Mẫu thiết kế - 2
(Design Patterns)

Nội dung

- Abstract Factory
- Adapter
- Decorator
- Một số nguyên lý thiết kế

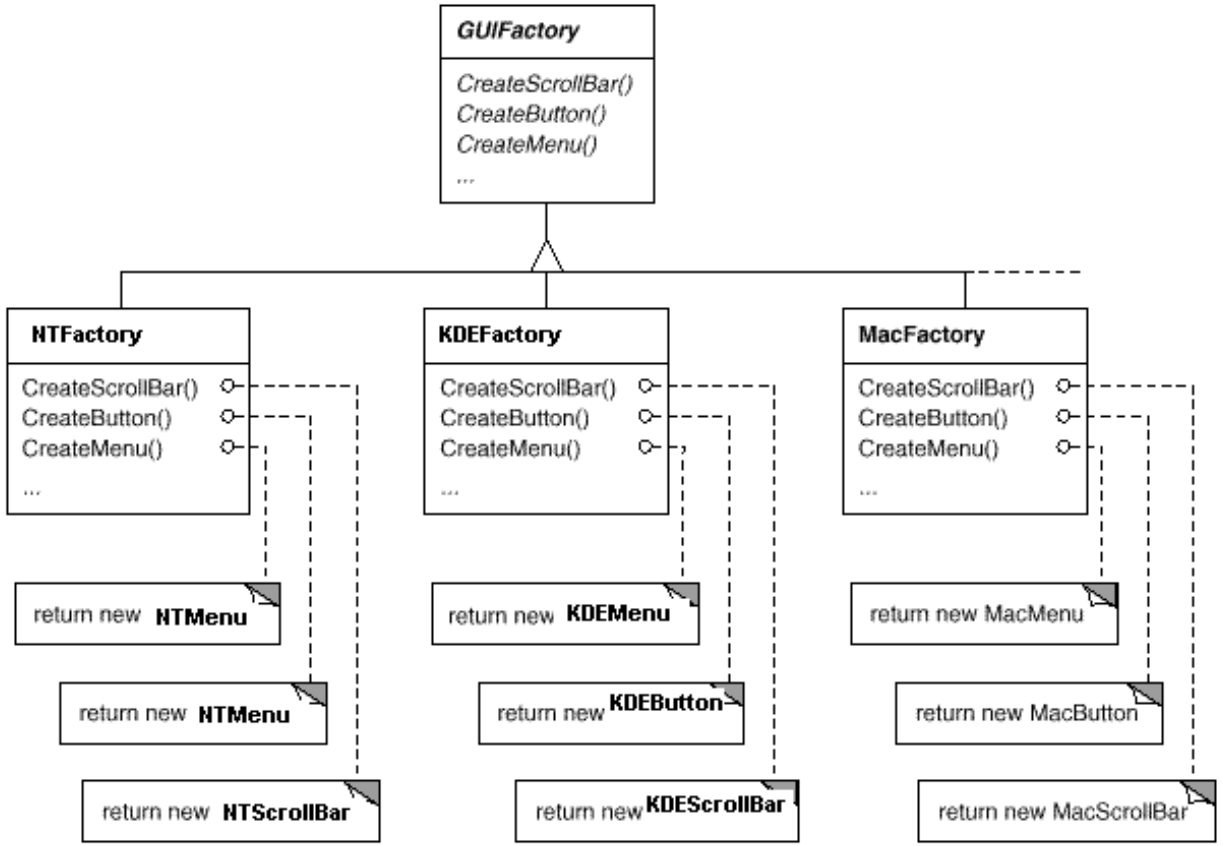
Abstract Factory

- Vấn đề: có nhiều "dòng" (họ các đối tượng được tạo đi kèm nhau); hệ thống cho phép ứng dụng tùy biện tạo "dòng" các đối tượng (tùy biến)
- Ngữ cảnh
  - Client độc lập với cách các đối tượng (product) được tạo ra
  - Client được cấu hình với một trong nhiều "dòng" đối tượng
  - Các đối tượng thuộc cùng dòng sẽ được sử dụng cùng nhau

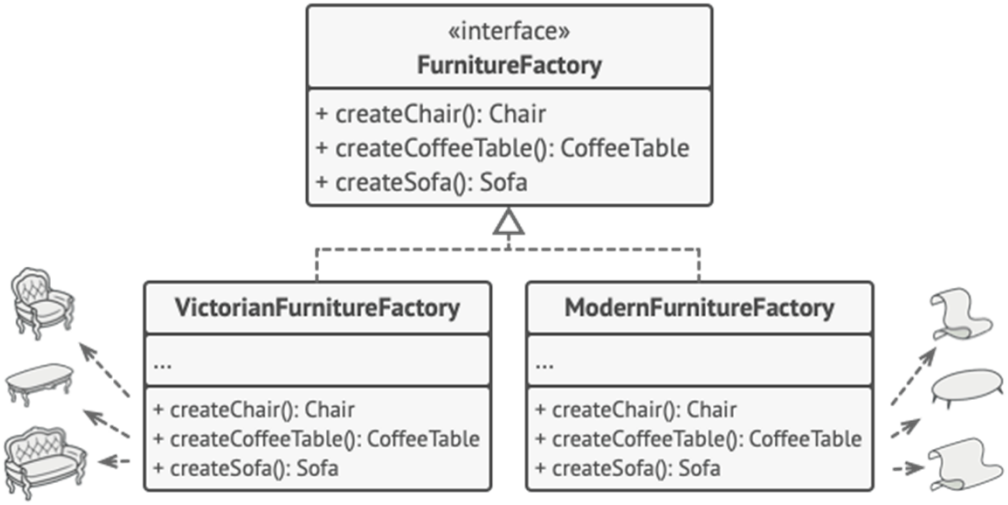
Abstract Factory – Case Study (1)

