实验十四 设计模块（三 ）

实验目的：

学习设计模式，能在项目设计中运用设计模式进行面向对象设计

实验内容：

1. 阅读下面设计模式资料（或查阅其它相关资料），结合项目的进程和开发历程，分析项目采用了那些设计模式

Design Patterns-Elements of Reusable Object-Oriented Software.pdf

The GoF Design Patterns Reference.pdf

[Design Patterns - Wikipedia](https://en.wikipedia.org/wiki/Design_Patterns)

2. 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 （保存到每个小组选定的协作开发平台上）

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。

1. **项目分析**
2. 单例模式（被用于数据库连接池管理类，确保全系统中只有一个数据库连接池。）

要素：

私有化构造函数：单例模式的类通常会将其构造函数私有化，以防止外部代码通过实例化来创建对象。

静态方法获取实例：类中会提供一个静态方法，用于获取该类的唯一实例。这个方法会检查实例是否已经存在，如果不存在，则创建实例并返回；如果已经存在，则直接返回已经存在的实例。

静态成员变量保存实例：类中会有一个静态成员变量来保存该类的唯一实例，保证在整个应用程序中只有一个实例存在

优点：

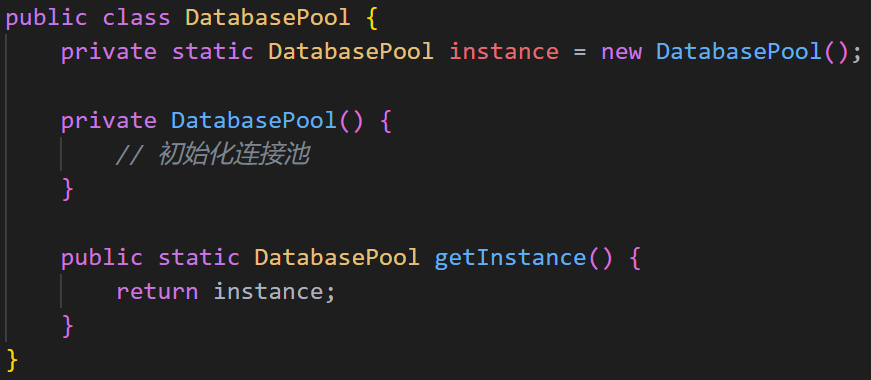
节省资源：由于单例模式只创建一个实例，因此可以节省系统资源，避免多次创建相同对象所带来的资源消耗。

全局访问：单例模式提供了一个全局访问点，方便在程序的任何地方都可以访问到该实例，简化了代码的调用流程。

然而，单例模式也有一些缺点：

可能造成性能瓶颈：由于单例模式只有一个实例，可能会造成在多线程环境下的性能瓶颈，需要考虑线程安全的实现方式。

隐藏依赖关系：单例模式会隐藏类之间的依赖关系，增加了代码的耦合性，使得代码难以扩展和维护。



1. 观察者模式（事件系统，用于通知用户新博客文章的发布，可以一对多的通知）

要素：

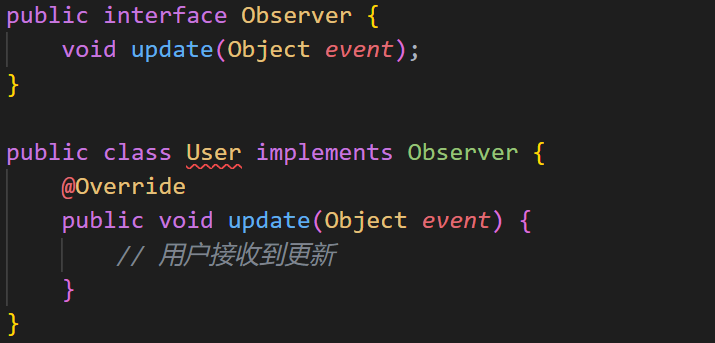
Subject（主题）：也称为可观察对象，负责维护一组观察者，并在状态发生变化时通知观察者。

