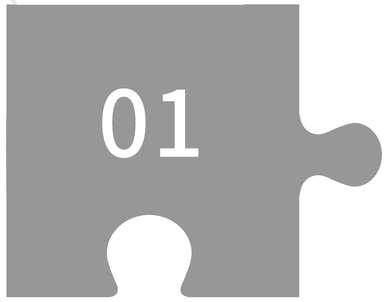
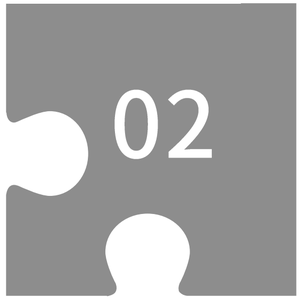
设计模式（二）

框架源码专题

课程目录



设计思想



设计原则

设计模式

总结

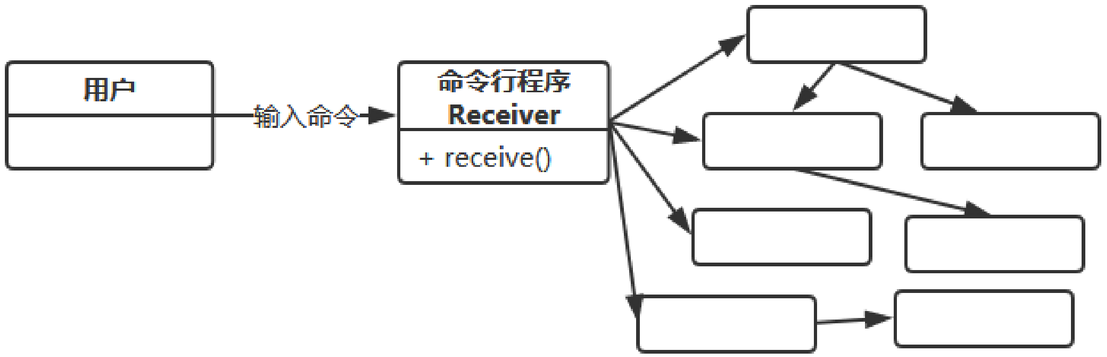
9 命令模式

- 示例：

请为你的系统设计一个命令行界面，用户可输入命令来执行某项功能。

系统的功能会不断添加，命令也会不断增加。

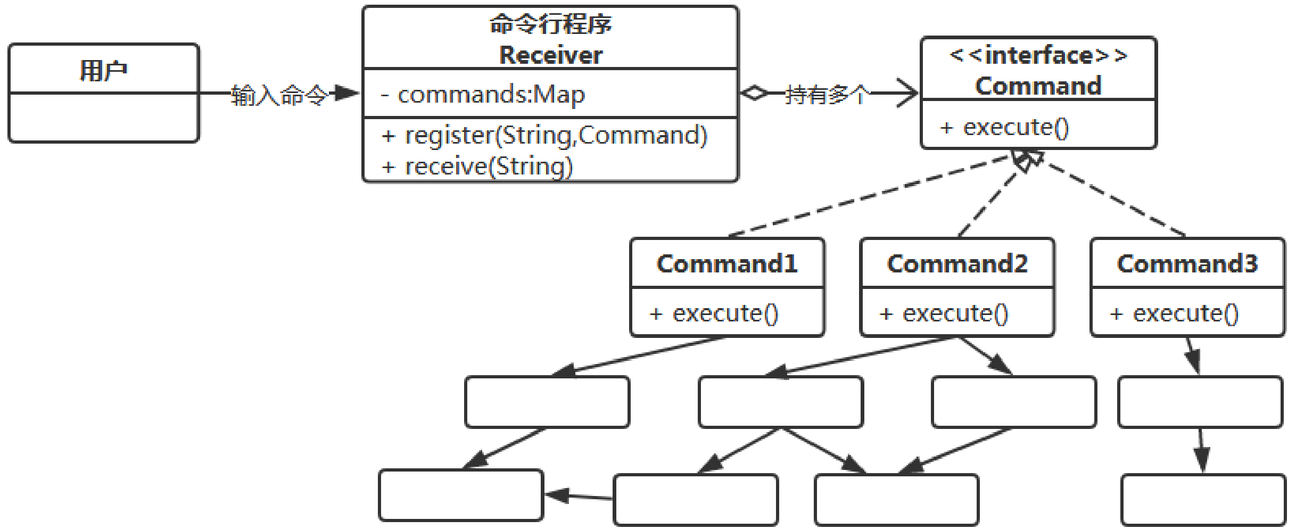
如何将一项一项的功能加入到这个命令行界面？



如何让我们的命令行程序写好后，不因功能的添加而修改，又可灵活加入命令、功能。请为此做设计。

```
public class Receiver{
    public void receive(String command) {
        switch (command) {
        case "command-1":
            // 执行功能1
            ……
            break;
        case "command-2":
            // 执行功能2
            ……
            break;
        case "command-3":
            // 执行功能3
            ……
            break;
        ……
        }
        System.out.println("不支持此命令" + command);
    }
}
```

9 命令模式-类图




```
public class Receiver{
    private Map<String,Command> commands;

    public void register(String strComm,Command command){
        commands.put(strComm,command);
    }

    public void receive(String command) {
        Command commandObj = commands.get(command);
        if(commandObj != null){
            commandObj.execute();
            return;
        }
        System.out.println("不支持此命令" + command);
    }
}
```

