设计模式总纲

设计模式7大设计原则

设计原则	简单理解	具体解释
单一职责原则	类的职责单一	一个类应该只负责一项职责
接口隔离原则	只依赖自己有用的接口方法	一个类对另一个类的依赖应该建立在最小的接口之上
里氏替换原则	子类尽量不要重写父类的方法	所有引用基类的地方必须能够透明地使用其子类的对象
依赖倒转原则	不依赖实现类，依赖接口	抽象不应该依赖细节，细节应该依赖抽象
开闭原则	对扩展开放，对修改关闭	增加功能的实现尽量通过扩展来实现，而不是修改
迪米特法则	最少知道，细节不需要了解	一个对象应该对其他对象保持最少的了解
合成复用原则	用组合或者聚合，而非继承	尽量使用合成或者聚合的方式，而不是继承

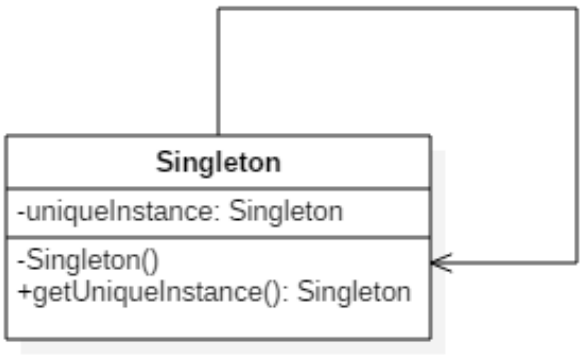
创建型设计模式

单例模式

- 解决问题

  保证每次使用的对象都是同一个对象

- 类图



- 代码实现

应用场景	实现方式	原理	优点	缺点	备注
• 初始化时就需创建单例 • 单例对象 要求初始化速度快 & 占用内存小	1. 饿汉式	依赖JVM类加载机制，保证单例只被创建1次	• 线程安全 (即多线程下适用，因为 JVM 只会加载1次单例类) • 初始化速度快 • 占用内存小	单例创建时机不可控制	饿汉式 与 懒汉式最大区别 = 单例创建时机 • 饿汉式：单例创建时机不可控，即类加载时 自动创建 单例 • 懒汉式：单例创建时机可控，即有需要时，才 手动创建 单例
	2. 枚举类型	• 枚举类型 = 不可被继承的类（final） • 每个枚举元素 = 类静态常量 = 依赖JVM类加载机制，保证单例只被创建1次 • 枚举元素 都通过静态代码块来进行初始化 • 构造方法 访问权限 默认 = 私有（private） • 大部分方法都是final	• 线程安全 • 自由序列化 • 实现更加简单、简洁		
• 按需、延迟创建单例 • 单例初始化的操作耗时长 & 应用要求启动速度快 • 单例的占用内存比较大	3. 懒汉式 (基础实现)	1. 类加载时，先不自动创建单例 (即，将单例的引用先赋值为 null) 2. 需要时才手动 创建 单例	• 按需加载单例 • 节约资源	线程不安全 (即多线程下不适用)	
	4. 同步锁 (懒汉式的改进)	使用同步锁 synchronized 锁住 创建单例的方法 (防止多个线程同时调用，从而避免造成单例被多次创建)	• 线程安全	造成过多的同步开销 (每次访问都要进行线程同步，加锁 = 耗时、耗能)	
	5. 双重校验锁 (懒汉式的改进)	• 校验锁1：若单例已创建，则直接返回已创建的单例，无需再执行加锁操作 • 校验锁2：防止多次创建单例问题	• 线程安全 • 节省资源（不需过多的同步开销）	实现复杂 (多种判断、易出错)	
	6. 静态内部类	• 按需加载：在静态内部类里创建单例，在装载该内部类时才会去创建单例 • 线程安全：类是由 JVM加载，而JVM只会加载1遍，保证只有1个单例	• 线程安全 • 节省资源（不需过多的同步开销） • 实现简单		

- 饿汉式（静态常量、静态代码块）

```
class Student{
    private final static Student instance = new Student();
    private Student(){

    }
    // Student.getInstance() 即可获取单例对象
    public static Student getInstance(){
        return instance;
    }
}
```

  - 优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。
  - 缺点：在类装载的时候就完成实例化，没有达到`Lazy Loading`的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费
  - 这种方式基于`classloder`机制避免了多线程的同步问题，不过，`instance`在类装载时就实例化，在单例模式中大多数都是调用`getInstance`方法，但是导致类装载的原因有很多种，因此不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化`instance`就没有达到`lazy loading`的效果
  - 总结：单例模式可用，线程安全，但可能造成内存浪费，在确定会用到的情况下是非常不错的。
- 懒汉式（线程不安全）

```
class Student{
    private static Student instance;
    private Student(){};
    // 提供一个公有方法，使用到的时候创建对象
    public static Student getInstance(){
        if (instance == null){
            instance = new Student();
        }
        return instance;
    }
}
```

- 起到了Lazy Loading 的效果，但是只能在单线程下使用。
- 如果在多线程下，一个线程进入了`if (instance == null)`判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式
- 结论：在实际开发中，不要使用这种方式

- 懒汉式（线程安全、同步方法）

```
class Student{
    private static Student instance;
    private Student(){};
    public static synchronized Student getInstance(){
        if (instance == null){
            instance =  new Student();
        }
        return instance;
    }
}
```

- 解决了线程安全问题
- 效率太低了，每个线程在想获得类的实例时候，执行`getInstance()`方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接`return`就行了。方法进行同步效率太低
- 结论：在实际开发中，不推荐使用这种方式

- 懒汉式（线程安全，同步代码块）

```
class Student{
    private static Student instance;
    private Student(){}
    public static Student getInstance(){
        if (instance == null){
            synchronized (Student.class) {
                // 这个地方加上的同步方法，只能保证在同一个时间内创建出的是一个对象
                // 如果两个线程都是null同时进入，加锁保证了先拿到锁的线程创建
instance
                // 后拿到锁的线程接下来再创建一个新的对象，这样并不是单例
                instance = new Student();
            }
        }
        return instance;
    }
}

```

- 不推荐使用，线程安全，但本质上这实现的不是单例模式

- 双重检查【推荐】

```
class Student {
```

```
    // volatile相当于轻量级的synchronized，告诉每个线程这个对象是不稳定的
    // 每次需要的时候要从内存中重新读，保证共享变量的可见性
    // 每个线程都有自己的高速缓冲区，该关键字保证每次取的时候都是最新的数据
    private static volatile Student instance;
    private Student(){};
    public static Student getInstance(){
        if (instance == null){
            synchronized (Student.class){
                if (instance == null){
                    instance = new Student();
                }
            }
        }
        return instance;
    }
}
```

