设计模式

（一）创建者模式

> 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。

1）工厂模式

- **核心思想**：定义一个创建对象的接口，使得创建过程延迟到子类进行，由子类自行定义创建逻辑。**调用者并不知道运行时真正的类名，只知道从“Circle"参数可以创建出一个shape接口的类，至于类的名称是否叫'Circle"，调用者并不知情。**
- **适用情况**：**当某一接口有多种实现类，但是希望封装具体实现类的实现逻辑，只提供接口类的实现。例如一个文件获取接口，文件获取的方式有多种，但是不希望暴露实际获取的逻辑，客户端只能够通过枚举等方式来实现接口对象，再获取文件。**
- **代码示例**：

```
/** 产品接口 */
public interface Shape{
    void draw();
}
/** 实际产品类 */
public class Rectangle implements Shape{
    @override
    void draw(){...}
}
/** 产品工厂类 */
public class ShapeFactory{
    // 参数枚举
    enum ShapeType{ CIRCLE, RECTANGLE, SQUARE }
    // 生产方法
    public Shape getShape(ShapeType shapeType){
        switch(shapeType){
            case CIRCLE: return new Circle();
            case RECTANGLE: return new Rectangle();
            case SQUARE: return new Square();
            default: throw new UnknownTypeException();
        }
    }
}
/** 客户端调用，假定客户端并不知道工厂方法具体实现类名 */
public static void main(String[] args){
    ShapeFactory shapeFactory = new ShapeFactory();
    Shape shape = ShapeFactory.getShape(ShapeFactory.ShapeType.CIRCLE);
    shape.draw();
}
```

2）抽象工厂模式

- **核心思想**：工厂模式是一类产品给一个工厂接口，而抽象工厂则是几类相关联的产品对应多个具体工厂，然后将这些具体工厂抽象到一个抽象工厂中，由抽象工厂来统一实现不同产品的生产。
- **适用情况**： **等同于工厂模式的扩展版，适用于多个相关联的接口，为了不暴露实际逻辑，将对应的实现都放在工厂类中，再将工厂集中在一个抽象工厂中，由选择器选择实际生产的工厂。例如文件获取接口和文件上传接口，都有多种方式实现各自的逻辑，分别将对象实例过程封装在获取工厂和上传工厂中，将这两个工厂统一封装在文件抽象工厂中。**
- **代码示例**：

```
/** 除了上面的Shape接口，新增一类产品Color */
public interface Color{
    void fill();
}
/** Color具体实现类 */
public class Red implements Color{
    @override
    void fill(){...}
}
/** 抽象工厂类 */
public abstract class AbstractFactory {
    public abstract Color getColor(ColorType colorType);
    public abstract Shape getShape(ShapeType shapeType);
}
/** Color工厂类 */
public class ColorFactory extends AbstractFactory{
    enum ColorType{ RED, BLUE, YELLOW }

    // 同样在ShapeFactory里重写的getColor方法也返回null
    @override
    public Shape getShape(ShapeType shapeType){ return null; }

    // 生产方法
    @Override
    public Color getColor(ColorType colorType){
        swtich(colorType){
            case RED: return new Red();
            case BLUE: return new Blue();
            case YELLOW: return new Yellow();
            default: throw new UnknownTypeException();
        }
    }
}
/** 工厂生成器或工厂选择器 */
public class FactoryProducer{
    enum FactoryType{ SHAPE_FACTORY, COLOR_FACTORY }
    public static AbstractFactory getFactory(FactoryType factoryType){
        switch(factoryType){
            case SHAPE_FACTORY: return new ShapeFactory();
            case COLOR_FACTORY: return new ColorFactory();
            default: throw new UnknownTypeException();
        }
    }
}
/** 客户端调用方法 */
public static void main(String[] args){
```

```
    FactoryProducer factoryProducer = new FactoryPorducer();
    AbstarctFactory abstarctFactory =
        factoryPorducer.getFactory(FactoryProducer.FactoryType.SHAPE_FACTORY);
    abstarctFactory.getShape(ShapeFactory.ShapeType.CIRCLE).draw();
    AbstarctFactory abstarctFactory2 =
        factoryPorducer.getFactory(FactoryProducer.FactoryType.SHAPE_FACTORY);
    abstractFactory2.getColor(ColorFactory.ColorType.RED).fill();
}
```

3）单例模式

- **核心思想**：保证一个类只有一个实例，提供一个全局访问点，避免频繁对象创建和销毁工作。**构造函数是私有的。**
- **适用情况**：**某个类的实例化过程复杂且消耗资源太多，为了避免不必要开销，只对该类实例化一次，调用时只返回该类的唯一实例。**
- **代码示例**：单例模式有较多实现方式，包括懒汉饿汉（区别在于是否声明时就初始化），是否线程安全等。

```
/** 懒汉单例模式 */
// 非线程安全
public class Singleton{
    private Singleton(){}
    private static Singleton instance;
    public static Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
// 线程安全，相比较于非线程安全就是加了个Synchronized
public class Singleton{
    private Singleton(){}
    private static Singleton instance;
    public static synchronized Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