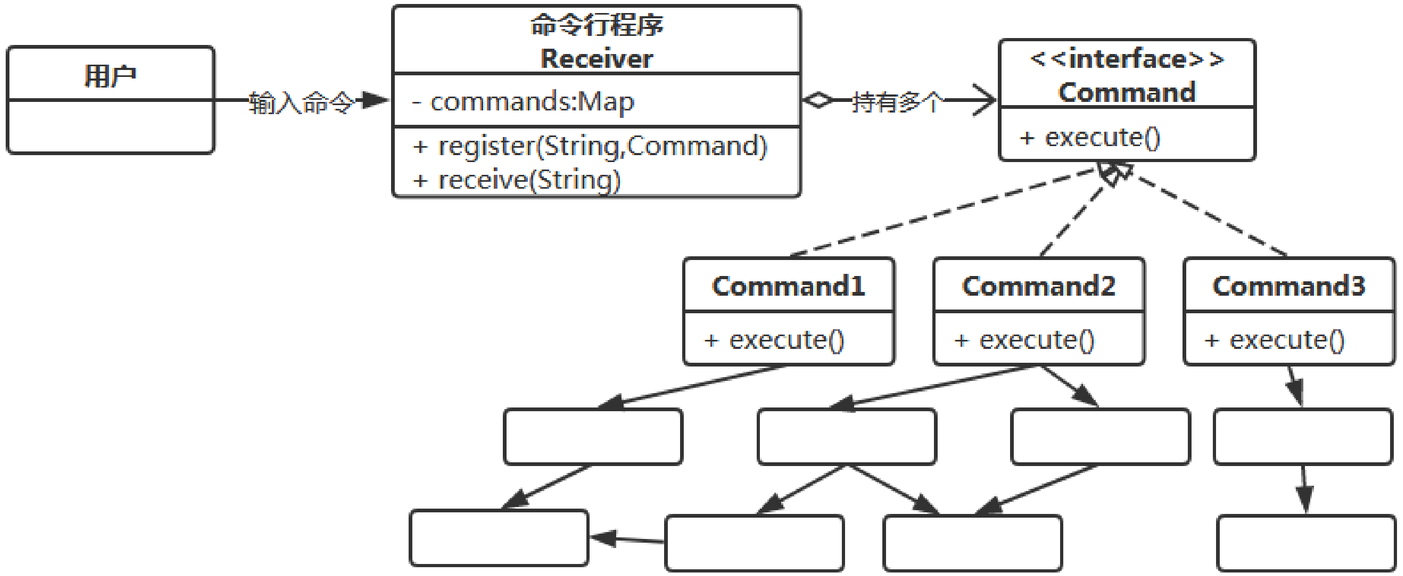
- **命令模式：**

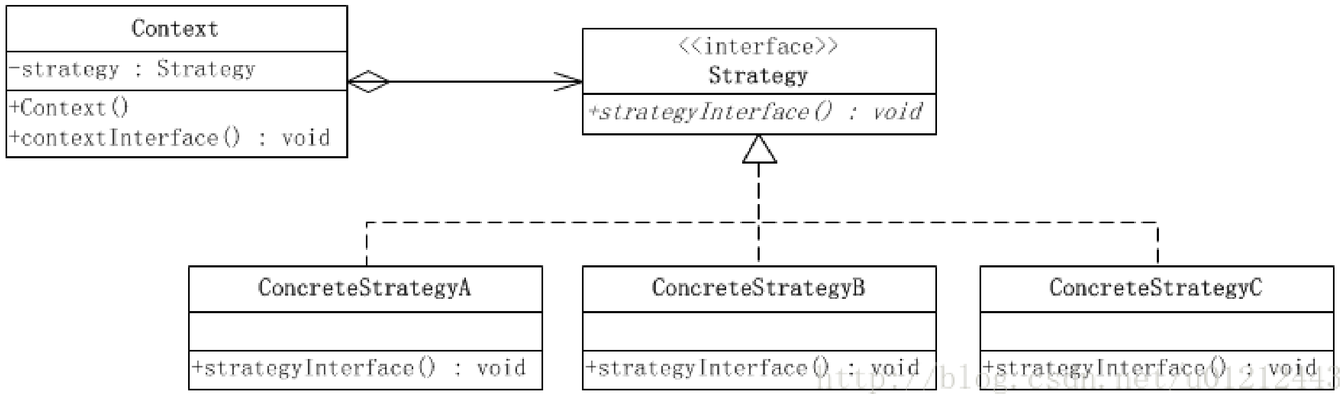
  以命令的方式，解耦调用者与功能的具体实现者，降低系统耦合度，提供了灵活性。

- **适用场景：交互场景**

  实例：Servlet　Controller　线程池

9 命令模式-策略模式的区别





- 策略模式侧重的是一个行为的多个算法实现，可互换算法。
- 命令模式侧重的是为多个行为提供灵活的执行方式。

10 状态模式

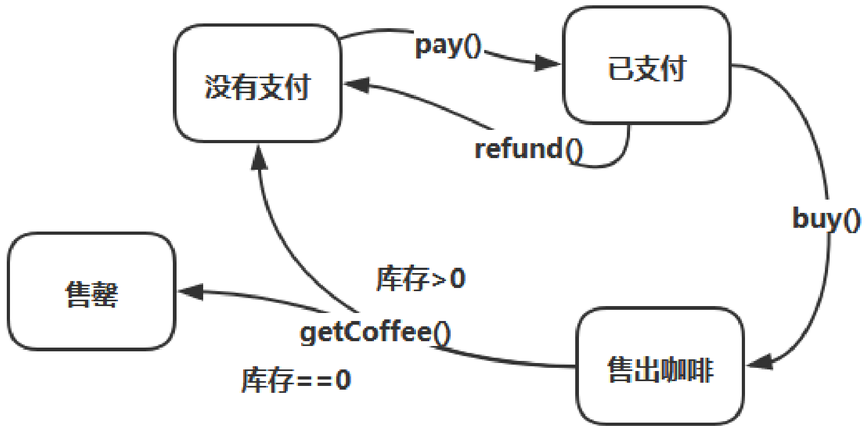
- **示例：** 一个类对外提供了多个行为，同时该类对象有多种状态，不同状态下对外的行为的表现不同，我们该如何来设计该类让它对状态可以灵活扩展？

如请为无人自动咖啡售卖机开发一个控制程序。

CoffeeMachine
- store:int //库存 - state:int //状态

+ pay() //支付
+ refund() //退款
+ buy() //购买
+ getCoffee() //打开阀门

咖啡机状态转换图



- 用户可在咖啡机上进行支付、退款、购买、取咖啡操作。
- 不同的状态下，这四种种操作将有不同的表现。

  如在没有支付状态下，用户在咖啡机上点退款、购买、取咖啡，和在已支付的状态下做这三个操作。

10 状态模式-代码示例

```
public class CoffeeMachine {

    final static int NO_PAY = 0;
    final static int PAY = 1;
    final static int SOLD = 2;
    final static int SOLD_OUT = 4;

    private int state = SOLD_OUT;
    private int store;

    public CoffeeMachine(int store) {
        this.store = store;
        if (this.store > 0) {
