Lập trình hướng đối tượng

Mẫu thiết kế (tt)

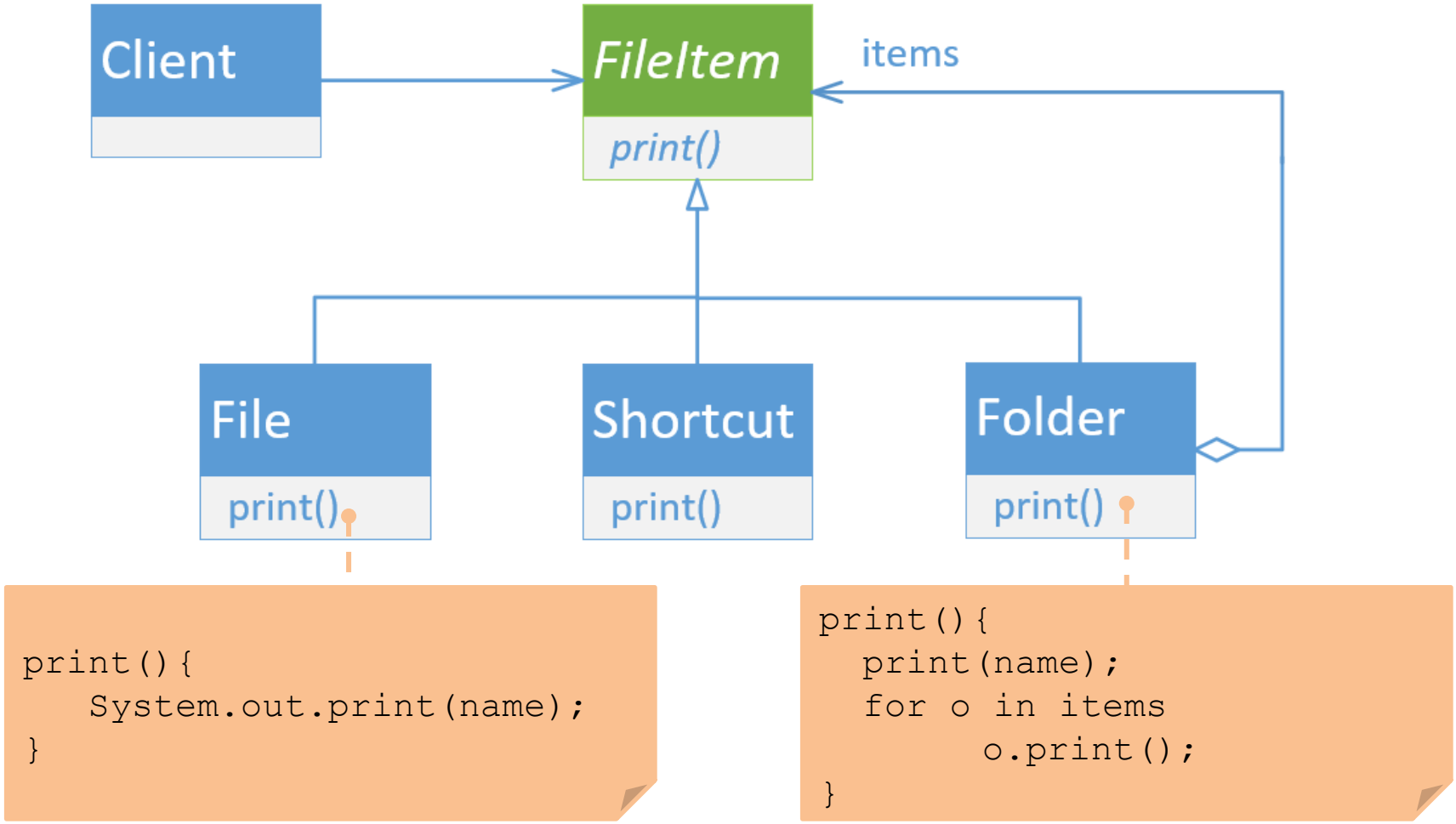
Nội dung

- Mẫu thiết kế Composite, Decorator
- Một số nguyên lý thiết kế

Composite

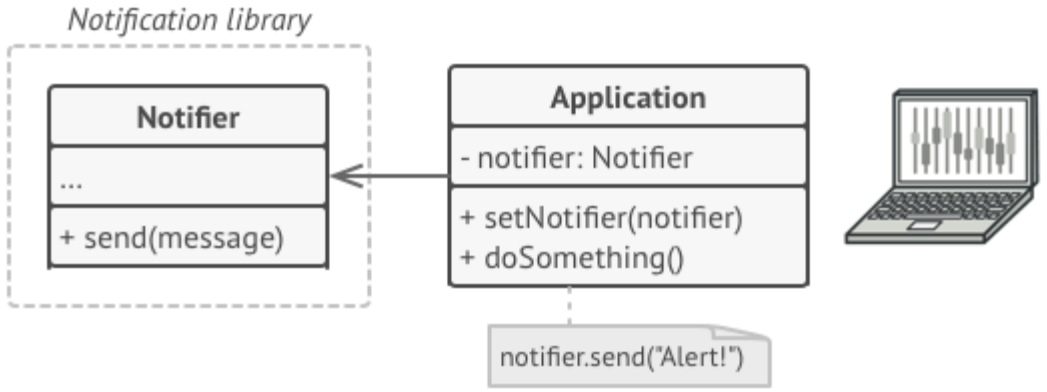
- Cần viết một công cụ quản lý hệ thống file. Các thành phần chính: file, shortcut, và folder. Folder có thể chứa folder, file, shortcut khác.
- Duyệt:
  - Duyệt file: in tên file, kích thước
  - Duyệt shortcut: in đường dẫn đến phần tử đích (phần tử mà shortcut làm đại diện)
