



T3H

IT-Institute

Lập trình Java – Design Pattern

Ths. Vũ Duy Khương

1 Giới thiệu Design Pattern

2 Phân loại

3 Decorator

4 Proxy

5 Observer

6 Singleton

7 Factory

☐ Ngữ cảnh

- Kiến trúc sư Christopher Alexander (“Timeless way of building”, 1979) đã phát triển ý tưởng để xác định các đối tượng khi đưa ra: Một giải pháp cho một vấn đề trong một ngữ cảnh – “A solution to a problem in a context”



- Để xác định một mẫu (pattern), cần đưa ra **lý do** và **cách thức** để xây dựng từng giải pháp

Giới thiệu

- ❑ Design Pattern lần đầu được giới thiệu bởi Gang of Four (GoF) : Gamma Erich (PhD thesis), Richard Helm, Ralph Johnson và John Vlissides (1995).
- ❑ Đây là một nét “văn hóa mới” trong cộng đồng lập trình
- ❑ Pattern là thiết bị cho phép các chương trình chia sẻ kiến thức về thiết kế. Khi xây dựng chương trình, có nhiều vấn đề gặp phải và nó xuất hiện lại thường xuyên.
- ❑ Thư viện các pattern là nơi để chúng ta có thể tìm thấy “Các giải pháp lập trình tốt nhất”.

□ Khái niệm

- Mẫu thiết kế (Design Pattern) là vấn đề thông dụng cần giải quyết và là cách giải quyết vấn đề đó trong một ngữ cảnh cụ thể
- Mẫu thiết kế không đơn thuần là một bước nào đó trong các giai đoạn phát triển phần mềm mà nó đóng vai trò là sáng kiến để giải quyết một vấn đề thông dụng nào đó.
- Mẫu thiết kế sẽ giúp cho việc giải quyết vấn đề nhanh, gọn và hợp lý hơn.
- Mẫu thiết kế còn được sử dụng nhằm cô lập các thay đổi trong mã nguồn, từ đó làm cho hệ thống có khả năng tái sử dụng cao.

Giới thiệu Design Pattern

❑ Lý do nên dùng Design Pattern

- Tái sử dụng: Việc thiết kế một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại là rất khó; cần phải xác định được có những lớp đối tượng nào, quan hệ giữa chúng ra sao, có kế thừa hay không,... Thiết kế phải đảm bảo không chỉ giải quyết được các vấn đề hiện tại, mà còn có thể tiến hành mở rộng trong tương lai. Vì vậy, nếu phần mềm không có một thiết kế tốt, việc sau này khi mở rộng phần mềm lại phải thiết kế lại từ đầu rất có thể xảy ra.

Giới thiệu Design Pattern

❑ Lý do nên dùng Design Pattern

- Kinh nghiệm quý báu: Design Pattern là những kinh nghiệm đã được đúc kết từ những người đi trước, việc sử dụng Design Pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các giải pháp để giải quyết những vấn đề đã có lời giải.

Phân loại Design Pattern

- ❑ Hệ thống các mẫu design pattern được chia thành 3 nhóm dựa theo vai trò

Creational	Structural	Behavioral
Abstract Factory, Factory, Builder, Prototype, Singleton	Adapter, Bridge, Composite, Decorator, Facade, Proxy, Flyweight	Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor.

Phân loại Design Pattern

❑ Trong đó:

▪ Nhóm Creational (nhóm kiến tạo)

- Hỗ trợ cho việc khởi tạo đối tượng trong hệ thống: khởi tạo một đối tượng cụ thể từ một định nghĩa trừu tượng (abstract, class, interface).
- Giúp khắc phục những vấn đề khởi tạo đối tượng, hạn chế sự phụ thuộc vào platform

Phân loại Design Pattern

❑ Trong đó:

▪ Nhóm Structural (nhóm cấu trúc)

- Các lớp đối tượng kết hợp với nhau tạo thành cấu trúc lớn hơn
- Cung cấp cơ chế xử lý những lớp không thể thay đổi, ràng buộc muộn (late binding) và giảm kết nối (lower coupling) giữa các thành phần và cung cấp các cơ chế khác để kế thừa.
- Diễn tả một cách có hiệu quả cả việc phân chia hoặc kết hợp các phần tử trong một ứng dụng

Phân loại Design Pattern

❑ Trong đó:

▪ Nhóm Behavioral (nhóm hành vi)