            this.state = NO_PAY;
        }
    }

    public void pay() {...}
    public void refund() {...}
    public void buy() {...}
    public void getCoffee() {...}
    …
}
```

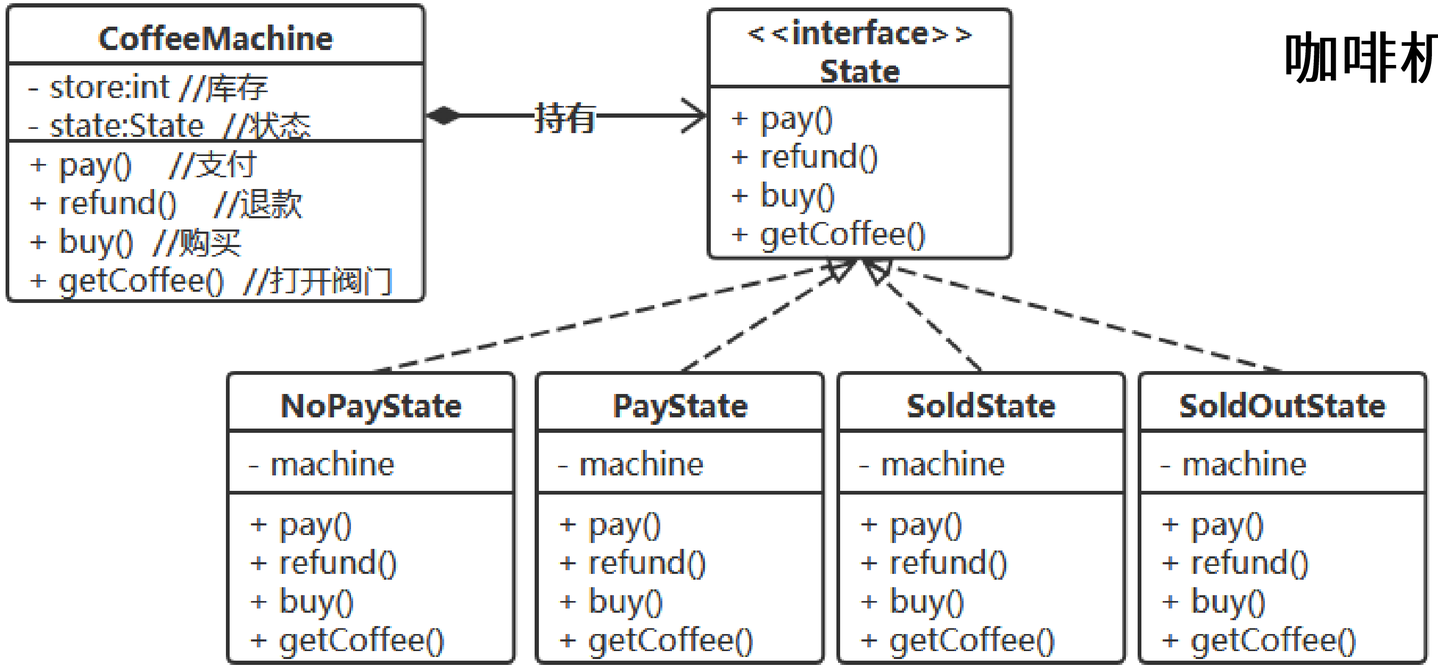
```
public void pay() {
    switch (this.state) {
    case NO_PAY:
        System.out.println("支付成功，请确定购买咖啡。");
        this.state = PAY;
        break;
    case PAY:
        System.out.println("已支付成功，请确定购买咖啡。");
        break;
    case SOLD:
        System.out.println("不可支付，已购买请取用咖啡！");
        break;
    case SOLD_OUT:
        System.out.println("咖啡已售罄，不可购买！");
    }
}
```

```
public void refund() {
    switch (this.state) {
    case NO_PAY:
        System.out.println("你尚未支付，请不要乱按！");
        break;
    case PAY:
        System.out.println("退款成功！");
        this.state = NO_PAY;
        break;
    case SOLD:
        System.out.println("已购买，请取用！");
        break;
    case SOLD_OUT:
        System.out.println("咖啡已售罄，不可购买！");
    }
}
```