  - Duyệt folder: in tên folder và duyệt tiếp nội dung bên trong folder

Composite


Client
FileItem
print()
items
File
print()
Shortcut
print()
Folder
print()
print(){
System.out.print(name);
}
print(){
print(name);
for o in items
o.print();
}

Decorator

- Ngữ cảnh: Xây dựng ứng dụng trong đó có chức năng gửi thông báo



- Phương thức send() sẽ gửi thông báo qua email

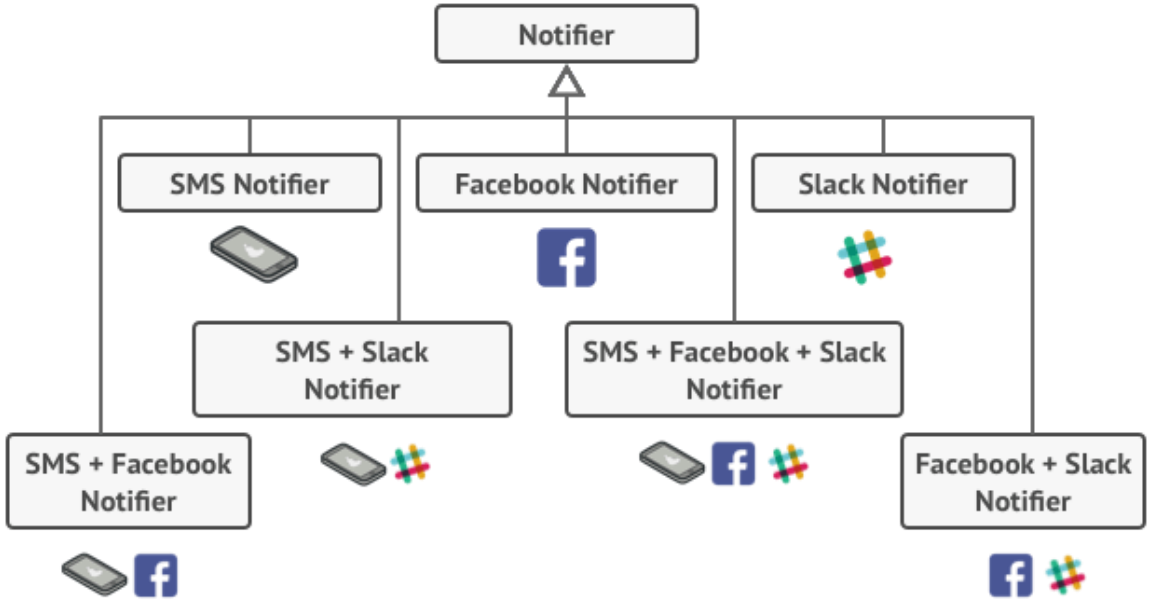
(Tham khảo tại https://refactoring.guru/design-patterns/decorator)

