

**软件设计模式**

第10小组项目文档

顶峰考研云

指导教师：史扬

专业名称：软件工程

2024年1月2日

2051498 储岱泽

2051828 莫益萌

2050780 焦骜

2051973 韩嘉睿

2151622 沙坚

2152835 周文玥

2154296 傅佳恒

2152826 夏尧民

2151929 王宜沣

2152494 徐嘉琪

2152402 段婷婷

目录

[1. 项目简介 5](#_Toc2096324997)

[选题背景 5](#_Toc1317511897)

[功能点介绍 5](#_Toc718568662)

[2. 设计模式汇总表 6](#_Toc1682955153)

[3. 设计模式详细介绍 10](#_Toc920141834)

[3.1 抽象工厂模式（GOF模式） 10](#_Toc794061991)

[3.1.1 实现描述 10](#_Toc1336500279)

[3.1.2 类图 11](#_Toc2028725180)

[3.1.3 代价分析 12](#_Toc1186236841)

[3.2 适配器模式（GOF模式） 12](#_Toc1991891586)

[3.2.1 实现描述 12](#_Toc599312819)

[3.2.1 类图 13](#_Toc952244503)

[3.2.1 代价分析 13](#_Toc1325224477)

[3.3 桥接模式（GOF模式） 14](#_Toc1494881902)

[3.3.1 实现描述 14](#_Toc1068940661)

[3.3.2 类图 15](#_Toc1984982272)

[3.3.3 代价分析 15](#_Toc438589359)

[3.4 建造者模式（GOF模式） 16](#_Toc1207480209)

[3.4.1 实现描述 16](#_Toc399408513)

[3.4.2 类图 17](#_Toc1972481116)

[3.4.3 代价分析 17](#_Toc785057873)

[3.5 责任链模式（GOF模式） 18](#_Toc328144343)

[3.6.1 实现描述 18](#_Toc383967305)

[3.6.2 类图 19](#_Toc150135900)

[3.6.3 代价分析 19](#_Toc40786075)

[3.6 命令模式（GOF模式） 20](#_Toc444279132)

[3.6.1实现描述 20](#_Toc198730905)

[3.6.2类图 21](#_Toc733249250)

[3.6.3 代价分析 22](#_Toc1458978264)

[3.7 组合模式（GOF模式） 22](#_Toc1079401602)

[3.7.1 实现描述 22](#_Toc1708358605)

[3.7.2 类图 23](#_Toc526713845)

[3.7.3 代价分析 23](#_Toc551999981)

[3.8 数据访问对象模式（架构模式） 24](#_Toc334325627)

[3.8.1 实现描述 24](#_Toc1193592437)

[3.8.2 类图 25](#_Toc1063342032)

[3.8.3 代价分析 25](#_Toc230621490)

[3.8.4 出处 26](#_Toc1994883242)

[3.9 装饰器模式（GOF模式） 26](#_Toc1487951330)

[3.9.1 实现描述 26](#_Toc550933995)

[3.9.2 类图 27](#_Toc1745651748)

[3.9.3 代价分析 27](#_Toc247343322)

[3.10 外观模式（GOF模式） 27](#_Toc1718355909)

[3.10.1 实现描述 27](#_Toc1047677707)

[3.10.2 类图 29](#_Toc1100799796)

[3.10.3 代价分析 29](#_Toc570552467)

[3.11 工厂方法模式（GOF模式） 30](#_Toc760829014)

[3.11.1 实现描述 30](#_Toc1135604060)

[3.11.2 类图 31](#_Toc1410265531)

[3.11.3 代价分析 31](#_Toc555767578)

[3.12 过滤器模式（其他模式） 32](#_Toc1379302643)

[3.12.1 实现描述 32](#_Toc2001035183)

[3.12.2 类图 32](#_Toc1804408661)

[3.12.3 代价分析 33](#_Toc2079786140)

[3.12.4 出处 33](#_Toc374332761)

[3.13 享元模式（GOF模式） 33](#_Toc1431112064)

[3.13.1 实现描述 33](#_Toc883613248)

[3.13.2 类图 34](#_Toc1038440131)

[3.13.3 代价分析 34](#_Toc463682548)

[3.14 解释器模式（GOF模式） 34](#_Toc2041912920)

[3.14.1 实现描述 34](#_Toc1641767380)

[3.14.2 类图 36](#_Toc166975357)

[3.14.3 代价分析 36](#_Toc1741182117)

[3.15 迭代器模式（GOF模式） 36](#_Toc288182750)

[3.15.1 实现描述 36](#_Toc911855265)

[3.15.2 类图 37](#_Toc1108133863)

[3.15.3 代价分析 37](#_Toc1427648657)

[3.16 延迟初始化模式（其他模式） 38](#_Toc656190268)

[3.16.1 实现描述 38](#_Toc1261306931)

[3.16.2 类图 39](#_Toc974509780)

[3.16.3 代价分析 39](#_Toc1875580438)

[3.16.4 出处 40](#_Toc2115450800)

[3.17 中介者模式（GOF模式） 40](#_Toc642335868)

[3.17.1 实现描述 40](#_Toc338640007)

[3.17.2 类图 41](#_Toc690933099)

[3.17.3 代价分析 41](#_Toc1068515564)

[3.18 备忘录模式（GOF模式） 42](#_Toc1282827934)

[3.18.1 实现描述 42](#_Toc1900754505)

[3.18.2类图 43](#_Toc14232763)

[3.18.3代价分析 43](#_Toc839362924)

[3.19 MVC模式（架构模式） 43](#_Toc352586525)

[3.19.1 实现描述 43](#_Toc1014343602)

[3.19.2 类图 44](#_Toc1347728928)

[3.19.3 代价分析 44](#_Toc1770067987)

[3.19.4 出处 45](#_Toc441695618)

[3.20 空对象模式（其他模式） 46](#_Toc1874767694)

[3.20.1 实现描述 46](#_Toc1340564274)

[3.20.2 类图 46](#_Toc1612812441)

[3.20.3 代价分析 46](#_Toc1000103453)

[3.20.4 出处 47](#_Toc384229502)

[3.21 观察者模式（GOF模式） 47](#_Toc261913585)

[3.21.1 实现描述 47](#_Toc1787630392)

[3.21.2 类图 48](#_Toc1407776814)

[3.21.3 代价分析 48](#_Toc1677573899)

[3.22 原型模式（GOF模式） 48](#_Toc671719030)

[3.22.1 实现描述 48](#_Toc260204931)

[3.22.2 类图 49](#_Toc987570025)

[3.22.3 代价分析 49](#_Toc188302512)

[3.23 代理模式（GOF模式） 50](#_Toc1556907153)

[3.23.1 实现描述 50](#_Toc1997765423)

[3.23.2 类图 50](#_Toc536643516)

[3.23.3 代价分析 50](#_Toc2083739659)

[3.24 雇工模式（其他模式） 51](#_Toc249133537)

[3.24.1 实现描述 51](#_Toc1741728356)

[3.24.2 类图 52](#_Toc878887035)

[3.24.3 代价分析 52](#_Toc1061873179)

[3.24.4 出处 52](#_Toc1313412883)

[3.25 简单工厂模式（其他模式） 53](#_Toc545917068)

[3.25.1 实现描述 53](#_Toc1178021892)

[3.25.2 类图 53](#_Toc1362197151)

[3.25.3 代价分析 53](#_Toc124356190)

[3.25.4 出处 54](#_Toc552896799)

[3.26 状态模式（GOF模式） 54](#_Toc374760224)

[3.26.1 实现描述 54](#_Toc25548117)

[3.26.2 类图 56](#_Toc2037956666)

[3.26.3 代价分析 56](#_Toc1720999459)

[3.27 单例模式（GOF模式） 57](#_Toc380665970)

[3.27.1 实现描述 57](#_Toc499173377)

[3.27.2 类图 57](#_Toc1535822057)

[3.27.3 代价分析 57](#_Toc1955358706)

[3.28 策略模式（GOF模式） 58](#_Toc771521701)

[3.28.1 实现描述 58](#_Toc458968121)

[3.28.2 类图 59](#_Toc115949623)

[3.28.3 代价分析 59](#_Toc997645932)

[3.29 模板模式（GOF模式） 59](#_Toc2030346995)

[3.29.1 实现描述 59](#_Toc526794135)

[3.29.2 类图 60](#_Toc1901434011)

[3.29.3 代价分析 60](#_Toc697271870)

[3.30 访问者模式（GOF模式） 61](#_Toc230057411)

[3.30.1 实现描述 61](#_Toc1104342077)

[3.30.2 类图 62](#_Toc2123610765)

[3.30.3 代价分析 62](#_Toc347914215)

[附录 63](#_Toc1943724371)

[项目分工说明与成员贡献 63](#_Toc654265233)

[非经典设计模式出处 64](#_Toc1119498391)

# 1.项目简介

## 选题背景

2024年考研在即，同学们都已经在如火如荼的准备啦！为了在新一年的考研中取得理想的成绩，同济大学民间推出了一款名叫“顶峰考研云”的应用，在该应用中，系统存储了大量往年的考研真题以及相关习题，并邀请名师进行考题预测，进行模拟试卷的编写，同学们可以在该应用中和研友一起进行模拟考试PK，考题交流，同时也能进行教辅资料的购买。现在，该应用受到了研友们的广泛使用，5位软件学院的小伙伴结成学习小组正在图书馆进行模拟考试PK。我们的项目使用多种软件设计模式模拟了同学们使用“顶峰考研云”进行模拟考试的过程，加深了同学们对于软件设计模式的理解和使用。

## 功能点介绍

系统提供了一个全面的虚拟学习和考试环境，旨在通过模拟真实的学习和考试体验来提高学生的学习效率和成绩。

在本系统中，学生可以创建和定制自己的角色，包括选择专业和学历水平。系统会根据学生的选择为其分配不同的属性和功能。学生可以使用角色进行多种活动，如参加模拟考试、购买教辅资料和文具、查看和修改个人信息。系统提供多种支付方式，包括Alipay、微信支付和银行卡支付，以便于完成购物和其他付款操作。

本系统涵盖多种模拟考试，例如高等数学、大学英语等，每场考试都具有独特的设置和评分标准。考试成绩会更新学生的状态和属性，这些成绩可以被打印出来以供学生查看。此外，系统还模拟了学生参加考试的完整流程，包括报名、排队、信息检查和入场等步骤。

为了增强用户体验，系统提供了一个模拟的购物中心，学生可以在其中购买书籍和学习工具。书籍购买过程中可以选择额外的配套文具，增加购物的灵活性和趣味性。学生还可以加入学习小组，与其他学生一起学习和交流。

# 2.设计模式汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **编号** | **设计模式名称** | **涉及的文件** | **功能** | **备注说明** |
| 1 | [抽象工厂模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#Ds9nd9yqnokCx7xcKSKcRiksnFf)（Abstract Factory pattern） | src/AbstractFactory | 创建监考老师及考试科目信息 | 考试科目和监考老师可以被抽象为一系列的产品，在此基础上使用抽象工厂模式创建监考老师及考试科目信息 |
| 2 | [适配器模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#AQMZddLfroRUmUxyHKLcvwsVnyd)  (Adapter pattern) | src/Adapter | 实现多种支付方式，包括Alipay、微信支付以及银行卡支付。 | 给Alipay配备适配器，从而当用户选择Alipay时可以选择Alipay内配备的两种方式进行支付（花呗和余额宝）。 |
| 3 | [桥接模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#NqWSdXpuKohfaGxj8OsczDZSnQf)  (Bridge pattern) | src/Bridge | 实现学生物品的购买与查看。 | 桥接模式将购物的行为与具体的购物类型分离，从而提供了更高的灵活性和扩展性。 |
| 4 | [建造者模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#UXbPdwnbvoIRSlx9otbcFTeznzB)  (Builder pattern) | src/Builder | 创建模拟考试 | 将模拟考试这样一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示，如高等数学考试、大学英语考试等。 |
| 5 | [责任链模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#T02td3emvoG4dexvvhAcMI5Unbc)  (Chain of responsibility pattern) | src/ChainOfResponsibility | 管理学生间的竞争逻辑 | 责任链模式通过创建一个链式结构，允许多个对象有机会处理请求。在竞争中中，每个竞争者都是链上的一个节点，负责处理竞争逻辑，并将请求传递给链上的下一个节点。这种模式提高了处理请求的灵活性和扩展性。 |
| 6 | [命令模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#IYlLd9Lo2oXKnhxHkKBc7hVMn8d)  (Command pattern) | src/Command | 进行顶峰教辅采购中心教辅资料以及文具的购买 | 将购买教辅材料以及文具的命令抽象成对象。 |
| 7 | [组合模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#HPprdSkLZoMkHEx8FbpcOds7nHe)  (Composite pattern) | src/Composite | 输出菜单 | 采用了组合模式来模拟文件与文件夹的关系，表示菜单中的单个选项和二级菜单 |
| 8 | [DAO模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#W2Zed3wU2oP0Rxxyi3xcvWWbnHc)  (DAO pattern) | src/DAO | 可以进行更新和删除学生编号的操作 | 提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了学生信息的访问操作。 |
| 9 | [装饰器模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#IMVqdWtKEo7cZ7xRKwqcA8TOn9f)  (Decorator pattern) | src/Decorator | 实现购买书籍的时候可以购买配套的学习工具。 | 相当于额外可以顺便选购配套文具在装饰购买的书籍。 |
| 10 | [外观模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#UVn1d3aJBosSYGxjXrFctepcnef)  (Facade pattern) | src/Facade | 完成学生参加考试流程 | 学生参加考试后，需要完成报名、排队、信息检查和入场等四个步骤，将这些步骤抽象化，使用外观模式设计一个统一的高层接口，以简化流程。 |
| 11 | [工厂方法模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#V5undpNAfon5bwxSZBJcGO2Lnpc)  (Factory method) | src/FactoryMethod | 完成物品的创建 | 实现对于学生可用物品的构建，是学生购买物品的基础。 |
| 12 | [过滤器模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#IGOZd4tcboxoECxU5AWcgSitnHc)  (Filter pattern) | src/Filter | 筛选符合过滤条件的类 | 定义用户的要求，然后根据要求筛选数据 |
| 13 | [享元模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#Sqbwdf5x6ogC8TxsqARcuoemnmg)  (Flyweight pattern) | src/Flyweight | 模拟学生加入学习小组 | 模拟学生加入学习小组，将学生的一部分属性共享存储，使用工厂方法进行共享属性的声明、分配 |
| 14 | [解释器模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#DPtVdLZZYoVs6pxuM2LcRUkmnHh)  （Interpreter pattern） | src/Interpreter | 完成货币汇率的转换 |  |
| 15 | [迭代器模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#IXuBdy0YfoMPqTxk3jmcbF4hnnf)  (Iterater pattern) | src/Iterater | 获取学生信息列表、教师信息列表和考试信息列表 | 提供一种方法顺序访问一个集合对象中的各个元素，如学生、教师、考试的信息列表，而又无需暴露该对象的内部表示。 |
| 16 | [延迟初始化模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#ImkTd2x5WoSD8kxXMUTcrwonnHh)  (Lazy initialization pattern) | src/Singleton\_LazyInitialization | 在需要时创建顶峰考研云的示例 | 在确定要开始使用顶峰考研云时，才对其进行初始化，以降低初始化的开销。 |
| 17 | [中介者模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#XBaVdoR78o7Z12xTLAdczJQlnfd)  (Mediator pattern) | src/Mediator | 充当中介者，提供中介接口。 | 实现了中介者接口，负责协调和管理具体的同事类对象。 |
| 18 | [备忘录模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#EYzWdmb18omqzExiVxYcaQoznue)  (Memento pattern) | src/Memento | 记录学生的购买记录，并且实现购买的撤销以及重新选择操作 |  |
| 19 | [MVC模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#VxEgdKAiuouCpjxKDmbcrW0bnpd)  (MVC pattern) | src/MVC | 修改学生的姓名 | 提供了控制器来接收用户输入的姓名，并调用模型和视图来对原有的姓名进行修改和将新的姓名进行显示。 |
| 20 | [空对象模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#XXKgd8WDko6Y8jxjJsuch3Ezn7g)  (NullObject pattern) | src/NullObject | 处理空的对象 |  |
| 21 | [观察者模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#SwCfdJTm6oLYYsxtQaJcrWRCnoc)  (Observer pattern) | src/Observer | 实现考试成绩（排名）公示的功能 | 主题角色维护观察者列表，并在成绩或排名发生变化时通知所有注册的观察者。当主题角色更新排名和分数时，每个观察者实例会将最新的成绩和排名信息进行同步。 |
| 22 | [原型模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#FtpAdlHdooljbMxxPUGcEZXUnNb)  (Prototype pattern) | src/Prototype | 实现学生原型 | 提供可克隆的学生原型，提供功能加入学习小组和退出学习小组，并统计在学习小组学习的次数 |
| 23 | [代理模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#YLkydzjO4o5EqmxAQ7kc1ixVn3H)  (Proxy pattern) | src/Proxy | 实现成绩单打印功能 | 提供了实际的打印不同考试成绩单的实现，成为代理，控制其他类的访问。同时会检验打印成绩单的类是否存在。 |
| 24 | [雇工模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#DKYudEl9Zo9C9HxMTgdcd1UUnvb)  (Servant pattern) | src/Servant | 完成考试的准备工作 | 为一组类提供通用的考试准备功能，而不需要类实现这些功能，它是命令模式的一种扩展。 |
| 25 | [简单工厂模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#OcThdJ1Tpog34qxoCzhcNQzDndb)  (Simple Factory pattern) | src/SimpleFactory | 根据学历水平和目标专业完成学生注册 | 通过创建一个学生工厂类，通过调用其方法来获取所需对象的实例，从而隐藏了学生注册的具体逻辑。 |
| 26 | [状态模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#VqhNdmeZCo3ZBixrJ9NcmN09n1g)  (State pattern) | src/State | 输出和改变学生状态 | 学生的状态会影响 |
| 27 | [单例模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#TGvzdzYKVoJ749xVZ48cRVeUnLb)  (Singleton pattern) | src/Singleton\_LazyInitialization | 提供全局访问的一个类 |  |
| 28 | [策略模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#CGj6doazroLuTLxjFKTcZvFtnZe)  (Strategy pattern) | src/TemplateAndStrategy | 处理不同科目考试成绩的计算逻辑 | 每个科目的成绩计算方法有所不同，这通过不同的策略类实现。在上下文类中根据考试科目选择使用哪个策略类。定义策略接口，用于计算考试成绩；在具体策略类实现具体的算法。 |
| 29 | [模版模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#UjcPd9JrborN66xkXnvcCKtPn2c)  (Template pattern) | src/TemplateAndStrategy | 定义考试流程的固定模板 | 在抽象基类中提供考试流程的框架，具体科目的考试类继承自抽象基类类，并实现这些方法以提供特定的行为。 |
| 30 | [访问者模式](https://oiiej48md2b.feishu.cn/docx/Vd4MddxCRoI8BgxVMXBc9akHn7b#I4MsdqGPGo9cPnxkH7lcxgiZnQb)  (Visiter pattern) | src/Visitor | 查看某位学生在某场考试中的名次 | 构建Element定义接受访问操作，并构建具体元素实现抽象元素定义的接受操作接口，提供了排名查看操作 |

# 3. 设计模式详细介绍

## 3.1 抽象工厂模式（GOF模式）

### 3.1.1 实现描述

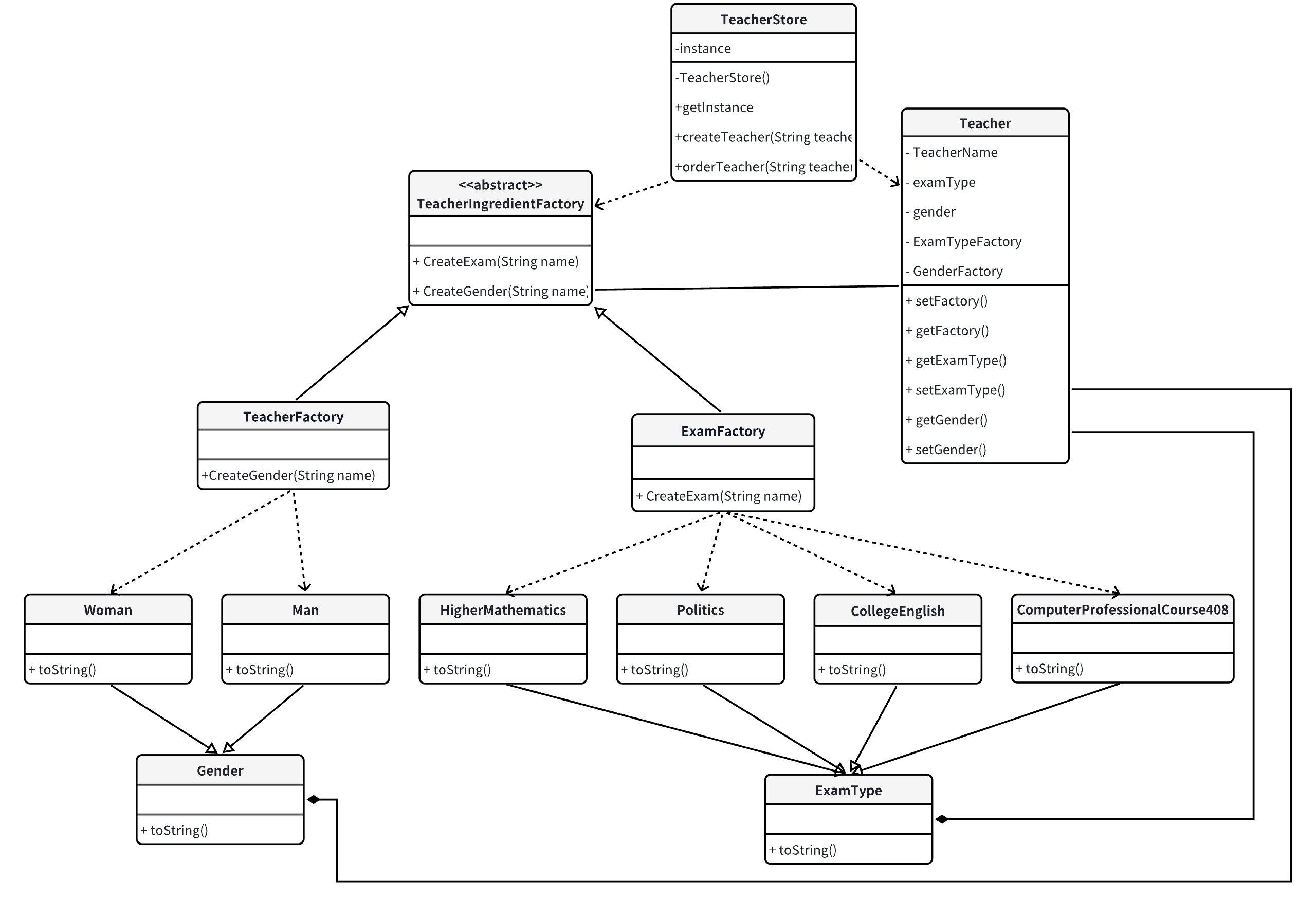
抽象工厂模式（Abstract Factory Pattern）是一种创建型设计模式，其核心在于构建一个所谓的超级工厂，用以生成其他工厂。这个超级工厂也被称作工厂的工厂。抽象工厂模式提供了一种创建对象的优雅方式，其中接口扮演着创建一系列相关或依赖对象的工厂的角色，而无需指定具体类。每个由此生成的工厂都遵循工厂模式，以提供对象。