```
/** 饿汉单例模式 */
public class Singleton{
    private Singleton(){}
    // 声明时即初始化，不存在多线程竞争初始化现象，线程安全
    private static Singleton instance = new Singleton();
    public static Singleton getInstance(){
        return instance;
    }
}
```

```
/** 双重检验锁，Double-Checked-Locking
 * 双重检验是为了避免出现两个线程都通过了第一层检验，之后都初始化
```

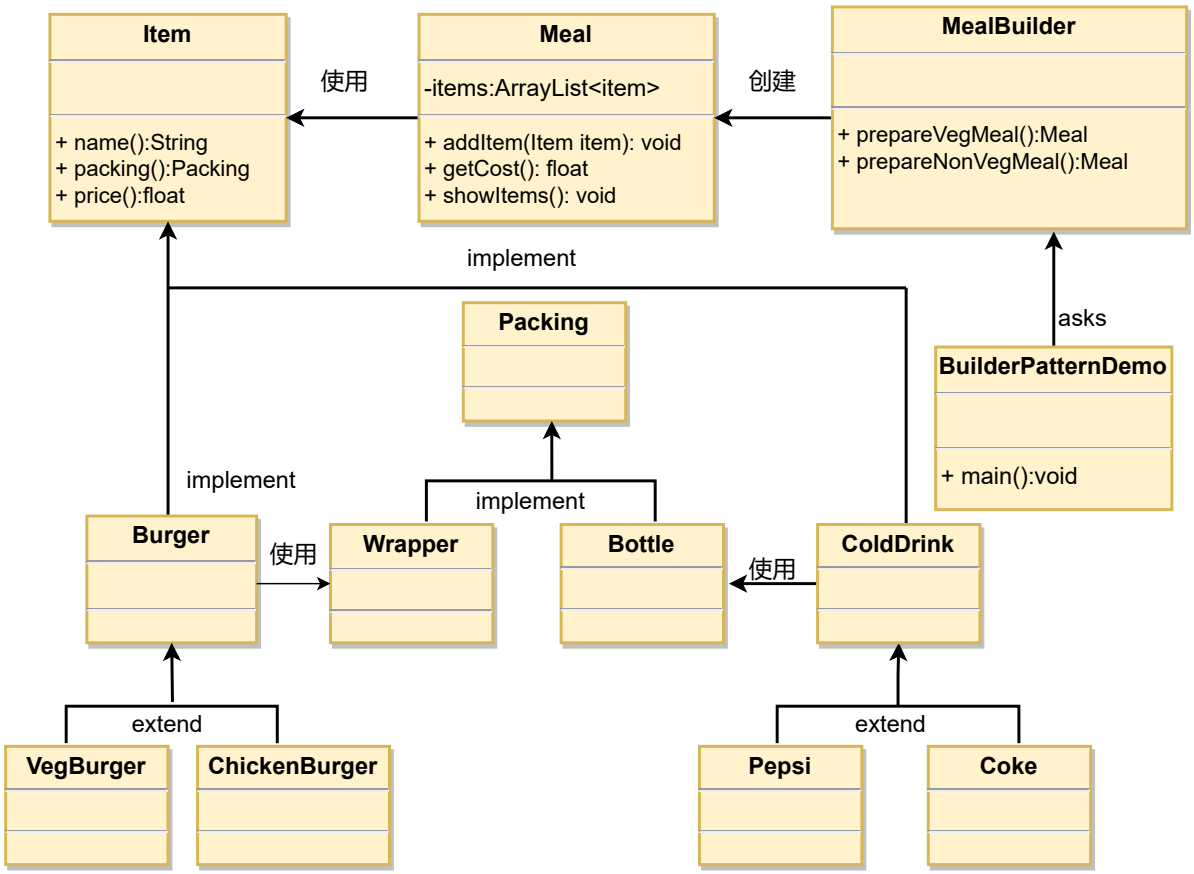
```
 * volatile修饰的变量读写是直接从主内存读写的，不加volatile的变量，线程访问到的其实是主内存的拷贝副本
 * 加上volatile是很有必要的，可以避免指令重排序的过程
 * new一个对象实际上分为3步指令，其中2和3是可以进行重排序的
 * 1、分配对象内存
 * 2、调用构造器方法，执行初始化
 * 3、将对象引用赋值给变量
 * 假设有2个线程A和B，都通过了第一次检查，此时时间片轮到A，A获得锁并通过第二轮检测
 * 此时，A开始new对象，A的指令顺序为1->3->2，然后在A执行到3的时候，时间片到期了
 * 此时，instance != null但是并未初始化完成。
 * B获得时间片和锁之后，在第二轮检测时候判断已经初始化完成，直接返回instance
 * 实际上B线程后续访问instance的时候由于未执行构造器，访问出现异常
*/
public class Singleton{
    private Singleton(){}
    private static volatile Singleton instance;
    public static Singleton getInstance(){
        if (instance == null){
            synchronized(Singleton.class){
                if (instance == null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

```
/** 静态内部类
 * 相对比于饿汉模式，静态内部类只有在被调用的时候才会进行初始化，因此可以保证延迟初始化
 * 同时也保证线程安全
*/
public class Singleton{
    private Singleton(){}
    private static class SingletonHolder{
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance(){
        return SingletonHolder.INSTANCE;
    }
}

/** 枚举实现单例模式
 * 避免多线程同步问题，绝对防止多次实例化
*/
public enum Singleton {
    INSTANCE;
    public void doSomething() {
        System.out.println("doSomething");
    }
    // 外部使用时可以直接Singleton.INSTANCE.doSomething()
}
```