如何让状态可以灵活扩展（加状态）？

如何做到开闭原则？

10 状态模式-类图



咖啡机的行为将委托给当前的状态实例

状态可以灵活扩展否?

符合开闭原则否?

10 状态模式-新代码示例

```
public interface State {
    void pay();
    void refund();
    void buy();
    void getCoffee();
}
```

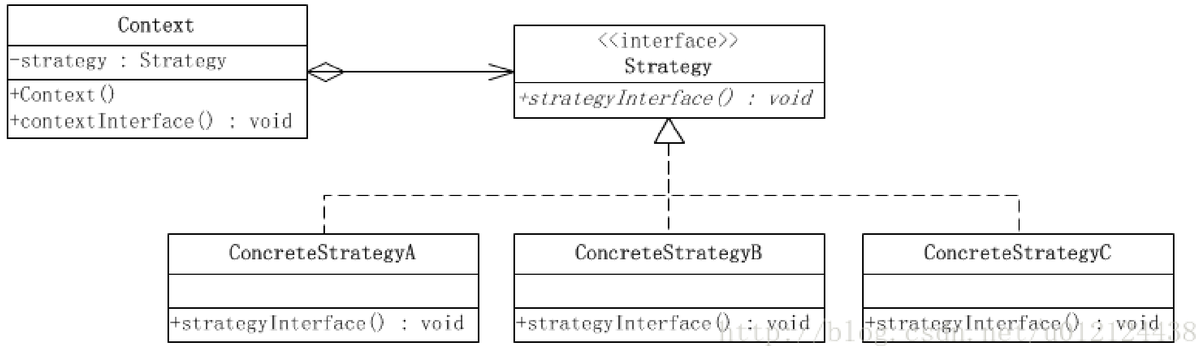
```
public class NoPayState implements State {
    private NewCoffeeMachine machine;

    public NoPayState(NewCoffeeMachine machine) {
        this.machine = machine;
    }

    public void pay() {
        System.out.println("支付成功，请去确定购买咖啡。");
        this.machine.state = this.machine.PAY;
    }
    public void refund() {
        System.out.println("你尚未支付，请不要乱按！");
    }
    public void buy() {
        System.out.println("你尚未支付，请不要乱按！");
    }
    public void getCoffee() {
        System.out.println("你尚未支付，请不要乱按！");
    }
}
```

```
public class PayState implements State {
    private NewCoffeeMachine machine;

    public PayState(NewCoffeeMachine machine) {
        this.machine = machine;
    }
    public void pay() {
        System.out.println("您已支付，请去确定购买！");
    }
    public void refund() {
        System.out.println("退款成功，请收好！");
        this.machine.state = this.machine.NO_PAY;
    }
    public void buy() {
        System.out.println("购买成功，请取用");
        this.machine.state = this.machine.SOLD;
    }
    public void getCoffee() {
        System.out.println("请先确定购买！");
    }
}
```