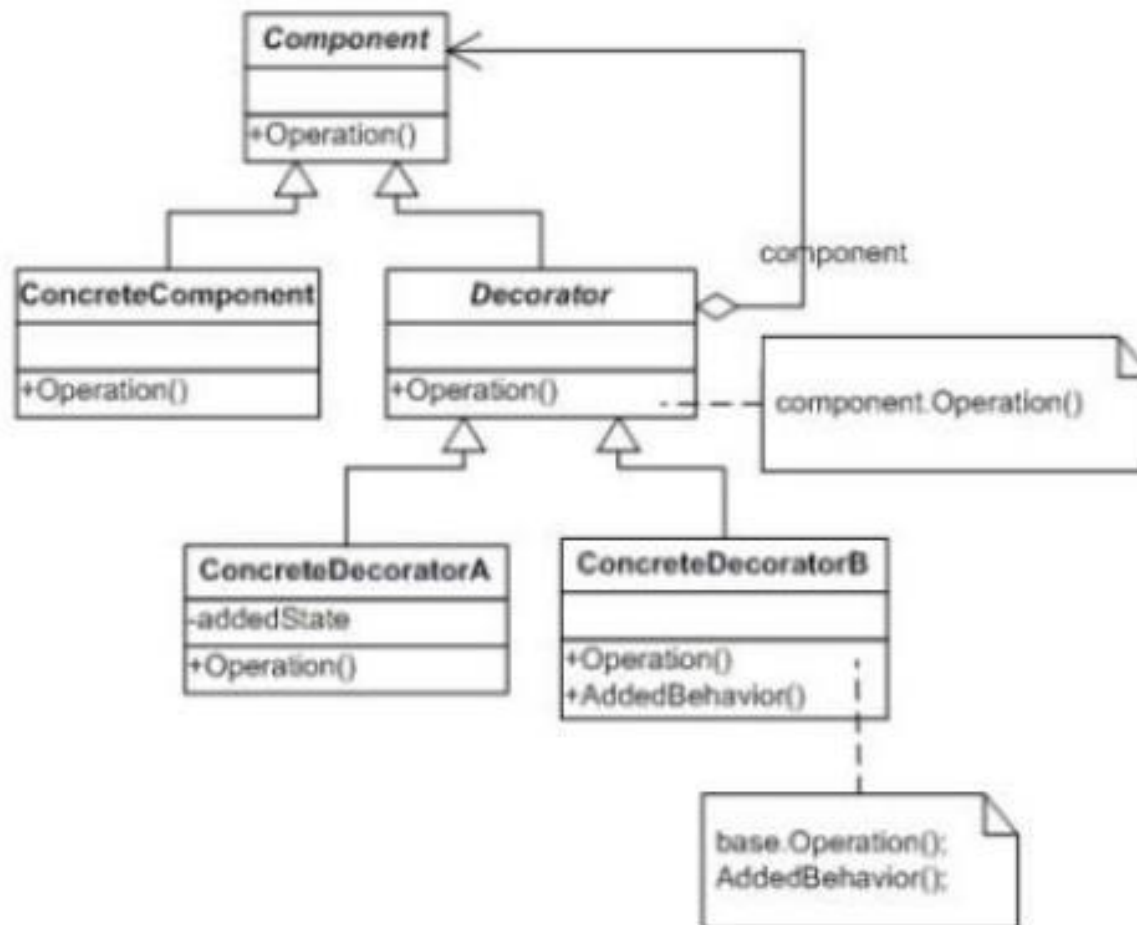
- Cách thức để các lớp và đối tượng có thể giao tiếp với nhau
- Che giấu hiện thực của đối tượng, che giấu giải thuật, hỗ trợ việc thay đổi cấu hình đối tượng một cách linh động.
- Có liên quan đến luồng điều khiển của hệ thống. Một vài cách của tổ chức điều khiển bên trong một hệ thống có thể mang lại các lợi ích cả về hiệu suất lẫn khả năng bảo trì hệ thống đó

□ Giới thiệu

- Là mẫu dung để bổ sung trách nhiệm cho đối tượng tại thời điểm thực thi.
- Được xem là sự thay thế hiệu quả cho phương pháp kế thừa trong việc bổ sung trách nhiệm cho đối tượng và mức tác động là ở mức đối tượng thay vì ở mức lớp như phương pháp kế thừa.

Decorator

□ Cấu trúc mẫu



Decorator

```
public interface Component {  
    void Operation();  
}
```

```
public class Concrete  
implements Component {  
    @Override  
    public void Operation() {  
        System.out.println("This  
is Concrete Component");  
    }  
}
```

```
public abstract class Decorator  
implements Component {  
    protected Component  
decoratedComponent;  
    public Decorator(Component  
decoratedComponent) {  
        this.decoratedComponent =  
decoratedComponent;  
    }  
    public void Operation() {  
decoratedComponent.Operation();  
    }
```

Decorator

```
public class ConcreteDecoratorA extends Decorator {  
    public ConcreteDecoratorA(Component decoratedComponent) {  
        super(decoratedComponent);  
    }  
    @Override  
    public void Operation() {  
        decoratedComponent.Operation();  
        addedState(decoratedComponent);  
    }  
    private void addedState(Component decoratedComponent) {  
        System.out.println("This is Concrete Decorator A");  
    }  
}  
// làm tương tự cho ConcreteDecoratorB
```

□ Trong đó:

- Component: là một interface chứa các phương thức ảo (ở đây là defaultMethod)
- ConcreteComponent: là một lớp kế thừa từ Component, cài đặt các phương thức cụ thể (defaultMethod được cài đặt tường minh)
- Decorator: là một lớp ảo kế thừa từ Component đồng thời cũng chứa 1 thể hiện của Component, phương thức defaultMethod trong Decorator sẽ được thực hiện thông qua thể hiện này.
- ConcreteDecoratorX: là các lớp kế thừa từ Decorator, khai báo tường minh các phương thức, đặc biệt trong các lớp này khai báo tường minh các “trách nhiệm” cần thêm vào khi run-time

□ Áp dụng

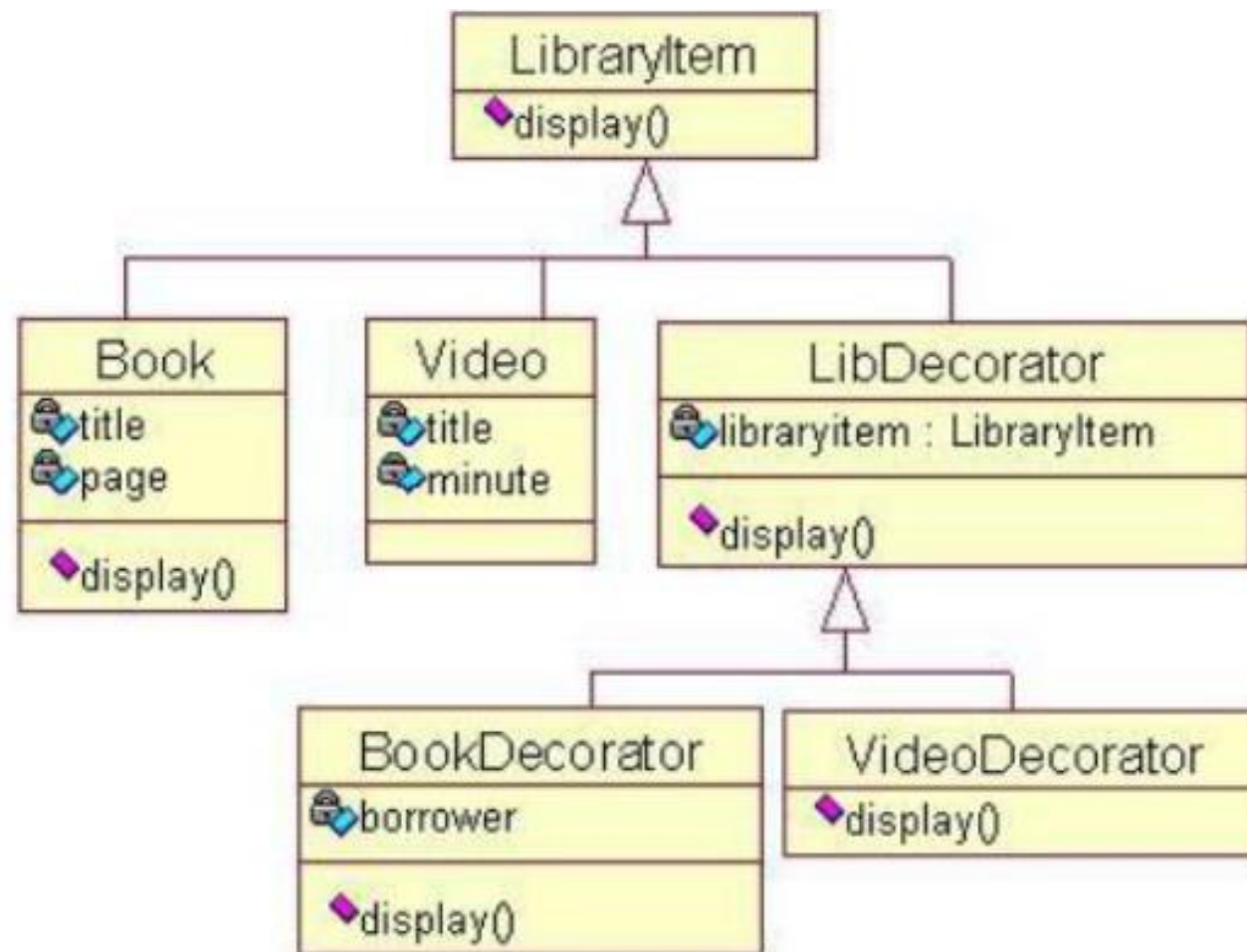
- Khi muốn thay đổi hành động mà không ảnh hưởng đến người dung, không phụ thuộc vào giới hạn các lớp con
- Khi muốn thành phần có thể thêm vào hoặc rút bỏ đi khi hệ thống đang chạy
- Có một số đặc tính phụ thuộc mà muốn ứng dụng một cách động và muốn kết hợp chúng vào trong một thành phần

❑ Ví dụ 1:

- Trong thư viện có các tài liệu: sách, video... Các loại tài liệu này có các thuộc tính khác nhau, phương thức hiển thị của giao tiếp Library Item sẽ hiển thị các thông tin này. Giả sử, ngoài các thông tin thuộc tính trên, đôi khi ta muốn hiển thị độc giả mượn một cuốn sách (chức năng hiển thị độc giả này sẽ không phải lúc nào cũng muốn hiển thị), hoặc muốn xem đoạn video (không thường xuyên).