- Double-Check概念是多线程开发中常使用到的，如代码中所示，我们进行了两次`if (instance == null)`检查，这样就可以保证线程安全了。
- 这样，实例化代码只用执行一次，后面再次访问时，判断`if (instance == null)`，直接`return`实例化对象，也避免的反复进行方法同步
- 线程安全；延迟加载；效率较高
- 结论：在实际开发中，推荐使用这种单例设计模式

- 静态内部类

```
class Student{
    private Student(){};
    private static class InnerStudent{
        private static final Student INSTANCE = new Student();
    }
    public static Student getInstance(){
        return InnerStudent.INSTANCE;
    }
}
```

- 这种方式采用了类装载的机制来保证初始化实例时只有一个线程。
- 静态内部类方式在`Student`类被装载时并不会立即实例化，而是在需要实例化时，调用`getInstance`方法，才会装载`Student`类，从而完成`Student`的实例化。
- 类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。
- 避免了线程不安全，利用静态内部类特点实现延迟加载，效率高
- 推荐使用

- 枚举

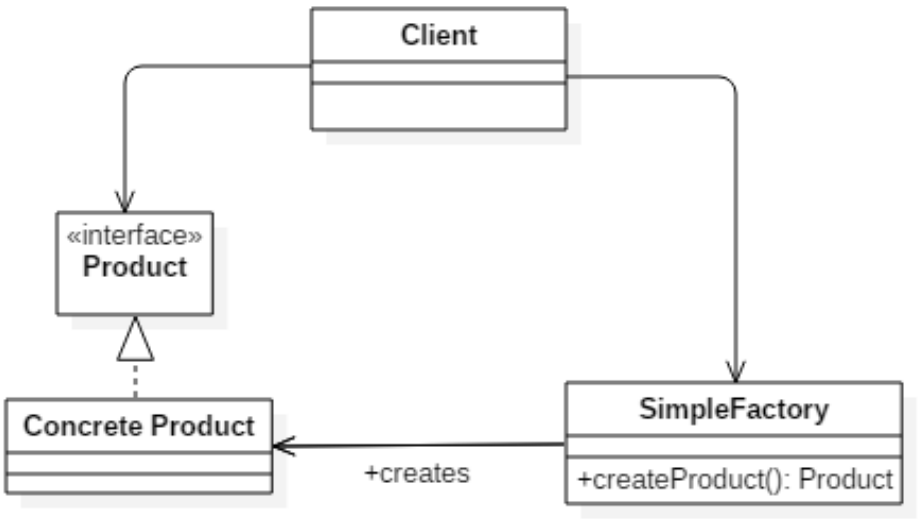
```java
enum Student{
    Instance;
    public void sayOK(){
        System.out.println("ok");
    }
}
```

    - 这借助JDK1.5 中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。
    - 结论：推荐使用
- 源码应用

