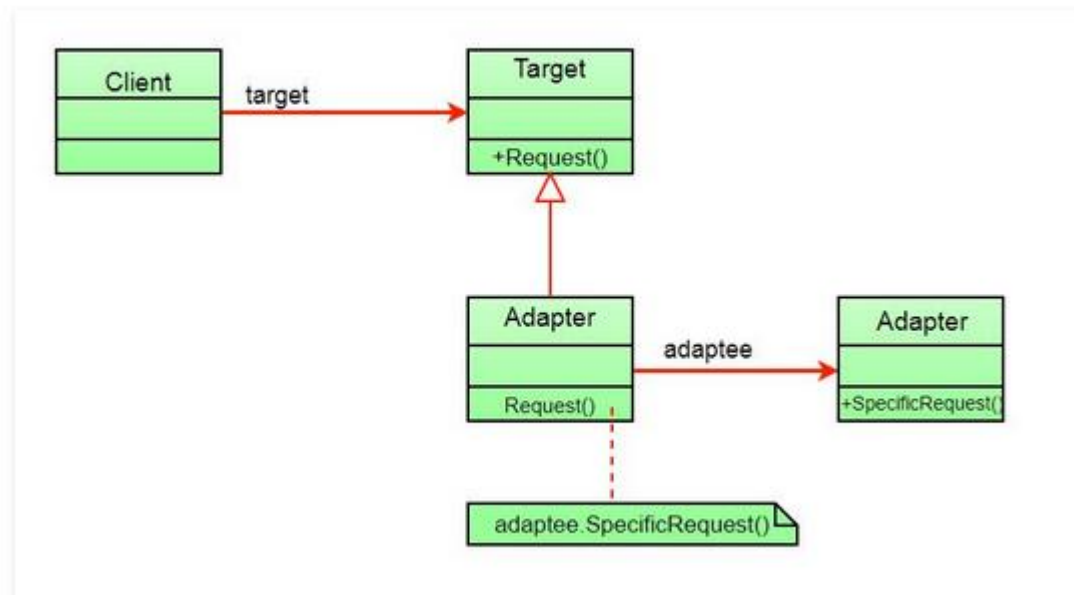


1. Adapter Pattern:

Why Adapter Pattern?

The adapter pattern is a [great pattern for connecting new code to legacy code without having to change the working contract that was produced from the legacy code originally.](#)

Class Diagram:



Contrast

Adapter

- Works after code is designed
- Legacy
- Retrofitted
- Provides different interface

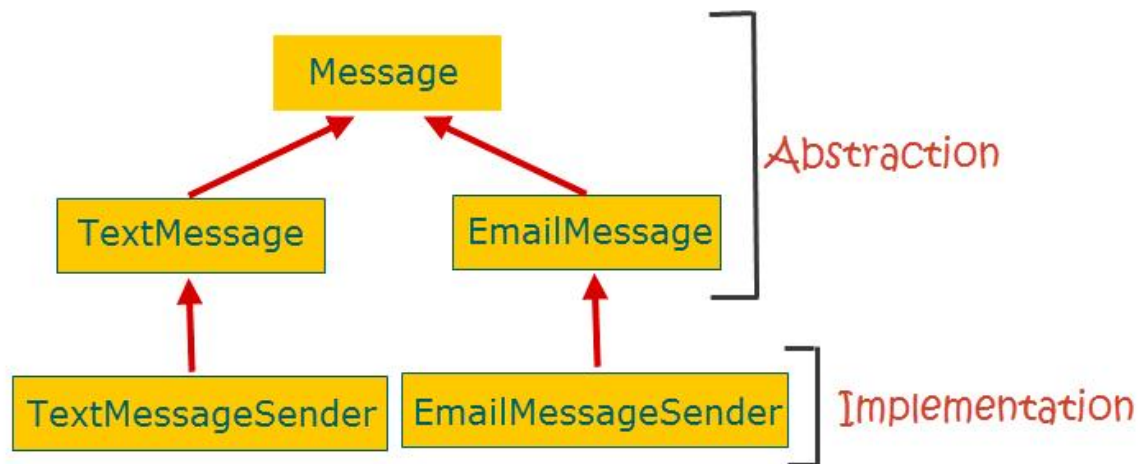
Bridge

- Designed upfront
- Abstraction and implementation vary
- Built in advance
- Both adapt multiple systems

2. Bridge Pattern

[The bridge pattern is very similar to the adapter with the main difference being that the bridge works with new code whereas the adapter works with legacy code. Decouples an abstraction so two classes can vary independently.](#)

To understand how the bridge pattern works, consider a messaging application that clients can use to send different types of messages, such as a text or an email message. The most intuitive approach is to first create an interface or an abstract base class, `Message`. Next, we create the derived classes: `TextMessage` and `EmailMessage`. Finally, to send messages, we create two message sender classes: `TextMessageSender` that extends `TextMessage` and `EmailMessageSender` that extends `EmailMessage`. This is how our inheritance hierarchy looks like.