在“顶峰考研云”系统的设计中，我们认识到考试科目和监考老师可以被抽象为一系列的产品，这为使用抽象工厂模式提供了理想的应用场景。具体实现涉及以下几个关键类：

1. **TeacherIngredientFactory (TeacherIngredientFactory.java)**接口，定义了创建监考老师所需的方法，包括 CreateExam 和 CreateGender。
2. **ExamFactory (ExamFactory.java)**实现了 TeacherIngredientFactory 接口，专门用于创建不同类型的考试（ExamType），如高等数学、大学政治、计算机专业课等。
3. **TeacherFactory (TeacherFactory.java)**也是 TeacherIngredientFactory 接口的实现，用于创建不同性别的老师（Gender），如男性或女性。
4. **TeacherStore (TeacherStore.java)**提供了创建和订购监考老师的方法。它使用 ExamFactory 和 TeacherFactory 来设置老师的考试类型和性别。

在本模式中，TeacherIngredientFactory 扮演了超级工厂的角色，而 ExamFactory 和 TeacherFactory 是其下属的两个具体工厂实现。这些工厂各自负责创建特定类型的考试和不同性别的老师。TeacherStore 类则利用这些工厂来构建具体的监考老师对象，从而实现了在客户端始终使用同一系列产品的目标。

### 3.1.2 类图



### 3.1.3 代价分析

1. 维护性方面，由于抽象工厂模式将对象的创建集中管理，当需要修改对象创建逻辑时，只需在工厂类中进行更改，这大大降低了维护成本。但是如果需要添加新的产品类别（如本项目中新增一种考试类型），可能需要修改工厂接口和所有实现该接口的工厂类，这会增加维护难度。
2. 拓展性方面，抽象工厂模式支持良好的扩展性，添加工厂类方便快捷。例如，如果要添加新的考试类型或老师类型，只需添加新的工厂类即可。但是扩展新的产品族（如新增不同的考试科目和对应老师）时，需要修改工厂接口和所有实现类，这可能导致代码的侵入性修改。
3. 复杂性方面，抽象工厂模式有助于分离对象的创建和使用，使得代码结构更清晰。但是引入多层抽象和多个类，增加了系统的复杂性，对于初学者或不熟悉设计模式的开发者来说，理解和使用可能较为困难。
4. 性能方面，抽象工厂模式可能会引入一定的性能开销，因为它涉及额外的间接层。但在大多数情况下，这种性能损失是可以接受的，特别是在不频繁创建对象的场景中。

## 3.2 适配器模式（GOF模式）

### 3.2.1 实现描述

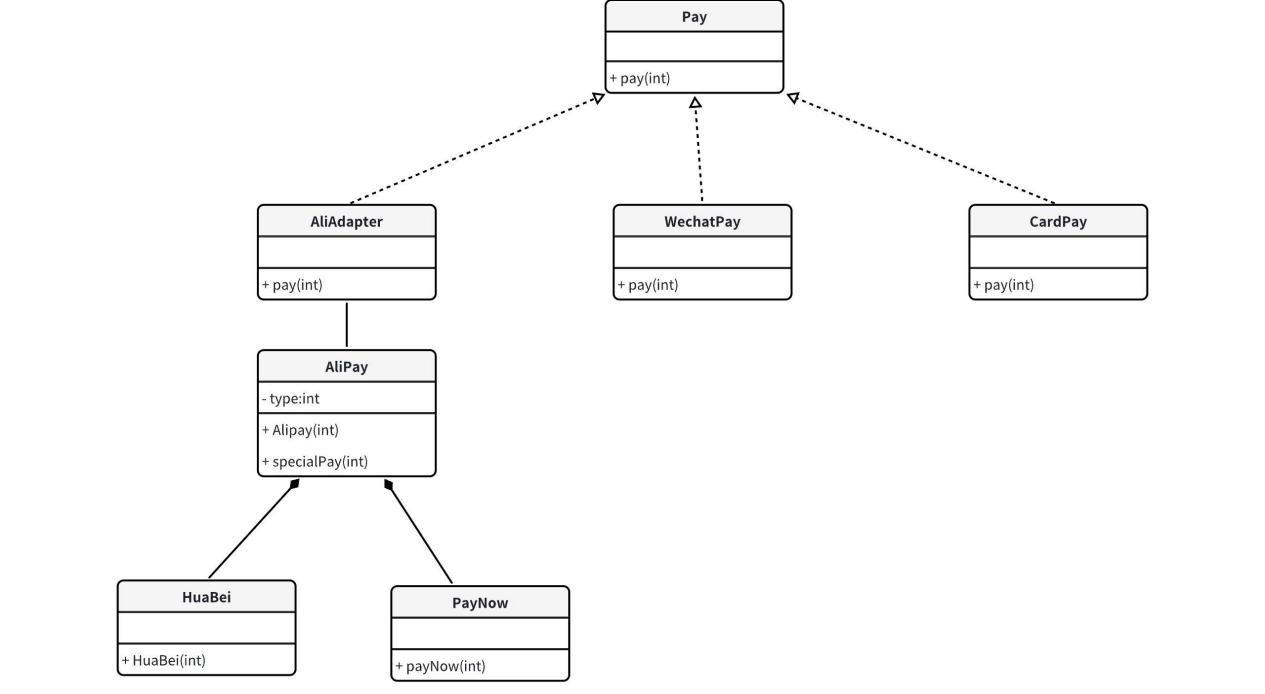
**适配器模式（Adapter Pattern）**是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。该模式主要是将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

在适配器模式中，通常包含以下几个角色：

1. **目标接口（Target）**：客户端所期望使用的接口。
2. **适配器（Adapter）**：实现了目标接口，并持有一个需要被适配的对象的引用。适配器将客户端的调用转换为对被适配对象的调用。
3. **被适配者（Adaptee）**：需要被适配的类或接口。其接口与目标接口不兼容。

在当用户在顶峰教辅超市选完资料之后，就需要进行付款。目前，常见的付款方式有Wechat支付、Ali支付以及银行卡支付，因为我们的支付系统涵盖了这几种支付方式。Wechat 支付和 Ali 支付都是支付方式，在只提供 WeChat 支付方式的时候，其他的支付方式并不适用与 WeChat支付，因此使用一个适配器来对接其他的支付方式，以实现不同的支付方式。同时，我们为支付宝设置了两种不同的支付方式：**余额支付和花呗支付**。对于一般支付，只需要调用 Pay 接口下的 pay()函数即可完成支付，如微信支付，只需要执行 WechatPay 下的同名函数 pay()，即可通过微信完成付款，对于银行卡支付，也只需要调用CardPay中的pay()函数就可以完成支付。**但是对于支付宝形式的支付，由于提供了两种更细化的支付方式，仅仅靠 pay()函数无法成功完成付款，需要一个适配器 Adapter 连接普通的支付方式 pay 和支付宝特殊形式的花呗与余额才能顺利完成支付。**该场景下设置了适配器 AliAdapter来继承 AliPay并且实现 Pay接口，在 pay()函数里调用 AliPay 独有的函数 specialPay()，完成本无法适配的支付。

### 3.2.1 类图



### 3.2.1 代价分析

**使用适配器模式的优点：**

1. 可以让任何两个没有关联的类一起运行。
2. 提高了类的复用。
3. 增加了类的透明度。
4. 灵活性好。

**使用适配器模式的缺点：**

1. 过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。
2. 由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

## 3.3 桥接模式（GOF模式）

### 3.3.1 实现描述

桥接模式（Bridge Pattern）是一种结构型设计模式，它的核心在于将抽象部分与实现部分分离，使它们可以独立变化。这种模式通过提供一个桥接结构，来连接不同的类层次结构，从而减少它们之间的耦合。在桥接模式中，抽象类和实现类不是通过继承，而是通过组合的方式关联起来。

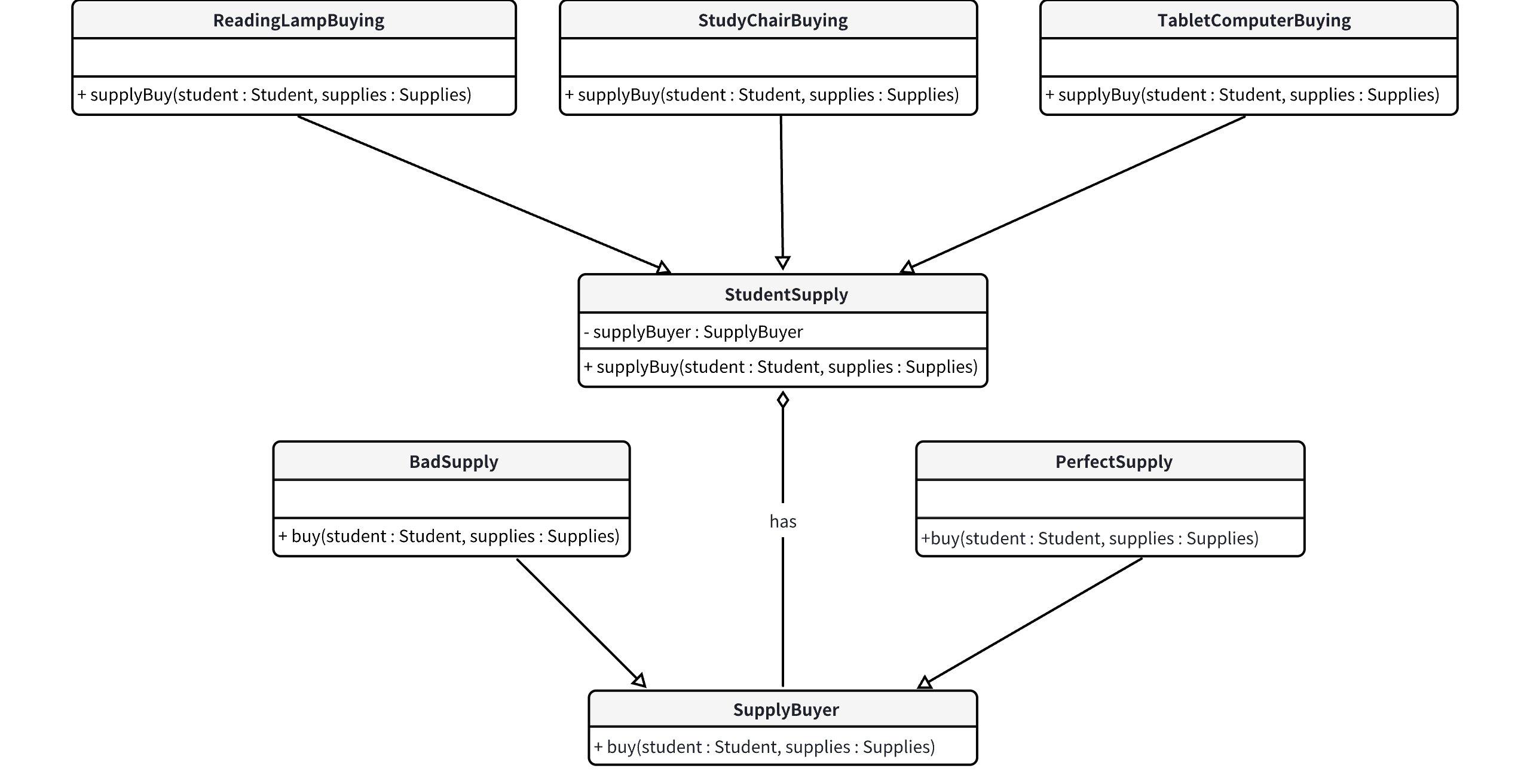
考虑到考研云系统涉及多种不同的学习工具，我们可以使用桥接模式来分离这些工具的抽象与实现，从而提高系统的灵活性和可扩展性。

在顶峰考研云中，桥接模式的实现涉及以下几个关键类：

1. SupplyBuyer 接口（SupplyBuyer.java）：这是桥接模式中的“实现（Implementation）”部分。它定义了购买行为的接口，其中 buy 方法用于执行购买操作。该接口有不同的实现，如 BadSupply 和 PerfectSupply，分别代表不同的购物体验。
2. 抽象类 StudentSupply（StudentSupply.java）：这是桥接模式中的“抽象（Abstraction）”部分。它包含了一个 SupplyBuyer 类型的引用，并定义了一个 supplyBuy 方法，该方法在学生购买物品时调用。该类允许其子类（如 ReadingLampBuying、StudyChairBuying、TabletComputerBuying）根据不同的需求扩展购买行为。
3. 具体的扩展类（ReadingLampBuying.java, StudyChairBuying.java, TabletComputerBuying.java）：这些类继承自 StudentSupply，代表具体的购买活动。它们通过调用 supplyBuy 方法利用 SupplyBuyer 的实现，并可添加额外功能，例如改变学生的属性。

在这个模式中，StudentSupply 类提供了一个桥梁，连接了购物行为的抽象表示（SupplyBuyer 接口）和具体的购物类型（如阅读灯购买、学习椅购买等）。这种设计允许在不影响现有类结构的情况下，灵活地添加新的购物类型或改变购物行为。通过桥接模式，可以在抽象层次和实现层次之间建立一个更灵活的关联，从而提高了代码的可维护性和扩展性。

### 3.3.2 类图



### 3.3.3 代价分析

1. 实现复杂度

* 增加：初期实现桥接模式可能会增加设计与开发的复杂度。需要更多的类和接口来分离抽象与实现，这可能会使得代码的结构变得更复杂。
* 减少：长期来看，一旦框架建立，新的功能添加和修改变得更加直观，因为它遵循了一个清晰定义的模式。

1. 可维护性

* 增加：由于分离了抽象与实现，修改或扩展一部分（例如，添加新的学习资源）不会影响到另一部分（如学习工具的实现）。这减少了代码修改时的不确定性和错误。
* 减少：初期，对于不熟悉桥接模式的开发团队来说，理解和维护这样的代码结构可能需要更多的时间和努力。

1. 性能

* 影响较小：在大多数情况下，使用桥接模式不会显著影响系统的性能。虽然增加了一层间接性，但这通常不会导致明显的性能下降。

1. 可扩展性

* 增加：桥接模式极大地增强了系统的可扩展性。添加新类型的学习资源或工具时，不需要修改现有类，只需扩展抽象类或实现新的接口即可。

## 3.4 建造者模式（GOF模式）

### 3.4.1 实现描述

建造者模式（Builder Pattern）是一种常用的软件设计模式，属于创建型模式。它的主要目的是将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。这种模式允许用户通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。

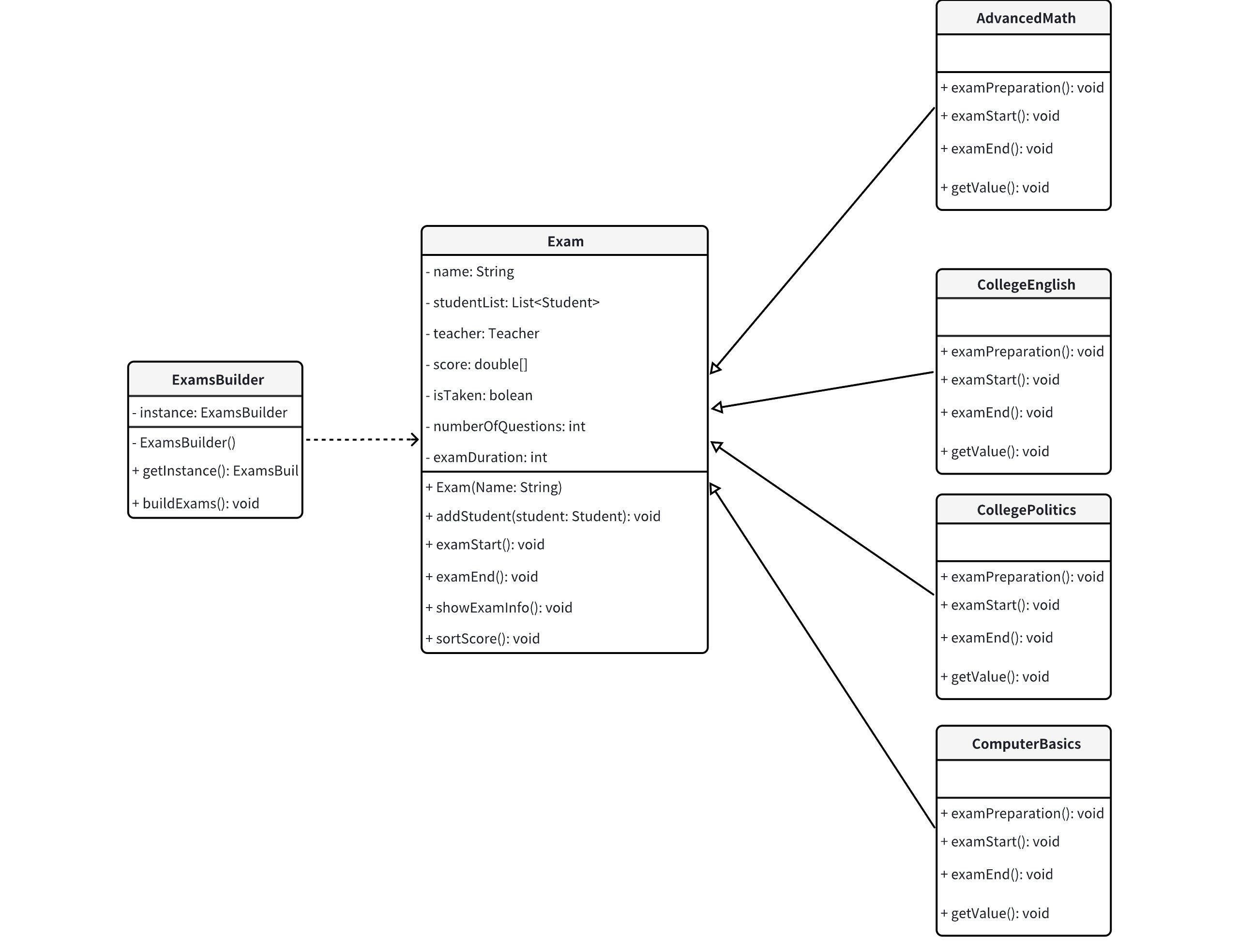
在“顶峰考研云”项目中，建造者模式被用于创建模拟考试的过程。这个系统设计了一个“Exam”类来表示一个模拟考试，其中包括了考试的各种属性，如考试的名称、参加考试的学生、出题教师、考试分数等。由于不同的考试类型（如高等数学、计算机408、大学英语、大学政治）拥有不同的特性和需求，使用建造者模式可以灵活地构建各种类型的考试，同时保持代码的清晰和可维护性。

具体来说，在“顶峰考研云”中，建造者模式的实现包括以下几个关键部分：

1. Exam类：这是一个抽象类，定义了模拟考试的基本框架和共同属性。它包括了考试名称、学生名单、出题教师、考试分数等属性，并提供了相应的方法来管理这些属性。此外，还包括了一些抽象方法，如examStart()、examEnd()和getValue(int no)，这些方法在具体的考试类型中将被实现。
2. 具体的考试类型类：例如AdvancedMath、CollegeEnglish、CollegePolitics和ComputerBasics等，这些类继承自Exam类，并实现了其中的抽象方法，以满足不同考试类型的特定需求。
3. ExamBuilder类：这个类使用单例模式设计，它负责构建具体的考试实例。通过buildExams()方法，它创建了不同类型的考试，并将它们添加到ExamContainer中。这个过程隔离了用户与考试对象创建过程的复杂性，用户只需要调用buildExams()方法就可以得到一系列预设的考试实例。

通过这种方式，建造者模式在“顶峰考研云”项目中实现了考试对象的灵活构建，使得代码更加清晰和易于维护，同时也为未来可能的扩展提供了便利。

### 3.4.2 类图



### 3.4.3 代价分析

**优点**

1. 分离构建与表示：  
   建造者模式提供了将对象的构建过程与其表示分离的能力，这样可以使客户端代码与对象的创建过程解耦，客户端只需要指定类型和内容即可。
2. 更好的可控性：  
   建造者模式允许逐步构建复杂对象，并允许过程逐步进行，这有助于产生更精细化的对象，并在构建过程中对其进行控制。
3. 易于引入变化：  
   由于建造者模式将产品的构建过程封装起来，因此更换一个不同的建造者就可以产生不同的产品，或者更改建造者的现有实现也易于操作。
4. 更好的复用性：  
   构建过程的相同部分可以在多个建造者之间复用，提高了代码的复用性。
5. 链式调用：  
   建造者模式常常与“链式调用”（fluent interface）结合使用，可以使代码的可读性和易用性得到提升。

**不足**

1. 复杂性增加：  
   如果原始对象比较简单，使用建造者模式可能会导致设计过于复杂，增加了系统的理解和实现难度。
2. 代码冗余：  
   需要编写多个建造者类和导演类，如果产品变化不大，可能会造成代码冗余。
3. 产品变化限制：  
   一旦建造者类被设计出来后，如果产品内部变化较大，可能会导致建造者类需要进行大量修改，影响已有的客户端代码。
4. 性能问题：  
   在某些情况下，如果对象的创建过程中无需按照步骤逐渐构建，或者构建过程不需要太多定制，则直接创建对象可能会更高效。
5. 设计复杂性：  
   对于设计人员来说，需要更深刻的理解需求和设计，以合理地拆分和封装构建过程，否则可能导致建造者模式的实现不够合理，不易于维护。

3.5 责任链模式（GOF模式）

### 3.6.1 实现描述

责任链模式（Chain of Responsibility Pattern）是一种行为设计模式，它允许将请求的发送者和接收者解耦，使多个对象都有机会处理这个请求。在这个模式中，请求从链中的一个对象传递到下一个对象，直到有一个对象处理它为止。

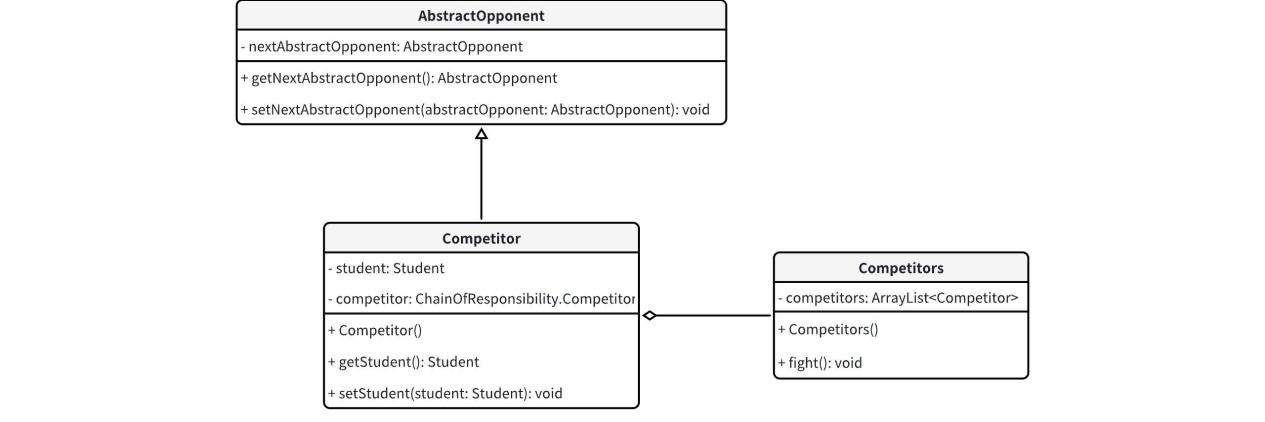
在“顶峰考研云”项目中，责任链模式被用于实现学生间的竞争对比逻辑。该模式包含以下三个关键部分：

