**西北工业大学实验报告**

课程名称: 高级语言程序设计实验

实验课次周次: 第四次（第十三周）

实验地点: 实验大楼B309-1

学生姓名： 刘力豪

学 号: 2022303210

实验时间: 2024-05-25

提交时间: 2024-05-29

|  |
| --- |
| **成绩评定：（满分为100分）**  **指导教师评语：**  **指导教师签名：** |

目 录

[1 UML实验七 包图 2](#_Toc4600)

[1.1 实验内容 2](#_Toc5451)

[2 UML实验八 活动图 5](#_Toc31103)

[2.1 实验内容 5](#_Toc15566)

[2.2 实验内容： 9](#_Toc13259)

[3 UML实验九 组件图 13](#_Toc6661)

[3.1 实验内容 13](#_Toc7159)

[3.2 UML实验结果 15](#_Toc30126)

[4 UML实验十 UML综合实验 16](#_Toc4494)

[4.1 实验内容 16](#_Toc11214)

[4.2 UML结果文件说明 17](#_Toc6305)

[4.3 UML实验结果 26](#_Toc20453)

[5 实验总结 29](#_Toc24468)

[5.1 实验中遇到的问题以及解决方法 29](#_Toc17410)

[5.2 实验体会 30](#_Toc119)

# UML实验七 包图

## 实验内容

### 包属于UML中的何种事物？有何作用？

解：

UML中，包是一种组织和分组机制，用于将模型中的元素分组在一起。包可以包含类、接口、组件、节点和其他包等模型元素。

作用：

<1>.逻辑分组：将相关的元素组织在一起，以便于理解和管理。

<2>.命名空间：提供命名空间，避免命名冲突。

<3>.层次结构：通过包的嵌套形成层次结构，清晰地展示系统的结构和模块化

### 2.UML中包的表示方法有哪几种？

解：

<1>.标准表示法：用一个带标签的矩形表示，标签即为包的名字。矩形的左上角有一个小矩形标签，表示这是一个包。

<2>.折叠表示法：用文件夹的图标表示，适用于在图中需要节省空间或简化显示时。

### 3.包与包之间存在哪几种关系？

解：

依赖关系：一个包中的元素依赖于另一个包中的元素。

包含关系：一个包包含另一个包。

导入关系：一个包导入另一个包中的元素。

### 4.包与包之间的依赖关系有何特点？

解：特点：

方向性：依赖关系是有方向的，从依赖的一方指向被依赖的一方。

传递性：依赖关系具有传递性，如果包A依赖包B，而包B依赖包C，那么包A间接依赖包C。

弱耦合：依赖关系相对比较松散，仅表示某种程度上的使用关系，而不涉及所有权或包含。

### 5.如何避免包与包之间出现循环依赖关系？各举一例。

解：避免包与包之间的循环依赖关系的方法包括：

<1>.重构代码：将共同依赖的部分抽取到一个新的包中。

<2>.使用接口：通过接口而不是具体实现进行依赖，从而减少耦合。

例：

重构代码：

原有结构：包A依赖包B，包B依赖包C，包C依赖包A。

重构后：将包A和包C的共同依赖部分抽取到新包D中，使包A、包B、包C都依赖于包D，从而消除循环依赖。

使用接口：

原有结构：包A直接依赖包B，包B直接依赖包A。

重构后：包A和包B各自定义接口I\_A和I\_B，并依赖对方的接口而不是具体实现。

### 6.建立包图应注意哪些问题？

解：

合理分组：确保包的分组逻辑合理，具有高内聚低耦合。

避免循环依赖：避免包之间形成循环依赖，保持依赖关系的清晰性。

命名空间管理：明确包的命名空间，避免命名冲突。

层次结构清晰：确保包的层次结构清晰，容易理解和维护。

合适的粒度：包的粒度不宜过细或过粗，应该适中，以便于管理和理解。

### 7.根据下图判断下面哪句话正确的说明了包之间的依赖关系。

解：由图片可知：Package1 依赖 Package2；Package2 依赖 Package3

因此B正确：

分析：

A) 描述的检查顺序错误。Package1 依赖 Package2，Package2 依赖 Package3，而不是 Package1 依赖 Package3。

C) 逻辑错误。修改 Package3 的元素后，不需要检查 Package1，因为依赖关系是单向的，从 Package1 到 Package2 再到 Package3。

D) 同样描述了错误的依赖关系检查顺序。

### 8.下面包图设计中是否存在问题？应如何改进。

解：在图A中存在循环依赖关系的问题。具体来说，包之间的依赖关系如下：

A -> B

B -> C

B -> E

E -> C

C -> A

这样的依赖关系形成了一个循环依赖链，即：A -> B -> C -> A。这会导致难以维护和扩展系统。

改进建议：

消除循环依赖：重构包的依赖关系，确保每个包只依赖于前一级的包，而不是相互依赖。

改进后的设计

图B

A 作为一个独立的包，不依赖其他包。

B 依赖 A。

C 依赖 B。

D 依赖 C。

图B的优点

无循环依赖：每个包的依赖关系是单向的，形成一个树状结构。

层次分明：包的层次结构清晰，每个包只依赖其上一级的包，方便管理和扩展。

总结

通过将图A中的循环依赖关系重新组织为图B中的层次结构，可以显著提高系统的可维护性和可扩展性。建议在实际设计中，尽量避免包之间的循环依赖，保持包的层次分明和依赖关系的单向性。