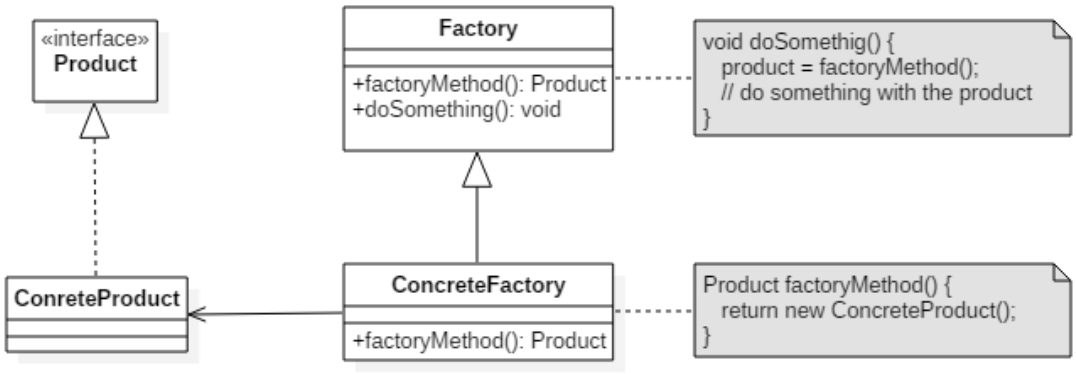
工厂模式

- 解决问题

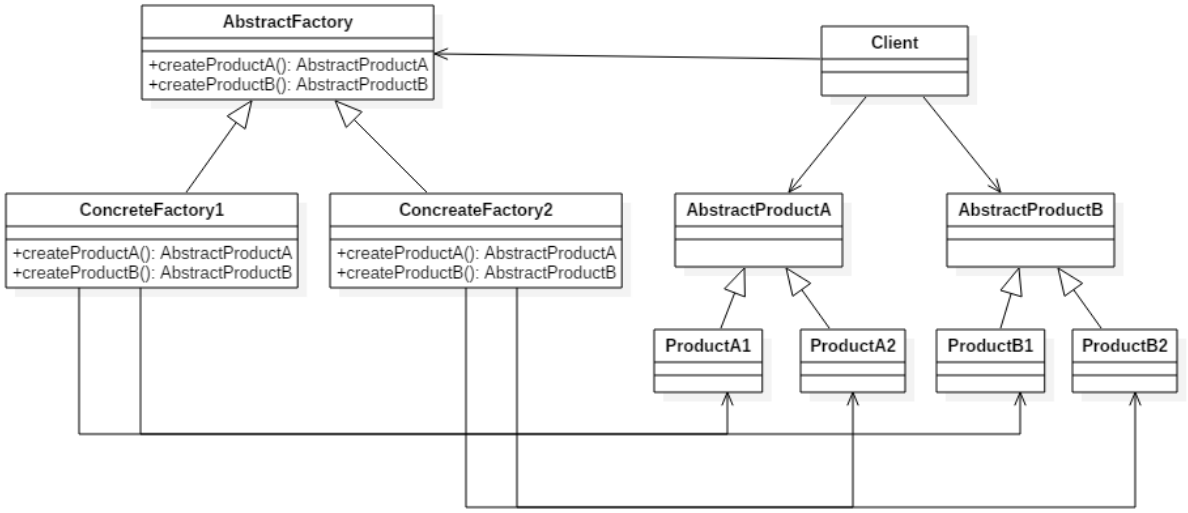
  批量化生产不同类型的对象
- 类图
    - 简单工厂



    - 工厂方法



- 抽象工厂



- 源码实现

建造者模式

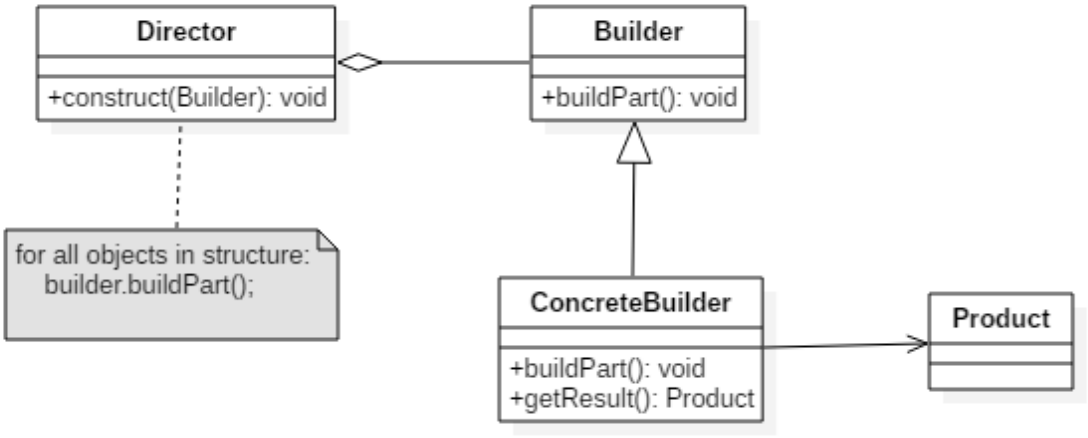
- 解决问题

  盖房子，地基、墙壁、屋顶组合才能盖出房子

  出一个抽象类，把流程定下来，具体怎么实现地基和墙壁屋顶，看子类的实现

  有三个角色，指挥者、具体建造者、抽象建造者

- 类图



- 源码实现
  - JDK中StringBuilder就是建造者模式

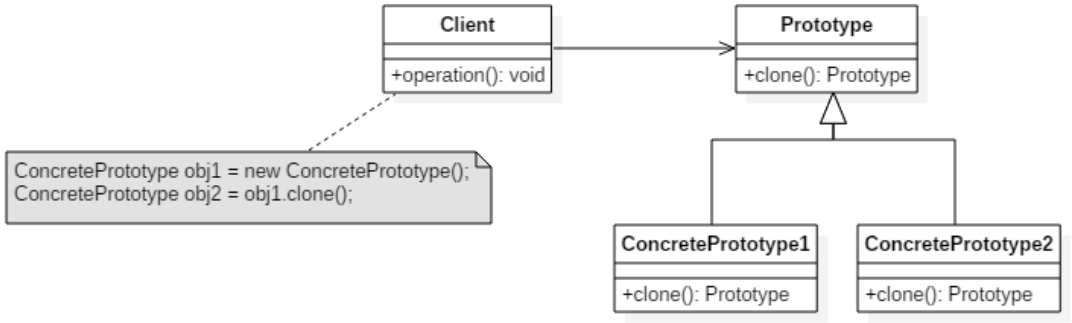
> - `StringBuilder`继承了`AbstractStringBuilder`
> - `AbstractStringBuilder`已经可以认为是建造者，只是不能实例化
> - `AbstractStringBuilder`实现了`Appendable`接口
> - `Appendable`接口，定义了多个抽象方法，实质上是抽象建造者
> - `StringBuilder`即是指挥者，也是具体的建造者
> - 总结：
>   - 具体的建造者；`StringBuilder` && `AbstractStringBuilder`
>   - 抽象建造者：`Appendable`接口
>   - 指挥者：`StringBuilder`

原型模式

- 解决问题

  克隆羊问题

- 类图



- 源码应用
  - JDK中的`clone()`方法

结构型设计模式

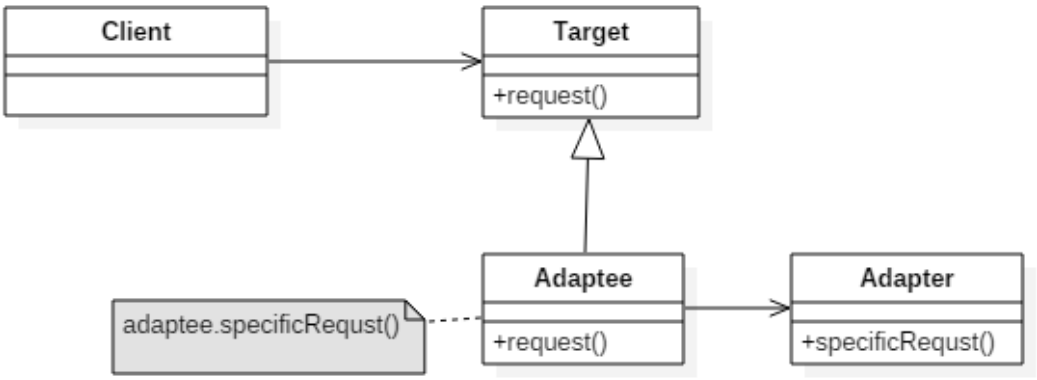
适配器模式

- 解决问题

  亡羊补牢模式，由于调用方和已经存在的两者之间不匹配，需要找一个翻译

- 类图

  这里`Adaptee`是适配器



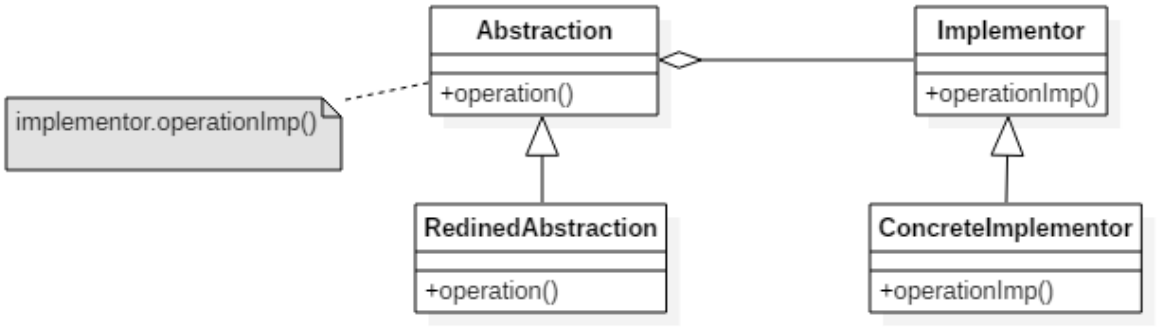
- 源码实现
  - SpringMVC中的具体实现就是一种适配器模式，`HandlerAdapter`