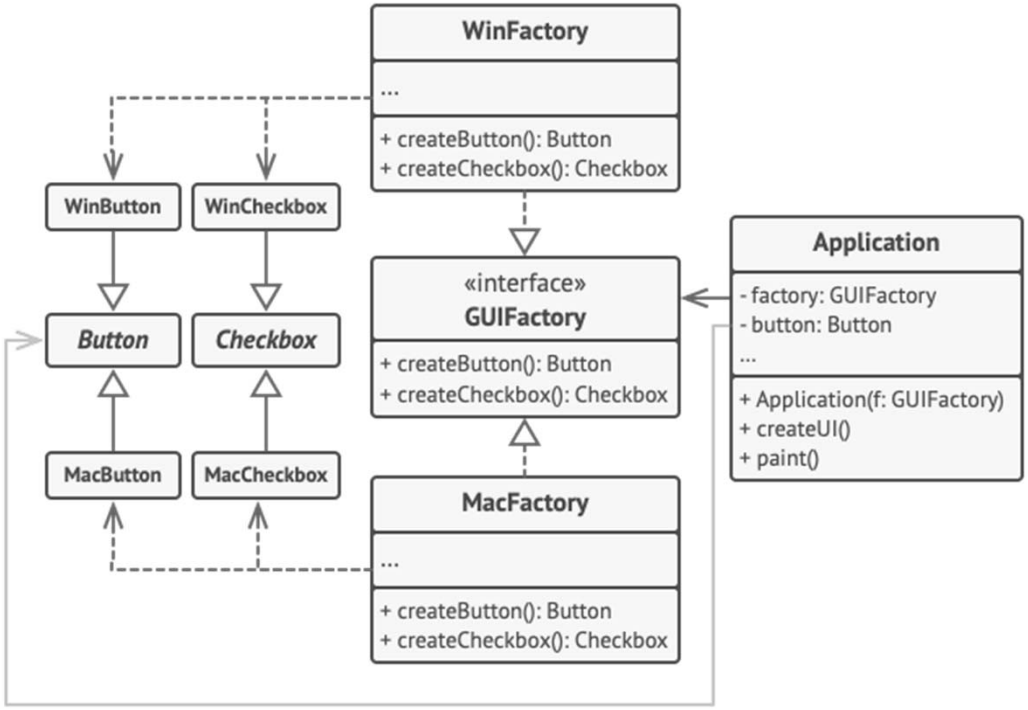
Supporting Multiple Look-and-Feel Standards

```
// define the product type at compile time or run-time (based on environment or user input)
// Creating a scrollbar…
ScrollBar* sb = guiFactory->CreateScrollBar();
```



Abstract Factory – Case Study (2)





Adapter

- Để sử dụng framework F, người phát triển ứng dụng phải cung cấp một lớp cài đặt giao diện IMath

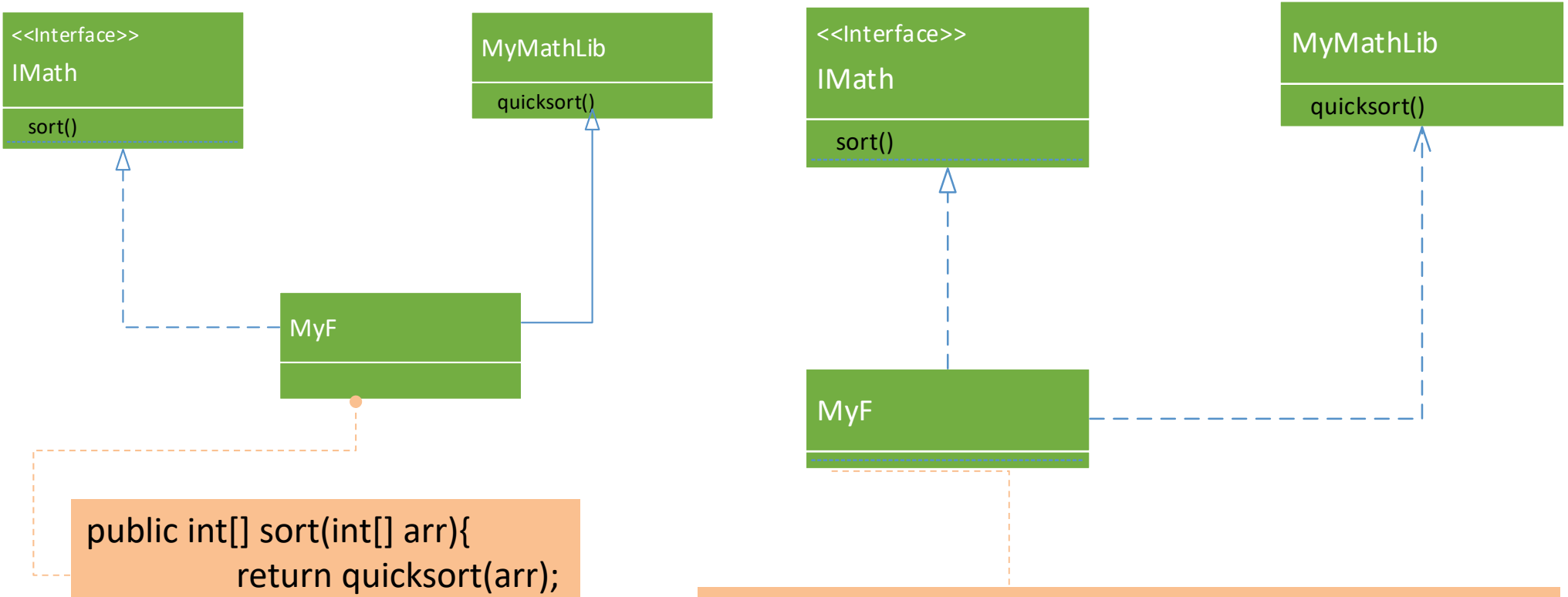
```
IMath{
        int[]  sort(int[] arr)
}
```

