些复杂的程序语言特性，如指针操作、多线程等，可能增加符号执行的复杂性和困难度。
5. 不完备的规格说明：如果规格说明不完整或模糊，符号测试的有效性和准确性可能受到影响。

4.
差分测试的基本原理与主要应用
阅读下面差分测试（Differential Testing)相关资料(或查阅其它相关资料)，了解差分测试的基本原理、主要应用等
Differential Testing for Software.pdf
Feedback-Directed Differential Testing of Interactive Debuggers.pdf

差分测试（Differential Testing）是一种基于比较软件系统不同实现或版本之间的差异来发现软件缺陷的测试方法。它的基本原理是将多个实现或版本的软件系统同时执行相同的测试集，并比较它们的输出结果。差异的输出被认为是潜在的缺陷或错误。

差分测试的基本原理如下：
1. 选择测试集：选择一组具有广泛覆盖性和代表性的测试用例，这些测试用例能够涵盖软件系统的不同功能和边界情况。
2. 执行多个实现或版本：将所选的测试用例同时输入到不同的实现或版本的软件系统中进行执行。
3. 比较输出结果：比较不同实现或版本之间的输出结果，检查是否存在差异。
4. 标识差异：差异的输出结果被标识为潜在的缺陷或错误，需要进一步调查和

修复。

差分测试的主要应用包括：
1. 编译器测试：在编译器的不同版本之间执行相同的源代码，并比较生成的目标代码的差异，以发现编译器的错误。
2. 操作系统测试：比较不同操作系统版本的行为和功能差异，检测和纠正可能存在的错误和不一致性。
3. 浏览器和 Web 应用测试：比较不同浏览器或 Web 应用的渲染和交互行为，识别可能的错误和兼容性问题。
4. 数据库测试：对不同数据库管理系统的查询和操作执行相同的 SQL 语句，比较结果以发现差异和问题。
5. 编码器和解码器测试：对不同的音频或视频编码器执行相同的输入进行编码，并比较输出以检测编解码器的问题。