桥接模式

- 解决问题

  将抽象和行为实现分离开来，就像实际中`@Autowired`注入的是接口，而不是具体的实现类

- 类图



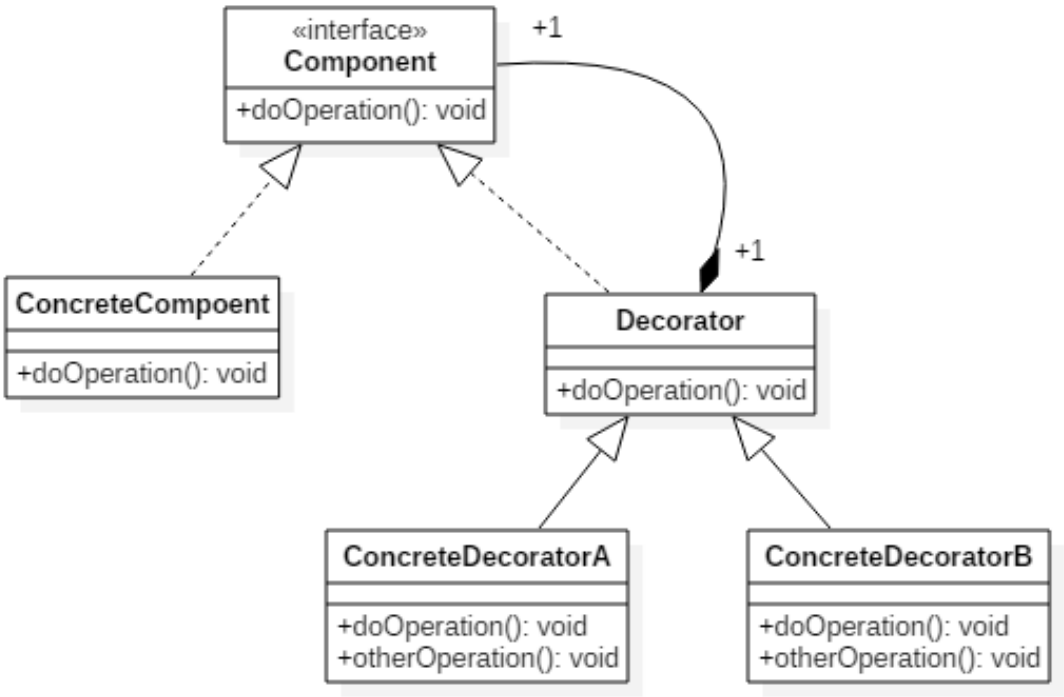
- 源码实现

装饰者模式

- 解决问题

  俗称套娃，A对象中有一个属性A，A的构造函数是传入A并返回包装好了的A+，A+继续可以作为A类的构造参数产生A++

- 类图



- 源码实现

组合模式

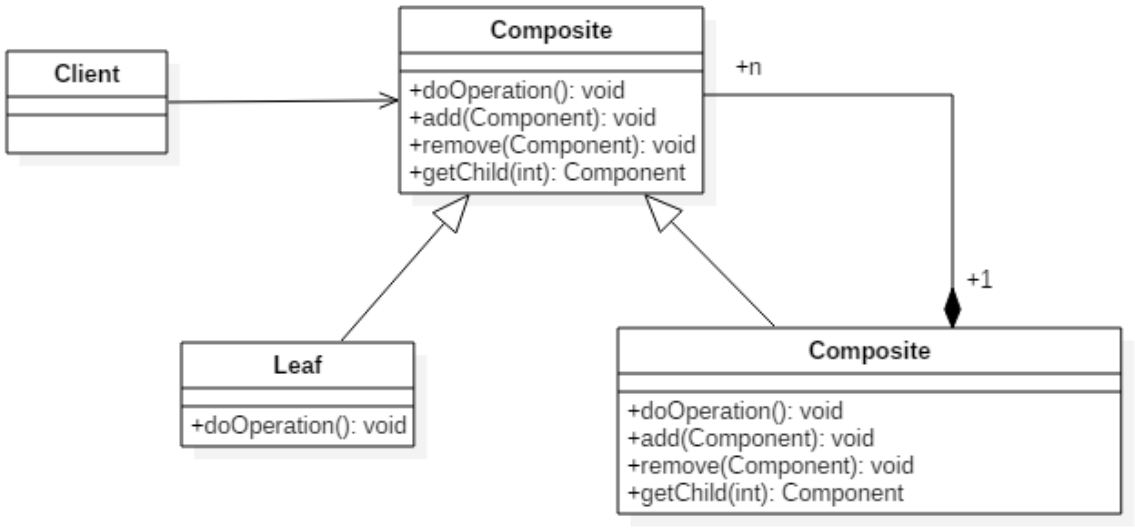
- 解决问题

  学校下面有学院，学院下面有班级

  想通过调用学校的`print()`方法打印出全体班级名单

  学校类里面放一个学院数组、每个学院里面放一个班级数组

- 类图



- 源码实现

外观模式

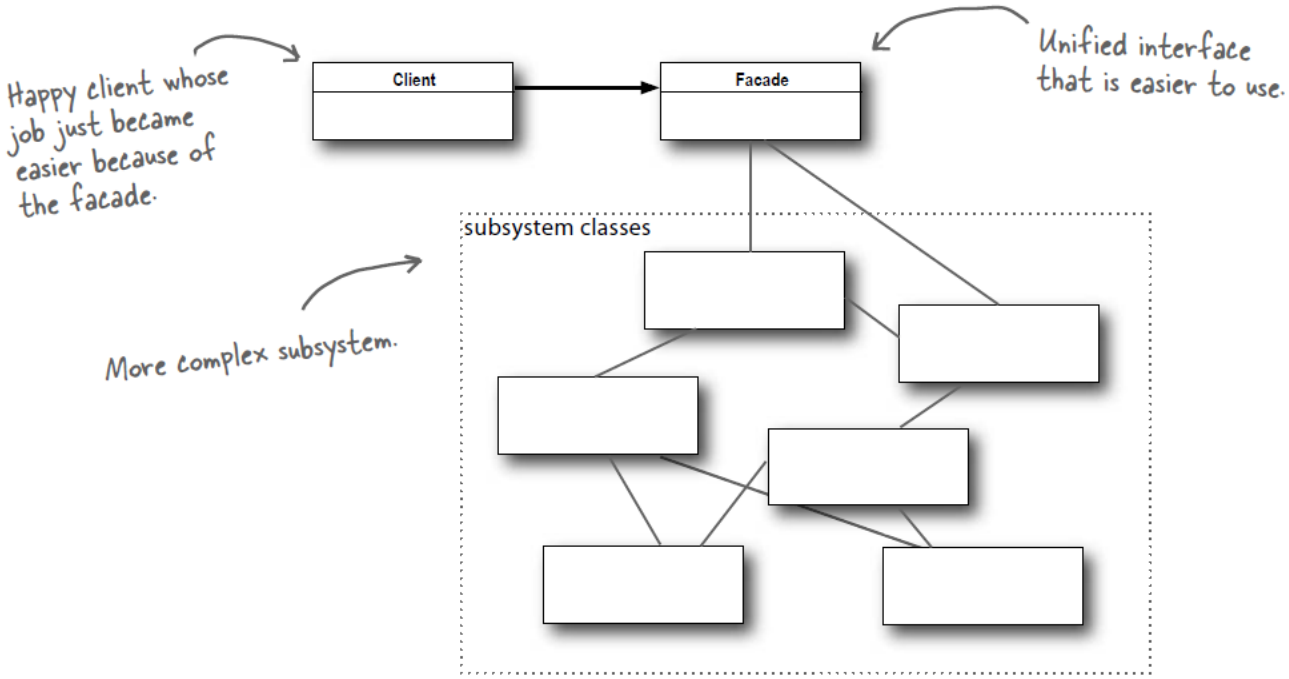
- 解决问题

  影院有各种设备，创建一个外观类统一管理，该外观类注入各种设备属性，实现批量管理