4）建造者模式

- **核心思想**：将一个复杂对象的构建过程分成几步来实现，即由各个部分的子对象实现，可以保障即使复杂对象的各个子部分剧烈变化，但组合在一起的复杂对象组建过程相对稳定。**与工厂模式的区别是建造者模式更关注与零件装配的顺序，工厂方法模式注重的是整体对象的创建方法，而建造者模式注重的是部件构建的过程。**
- **适用情况**：**建造者模式更注重于将构造器分段实现。例如一辆车的实现，可以由多种轮胎、多种引擎、多种座椅等组成，通过建造者模式进行分段建造和组装。**
- **代码示例**：




```java
/** 物品接口 */
public interface Item{
    public String name();
    public Packing packing();
    public float price();
}
/** 打包方式接口 */
public interface Packing(){
    public String pack();
}
/** 打包方式接口实现类 */
public class Wrapper implements Packing{
    @override
    public String pack() { return "Wrapper"; }
}
public class Bottle implements Packing{
    @override
    public String pack() { return "Bottle"; }
```

```
}
/** 物品接口抽象类，进行物品的分类 */
public abstarct class Burger implements Item{
    @override
    public Packing packing() { return new Wrapper(); }
    @override
    public abstract float price();
}
public abstract class ColdDrink implements Item{
    @override
    public Packing packing() { return new Bottle(); }
    @override
    public abstract float price();
}
/** 物品具体实现类 */
public class VegBurger extends Burger {
   @Override
   public float price() {
      return 25.0f;
   }
   @Override
   public String name() {
      return "Veg Burger";
   }
}
public class Coke extends ColdDrink {
   @Override
   public float price() {
      return 30.0f;
   }
   @Override
   public String name() {
      return "Coke";
   }
}
/** 物品综合类 */
public class Meal {
   private List<Item> items = new ArrayList<Item>();
   public void addItem(Item item){
      items.add(item);
   }
   public float getCost(){
      float cost = 0.0f;
      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }
   public void showItems(){
      for (Item item : items) {
         System.out.print("Item : "+item.name());
         System.out.print(", Packing : "+item.packing().pack());
         System.out.println(", Price : "+item.price());
      }
   }
}
/** 物品建造者类 */
public class MealBuilder extends Builder{
```

```
   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }
   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
/** 监管类，有时候也可以由客户端直接进行 */
public class Director {
    private Builder builder;
    public Director(Builder builder) {
        this.builder = builder;
    }
    public void setBuilder(Builder builder) {
        this.builder = builder;
    }
    public void construct() { ... }
}
/** 客户端调用方法 */
public static void main(String[] args){
    Director director = new Director(new MealBuilder());
    director.construct();
    // 或者不用监管
    Builder builder = new MealBuilder();
    Meal meal = builder.prepareNonVegMeal();
    Meal meal2 = builder.prepareVegMeal();
}
```

5）原型模型

- **核心思想**：这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。
- **适用情况**：**情况类似于单例模式，都是实例化对象的过程消耗资源太多。区别在于原型模型适用于那种不希望频繁实例化，但是又必须要新的对象的情况。**
- **代码示例**：

```
/** 实现克隆的抽象类 */
public abstract class Shape implements Cloneable {
   abstract void draw();
   public Object clone() {
      Object clone = null;
      try {
         clone = super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
      return clone;
   }
}
/** 对象缓存 */
```

```
public class ShapeCache {
   private static Hashtable<String, Shape> shapeMap
      = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(),rectangle);
   }
}
```

（二）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。

1）适配器模式

- **核心思想**：作为两个不兼容的接口之间的桥梁。常用于系统需要使用现有的类，而此类的接口不符合系统的需要。
- **适用情况**：常用于给已经服役的接口添加适配器以满足新的需求，在未服役的接口设计时不需要使用此模式。
- **代码示例**：假设电脑原有SD接口可以读写SD卡，现在需要通过该接口读取TF卡。

```
/** SD卡接口与实现类 */
public interface SDCard{
    String readSD();
}
public class SDCardImpl implements SDCard{
    @override
    public String readSD(){ ... }
}
/** 计算机接口与实现类 */
public interface Computer{
    String readSD(SDCard sdCard);
}
public class MacBook implements Computer{
    @override
    public String readSD(SDCard sdCard){ return sdCard.readSD(); }
}
```

```
/** 通过适配器使得SD卡接口同样可以读取TF卡 */
public interface TFCard{
    String readTF();
}
public class TFCardImpl implements TFCard{
    @override
    public String readTF(){ ... }
}
public class SDAdapterTF implements SDCard{
    private TFCard tfCard;
    public SDAdapterTF(TFCard tfCard){
        this.tfCard = tfCard;
    }
    @override
    public String readSD(){
        return tfCard.readTF();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Computer computer = new MacBook();
    // 原始接口功能，读取SD卡
    SDCard sdCard = new SDCardImpl();
    computer.readSD(sdCard);

    // 扩展接口功能，读取TF卡。可以看出Computer接口与实现类无需做改动
    TFCard tfCard = new TFCardImpl();
    SDCard sdAdapterTF = new SDAdapterTF(tfCard);
    computer.readSD(sdAdapterTF);
}
```