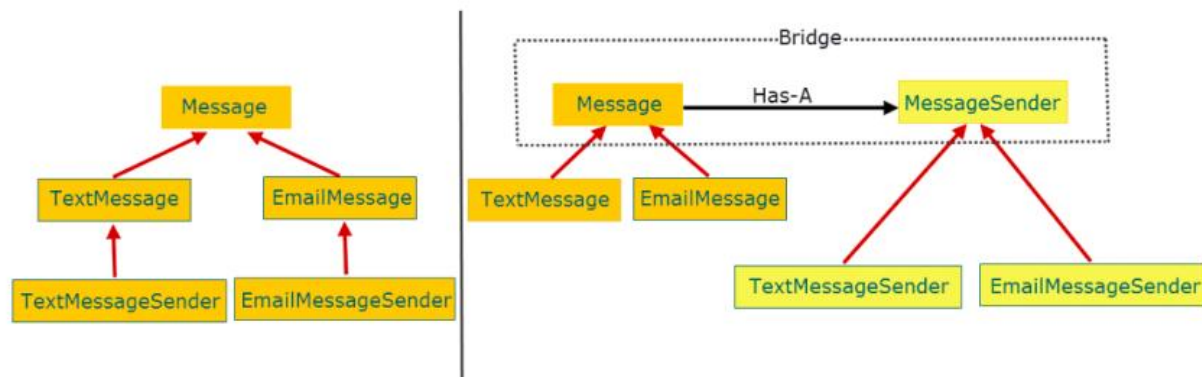


At first sight there appears nothing wrong in the design above. But if you look deep you will notice that, the abstraction part- the part that clients interact with, and the implementation part- the part that provides the core functionality of sending messages, are tightly integrated. Our design relies on inheritance and one inherent disadvantage is that it breaks encapsulation. As a developer of the `EmailMessageSender` subclass, you have to know about the internals of the `EmailMessage` superclass, which means the encapsulation in the superclass is broken.

Our design is also fragile. As an example, if we change the implementation to allow clients to optionally encrypt message before sending, we will need to update the abstraction part to make the encryption functionality available to clients.

Another issue is reusability. If we want to reuse only the implementation (message sending) part in some other application, we will have to take along the abstraction part as extra baggage.

The bridge pattern addresses all such issues by separating the abstraction and implementation into two class hierarchies. This figure shows the design without and with the bridge pattern.