  将各个小部件封装到一个外观中

  【最小知道原则，里面组件不管，我只管这个外观有哪些方法】

- 类图



- 源码实现

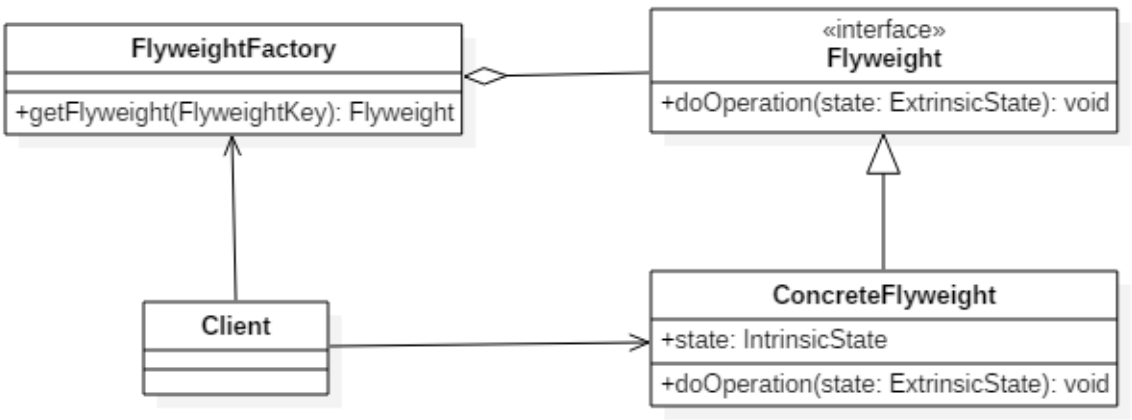
享元模式

- 解决问题

  有一些对象具有共性，就可以抽象出来供大家一起使用；有一些完全是独立的个性，就可以用子类的形式分别封装

  类似于棋盘上的每个棋子，基本属性黑色或白色是可以共享的属性（内部状态，可共享的），但是坐标位置是不同的，所以颜色可以共享，位置信息单独实现（外部状态，不可共享）

- 类图
  - `IntrinsicState`：内部状态，享元对象内部状态
  - `ExtrinsicState`：外部状态，每个享元对象的外部状态不同



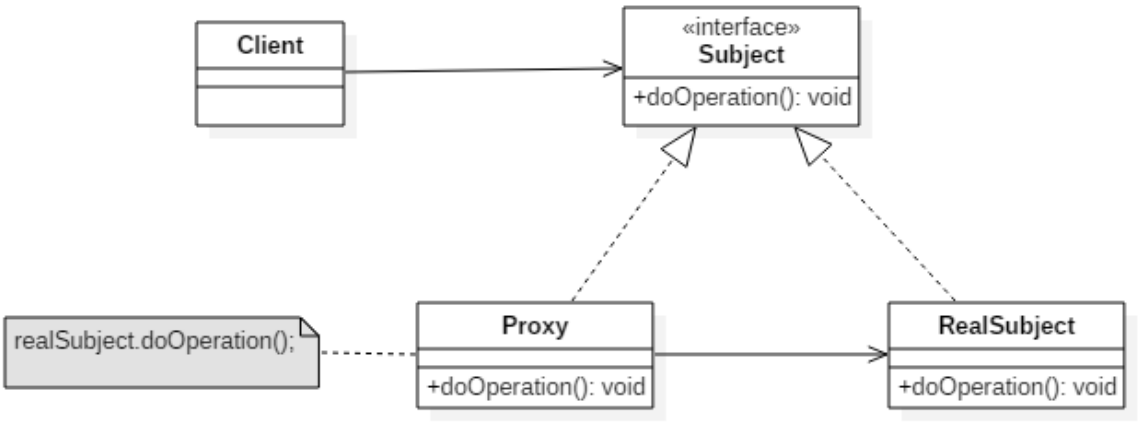
- 源码实现
  - `Integer.valueOf()`中缓冲池的实现，类似