- Người phát triển ứng dụng đã download được một thư viện (.class, không chỉnh sửa được) có lớp MyMathLib với phương thức quicksort()

```
public class MyMathLib{
        int[]  quickSort(int[] arr){
        …
        }
}
```

Làm thể nào để sử dụng F với MyMathLib?

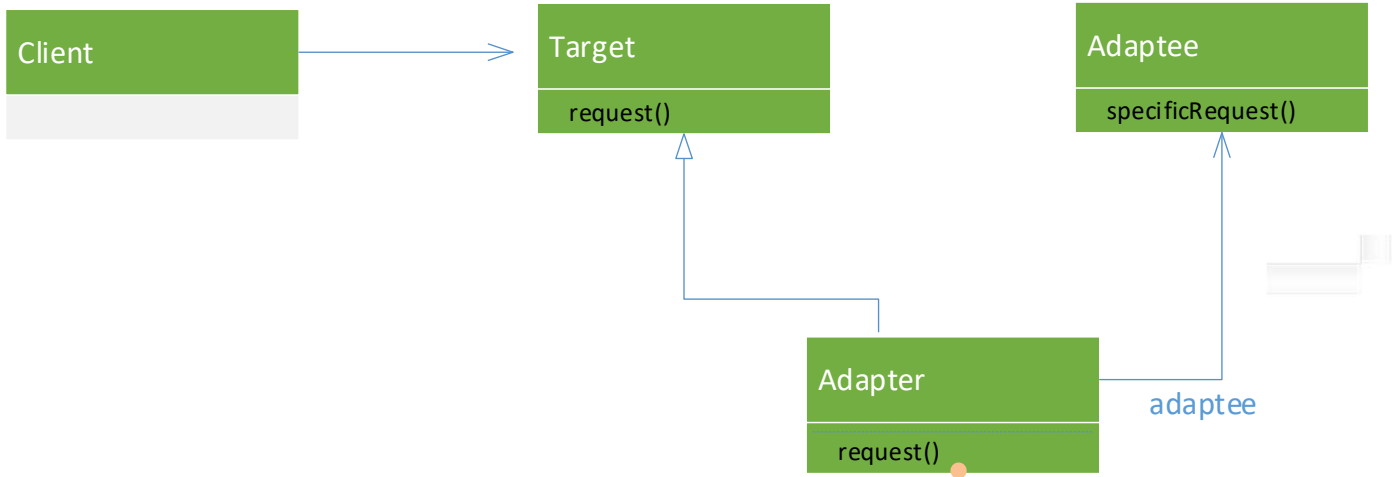
Adapter



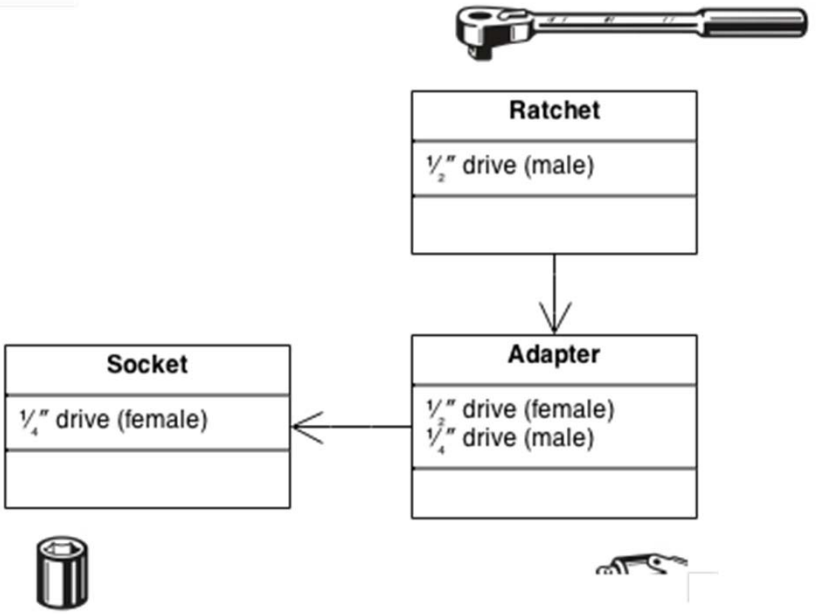

```
public int[] sort(int[] arr){
        return quicksort(arr);
}
```

```
public void sort(int[] arr){
      MyMathLib lib=new MyMathLib();
      return lib.quicksort(arr)
}
```

