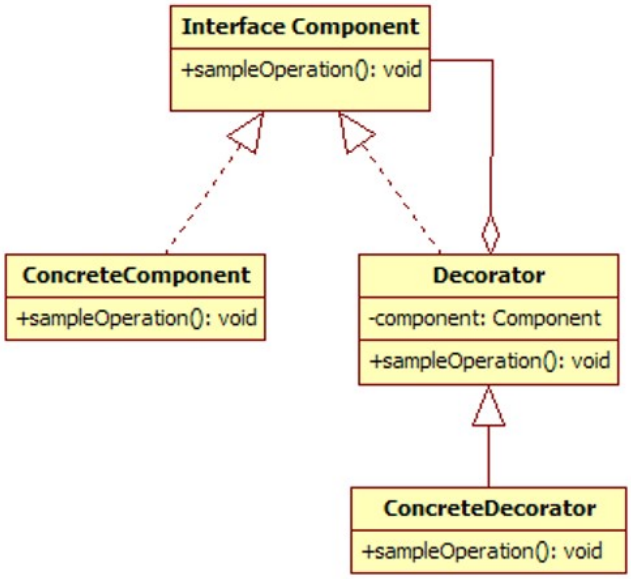
1 装饰模式 (Decoration)

装饰模式又称包装（Wrapper）模式。装饰模式以对客户端透明的方式扩展对象的功能。是继承关系的一种替代方案。

1.1 装饰模式的结构

装饰模式使用装饰类的一个子类的实例，把客户端的调用委派到其它装饰类或具体构件类。装饰模式的关键在于这种扩展是完全透明的。



模式中的角色:

- 抽象构件（Component）角色：给出一个接口，以规范准备接收附加责任（扩展功能）的对象。
- 具体构件（Concrete Component）角色：定义一个准备接收附加责任的类。
- 装饰（Decorator）角色：持有一个构件的实例，并定义与抽象构件接口一致的接口。
- 具体装饰（Concrete Decorator）角色：负责给构件对象“贴上”附加的责任。

下面给出装饰模式的示意性源代码。首先是抽象构件角色的源代码:

```
1 public interface Component {
2   void sampleOperation(); // 某业务逻辑方法
3 }
```

下面是装饰角色的源代码:

```
1 public class Decorator implements Component {
2   private Component component;
3   public Decorator(Component component) {
4     this.component = component;
```

```
  }
  /* 实现业务方法，委派给构件*/
  public void sampleOperation() { component.sampleOperation(); }
}
```

应当指出的有以下几点：

- 在上面的装饰类里，有一个私有的属性 component，其数据类型是构件 (Component)。
- 此装饰类实现了构件 (Component) 接口。
- 接口的实现方法也值得注意，每一个实现的方法都是委派给私有的属性 component 对象，但并仅是不单纯的委派，而是将有功能的增强。

虽然 Decorator 类不是一个抽象类，在实际应用中也不一定是抽象类，但是由于他的功能是一个抽象角色，因此也常常称它为抽象装饰。

定义中的具体构件类的示意性源代码:

```
public class ConcreteComponent implements Component {
  public ConcreteComponent() { /* Write your code here */ }
  public void sampleOperation() { /* Write your code here */ }
}
```

具体装饰类实现了抽象装饰类所声明的 sampleOperation() 方法:

```
public class ConcreteDecorator extends Decorator {
  public void sampleOperation() {
    //此处可写功能增强的代码
    super.sampleOperation();
    //或在此处写功能增强的代码
  }
}
```

使用装饰模式的优点和缺点:

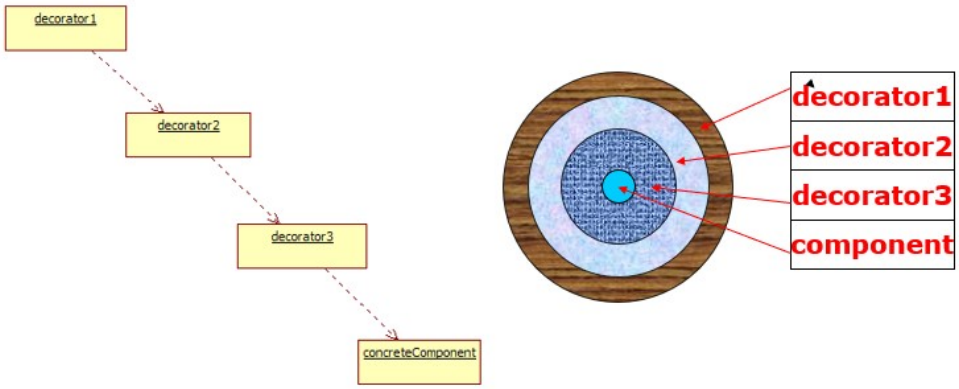
- 优点:
  1. 装饰模式与继承关系的目的都是要扩展对象的功能, 但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态地决定 “贴上” 一个需要的 “装饰”，或者除掉一个不需要的 “装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。
  2. 通过使用不同的具体装饰类以及这些装饰类的排列组合, 可以创造出很多不同的行为的组合。
- 缺点: 产生出较多的对象; 比继承更易出错.

对象图: 装饰模式的对象图呈链状结构，假设共有三个具体装饰类，分别称为 Decorator1, Decorator2 和 Decorator3，具体构件类是 ConcreteComponent。

一个典型的创建过程:

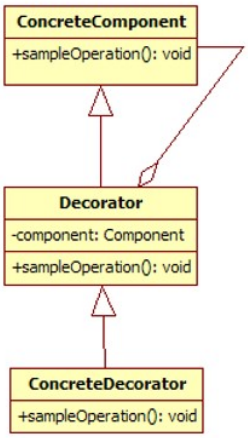
```
new Decorator1(new Decorator2(new Decorator3(new ConcreteComponent())));
```

这就意味着 Decorator1 的对象持有一个对 Decorator2 对象的引用，后者则持有一个对 Decorator3 对象的引用，再后者持有一个对具体构件 ConcreteComponent 对象的引用，这种链式的引用关系使装饰模式看上去像是一个 LinkedList，如图所示。

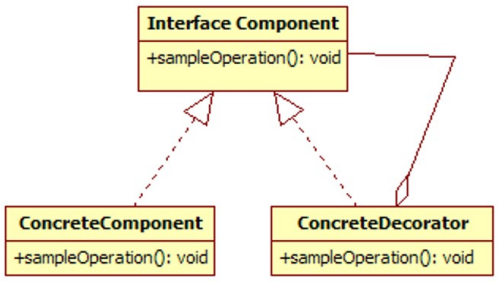


在具体实现时，要注意以下几点：

- 一个装饰类的接口必须与被装饰类的接口相容；
- 尽量保持 Component 作为一个“轻”类；
- 如果 35 有一个 ConcreteComponent 类，而没有抽象的 Component 类，则可以把 Decorator 作为 ConcreteComponent 的子类；如图：



- 若只有一个 ConcreteDecorator 类，则没必要建立一个单独的 Decorator 类。如图：



透明性的要求：不能向客户端“暴露”装饰对象的具体类型，如下列代码所示：

```
1 Component c=new ConcreteComponent();
2 Component c1=new ConcreteDecorator1(c);
3 Component c2=new ConcreteDecorator2(c1);
```

而下面的做法是不对的:

`ConcreteDecorator1 c1=new ConcreteDecorator1();`

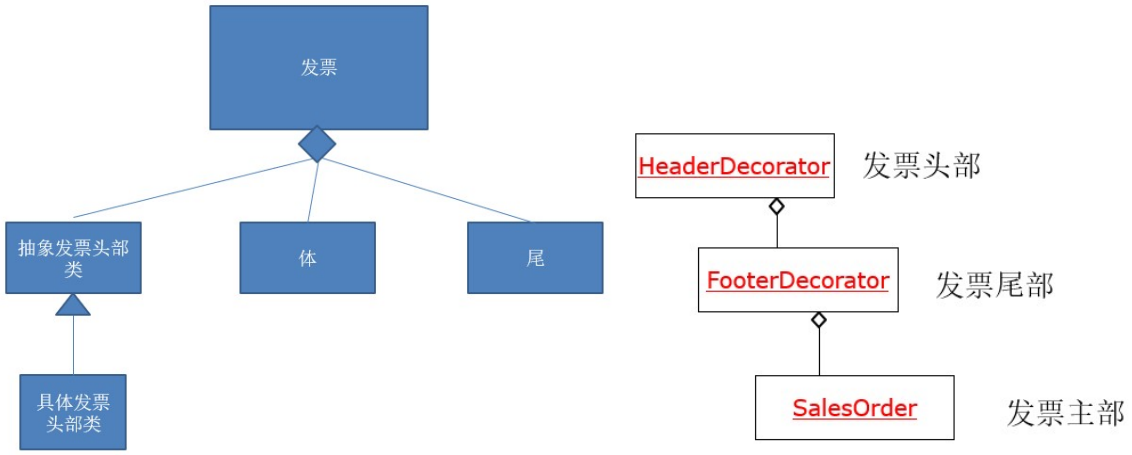
这就是前面所说的, 装饰模式对客户端是完全透明的含义. 因此，ConcreteDecorator 里不能有 Component 类中没有的公有方法.