# UML实验八 活动图

## 实验内容

### 1.什么是活动？UML中如何表示活动？

解：

活动是指业务流程或操作中的一个步骤或任务，通常表示一系列动作的执行。活动可以是一个简单的任务，也可以是复杂的操作，包括多步骤的过程。在UML中，活动使用一个椭圆形或矩形来表示，并在其中标注活动的名称。

### 活动图中包括哪些元素？分别如何表示？

解：

活动图中的主要元素包括：

活动：表示为一个椭圆形或矩形，标注活动名称。

初始状态：表示流程的起点，用一个填充的黑色圆圈表示。

终止状态：表示流程的终点，用一个填充的黑色圆圈包围另一个未填充的圆圈表示。

动作：表示具体的操作，用椭圆形或矩形表示，类似于活动。

决策节点：表示流程中的决策点，用菱形表示，并标注不同条件的箭头。

合并节点：表示多个分支汇合处，用菱形表示，类似于决策节点，但没有条件。

分叉节点：表示流程的并行分支，用一个粗横线或竖线表示，箭头分叉。

汇合节点：表示并行分支的汇合，用一个粗横线或竖线表示，箭头汇合。

对象流：表示对象在活动之间的传递，用带箭头的实线表示。

控制流：表示活动之间的控制转移，用带箭头的实线表示。

### 分岔和分支有什么区别,分别适用于什么样的建模目标？

解：

分岔：

表示：使用粗横线或竖线。

作用：用于表示并行流程，即多个活动可以同时进行。

适用场景：当需要表示多个活动可以同时执行时使用，例如并行任务的建模。

分支：

表示：使用菱形。

作用：用于表示条件选择，即根据条件的不同执行不同的活动路径。

适用场景：当需要表示基于条件的流程选择时使用，例如判断和条件分流。

### 活动图中监护条件和改变时间分别有何作用？两者间的区别是什么？

解：

监护条件：

作用：用于控制流程的转移。表示一个控制流或对象流的执行条件。

表示：在控制流或对象流的箭头旁边，用方括号标注条件表达式。

使用场景：用于判断条件，决定流程走向。

改变时间：

作用：表示某个特定的事件发生时会触发的动作或活动。

表示：在箭头旁边，用 when 关键字标注时间表达式。

使用场景：用于表示特定时间点或事件触发的流程转移。

区别：

监护条件是基于状态或数据的条件判断，决定某个路径是否可以被执行。

改变时间则是基于时间或事件的触发，决定某个活动或动作何时执行。

### 说明下面活动图所表达的信息。

解：

起始点：表示流程的开始。

活动：用户下订单：表示用户在系统中下订单。

生成发货单：

用户下订单后，进入 生成发货单 活动。

用户选择支付方式：

生成发货单后，用户选择支付方式。

决策节点：根据用户选择的支付方式，流程会进行不同的路径：

用户通过货到付款：进入 收款 活动。

用户通过预付款：进入 用户通过预付款超过15分钟 的监护条件判断。

判断条件：用户通过预付款超过15分钟：

如果超过15分钟未付款（[No]），订单被取消，流程终止。

如果在15分钟内付款（[Yes]），进入 收款 活动。

收款：

收款完成后，流程进入并行操作。

并行操作：供应商配送货物 和 检查订单质量缺陷：

供应商配送货物：供应商将货物配送至用户。

检查订单质量缺陷：检查订单是否有质量问题。

这两个活动是并行的，表明可以同时进行。

决策节点：所有订单项是否都完成：

在两个并行活动都完成后，进入决策节点。

判断所有订单项是否都完成。

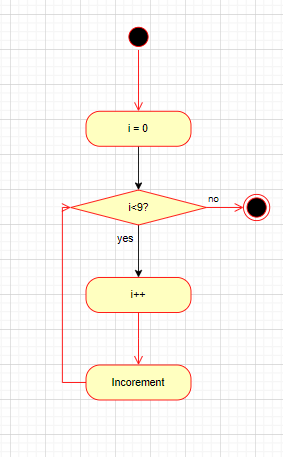
如果是（[Yes]），进入 订单完成 活动，表示整个订单流程完成。

如果存在未完成的订单项，流程未明确表示会如何处理，但一般会返回并重新进行相关检查或配送。

终止点：订单完成 活动后，流程结束，表示整个订单处理流程完成。

### 用活动图表示for（i=0；i<9;i++）循环。

解：



### 对于如下图所示的活动图，最大可能的并发线程数是几个？

解：

Activity1 分叉成 Activity2 和 Activity3，此时并发线程数为2。

Activity2 进一步分叉成 Activity4 和 Activity6，此时再增加1个并发线程。

因此，最大并发线程数 = 2（Activity2和Activity3） + 1（Activity4和Activity6） = 3

## 实验内容：

**[分析与讨论]**

### 1.总结UML在软件工程中的作用以及使用UML建模的必要性。

解：

UML在软件工程中的作用主要体现在以下几个方面：

（1）.可视化建模：

UML提供了一种标准化的图形表示法，使得复杂的系统设计可以通过图形化的方式展示。这样不仅有助于开发人员理解系统，还可以向非技术人员（如管理层、客户）清晰地展示系统的设计和功能。

