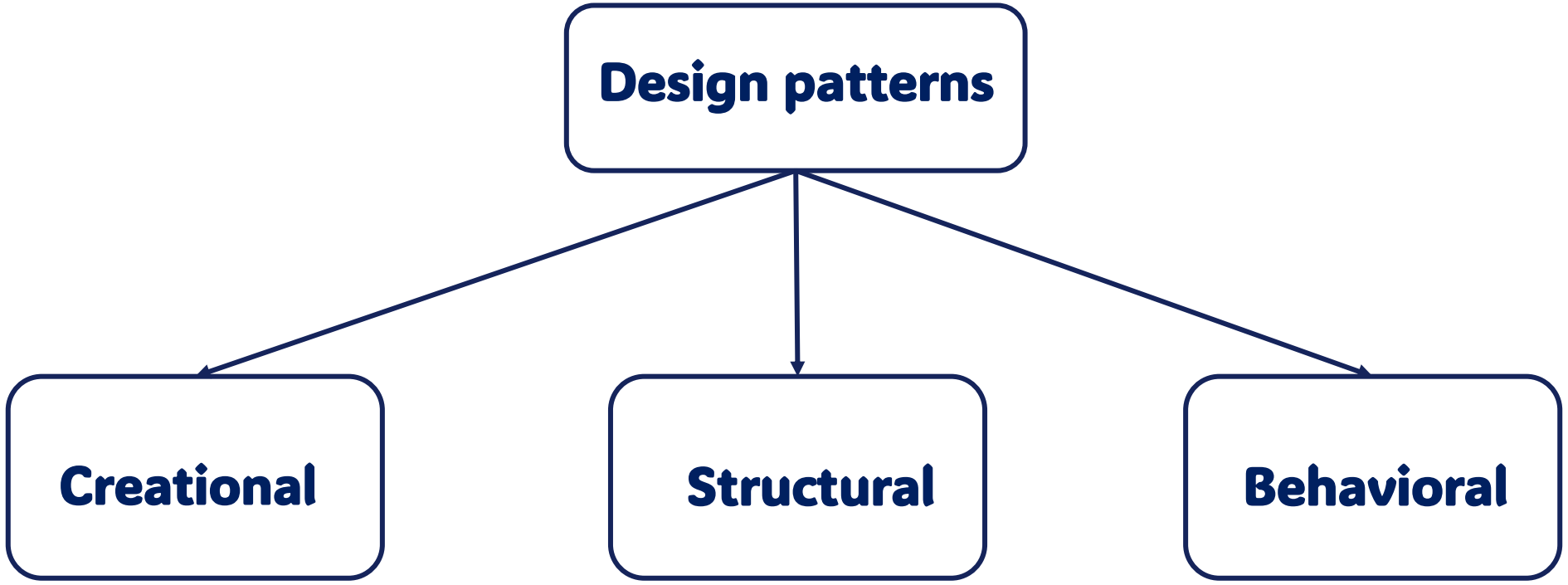
الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Software system design – practical

Lecture 06 – Decorator design pattern

Eng. Raghad al-hossny

Design patterns:



- ***Factory method***
- ***Singelton***

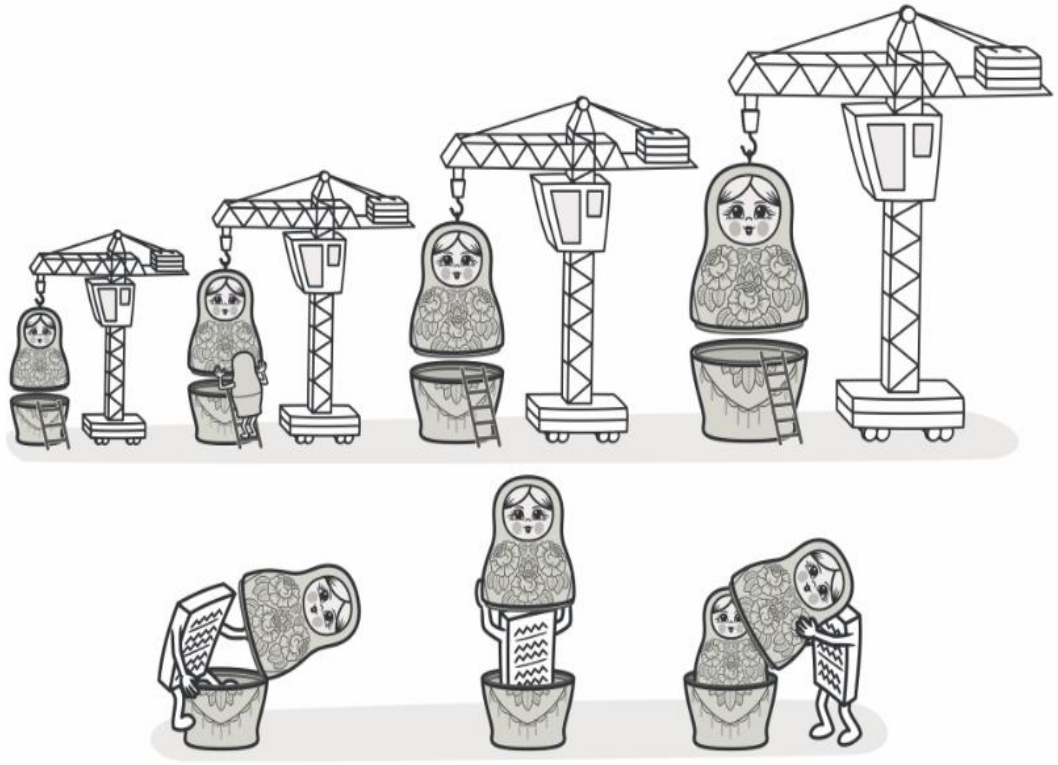
- ***Decorator***

- ***Strategy.***
- ***State.***
- ***Template method.***

Decorator design pattern:

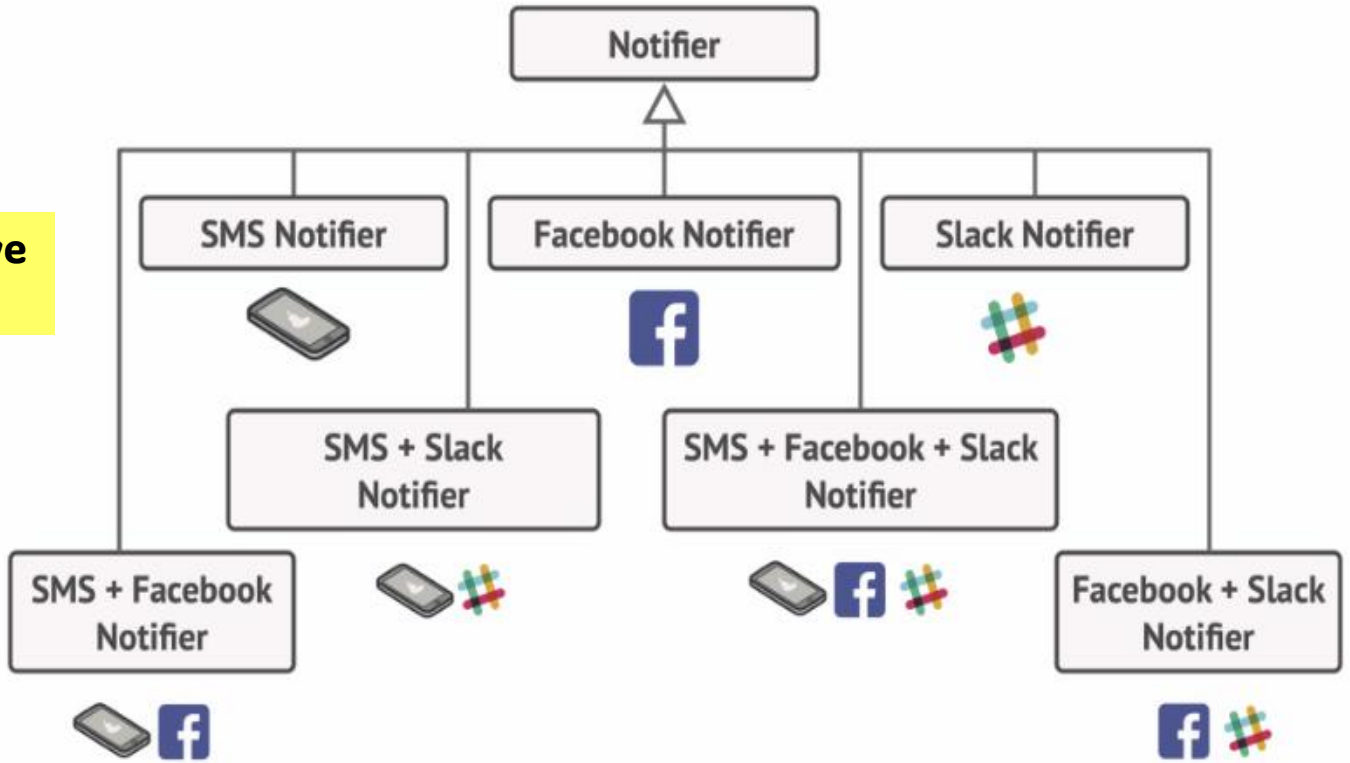
Decorator Design Pattern is a structural pattern that lets you dynamically add behavior to individual objects without changing other objects of the same class.