Decorator

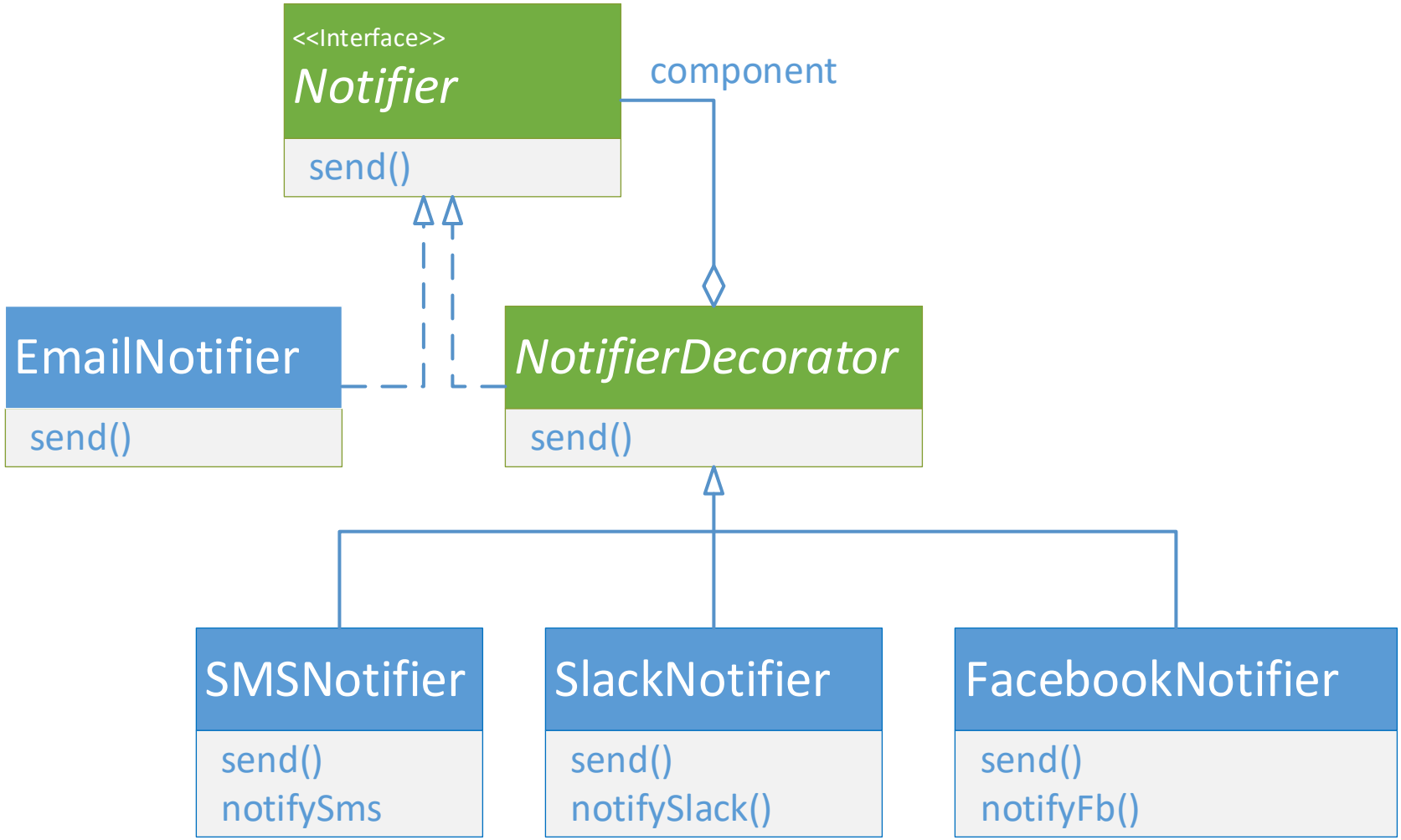
- Sau một thời gian, người sử dụng có nhu cầu gửi thông báo qua nhiều kênh khác nhau (ngoài qua email)
- Giải pháp (tạm thời): tạo các lớp mới kế thừa từ Notifier

