Homework：

1、 **试用 Bridge 模式完成下列事情：饮料的杯子有大、中、小；行为有：加奶，加糖，啥都不加。**

答：使用桥接模式（Bridge Pattern）可以分离抽象和实现，使得两者可以独立地变化。Drink 类充当抽象部分。它包含一个 Additive 类型的引用，这个引用是一个接口，表明饮料可以添加的成分。Drink 类自身是抽象的，定义了 serve() 方法的框架，但不实现具体的加料逻辑。

```
abstract class Drink {
    4 个用法
    protected Additive additive;
    3 个用法
    public Drink(Additive additive) {
        this.additive = additive;
    }
    3 个用法  3 个实现
    abstract void serve();
}

1 个用法
class SmallDrink extends Drink {
    1 个用法
    public SmallDrink(Additive additive) {
        super(additive);
    }

    3 个用法
    public void serve() {
        System.out.print("饮料-小杯-");
        additive.add();
    }
}

1 个用法
class MediumDrink extends Drink {
    1 个用法
    public MediumDrink(Additive additive) {
        super(additive);
    }

    3 个用法
    public void serve() {
        System.out.print("饮料-中杯-");
        additive.add();
    }
}

1 个用法
class LargeDrink extends Drink {
    1 个用法
    public LargeDrink(Additive additive) {
        super(additive);
    }

    3 个用法
    public void serve() {
        System.out.print("饮料-大杯-");
        additive.add();
    }
}
```

Additive 接口及其实现类（Milk,Sugar,NoAdditive）充当实现部分。这些类实现了 Additive 接口，具体定义了可以添加哪些配料。

```
interface Additive {
    3 个用法  3 个实现
    void add();
}

1 个用法
class Milk implements Additive {
    3 个用法
    public void add() {
        System.out.println("加奶");
    }
}

1 个用法
class Sugar implements Additive {
    3 个用法
    public void add() {
        System.out.println("加糖");
    }
}

1 个用法
class NoAdditive implements Additive {
    3 个用法
    public void add() {
        System.out.println("无额外添加");
    }
}
```

在 main 方法中使用并测试桥接模型：

```
public class Main {
    0 个用法
    public static void main(String[] args) {
        Drink smallWithSugar = new SmallDrink(new Sugar());
        smallWithSugar.serve();
        Drink mediumWithMilk = new MediumDrink(new Milk());
        mediumWithMilk.serve();
        Drink largeNoAdd = new LargeDrink(new NoAdditive());
        largeNoAdd.serve();
    }
}
```

运行结果如下：

可以发现，程序中饮料的大小和加料的种类可以独立进行修改和扩展，而不会相互影响。例如，如果需要添加一种新的饮料大小（比如超大杯），只需扩展 Drink 类。同样，若需添加新的配料（如加蜂蜜），只需实现 Additive 接口。在运行时，可以灵活地组合任意大小的饮料与任意类型的添加剂，这是通过在 Drink 的构造函数中传递不同的 Additive 实例实现的。这种组合是动态的，提供了极高的灵活性。

2、 附录

1) Additive

```
interface Additive {
    void add();
}
```

2) Milk

```
class Milk implements Additive {
    public void add() {
        System.out.println("加奶");
    }
}
```

3) Sugar

```
class Sugar implements Additive {
    public void add() {
        System.out.println("加糖");
    }
}
```

4) NoAdditive

```
class NoAdditive implements Additive {
    public void add() {
        System.out.println("无额外添加");
    }
}
```

5) Drink

```
abstract class Drink {
    protected Additive additive;
    public Drink(Additive additive) {
        this.additive = additive;
    }
    abstract void serve();
}
```

6) SmallDrink

```
class SmallDrink extends Drink {
    public SmallDrink(Additive additive) {
        super(additive);
    }

    public void serve() {
        System.out.print("饮料-小杯-");
        additive.add();
    }
}
```

7) MediumDrink

```
class MediumDrink extends Drink {
    public MediumDrink(Additive additive) {
        super(additive);
    }

    public void serve() {
        System.out.print("饮料-中杯-");
        additive.add();
    }
}
```

8) LargeDrink

```
class LargeDrink extends Drink {
    public LargeDrink(Additive additive) {
        super(additive);
    }

    public void serve() {
        System.out.print("饮料-大杯-");
        additive.add();
    }
}
```

9) Main

```
public class Main {
    public static void main(String[] args) {
        Drink smallWithSugar = new SmallDrink(new Sugar());
        smallWithSugar.serve();
        Drink mediumWithMilk = new MediumDrink(new Milk());
```

```
        mediumWithMilk.serve();
        Drink largeNoAdd = new LargeDrink(new NoAdditive());
        largeNoAdd.serve();
    }
}
```