代理模式

- 解决问题

  调用代理就行了，不用管具体的实现，或者是由于目前的实现不够强大，需要使用新的代理对象进行调整

- 类图



- 源码实现
  - JDK中反射包下的所有方法

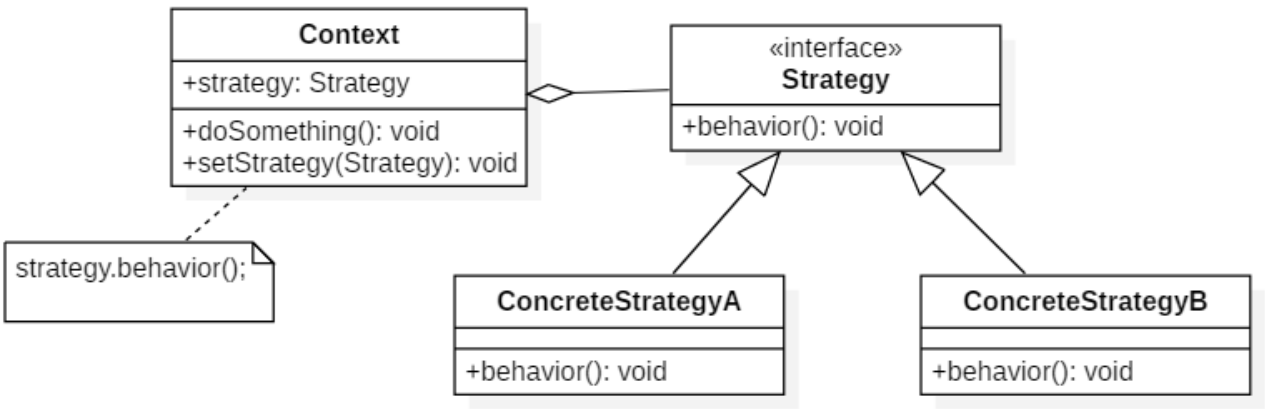
行为型设计模式

策略模式

- 解决问题

  将功能独立出来变成一些功能聚簇，根据每个对象的实际情况分别适配

- 类图



- 源码实现

实习中优惠券匹配规则逻辑

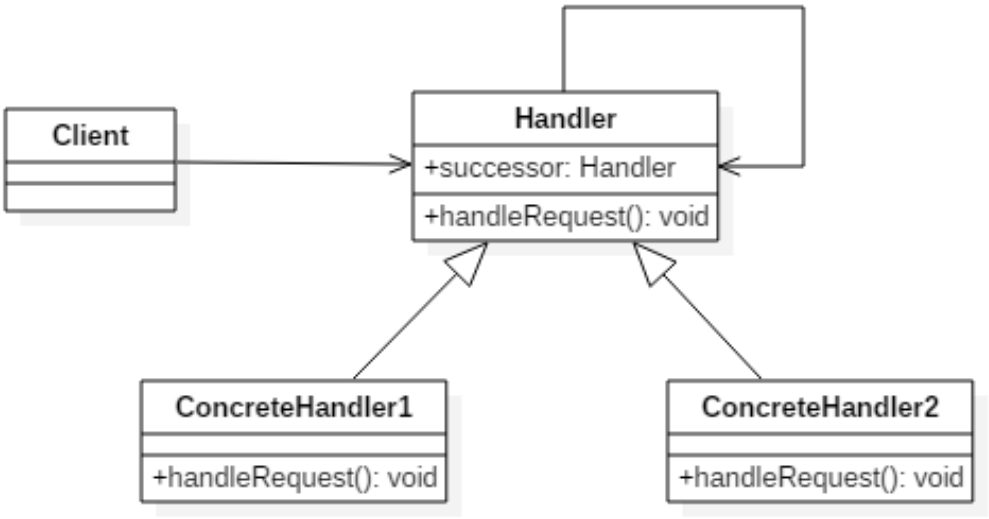
责任链模式

- 解决问题

  抽象父类，是各种的处理器，有个属性，就是对应的下一级处理器

  每个实现子类将自己的处理方法实现，并完善下一处理器是什么

- 类图



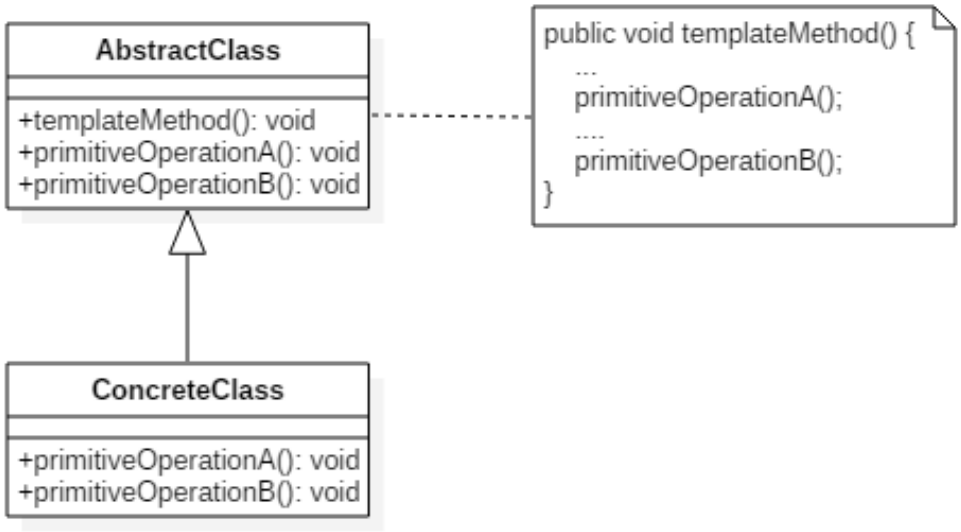
- 源码实现