Decorator

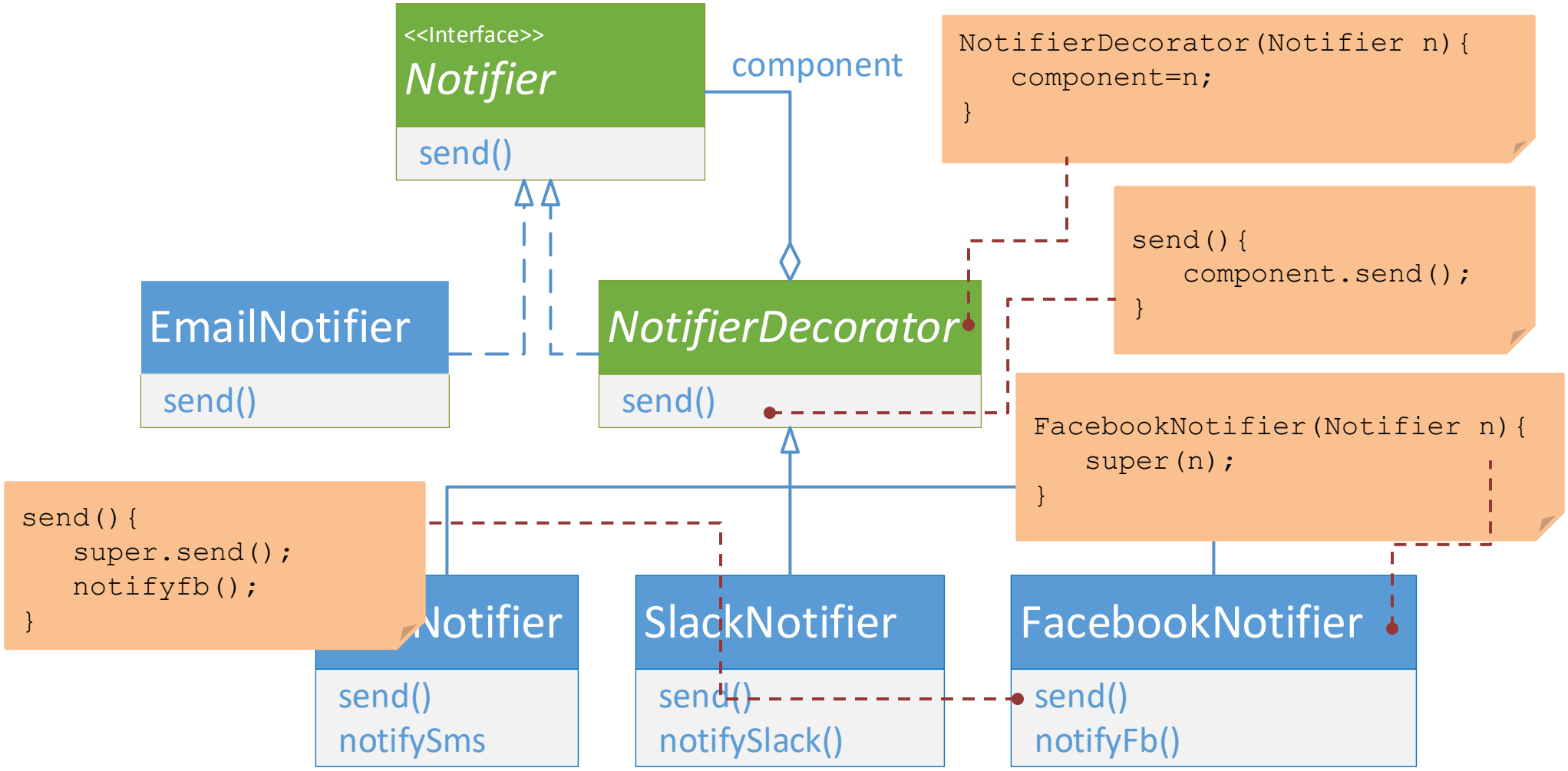
- Nhưng có trường hợp người dùng muốn gửi qua nhiều kênh cho cùng 1 thông điệp ➔ Số lớp tăng, không hợp lý



Decorator


<<Interface>>
Notifier
send()
component
EmailNotifier
send()
NotifierDecorator
send()
SMSNotifier
send()
notifySms
SlackNotifier
send()
notifySlack()
FacebookNotifier
send()
notifyFb()