（2）.规范化：

UML作为一种标准化语言，提供了统一的建模语法和语义，消除了不同开发团队之间的沟通障碍。无论开发者身处何地，只要遵循UML标准，就能确保对系统的理解一致。

（3）.文档生成：

UML模型可以自动生成系统的设计文档，这对于项目的长期维护、版本控制和知识传递都非常重要。它确保了设计文档的及时更新和一致性。

系统设计和架构：

UML支持从需求分析到系统设计的全生命周期建模，包括用例图、类图、活动图、状态图、序列图等。通过这些图，开发团队可以详细描述系统的静态结构和动态行为，帮助进行系统架构设计和组件划分。

需求分析：用例图帮助捕捉和展示用户需求，确保开发的系统能够满足用户的实际需求。这在需求获取和需求变更管理中尤为重要。

代码生成与逆向工程：现代开发工具支持从UML模型生成代码（如类图生成代码骨架）和从代码生成UML模型（逆向工程），这使得模型和代码之间保持同步，有助于提高开发效率和减少错误。

使用UML建模的必要性

提高沟通效率：UML提供了一种标准化的表示方法，使得开发团队成员之间，以及与客户和其他利益相关者之间的沟通更加顺畅、准确，减少了由于误解而导致的开发偏差。

简化复杂系统：通过分层和模块化的方式，UML可以将复杂的系统拆解成更小、更易于理解和管理的部分，从而简化系统设计和开发过程。

改进设计质量：使用UML进行建模有助于在设计阶段发现潜在的问题和瓶颈，并通过优化模型来改进系统的设计质量，降低开发和维护成本。

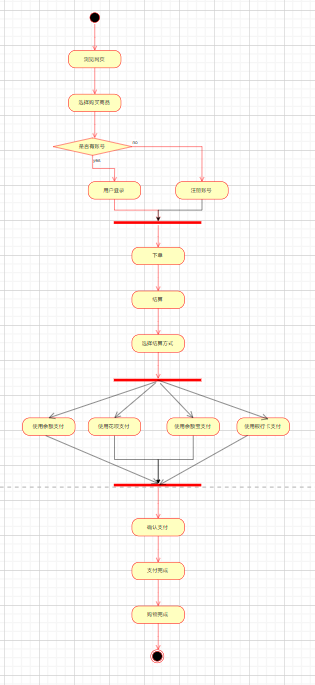
增强可维护性：UML模型提供了系统的结构和行为的清晰文档，这对于系统的维护和扩展非常有帮助。开发人员可以通过模型快速理解系统的设计和逻辑，减少了维护的难度和出错的可能。

总结

UML在软件工程中扮演着重要的角色，通过提供一种标准化的、图形化的建模语言，帮助开发团队进行有效的沟通、设计、文档生成和系统维护。使用UML进行建模不仅可以提高开发效率和设计质量，还能增强系统的可维护性和扩展性，是现代软件工程中不可或缺的工具。

