实验九 UML，逻辑， 软件体系结构设计（一）

实验目的：

1. 深入理解UML

2. 了解计算机学科中的逻辑

3. 学习对比软件体系结构设计GB和IEEE最新SAD (Software Architecture Document)的标准

4. 研究经典软件体系结构案例

5. 完成自己项目的SRS

实验内容：

1. 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

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

3. 分工协作，参考国标“13 - 软件(结构)设计说明(SDD)”等资料，对比参考SAD最新标准IEEE-42010.pdf，针对自己的项目设计SAD初稿。

4. 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

http://www.cs.cmu.edu/~ModProb/index.html

5. 完成软件需求规格说明SRS

**下周五（含）前将软件需求规格说明提交给相应的助教**

项目跟踪，建立能反映项目及小组每个人工作的进度、里程碑、工作量的跟踪图或表，将其保存到每个小组选定的协作开发平台上，每周更新。

**实验内容：**

**1.深入学习UML（Unified Modeling Language，统一建模语言）的基本知识，不仅可以帮助我们理解软件系统的结构和行为，还能提高我们在软件开发过程中的效率和准确性。以下是关于UML深入学习的关键要点：**

### UML的核心概念

* **面向对象**：UML是面向对象的分析与设计的一种标准表示，它采用类、对象、抽象、封装、继承、多态等面向对象的概念和原则来构造软件系统。
* **图形化建模**：UML通过图形化的方式表示系统的结构、行为和交互，使得系统模型更易于理解和沟通。

### UML的主要组成

* **基本构造块**：包括建模元素（如类、对象、接口等）、关系（如依赖、关联、泛化、实现等）以及图（如用例图、类图、状态图等）。
* **UML规则**：定义了建模过程中的命名、范围、可见性、完整性和执行等规范。
* **公共机制**：包括规格说明、修饰、通用划分以及扩展机制（如构造型、标记值、约束等）。

### UML的视图

UML定义了五种主要的视图，用于从不同角度描述系统：

* **用例视图**：关注外部参与者看到的或需要的系统功能。
* **逻辑视图**：展示系统的静态结构和动态行为。
* **组件视图**：显示代码组件的组织结构。
* **并发视图**：处理并发系统中的通信和同步问题。
* **配置视图**：描述系统的具体部署情况。

### UML的建模方法

UML提供了多种建模方法，用于构建和分析系统模型：

* **类图**：用于描述对象、类、接口以及它们之间的关系，是面向对象分析的核心。
* **状态图**：展示系统行为与状态的变化。
* **活动图**：描述从一个状态到另一个状态的行为过程。
* **部署图**：描述系统的物理部署情况。
* **交互图**：用于描述不同构件间及操作的同步过程。

### UML的应用领域

UML不仅适用于软件系统的设计和开发，还广泛应用于系统分析、数据库设计、代码生成以及软件测试等领域。

### 学习UML的建议

* **理论与实践相结合**：通过实际案例和项目来应用UML知识，加深理解。
* **参考专业书籍和教程**：系统学习UML的理论知识和建模技巧。
* **参与社区讨论**：与其他UML学习者交流经验，共同解决问题。

总之，深入学习UML的基本知识需要系统地掌握其核心概念、组成、视图、建模方法以及应用领域。通过理论与实践的结合，我们可以更好地应用UML进行软件系统的建模和分析，提高软件开发的效率和质量。