Observer（观察者）：观察主题的对象，当主题状态发生变化时，观察者会接收到通知并进行相应的更新操作。

优点：

解耦性：主题和观察者之间是松耦合的，主题只需要知道观察者接口，而不需要知道具体的观察者实现。

扩展性：可以方便地新增观察者或者主题，使得系统更易于扩展。



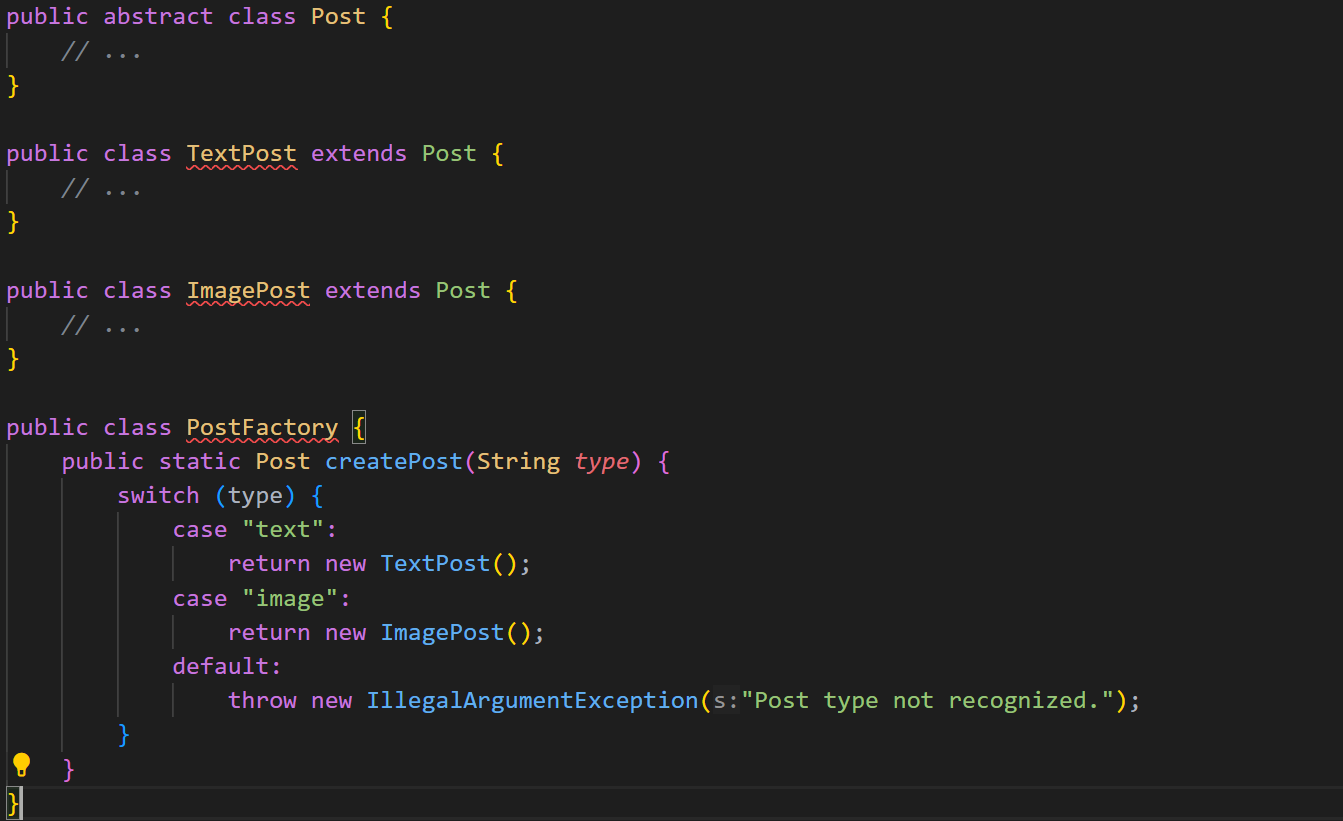
1. 工厂模式（工厂（Factory）模式被用于创建博客文章的实例，不同类型的博客文章类型有着共同的接口）

形式：

简单工厂模式（Simple Factory Pattern）：简单工厂模式是最基本的工厂模式，它通过一个工厂类来集中管理对象的创建。客户端只需要提供所需对象的类型给工厂类，工厂类根据类型来实例化对象并返回给客户端。

工厂方法模式（Factory Method Pattern）：工厂方法模式定义了一个创建对象的接口，但将具体的实例化延迟到子类中。每个具体的子类都可以实现工厂接口以提供自己特定的实例化逻辑，从而让客户端可以选择使用哪个子类来创建对象。

抽象工厂模式（Abstract Factory Pattern）：抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。通过抽象工厂模式，客户端可以创建一组对象而不必关心这些对象是如何被创建、组合以及表达的