2）桥接模式

- **核心思想**：桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。
- **适用情况**：为了在抽象化角色和具体化角色之间增加更多的灵活性，避免两个层次之间建立静态的继承联系，**转而使用桥接模式在它们之间建立一个关联关系，使用桥接接口来实现抽象部分。**
- **代码示例**：

```
/** 创建桥接接口与实现类 */
public interface DrawAPI{
    public void drawCircle(int radius, int x, int y);
}
public class RedCricle implements DrawAPI{
    @override
    public void drawCircle(int radius, int x, int y) { ... }
}
/** 抽象类使用桥接 */
public abstract class Shape{
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}
```

```
/** 抽象类实现类 */
public class Circle extends Shape{
    private int x, y, radius;
    public Circle(int x, int y, int radius, DrawAPI drawAPI){
        this.x = x;
        this.y = y;
        this.radius = radius;
        this.drawAPI = drawAPI;
    }
    @override
    public void draw(){
        this.drawAPI.drawCircle(radius, x, y);
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    DrawAPI drawAPI = new RedCircle();
    Shape redCircle = new Circle(100, 100, 10, drawAPI);
    redCircle.draw();
}
```

3）过滤器模式/标准模式

- **核心思想**：允许使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们联合起来，通过结合多个标准来获得单一标准。
- **适用情况**：用于过滤对象，本质上和jdk8之后流的过滤一样
- **代码示例**：

```
/** 用于过滤的对象类 */
public class Person{
    private String gender;
    private Integer age;
    // set/get方法
}
/** 过滤器接口与实现类 */
public interface Criteria{
    public List<Person> meetCriteria(List<Person> persons);
}
public class CriteriaMale implements Criteria{
    @override
    public List<Person> meetCriteria(List<Person> persons){
        List<Person> result = new ArrayList<>();
        for (Person person : persons){
            if (person.getGender().equals("MALE")){
                result.add(person);
            }
        }
        return result;
    }
}
public class CriteriaAge implements Cirteria{
    @override
    public List<Person> meetCriteria(List<Person> persons){
        List<Person> result = new ArrayList<>();
        for (Person person : persons){
            if (person.getAge() >= 20){
```

```
                result.add(person);
            }
        }
        return result;
    }
}
/** 组合过滤器 */
public class OrCriteria implements Criteria{
   private Criteria criteria;
   private Criteria otherCriteria;

   public OrCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
           firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;
   }
}
/** 客户端调用方式 */
public static void main(String[] args){
    List<Person> persons = new ArrayList<>();
    persons.add(new Person("MALE", 20));
    Criteria criteria =
        new OrCriteria(new CirteriaMale(), new CirteriaAge());
    List<Person> result = criteria.meetCriteria(persons);
}
```

4）组合模式

- **核心思想**：**将对象组合成树形结构**以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。
- **适用情况**：希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。
- **代码示例**：

```
/** 组合对象类，对象中包含对其他对象的引用，即类似于树形结构 */
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;
   //构造函数
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
```

```
        this.salary = sal;
        subordinates = new ArrayList<Employee>();
    }
    public void add(Employee e) { subordinates.add(e);}
    public void remove(Employee e) { subordinates.remove(e);}
    public List<Employee> getSubordinates(){ return subordinates;}
}
/** 客户端调用方式 */
public static void main(String[] args){
    Employee CEO = new Employee("John","CEO", 30000);
    Employee headSales = new Employee("Robert","HeadSales", 20000);
    Employee headMarketing =
        new Employee("Michel","HeadMarketing", 20000);
    Employee clerk1 = new Employee("Laura","Marketing", 10000);
    Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
    CEO.add(headSales);
    CEO.add(headMarketing);
    headSales.add(salesExecutive1);
    headMarketing.add(clerk1);
    // 通过这种方式组成树形对象结构，再采用树的遍历方法进行遍历
}
```