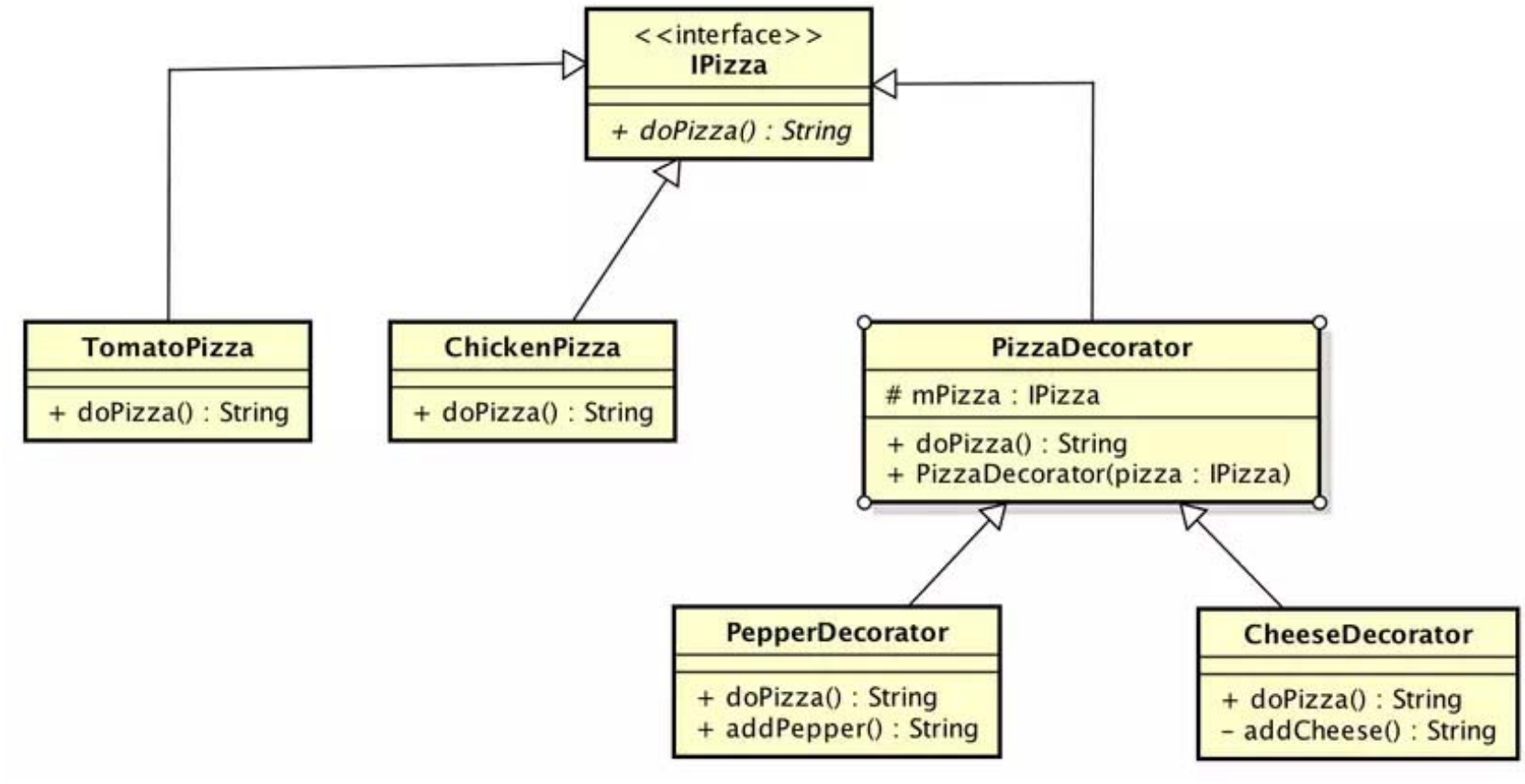
Adapter




```
public request(){
  adaptee.specificRequest();
}
```



Decorator – Pizza example



Source: https://viblo.asia/

Decorator – Pizza example

```
public interface IPizza {
        String doPizza();
}
public class TomatoPizza implements IPizza {
  public String doPizza() {
        return "I am a Tomato Pizza";
} }
public class ChickenPizza implements IPizza {
  public String doPizza() {
        return "I am a Chicken Pizza";
} }
public abstract class PizzaDecorator implements IPizza{
  protected IPizza mPizza;
  public PizzaDecorator(IPizza pizza) {
        mPizza = pizza;
}}
```

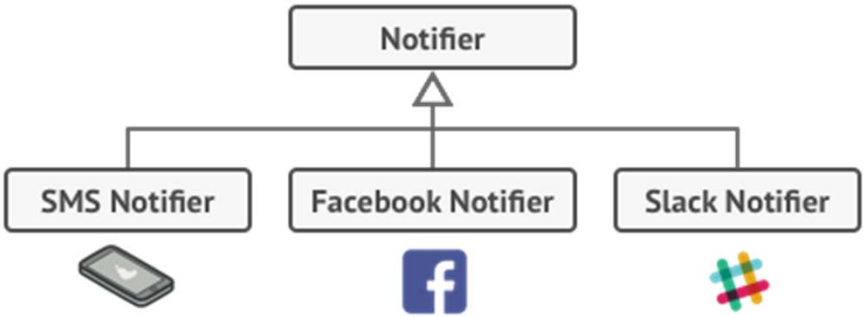
```
public class CheeseDecorator extends PizzaDecorator
{
  public CheeseDecorator(IPizza pizza) {
        super(pizza);
}
  public String doPizza() {
        String type = mPizza.doPizza();
        return type + " plus Cheese";
}}
public class PepperDecorator extends PizzaDecorator
{
  public PepperDecorator(IPizza pizza) {
        super(pizza);
}
  public String doPizza() {
        String type = mPizza.doPizza();
        return type + " plus Pepper";
}}
```