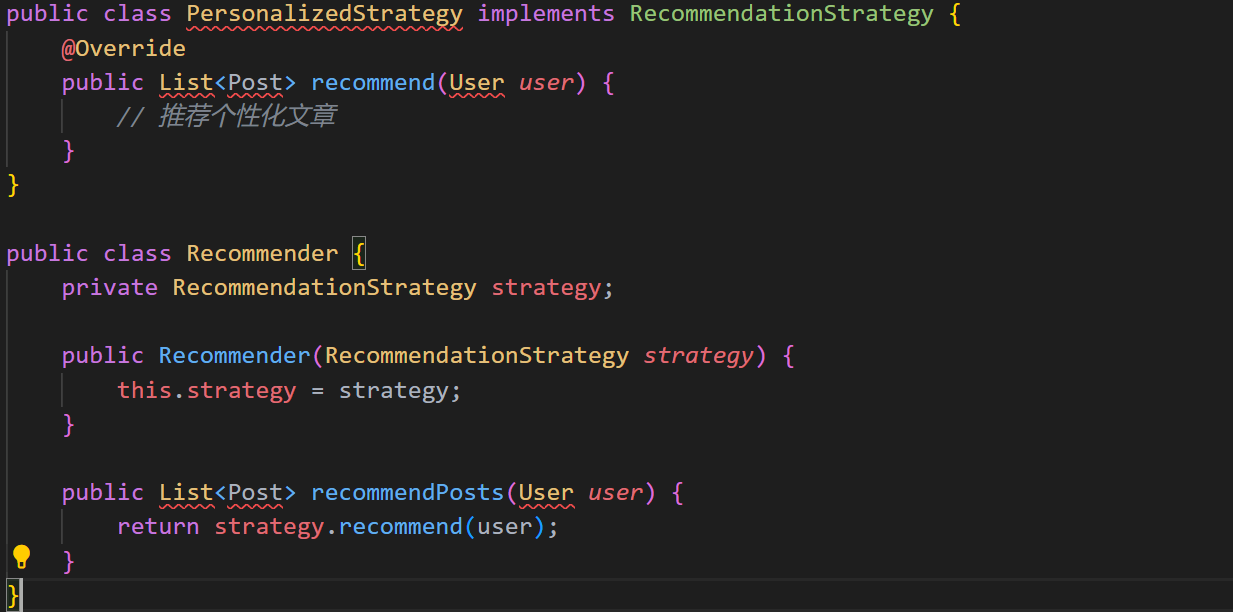
1. 策略模式（博客系统根据用户的偏好使用不同的文章推荐算法，各自算法封装独立）

元素：

上下文(Context)：上下文是客户端使用策略的地方，在上下文内部维护一个对策略对象的引用。上下文提供了一个接口供客户端调用，通过这个接口客户端可以选择并应用特定的策略。上下文负责将请求委派给策略对象，并可能在运行时动态改变所使用的策略。

策略(Strategy)：策略是一个接口或抽象类，它定义了一个算法族，这些算法可以被上下文使用。通常策略模式会把同一类算法封装到不同的策略类中，从而实现了算法的独立性和可替换性。

具体策略(Concrete Strategy)：具体策略是策略的具体实现，它实现了策略接口所定义的算法。



1. **设计模式**
2. 访问者模式

特点：

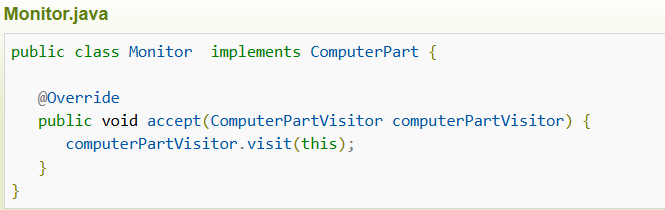
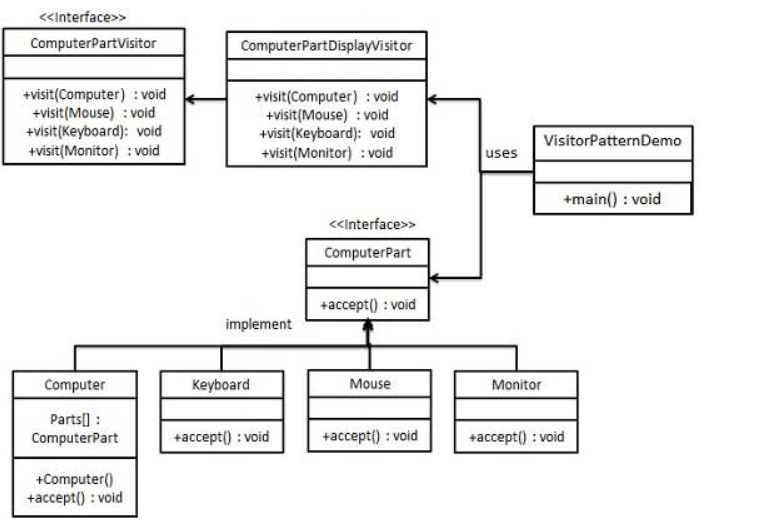
分离数据结构和数据操作：访问者模式通过将数据结构与数据操作进行分离，能够使得系统更加灵活和易于扩展。新增的操作可以通过添加新的访问者而不需要修改原有的数据结构。