Decorator

❑ Ví dụ 1:



❑ Ví dụ 2:

- Một hệ thống quản lý dự án, nơi nhân viên đang làm việc với các vai trò khác nhau, chẳng hạn như thành viên nhóm (team member), trưởng nhóm (team lead) và người quản lý (manager). Một thành viên trong nhóm chịu trách nhiệm thực hiện các nhiệm vụ được giao và phối hợp với các thành viên khác để hoàn thành nhiệm vụ nhóm. Mặt khác, một trưởng nhóm phải quản lý và cộng tác với các thành viên trong nhóm của mình và lập kế hoạch nhiệm vụ của họ. Tương tự như vậy, một người quản lý có thêm một số trách nhiệm đối với một trưởng nhóm như quản lý yêu cầu dự án, tiến độ, phân công công việc.

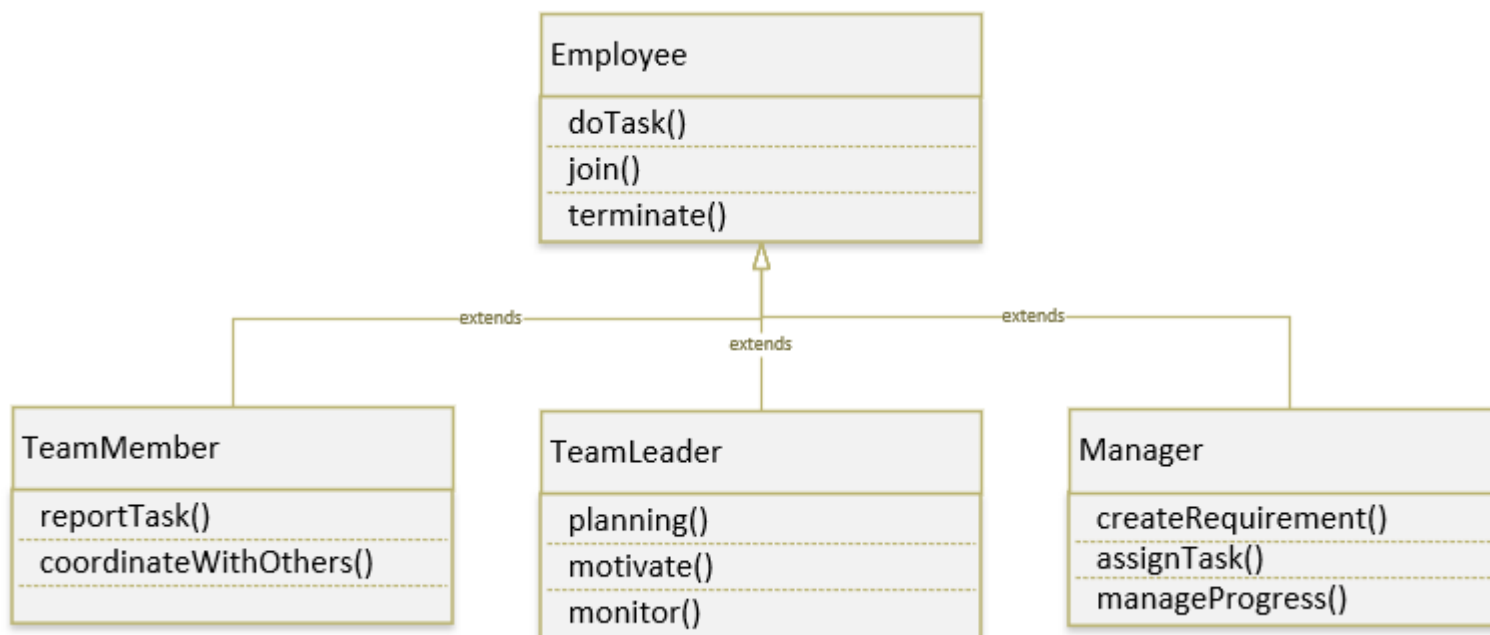
❑ Ví dụ 2:

Sau đây là các thành phần tham gia vào hệ thống và hành vi của chúng:

- **Employee:** thực hiện công việc (doTask), tham gia vào dự án (join), rời khỏi dự án (terminate).
- **Team member:** báo cáo task được giao (report task), cộng tác với các thành viên khác (coordinate with others).
- **Team lead:** lên kế hoạch (planning), hỗ trợ các thành viên phát triển (motivate), theo dõi chất lượng công việc và thời gian (monitor).
- **Manager:** tạo các yêu cầu dự án (create requirement), giao nhiệm vụ cho thành viên (assign task), quản lý tiến độ dự án (progress management).