Decorator – Pizza example

```
public class PizzaShop {
  public static void main(String[] args) {
            IPizza tomato = new TomatoPizza();
            IPizza chicken = new ChickenPizza();
            // Add pepper to tomato-pizza
            PepperDecorator pepperDecorator = new PepperDecorator(tomato);
            System.out.println(pepperDecorator.doPizza());
            // Add cheese to chicken-pizza
            CheeseDecorator cheeseDecorator = new CheeseDecorator(chicken);
            System.out.println(cheeseDecorator.doPizza());
            // Add cheese and pepper to tomato-pizza
            CheeseDecorator cheese_pepperDec = new CheeseDecorator(pepperDecorator);
            System.out.println(cheese_pepperDec.doPizza());
} }
---------------------------------
I am a Tomato Pizza plus Pepper
I am a Tomato Pizza plus Cheese
I am a Tomato Pizza plus Pepper plus Cheese
```

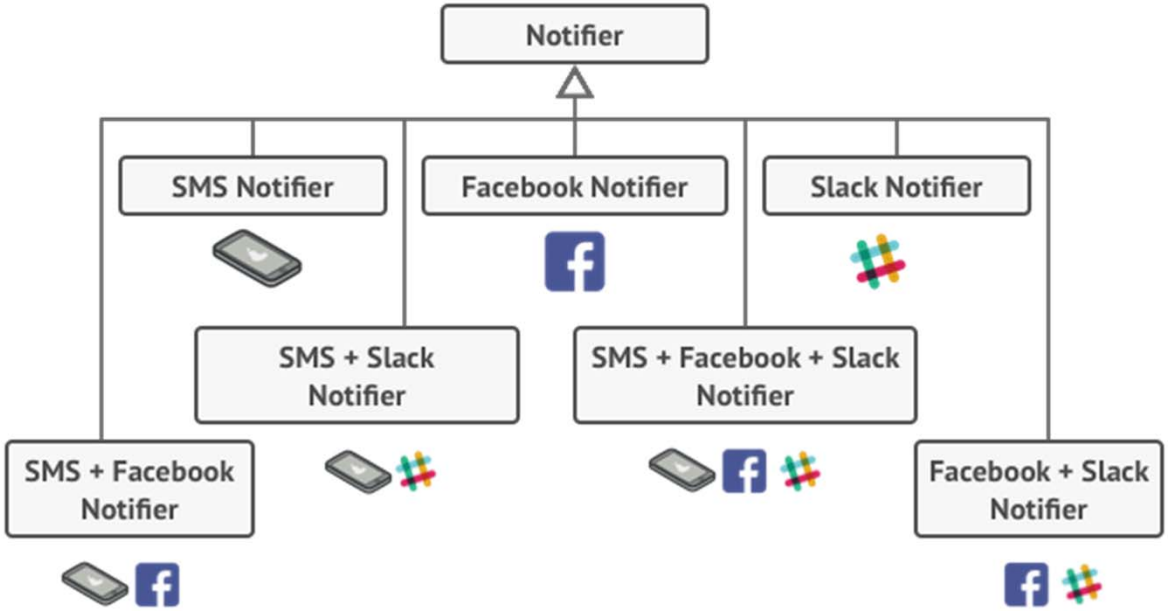
Decorator

- Sau một thời gian, người sử dụng nhu cầu gửi thông báo qua nhiều kênh khác nhau (ngoài qua email)
- Giải pháp (tạm thời): tạo các lớp mới kế thừa từ Notifier