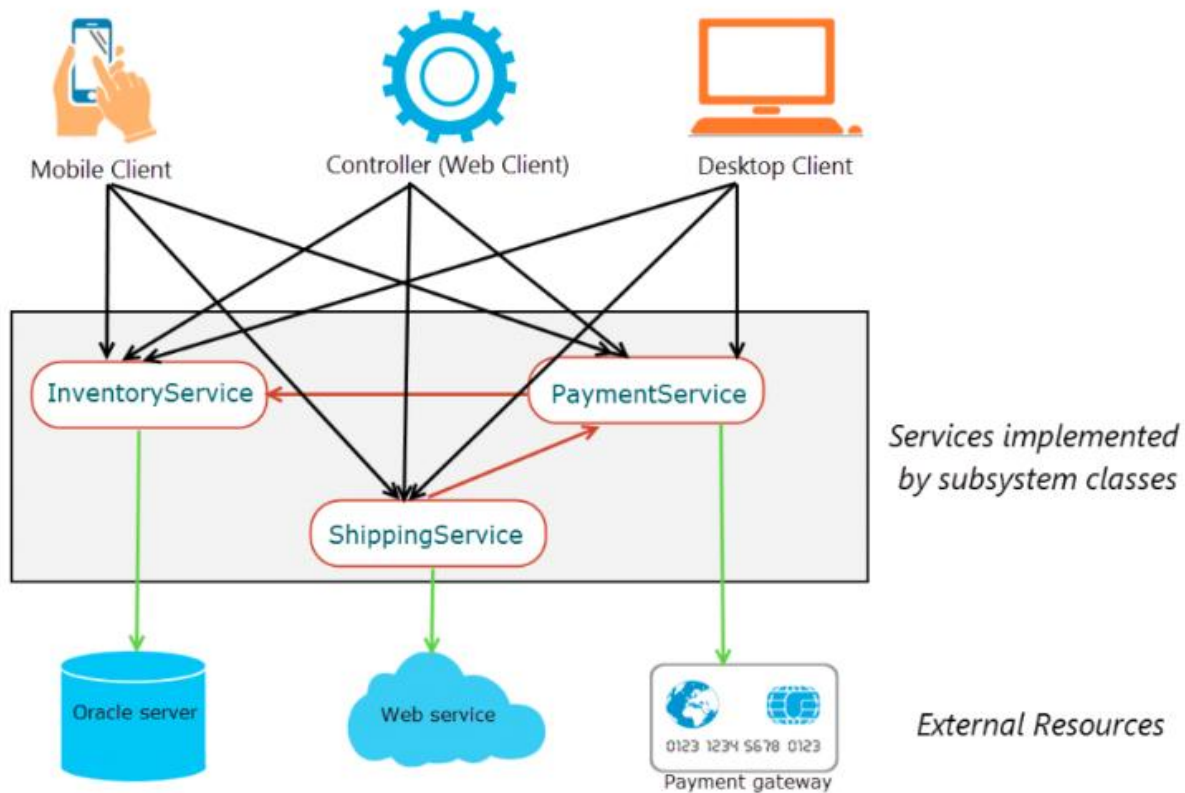
With the bridge pattern, the abstraction maintains a **Has-A** relationship with the implementation instead of a **IS-A** relationship. The **Has-A** relationship is achieved through composition where the abstraction maintains a reference of the implementation and forwards client requests to it.

```
public abstract class Message {
    MessageSender messageSender;
    public Message(MessageSender messageSender){
        this.messageSender=messageSender;
    }
    abstract public void send();
}
```

```
5 public class TextMessage extends Message{
6
7     public TextMessage(MessageSender messageSender){
8         super(messageSender);
9     }
10    @Override
11    public void send(){
12        messageSender.sendMessage();
13    }
14
15 }
```

```
public class EmailMessage extends Message{
    public EmailMessage(MessageSender messageSender){
        super(messageSender);
    }
    @Override
    public void send(){
        messageSender.sendMessage();
    }
}
```

3. Facade Pattern



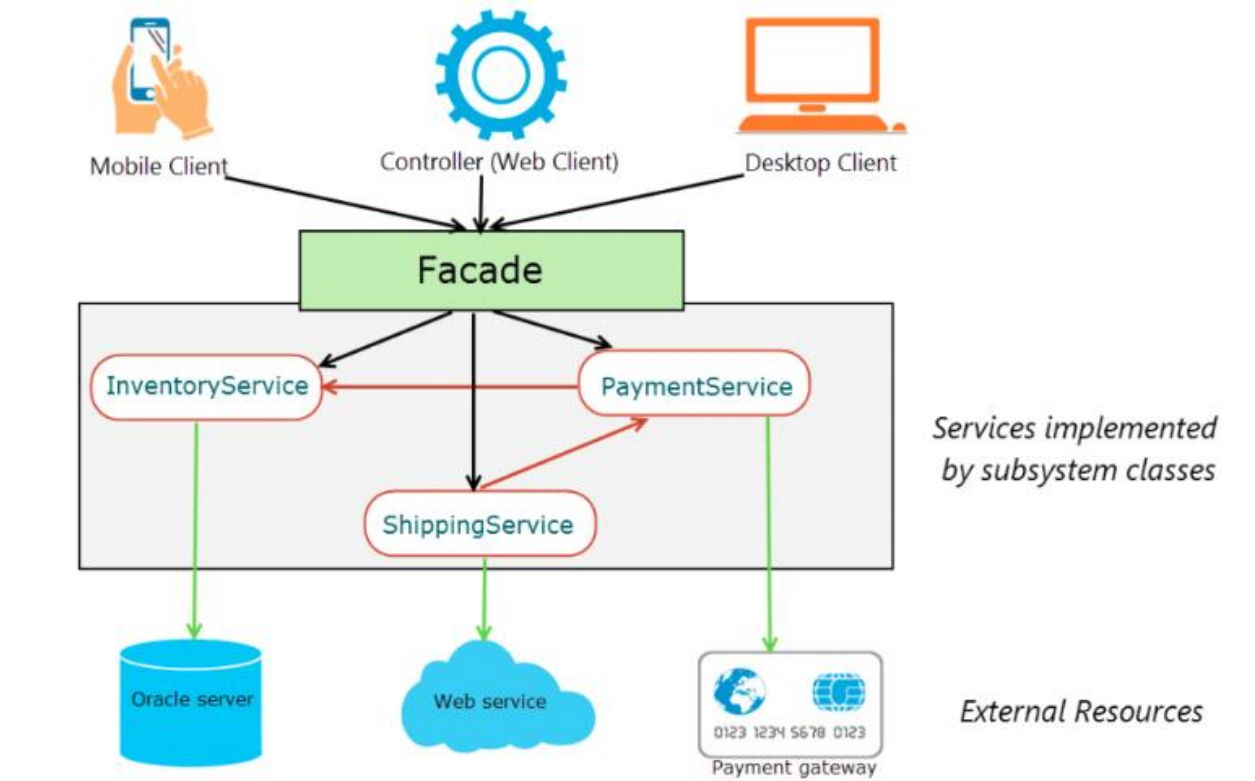
Problem with the above Design:

1. the clients need to make multiple interactions with the services implemented by subsystem classes in order to submit an order fulfillment process of an e-commerce store. It means, our clients are tightly coupled with the subsystem classes – a fundamental violation of the [SOLID design principles](#)
2. Things can get worse if a new **InvoicingService** is introduced in the service layer or the existing **ShippingService** is updated to make the logistic part internal to the organization

Solution:

Rather than having the clients tightly coupled to the subsystems, we need an interface which makes the subsystems easier to use. In our example, our clients just want to place an order. They don't really need to care about dealing with inventory,

shipping or payments. The Facade pattern is a way of providing a simple way for the clients to interact with the subsystems.



- **Facade:** Delegates client requests to appropriate subsystem classes.
- **Subsystem classes:** Implements subsystem functionalities. Subsystem classes are used by the facade, but not the other way around. We will come to it later in this post.
- **Client:** Requests the facade to perform some action.

Applying the Facade Pattern

```
public class Product {
    public int productId;
    public String name;
    public Product(){}
    public Product(int productId, String name){
        this.productId=productId;
        this.name=name;
    }
}
```

```
public class InventoryService {
    public static boolean isAvailable(Product product){
        /*Check Warehouse database for product availability*/
        return true;
    }
}
```

```
public class PaymentService {
    public static boolean makePayment(){
        /*Connect with payment gateway for payment*/
        return true;
    }
}
```

```
public class ShippingService {
    public static void shipProduct(Product product){
        /*Connect with external shipment service to ship product*/
    }
}
```

```
public interface OrderServiceFacade {
    boolean placeOrder(int productId);
}
```

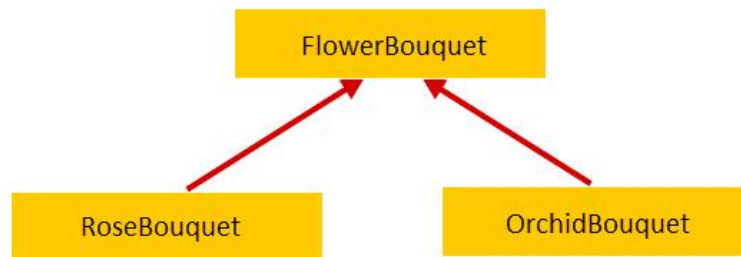
```
public class OrderServiceFacadeImpl implements OrderServiceFacade{

    public boolean placeOrder(int pId){
        boolean orderFulfilled=false;
        Product product=new Product();
        product.productId=pId;
        if(InventoryService.isAvailable(product))
        {
            System.out.println("Product with ID: "+ product.productId+" is available.");
            boolean paymentConfirmed= PaymentService.makePayment();
            if(paymentConfirmed){
                System.out.println("Payment confirmed...");
                ShippingService.shipProduct(product);
                System.out.println("Product shipped...");
                orderFulfilled=true;
            }
        }
        return orderFulfilled;
    }
}
```