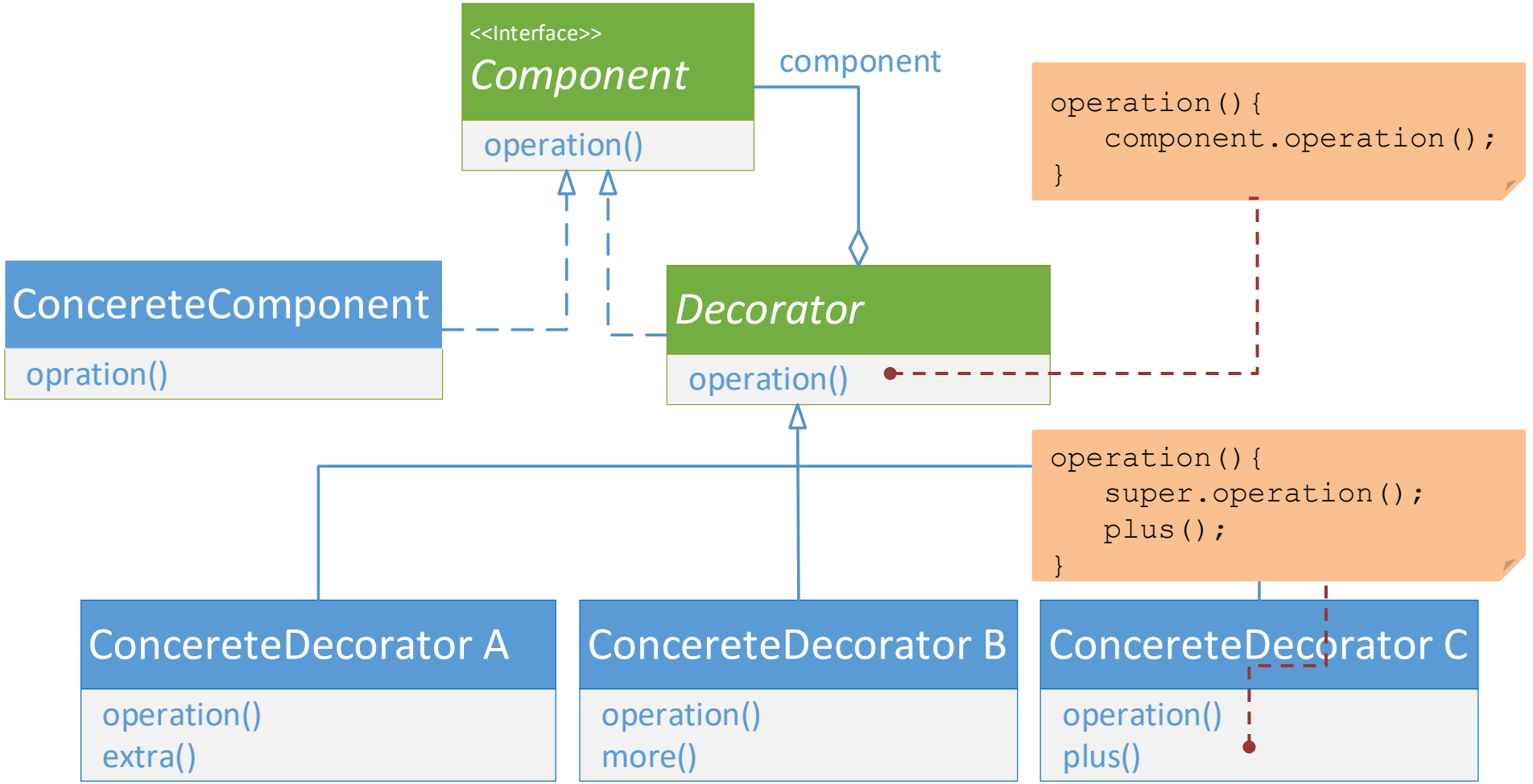
Decorator


<<Interface>>
Notifier
send()
component
NotifierDecorator(Notifier n){
component=n;
}
send(){
component.send();
}
EmailNotifier
send()
NotifierDecorator
send()
FacebookNotifier(Notifier n){
super(n);
}
send(){
super.send();
notifyfb();
}
Notifier
send()
notifySms
SlackNotifier
send()
notifySlack()
FacebookNotifier
send()
notifyFb()

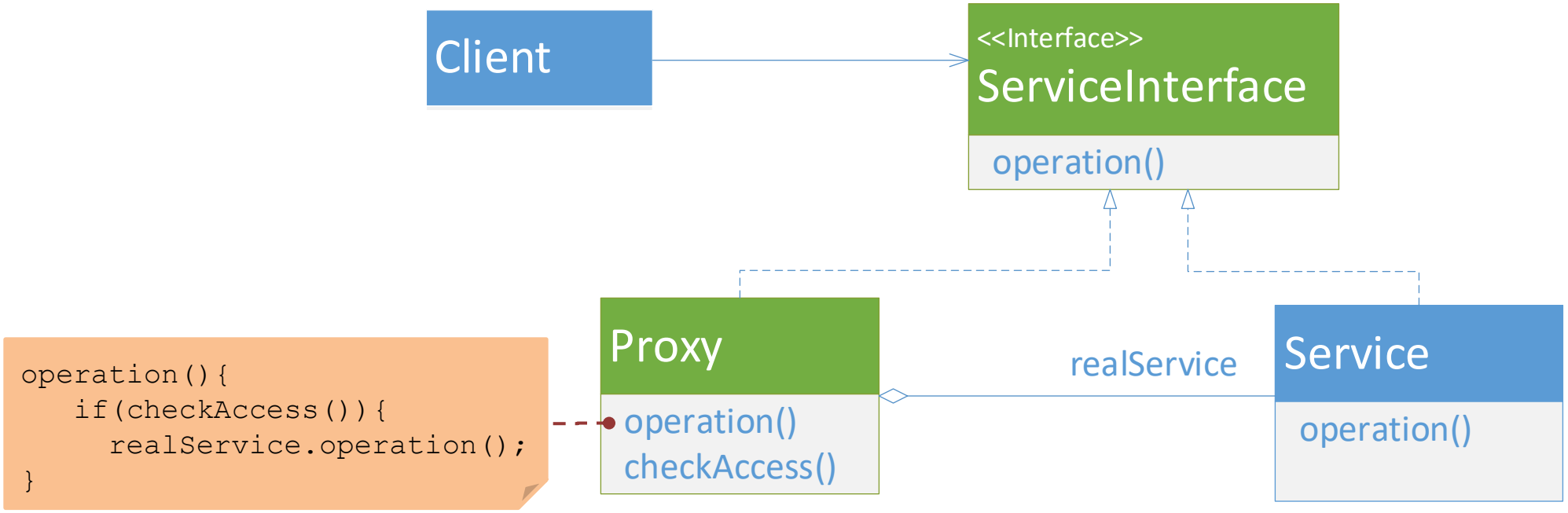
Decorator

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

Decorator


<<Interface>>
Component
operation()
component
operation(){
component.operation();
}
ConcereteComponent
opration()
Decorator
operation()
operation(){
super.operation();
plus();
}
ConcereteDecorator A
operation()
extra()
ConcereteDecorator B
operation()
more()
ConcereteDecorator C
operation()
plus()