应用UML对系统进行建模是一个系统性的过程，它可以帮助我们更好地理解、设计和分析软件系统。以下是关于如何应用UML进行系统建模的一些基本步骤：

1. **确定系统需求**：
   1. 在开始建模之前，首先要明确系统的需求，包括功能需求、性能需求、安全需求等。
   2. 通过与项目团队、业务用户和其他利益相关者进行沟通，收集并整理这些需求。
2. **识别系统实体**：
   1. 分析系统中的主要对象、类、接口以及它们之间的关系。
   2. 把握实体间的交互方式，理解它们之间的关系及联系。
3. **选择适当的UML图**：
   1. UML提供了多种类型的图，如用例图、类图、状态图、活动图、部署图、交互图等。
   2. 根据系统的需求和建模目的，选择适合的UML图进行建模。例如，用例图用于描述系统的功能需求，类图用于描述系统的数据结构和对象之间的关系。
4. **构建UML图**：
   1. 使用所选的UML图来构建系统的模型。这通常包括定义类、对象、关系、状态、活动等。
   2. 对于复杂的系统，可能需要使用多个UML图从不同的角度来描述系统。
5. **分析模型**：
   1. 对构建的UML模型进行分析，检查其是否符合系统的需求。
   2. 通过分析模型，可以发现潜在的问题和缺陷，进而进行优化和调整。
6. **编写文档**：
   1. 使用UML的各种图形符号和规范，编写系统的文档。
   2. 文档应详细记录系统的设计、实现细节以及与其他部分的交互方式。
7. **迭代与改进**：
   1. 在系统开发过程中，随着需求的变化和技术的更新，可能需要不断地对UML模型进行迭代和改进。
   2. 通过持续的反馈和调整，确保UML模型始终与实际系统保持一致。

通过以上步骤，我们可以应用UML对系统进行建模，从而更好地理解系统的结构和行为，为系统的设计和开发提供有力的支持。同时，UML的图形化表示方式也使得团队成员之间的沟通更加高效和准确。

**2.“LOGIC IN COMPUTER SCIENCE--Modelling and Reasoning about Systems”是一本探讨逻辑在计算机科学中建模和推理的书籍。它详细阐述了逻辑在计算机系统设计和分析中的重要作用，以及如何通过逻辑来进行系统建模和推理。**

在计算机科学中，逻辑被广泛应用于多个领域，它是计算机程序设计和算法设计的基础。常用的逻辑包括命题逻辑、谓词逻辑、模态逻辑等。这些逻辑为计算机科学提供了严谨的数学基础和推理工具，使得计算机系统能够准确地处理信息、做出决策和进行推理。

命题逻辑是最基础的逻辑，它研究的是由命题构成的复合命题的真假关系。在计算机科学中，命题逻辑常用于描述程序的状态和条件，以及进行逻辑运算和判断。

谓词逻辑则进一步扩展了命题逻辑，引入了变量和量词，能够描述更为复杂的对象和关系。在计算机科学中，谓词逻辑常用于描述数据结构、算法和程序的行为，以及进行形式化验证和推理。

模态逻辑则考虑了时间和可能性等因素，进一步丰富了逻辑的表达能力。在计算机科学中，模态逻辑可以用于描述系统的动态行为和并发性，以及进行时态推理和概率推理。

除了以上几种常用的逻辑外，计算机科学还涉及许多其他类型的逻辑，如模糊逻辑、多值逻辑等。这些逻辑为计算机科学提供了更为丰富的表达手段和处理方式，使得计算机系统能够更好地适应复杂多变的环境和需求。

在计算机科学中，逻辑的应用非常广泛。例如，在人工智能领域，逻辑被用于知识表示、推理和决策等方面；在数据库系统中，逻辑用于查询优化和完整性检查；在形式化方法中，逻辑用于描述和验证系统的性质和行为；在编程语言和编译器设计中，逻辑用于类型检查和语义分析等方面。

总之，“LOGIC IN COMPUTER SCIENCE--Modelling and Reasoning about Systems”这本书深入探讨了逻辑在计算机科学中的应用，展示了逻辑在计算机系统建模和推理中的重要作用。通过学习这本书，读者可以更好地理解逻辑的原理和方法，并将其应用于实际的计算机系统设计和分析中。

**3.学习对比软件体系结构设计GB（国家标准）和IEEE最新SAD（Software Architecture Document，软件架构文档）的标准，可以帮助我们深入理解两者的异同，从而更好地应用于实际的软件设计和开发过程中。以下是对这两个标准的对比分析：**