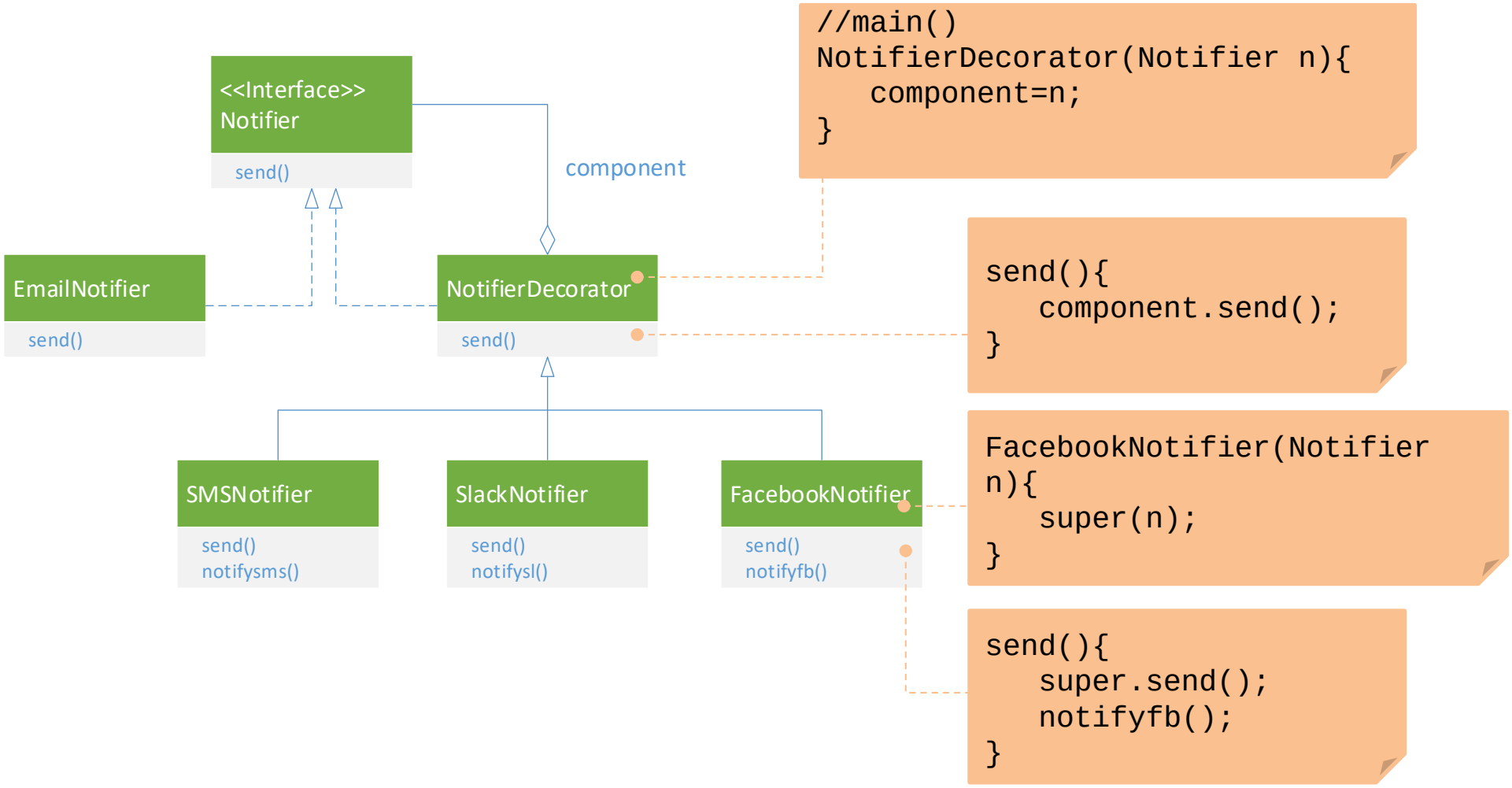


Decorator

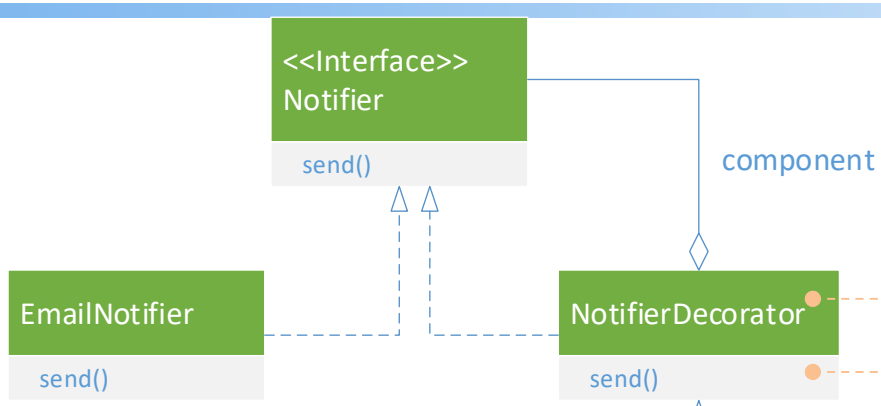
- Nhưng có trường hợp người dùng muốn gửi qua nhiều kênh cho cùng 1 thông điệp ➔ Số lớp tăng, không hợp lý



Decorator


<<Interface>>
Notifier
send()
component
EmailNotifier
send()
NotifierDecorator
send()
SMSNotifier
send()
notifysms()
SlackNotifier
send()
notifysl()
FacebookNotifier
send()
notifyfb()
//main()
NotifierDecorator(Notifier n){
component=n;
}
send(){
component.send();
}
FacebookNotifier(Notifier n){
super(n);
}
send(){
super.send();
notifyfb();
}