1. AbstractOpponent.java（抽象竞争者类）:  
   这是一个抽象类，定义了责任链中的一个成员。它包含了一个指向链中下一个竞争者的引用。通过getNextAbstractOpponent()方法可以获取下一个竞争者，而setNextAbstractOpponent()方法则用于设置下一个要比拼的竞争者。这种设置形成了一个链，其中每个竞争者都指向另一个竞争者。
2. Competitor.java（竞争者类）:  
   这个类继承自AbstractOpponent，代表责任链中的一个具体节点。每个Competitor对象包含一个Student对象，表示参与竞争的学生。Competitor类通过继承AbstractOpponent获得了参与责任链的能力。
3. Competitors.java（竞争者集合类）:  
   这个类负责创建和管理Competitor对象的集合。在构造函数中，它初始化了一系列的Competitor对象，并将它们连接成一个责任链。fight()方法实现了竞争的逻辑，其中每个Competitor对象依次与链上的下一个竞争者进行比较，直到决出胜者。

在比赛过程中，每个Competitor对象都有机会与下一个竞争者进行对比。对比的结果决定了责任链上的下一个处理者（即下一个竞争者）。这种方式使得竞争的流程可以动态地在责任链中传递，直到找到最终的胜者。

通过责任链模式，竞争逻辑的实现变得灵活且易于扩展。每个Competitor对象只需关注与下一个竞争者的比较，而无需了解整个竞争流程的细节。这样不仅简化了竞争逻辑的实现，还提高了代码的可维护性和扩展性。

### 3.6.2 类图



### 3.6.3 代价分析

**优点**

1. 降低耦合度：  
   责任链模式使得发出的请求和处理请求的对象解耦，增强了系统的灵活性。请求的发送者不需要知道是哪个对象处理了请求，处理者也不需要知道请求的全貌。
2. 增强可扩展性：  
   可以在运行时动态地添加或修改处理链。这使得修改处理逻辑或增加新的处理类变得更加容易，有利于系统的扩展。
3. 增强给定功能的灵活性：  
   通过改变链中的成员或调整它们的顺序，可以动态地新增或改变处理过程。
4. 分散请求处理：  
   责任链模式允许多个对象处理同一个请求，可以分散请求处理的责任和压力。

**不足**

1. 性能问题：  
   在责任链模式中，请求可能需要遍历多个处理者才能被处理，这可能会导致一定的性能损耗。
2. 调试困难：  
   由于处理过程分散在链的各个节点，当链较长时，调试和追踪错误可能会比较困难。
3. 链的构建问题：  
   责任链的构建需要谨慎，错误的构建可能会导致链中某些处理者永远无法达到或请求无法得到正确处理。
4. 过度使用：  
   过度使用责任链模式可能会导致系统中出现许多小对象，增加系统的复杂性。
5. 不保证被处理：  
   在责任链模式中，一个请求可能最终不被任何处理者处理，这需要在设计时特别注意。

## 3.6 命令模式（GOF模式）

### 3.6.1实现描述

**命令模式(Command Pattern)**是一种**行为型设计模式**，它允许将请求封装成一个独立的对象，并将请求的发送者和接收者解耦，使得请求可以被**保存、传递、记录、撤销或重做**，从而提供更大的灵活性和可扩展性。在命令模式中，通常会定义一个命令接口或抽象类，其中包含了执行命令所需的方法。具体的命令类则实现了命令接口或抽象类，并包含了实际的命令代码。命令类还可以包含一些额外的信息，如命令的参数、接收者等。命令模式通过引入命令对象，将请求者和接收者解耦。请求者只需要知道调用命令对象的execute()方法即可，而不需要知道命令是如何被执行的以及命令所涉及的接收者是谁。接收者则只需要实现命令接口，并负责实际执行命令所涉及的操作，而不需要知道该命令是如何被调用和如何被使用的。

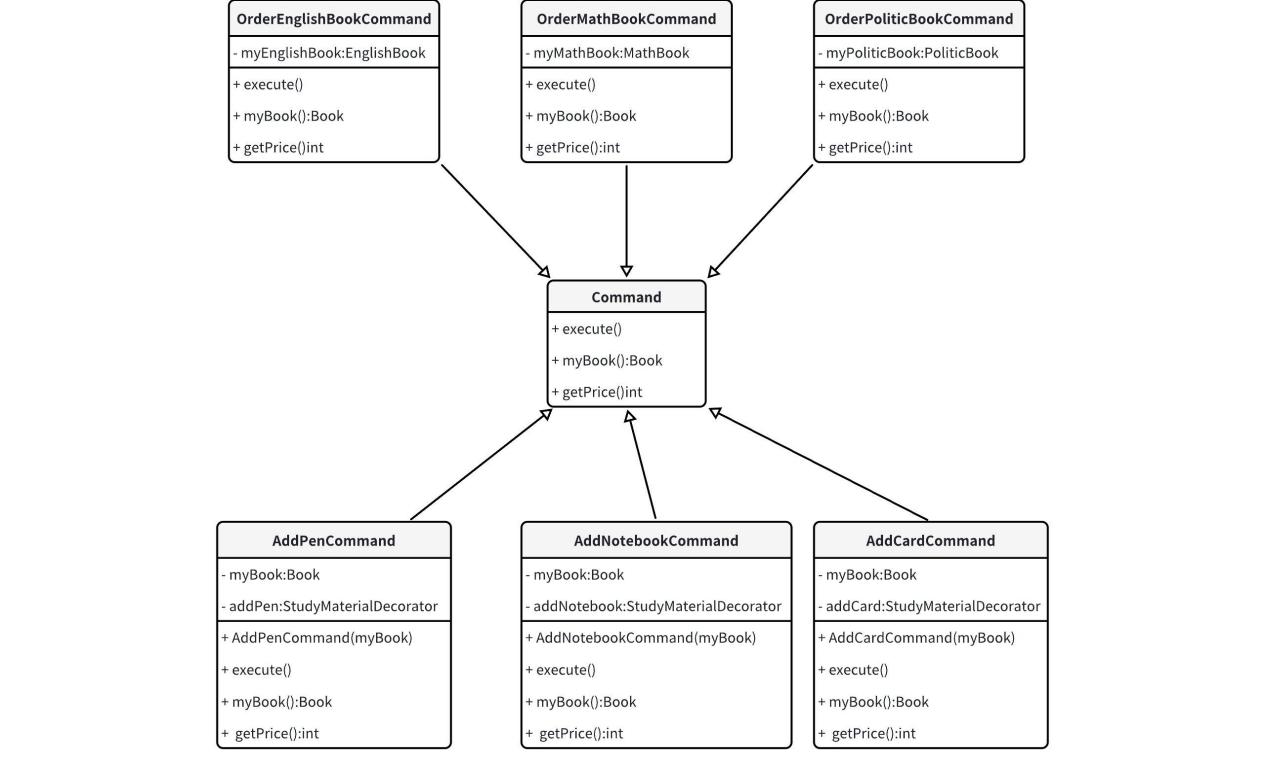
在顶峰考研云中，注册用户可以进行教辅资料的采购，在顶峰资料采购中心中，用户可以在商城购买“考研数学武忠祥”、“考研英语红宝书”、“考研政治肖秀荣”等书籍用于复习和刷题，用户在购买书籍后可以顺便购买一些学习辅助材料进行学习刷题的辅助，可提供的学习辅助材料包括：“水笔”，“错题本”，“答题卡”等。在此场景下，由于用户在选择了自己购买的书籍之后可以无限添加所需要的学习辅助材料，所以“水笔”，“错题本”，“答题卡”的添加没有次数的限制，同时，用户可以对自己的点单进行撤销操作，所以为了能够方便对于同一条命令的多次调用以及对于调用命令的撤销，我们选择使用命令模式进行该功能的实现，这将有利于选择调用和实际操作的解耦合，增加采购方案选择的灵活性。

在顶峰考研云系统中，我们有以下几个关键类：

* **OrderEnglishBookCommand：**用于处理“采购考研英语红宝书”的命令，其中execute函数用于处理与购买相关的操作，myBook函数用于获得该书的名称，getPrice函数用于获得考研英语红宝书的价格信息。
* **OrderMathBookCommand：**用于处理“采购高等数学武忠祥”的命令，其中execute函数用于处理与购买相关的操作，myBook函数用于获得该书的名称，getPrice函数用于获得高等数学武忠祥的价格信息。
* **OrderPoliticBookCommand：**用于处理“采购考研政治肖秀荣”的命令，其中execute函数用于处理与购买相关的操作，myBook函数用于获得该书的名称，getPrice函数用于获得考研政治肖秀荣的价格信息。
* **AddCardCommand：**用于在用户选择了需要购买的书籍之后购买配套的答题卡。
* **AddPenCommand：**用于在用户选择了需要购买的书籍之后购买配套的水笔。
* **AddNotebookCommand：**用于在用户选择了需要购买的书籍之后购买配套的错题本。

1. **Command**： 定义了命令的接口，包含了对象之间交互的方法。具体的实现了这个接口，并负责协调和管理对象之间的通信。

### 3.6.2类图



### 3.6.3 代价分析

**命令模式的主要优点包括：**

1. 降低系统的耦合度，将请求者和接收者解耦，使得两者可以独立变化。
2. 可以容易地扩展新的命令类，而无需修改原有的代码。
3. 可以方便地实现撤销和重做操作，因为所有的命令都被封装成了对象。
4. 可以将命令作为参数进行传递、保存和调用。

**命令模式的主要缺点包括：**

1. 命令的数量可能会增加系统的复杂度。
2. 命令模式可能引入过多的具体命令类，从而增加系统的开销。
3. 在需要支持多级撤销、重做等操作时，可能会增加额外的复杂度。

## 3.7 组合模式（GOF模式）

### 3.7.1 实现描述

组合模式（Composite Pattern），也被称为部分-整体模式，是一种结构型设计模式，用于将一组相似的对象视为一个单一的对象。该模式基于树形结构来组织对象，以表示对象的部分及整体层次关系。这种模式不仅彰显了结构型模式的特点，而且通过创建对象的树形结构，有效地管理了对象组。

核心思想在于，组合模式使得客户端能够统一对待单个对象和组合对象。这种模式下，我们构建了一个包含自身对象组的类，这个类提供了一种方式来统一修改和管理这一组相似的对象。在实际应用中，这意味着无论是单个对象还是对象的组合，都可以通过同一接口进行操作，大大增强了代码的灵活性和可扩展性。

主要囊括的类如下：

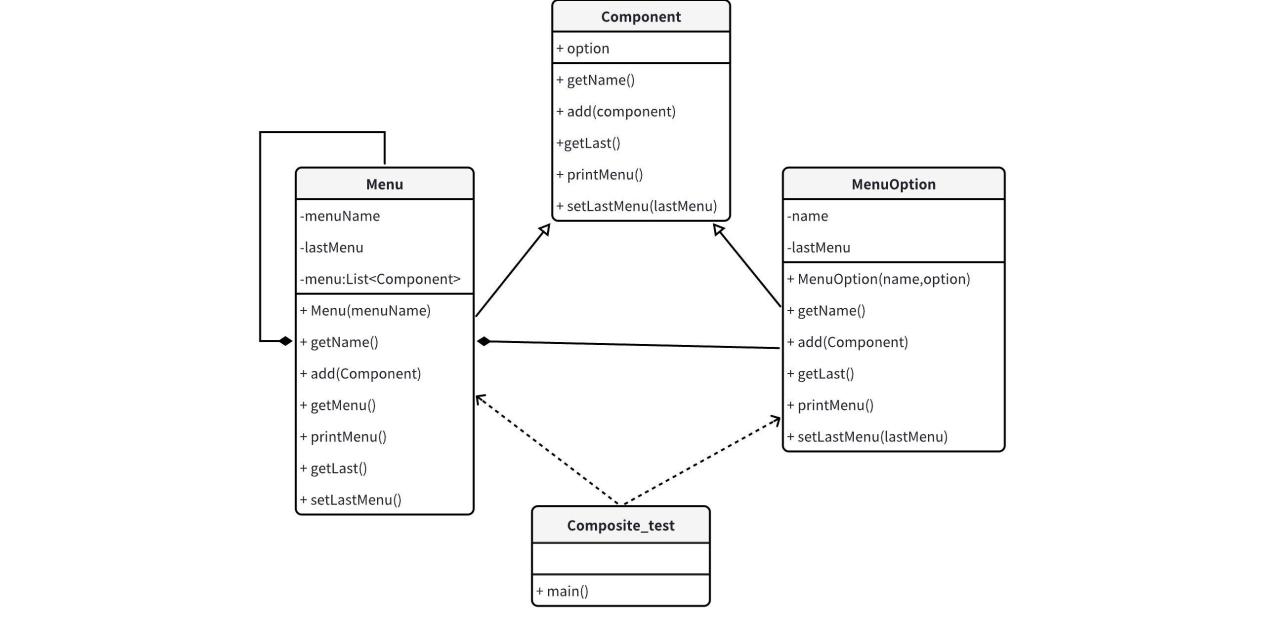
1. Component (Component.java): 这是一个抽象类，定义了菜单和菜单选项的共同行为。它包含了如 getName、add、printMenu 等抽象方法，以及用于设置和获取上一级菜单的方法。
2. Menu (Menu.java): 继承自 Component 类，代表菜单。它包含一个 Component 类型的列表，用于存储子菜单项或子操作项。Menu 类重写了 add 和 printMenu 方法，用于添加子菜单项和打印菜单。
3. MenuOption (MenuOption.java): 也是 Component 类的子类，代表单个菜单选项。它实现了 getName 方法，并可能包含与选项相关的特定操作或数据。

在项目中的 CompositeTest 类里，我们采用了组合模式来模拟文件与文件夹的关系，通过设计 MenuOption 和 Menu 两种类，这两者都继承自 Component 类，来分别表示菜单中的单个选项和二级菜单。这种设计思路源于 Menu 和 MenuOption 在本质上的高度相似性：它们都是用于存储菜单选项的类，并且它们的结合非常适合于简化整体结构，减少复杂性。

选择组合模式进行设计，我们充分利用了这种模式中“将相似的对象组合在一起”的核心思想。这不仅大大减少了开发和维护的工作量，而且对外表现为一个统一的 Menu 对象，有效避免了在对外展示时可能出现的混乱和无序。组合模式使得对同一组对象的管理和控制变得更加高效和有序。

在 CompositeTest 中，我们通过 add 等方法向现有的 Menu 对象中添加新的菜单选项。这些菜单选项可以是单个的 MenuOption 或者是包含更多选项的子 Menu。这种方式不仅使得菜单的构建过程变得灵活，而且也便于在适当的时机进行菜单的打印和跳转操作。通过这种设计，我们实现了一个既清晰又易于扩展的菜单系统，有效地展示了组合模式在实际应用中的强大功能。

### 3.7.2 类图



### 3.7.3 代价分析

组合模式优势如下：

1. 统一的对象处理：组合模式使得客户端可以统一对待单个对象和组合对象，简化了客户端代码。
2. 清晰的层次结构：该模式清晰地定义了对象之间的层次关系，有助于更好地理解和管理对象结构。
3. 增强的灵活性：在组合模式中，可以轻松地添加新的组件，无需修改现有代码，这提高了代码的可扩展性。
4. 简化的设计：对于需要构建复杂对象树的系统，组合模式提供了一种简化的解决方案。

组合模式不足如下：

1. 设计的复杂性：实现组合模式可能会增加设计的复杂性，特别是在构建复杂的组合结构时。
2. 难以限制组件：在组合模式中，很难对组件的类型进行限制。例如，某些组件可能不应该有子组件，但在模式中强制实现相关方法。
3. 性能考虑：在某些情况下，组合模式可能会引入过多的对象和层次，可能对性能产生负面影响。
4. 过度一般化：组合模式可能会导致系统设计过度一般化，使得系统难以理解和维护。

## 3.8 数据访问对象模式（架构模式）

### 3.8.1 实现描述

DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。

在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。

用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。

DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。有如下优点：

1、隔离了数据访问代码和业务逻辑代码。业务逻辑代码直接调用DAO方法即可，完全感觉不到数据库表的存在。分工明确，数据访问层代码变化不影响业务逻辑代码,这符合单一职能原则，降低了藕合性，提高了可复用性。

2、隔离了不同数据库实现。采用面向接口编程，如果底层数据库变化，如由 MySQL 变成 Oracle 只要增加 DAO 接口的新实现类即可，原有 MySQ 实现不用修改。这符合 "开-闭" 原则。该原则降低了代码的藕合性，提高了代码扩展性和系统的可移植性

主要有三个函数构成：

1.public List<Student> getAllStudent();本函数定义于StudentDao类中，在StudentDaoImpl类中进行实现

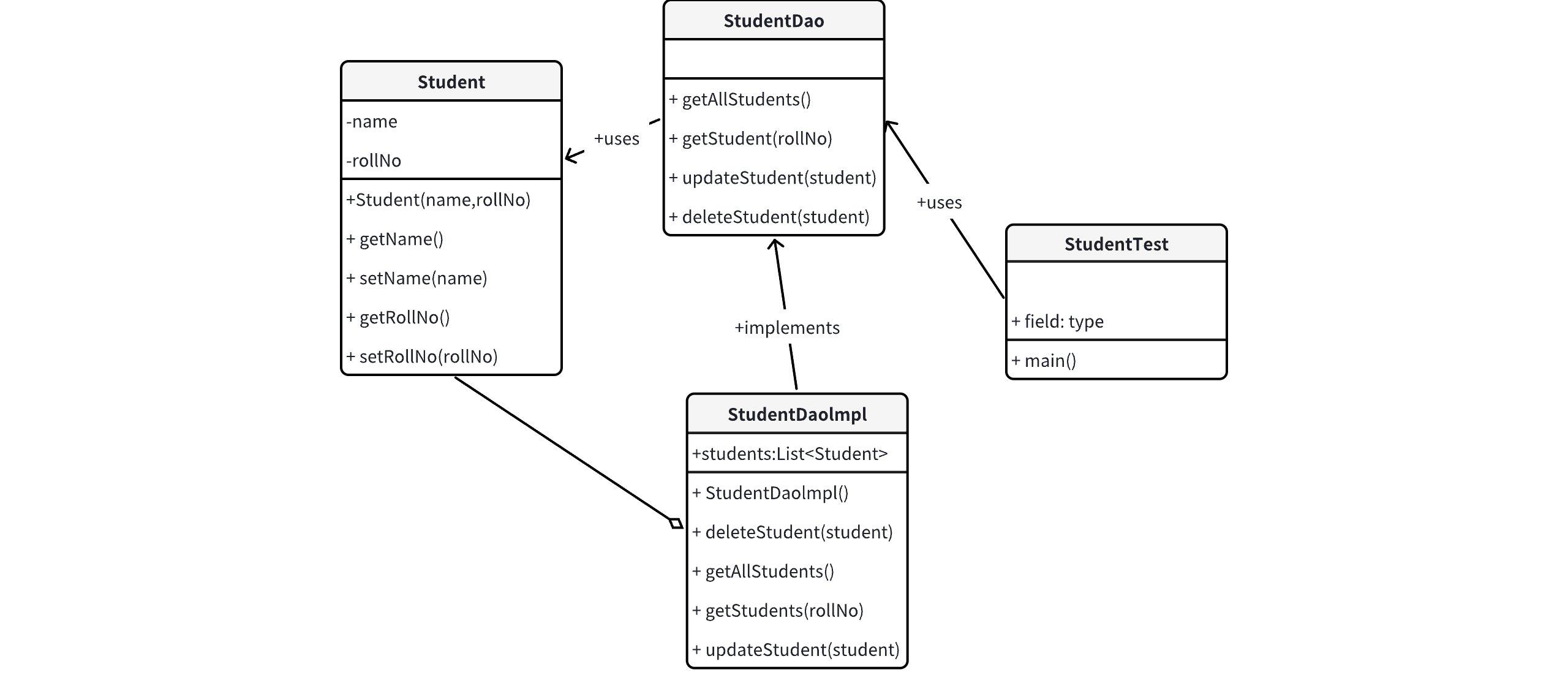
2.public void updateStudent();本函数定义于StudentDao类中，在StudentDaoImpl类中进行实现

3.public void deleteStudent();如上说明，本函数定义于StudentDao类中，在StudentDaoImpl类中进行实现

在该API中将学生编号以及对于学生编号的操作抽象化为数据访问，如此一来，用户就可以进行编辑学生编号的操作。

在该API中，可以进行更新和删除学生编号的操作。根据项目的不同可以编辑学生编号。从用户角度来看，我们为考试制作了可编辑的学生名单，符合真实的考试场景，有利于用户体验。

### 3.8.2 类图



### 3.8.3 代价分析

**优点：**

1. **关注点分离：** DAO模式将数据访问逻辑与业务逻辑分离，促进了更清晰和可维护的体系结构。
2. **集中化数据访问：** 提供了一个集中的位置来管理数据访问代码，更容易更新和维护。
3. **可测试性：** 数据访问逻辑的分离使得更容易独立测试数据访问和业务逻辑。

**缺点：**

1. **增加复杂性：** 引入DAO可能会增加系统的整体复杂性，尤其是对于较小的应用程序。
2. **潜在的重复：** 如果管理不当，可能存在在不同DAO之间重复数据访问逻辑的风险。

### 3.8.4 出处

出自论文集：[7]欧阳宏基,解争龙,黄素萍等.一种基于DAO设计模式与Hibernate框架的数据持久化层模型[J].微计算机应用,2009,30(03):36-40.