5）装饰器模式

- **核心思想**：允许向一个现有的对象添加新的功能，同时又不改变其结构。**通过创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。**
- **适用情况**：在不想增加很多子类的情况下扩展类。**动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。**
- **代码示例**：

```
/** 接口类与实现类 */
public interface Hero{
    void learnSkills();
}
public class BlindMonk implements Hero{
    private String name;
    public BlindMonk(String name){ this.name = name; }
    @override
    public void learnSkills(){
        System.out.println(name + " has learn this skills!");
    }
}
/** 装饰类与实现类 */
public class Skills implements Hero{
    private Hero hero;
    public Skills(Hero hero) { this.hero = hero;}
    @Override
    public void learnSkills() {
        if(hero != null) hero.learnSkills();
    }
}
public class Skill_Q extends Skills{
    private String skillName;
    public Skill_Q(Hero hero,String skillName) {
        super(hero);
        this.skillName = skillName;
```

```
    }
    @Override
    public void learnSkills() {
        System.out.println("学习了技能Q:" +skillName);
        super.learnSkills();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Hero hero = new BlindMonk("李青");
    Skills skills = new Skills(hero);
    Skills q = new Skill_Q(skills,"天音波/回音击");
    q.learnSkills();
}
```

6）外观模式

- **核心思想**：向现有的系统添加一个接口，来隐藏系统的复杂性。**为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。**
- **适用情况**：为复杂的模块或子系统提供外界访问的模块，客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。
- **代码示例**：

```
/** 外观类 */
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;
   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }
   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
/** 客户端调用方式 */
public static void main(String[] args){
    ShapeMaker shapeMaker = new ShapeMaker();
    shapeMaker.drawCircle();
    shapeMaker.drawSquare();
    shapeMaker.drawRectangle();
}
```

7）享元模式

- **核心思想**：主要用于减少创建对象的数量，以减少内存占用和提高性能。尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。
- **适用情况**：**在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。**
- **代码示例**：

```
/** 工厂类添加缓存，当缓存不覆盖时才新建对象，颜色 -> 圆对象 */
public class ShapeFactory {
   private static final HashMap<String, Shape> circleMap = new HashMap<>();
   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);
      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
/** 客户端调用方式，circle = circle2 */
public static void main(String[] args){
    Circle circle = (Circle)ShapeFactory.getCircle("RED");
    Circle circle2 = (Circle)ShapeFactory.getCircle("RED");
}
```

8）代理模式

- **核心思想**：为其他对象提供一种代理以控制对这个对象的访问。**和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。**
- **适用情况**：想在访问一个类时做一些控制。
- **代码示例**：

```
/** 接口与实现类 */
public interface Image{
    void display();
}
public class RealImage implements Image{
    private String fileName;
    public RealImage(String fileName){
        this.fileName = fileName;
        loadFromDisk(fileName);
    }
    @override
    public void display(){
        System.out.println("Displaying " + fileName);
    }
    private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
/** 代理类 */
public class ProxyImage implements Image{
```

```
    private RealImage realImage;
    private String fileName;
    public ProxyImage(String fileName){ this.fileName = fileName; }
    @override
    public void display(){
        if (realImage == null) realImage = new RealImage(fileName);
        realImage.display();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Image image = new ProxyImage("1.jpg");
    image.display();
}
```

```
/** 动态代理 */
public interface Subject{
    int sellBooks();
    String speak();
}
public class RealSubject implements Subject{
    @Override
    public int sellBooks() {
        System.out.println("卖书");
        return 1 ;
    }
    @Override
    public String speak() {
        System.out.println("说话");
        return "张三";
    }
}
/** 处理器 */
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
public class MyInvocationHandler implements InvocationHandler{
    Subject realSubject;
    public MyInvocationHandler(Subject realSubject){
        this.realSubject = realSubject;
    }
    // 主要重写方法
    @override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable{
        System.out.println("调用代理类");
        if(method.getName().equals("sellBooks")){
            int invoke = (int)method.invoke(realSubject, args);
            System.out.println("调用的是卖书的方法");
            return invoke ;
        }else {
            String string = (String) method.invoke(realSubject,args) ;
            System.out.println("调用的是说话的方法");
            return  string ;
        }
    }
}
/** 客户端调用方法 */
```

```
public static void main(String[] args){
    Subject realSubject = new RealSubject();
    MyInvocationHandler myInvocationHandler =
        new MyInvocationHandler(realSubject);
    Subject proxyClass = (Subject)Proxy.newProxyInstance(
        ClassLoader.getSystemClassLoader(), // 类加载器
        new Class[]{Subject.class},         // 类class对象
        MyInvocationHandler);               // 代理器
    proxyClass.sellBooks();
    proxyClass.speak();
}
```

```
/** Cglib
 * 用于加强JDK的动态代理以及实现动态代理所不能实现的功能
 * Proxy.newProxyInstance代理生成的前提是“被代理的类实现了某个接口”
 * 若被代理的类并没有实现接口，或者希望加强性能的时候，应该考虑使用Cglib
 * Cglib的底层原理是生成被代理类的子类，然后子类覆盖被代理类的方法，除了final与private域
*/
public class Engineer {
    // 可以被代理
    public void eat() {
        System.out.println("工程师正在吃饭");
    }
    // final 方法不会被生成的字类覆盖
    public final void work() {
        System.out.println("工程师正在工作");
    }
    // private 方法不会被生成的字类覆盖
    private void play() {
        System.out.println("this engineer is playing game");
    }
}
/** Cglib代理类 */
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class CglibProxy implements MethodInterceptor {
    private Object target;
    public CglibProxy(Object target) { this.target = target; }

    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        System.out.println("###   before invocation");
        Object result = method.invoke(target, objects);
        System.out.println("###   end invocation");
        return result;
    }

    public static Object getProxy(Object target) {
        Enhancer enhancer = new Enhancer();
        // 设置需要代理的对象
        enhancer.setSuperclass(target.getClass());
        // 设置代理人
        enhancer.setCallback(new CglibProxy(target));
```

```
        return enhancer.create();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Engineer engineer = (Engineer) CglibProxy.getProxy(new Engineer());
    engineer.eat();
}
```