It uses decorator classes to wrap concrete components, making functionality more flexible and reusable.



Flash back to the notification problem:

1. We can easily solve it using inheritance.



Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.

Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

Flash back to the notification problem:

One of the ways to overcome these caveats is by using Aggregation or Composition instead of Inheritance.

Both of the alternatives work almost the same way: one object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

Inheritance vs. Aggregation

Flash back to the notification problem:

An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work.

Aggregation/composition is the key principle behind many design patterns, including Decorator.

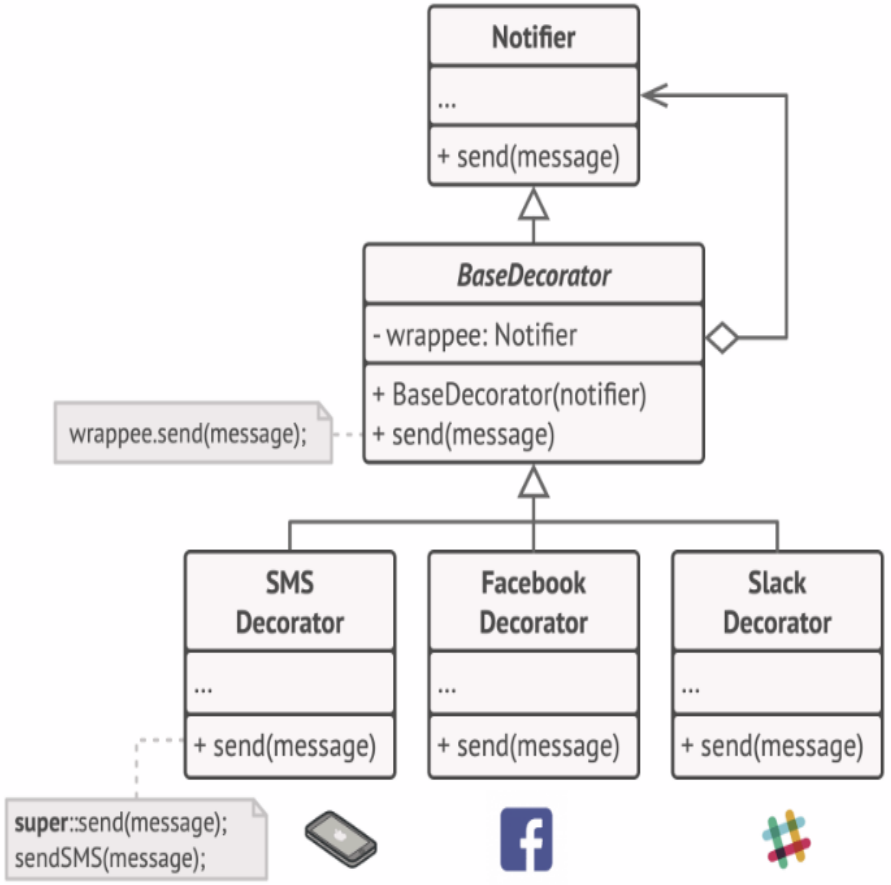
Inheritance vs. Aggregation

Flash back to the notification problem:

"Wrapper" is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern.

A wrapper is an object that can be linked with some target object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives.

However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

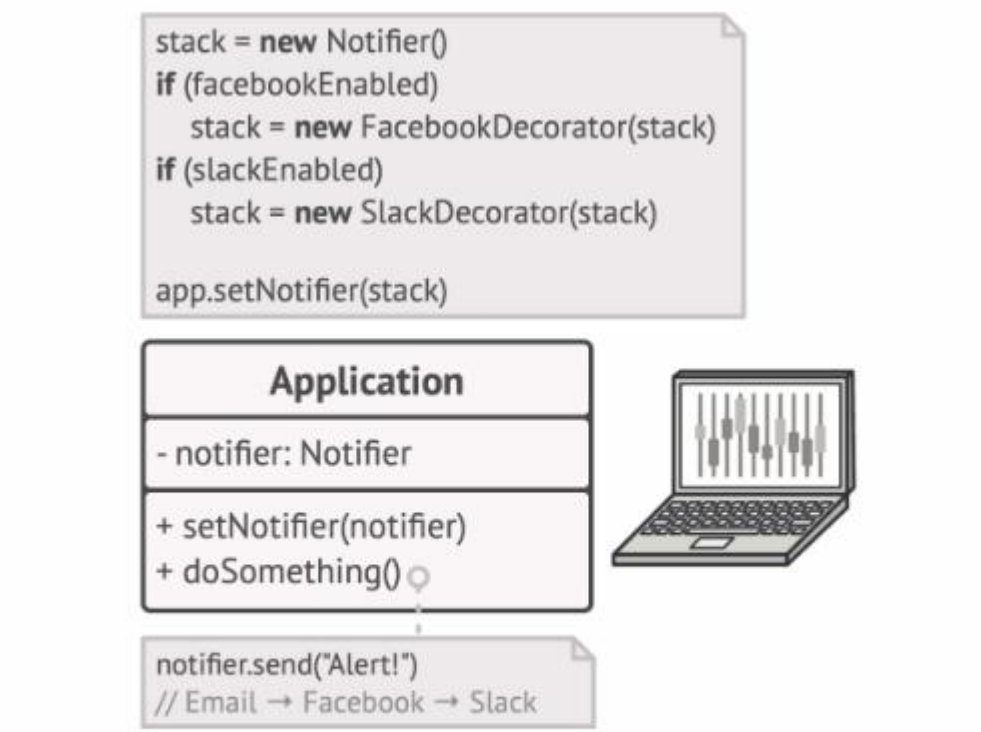


Various notification methods become decorators.

Flash back to the notification problem:

The client code would need to wrap a basic notifier object into a set of decorators that match the client's preferences. The resulting objects will be structured as a stack.

The last decorator in the stack would be the object that the client actually works with. Since all decorators implement the same interface as the base notifier, the rest of the client code won't care whether it works with the "pure" notifier object or the decorated one.