## 3.9 装饰器模式（GOF模式）

### 3.9.1 实现描述

装饰器模式是一种**结构型设计模式**，它允许向现有对象动态地添加新功能，同时不改变其结构。通过这种方式，可以在不修改现有对象的情况下，对其进行功能扩展和修饰。

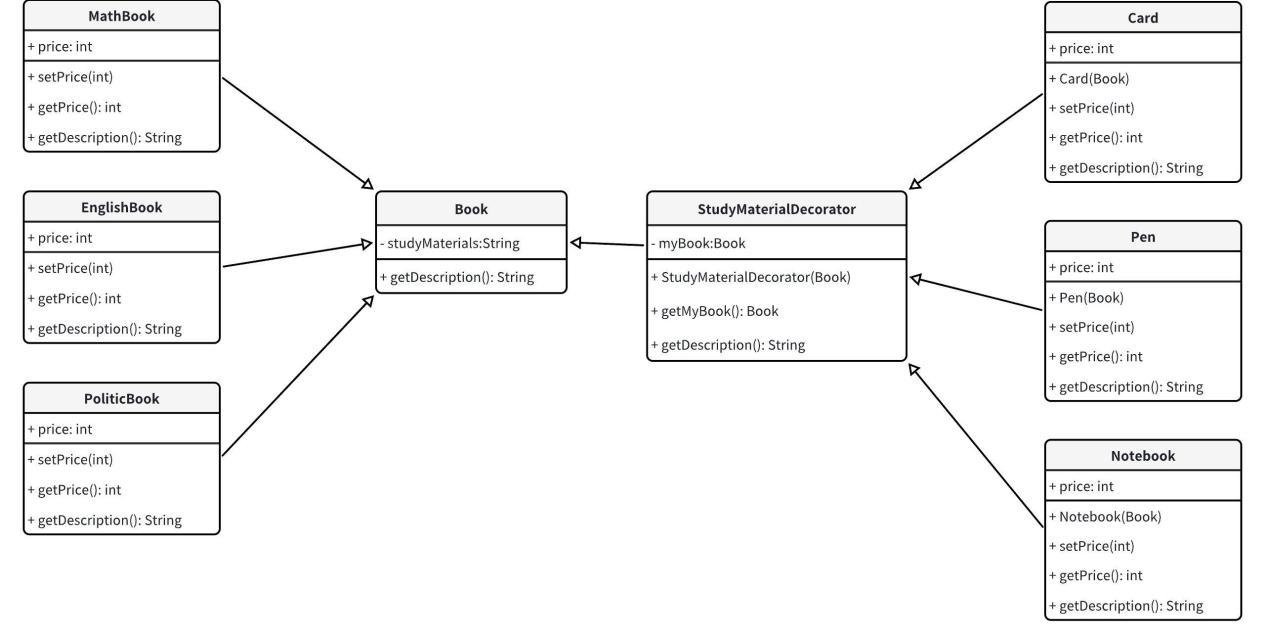
在装饰器模式中，通常包含以下几个角色：

1. **Component（组件）**：定义一个对象接口，可以给这些对象动态地添加新的职责。
2. **ConcreteComponent（具体组件）**：实现了Component接口，并定义了需要被装饰的具体对象。
3. **Decorator（装饰器抽象类）**：继承自Component，并持有一个Component对象的引用。其目的是为了扩展Component对象的功能，同时保持接口的一致性。
4. **ConcreteDecorator（具体装饰器）**：实现了Decorator接口，负责给具体组件对象添加具体的装饰功能。

装饰器模式的核心思想是通过递归组合来实现功能的动态拓展。具体地说，当需要为一个对象添加新的功能时，可以创建一个具体的装饰器类并将原始对象作为参数传递进去，然后在具体装饰器类中扩展功能并调用原始对象的方法。

在我们的顶峰考研云系统中，装饰器模式应用在顶峰考研云教辅资料购买商店中，教辅资料售卖商店中卖教辅书籍的方式是教辅书籍和配套文具一起进行捆绑售卖，即用户可以在购买了教辅书籍之后选择需要的配套书籍一起买，所以可以视作配套文具是在装饰对应的书籍。

### 3.9.2 类图



### 3.9.3 代价分析

装饰器模式的优点包括：

1. **灵活性**：可以动态地给对象添加新的功能，而无需修改其原始代码。
2. **遵循开闭原则**：可以通过添加新的装饰器类来扩展对象的功能，而无需修改现有的代码。
3. **保持接口一致性**：装饰器类与原始对象都实现了相同的接口，使得客户端无需知道具体的装饰器类，而只需使用原始对象即可。

然而，装饰器模式也存在一些缺点:可能产生大量的小对象，增加系统的复杂性，以及可能引起性能上的损失。

因此，在使用装饰器模式时，需要根据具体的场景权衡利弊，并确保设计合理、清晰。

## 3.10 外观模式（GOF模式）

### 3.10.1 实现描述

外观模式（Facade Pattern）是一种常用的软件设计模式，旨在为复杂的系统提供一个统一、简化的接口。通过这种方式，外部代码可以通过这个“外观”接口与系统交互，而无需直接处理复杂的内部逻辑。这种模式特别适用于具有复杂子系统的大型应用程序，可以减少系统间的依赖，提高模块的独立性和可用性。

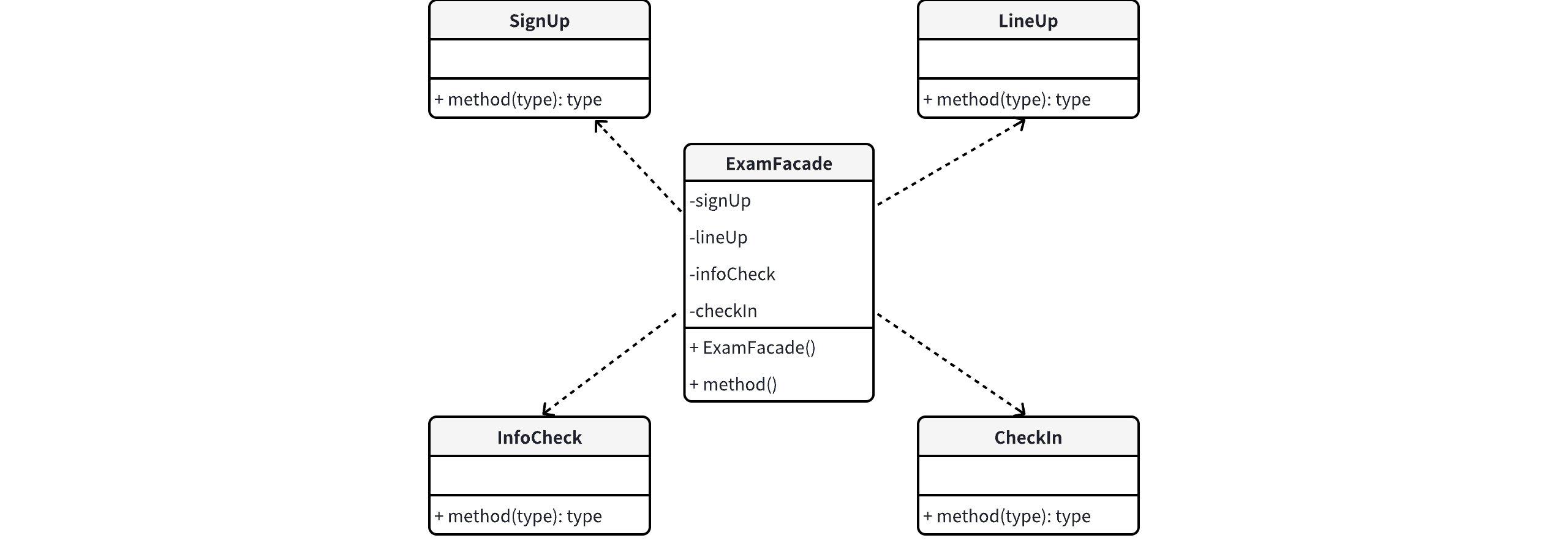
在顶峰考研云的系统中，当学生选择参加考试后，他们需要完成报名、排队、信息检查和入场等四个步骤，方可正式开始考试。这四个步骤虽然各自独立，但都是考试准备过程的一部分，且最终目的一致。因此，我们可以将这些步骤抽象化，并设计一个统一的高层接口，以简化对学生而言可能较为复杂的报名流程。在这个接口中，某些对学生来说不太重要或不必详细了解的步骤，如排队和信息检查，可以对学生隐藏。这也就提供了外观模式的理想使用场景。具体实现方式如下：

1. **SignUp (SignUp.java)**: 负责处理学生报名的过程。
2. **LineUp (LineUp.java):** 负责管理学生排队的过程。
3. **InfoCheck (InfoCheck.java)**: 负责进行学生信息的检查。
4. **CheckIn (CheckIn.java)**: 负责学生入场的流程。
5. **ExamFacade (ExamFacade.java):** 这是外观类，它封装了上述所有过程。在 ExamFacade 类中，创建了 SignUp、LineUp、InfoCheck 和 CheckIn 的实例，并提供了一个 method 方法来统一管理这些过程。当调用 method 方法时，它依次调用 SignUp、LineUp、InfoCheck 和 CheckIn 的 doSomething 方法，从而简化了客户端与这些子系统的交互。

这个接口为四个子系统中的一组用户接口提供了一个统一的界面，外观模式定义了一个更高层次的接口，使得整个子系统的使用变得更加简单。这降低了学生访问这个复杂系统内部子系统的难度，简化了学生与系统的交互。我们的学生用户不需要了解系统内部报名、排队、信息检查、入场等步骤的复杂关系，因此我们提供了一个外观（Facade）接口，定义了系统的入口。这样一来，学生在进入系统准备开始考试时，可以迅速高效地完成准备工作，避免因子系统问题而造成的延误，更符合实际应用需求，便于管理，提升用户体验。

外观模式是迪米特法则和依赖倒置原则的典型应用，但它并不完全符合开放封闭原则。因为用户无法了解子系统内部的行为模式，如果出现问题可能会比较棘手。同时，由于用户不知道且无法修改内部行为，这可能使得系统的扩展或重写变得困难。但是，外观模式极大地优化了用户的功能体验，通过整合报名、排队、信息检查、入场等过程，并创建一个更高层的接口，用户可以直接调用这个接口来完成复杂的准备流程，使得整个过程更加方便快捷。

### 3.10.2 类图



### 3.10.3 代价分析

外观模式的优势方面如下：

1. **简化接口：**外观模式为复杂的子系统提供了一个简单的接口，使得客户端在使用时不需要直接与复杂的内部子系统交互，降低了使用复杂度。
2. **降低耦合度：**通过引入外观类，可以降低系统的耦合度，因为客户端只需要与外观类交互，而不是直接与复杂的子系统打交道。
3. **提高可用性：**对于那些需要频繁使用某个复杂子系统的客户端来说，外观模式提供了一种快捷方便的方式来访问该子系统，提高了系统的整体可用性。
4. **易于维护和扩展：**由于外观类封装了子系统的交互，因此在子系统内部发生变化时，只需修改外观类即可，而不需要修改客户端代码。

外观模式的不足之处如下：

1. **不符合开放封闭原则：**外观模式可能不完全遵循开放封闭原则。当子系统发展变化时，可能需要修改外观类，这违反了开放封闭原则中“对扩展开放，对修改封闭”的理念。
2. **隐藏系统复杂性：**虽然隐藏复杂性对用户友好，但同时也可能导致开发者忽视系统内部的复杂性，从而在系统设计和调试时遇到难题。
3. **过度使用可能导致系统变得难以理解：**如果过度依赖外观模式，可能会使得系统的架构变得难以理解，特别是在涉及多个外观类的情况下。
4. **性能考虑：**虽然通常不显著，但外观类的额外抽象层可能会引入一些性能开销。

## 3.11 工厂方法模式（GOF模式）

### 3.11.1 实现描述

工厂方法模式（Factory Method Pattern）是一种创建型设计模式，其核心在于定义一个用于创建对象的接口，但让子类决定要实例化的类是哪一个。工厂方法模式使一个类的实例化延迟到其子类。

在“顶峰考研云”系统中，工厂方法模式的实现可以针对不同类型的学习资源或工具进行优化。该系统可能包含多种类型的学习辅助工具和资源，如阅读灯、学习椅、电脑等，每种都有其特定的属性和功能。工厂方法模式在这个场景下的应用，不仅有助于创建这些复杂的对象，还提供了一个清晰、灵活的架构来处理不断变化和增加的产品类型。

在顶峰考研云中，工厂方法模式的实现主要涉及以下几个关键类和接口：

1. **接口** ISupplyFactory**（ISupplyFactory.java）**

这是工厂方法模式的核心，它定义了一个创建对象的接口，这里是 createSupply 方法。该方法接受一个 Student 对象作为参数，并返回 Supplies 类的一个实例。

1. **具体工厂类**

* ReadingLampFactory（ReadingLampFactory.java）：实现 ISupplyFactory 接口，重写 createSupply 方法来创建 ReadingLamp 的实例。
* StudyChairFactory（StudyChairFactory.java）：同样实现 ISupplyFactory 接口，重写 createSupply 方法来创建 StudyChair 的实例。
* TabletComputerFactory（TabletComputerFactory.java）：实现 ISupplyFactory 接口，重写 createSupply 方法来创建 TabletComputer 的实例。

1. **抽象产品类** Supplies**（Supplies.java）**

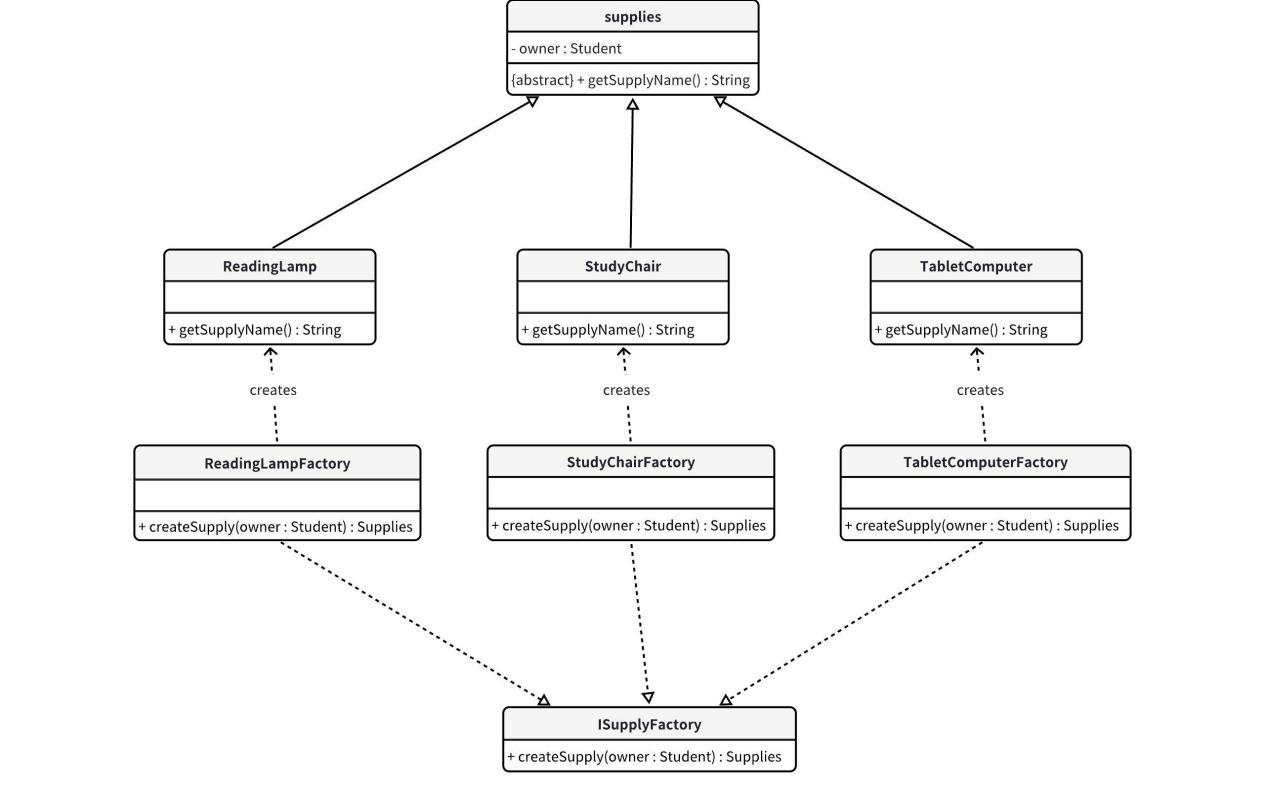
这是一个抽象类，定义了产品的共有属性和方法。在这个例子中，它包含了一个 Student 对象，并定义了一个抽象方法 getSupplyName，由子类实现。

1. **具体产品类**

* ReadingLamp（ReadingLamp.java）：Supplies 的子类，实现 getSupplyName 方法，返回 "阅读灯"。
* StudyChair（StudyChair.java）：同样是 Supplies 的子类，实现 getSupplyName 方法，返回 "学习椅"。
* TabletComputer（TabletComputer.java）：Supplies 的另一个子类，实现 getSupplyName 方法，返回 "性能++的电脑"。

在此实现中，ISupplyFactory 接口定义了一个创建 Supplies 对象的标准，而具体的创建工作由实现了 ISupplyFactory 的具体工厂类来完成。每个具体工厂类负责创建一种特定类型的 Supplies。这样，当需要创建新的 Supplies 类型时，只需添加一个新的具体工厂类，而无需修改现有代码。这提高了代码的可维护性和扩展性，同时遵循了开闭原则（对扩展开放，对修改封闭）。

### 3.11.2 类图



### 3.11.3 代价分析

1. **实现复杂度**

* 增加：引入工厂方法模式意味着需要编写额外的代码来定义接口和实现具体的工厂类。这增加了初期开发的复杂度。
* 减少：随着系统发展，添加新类型的供应品变得更简单，因为只需添加一个新的具体工厂类而不需要修改现有代码。

1. **可维护性**

* 增加：通过将对象创建逻辑封装在各自的工厂类中，简化了维护工作，因为更改特定类型的创建逻辑不会影响其他部分。
* 减少：对于新加入的开发人员来说，理解多个工厂类和它们之间的关系可能需要一定的学习曲线。

1. **可扩展性**

* 增加：工厂方法模式提高了系统的可扩展性。在需要新增供应品类型时，可以轻松扩展而无需修改现有的工厂结构。

1. **性能影响**

* 影响较小：工厂方法模式对性能的影响通常很小，除非在非常密集的对象创建场景中，额外的函数调用可能会产生轻微的性能开销。

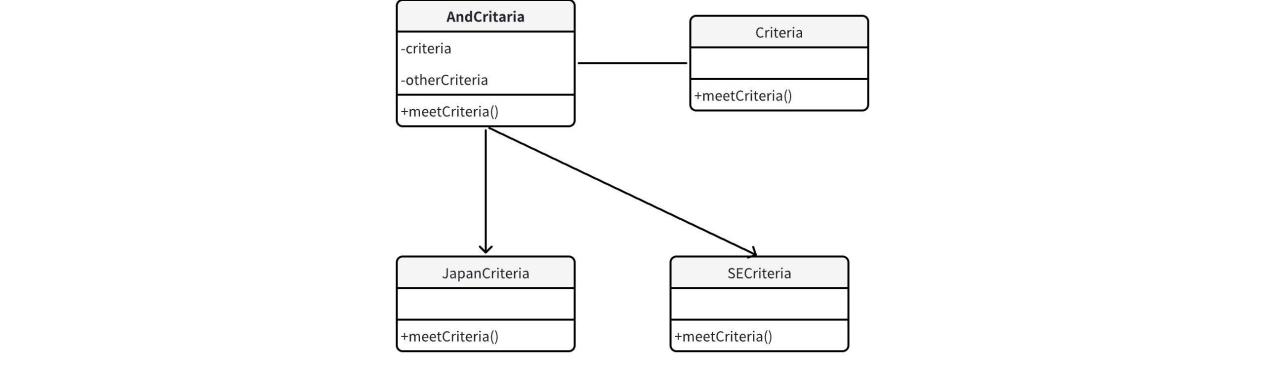
## 3.12 过滤器模式（其他模式）

### 3.12.1 实现描述

过滤器模式（Filter Pattern）是一种结构型设计模式，它允许通过一系列条件来过滤复杂对象，从而实现对对象的筛选和过滤。主要包含以下几个关键角色：

1. **Filter（过滤器接口）：** 定义了过滤器的接口，通常包含一个用于过滤的方法。
2. **ConcreteFilter（具体过滤器类）：** 实现了过滤器接口，具体定义了过滤的条件和规则。
3. **Criteria（标准接口）：** 定义了过滤的标准，通常包含一些用于检查对象是否符合标准的方法。
4. **ConcreteCriteria（具体标准类）：** 实现了标准接口，具体定义了过滤的标准，用于检查对象是否满足某些条件。
5. **FilterChain（过滤器链）：** 可选，用于组合多个过滤器，形成一个过滤器链，依次对对象进行过滤。
6. **Client（客户端）：** 使用过滤器模式的客户端，通过过滤器来筛选和过滤对象。

### 3.12.2 类图



### 3.12.3 代价分析

1. **维护性方面：** 过滤器模式通过将过滤逻辑封装到不同的过滤器类中，使得新增或修改过滤条件更加方便。当需要添加新的过滤条件时，只需创建新的过滤器类，而不需要修改已有的代码，降低了维护成本。
2. **拓展性方面：** 过滤器模式支持良好的拓展性。可以轻松地新增过滤器类来添加新的过滤条件，也可以通过组合不同的过滤器来创建不同的过滤策略。这有助于系统的灵活性和可拓展性。
3. **复杂性方面：** 过滤器模式的实现相对简单，易于理解。然而，当涉及到多个过滤条件的组合时，可能需要合理设计过滤器链，以确保过滤器的顺序和组合逻辑正确。
4. **性能方面：** 过滤器模式可能会引入一些性能开销，特别是当存在多个过滤器并且需要依次进行过滤时。在处理大量数据的情况下，可能需要考虑性能影响。

总体而言，过滤器模式是一种特别适用于需要灵活组合多个过滤条件的场景。在实际使用中，需要根据具体的需求和系统的性能要求权衡使用过滤器模式的利弊。

### 3.12.4 出处

出自专(译)著:[6]Deepak Alur,John Crupi,Dan Malks.J2EE核心模式(,刘天北,熊杰).机械工业出版社,2003.98-113

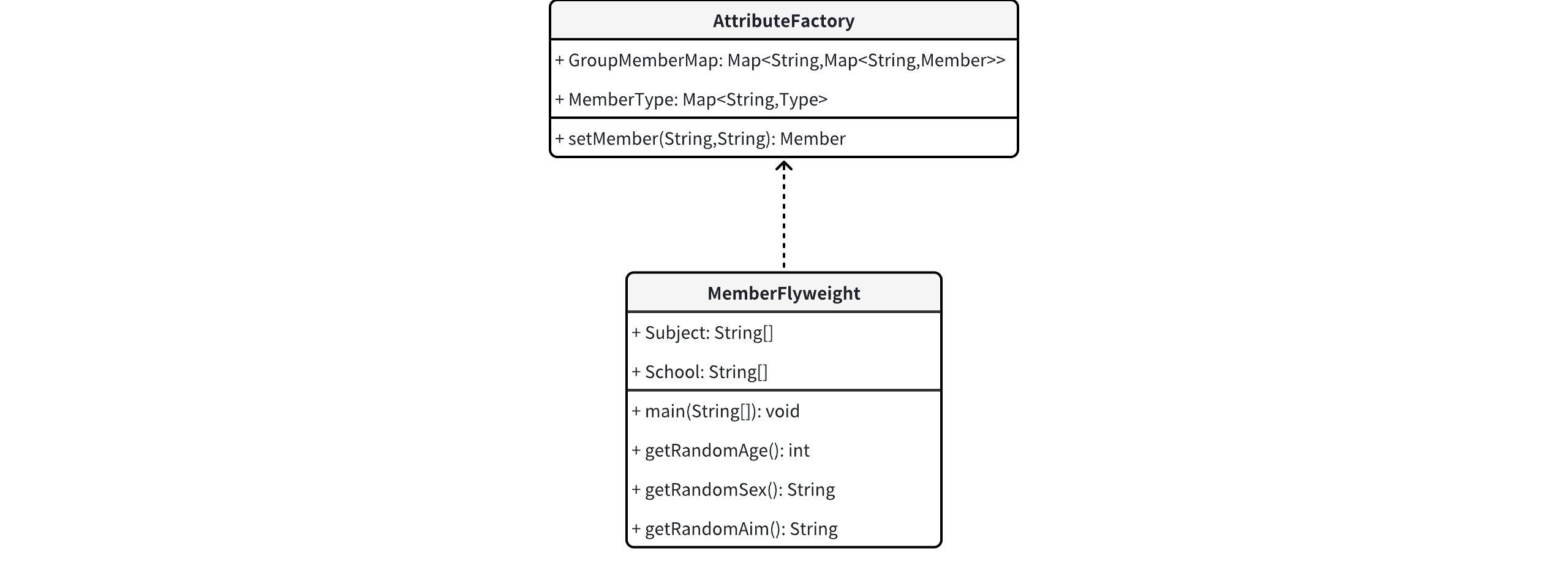
## 3.13 享元模式（GOF模式）

### 3.13.1 实现描述

享元模式（Flyweight Pattern）是一种结构型设计模式，它通过共享多个对象所共有的相同状态，使在有限的内存空间中载入更多的对象，适用于系统中存在大量相似对象的情况。