10 状态模式-新代码示例

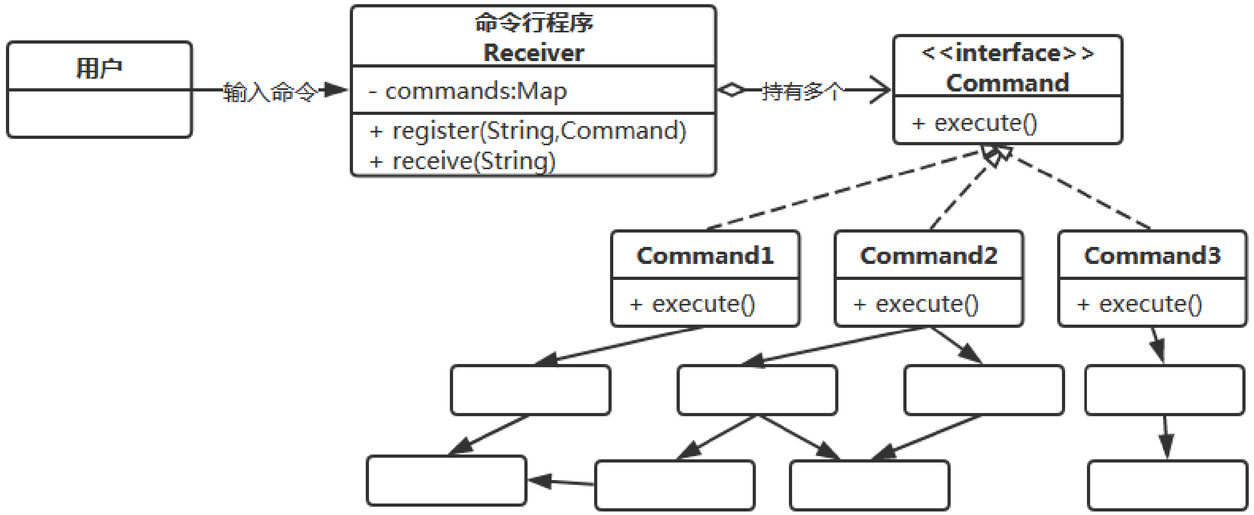
```
public class NewCoffeeMachine {
    final State NO_PAY, PAY, SOLD, SOLD_OUT;
    State state;
    int store;

    public NewCoffeeMachine(int store) {
        NO_PAY = new NoPayState(this);
        PAY = new PayState(this);
        SOLD = new SoldState(this);
        SOLD_OUT = new SoldOutState(this);

        this.store = store;
        if (this.store > 0) {
            this.state = NO_PAY;
        }
    }

    public void pay() {
        this.state.pay();
    }
    public void refund() {
        this.state.refund();
    }
    public void buy() {
        this.state.buy();
    }
    public void getCoffee() {
        this.state.getCoffee();
    }
}
```