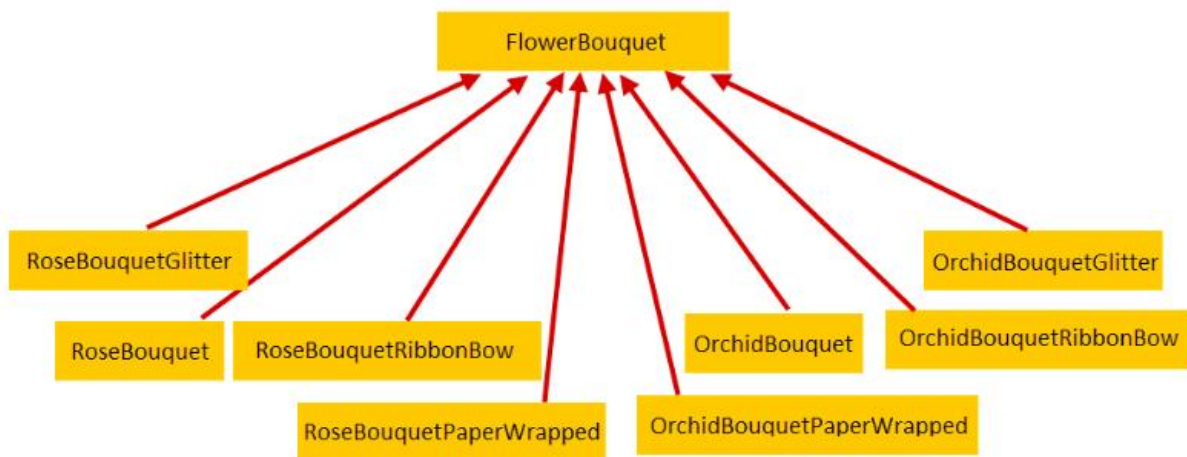
4. Decorator Pattern

Allows for an object's behavior to be extended dynamically at run time

Consider a florist who sells flower bouquets, such as rose bouquet, orchid bouquet, and so on. When a customer walks in, the florist describes the bouquets and tells their prices. We can model the requirements with a **FlowerBouquet** abstract base class and subclasses for specific bouquets.



Customers, in addition to a bouquet, can ask the florist to decorate it with paper wrappings, ribbon bows, and glitters for which the florist charges extra. To address the new requirement, we can add new subclasses of **FlowerBouquet**, one each to represent a bouquet with an additional decoration, and this is how our design looks like now.

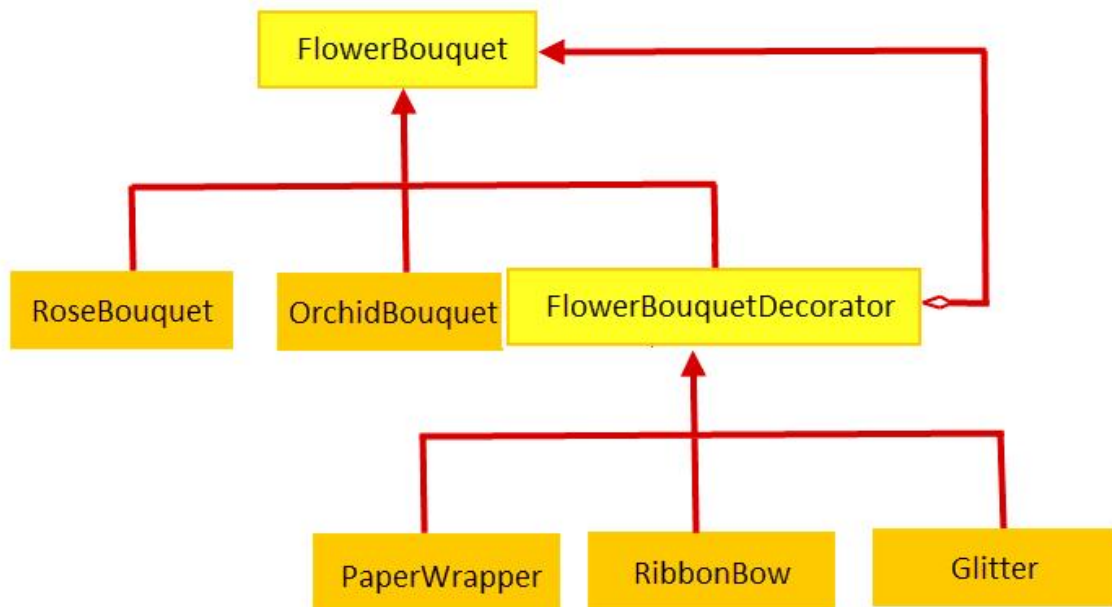


Problems:

- What if we want a rose bouquet with double paper wrap?
- What if we want to add a new lily bouquet?
- What if we want to add new ornamental leaves decoration?