  - `javax`的`doFilter()`

模版方法模式

- 解决问题

  在抽象类中定义基本流程，由子类重写各个小步骤

- 类图



- 源码实现

命令模式

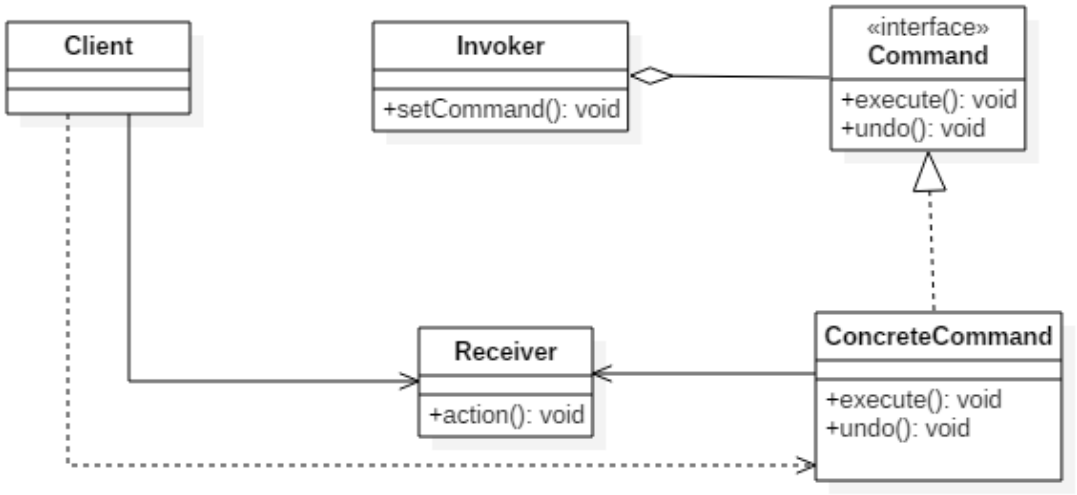
- 解决问题

  将军负责发送命令，对于将军来说并不关系具体是哪一个士兵正在执行任务

  将军只看命令接口，命令接口的实现类持有具体实现对象

  遥控器案例，遥控器有多个按钮，下面具体的实现则是早就安排好的，用户只关心我按下哪一个按键，会产生什么效果

- 类图



- 源码实现

访问者模式

- 解决问题

  解决了数据结构和操作的耦合性问题

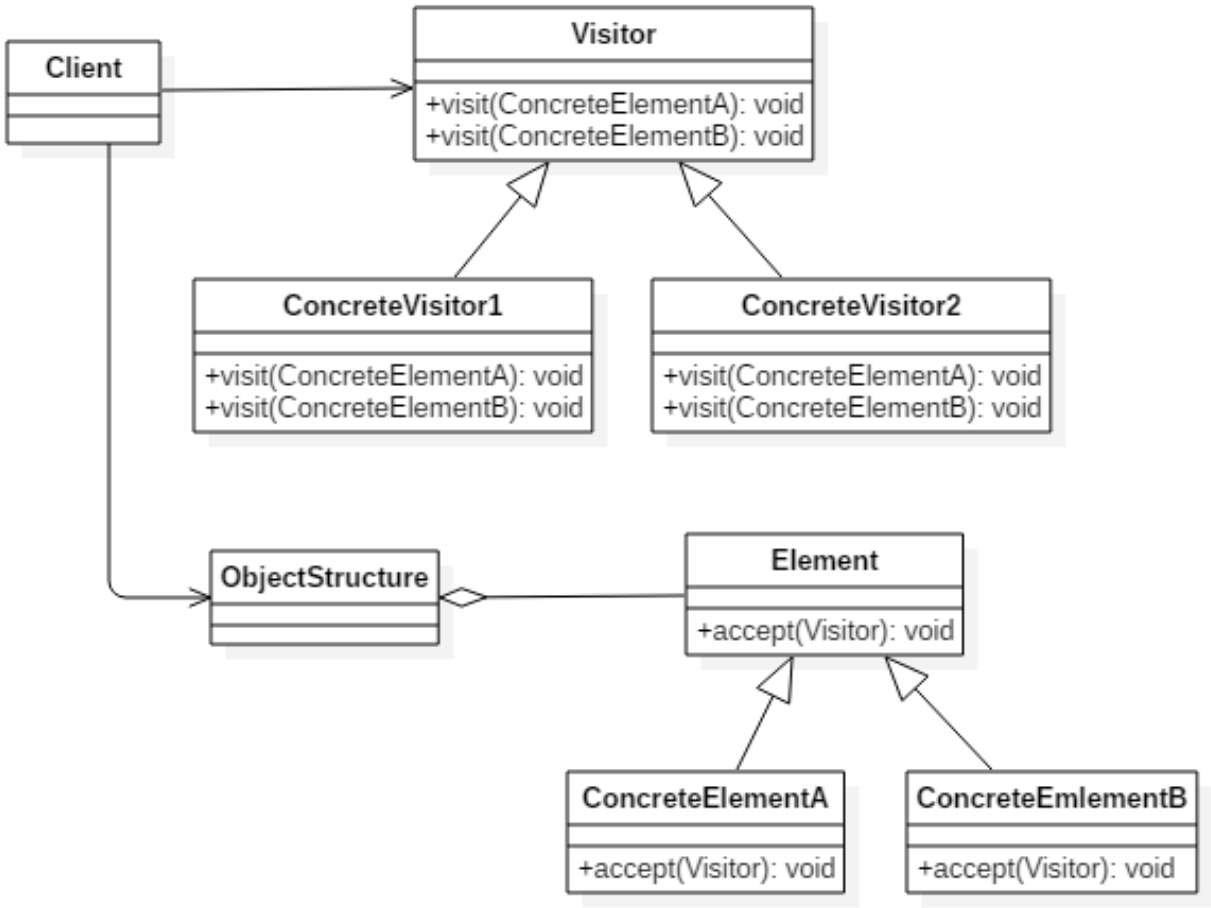
  访问者（数据结构）需要调用一个方法（操作），该方法根据访问者的类型找到对应的实现类，告诉访问者应该使用自己的哪一个方法