一、GB（国家标准）

GB是中国的国家标准，针对软件体系结构设计提出了详细的规范和要求。这些标准通常涵盖了软件项目管理的各个方面，如项目立项与组织、项目计划与控制、质量管理、配置管理以及缺陷管理等。在软件需求工程方面，GB也强调了需求获取与分析、需求规格说明以及需求确认和变更管理的重要性。

在软件体系结构设计方面，GB可能强调系统的层次结构、模块和组件的划分，以及数据流和信息流的设计。此外，GB还可能关注系统的可扩展性、可维护性、性能和安全性等方面的设计原则。

二、IEEE最新SAD（Software Architecture Document）

IEEE（电气和电子工程师协会）是全球领先的电气和电子工程领域的标准制定机构。其最新的SAD标准旨在为软件架构文档提供统一的格式和规范，以确保软件架构的清晰、一致和可维护。

IEEE SAD标准通常包括软件架构的描述、设计决策的理由、架构与其他视图的关系以及架构的验证和确认等内容。它强调了对软件架构的全面描述，以便项目团队和其他利益相关者能够清晰地理解系统的整体结构和行为。

此外，IEEE SAD标准还可能关注软件架构的非功能性需求，如性能、安全性、可用性等，并提供相应的设计和实现指导。

三、对比分析

1. 共同点：GB和IEEE SAD标准都强调软件体系结构设计的重要性，并提供了相应的规范和指导。它们都关注系统的结构、行为、性能和安全性等方面，以确保软件的质量和可维护性。
2. 不同点：GB可能更侧重于国内的实际情况和需求，提供适合国内软件开发环境的规范和要求。而IEEE SAD标准则更偏向于国际化和通用性，为全球范围内的软件开发者提供统一的软件架构文档标准。此外，两者在细节上可能也存在差异，如术语定义、设计原则、验证方法等方面。

四、总结

GB和IEEE最新SAD标准在软件体系结构设计方面都有其独特的优势和价值。学习和掌握这两个标准，可以帮助我们更好地理解和应用软件体系结构设计的方法和原则，提高软件开发的效率和质量。在实际应用中，我们可以根据项目需求和团队能力选择合适的标准，并结合实际情况进行灵活应用。

**4. 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**

**http://www.cs.cmu.edu/~ModProb/index.html**

《On the Criteria To Be Used in Decomposing Systems into Modules》是一篇经典的计算机科学论文，由David L. Parnas于1972年发表。该论文首次提出了信息隐藏的概念，并讨论了如何将复杂的系统分解为模块的技术。以下是关于这篇论文的学习和研究的初步理解：

**论文背景与目的：**

了解论文发表的背景，包括当时软件工程的挑战和局限性。

明确论文的主要目的，即探讨如何有效地将系统分解为模块，以提高软件的可维护性、可扩展性和可重用性。

**信息隐藏概念：**

深入理解信息隐藏的概念，它是这篇论文的核心思想。信息隐藏是指将计算机程序中最有可能改变的设计决策进行隔离，从而当设计决策发生变化时，程序的其他部分不需要被大量修改。

探讨信息隐藏如何帮助实现更好的软件设计，包括降低模块之间的耦合度、提高内聚性等。

**系统分解与模块设计：**

研究论文中提出的系统分解策略，包括如何从一系列困难的设计决策或可能改变的设计决策开始，然后设计每个模块以隐藏这些决策。

分析论文中给出的示例和案例，了解如何在实际应用中应用这些策略。

**模块化技术的优势与局限性：**

评估模块化技术在提高软件质量、降低开发成本等方面的优势。

探讨模块化技术的局限性，如可能增加系统的复杂性、影响性能等。

对《On the Criteria To Be Used in Decomposing Systems into Modules》的再次理解

《On the Criteria To Be Used in Decomposing Systems into Modules》这篇论文主要探讨了如何将复杂的软件系统分解为独立的模块，以提高软件的可维护性、可扩展性和可重用性。Parnas在论文中提出了信息隐藏的概念，并强调了其在模块化设计中的重要性。他认为，一个良好的软件系统应该能够隐藏其内部实现细节，只通过明确的接口与外部交互。这种信息隐藏有助于减少模块之间的耦合度，提高模块的内聚性，从而使得系统更加易于理解和维护。