Decorator




```
//main()
NotifierDecorator(Notifier n){
    component=n;
}
```

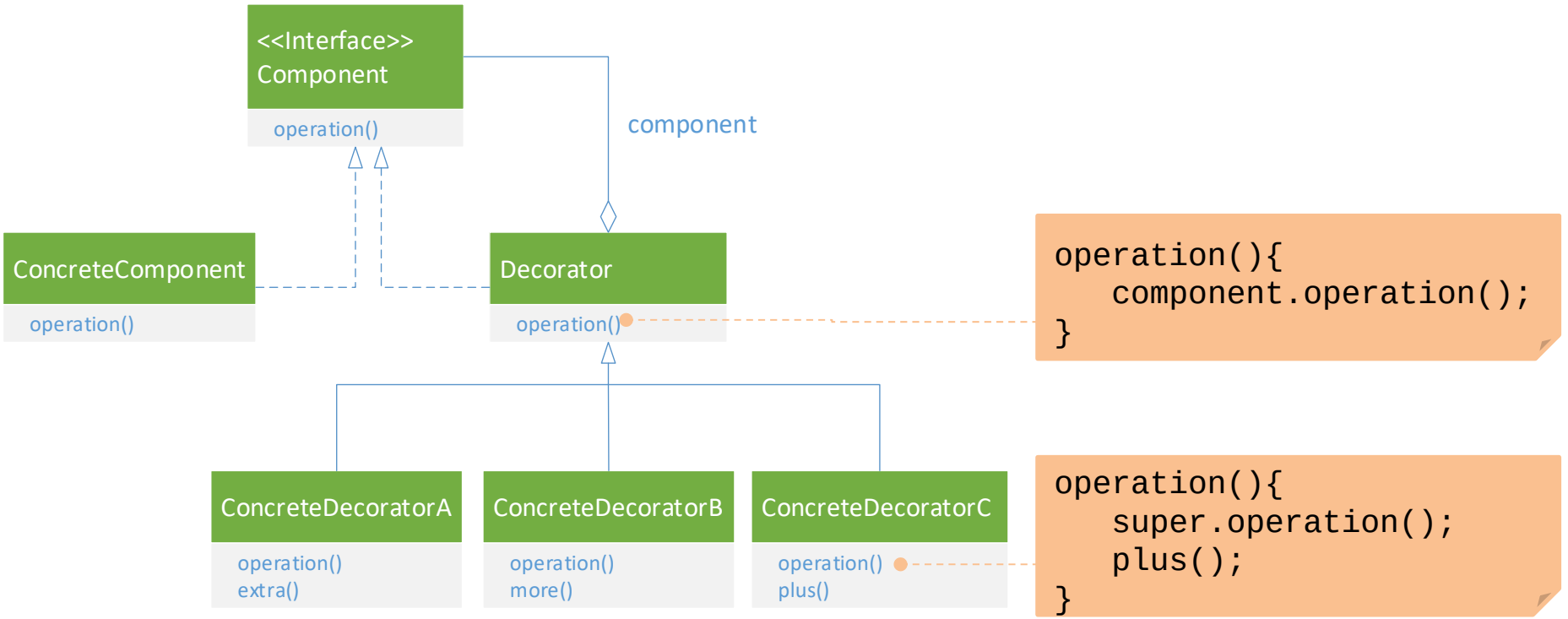
```
send(){
    component.send();
}
```

```
FacebookNotifier(Notifier
n){
    super(n);
}
```

```
send(){
    super.send();
    notifyfb();
}
```

```
//main()
Notifier n=new EmailNotifier();
         n=new FacebookNotifier(n);
         n=new SMSNotifier(n);
         n.send();
```

Decorator


<<Interface>>
Component
operation()
component
ConcreteComponent
operation()
Decorator
operation()
operation(){
component.operation();
}
ConcreteDecoratorA
operation()
extra()
ConcreteDecoratorB
operation()
more()
ConcreteDecoratorC
operation()
plus()
operation(){
super.operation();
plus();
}

Một số nguyên lý thiết kế

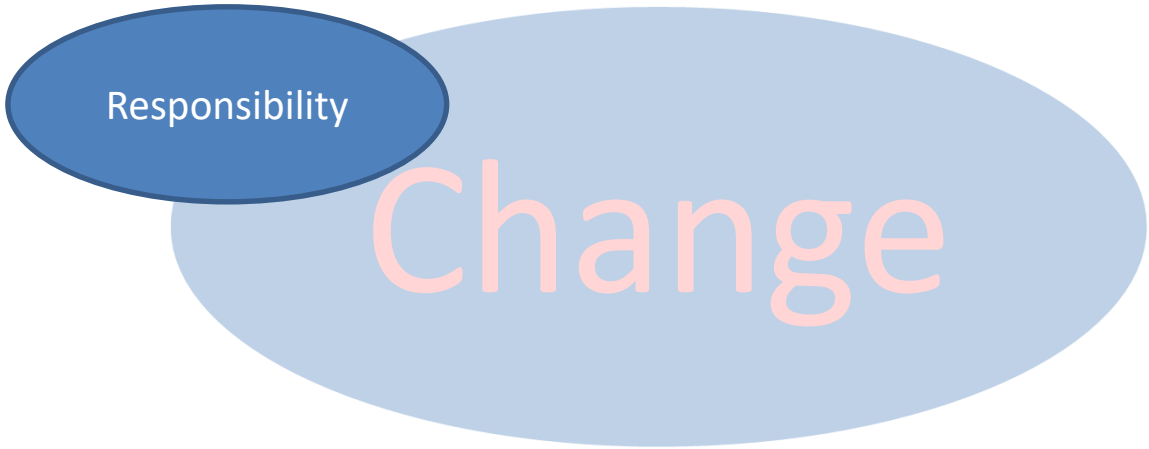
- OO với các nguyên lý đóng gói (encapsulation), trừu tượng hóa (abstraction), đa hình (polymorphism), inheritance (kế thừa) giúp cho lập trình viên viết các chương trình chất lượng cao
- Không phải chương trình nào viết bằng OO cũng có chất lượng cao
- Có một số các nguyên lý để có chương trình dễ bảo trì, tái sử dụng, và mở rộng