在“顶峰考研云”项目中，享元模式用于管理学生加入学习小组的行为，它提供统一的接口用于将学生加入学习小组，其中AttributeFactory类充当享元工厂，负责创建和管理共享的学生对象，MemberFlyweight类作为客户端，演示了如何使用享元模式创建和管理学生，同时统计学生数量和详情。

### 3.13.2 类图



### 3.13.3 代价分析

* 优点

1. 内存优化：通过共享相似对象实例，享元模式减少了内存占用，适用于存储大量细粒度对象的场景
2. 性能提升：共享对象实例减少了对象的创建和销毁次数，提高系统性能

* 缺点

1. 共享状态需谨慎对待：由于多个对象共享同一实例，需要谨慎修改操作，以防对一个对象的修改影响到其他对象
2. 工厂较复杂：管理享元对象的工厂比较复杂，需要考虑对象的创建、共享、销毁等细节。

## 3.14 解释器模式（GOF模式）

### 3.14.1 实现描述

**解释器（Interpreter）模式的定义：**给分析对象定义一个语言，并定义该语言的文法表示，再设计一个解析器来解释语言中的句子。也就是说，用编译语言的方式来分析应用中的实例。这种模式实现了文法表达式处理的接口，该接口解释一个特定的上下文。这里提到的文法和句子的概念同编译原理中的描述相同，“文法”指语言的语法规则，而“句子”是语言集中的元素。在软件开发中，会遇到有些问题多次重复出现，而且有一定的相似性和规律性。如果将它们归纳成一种简单的语言，那么这些问题实例将是该语言的一些句子，这样就可以用“编译原理”中的解释器模式来实现了。但对于满足以上特点，且对运行效率要求不是很高的应用实例，如果用解释器模式来实现，其效果是非常好的。

**解释器模式包含以下几个角色：**

1. 抽象表达式（Abstract Expression）：定义了解释器的接口，声明了解释方法 interpret()，所有的具体表达式都需要实现这个接口。
2. 终结符表达式（Terminal Expression）：终结符表达式代表语言中的最小的单位，通常是非可分割的单个字符或单词。终结符表达式没有子表达式，它负责实际的解释操作。
3. 非终结符表达式（Non-terminal Expression）：非终结符表达式由一个或多个终结符表达式组成，它负责解释语法规则中的非终结符部分，可能包含其他的非终结符表达式或终结符表达式。
4. 上下文（Context）：上下文对象包含解释器之外的全局信息，它提供了解释器执行所需的数据或其他全局环境。

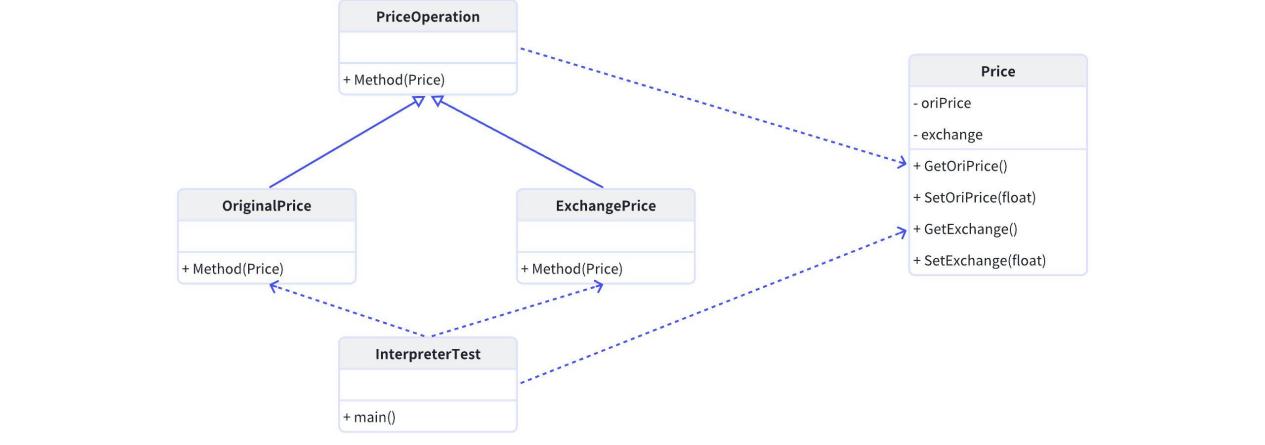
对于像价格这样重复出现但又在不同场景有不同变化的对象，我们可以通过设计Price这样一个存储了价格和汇率的类，以及几个对 Price进行不同解析方式的解释器来输出对应价格。

在合适的场合设置 Price的汇率，利用对应场合的价格解释器，将 Price对象输入到解释器中即可输出该场合所需的最终价格。由解释器来帮我们进行这样重复但又有不同输入对象的操作。该模式应用在选手购买外设场景下，选手在付款时应用虚拟货币与实际流通货币不是 1：1 等值的，而是存在一定的货币汇率，在此设定为 1：10，若一款阅读灯 10 元，则选手实际需要支付的代币为 1000，而这一转换过程需要 Interpreter 来实现，该模式解读了当前货币的汇率并将之直接转换为选手需要支付的虚拟货币，从而便于用户付款购买学习装备。

**解释器模式的工作流程如下：**

1. 客户端创建具体的表达式对象，并将其组合成一个抽象的语法树结构。
2. 客户端准备上下文对象，并将其传递给解释器。
3. 解释器根据语法规则逐步解释和执行语法树中的各个元素，直到达到最终的解释结果。

### 3.14.2 类图



### 3.14.3 代价分析

解释器模式适用于以下场景：

* 当有一些特定问题需要解决，并且问题可以表示为一种语言或表达式时，可以考虑使用解释器模式。
* 当语法规则相对稳定但频繁变化时，解释器模式可以提供一种灵活的方式来处理这种变化。
* 当需要对语言或表达式进行扩展时，解释器模式可以很方便地添加新的解释器来支持新增的语法规则。

然而**解释器模式可能会引入较高的复杂性，因此在使用时需要权衡利弊，并根据具体情况选择合适的设计方案**。

## 3.15 迭代器模式（GOF模式）

### 3.15.1 实现描述

迭代器模式（Iterator Pattern）是一种行为设计模式，它提供一种方法顺序访问一个集合对象中的各个元素，而又无需暴露该对象的内部表示。迭代器模式可以用来遍历容器并访问容器的公共接口。

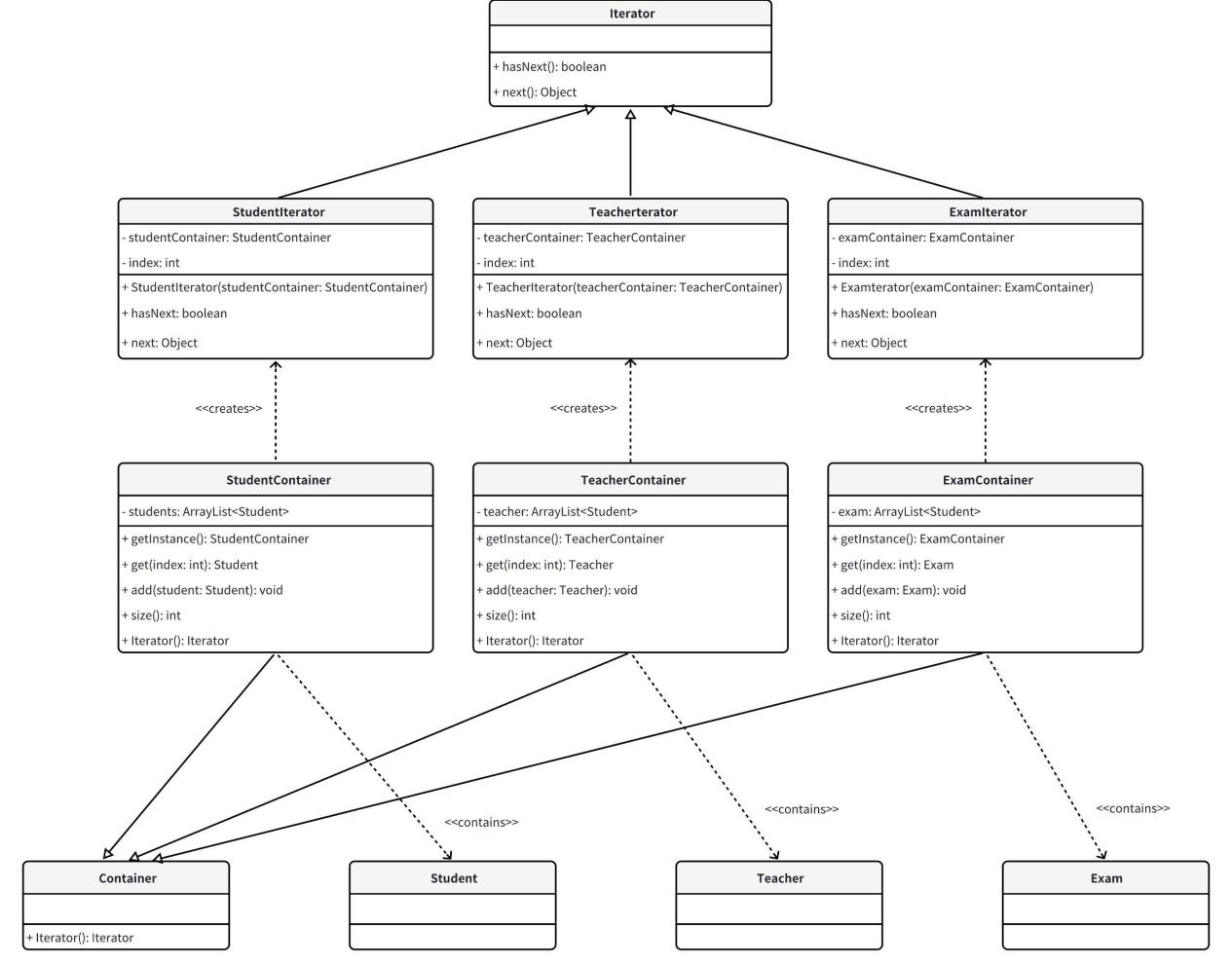
在“顶峰考研云”项目中，迭代器模式用于管理学生集合。Container接口定义了创建迭代器的方法，而Iterator接口提供了遍历集合的标准操作，如hasNext()和next()。StudentContainer类实现了Container接口，它维护学生对象的集合，并提供了添加学生和获取学生的方法。该类使用了单例模式，确保整个应用中学生容器的唯一性。

StudentIterator类实现了Iterator接口，它是StudentContainer的内部迭代器，负责遍历学生容器。它维护一个索引来跟踪当前遍历的位置。当next()方法被调用时，它返回当前学生并将索引向前移动。hasNext()方法检查是否还有更多的学生可以遍历。

通过迭代器模式，StudentContainer的客户端可以不了解集合的内部结构而遍历学生集合。这不仅简化了客户端的操作，而且使得不同的集合结构（如数组、链表等）能够使用相同的方式进行遍历，提高了代码的复用性。

Teacher和Exam的情况与Student同理。

### 3.15.2 类图



### 3.15.3 代价分析

**优点**

1. 抽象化：  
   迭代器模式提供了一个统一的接口，用于访问集合的元素，这使得客户端不需要知道集合的内部结构就可以遍历集合中的所有元素。
2. 解耦合：  
   它解耦了集合对象的元素遍历与实现，允许独立于数据结构的操作。这意味着可以更换底层的数据结构而不影响到遍历操作的实现。
3. 支持多种遍历：  
   可以为同一个集合提供多种迭代器，每个迭代器实现不同的遍历策略。
4. 同时遍历多个集合：  
   客户端可以同时对同一集合进行多个遍历，因为每个迭代器实例维护其自己的遍历状态。
5. 控制遍历的粒度：  
   遍历集合的粒度由迭代器控制，客户端只需请求下一个元素，无需关心遍历的逻辑。

**不足**

1. 额外的对象：  
   实现迭代器模式需要额外的对象和接口，这可能会稍微增加系统的复杂度。
2. 性能开销：  
   创建迭代器可能会引入一定的性能开销，特别是在遍历简单集合或需要高效遍历的场景中。
3. 迭代器失效：  
   如果集合在迭代过程中被修改，可能会导致迭代器失效，需谨慎处理这种情况，以避免出现运行时错误。
4. 过于通用：  
   在一些特定的情况下，迭代器模式可能提供了一种过于通用的解决方案，而客户端实际上只需要一个简单的遍历方式。
5. 复杂的迭代逻辑：  
   对于一些复杂的数据结构，实现迭代器可能会变得非常复杂，尤其是在保持迭代器状态一致性和效率方面。

## 3.16 延迟初始化模式（其他模式）

### 3.16.1 实现描述

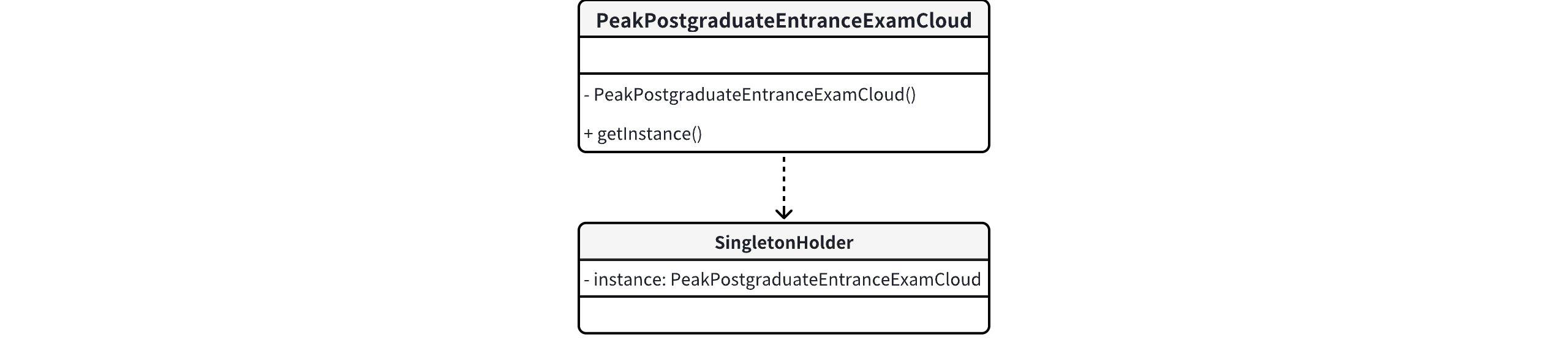
延迟初始化模式是一种在编程和设计中广泛使用的方法，其核心思想是将对象的实例化推迟到真正需要的时刻。这种模式适用于创建成本高昂或仅在特定条件下才需要的对象。通过延迟对象的创建，可以显著节省资源和提高程序的启动性能。

在“顶峰考研云”项目中，延迟初始化模式用于管理系统的主控制类 PeakPostgraduateEntranceExamCloud。此类采用私有构造函数，确保只能通过类内部进行实例化。类中包含一个静态内部类 SingletonHolder，其中维护了 PeakPostgraduateEntranceExamCloud 的单一实例。当 getInstance() 方法被调用时，如果实例尚未创建，它将被创建并初始化；如果已经创建，直接返回现有实例。

这种模式通过 Java 的类加载机制保证了实例的唯一性及线程安全性。当 PeakPostgraduateEntranceExamCloud 类被加载时，其内部类 SingletonHolder 不会被立即加载，而是在 getInstance() 方法第一次被调用时加载，从而实现延迟初始化。

此模式的主要优点是资源利用率高，减少了内存占用，并提高了应用性能。同时，在多线程环境下，无需额外的同步机制，就能保证实例的唯一性和线程安全。

### 3.16.2 类图



### 3.16.3 代价分析

**优点**

1. 减少资源消耗：  
   延迟初始化模式通过只在需要时创建对象，减少了资源消耗，尤其是对于那些创建成本较高的对象。
2. 提高性能：  
   应用启动时，不需要立即加载所有对象，从而提高应用的启动速度和运行效率。
3. 线程安全：  
   在多线程环境下，可以通过特定的实现方式（例如静态内部类）来确保对象的唯一性和线程安全，无需额外的同步处理。
4. 按需加载：  
   对象只有在真正被需要时才创建，有助于节省内存和其他计算资源。
5. 避免对象的重复创建：  
   保证在整个应用中，某些特定类的对象只会被创建一次，避免了重复创建的资源浪费。

**不足**

1. 复杂性增加：  
   实现延迟初始化可能会增加代码的复杂性，特别是在涉及多线程环境时。
2. 初始化延迟：  
   对象首次被使用时的初始化可能导致延迟，尤其是当对象的创建过程复杂或资源密集时。
3. 内存泄漏的风险：  
   如果实现不当，延迟初始化的对象可能导致内存泄漏，特别是在使用静态字段时。

### 3.16.4 出处

出自Erich Gamma, Richard Helm, Ralph Johnson和John Vlissides 的《设计模式：可复用面向对象软件的基础》。

## 3.17 中介者模式（GOF模式）

### 3.17.1 实现描述

中介者模式（Mediator Pattern）是一种行为型设计模式，旨在减少对象之间的直接通信，而是通过一个中介者对象来协调和管理各个对象之间的交互。这种模式有助于降低系统中对象之间的耦合度，使得对象之间更加独立、可复用和易扩展。

在中介者模式中，定义了一个中介者接口，该接口包含了对象之间交互的方法。具体的中介者类实现了这个接口，并负责协调和管理对象之间的通信。对象之间不再直接引用其他对象，而是通过中介者对象进行通信。

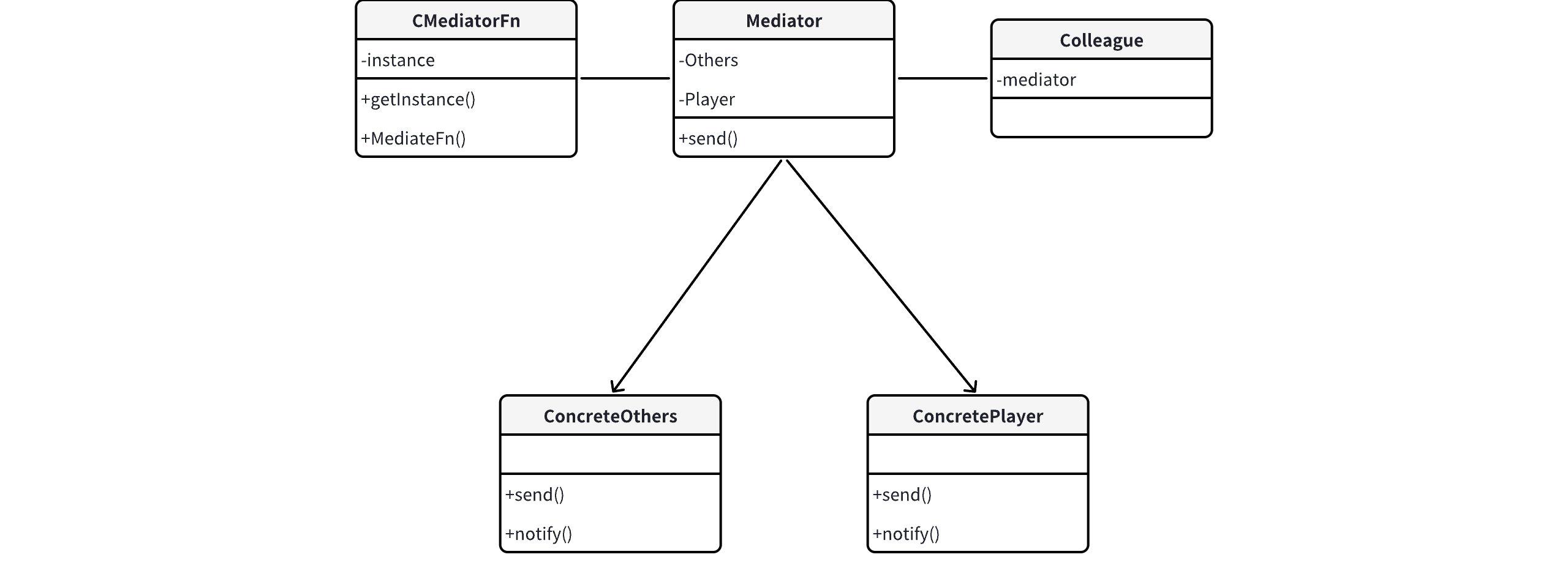
以下是中介者模式的主要组成部分：

1. Mediator（中介者）： 定义了中介者的接口，包含了对象之间交互的方法。具体的中介者类实现了这个接口，并负责协调和管理对象之间的通信。
2. ConcreteMediator（具体中介者）： 实现了中介者接口，负责具体的中介逻辑。它维护了对象之间的关系，并处理对象之间的通信。
3. Colleague（同事）： 定义了对象的接口，每个同事对象都知道中介者对象，并通过中介者来进行通信。
4. ConcreteColleague（具体同事）： 实现了同事接口，是真正参与到对象之间通信的具体对象。它们通过中介者对象来与其他同事对象进行交互。

在中介者模式中，中介者对象扮演了协调者的角色，负责处理对象之间的通信和协作。这种模式适用于对象之间有复杂的相互关系，而且这些关系不应该耦合在一起的场景。

中介者模式的优点包括降低系统的耦合度、集中管理对象之间的关系、使系统更易扩展。然而，中介者模式也可能导致中介者对象变得过于复杂，因此在设计时需要权衡对象之间的关系和中介者的复杂性。

### 3.17.2 类图



### 3.17.3 代价分析

1. 维护性方面： 中介者模式有助于降低对象之间的直接耦合，通过中介者对象进行通信，使得系统更容易维护。当需要修改对象之间的交互逻辑时，只需修改中介者类，而不需要修改每个同事类。这有助于降低维护成本。
2. 拓展性方面： 中介者模式支持相对良好的扩展性。如果需要添加新的同事类，只需创建新的同事类并在中介者中进行适当的调整。这样的变化不会影响其他同事类。但是，如果要添加新的交互逻辑，可能需要修改中介者类，这可能影响其他同事类。
3. 复杂性方面： 中介者模式有助于降低对象之间的复杂直接关系，使得系统更清晰。然而，引入中介者可能会增加一个额外的间接层，增加一些复杂性。对于初学者来说，理解中介者模式可能需要一定的学习成本。
4. 性能方面： 中介者模式引入了中介者对象来协调同事对象之间的通信，可能会带来一定的性能开销。在一些需要高性能的场景下，可能需要谨慎使用中介者模式，特别是当对象之间的直接通信开销相对较小的情况下。