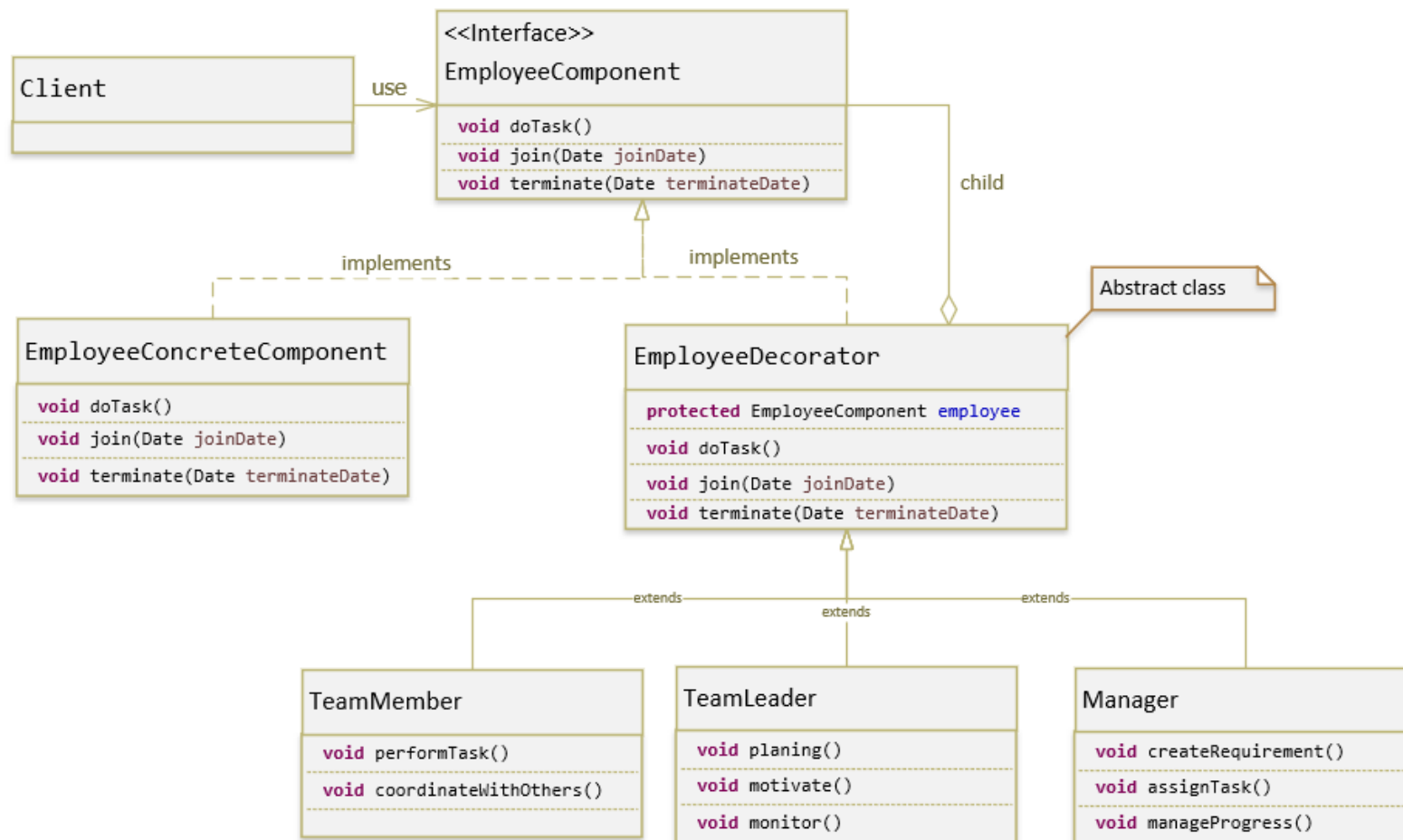
Decorator

❑ Ví dụ 2:

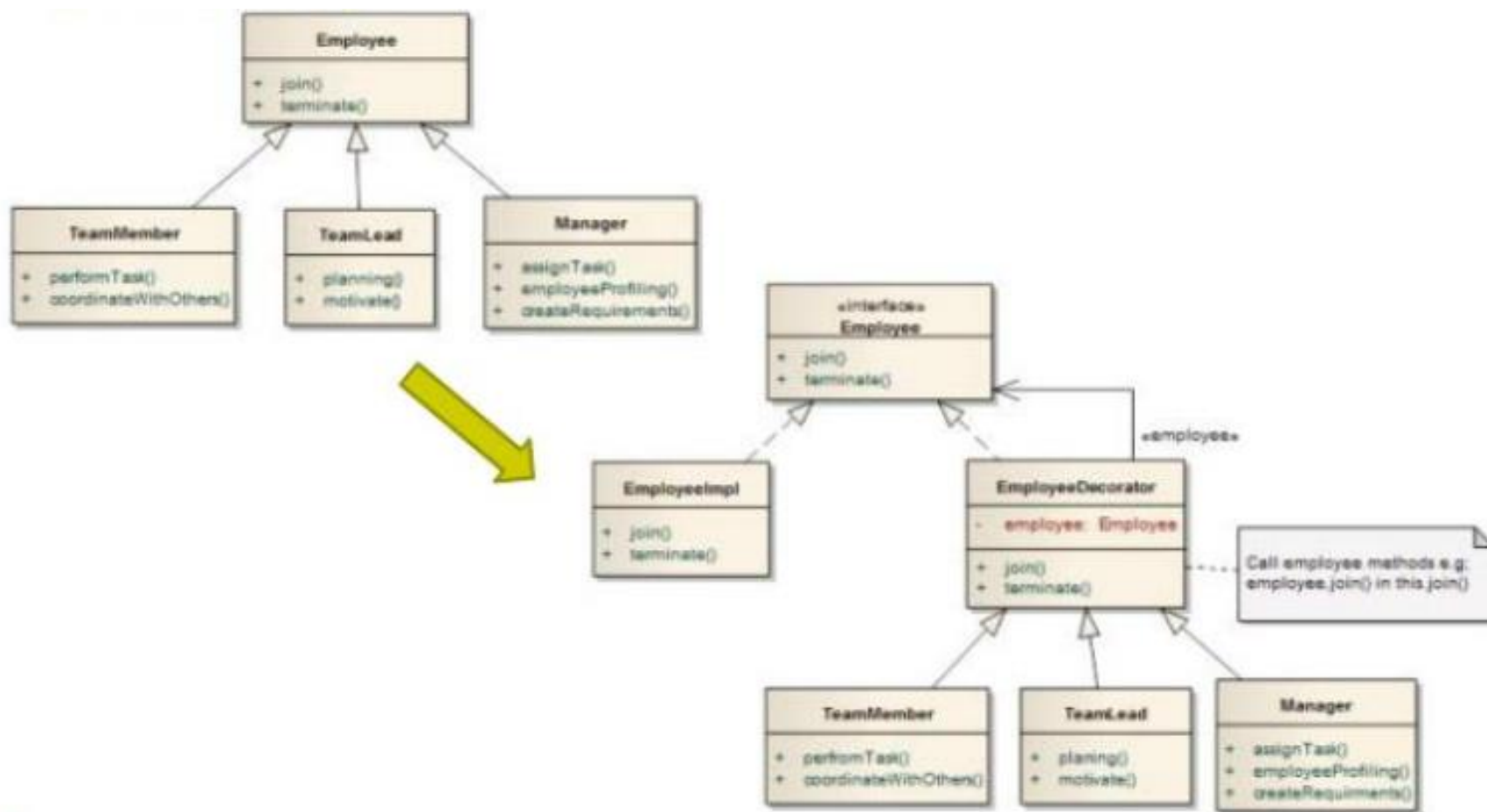


Decorator

❑ Ví dụ 2:



❑ Ví dụ 2:

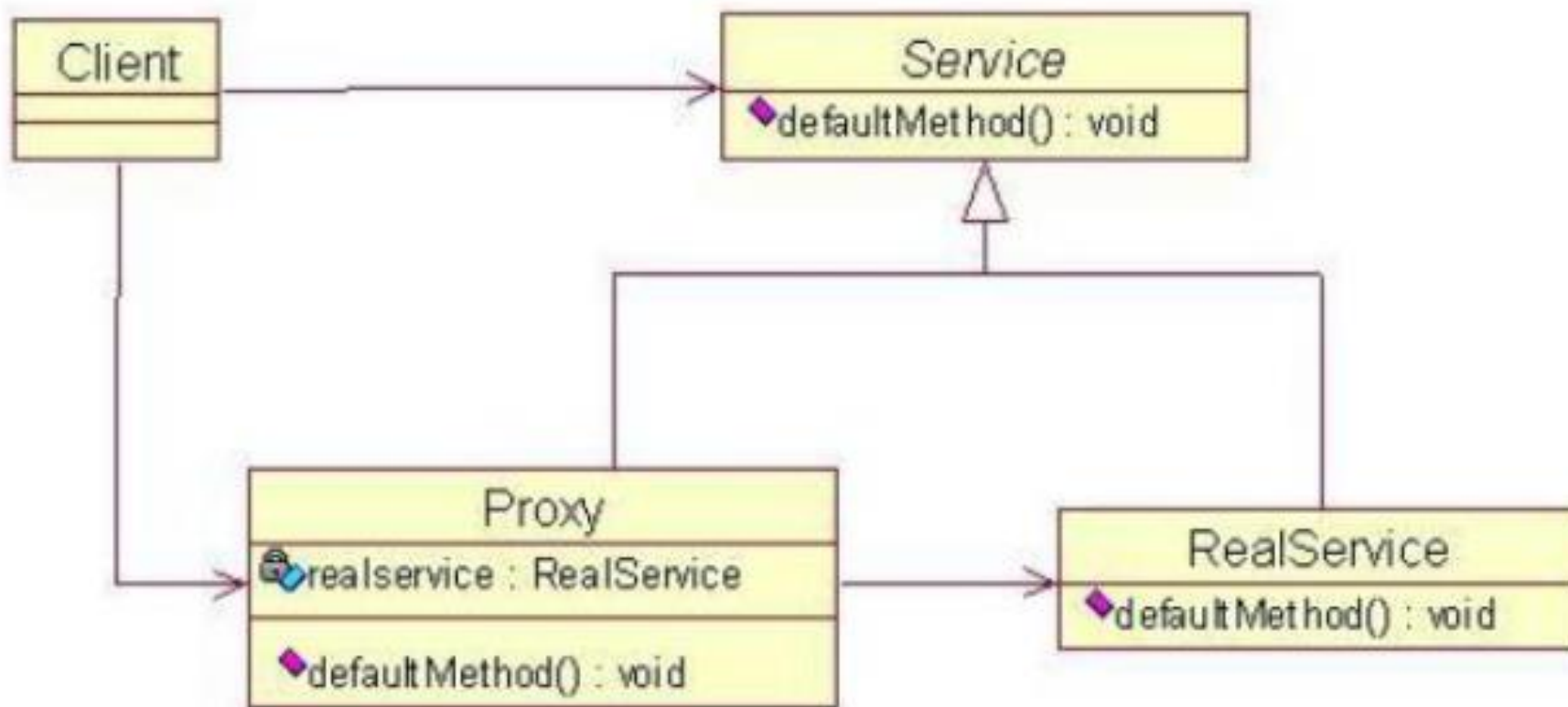


□ Giới thiệu

- Đại diện một đối tượng phức tạp bằng một đối tượng đơn giản, vì các mục đích truy xuất, tốc độ và bảo mật
- Là một mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian

Proxy

□ Cấu trúc mẫu



Proxy

```
public interface Service {
    void defaultMethod();
}
```

```
public class RealService
implements Service {
    // khởi tạo RealService
    public RealService(...) {
        ...
    }
    @Override
    public void defaultMethod() {
        // code của việc sẽ thực hiện
        trong RealService
    }
    ...
}
```

```
public class Proxy implements
Service{
    private RealService
    realService;
    public Proxy (...){
        ...
    }
    @Override
    public void defaultMethod() {
        if(realService == null){
            realService = new
            RealService(...);
        }
        realService.defaultMethod ();
    }
}
```

□ Trong đó:

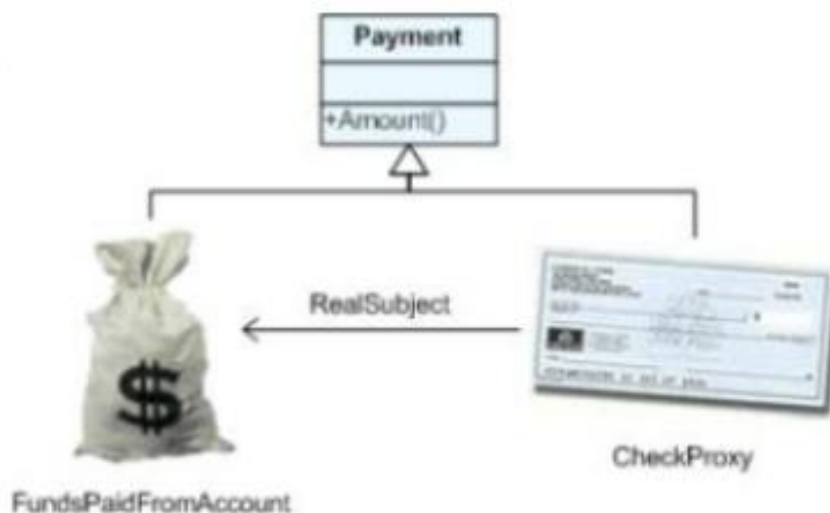
- **Service:** là giao tiếp định nghĩa các phương thức chuẩn cho một dịch vụ nào đó
- **RealService:** là một thực thi của giao tiếp Service, lớp này sẽ khai báo tường minh các phương thức của Service, lớp này xem như thực hiện tốt tất cả các yêu cầu từ Service
- **Proxy:** kế thừa Service và sử dụng đối tượng của RealService

□ Áp dụng

- Sử dụng mẫu Proxy khi bạn cần một tham chiếu phức tạp đến một đối tượng thay vì chỉ một cách bình thường
- Remote proxy – sử dụng khi bạn cần một tham chiếu định vị cho một đối tượng trong không gian địa chỉ (JVM)
- Virtual proxy – lưu giữ các thông tin thêm vào về một dịch vụ thực vì vậy chúng ta có thể hoãn lại sự truy xuất vào dịch vụ này
- Protection proxy – xác thực quyền truy xuất vào một đối tượng thực