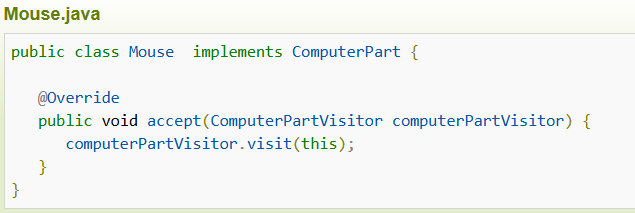
增加新的操作无需修改元素类：在访问者模式中，可以通过添加新的访问者来增加新的操作，而无需修改元素类，从而遵循了开闭原则。

对数据结构的访问逻辑集中化：访问者模式将对数据结构的访问逻辑集中在访问者对象中，使得数据结构本身不需要知道具体的访问方式，降低了数据结构的复杂度。

支持对数据结构的不同访问方式：访问者模式支持定义不同的访问者来实现对同一组数据结构的不同访问方式，从而提高了系统的灵活性。

适用于稳定的数据结构和频繁变化的操作：当数据结构相对稳定，但是需要经常新增新的操作时，访问者模式特别有用。







1. 适配器模式

特点：

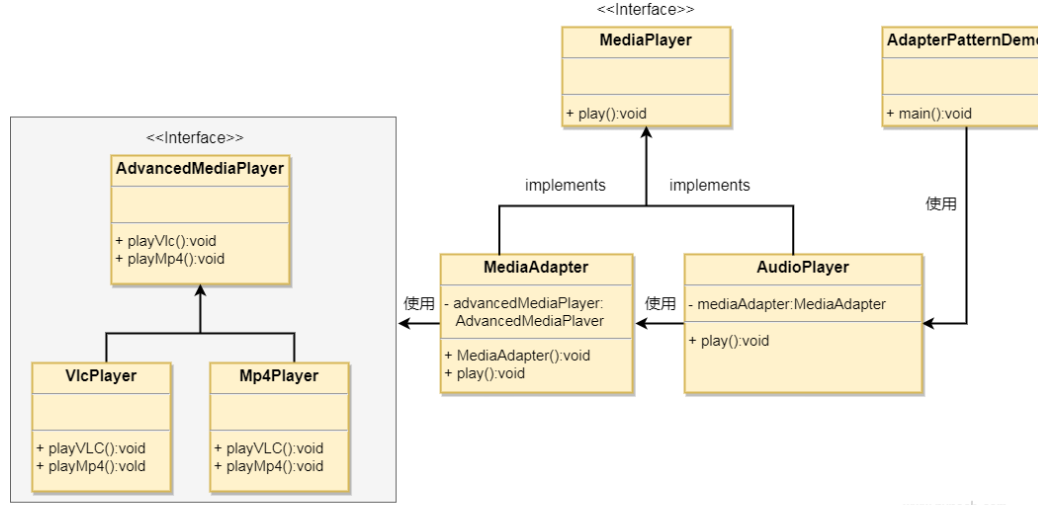
解决接口不匹配问题：适配器模式主要用于解决两个接口不兼容的情况，使得原本由于接口不匹配而无法在一起工作的类可以协同工作。

包括类适配器和对象适配器：适配器模式主要有两种形式，即类适配器模式和对象适配器模式。类适配器通过多重继承获得适配的能力，对象适配器通过组合被适配者对象获得适配的能力。