### 分小组讨论利用“支付宝”进行网上购物的活动过程，并画出活动图，关于支付宝的相关说明可以登录<http://www.alipay.com/>查看。

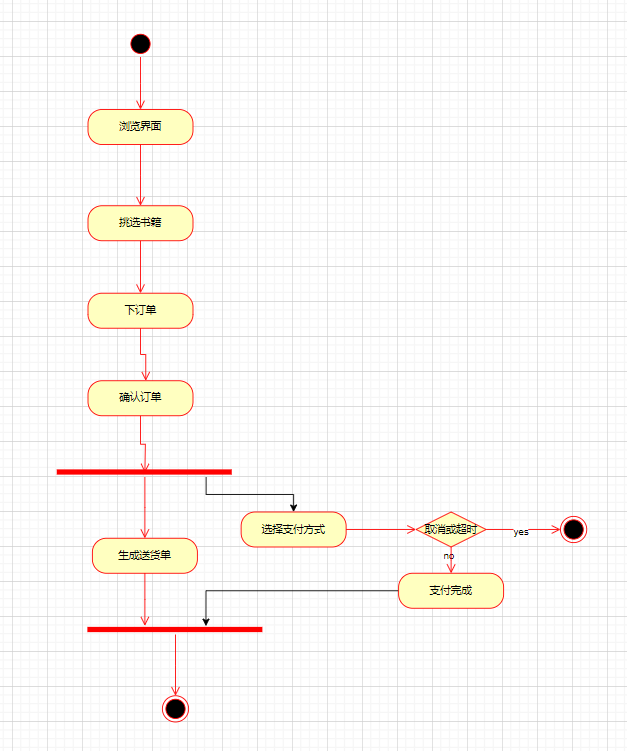
解：如图：

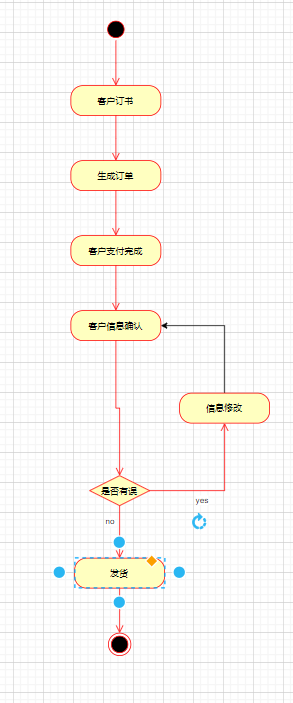


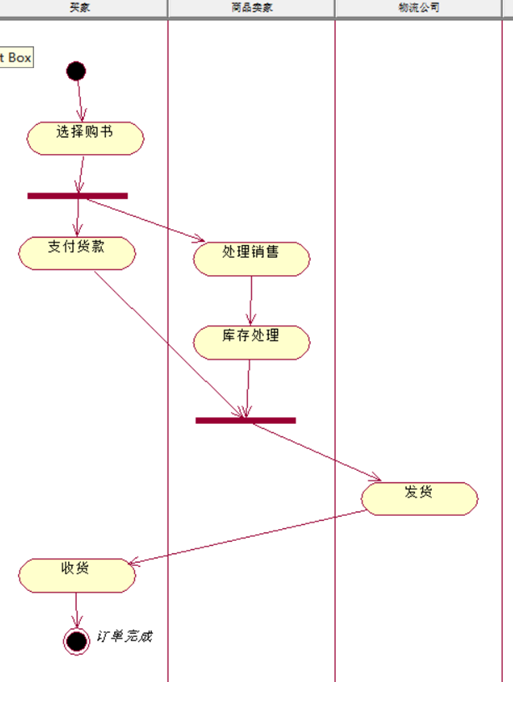
### 3.针对现有的网上书店系统，画出用户购书，商家发货等过程的活动图。

解：

有用户购书：



商家发货：

泳道实现：

# UML实验九 组件图

## 实验内容

### 什么是组件？组件有何作用？

解：组件是软件工程中的一个概念，它表示一个独立的模块或部分，具有明确的功能，可以被重复使用。组件通常封装了其实现细节，并通过明确的接口与其他组件进行交互。组件的作用主要包括：

模块化：通过将系统划分为独立的组件，可以提高系统的模块化程度，使得系统更容易维护和扩展。

重用性：组件具有独立性和明确的功能，可以在不同的系统中重复使用，减少重复劳动。

可维护性：组件的独立性使得它们可以独立进行开发、测试和维护，降低了系统的复杂性。

灵活性：通过重新组合组件，可以快速构建或修改系统，提高开发的灵活性和效率。

### 组件的要素是什么？

解：接口：定义组件与外部交互的方式，包括提供的服务接口和所需的服务接口。

实现：组件的内部实现，包括代码和逻辑。

依赖关系：组件对其他组件或外部资源的依赖。

配置：组件的配置参数和环境要求。

### 组件和类的关系是怎样的？它们有何异同之处？

解：组件和类是软件设计中的两个不同层次的概念：组件：通常表示系统中的一个功能模块，可以由多个类组成。组件是较高层次的抽象，注重于功能的划分和模块化。类：是面向对象编程中的基本构建单元，表示对象的蓝图或模板。类关注的是对象的状态和行为。