（三）行为型模型

> 这些设计模式特别关注对象之间的通信。

1）责任链模式

- **核心思想**：为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者。
- **适用情况**：有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。不明确指定接收者的情况下，向多个对象中的一个提交一个请求或可动态指定一组对象处理请求。
- **代码示例**：

```
/** 抽象记录器 */
public abstract class AbstractLogger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;
    //责任链中的下一个元素
    protected AbstractLogger nextLogger;

    public void setNextLogger(AbstractLogger nextLogger){
        this.nextLogger = nextLogger;
    }

    public void logMessage(int level, String message){
        if(this.level <= level){
            write(message);
        }
        if(nextLogger !=null){
            nextLogger.logMessage(level, message);
        }
    }

    abstract protected void write(String message);
}
/** 记录器实现类 */
public class ConsoleLogger extends AbstractLogger {
    public ConsoleLogger(int level){
        this.level = level;
    }
    @Override
```

```
    protected void write(String message) {
        System.out.println("Standard Console::Logger: " + message);
    }
}
public class ErrorLogger extends AbstractLogger {
    public ErrorLogger(int level){
        this.level = level;
    }
    @Override
    protected void write(String message) {
        System.out.println("Error Console::Logger: " + message);
    }
}
public class FileLogger extends AbstractLogger {
    public FileLogger(int level){
        this.level = level;
    }
    @Override
    protected void write(String message) {
        System.out.println("File::Logger: " + message);
    }
}
/** */
public static void main(String[] args){
    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
    AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
    AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
    errorLogger.setNextLogger(fileLogger);
    fileLogger.setNextLogger(consoleLogger);
    AbstractLogger loggerChain = errorLogger;
    loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");
    loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is a debug level information.");
    loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
}
```

2）命令模式

- **核心思想**：请求以命令的形式包裹在对象中，并传给调用对象。定义三个角色：received 真正的命令执行对象、Command、invoker 使用命令对象的入口
- **适用情况**：为了将"行为请求者"与"行为实现者"解耦，将一组行为抽象为对象，可以实现二者之间的松耦合。
- **代码示例**：

```
/** 请求类 */
public class Stock{
    private String name = "ABC";
    private int quantity = 10;
    public void buy(){
        System.out.println("Stock [ Name: "+name+",
           Quantity: " + quantity +" ] bought");
    }
    public void sell(){
        System.out.println("Stock [ Name: "+name+",
           Quantity: " + quantity +" ] sold");
```

```
    }
}
/** Command接口与实现类 */
public interface Command{
    void execute();
}
public class BuyStock implements Command{
    private Stock abcStock;
    public BuyStock(Stock abcStock){
        this.abcStock = abcStock;
    }
    public void execute() {
        abcStock.buy();
    }
}
public class SellStock implements Command {
    private Stock abcStock;
    public SellStock(Stock abcStock){
        this.abcStock = abcStock;
    }
    public void execute() {
        abcStock.sell();
    }
}
/** 命令调用类 */
public class Broker {
    private List<Command> commandList = new ArrayList<Command>();
    public void takeCommand(Command command){
        commandList.add(command);
    }
    public void placeCommands(){
        for (Command command : commandList) {
            command.execute();
        }
        commandList.clear();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Stock stock = new Stock();
    Command buyStock = new BuyStock(stock);
    Command sellStock = new SellStock(stock);
    Broker broker = new Broker();
    broker.takeCommand(buyStock);
    broker.takeCommand(sellStock);
    broker.placeCommands();
}
```

3）解释器模式

- **核心思想**：这种模式实现了一个表达式接口，给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。
- **适用情况**：**如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。**
- **代码示例**：

```
/** 表达式接口与实现类 */
public interface Expression {
    public boolean interpret(String context);
}
public class TerminalExpression implements Expression {
    private String data;
    public TerminalExpression(String data){
        this.data = data;
    }
    @Override
    public boolean interpret(String context) {
        if(context.contains(data)){
            return true;
        }
        return false;
    }
}
public class OrExpression implements Expression {
    private Expression expr1 = null;
    private Expression expr2 = null;
    public OrExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}
public class AndExpression implements Expression {
    private Expression expr1 = null;
    private Expression expr2 = null;
    public AndExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) && expr2.interpret(context);
    }
}
/** 客户端调用方式 */
//规则：Robert 和 John 是男性
public static Expression getMaleExpression(){
    Expression robert = new TerminalExpression("Robert");
    Expression john = new TerminalExpression("John");
    return new OrExpression(robert, john);
}
//规则：Julie 是一个已婚的女性
public static Expression getMarriedWomanExpression(){
    Expression julie = new TerminalExpression("Julie");
    Expression married = new TerminalExpression("Married");
    return new AndExpression(julie, married);
}
public static void main(String[] args) {
    Expression isMale = getMaleExpression();
    Expression isMarriedWoman = getMarriedWomanExpression();
```

```
    System.out.println("John is male? " + isMale.interpret("John"));
    System.out.println("Julie is a married women? "
      + isMarriedWoman.interpret("Married Julie"));
}
```