提高代码复用性：适配器模式可以让原本不兼容的接口协同工作，从而提高了代码的复用性。由于适配器模式可以重用现有的类，因此可以减少重复编码。

解耦客户端和被适配者：适配器模式可以将客户端和被适配者解耦，客户端只需要针对目标接口编程，而不需要了解具体的被适配者。

灵活性和可扩展性：适配器模式可以灵活地扩展和替换适配器，而不影响客户端的代码。这样可以更容易地应对未来的变化和需求。







1. 过滤器模式

特点：

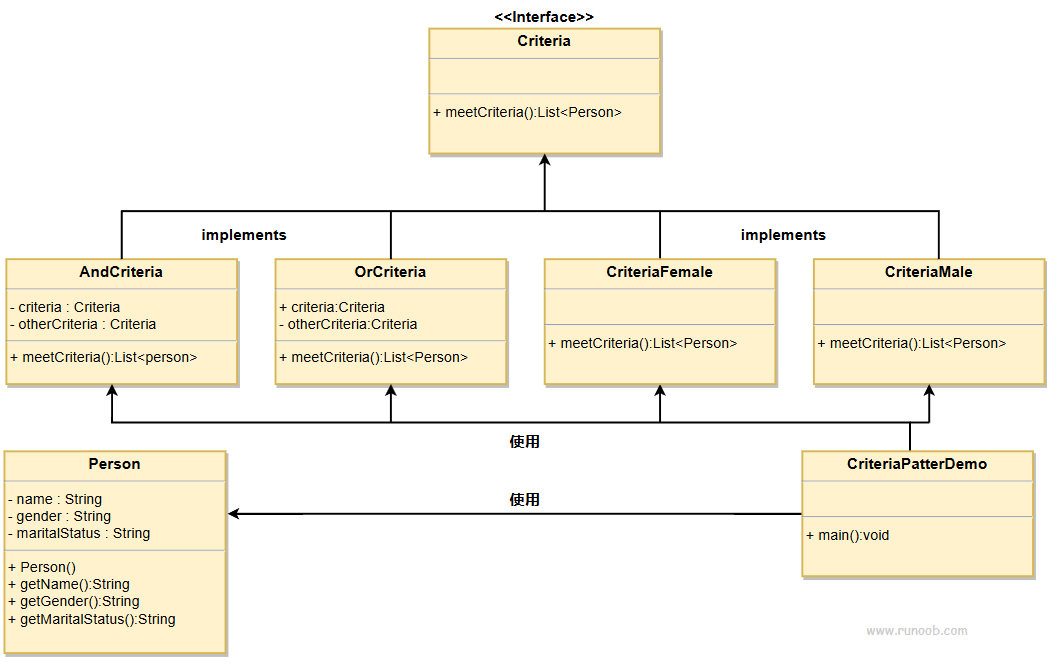
封装过滤条件：过滤器模式将过滤条件封装成独立的过滤器对象，使得客户端可以灵活地组合和重用过滤条件，而不需要修改原有的代码。

支持多种过滤条件：过滤器模式支持多种过滤条件的组合，可以根据需要同时应用多个过滤条件来筛选集合中的元素。

解耦过滤逻辑：过滤器模式将过滤逻辑与集合的实现进行了解耦，使得过滤条件的变化不会影响到集合的实现，同时也使得集合的实现可以独立于过滤条件的变化而变化。

提高代码的可读性和可维护性：通过使用过滤器模式，可以将过滤条件的实现细节封装在过滤器对象中，使得代码更加清晰、易于理解和维护。

灵活性和可扩展性：过滤器模式可以灵活地添加新的过滤条件，而不需要修改现有的代码，从而使得系统更易于扩展和维护。







1. 中介者模式

特点：

解耦对象之间的关系：中介者模式通过引入一个中介者对象，可以将对象之间的直接通信转变为间接通信，从而解耦了对象之间的关系。对象只需要和中介者进行通信，而无需直接与其他对象进行交互。

集中控制交互逻辑：中介者模式将对象之间的交互逻辑集中到中介者对象中，使得系统更加清晰和易于理解。中介者对象负责协调对象之间的交互，从而降低了系统的复杂度。

促进重用和扩展：由于中介者模式将交互逻辑封装在中介者对象中，因此可以更容易地重用和扩展这些逻辑。新增新的对象或调整现有对象之间的交互关系也更加容易。

降低维护成本：中介者模式可以使系统更加灵活和可维护，当系统中的对象之间的交互关系发生变化时，只需要修改中介者对象而不需要修改大量的对象之间的交互逻辑。

适用于多对多的交互场景：中介者模式适用于多个对象之间存在复杂的交互关系的场景，尤其适用于多对多的交互情况。