相同点：都是软件构建中的基本单元。都具有封装性，可以隐藏内部实现细节。都可以通过接口与外部交互。

不同点：组件通常包含多个类，并关注系统的功能模块划分和模块化。类是面向对象的基本单元，关注对象的属性和方法。组件可以独立部署和重用，而类通常是程序内部的构建块。

### UML中组件有哪些表示方法，分别代表什么含义？

解：在UML中，组件通常用矩形表示，内部可以包含组件的名称和接口。组件的符号和含义包括：

组件：用一个矩形表示，右上角带有两个小矩形，表示这是一个组件。

接口：接口可以用一个带有圆形或半圆形的连线表示：

提供的接口：用一个圆形表示，从组件内部指向外部。

所需的接口：用一个半圆形表示，从外部指向组件内部。

### 组件中的接口有哪些？分别如何表示？

解：组件中的接口是组件与外部交互的主要方式，分为：

提供的接口：组件提供给外部使用的功能或服务。用一个圆形表示，从组件内部指向外部。

所需的接口：组件运行时所需的外部功能或服务。用一个半圆形表示，从外部指向组件内部。

### 在一张基本组件图中，组件之间最常见的关系是什么？

解：在基本组件图中，组件之间最常见的关系包括：

依赖关系：一个组件依赖于另一个组件提供的功能或服务，通常用虚线箭头表示。

接口关系：组件通过接口进行交互，包括提供的接口和所需的接口，用圆形和半圆形表示。

组件图通过这些关系来展示组件之间的交互和依赖，从而帮助开发人员理解和设计系统的架构和模块化。

## UML实验结果

**[分析与讨论]**

１．讨论组件图适用于哪些建模需求。

**解：组件图是一种UML图，用于描述系统中各个组件及其相互之间的关系。组件图主要适用于以下建模需求：**

1. 系统架构设计：

- 描述系统的高层次结构，展示系统的主要构建块及其相互关系。

- 帮助架构师和开发人员理解系统的模块化设计和组件之间的依赖关系。

2. 模块化设计：

- 展示系统中的各个模块（组件）及其接口，帮助确定模块的职责和功能范围。

- 支持模块化开发和维护，便于团队协作和独立开发。

3. 重用性分析：

- 识别系统中的可重用组件，分析哪些组件可以在不同系统或项目中重复使用。

- 促进组件库的建立和维护，提高开发效率。

4. 部署规划：

- 描述组件在物理节点上的分布，帮助进行系统部署和配置管理。

- 规划组件的物理部署位置，确定组件间的通信方式和依赖关系。

5. 系统集成：

- 展示系统内部和外部组件的集成关系，明确组件间的接口和通信协议。

- 帮助解决系统集成中的依赖问题，确保组件间的兼容性和协作。

6. 接口定义：

- 明确组件对外提供的服务接口和所需的服务接口，定义接口的规范和协议。

- 有助于接口的设计和验证，确保组件间的正确交互。

7. 维护和演化：

- 通过组件图可以清晰地看到系统的模块划分和依赖关系，有助于系统的维护和演化。

- 在系统演化过程中，可以方便地调整组件之间的关系和接口，降低维护成本。

实际应用场景

- 软件产品线开发：组件图用于展示产品线中各个产品的公共组件和特定组件，支持产品定制和变种管理。

- 企业系统集成：在企业级系统集成项目中，组件图帮助描述不同子系统之间的集成关系和接口规范。

- 微服务架构设计：在微服务架构中，组件图用于展示各个微服务的职责、接口及其相互依赖关系，支持微服务的开发和部署。

通过组件图，开发人员可以更好地理解和设计系统的结构，提高系统的模块化和可维护性，促进团队协作和系统演化。

# UML实验十 UML综合实验

## 实验内容

在Java综合实验部分，我们已经完成了学生管理和书籍管理的基本功能。现在要利用已有的学生类和书籍类，完成一个图书借阅系统。系统由图书管理员操作，进行图书的借出和归还业务。

## UML结果文件说明

1. 完成系统的类图设计。

解：由要求可知：该类图由如下java语言表示建模：

Student类：

class Student {

private String studentId;

private String name;

private String department;

public String getStudentId() { return studentId; }

public void setStudentId(String studentId) { this.studentId = studentId; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

// toString method

@Override

public String toString() {

return "Student{" +

"studentId='" + studentId + '\'' +

", name='" + name + '\'' +

", department='" + department + '\'' +

'}';

}

}

Book类：

class Book {

private String bookId;

private String title;

private String author;

private int stock;

public String getBookId() { return bookId; }

public void setBookId(String bookId) { this.bookId = bookId; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getAuthor() { return author; }

public void setAuthor(String author) { this.author = author; }

public int getStock() { return stock; }

public void setStock(int stock) { this.stock = stock; }

// toString method

@Override

public String toString() {

return "Book{" +

"bookId='" + bookId + '\'' +

", title='" + title + '\'' +

", author='" + author + '\'' +

", stock=" + stock +

'}';

}

}

BorrowInfo类：

class BorrowInfo {

private Student student;

private Book book;

private Date borrowDate;

private Date returnDate;

public Student getStudent() { return student; }

public void setStudent(Student student) { this.student = student; }

public Book getBook() { return book; }

public void setBook(Book book) { this.book = book; }

public Date getBorrowDate() { return borrowDate; }

public void setBorrowDate(Date borrowDate) { this.borrowDate = borrowDate; }

public Date getReturnDate() { return returnDate; }

public void setReturnDate(Date returnDate) { this.returnDate = returnDate; }

// toString method

@Override

public String toString() {

return "BorrowInfo{" +

"student=" + student +

", book=" + book +

", borrowDate=" + borrowDate +

", returnDate=" + returnDate +

'}';

}

}

BorrowManager类：

class BorrowManager {

private List<BorrowInfo> borrowInfos;

public BorrowManager() {

this.borrowInfos = new ArrayList<>();

}

public void addBorrowInfo(BorrowInfo borrowInfo) {

borrowInfos.add(borrowInfo);

}

public List<BorrowInfo> getBorrowInfos() {

return borrowInfos;