一、信息隐藏的重要性

信息隐藏是Parnas在论文中提出的核心概念之一。它强调了在模块化设计中，每个模块都应该隐藏其内部实现细节，只通过明确的接口与外部交互。这种设计原则有助于减少模块之间的耦合度，提高模块的内聚性，从而使得系统更加稳定、可维护和可扩展。信息隐藏的重要性主要体现在以下几个方面：

1. 降低复杂性：通过将系统的复杂性限制在特定的模块内部，信息隐藏有助于降低整个系统的复杂性。这使得开发人员能够更加专注于每个模块的具体实现，而无需过多关注其他模块的内部细节。
2. 提高可维护性：由于每个模块都隐藏了其内部实现细节，因此当某个模块出现问题时，其他模块不会受到影响。这使得开发人员能够更加容易地定位问题并进行修复，从而提高了系统的可维护性。
3. 促进重用和共享：通过隐藏内部实现细节，模块可以更加容易地被其他系统或项目重用和共享。这有助于减少开发成本和时间，并提高代码的质量和可靠性。

二、系统分解的标准

Parnas在论文中提出了系统分解的标准，这些标准为我们提供了如何有效地将系统分解为模块的指导。根据这些标准，我们可以将系统分解为一系列独立的模块，每个模块都负责实现一组特定的功能，并隐藏其内部实现细节。这些模块之间通过明确的接口进行交互，以确保系统的稳定性和可维护性。系统分解的标准主要包括以下几个方面：

1. 功能需求：根据系统的功能需求，我们可以将系统分解为多个模块，每个模块都负责实现一组特定的功能。这有助于将系统的复杂性分散到各个模块中，使得每个模块都更加易于理解和维护。
2. 设计决策：在系统设计过程中，我们会做出一系列的设计决策。这些设计决策可能涉及到数据结构、算法、接口等方面。根据这些设计决策，我们可以将系统分解为多个模块，每个模块都负责实现一组相关的设计决策。这有助于将系统的复杂性限制在特定的模块内部，减少模块之间的耦合度。
3. 耦合度和内聚性：耦合度和内聚性是衡量模块质量的重要指标。在分解系统时，我们应该尽量降低模块之间的耦合度，提高模块的内聚性。这有助于使得系统更加稳定、可维护和可扩展。

三、模块化设计的实践应用

模块化设计在软件工程领域具有广泛的应用。通过将系统分解为独立的模块，我们可以更好地管理系统的复杂性，提高开发效率和质量。以下是一些模块化设计的实践应用：

1. 面向对象编程：面向对象编程是一种支持模块化设计的编程范式。在面向对象编程中，我们将系统分解为多个对象（即模块），每个对象都负责实现一组特定的功能，并隐藏其内部实现细节。对象之间通过消息传递进行交互，从而实现系统的整体功能。
2. 微服务架构：微服务架构是一种将系统分解为多个独立服务的架构模式。每个服务都是一个独立的模块，负责实现一组特定的功能，并隐藏其内部实现细节。服务之间通过轻量级的通信协议（如RESTful API）进行交互，从而实现系统的整体功能。微服务架构具有高度的可扩展性、可维护性和可重用性，适用于构建大型分布式系统。
3. 插件化设计：插件化设计是一种将系统分解为多个可插拔模块的设计方法。每个插件都是一个独立的模块，负责实现一组特定的功能，并隐藏其内部实现细节。插件之间通过统一的接口进行交互，从而实现系统的整体功能。插件化设计使得系统具有高度的灵活性和可扩展性，可以根据需要添加或删除插件来扩展系统的功能。