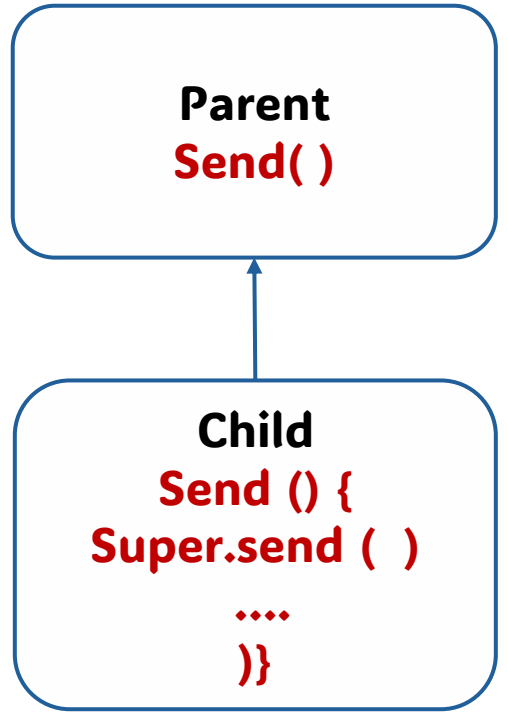
Apps might configure complex stacks of notification decorators.

OOP revision - super key word:

The super keyword is a reference in object-oriented programming that refers to the immediate parent class from a child class.

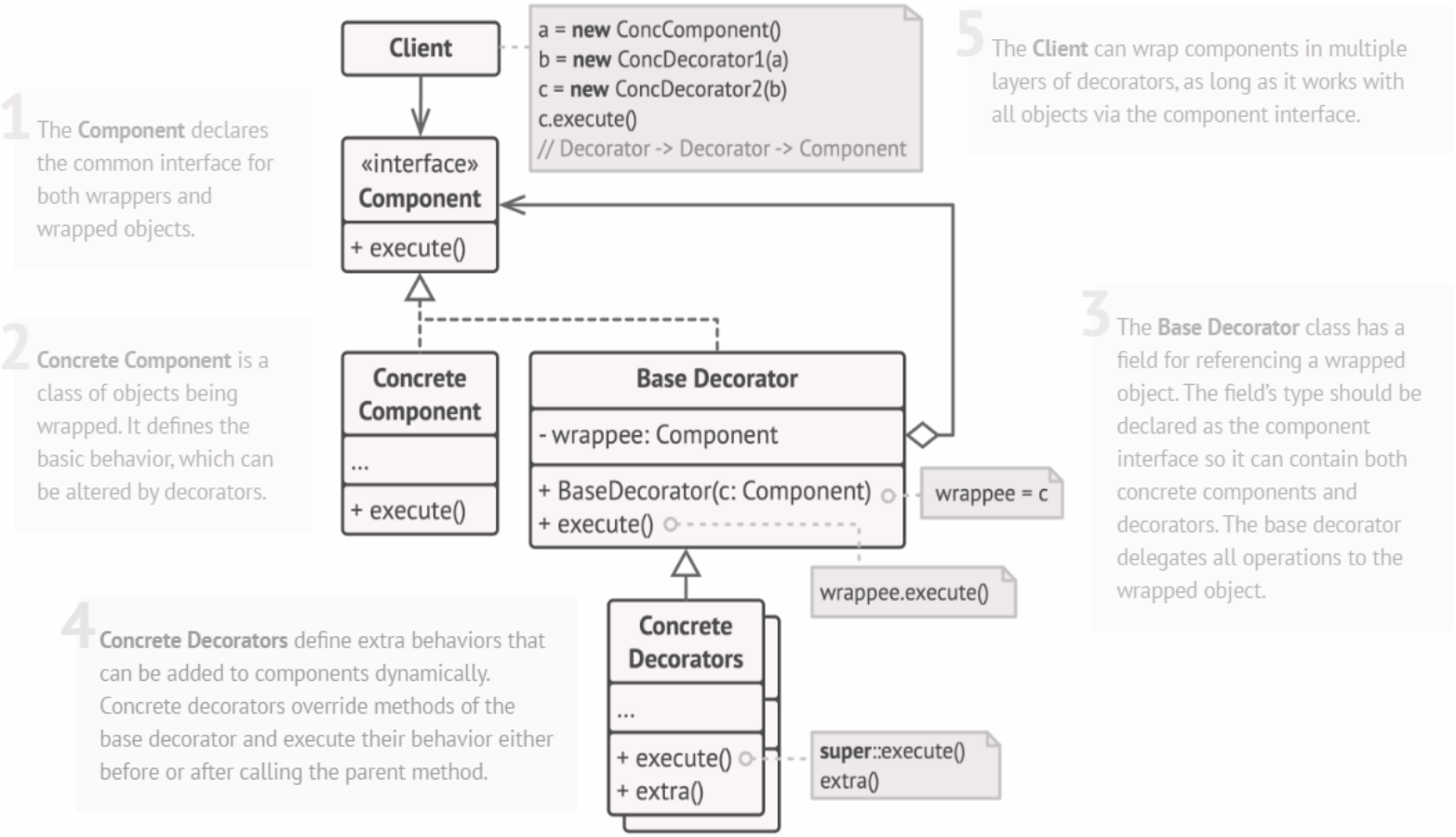
It is used to call the parent's constructor and methods, which helps avoid naming conflicts between parent and child classes.