}

// Methods to manage borrowInfos, e.g., get borrow info by student or book

}

BookManager类：

class BookManager {

private List<Book> books;

public BookManager() {

this.books = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

public void borrowBook(Book book) {

if (book.getStock() > 0) {

book.setStock(book.getStock() - 1);

}

}

public void returnBook(Book book) {

book.setStock(book.getStock() + 1);

}

public Book getBookById(String bookId) {

for (Book book : books) {

if (book.getBookId().equals(bookId)) {

return book;

}

}

return null;

}

public List<BorrowInfo> getBorrowInfo(Book book, BorrowManager borrowManager) {

List<BorrowInfo> result = new ArrayList<>();

for (BorrowInfo info : borrowManager.getBorrowInfos()) {

if (info.getBook().equals(book)) {

result.add(info);

}

}

return result;

}

}

StudentManager类：

class StudentManager {

private List<Student> students;

public StudentManager() {

this.students = new ArrayList<>();

}

public void addStudent(Student student) {

students.add(student);

}

public Student getStudentById(String studentId) {

for (Student student : students) {

if (student.getStudentId().equals(studentId)) {

return student;

}

}

return null;

}

public List<BorrowInfo> getBorrowInfo(Student student, BorrowManager borrowManager) {

List<BorrowInfo> result = new ArrayList<>();

for (BorrowInfo info : borrowManager.getBorrowInfos()) {

if (info.getStudent().equals(student)) {

result.add(info);

}

}

return result;

}

}

2.设计图书借出的工作流程，并用活动图建模。

3.学生归还图书时，图书管理员核对完信息后，录入还书信息，提交后由系统进行还书处理。绘制还书处理的顺序图。

解：

1. 图书管理员：接收到学生的还书请求。

2. 图书管理员：向借阅信息管理器查询借阅信息。

3. 借阅信息管理器向图书管理器验证归还的图书信息。

4. 图书管理器确认图书信息有效后，通知借阅信息管理器。

5. 借阅信息管理器向学生管理器更新借阅信息。

6. 学生管理器确认更新信息后，通知借阅信息管理器。

7. 借阅信息管理器向图书管理器更新图书库存。

8.图书管理器更新库存后，通知借阅信息管理器。

9.借阅信息管理器确认还书处理完成，通知图书管理员。

10.图书管理员向学生确认归还图书。

解：

public class Book {

private String bookId;

private String title;

private String author;

private int stock;

private String status; // 新增属性，表示书本状态

public Book(String bookId, String title, String author, int stock) {

this.bookId = bookId;

this.title = title;

this.author = author;

this.stock = stock;

this.status = "Available"; // 初始状态为“可借”

}

// 各属性的getter和setter方法，包括状态属性

public String getBookId() { return bookId; }

public void setBookId(String bookId) { this.bookId = bookId; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getAuthor() { return author; }

public void setAuthor(String author) { this.author = author; }

public int getStock() { return stock; }

public void setStock(int stock) { this.stock = stock; }

public String getStatus() { return status; }

public void setStatus(String status) { this.status = status; }

@Override

public String toString() {

return "Book{" +

"bookId='" + bookId + '\'' +

", title='" + title + '\'' +

", author='" + author + '\'' +

", stock=" + stock +

", status='" + status + '\'' +

'}';

}

}

public class BookManager {

private List<Book> books;

public BookManager() {

this.books = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

public void borrowBook(Book book) {

if (book.getStock() > 0 && "Available".equals(book.getStatus())) {

book.setStock(book.getStock() - 1);

book.setStatus("Borrowed");

}

}

public void returnBook(Book book) {

book.setStock(book.getStock() + 1);

book.setStatus("Available");

}

public void reportLost(Book book) {

book.setStatus("Lost");

}

public Book getBookById(String bookId) {

for (Book book : books) {

if (book.getBookId().equals(bookId)) {

return book;

}

}

return null;