Solution:

Clearly our design is flawed, but this is an ideal use case for the decorator pattern. By using the decorator pattern, we can create a flower bouquet and dynamically decorate it with any numbers of features at run time. Without the flower bouquet knowing it that is “*being decorated*”. For that, we will create an abstract base class **FlowerBouquet** and specific subclasses **RoseBouquet** and **OrchidBouquet** to extend from it. We will also create an abstract **FlowerBouquetDecorator** class to extend **FlowerBouquet**. For each decorator, we will create the **Glitter**, **PaperWrapper**, and **RibbonBow** classes to extend from **FlowerBouquetDecorator**. This is how the design will be.



- **Component (FlowerBouquet)**: Is an abstract base class that can be decorated with responsibilities dynamically.
- **ConcreteComponent (RoseBouquet and OrchidBouquet)**: Are concrete classes that extends **Component** to represent objects to which additional responsibilities can be attached.
- **Decorator (FlowerBouquetDecorator)**: Is an abstract class that extends **Component** and acts as the base class for concrete decorator classes.
- **ConcreteDecorator (PaperWrapper, RibbonBow, and Glitter)**: Are concrete classes that extends **Decorator** to decorate **Components** with responsibilities.


```
public abstract class FlowerBouquet {
    String description;
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
```

```
public class RoseBouquet extends FlowerBouquet{

    public RoseBouquet(){
        description = "Rose bouquet";
    }
    public double cost(){
        return 12.0;
    }
}
```

```
public class OrchidBouquet extends FlowerBouquet{
    public OrchidBouquet(){
        description = "Orchid bouquet";
    }
    public double cost(){
        return 29.0;
    }
}
```

```
public abstract class FlowerBouquetDecorator extends FlowerBouquet {
    public abstract String getDescription();
}
```

```
public class Glitter extends FlowerBouquetDecorator{
    FlowerBouquet flowerBouquet;
    public Glitter(FlowerBouquet flowerBouquet){
        this.flowerBouquet=flowerBouquet;
    }
    public String getDescription(){
        return flowerBouquet.getDescription()+" , glitter";
    }
    public double cost()
    {
        return 4+flowerBouquet.cost();
    }
}
```

```
public class PaperWrapper extends FlowerBouquetDecorator{

    FlowerBouquet flowerBouquet;
    public PaperWrapper(FlowerBouquet flowerBouquet){
        this.flowerBouquet=flowerBouquet;
    }
    public String getDescription(){
        return flowerBouquet.getDescription()+" , paper wrap";
    }
    public double cost()
    {
        return 3+flowerBouquet.cost();
    }
}
```

```

public static void main(String[] args) {
    FlowerBouquet roseBouquet = new RoseBouquet();
    System.out.println(roseBouquet.getDescription()
        + " $ " + roseBouquet.cost());

    /*Rose bouquet with paper wrapper, ribbon bow, and glitter*/
    FlowerBouquet decoratedRoseBouquet = new RoseBouquet();
    FlowerBouquet decoratedRosePaperWrapperBouquet=new PaperWrapper(decoratedRoseBouquet);
    FlowerBouquet decoratedRosePaperWrapperRibbonBowBouquet=new RibbonBow(decoratedRosePaperWrapperBouquet);
    FlowerBouquet decoratedRosePaperWrapperRibbonBowGlitterBouquet=new Glitter(decoratedRosePaperWrapperRibbonBowBouquet);
    System.out.println(decoratedRosePaperWrapperRibbonBowGlitterBouquet.getDescription()
        + " $ " + decoratedRosePaperWrapperRibbonBowGlitterBouquet.cost());

    /*Orchid bouquet with double paper wrapper and ribbon bow*/
    FlowerBouquet decoratedOrchidBouquet = new OrchidBouquet();
    decoratedOrchidBouquet=new PaperWrapper(decoratedOrchidBouquet);
    decoratedOrchidBouquet=new PaperWrapper(decoratedOrchidBouquet);
    decoratedOrchidBouquet=new RibbonBow(decoratedOrchidBouquet);
    System.out.println(decoratedOrchidBouquet.getDescription()
        + " $ " + decoratedOrchidBouquet.cost());
}

```