super is used to call the parent class version of a method when the child overrides it.



Decorator design pattern – final look:

Structure



Implementation:

The main interface:

```
public interface Notifier {

    void send(String message);

}
```

The base behavior we wrap over it:
email notification

```
public class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }

}
```

The base decorator:

```
package decorator;

public class NotifierDecorator implements Notifier{

    protected Notifier wrappee;

    public NotifierDecorator(Notifier wrappee) {
        this.wrappee = wrappee;
    }

    @Override
    public void send(String message) {
        wrappee.send(message);
    }

}
```

Implementation:

Other concrete decorators:

```
package decorator;

public class FacebookNotifier extends NotifierDecorator {

        public FacebookNotifier(Notifier wrappee) {
        super(wrappee);
    }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Sending Facebook Message: " + message);
    }

}
```

```
public class SMSNotifier extends NotifierDecorator{

     public SMSNotifier(Notifier wrappee) {
         super(wrappee);
     }

     @Override
     public void send(String message) {
         super.send(message);
          System.out.println("Sending SMS: " + message);
     }

}
```

```
public class SlackNotifier extends NotifierDecorator  {

      public SlackNotifier(Notifier wrappee) {
        super(wrappee);
     }
     @Override
     public void send(String message) {
         super.send(message);
         System.out.println("Sending Slack Notification: " + message);
     }
```

Implementation:

```
public class Decorator {

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier();

        notifier = new SMSNotifier(notifier);
        notifier = new FacebookNotifier(notifier);
        notifier = new SlackNotifier(notifier);

        notifier.send("Hello design students!");
    }

}
```

The client code:

The final result:

```
run:
Sending Email: Hello design students!
Sending SMS: Hello design students!
Sending Facebook Message: Hello design students!
Sending Slack Notification: Hello design students!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Class activity: Coffee house system & decorator design pattern

You are asked to design a simple Coffee Ordering System using the Decorator Design Pattern. The system should allow a customer to start with a basic coffee and then decorate it by adding extra ingredients such as milk, sugar or whipped cream.

Each added ingredient must increase both the description of the drink and the total cost.

The base drink is a simple Coffee ($3).

Coffee ($3)
→ with Milk (+$1)
→ with sugar (+$2)
→ with Whipped Cream (+$1)

Design the solution using the Decorator Design Pattern, and implement it in Java to print the final coffee description and price.