  Visitor: 访问者，为每一个 ConcreteElement 声明一个 visit 操作

  ConcreteVisitor: 具体访问者，存储遍历过程中的累计结果

  ObjectStructure: 对象结构，可以是组合结构，或者是一个集合。

- 类图



- 源码实现

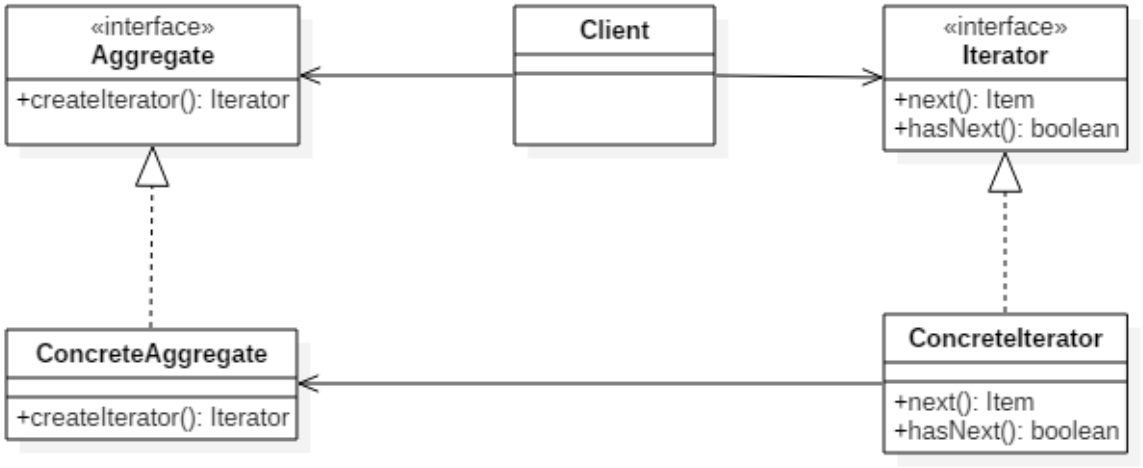
迭代器模式

- 解决问题

  用于遍历，一般来说迭代器模式需要实现三个接口、HasNext、next、remove

  封装不同底层的实现原理，比如链表实现或者数组实现的，统一使用迭代器就可以遍历

- 类图



- 源码实现

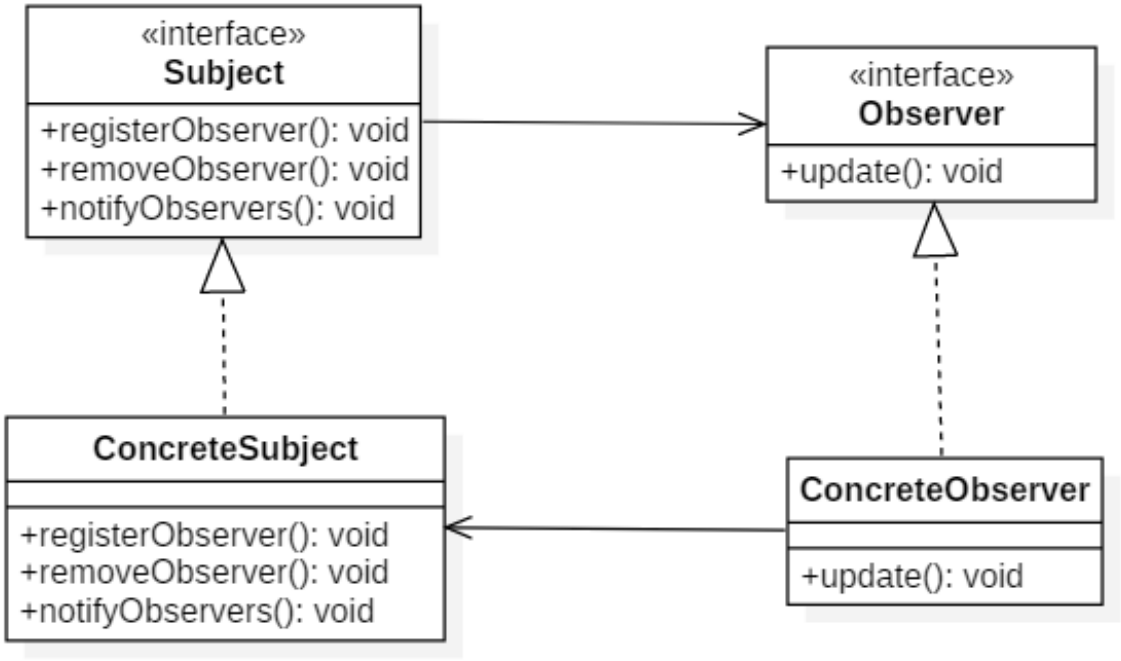
观察者模式

- 解决问题

  观察者模式：对象之间多对一依赖的一种设计方案，被依赖的对象为Subject，依赖的对象为Observer

  一般需要三个方法，注册、移除、通知观察者

- 类图



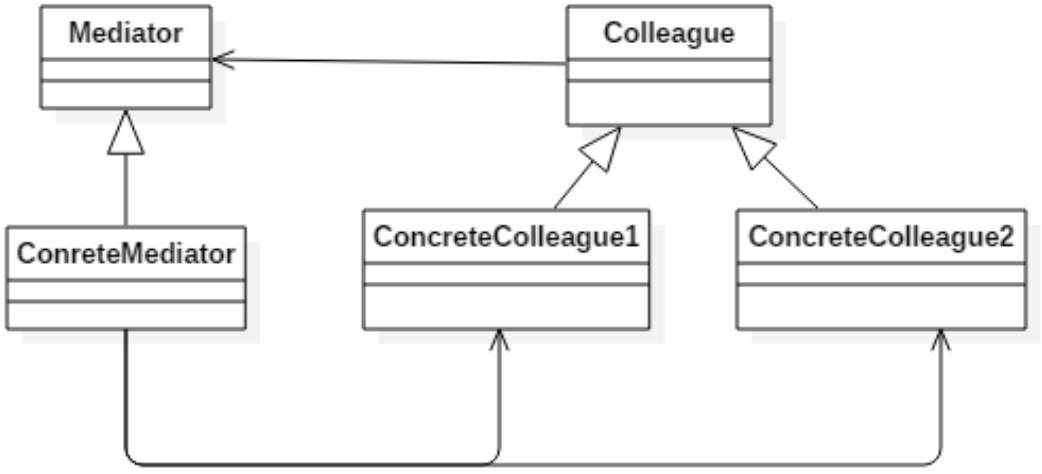
- 源码实现

中介者模式

- 解决问题

  由中介负责复杂的映射关系管理

- 类图



- 源码实现
  - MVC模型就是一个例子，有C控制器作为Model模型和View视图之间的中介者

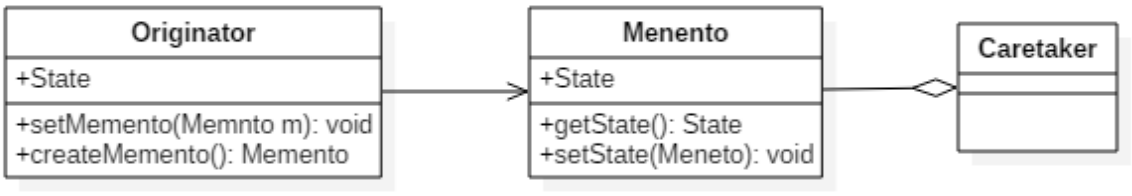
备忘录模式

- 解决问题

  数据备份，用的很少

  生成当前状态、回复之前的状态

- 类图



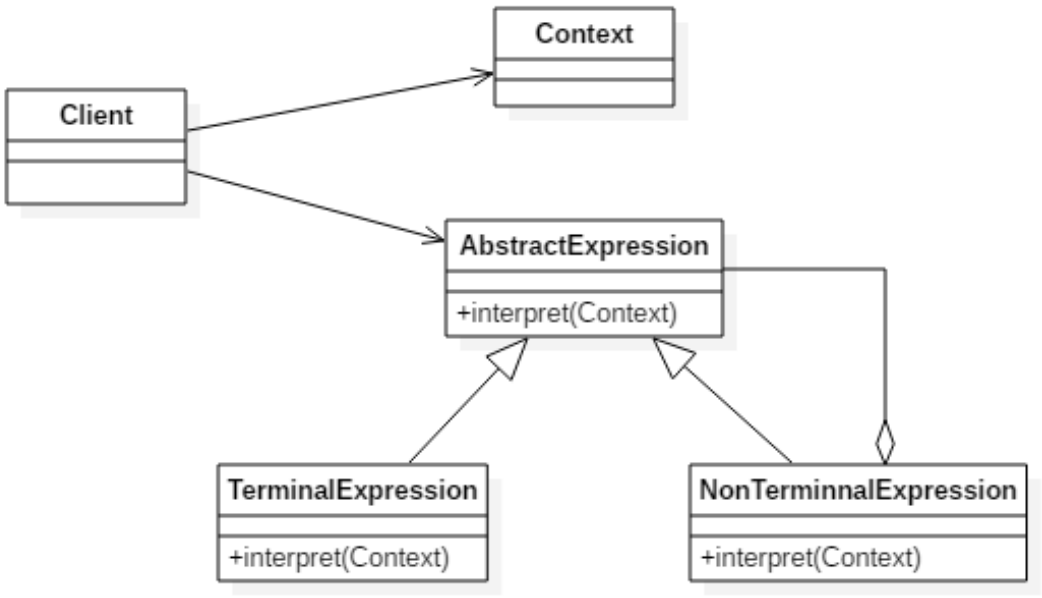
- 源码实现

解释器模式

- 解解决问题

  类似语法分析等，应用极少

- 类图



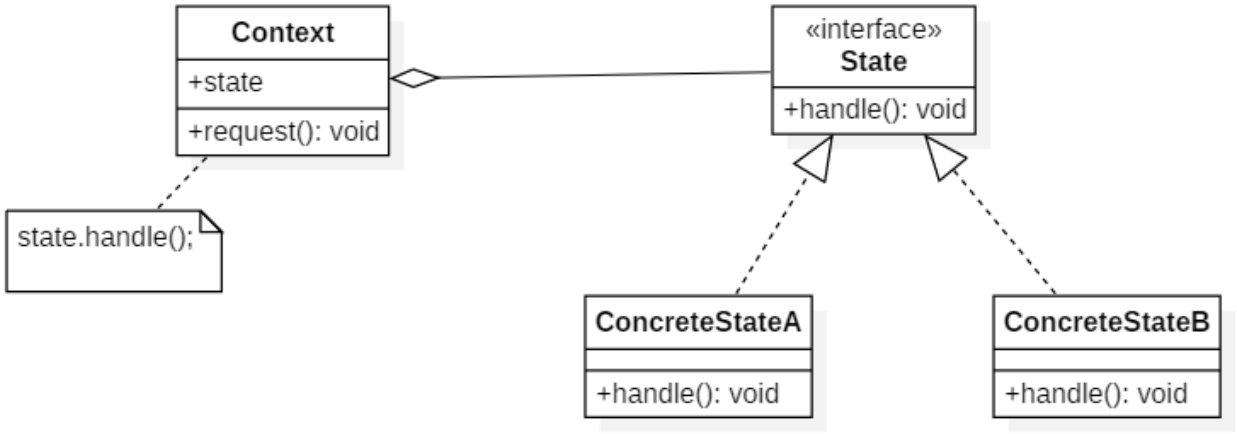
- 源码实现

状态模式

- 解决问题

  类似自动状态机，状态之间的转换。用的很少

- 类图



- 源码实现