4）迭代器模式

- **核心思想**：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。
- **适用情况**：把在元素之间游走的责任交给迭代器，而不是聚合对象。
- **代码示例**：

```
/** 迭代器与容器接口 */
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
public interface Container {
    public Iterator getIterator();
}
/** 容器实现类 */
public class NameRepository implements Container {
    public List<String> names = new ArrayList<>();
    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }
    // 内部私有类实现迭代器
    private class NameIterator implements Iterator {
        int index;
        @Override
        public boolean hasNext() {
            if(index < names.size()){
                return true;
            }
            return false;
        }

        @Override
        public Object next() {
            if(this.hasNext()){
                return names.get(index++);
            }
            return null;
        }
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    NameRepository namesRepository = new NameRepository();
    namesRepository.names.add("a");
    namesRepository.names.add("b");
    for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
        String name = (String)iter.next();
        System.out.println("Name : " + name);
    }
}
```

5）中介者模式

- **核心思想**：这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。
- **适用情况**：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂。用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
- **代码示例**：

```
/** 中介类 */
public class ChatRoom {
    public static void showMessage(User user, String message){
        System.out.println(new Date().toString()
            + " [" + user.getName() +"] : " + message);
    }
}
/** 用户类 */
public class User {
    private String name;
    public User(String name){
        this.name  = name;
    }
    public void sendMessage(String message){
        ChatRoom.showMessage(this,message);
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    User robert = new User("Robert");
    User john = new User("John");
    robert.sendMessage("Hi! John!");
    john.sendMessage("Hello! Robert!");
}
```

6）备忘录模式

- **核心思想**：在不破坏封装性的前提下，捕获一个对象的内部状态，在该对象外保存这个状态。备忘录模式使用三个类 *Memento*、*Originator* 和 *CareTaker*。**Memento 包含了要被恢复的对象的状态。Originator 创建并在 Memento 对象中存储状态。Caretaker 对象负责从 Memento 中恢复对象的状态。**
- **适用情况**：为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，提供一个可回滚的操作。
- **代码示例**：

```
/** 负责存储状态 */
public class Memento {
    private String state;
    public Memento(String state){
        this.state = state;
    }
    public String getState(){
        return state;
    }
}
/** 负责创建对象并且保存状态 */
```

```
public class Originator {
    private String state;
    public void setState(String state){
        this.state = state;
    }
    public String getState(){
        return state;
    }
    public Memento saveStateToMemento(){
        return new Memento(state);
    }
    public void getStateFromMemento(Memento Memento){
        state = Memento.getState();
    }
}
/** 负责恢复对象状态 */
public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();
    public void add(Memento state){
        mementoList.add(state);
    }
    public Memento get(int index){
        return mementoList.get(index);
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Originator originator = new Originator();
    CareTaker careTaker = new CareTaker();
    originator.setState("State #1");
    originator.setState("State #2");
    careTaker.add(originator.saveStateToMemento());
    originator.setState("State #3");
    careTaker.add(originator.saveStateToMemento());
    originator.setState("State #4");

    System.out.println("Current State: " + originator.getState());
    originator.getStateFromMemento(careTaker.get(0));
    System.out.println("First saved State: " + originator.getState());
    originator.getStateFromMemento(careTaker.get(1));
    System.out.println("Second saved State: " + originator.getState());
}
```

7）观察者模式

- **核心思想**：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法，**Subject中有一个List存放Observer对象。**
- **适用情况**：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。
- **代码示例**：

```
/** 负责存放观察者以及通知 */
public class Subject {
    private List<Observer> observers = new ArrayList<Observer>();
```

```
    private int state;
    public int getState() {
        return state;
    }
    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }
    public void attach(Observer observer){
        observers.add(observer);
    }
    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
/** 观察者抽象类与实现类 */
public abstract class Observer {
    public abstract void update();
}
public class BinaryObserver extends Observer{
    @Override
    public void update() {
        System.out.println( "Binary String: "
            + Integer.toBinaryString( subject.getState() ) );
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Subject subject = new Subject();
    Observer binaryObserver = new BinaryObserver();
    subject.attach(binaryObserver);

    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
}
```