总体而言，中介者模式在提高系统灵活性和降低耦合度的同时，可能会引入一些复杂性和性能开销。在设计中需要根据具体的需求和场景权衡使用中介者模式的利弊

## 3.18 备忘录模式（GOF模式）

### 3.18.1 实现描述

**备忘录（Memento）模式**，在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式 。

备忘录模式由三个主要组件组成：**发起人（Originator）、备忘录（Memento）和管理者（Caretaker）**。

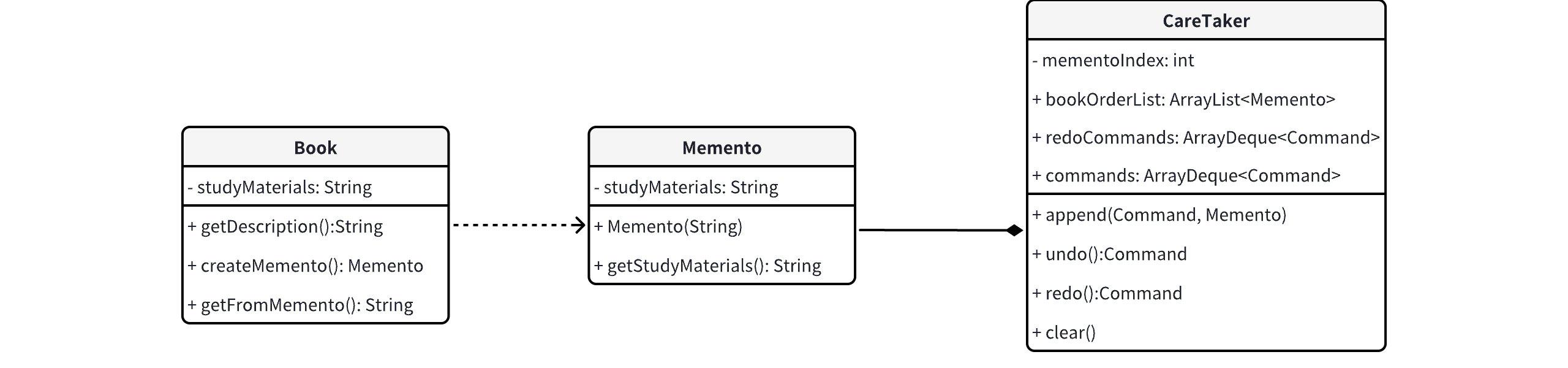
* **发起人（Originator）**是需要保存状态的对象。它会创建备忘录对象以保存当前的状态，并可以使用备忘录对象来恢复自身状态。
* **备忘录（Memento）**是存储发起人对象状态的对象。它包含了发起人对象的部分或全部状态信息，但无法被外部访问或修改。
* **管理者（Caretaker）**负责管理备忘录对象。它可以保存多个备忘录对象，以支持多次撤销或恢复操作。

在我们的顶峰考研云系统中，我们主要使用该模式来实现对用户点单命令流程的记录，从而辅助实现最后**总价的结算**，以及用户在点单过程中的**撤销和重做**操作。

在Memento中主要有以下两个主要类：

1. **Memento类：**在顶峰考研云的场景下，Memento是Book的备份类，CareTaker用于维护一个以Memento为元素的List，以便随时调取和增加备份，支持用户在点单的过程中进行点单操作的undo（撤销）和redo（重做）。在点单完毕后，教辅书籍的初始态（还未选择配套文具的时候）被保存在 CareTaker 中的 MementoList 中，进入配料选择后，每一次选择都将改变食物的状态，每个状态都会被创建为一个新的Memento 对象并存入 List 中。
2. **CareTaker类：**当用户选择 undo 时，CareTaker 会从 List 中选取最近一次的先前备份，通过getStudyMaterial()函数返回在购买此书籍的先前状态下的配套文具的选择，并赋值给当前选择的书籍，完成 undo。同时 CareTaker 中包含了一个 commandsList 和一个 redoCommandsList，分别存储了点单命令和被撤回的点单命令，当执行 undo 时，pop 出 commandsList 的一条命令并 push 进 redoCommandsList，当执行 redo 时，pop 出 redoCommandsList 中的一条命令并 push进 commandsList，当有新的点单命令时，清空 redoCommandsList中的所有命令，此时只能执行 undo，不能执行 redo。与购买书籍和添加配套文具相关的代码中包含了控制台的菜单输出以及选择输入，实际编程过程中将这些代码封装在 OrderStudyMaterial 类中。

### 3.18.2类图



### 3.18.3代价分析

备忘录模式记录了需要保存的对象的历史行为，并且有助于用户对其的撤销以及重做操作，然而在提供便利性的同时，备忘录模式也有其缺点以及使用所需要付出的代价：

1. **增加了系统的复杂性：**引入备忘录模式会增加系统中的额外类和对象，从而增加了系统的复杂性。这可能导致代码量的增加以及对额外类和对象的管理和维护。
2. **内存开销：**每个备忘录对象都需要存储一定量的状态信息，如果需要保存大量的状态信息或频繁创建备忘录对象，可能会消耗大量的内存空间。
3. **性能开销：**在某些情况下，创建和恢复备忘录对象可能会引入一定的性能开销。特别是当需要保存和恢复大量的状态信息时，可能需要花费较长的时间。
4. **安全性考虑：**备忘录模式涉及到对对象状态的读写操作，需要确保备忘录对象的数据不被外部篡改或访问，以保证对象状态的完整性和安全性。

不过，虽然备忘录模式存在一些代价和注意事项，但它也提供了一种有效的方式来保存和恢复对象的状态，特别是在需要支持撤销和重做操作、历史记录功能等场景中。在设计时，需要综合考虑系统的要求、性能需求和可维护性等因素，权衡利弊，选择合适的设计方案。

## 3.19 MVC模式（架构模式）

### 3.19.1 实现描述

MVC模式（Model-View-Controller Pattern）是一种架构设计模式，用于将应用程序分为三个主要的逻辑组件：模型（Model）、视图（View）和控制器（Controller）。这种分离有助于管理复杂的应用程序，因为可以独立地维护或修改每个部分。

在“顶峰考研云”项目中，MVC模式被用于管理学生信息的展示和更新。该模式的实现涉及三个主要组件：Student模型、StudentView视图和StudentController控制器。

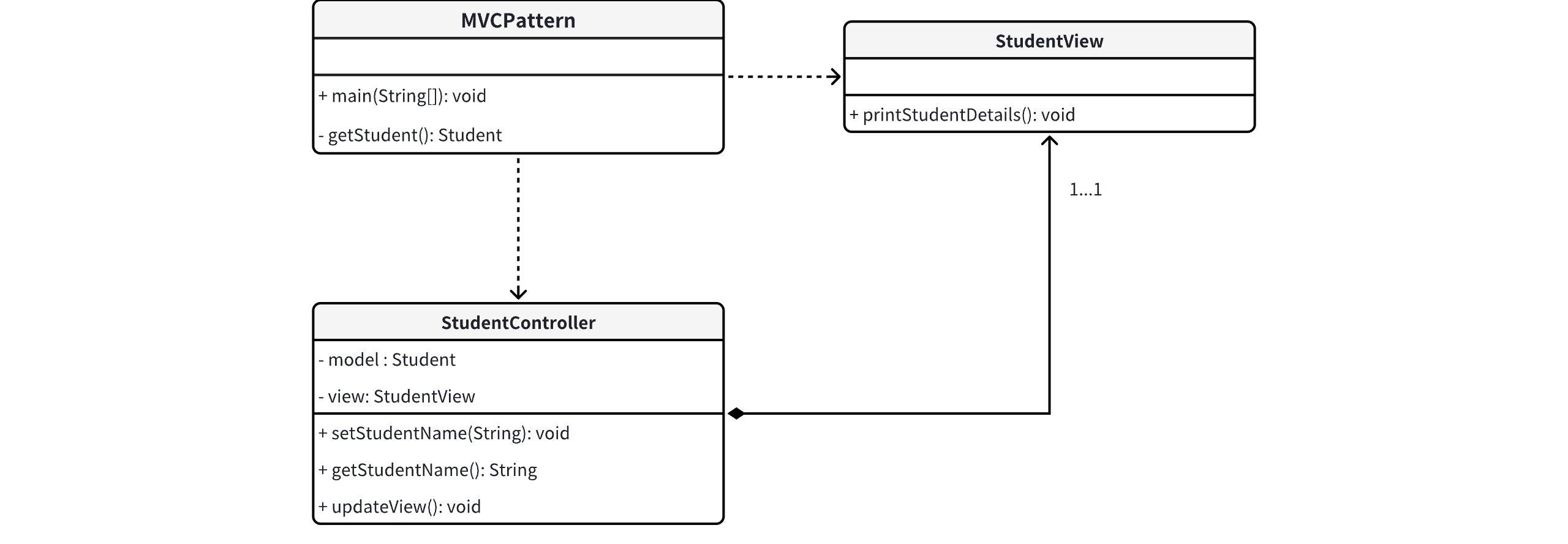
Student模型（Model）：这是应用程序的核心部分，代表了应用程序的数据结构和业务逻辑。在MVCPattern.java中，Student模型通过SimpleFactory模式创建，封装了学生的基本信息和行为。例如，它包含了学生的姓名和国籍信息，并提供了获取和设置这些信息的方法。

StudentView视图（View）：视图负责呈现数据（模型）给用户。在StudentView.java中，StudentView类负责显示学生的详细信息。它提供了printStudentDetails方法，该方法接收一个字符串参数（学生的姓名），并将其输出到控制台。这样，视图与模型的数据处理逻辑分离，确保了视图的简洁性和专注性。

StudentController控制器（Controller）：控制器作为模型和视图之间的中介，处理用户的输入并调用视图和模型适当的方法。在StudentController.java中，StudentController类包含了对Student模型和StudentView视图的引用。它提供了setStudentName和getStudentName方法来更新模型，并通过调用updateView方法来刷新视图。这样，控制器将用户的输入转化为对模型的操作，并选择合适的视图进行显示。

在MVCPattern.java的main方法中，这三个组件被实例化并用于展示和更新学生信息。首先，通过StudentFactory创建一个Student对象，并初始化StudentView和StudentController。然后，通过控制器来更新和展示学生信息。当用户输入新的学生姓名时，控制器更新模型并刷新视图，从而将新的信息呈现给用户。

### 3.19.2 类图



### 3.19.3 代价分析

**优点**

1. 分离关注点：  
   MVC模式通过将应用程序分为模型、视图和控制器三部分，实现了关注点的分离。这种分离使得管理复杂应用程序变得更加容易，因为可以独立地维护或修改每个部分。
2. 提高可维护性：  
   由于MVC分离了数据模型和用户界面，因此在修改用户界面或业务逻辑时，可以减少对其他部分的影响，从而提高应用程序的可维护性。
3. 增强可测试性：  
   在MVC模式中，由于模型与视图分离，可以独立对它们进行测试。特别是模型（业务逻辑）部分，可以在不涉及用户界面的情况下进行测试。
4. 灵活性和可扩展性：  
   视图和模型的独立性使得更换用户界面或修改业务逻辑变得更加容易。可以在不改变其他部分的情况下，添加新的视图或模型。
5. 促进团队协作：  
   MVC模式允许不同的开发者分别在模型、视图和控制器上工作，提高了团队协作的效率。

**不足**

1. 增加了系统的复杂性：  
   MVC模式引入了额外的层次和组件，这可能会增加系统的复杂性，特别是对于小型项目或简单的应用程序来说可能是不必要的。
2. 性能考虑：  
   由于模型、视图和控制器之间的频繁交互，可能会对性能产生一定影响。特别是在大型应用程序中，不恰当的设计可能导致性能瓶颈。
3. 过度工程化：  
   在一些简单的应用场景中，使用MVC模式可能会导致过度工程化，增加了开发的复杂性和时间成本，而不一定带来相应的好处。
4. 视图与模型间的潜在耦合风险：  
   虽然MVC旨在分离视图和模型，但在某些实现中，视图和模型之间可能仍然存在隐性的耦合。例如，视图可能需要依赖于模型的特定状态，这可能限制了模型的独立性和可重用性。

### 3.19.4 出处

出自Erich Gamma, Richard Helm, Ralph Johnson和John Vlissides 的《设计模式：可复用面向对象软件的基础》。

以及连续出版物：[4]任中方,张华,闫明松等. MVC模式研究的综述 [J]. 计算机应用研究, 2004, (10): 1-4+8.

## 3.20 空对象模式（其他模式）

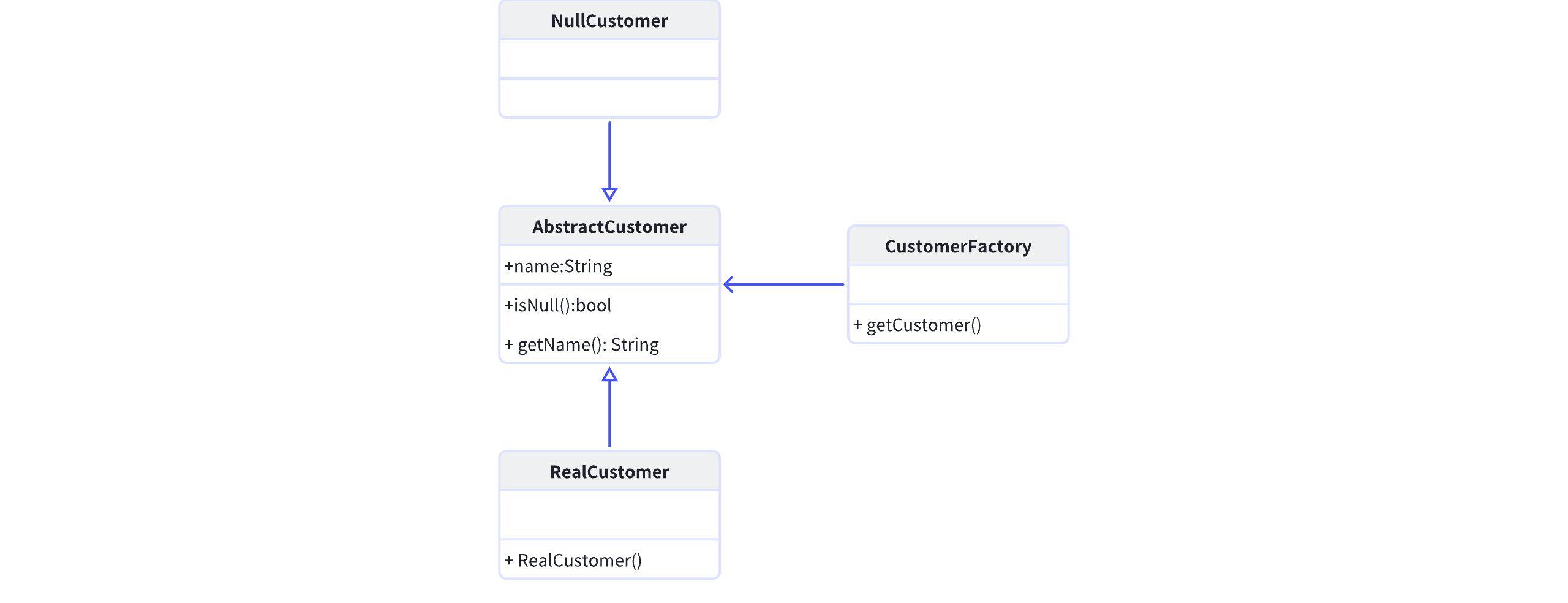
### 3.20.1 实现描述

空对象模式（Null Object Pattern）是一种设计模式，用于处理对象引用为空的情况，以避免空指针异常（NullPointerException）的发生。

在软件开发中，经常会遇到需要使用对象的情况，但有时候并不总是能够获取到有效的对象实例。传统的做法是在使用对象之前进行非空检查，以确保对象引用不为空。然而，这种方式需要频繁地进行空值检查，使代码变得冗长且难以维护。

空对象模式通过引入一个空对象来解决这个问题。这个空对象是一个实现了特定接口或继承自特定类的对象，它的方法实现通常是空操作或默认操作。当一个对象引用为空时，可以使用这个空对象来代替，从而避免了空指针异常。

### 3.20.2 类图



### 3.20.3 代价分析

**使用空对象模式的好处包括：**

1. **简化代码：**通过使用空对象，可以避免频繁的空值检查，使代码更加简洁和易读。
2. **避免异常：**空对象的方法实现通常是空操作或默认操作，因此在使用空对象时不会抛出空指针异常，增强了代码的健壮性。
3. **提供默认行为：**空对象可以提供一些默认的行为，当具体对象不可用时，可以使用这些默认行为进行处理。

空对象模式的一个典型应用是在集合类中，当集合为空时，返回一个空集合对象而不是 null，这样可以避免在对集合进行操作时出现空指针异常。然而在某些情况下，空对象可能并不适用或不符合需求，因此在使用空对象模式时需要根据具体情况进行评估和选择。

### 3.20.4 出处

出自连续出版物：[3]吴清寿. Null Object模式研究 [J]. 吉林师范大学学报(自然科学版), 2013, 34 (01): 120-122.

## 3.21 观察者模式（GOF模式）

### 3.21.1 实现描述

观察者模式（Observer Pattern）也称为发布-订阅模式（Publish-Subscribe Pattern），是一种对象行为型模式，它定义了对象之间的一种一对多的依赖关系，使得每当一个对象改变状态时，所有依赖于它的对象都会得到通知并被自动更新。观察者模式是分离表示层和数据或逻辑层的关键设计模式之一，广泛应用于实现事件处理系统。

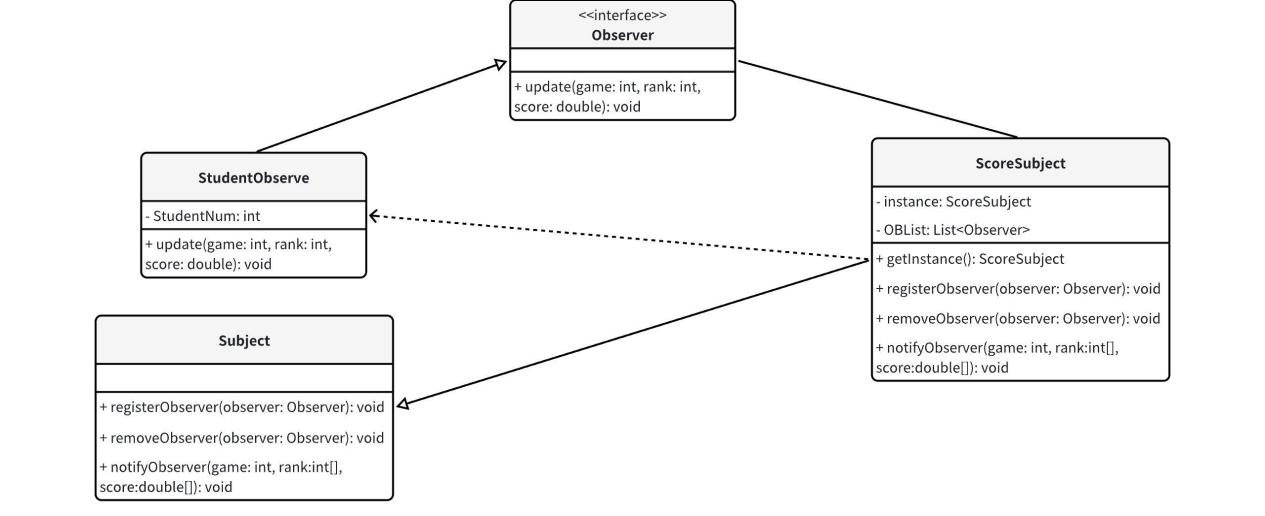
在观察者模式的经典实现中，包含以下主要组件：

* Subject（主题）：可以有多个观察者对象依赖于主题。主题提供了注册和移除观察者对象的接口，并且负责通知观察者其状态的变化。
* Observer（观察者）：定义了更新接口，用于在主题状态变化时获取状态更新。观察者在这种模式下必须实现该接口，以便确保它们可以接收到状态更新的通知。
* ConcreteSubject（具体主题）：具体主题保存具体观察者的状态，并在状态发生变化时发送通知。
* ConcreteObserver（具体观察者）：具体观察者实现了观察者接口，保持一个指向具体主题的引用，存储有关状态的信息，这些状态应与主题的状态保持一致。

观察者模式在设计中的应用使得主题和观察者之间的通信变得抽象和自动化。它特别适用于分布式系统的事件处理，模型-视图-控制器（MVC）架构的实现，以及跨多个层次的交互。

在“顶峰云”考研模拟系统中，观察者模式被用来实现成绩公示板的功能。系统中的ScoreSubject类充当主题角色，负责维护观察者列表，并在成绩或排名发生变化时通知所有注册的观察者。每个观察者是一个StudentObserve实例，代表不同的学生。当ScoreSubject更新排名和分数时，它会调用每个StudentObserve的update方法，将最新的成绩和排名信息同步给学生。通过这种方式，系统能够模拟现实世界中公示板上公布考试排名的场景，提升用户体验。

### 3.21.2 类图



### 3.21.3 代价分析

优点

* 高度的解耦：主题和观察者之间的松耦合性质使得我们可以复用和扩展他们而无需太多的修改。
* 动态联动：一旦主题的状态发生变化，所有的观察者都会得到通知并自动更新，这提供了即时反馈。
* 抽象化的交互：观察者和主题之间的交互是通过抽象接口进行的，不依赖于具体的类，易于扩展。

缺点

* 潜在的过度更新：如果主题的状态改变频繁，观察者可能会收到大量的更新，这可能导致性能问题。
* 难以确保观察者处理顺序：由于观察者是不同的对象，不能保证它们更新的顺序，这在某些情况下可能是必要的。
* 内存泄漏风险：如果观察者没有正确地注册和注销，可能会导致内存