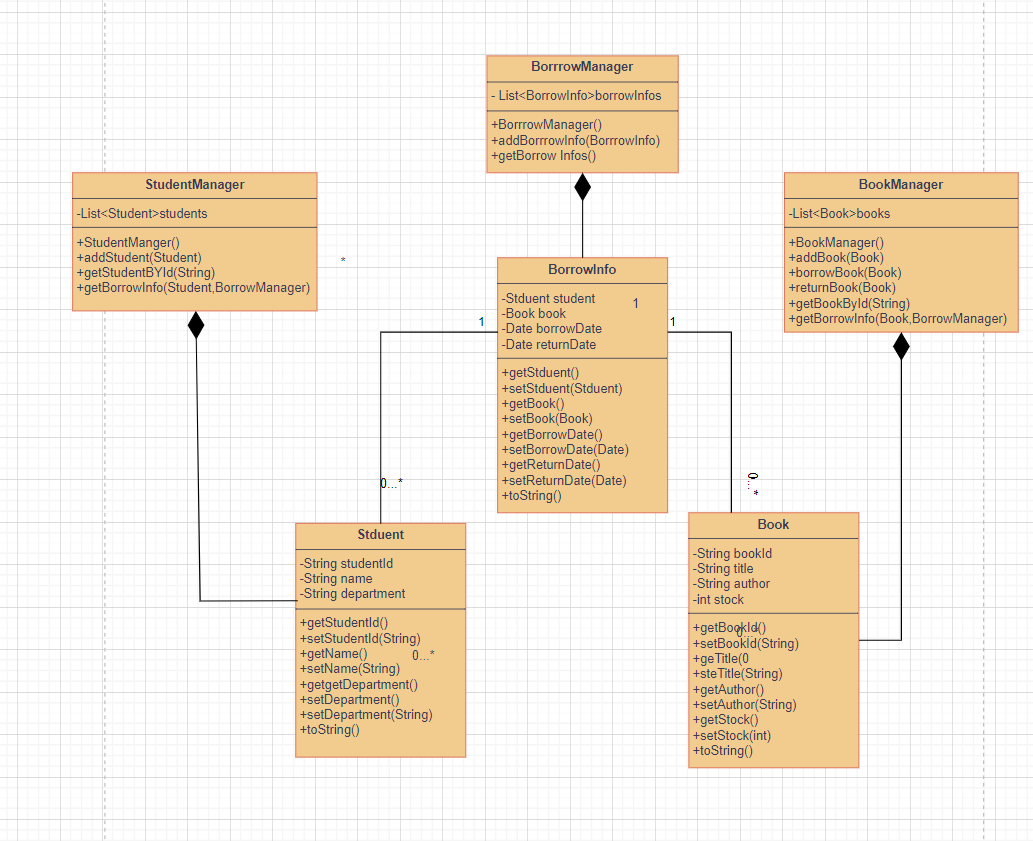
}

// 其他管理书本的方法

}

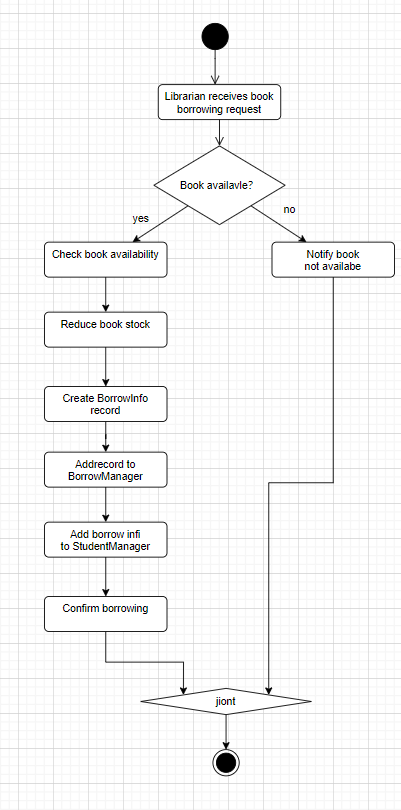
## UML实验结果

1. 类图

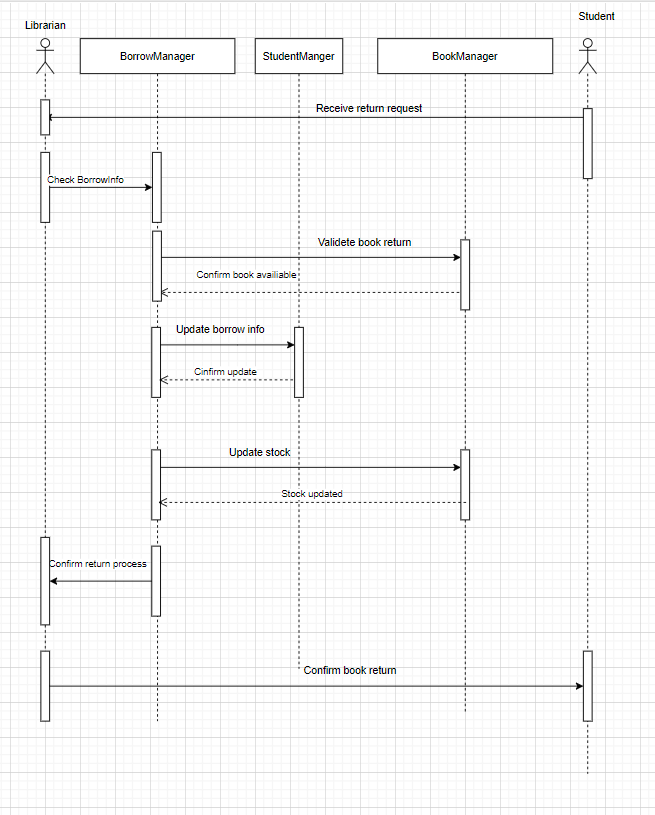


1. 活动图：

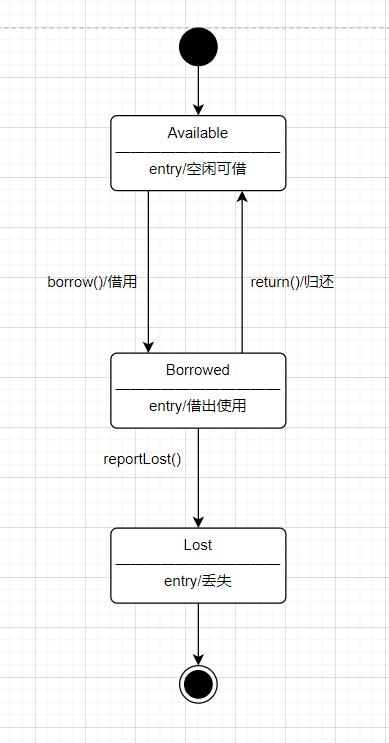
如图：



1. 顺序图



1. 状态图



# 实验总结

## 实验中遇到的问题以及解决方法

1. 问题：

在判断选课系统中的依赖关系时，由于类图为双向依赖，不清楚依赖关系具体。

解决方法：

通过判断类图的多重性以及常识，得到老师、学生依赖课程，课程依赖课本。

### 问题：

一开始对活动图的概念不是很清楚，导致一开始画出的图并不符合要求，而且在画支付宝、用户购书和商家发货的活动图的时候，并没有找到一个平衡点，不能太详细也不能太概括。

解决方法：将大的事件拆开，按照一次一步进行拆分，保证一次只完成一件事，而没有和其他过程进行耦合。

### 问题：

对包的组织结构不够清晰，难以理解和维护。

解决方法：应该使用层次结构来组织包，以便于理解和维护。

## 实验体会

在本次实验中，我深入学习和掌握了UML中的包、组件和活动图的相关概念、作用及其适用场景，并学会了如何设计和改进这些图。首先，我学习了包的概念以及包与包之间的依赖关系。通过实验，我掌握了如何建立包与包之间的依赖关系，并了解了在建立包图过程中需要遵循的四大原则：

1. 重用等价原则：确保包中的类具有相似的重用性。

2. 共同闭包原则：包中的类应该一起被修改，保持包的一致性。

3. 共同重用原则：包中的类应该一起被重用，避免不必要的依赖。

4. 非循环依赖原则：包与包之间不应存在循环依赖，确保包图的层次结构清晰。

特别是非循环依赖原则，这是包图设计中必须注意不能犯的错误。我还学习了如何消除包图中的循环依赖关系。通常在系统开发前期，开发原则以共同闭包为主，而在系统稳定后，可以对包进行一些重构，此时应以共同重用原则为主。通过实验，我还练习了如何改进包图设计，这有助于我们在未来设计UML图时能够设计出合理、清晰的包图。