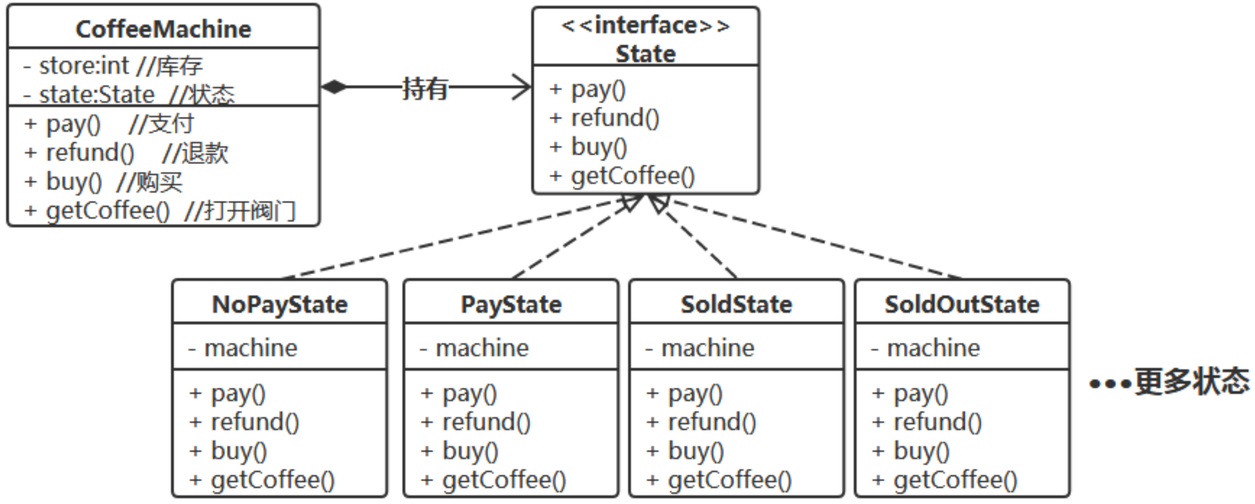
10 状态模式-命令模式-策略模式



- 策略模式侧重的是一个行为的多个算法实现，可互换算法。
- 命令模式侧重的是为多个行为提供灵活的执行方式。
- 状态模式，应用于状态机的情况。

设计原则：区分变与不变，隔离变化
设计原则：面向接口编程
设计原则：多用组合，少用继承

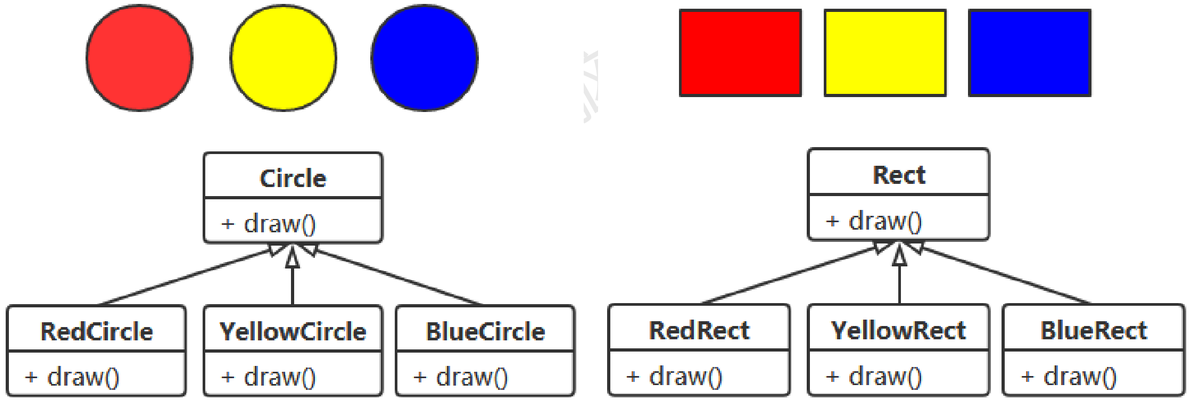




11 桥接模式

- **示例：** 请开发一个画图程序，可以画各种颜色不同形状的图形，请用面向对象的思想设计图形。

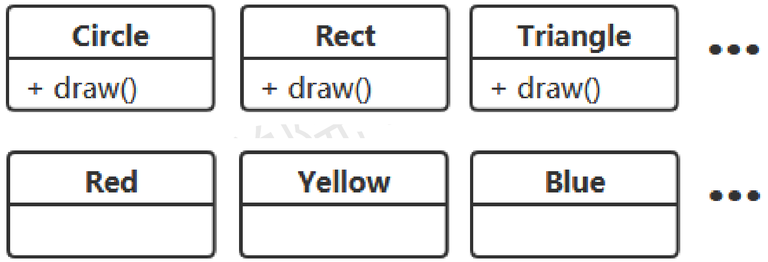
这样设计类可否？　　加颜色、加形状，会类爆炸！



- **分析：**

1、比如有红、黄、蓝三种颜色
2、形状有方形、圆、三角形
3、圆可以是红圆、黄圆、蓝圆

- **变化：会从两个维度发生变化：形状、颜色**

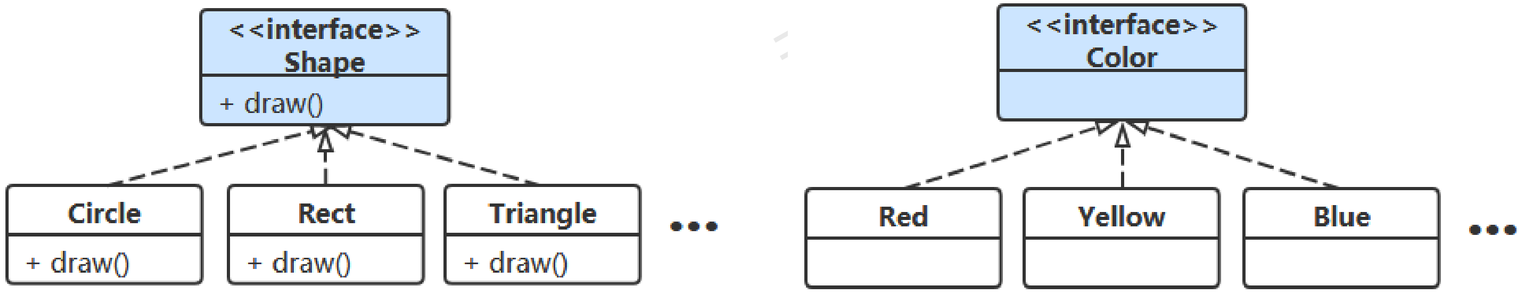


任其在两个维度各自变化，为这两个维度搭个桥，让它们可以融合在一起：桥接模式

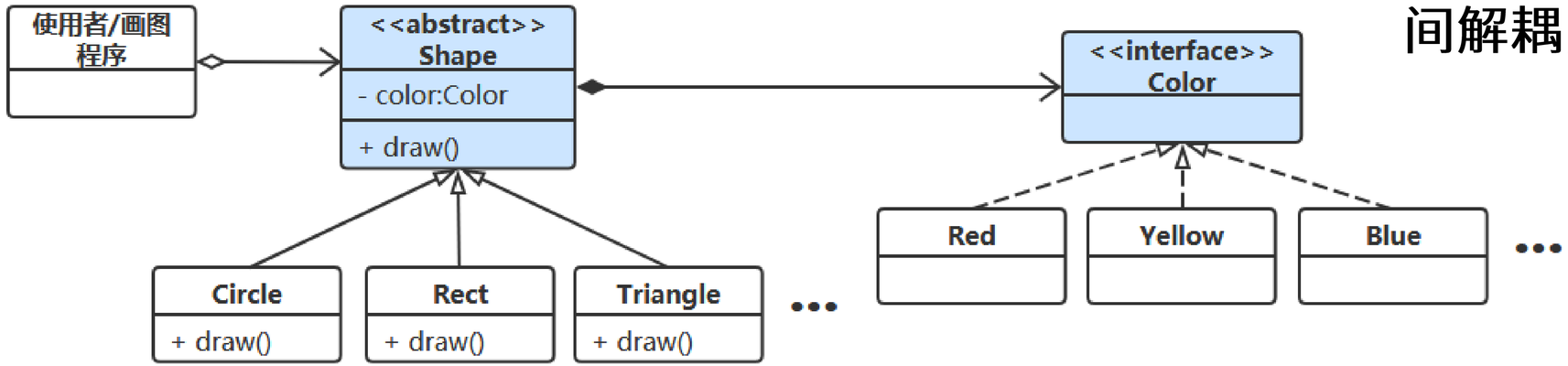
如何搭？

11 桥接模式

- 1、抽象：分别对各个维度进行抽象，将共同部分抽取出来



- 2、组合：将抽象组合在一起（桥接）



桥接：将多个维度的变化以抽象的方式组合在一起。使用者面向抽象。各维度间解耦，可自由变化。

12 单例模式

- 饥汉式 【可用】

```
public class Singleton {

    private final static Singleton INSTANCE = new Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