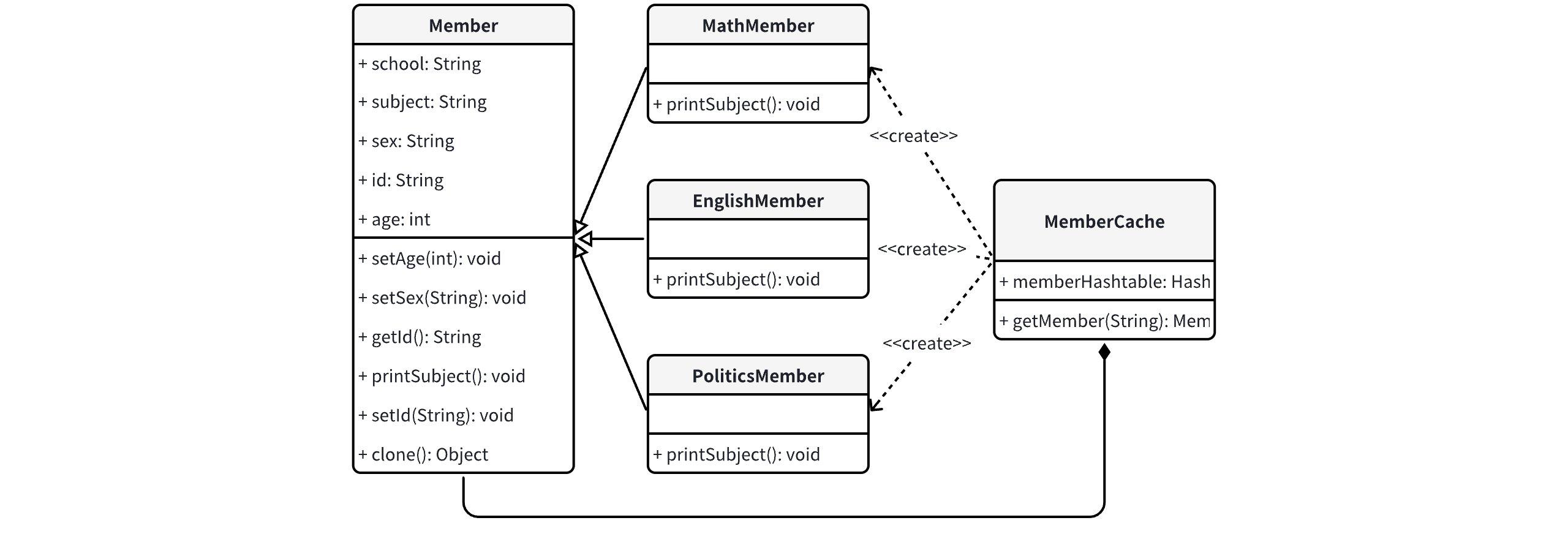
## 3.22 原型模式（GOF模式）

### 3.22.1 实现描述

原型模式是一种创建型设计模式， 其能够复制已有对象， 而又无需使代码依赖它们所属的类。

在“顶峰考研云”中，Member类实现了Cloneable接口，包含学校、学科、性别、年龄等属性，具体的学科成员类MathMember、EnglishMember、PoliticsMember集成自Member类，实现了具体的构造方法和打印学科信息的方法。PrototypeFunction类作为原型模式的客户端，通过输入学科名称来获取该学科的学生对象。MemberCache类用来存储不同学科的学生对象，getMember()方法用来根据学科名称获取对应学科的学生对象的克隆。

### 3.22.2 类图



### 3.22.3 代价分析

* 优点

1. 对象创建效率高： 通过复制现有对象，避免了对象的重复创建和初始化过程，提高了对象的创建效率。
2. 简化对象创建： 客户端无需知道具体类，只需知道对象的类型和克隆方法即可创建新对象。
3. 减少子类数量： 由于可以通过克隆来创建新对象，无需创建大量具体子类，减少了类的数量。

* 缺点

1. 深拷贝问题： 如果对象包含引用类型的成员变量，克隆时可能需要考虑深拷贝，否则共享相同的引用。
2. 克隆方法的实现： 需要在每个具体类中实现克隆方法，确保正确复制对象的内部状态。
3. 对象构造复杂： 如果对象的构造过程比较复杂，克隆时可能需要进行一定的调整和处理。

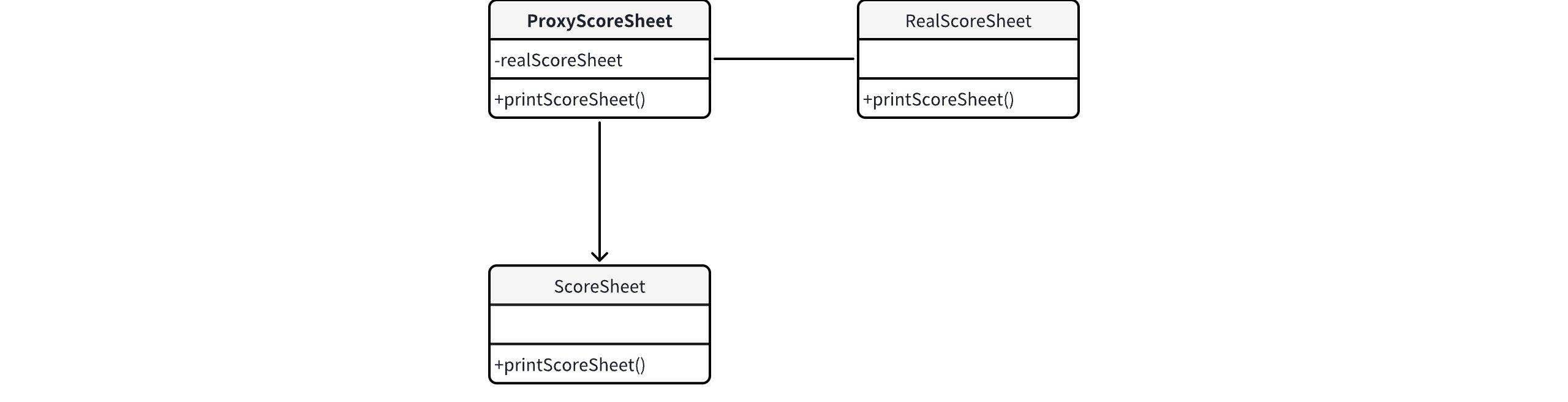
## 3.23 代理模式（GOF模式）

### 3.23.1 实现描述

代理模式（Proxy Pattern）是一种结构型设计模式，它通过引入代理对象来控制对其他对象的访问。代理对象充当客户端与实际对象之间的中介，可以起到控制、保护和增强实际对象的作用。代理模式主要包括以下几个角色：

1. Subject（抽象主题）： 定义了实际对象和代理对象的共同接口，客户端通过这个接口访问实际对象。
2. RealSubject（真实主题）： 实际对象，实现了抽象主题接口，是客户端真正关心的对象。
3. Proxy（代理）： 代理对象，实现了抽象主题接口，同时持有对真实主题的引用。代理对象通常负责控制对真实主题的访问，可以在访问前后执行一些额外的逻辑。
4. Client（客户端）： 使用代理模式的客户端，通过抽象主题的接口与代理对象交互，无需直接访问真实主题。

### 3.23.2 类图



### 3.23.3 代价分析

1. 维护性方面： 代理模式可以通过引入代理对象来隔离客户端和真实对象，使得在不影响客户端的情况下修改真实对象变得更容易。当需要修改或扩展真实对象的行为时，只需修改或扩展代理对象，而不必修改客户端代码，降低了维护成本。
2. 拓展性方面： 代理模式支持良好的拓展性。可以通过添加新的代理对象来实现不同的控制逻辑，而无需修改客户端或真实对象的代码。这种灵活性使得系统更易于扩展。
3. 复杂性方面： 代理模式引入了代理对象，可能增加了系统的复杂性。特别是当存在多个不同类型的代理对象时，可能需要更复杂的管理和设计。
4. 性能方面： 代理模式可能会引入一些性能开销，因为访问真实对象的过程需要通过代理对象。在某些对性能要求极高的场景中，需要谨慎使用代理模式。

总体而言，代理模式在提高系统灵活性、降低维护成本和实现横切关注点等方面具有优势。然而，在一些特定场景下，可能需要权衡使用代理模式所带来的复杂性和性能开销。

## 3.24 雇工模式（其他模式）

### 3.24.1 实现描述

雇工模式（Servant Pattern）也叫做仆人模式，是行为模式的一种，其意图是：为一组类提供通用的功能，而不需要类实现这些功能，它是命令模式的一种扩展。

其中有如下三种角色:

IServiced : 用于定义服务内容的接口

Serviced : 具体的服务提供者

Servant : 执行者,即雇工类, 用于执行服务

IServiced用于定义一组类所具有的功能，而Serviced针对不同的服务对象有不同的服务内容，也就是具体实现IServiced接口所定义的功能。功能定义完毕后，我们需要由一个雇工来执行这些功能。简单而言，就是需要有一个执行者，可以把一组功能聚集起来，这个执行者就是servant。

在整个雇工模式中，所有具有IServiced功能的类可以实现该接口，然后由雇工类Servant进行集合，完成一组类不用实现通用功能而具有相应职能的目的。

在日常的开发中, 我们可能已经接触过雇工模式,只是没有把它抽离出来,也没有汇编成册. 雇工模式是命令模式的一种简化, 但它更符合我们实际的需求,更容易引入开发场景中.

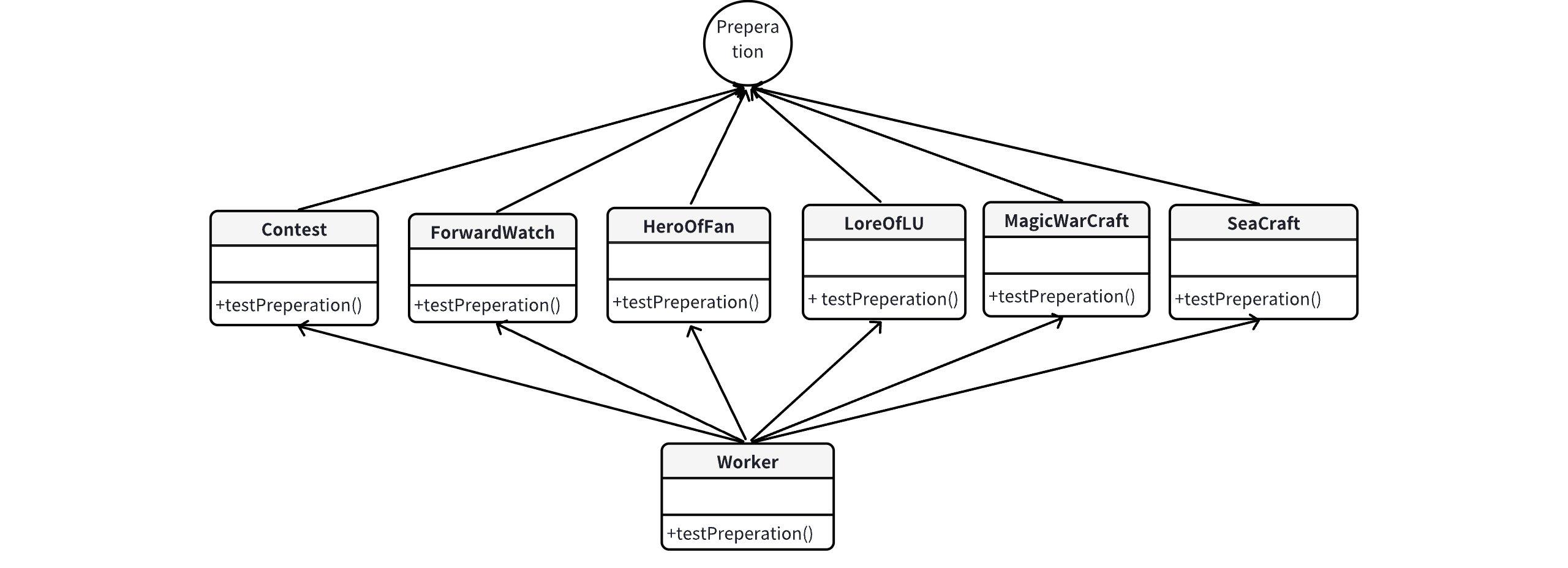
该模式由两个函数构成

1.examPreparation()进行赛前准备

2.prepare()执行examPreparation()

在每次考试开始前，都需要进行一系列的准备过程，这些考前准备事项在本项目中模拟成一个函数examPreparation()。 我们为准备设置了一个专门的接口类Preparation，其中设置了examPreparation()每个考试都实现了该接口，并重写了examPreparation()。每个考试类不用自己去实现这个准备函数，我们另外设置了一个Worker类，其有一个特定的准备函数prepare()，用于执行各个考试类的准备函数。在实际调用时，考试正式开启前会创建一个Worker对象，并使其执行对应的考试类的gamePreparation()，完成考前准备后，考试正式开启。

### 3.24.2 类图



### 3.24.3 代价分析

**优点：**

1. **代码可重用性：** 雇工模式通过将通用功能封装到一个雇工类中，促进了代码的可重用性，可以在多个类或对象之间共享。
2. **中心化维护：** 它允许在一个地方集中维护通用服务，更容易更新和增强功能。
3. **封装性：** 雇工模式封装了通用行为，使代码更清晰和模块化。

**缺点：**

1. **增加耦合度：** 雇工和其客户之间的耦合可能增加，对雇工的更改可能影响多个客户。
2. **复杂性：** 当处理大量服务或多样化客户时，该模式可能引入复杂性。

### 3.24.4 出处

出自专(译)著:[1]谭勇德.设计模式就该这样学:基于经典框架源码和真实业务场景.电子工业出版社,2020.754-757

## 3.25 简单工厂模式（其他模式）

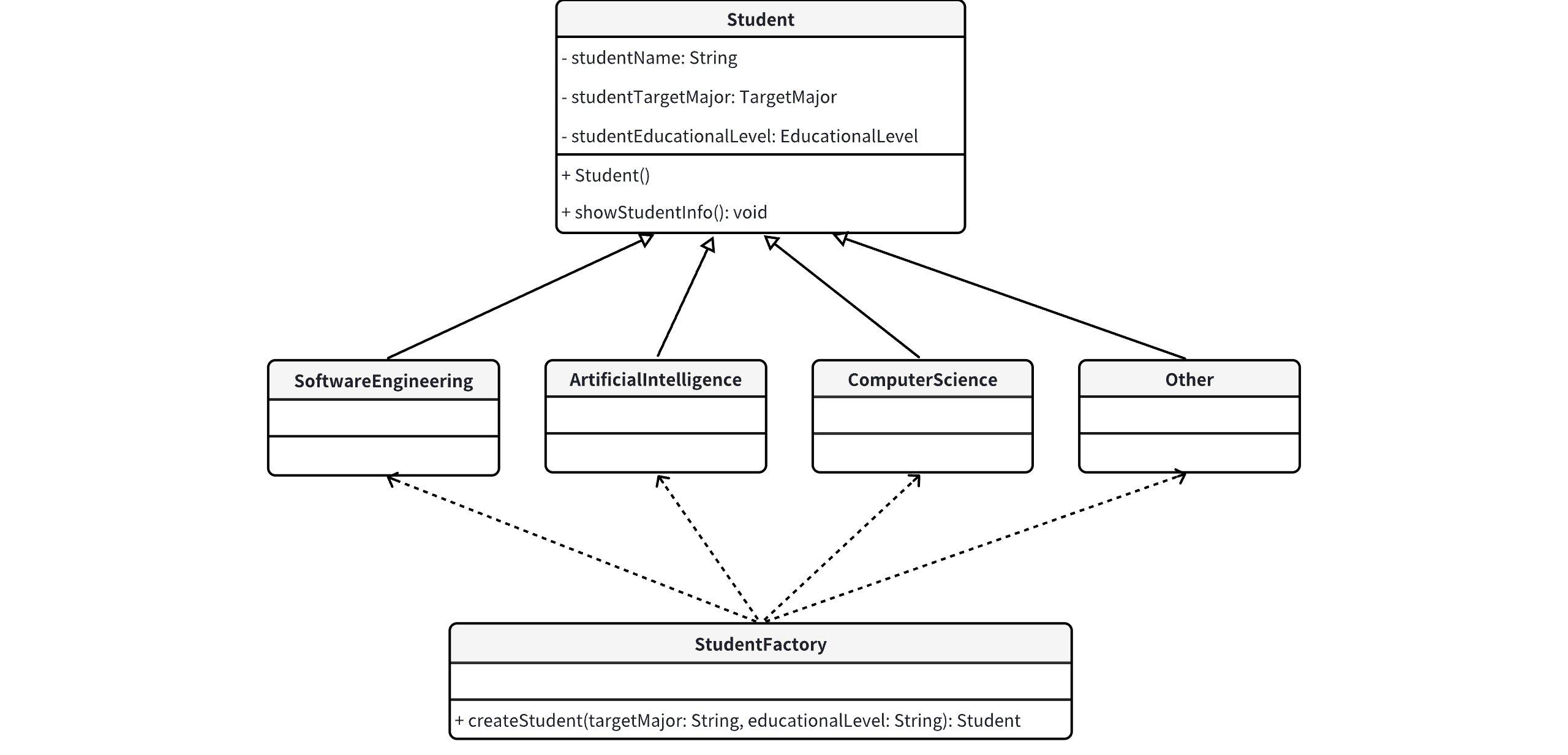
### 3.25.1 实现描述

简单工厂模式（Simple Factory Pattern）是一种创建型设计模式，用于在不直接实例化对象的情况下创建对象。这种模式通过创建一个工厂类，通过调用其方法来获取所需对象的实例，从而隐藏了创建对象的具体逻辑。简单工厂模式在对象的创建逻辑复杂，或者客户端不需要知道它所创建的对象的类时特别有用。

在“顶峰考研云”项目中，简单工厂模式用于基于学生的目标专业和教育水平（本科或本科毕业生）来创建和初始化学生对象。StudentFactory类通过其createStudent方法实现了这一点。

StudentFactory类中的createStudent方法接收目标专业和教育水平作为参数。它首先使用StudentIngredientFactory来创建并设置EducationalLevel和TargetMajor对象。然后，它创建一个Student对象，并根据提供的参数初始化学生的特征。根据学生是否为本科毕业生（二战考研），这些特征会有不同的初始化值，以反映现实生活中的不同情况。

### 3.25.2 类图



### 3.25.3 代价分析

**优点**

1. 封装性：  
   简单工厂模式封装了创建实例的逻辑，客户端不需要知道具体的类名，只需要知道所需对象的类型。这有利于隔离复杂的创建逻辑，降低客户端与具体类的依赖。
2. 集中管理：  
   所有的对象创建都通过一个工厂来管理，当需要更改、扩展或维护对象的创建逻辑时，只需要修改工厂类，便于管理和维护。
3. 客户端代码简洁：  
   客户端代码通过简单的接口请求对象，使得客户端代码更加简洁，易于理解和使用。
4. 分离职责：  
   工厂类负责对象的创建，客户端类负责对象的使用，实现了职责分离。

**不足**

1. 违反开闭原则：  
   当需要添加新的类时，工厂类可能需要修改其代码来支持新的类，这违反了开闭原则（软件实体应当对扩展开放，对修改封闭）。
2. 扩展困难：  
   如果产品类族频繁变动min，每次变动都可能需要修改工厂类的代码，导致系统的扩展性不好。
3. 逻辑复杂时不适用：  
   当创建逻辑非常复杂或者类的实例化过程涉及大量的配置时，简单工厂模式的一个工厂类可能会变得非常庞大、复杂，不易管理。
4. 过多的静态方法：  
   使用简单工厂模式通常会导致工厂类中有过多的静态方法，这可能会影响到系统的灵活性和可测试性。

**3.25.4 出处**

出自连续出版物：[2]秦澎涛,王苏文. 简单工厂模式在数据访问层中的应用 [J]. 计算机工程与设计, 2009, 30 (07): 1799-1801.

## 3.26 状态模式（GOF模式）

### 3.26.1 实现描述

状态模式（State Pattern）是一种行为型设计模式，旨在通过将对象的各种状态封装为独立的类来简化复杂的条件选择逻辑。这种模式允许对象在内部状态改变时改变其行为，使得对象看起来似乎改变了其类。在这种模式下，每个状态都定义了一系列相关的行为，而状态转换可以被封装在状态对象内部，从而实现了状态和行为之间的解耦。

应用于“顶峰考研云”系统，状态模式可以有效地管理学生的学习状态，如“良好”、“完美”和“疲劳”。这些状态直接关联到学生的行为，如购买书籍或参加考试，以及这些行为如何影响学生的状态。

以下是状态模式的关键组件及其作用：

1. 上下文类 StateContext

* 作用：维护指向当前状态对象的引用，代表学生的当前学习状态。
* 功能：提供 setStudentState 方法以允许状态对象触发状态转换，以及 getState 方法来获取当前状态的数值表示，这对于分析学生的学习表现或情绪状态十分有用。

1. 抽象状态类 StudentState

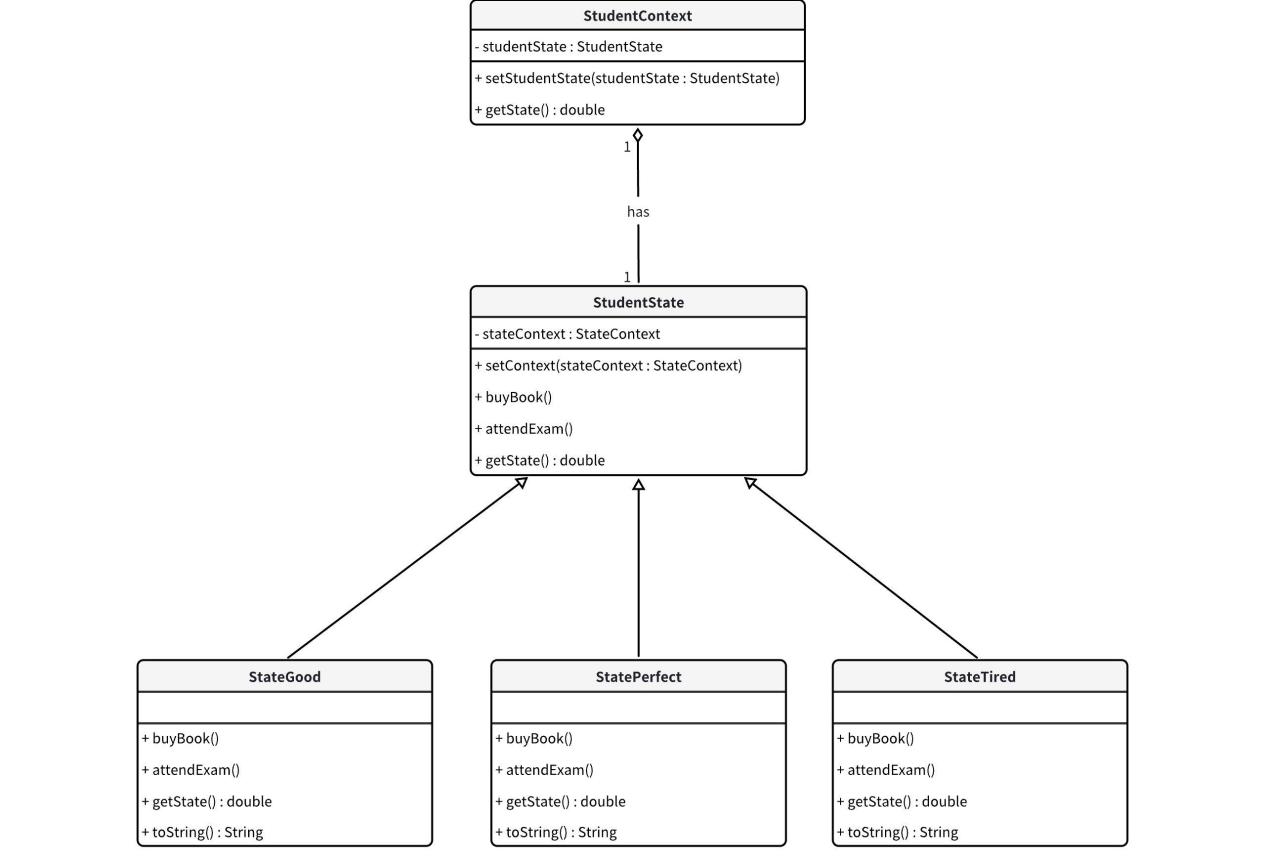
* 作用：定义所有具体状态类必须实现的行为接口。
* 功能：包含对 StateContext 的引用，使状态对象能够请求改变上下文的状态。定义了如 buyBook 和 attendExam 的抽象方法，以及一个方法来返回状态的数值表示，这些方法决定了学生在特定状态下的行为和状态转换逻辑。