Proxy


Client
<<Interface>>
ServiceInterface
operation()
Proxy
operation()
checkAccess()
realService
Service
operation()
operation(){
if(checkAccess()){
realService.operation();
}

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

DRY

Don’t Repeat Yourself

KISS

Keep It Simple, Silly

Keep It Simple, Stupid

Keep It Short and Simple

Keep It Simple and Straightforward

YAGNI

You Aren’t Gonna Need It

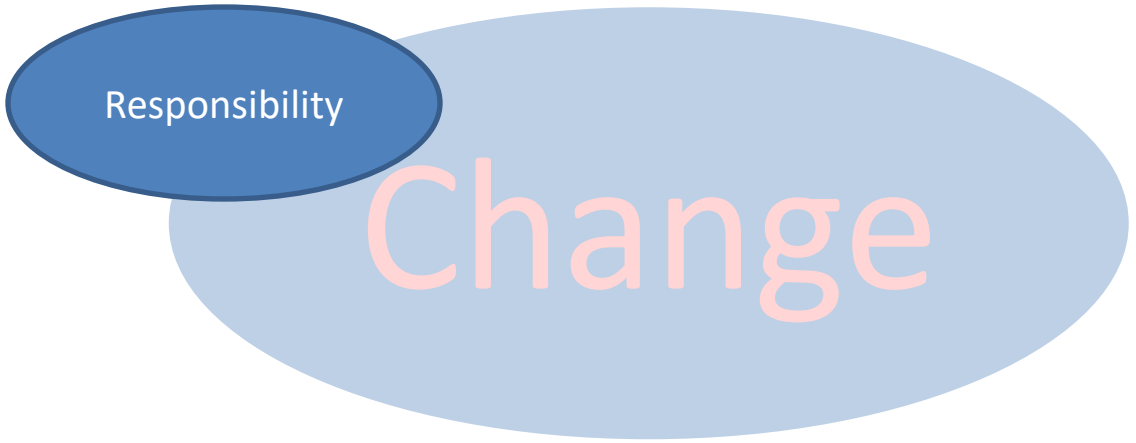
SOLID

SOLID gồm 5 nguyên lý thiết kế hướng đối tượng đã được vận dụng nhiều trong thực tế

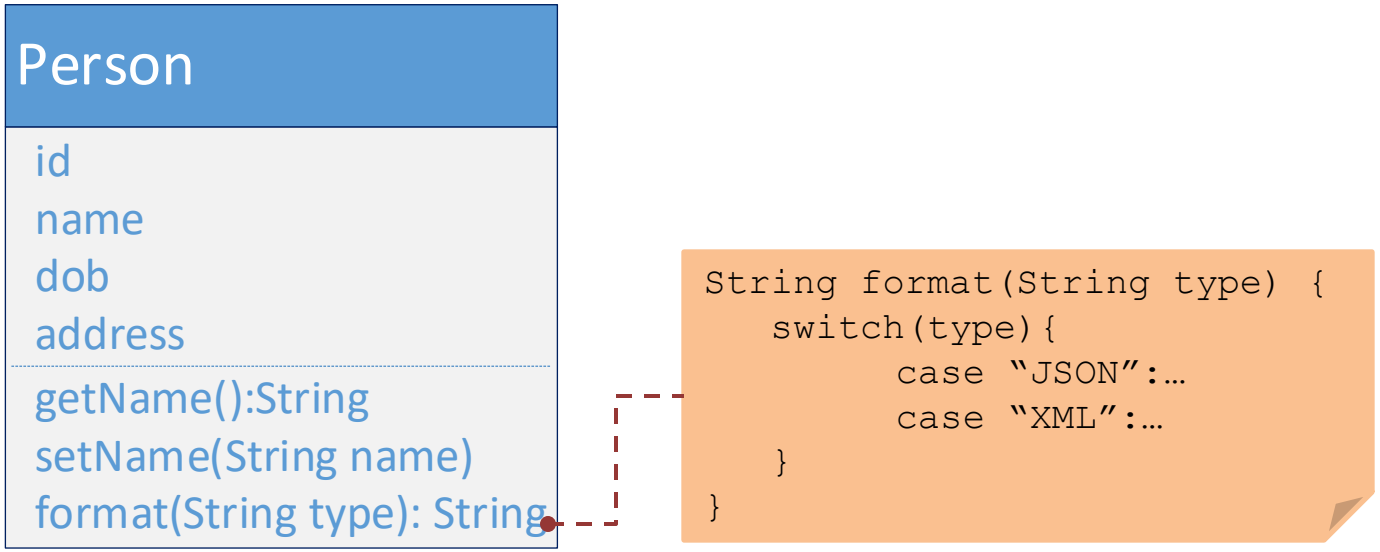
- **S**ingle Responsibility
- **O**pen-close
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Single-Responsibility Principle (SRP)