8）状态模式

- **核心思想**：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。**状态属性允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。**
- **适用情况**：代码中包含大量与对象状态有关的条件语句。
- **代码示例**：

```
/** 状态接口与实现类 */
public interface State {
    void start(Context context);
    void close(Context context);
}
public class StartState implements State {
    public void start(Context context) {}
    public void close(Context context) {
        context.setState(new CloseState());//注意状态的切换
```

```
        System.out.println("close State");
    }
}
public class CloseState implements State {
    public void start(Context context) {
        context.setState(new StartState());//注意状态的切换
        System.out.println("start State");
    }
    public void close(Context context) {}
}
/** 带有状态的实体类 */
public class Context {
    private State state;
    // 省略get/setState方法
    public void start() {
        getState().start(this);
    }
    public void close() {
        getState().close(this);
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Context context = new Context();
    // 设定初始状态为关闭状态
    CloseState closeState = new CloseState();
    context.setState(closeState);

    context.close();    // 当前为关闭状态，关闭操作生效
    context.start();    // 当前为关闭状态，开启操作无效，关闭->开启
    context.start();    // 当前为开启状态，开启操作无效
    context.close();    // 当前为开启状态，关闭操作生效，开启->关闭
}
```

9）空对象模式

- **核心思想**：在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。
- **适用情况**：希望用空对象来替代NULL对象实例的检查
- **代码示例**：

```
/** 抽象客户类与实现类、空对象类 */
public abstract class AbstractCustomer {
    protected String name;
    public abstract boolean isNil();
    public abstract String getName();
}
public class RealCustomer extends AbstractCustomer {
    public RealCustomer(String name) {
        this.name = name;
    }
    @Override
    public String getName() {
        return name;
    }
    @Override
```

```
    public boolean isNil() {
        return false;
    }
}
public class NullCustomer extends AbstractCustomer {
    @Override
    public String getName() {
        return "Not Available in Customer Database";
    }
    @Override
    public boolean isNil() {
        return true;
    }
}
/** 抽象类工厂 */
public class CustomerFactory {
    public static final String[] names = {"Rob", "Joe", "Julie"};
    public static AbstractCustomer getCustomer(String name){
        for (int i = 0; i < names.length; i++) {
            if (names[i].equalsIgnoreCase(name)){
                return new RealCustomer(name);
            }
        }
        return new NullCustomer();
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");
    AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");
    AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");
    AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");
    System.out.println(customer1.getName());
    System.out.println(customer2.getName());
    System.out.println(customer3.getName());
    System.out.println(customer4.getName());
}
```

10）策略模式

- **核心思想**：在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。
- **适用情况**：如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为；一个系统需要动态地在几种算法中选择一种；如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。
- **与状态模式区别**：
    - **状态模式是通过状态转移（指 Context 在运行过程中由于一些条件发生改变而使得 State 对象发生改变，注意必须要是在运行过程中）来改变 Context 所组合的 State 对象，而策略模式是通过 Context 本身的决策来改变组合的 Strategy 对象。**
    - **状态模式主要是用来解决状态转移的问题，当状态发生转移了，那么 Context 对象就会改变它的行为；而策略模式主要是用来封装一组可以互相替代的算法族，并且可以根据需要动态地去替换 Context 使用的算法。**

- **代码示例**：

```
/** 策略接口与实现类 */
public interface Strategy {
    public int doOperation(int num1, int num2);
}
public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}
public class OperationSubtract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}
/** 根据策略修改行为的实体类 */
public class Context {
    private Strategy strategy;
    public SetStrategy(Strategy strategy){
        this.strategy = strategy;
    }
    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Context context = new Context();
    context.SetStrategy(new OperationAdd());
    System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
    context.SetStrategy(new OperationSubtract());
    System.out.println("10 - 5 = " + context.executeStrategy(10, 5));
}
```

11）模板模式

- **核心思想**：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。**为防止恶意操作，一般模板方法都加上 final 关键词。**
- **适用情况**：通用方法，但内部实现步骤每个子类稍有不同。
- **代码示例**：

```
/** 抽象类与实现类 */
public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //模板
    public final void play(){
        initialize();
        startPlay();
        endPlay();
```

```
    }
}
public class Cricket extends Game {
    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }
    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }
    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}
public class Football extends Game {
    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }
    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }
    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    Game game = new Cricket();
    game.play();
    System.out.println();
    game = new Football();
    game.play();
}
```

12）访问者模式

- **核心思想**：主要将数据结构与数据操作分离。在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。
- **适用情况**：对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作；需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。
- **代码示例**：

```
/** 元素接口与实现类 */
public interface ComputerPart {
    void accept(ComputerPartVisitor computerPartVisitor);
}
public class Keyboard implements ComputerPart {
    @Override
    // 操作由访问者实现
```

```
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}
public class Mouse implements ComputerPart {
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}
public class Computer implements ComputerPart {
    ComputerPart[] parts;
    public Computer(){
        parts = new ComputerPart[] {new Mouse(), new Keyboard()};
    }
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        for (int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
        computerPartVisitor.visit(this);
    }
}
/** 访问者接口与实现类 */
public interface ComputerPartVisitor {
    void visit(Computer computer);
    void visit(Mouse mouse);
    void visit(Keyboard keyboard);
}
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {
    @Override
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }
    @Override
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }
    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }
}
/** 客户端调用方式 */
public static void main(String[] args){
    ComputerPart computer = new Computer();
    computer.accept(new ComputerPartDisplayVisitor());
}
```