1. 具体状态类（例如 StateGood, StatePerfect, StateTired）

* 作用：实现具体状态下的行为。
* 功能：每个状态类根据当前的状态对同一行为做出不同的反应，从而导致状态的转换。例如，StateGood 状态下，购买书籍可能会转换到 StatePerfect，而参加考试可能导致转换到 StateTired。

在“顶峰考研云”系统中，使用状态模式使得学生状态的管理和行为的实现更加清晰、灵活，并且易于维护和扩展。它允许状态逻辑集中管理，同时减少了代码中的条件判断，提高了代码的可读性和可维护性。此外，该模式也便于未来添加新的状态或修改现有状态的行为。

### 3.26.2 类图



### 3.26.3 代价分析

1. 实现复杂度

* 增加：引入状态模式增加了初期的设计和实现复杂度。您需要为每种状态创建一个单独的类，并定义状态间的转换逻辑。
* 减少：长期来看，状态模式简化了维护和扩展与状态相关的行为。例如，添加新状态或修改现有状态的行为更加直接，不需要修改大量的条件判断逻辑。

1. 可维护性

* 增加：状态模式使得状态相关代码更加模块化和集中，从而简化了维护。状态逻辑封装在单独的类中，易于理解和修改。
* 减少：初期，理解和实施状态模式可能需要更多时间和学习，尤其是对于不熟悉该模式的开发者。

1. 可扩展性

* 增加：状态模式提供了极高的可扩展性。要引入新的状态或改变状态转换逻辑，只需添加或修改相应的状态类即可。
* 减少：状态类可能变得众多，导致系统中的类数量增加，这需要更好的管理和组织。

1. 性能

* 影响较小：状态模式通常对性能影响不大。它主要影响的是代码的组织结构，而不是执行效率。

## 3.27 单例模式（GOF模式）

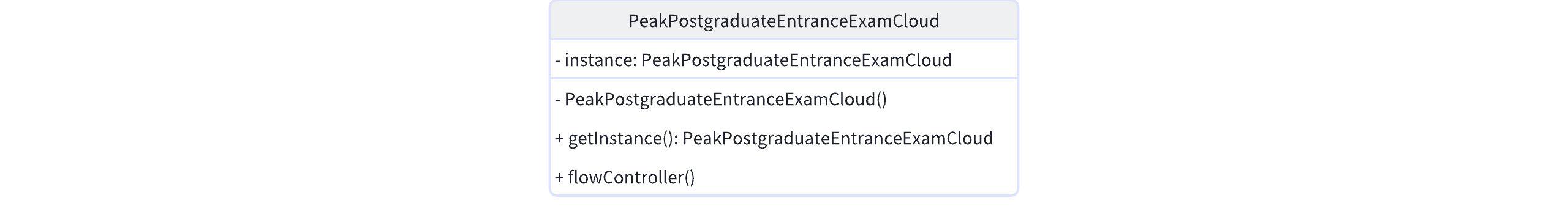
单例模式（Singleton Pattern）是一种创建型设计模式，用于保证一个类只有一个实例，并提供全局访问点。

单例模式的核心思想是通过限制类的实例化过程，确保一个类在任何情况下都只能创建一个对象实例。这个实例通常被定义为静态成员变量，并在类内部进行实例化。对外提供一个公共的静态方法来获取该实例，以实现全局的访问。

### 3.27.1 实现描述

在我们的设计中，此类只有一个顶峰考研云对象，因此PeakPostgraduateEntranceExamCloud 类 采 用 单 例 模 式 来 实 现 。PeakPostgraduateEntranceExamCloud 类的实例生成只在第一次调用其构造函数的时候使用。使用 getInstance 函数创建唯一实例。

### 3.27.2 类图



### 3.27.3 代价分析

单例模式的主要代价包括以下几个方面：

1. **可测试性降低：**由于单例对象的全局性质，导致单元测试时难以通过 mock 或者 stub 的方式进行替换，从而对单元测试造成一定的影响。
2. **系统耦合增加：**单例对象是全局唯一的，其他对象需要使用该单例对象的时候，都必须直接或间接地引用该单例对象，这样会增加系统对象之间的耦合性，从而对系统架构设计造成影响。
3. **并发性能降低：**在多线程环境下，如果单例对象本身不具备并发安全性，那么就需要加锁等同步机制来保证其线程安全性，这样会降低系统的并发性能。
4. **私有构造方法限制了子类化：**由于单例模式的实现方式，往往需要将构造方法私有化，这样就限制了单例类的子类化能力，从而可能对系统的扩展性造成一定的影响。

在使用单例模式时，需要权衡其优缺点，并根据具体情况来进行选择和应用，避免过度使用单例模式，从而导致系统的设计复杂性增加，可维护性降低等问题。

3.28 策略模式（GOF模式）

### 3.28.1 实现描述

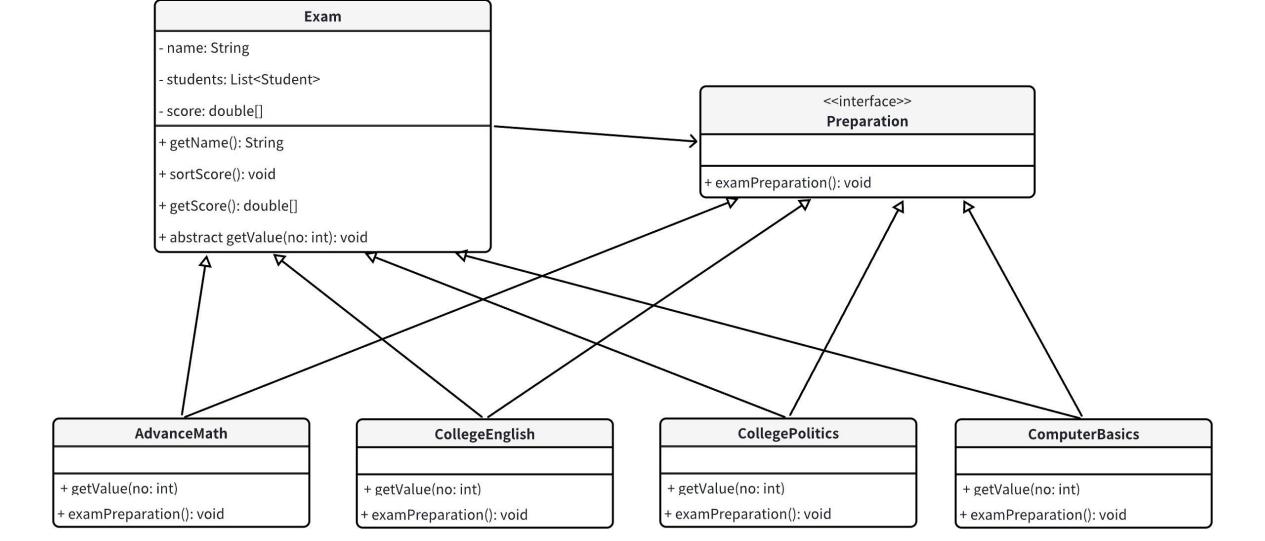
策略模式（Strategy Pattern）是一种软件设计模式，它允许在运行时选择算法的行为。在这种模式下，算法被封装在独立的策略类中，且它们是可互换的。策略模式是一种行为型模式，它主要用于分离算法，根据上下文环境动态选择算法。策略模式通常包含三个部分：

* 策略接口（Strategy Interface）：这是一个接口或抽象类，定义了所有支持的算法的公共操作。
* 具体策略类（Concrete Strategies）：实现了策略接口的类，提供具体的算法实现。
* 上下文（Context）：使用策略接口的类。上下文不直接执行算法，而是将工作委托给策略对象。

在“顶峰云”考研模拟系统中，策略模式用于处理不同科目考试成绩的计算逻辑。每个科目的考试流程由相应的策略类实现，例如AdvancedMath、CollegePolitics和CollegeEnglish，而每个科目的成绩计算方法有所不同，这通过不同的策略类实现。例如，AdvancedMath类重写了getValue方法来特定地计算高等数学的考试成绩。系统中的Exam类扮演上下文的角色，根据提供的exam name动态选择合适的策略类，从而实现不同科目考试流程的灵活处理。

* 策略接口：定义了一个getValue方法，用于计算考试成绩。
* 具体策略类：AdvancedMath, CollegeEnglish, CollegePolitics, ComputerBasics类，实现了getValue方法，根据学生的能力值和状态参数来计算特定科目的成绩。
* 上下文（Context）：Exam类，根据考试科目选择使用哪个策略类。

### 3.28.2 类图



### 3.28.3 代价分析

优点：

* 灵活性：不同科目的成绩计算方法可以独立变化，不影响其他科目。
* 可扩展性：新增科目时，只需添加新的策略类，无需修改现有代码。
* 封装性：每个科目的成绩计算逻辑被封装在单独的策略类中，易于维护。

缺点：

* 类的数量：每增加一个科目，就需要增加一个新的策略类，可能导致类的数量急剧增加。
* 复杂度：策略模式增加了代码的复杂度，需要理解和维护更多的类。

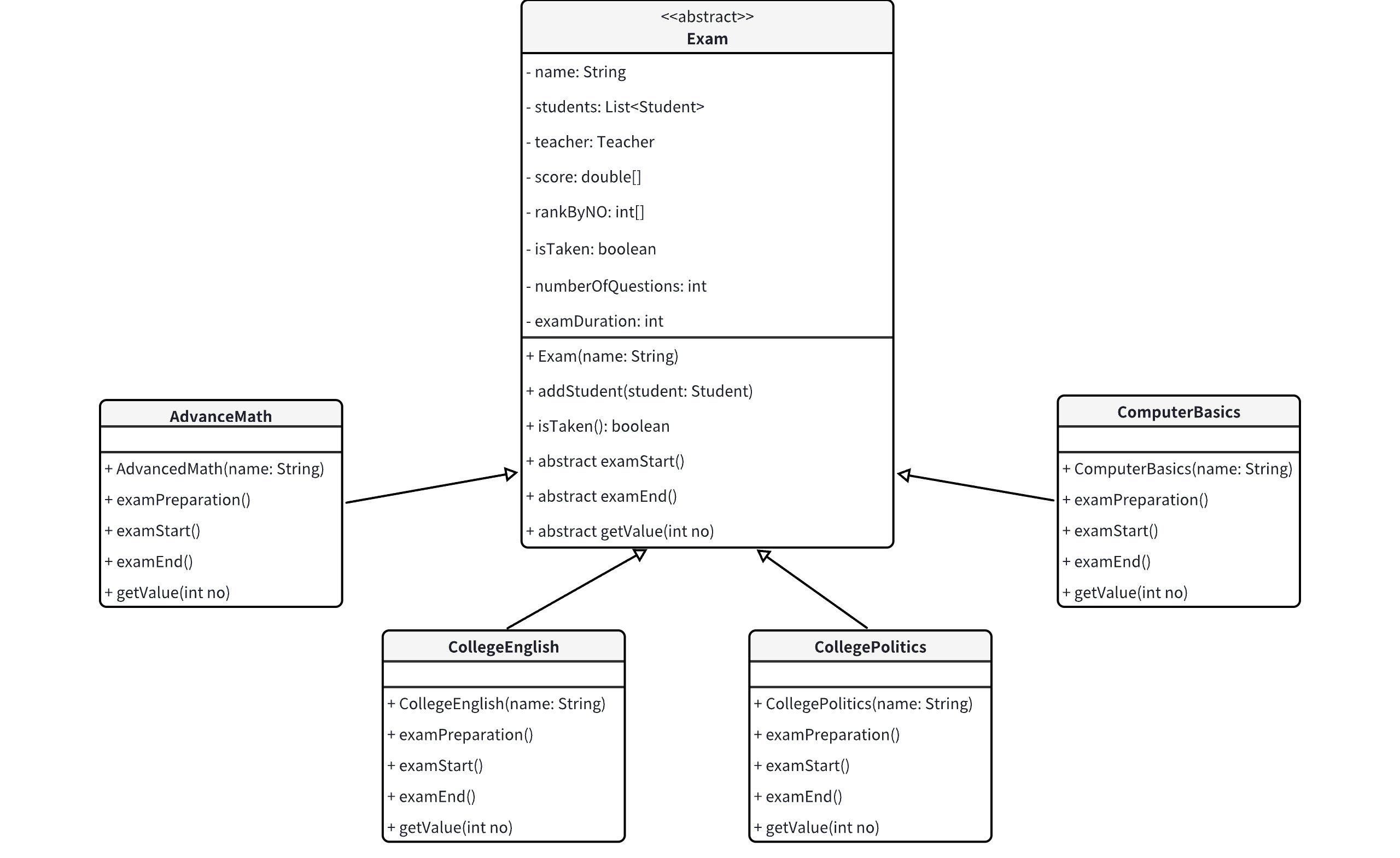
## 3.29 模板模式（GOF模式）

### 3.29.1 实现描述

模板模式（Template Pattern）是一种基于继承的设计模式，它定义了操作中的算法的骨架，允许子类在不改变算法结构的前提下重写算法的某些步骤。这种模式属于行为型模式类别，它主要是为了封装一系列步骤和流程，提供一个方法模板。该模式中的关键是有一个抽象类（通常是一个具有多个抽象方法和一个模板方法的基类），这个模板方法定义了一系列基本操作的执行顺序。这些基本操作定义为抽象方法，具体实现则留给子类去完成。通过这种方式，模板方法在超类中建立了算法的框架，而步骤的具体实现则被推迟到子类中。

在“顶峰云”考研模拟系统中，我们定义了一个抽象基类Exam，其中封装了考试流程的基本结构，如考试开始（examStart）以及考试结束（examEnd）。每种具体的考试类型如AdvancedMath都继承自Exam类，并实现了这些方法以适应特定科目的需求。这样，所有考试都遵循相同的基本流程，但每个具体考试可以有其特定的行为和规则。

### 3.29.2 类图



### 3.29.3 代价分析

优点

* 代码复用：模板模式允许在不同考试科目间重用考试流程的代码，减少重复代码。
* 扩展性：新的考试科目可以通过继承基类Exam并实现特定的方法来轻松添加。
* 维护性：具有共同行为的考试流程集中在一处，便于管理和维护。

缺点

* 限制性：模板模式可能会限制某些子类的灵活性，因为它们需要遵循定义好的步骤。
* 复杂性：对于只有少量变化或非常简单的流程，模板模式可能会导致不必要的复杂性。

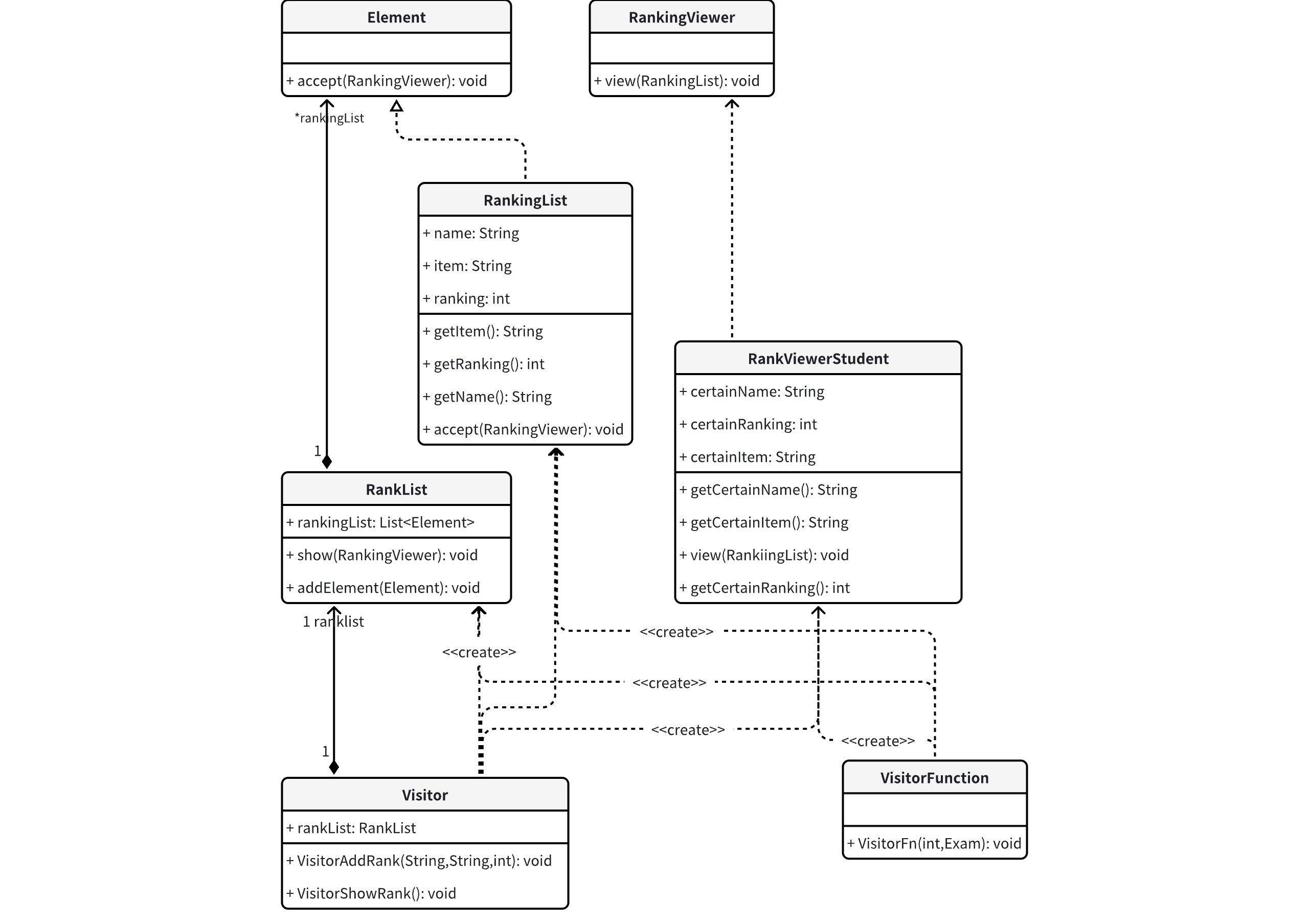
## 3.30 访问者模式（GOF模式）

### 3.30.1 实现描述

访问者模式（Visitor Pattern）是一种行为设计模式，用于将算法与对象结构分离，使得可以在不修改对象结构的前提下定义新的操作。该模式适用于对象结构较为稳定，但经常需要在此结构上定义新的操作的场景。

在“顶峰考研云”中，Element是单个元素的接口，声明了accept方法用于接收访问者。RankingList类实现了Element接口，表示排行榜信息、包括学科、姓名、排名等属性，通过实现accept方法，允许RankingViewer访问者查看排行榜信息。RankingView接口是查看排名的接口，声明了view方法，用于查看排行榜信息。RankList类表示排名表，包含一个排名列表，通过addElement方法添加元素，通过show方法供排名查看者查看排名。RankViewingStudent是观察者类，实现了RankingViewer接口，它记录查看的学科、姓名、排名并提供相应的获取方法。Visitor类是客户端，用于使用访问者模式，将学科的某个学生的排名添加到排名表中和查询指定学科中某个学生的排名，最后是VisitorFunction类，它包含了一个visotorFunction方法，该方法接受学生编号和考试对象，将学生的排名信息添加到排名表中，并通过观察者查看排名信息。

### 3.30.2 类图



### 3.30.3 代价分析

* 优点

1. 解耦操作和元素结构： 可以在不修改元素结构的情况下定义新的操作，提高了系统的可扩展性。
2. 增加新操作更灵活： 新的操作可以通过增加新的访问者类来实现，而无需修改现有元素类。
3. 集中相关操作： 将相关的操作集中在一个访问者类中，便于维护和管理。

* 缺点

1. 增加新元素较困难： 如果需要在元素结构中增加新的元素，需要修改所有访问者类，不符合开闭原则。
2. 破坏封装： 访问者模式会使得元素的内部细节暴露给访问者，可能破坏元素的封装性。
3. 违反依赖倒置原则： 具体元素类依赖于具体访问者类，违反了依赖倒置原则，使得系统较为僵硬。

# 附录

## 项目分工说明与成员贡献

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 学号 | 姓名 | 贡献 |
| 1 | 2051498 | 储岱泽 | 完成命令模式、装饰器模式、备忘录模式、空对象模式、适配器模式的代码和文档书写 |
| 2 | 2152826 | 夏尧民 | 完成享元模式、原型模式和观察者模式三种模式对应板块的代码和文档书写 |
| 3 | 2151929 | 王宜沣 | 完成责任链模式、延迟初始化模式、MVC模式的代码和文档书写 |
| 4 | 2151622 | 沙坚 | 项目简介撰写、ppt制作、答辩演讲、过滤器、代理模式等代码梳理检验 |
| 5 | 2051828 | 莫益萌 | 完成桥接模式，工厂方法模式，状态模式的代码和文档书写 |
| 6 | 2152494 | 徐嘉琪 | 完成中介者模式，过滤器模式，代理模式的代码和文档书写 |
| 7 | 2154296 | 傅佳恒 | 完成抽象工厂模式，外观模式，组合模式的代码和文档书写 |
| 8 | 2051973 | 韩嘉睿 | 完成观察者模式、策略模式，模版模式的代码和文档书写 |
| 9 | 2050780 | 焦骜 | 设计模式介绍部分PPT制作、答辩演讲、代码梳理检查 |
| 10 | 2152835 | 周文玥 | 完成DAO模式，雇工模式的代码和文档书写 |
| 11 | 2152402 | 段婷婷 | 完成简单工厂模式，迭代器模式和建造者模式的代码和文档书写。 |

## 非经典设计模式出处

* **雇工模式：**专(译)著:[1]谭勇德.设计模式就该这样学:基于经典框架源码和真实业务场景.电子工业出版社,2020.754-757
* **简单工厂模式：**连续出版物：[2]秦澎涛,王苏文. 简单工厂模式在数据访问层中的应用 [J]. 计算机工程与设计, 2009, 30 (07): 1799-1801.
* **空对象模式：**连续出版物：[3]吴清寿. Null Object模式研究 [J]. 吉林师范大学学报(自然科学版), 2013, 34 (01): 120-122.
* **MVC模式：**连续出版物：[4]任中方,张华,闫明松等. MVC模式研究的综述 [J]. 计算机应用研究, 2004, (10): 1-4+8.
* **延迟初始化模式：**连续出版物：[5]齐鑫. 责任链设计模式的改进 [J]. 计算机工程, 2010, 36 (10): 56-57+60.
* **过滤器模式：**专(译)著:[6]Deepak Alur,John Crupi,Dan Malks.J2EE核心模式(,刘天北,熊杰).机械工业出版社,2003.98-113
* **数据访问对象模式：**论文集：[7]欧阳宏基,解争龙,黄素萍等.一种基于DAO设计模式与Hibernate框架的数据持久化层模型[J].微计算机应用,2009,30(03):36-40.