```
public class Singleton {

    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

12 单例模式

懒汉式

懒汉式1

【不可用】

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

懒汉式2

【线程安全，但不推荐】

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

缺点：实例化后就不应该再同步了，效率低。

12 单例模式

懒汉式

懒汉式3

【不可用】

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                singleton = new Singleton();
            }
        }
        return singleton;
    }
}
```

做不到单例

懒汉式4 双重检查

【推荐使用】

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

注意： volatile关键字修饰很关键

优点：线程安全；延迟加载；效率较高。

12 单例模式

懒汉式

懒汉式5 静态内部类方式

```
public class Singleton {

    private Singleton() {}

    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

【推荐使用】

懒汉式6 用枚举

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {

    }
}
```

【推荐使用】

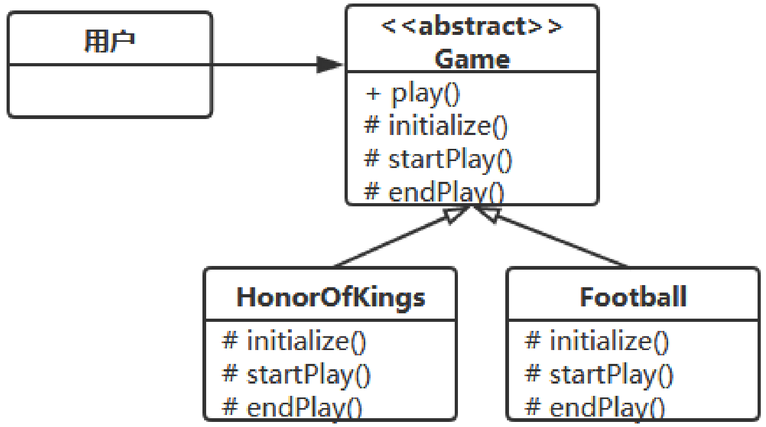
优点：避免了线程不安全，延迟加载，效率高。

原理：类的静态属性只会在第一次加载类的时候初始化。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。

13 模板方法模式

- **示例：** 当我们设计一个类时，我们能明确它对外提供的某个方法的内部执行步骤，但一些步骤，不同的子类有不同的行为时，我们该如何来设计该类？

可以用模板方法模式




```
public abstract class Game {
  protected abstract void initialize();
  protected abstract void startPlay();
  protected abstract void endPlay();
  //模板方法
  public final void play(){
    //初始化游戏
    initialize();
    //开始游戏
    startPlay();
    //结束游戏
    endPlay();
  }
}
```

- **优点：**

1、封装不变部分，扩展可变部分。
2、提取公共代码，便于维护。
3、行为由父类控制，子类实现。

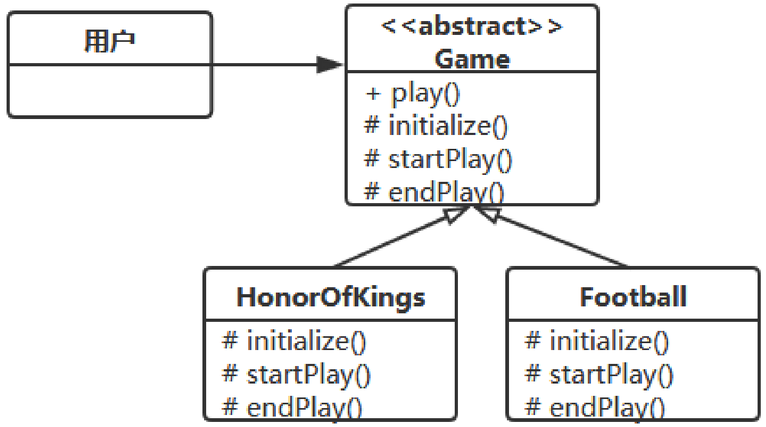
- **适用场景：**

1、有多个子类共有的方法，且逻辑相同。
2、重要的、复杂的方法，可以考虑作为模板方法。

13 模板方法模式

- **示例：** 当我们设计一个类时，我们能明确它对外提供的某个方法的内部执行步骤，但一些步骤，不同的子类有不同的行为时，我们该如何来设计该类？

可以用模板方法模式