A class should have only one reason to change



Single-Responsibility Principle (SRP)



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;
}
```

The Open/Closed Principle (OCP)

```
class Shape{}
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

```
class Connection{
        public void connect() throws IOException{…}
}
```

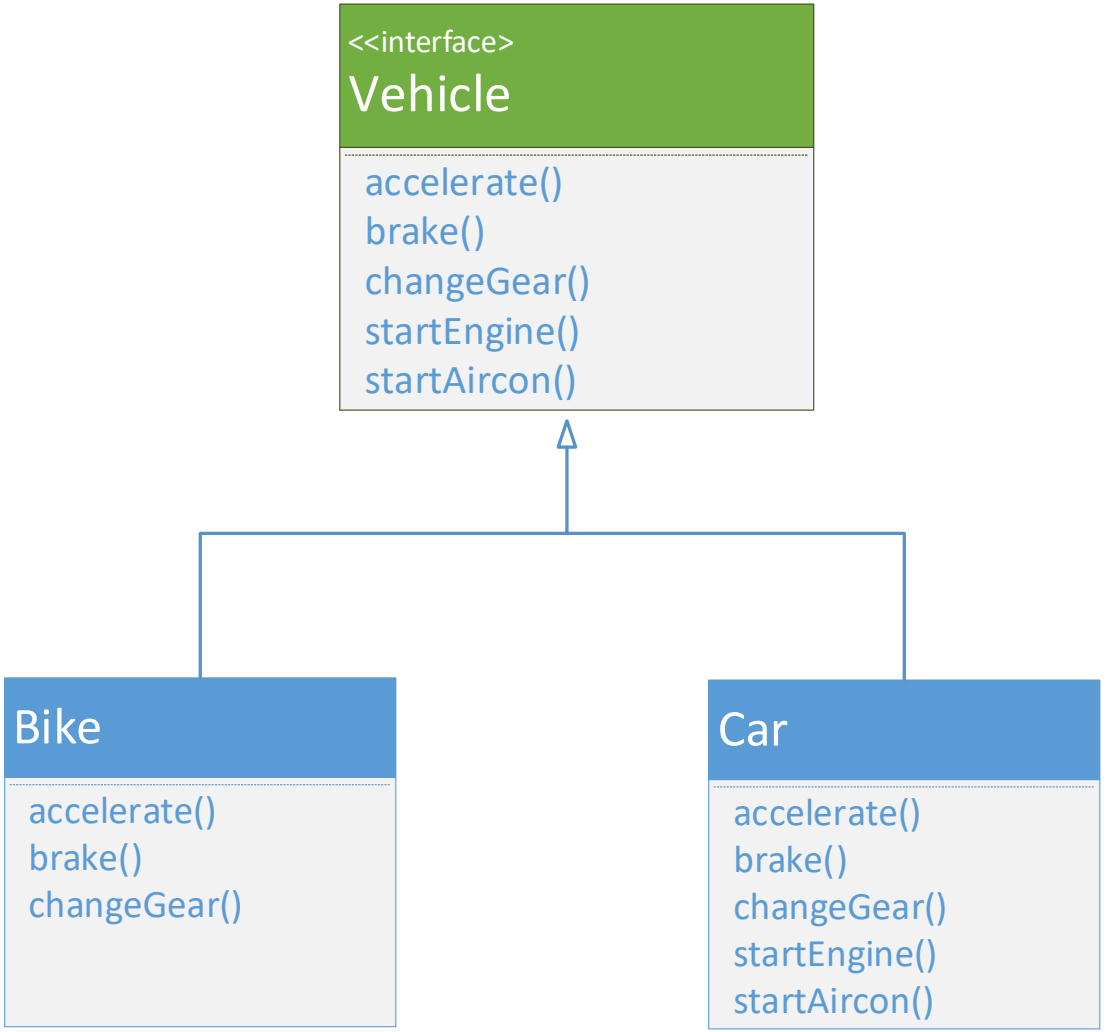
```
class DBConnection extends Connection{
        public void connect() throws Exception{…}
}
```

```
class Client{
        public void run(Connection con){…}
}
```

Interface Segregation Principle (ISP)

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

Liskov Substitution Principle (LSP)