其次，我学习了组件的概念和组件图的适用场景。通过学习，我掌握了组件的几大要素，并了解了组件和类的相同与不同之处。通过根据类图绘制组件图的过程，我掌握了UML组件图的绘制方法，包括如何绘制组件图中的各个组件及其接口，以及组件之间的依赖关系。通过实验，我掌握了阅读和绘制组件图时需要注意的事项，并通过查询资料了解了组件图的应用场景和使用方法。

组件图可以用来描述组成软件的组件之间的关系以及组件的内部结构。可以用组件图描述系统的文件组成，适用于面向对象系统的物理方面建模。组件图能够帮助客户理解最终的系统结构，使开发工作有一个明确的目标，有利于软件系统的重用。因此，我们应该掌握组件图的画法，以便在未来的软件设计中应用。

最后，我学习了UML活动图（Activity Diagram）的概念及其用途。活动图用于描述系统中活动的流程和控制流，适用于以下场景：

1. 业务流程建模：描述业务流程的步骤和顺序，帮助分析和优化业务流程。

2. 系统功能流程：描述系统中各功能模块的操作流程和交互关系。

3. 用户交互流程：描述用户与系统之间的交互步骤，帮助设计和评估用户界面。

通过实验，我掌握了活动图的绘制方法，包括如何表示活动、决策节点、合并节点、并行分支和同步节点等元素。活动图的绘制可以帮助我们清晰地展示系统中各活动的执行顺序和逻辑关系，有助于发现和解决流程中的问题，优化系统设计。

通过本次实验，我不仅掌握了包图、组件图和活动图的相关知识和技能，还增强了我对UML图设计的理解和应用能力。在系统开发的不同阶段，这些图都发挥着重要作用：

- 包图：有助于系统的模块化设计和维护，确保系统结构的清晰和依赖关系的合理。

- 组件图：帮助描述系统的物理构成和组件间的交互，支持模块化开发和重用。

- 活动图：用于描述系统中的流程和控制流，优化业务流程和系统功能设计。

这些UML图的综合应用，使得系统设计更加直观和合理，提高了系统的可维护性和扩展性，为未来的软件开发工作奠定了坚实的基础。通过不断练习和应用这些工具，我在未来的系统设计和开发工作中能够更好地应用这些方法，设计出合理、清晰的UML图。