SOLID

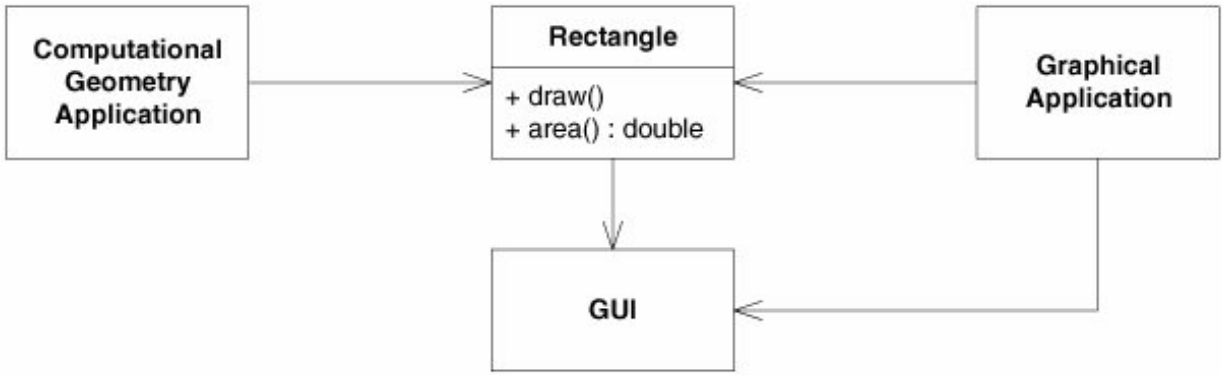
- SOLID gồm 5 nguyên lý thiết kế hướng đối tượng đã được vận dụng nhiều trong thực tế
  - **S**ingle Responsibility
  - **O**pen-close
  - **L**iskov Substitution
  - **I**nterface Segregation
  - **D**ependency Inversion

Single-Responsibility Principle (SRP)

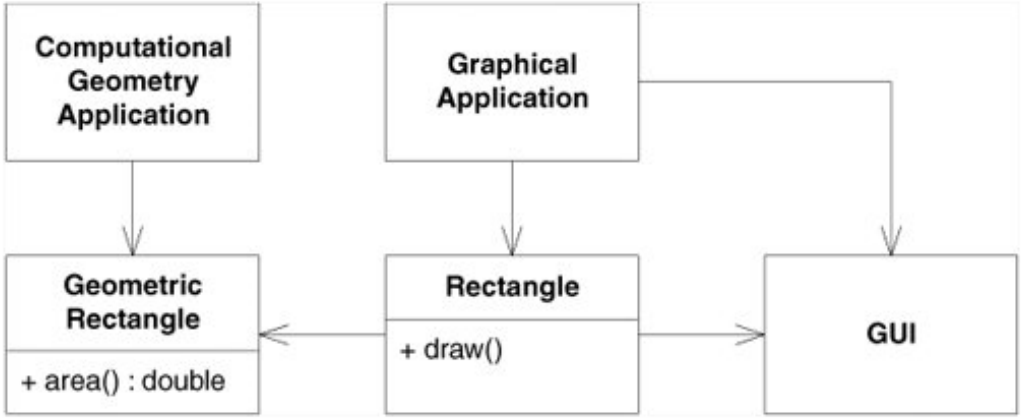
A class should have only one reason to change



Single-Responsibility Principle (SRP)



Single-Responsibility Principle (SRP)



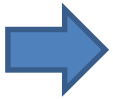
The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification

Software entities can be extended without modifying

The Open/Closed Principle (OCP)

```
class Rectangle{
…
}
class Circle{
…
}
class Diagram{
   LinkedList<Circle>
circles;
LinkedList<Rectangle>
rectangles;
}
```



```
class Rectangle extends
Shape{
…
}
class Circle extends Shape{
…
}
class Diagram{
   LinkedList<Shape> shapes;
}
```

The Open/Closed Principle (OCP)

```
class Rectangle extends Shape{
}
class Circle extends Shape{
}
class Diagram{
   LinkedList<Shape> shapes;
   public void DrawAllShapes(){
       for(Shape shape: shapes){
             if(shape.isInstanceOf(Rectangle))
        //draw rectangle
              else
        //draw circle
       }
   }
}
```

The Open/Closed Principle (OCP)


```
class Shape {
    public void Draw(){
    }
}
class Rectangle extends Shape{
    @Override
    public void Draw(){
    }
}
class Circle extends Shape{
    @Override
    public void Draw(){
    }
}
```

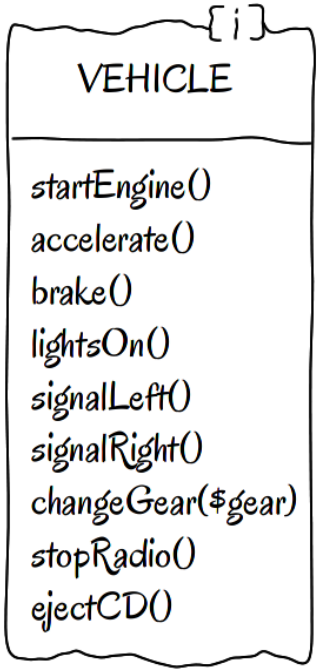
```
class Diagram{
   LinkedList<Shape> shapes;
   public void DrawAllShapes(){
       for(Shape shape: shapes){
           shape.Draw();
       }
   }
}
```

Liskov Substitution Principle (LSP)

Derived classes must be substitutable for their base classes

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use.



Dependency-Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend upon details. Details should depend upon abstractions.