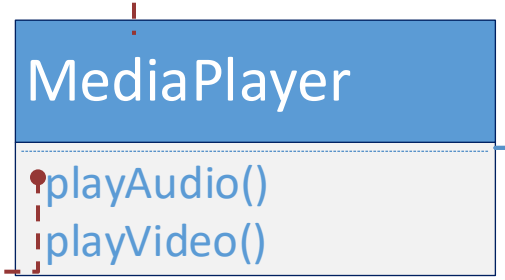


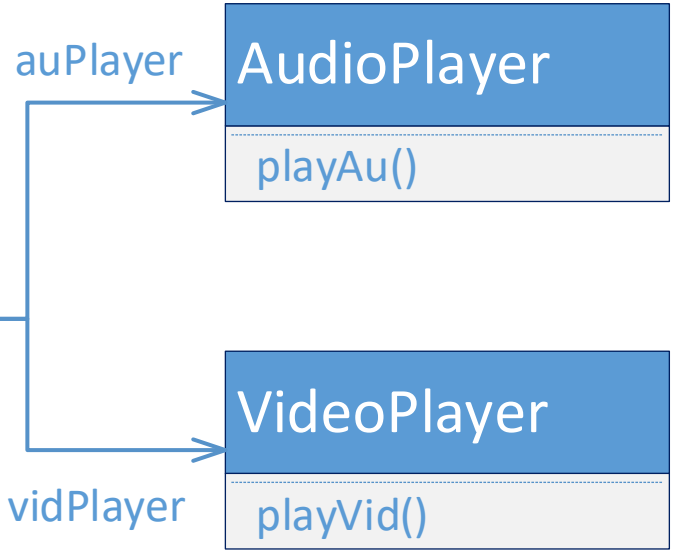
Dependency-Inversion Principle (DIP)

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.

Dependency-Inversion Principle (DIP)

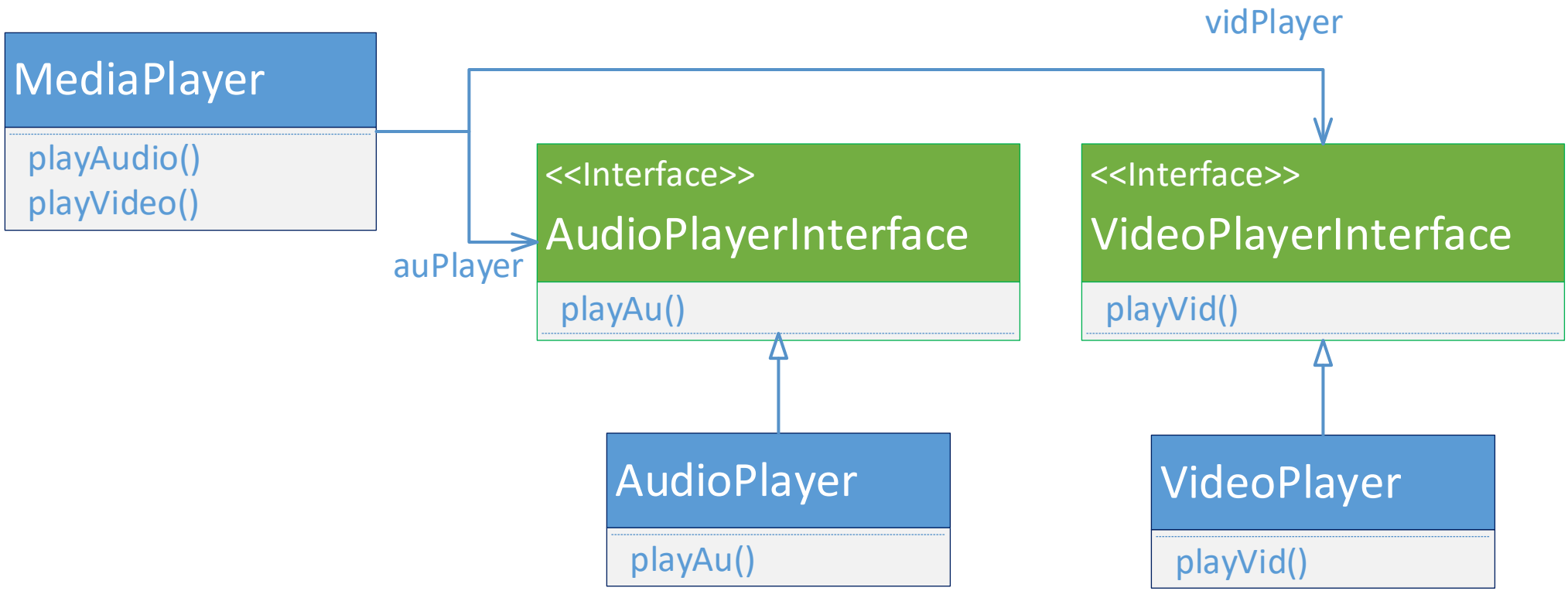
```
public MediaPlayer(){
        auPlayer=new AudioPlayer();
        vidPlayer=new VideoPlayer()
}
```






```
public void playAudio(String input){
        auplayer.playAu(input)
}
```

Dependency-Inversion Principle (DIP)



Tổng kết

- Mẫu thiết kế Composite và Decorator
- Một số nguyên lý thiết kế