```
public abstract class Game {
  protected abstract void initialize();
  protected abstract void startPlay();
  protected abstract void endPlay();
  //模板方法
  public final void play(){
    //初始化游戏
    initialize();
    //开始游戏
    startPlay();
    //结束游戏
    endPlay();
  }
}
```

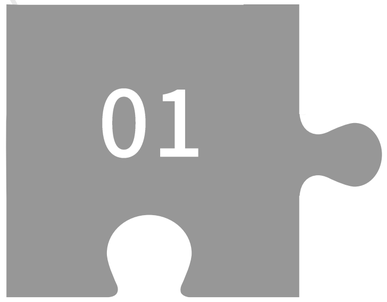
- **优点：**

1、封装不变部分，扩展可变部分。
2、提取公共代码，便于维护。
3、行为由父类控制，子类实现。

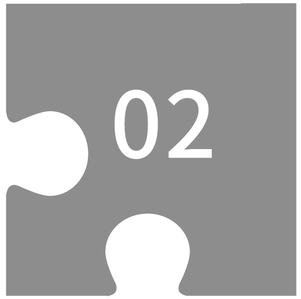
- **适用场景：**

1、有多个子类共有的方法，且逻辑相同。
2、重要的、复杂的方法，可以考虑作为模板方法。

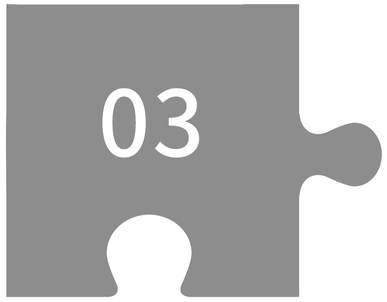
课程目录



设计思想



设计原则



设计模式

总结

设计模式总结

序号	模式 & 描述	包括
1	**创建型模式** 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	•工厂模式（Factory Pattern） •抽象工厂模式（Abstract Factory Pattern） •单例模式（Singleton Pattern） •建造者模式（Builder Pattern） •原型模式（Prototype Pattern）
2	**结构型模式** 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	•适配器模式（Adapter Pattern） •桥接模式（Bridge Pattern） •组合模式（Composite Pattern） •装饰器模式（Decorator Pattern） •外观模式（Facade Pattern） •享元模式（Flyweight Pattern） •代理模式（Proxy Pattern）
3	**行为型模式** 这些设计模式特别关注对象之间的通信。	•责任链模式（Chain of Responsibility Pattern） •命令模式（Command Pattern） •解释器模式（Interpreter Pattern） •迭代器模式（Iterator Pattern） •中介者模式（Mediator Pattern） •备忘录模式（Memento Pattern） •观察者模式（Observer Pattern） •状态模式（State Pattern） •空对象模式（Null Object Pattern） •策略模式（Strategy Pattern） •模板模式（Template Pattern） •访问者模式（Visitor Pattern）

设计原则总结

- **变化隔离原则：找出变化，分开变化和不变的**

  隔离，封装变化的部分，让其他部分不受它的影响。

- **面向接口编程　依赖倒置**　　隔离变化的方式

  使用者使用接口，提供者实现接口。“接口”可以是超类！

- **开闭原则：对修改闭合，对扩展开放**　　隔离变化的方式

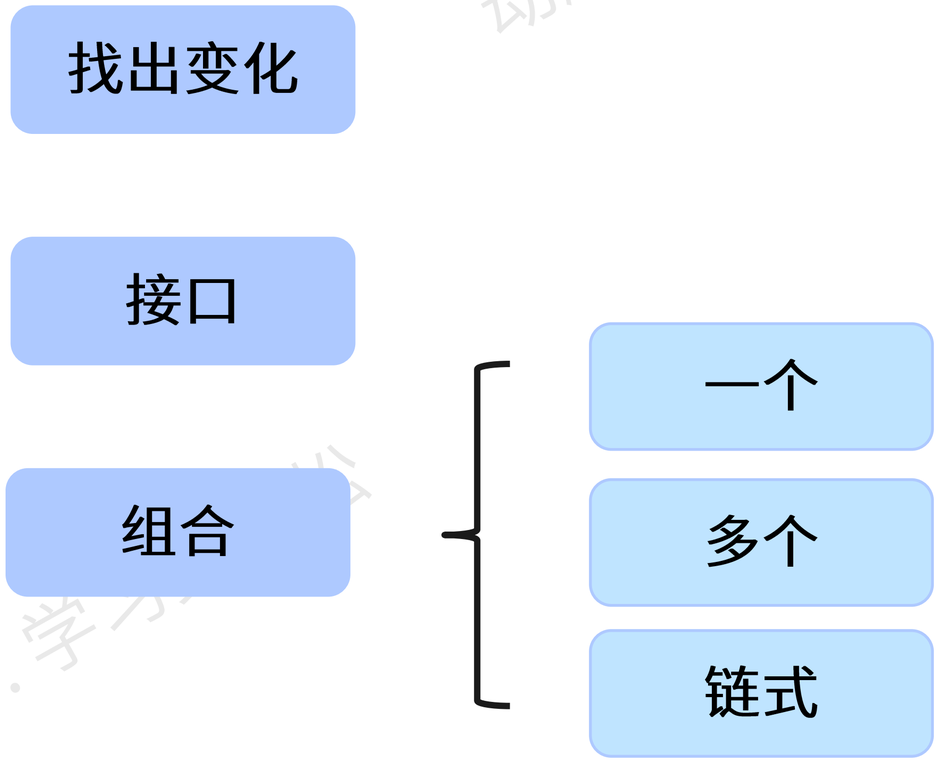
- **多用组合，少用继承**　　灵活变化的方式

- **最少知道原则　又称迪米特法则**

- **单一职责原则**　　方法设计的原则

设计原则总结

- 最后，如果都忘记了，请一定要记住这三！！！



谢谢观看