1.2 实例: 发票系统

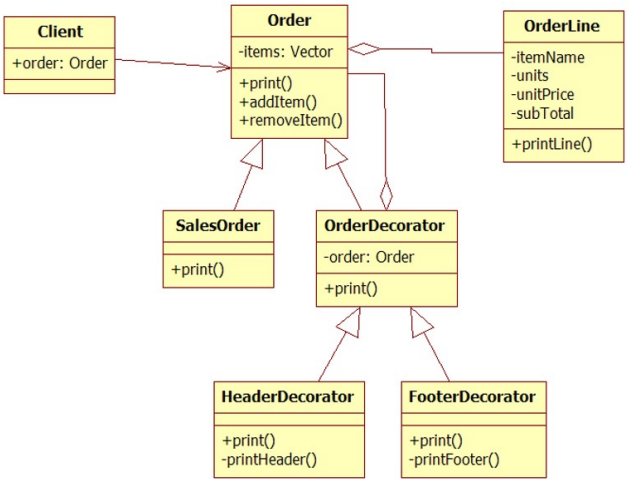
要求: 有一个电子销售系统需要打印出顾客的购物发票。一张发票可以分为三个部分：

- 发票头部（Header）：上面有顾客的名字，销售的日期；
- 发票主部：销售的货物清单，包括商品名字、购买的数量、单价、小计；
- 发票尾部：总金额

发票的头部、尾部可能有多种形式；要求系统能以灵活的方式更换头、尾，可以灵活地组合头部尾部，形成新的发票。



发票系统的类图:




```
1  // 抽象发票类
2  abstract public class Order {
3    private Vector<OrderLine> items = new Vector();
4    public void print() {
5      for (int i = 0; i < items.size(); i++) {
6        OrderLine item = (OrderLine)items.get(i);
7        item.printLine();
8      }
9    }
10   // ...
11 }
```

```
1  // 具体发票主部类
2  public class SalesOrder extends Order {
3    public void print() { super.print(); }
4    // ...
5  }
```

```
1  // 抽象装饰类
2  abstract public class OrderDecorator extends Order {
3    protected Order order;
4    public OrderDecorator(Order order) { this.order = order; }
5    // ...
6  }
```

```
1  // 具体装饰类—发票头部
2  abstract public class HeaderDecorator extends OrderDecorator {
3    private String cusName;
```

```
  private Date date;
  public HeaderDecorator(Order order) { super(order); }
  public void print() {
    printHeader();
    order.print();
  }
  private void printHeader() {
    out.println("发票头……..");
    out.println("顾客名: " + cusName + ";购物日期: " + date);
    // ...
  }
  // ...
}
```

```
// 具体装饰类—发票尾部
public class FooterDecorator extends OrderDecorator {
  public FooterDecorator(Order order) { super(order); }
  public void print() {
    order.print();
    printFooter();
  }
  private void printFooter() {
    out.println("发票尾……..");
    // ...
  }
  // ...
}
```

```
// 客户端类
public class Clint {
  private Order order;
  public static void main(String args[]) {
    order=new SalesOrder();
    OrderLine line1=new OrderLine();
    line1.setName("毛巾");
    line1.setUnits(1);
    line1.setPrice(10);
    order.addItem(line1);
    order=new HeaderDecorator(new FooterDecorator(order));
    order.print();
```

```
13    }
14  }
```