Proxy

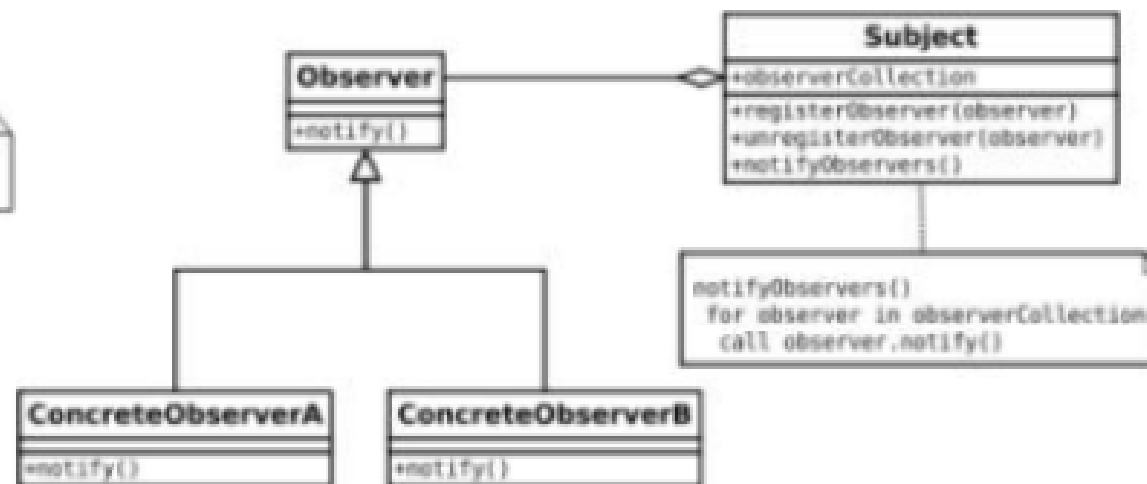
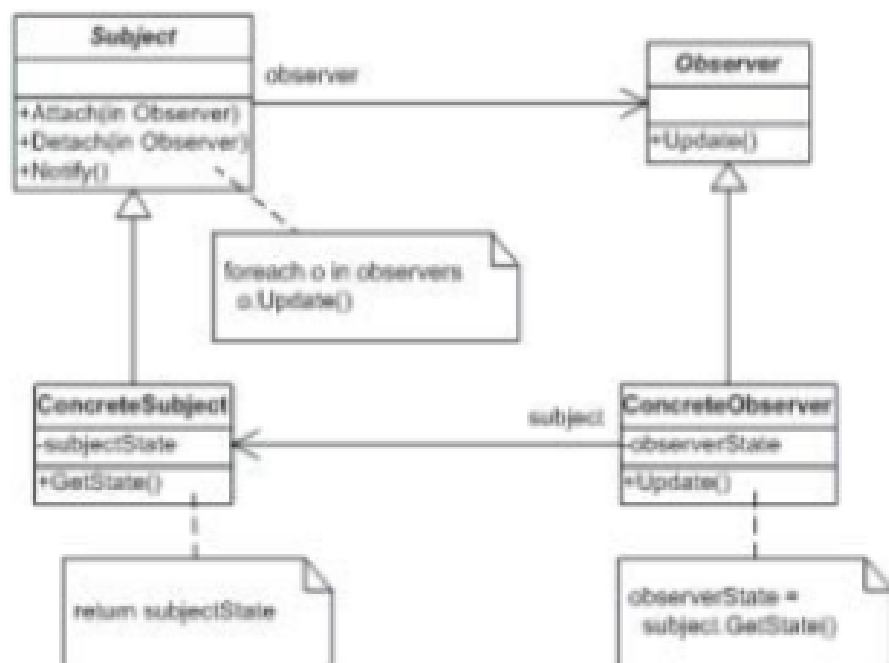
- ❑ **Ví dụ:** Proxy cung cấp một vật thay thế hoặc một nơi lưu trữ để cung cấp quyền truy cập vào một đối tượng. Một tấm séc hay ngân phiếu là một proxy cho các quỹ trong một tài khoản. Tấm séc có thể được sử dụng thay cho tiền mặt khi mua sắm và kiểm soát truy cập tài khoản của người phát hành



□ Giới thiệu

- Observer là một pattern được dùng trong trường hợp ta muốn cài đặt những lớp – đối tượng phụ thuộc vào đối tượng khác, gọi là các đối tượng quan sát(observer) và đối tượng được quan sát (observable).
- Khi trạng thái của đối tượng được quan sát thay đổi thì những đối tượng quan sát sẽ thực hiện hành động nào đó.
- Pattern này cũng được dùng khá phổ biến.

□ Cấu trúc mẫu



□ Trong đó

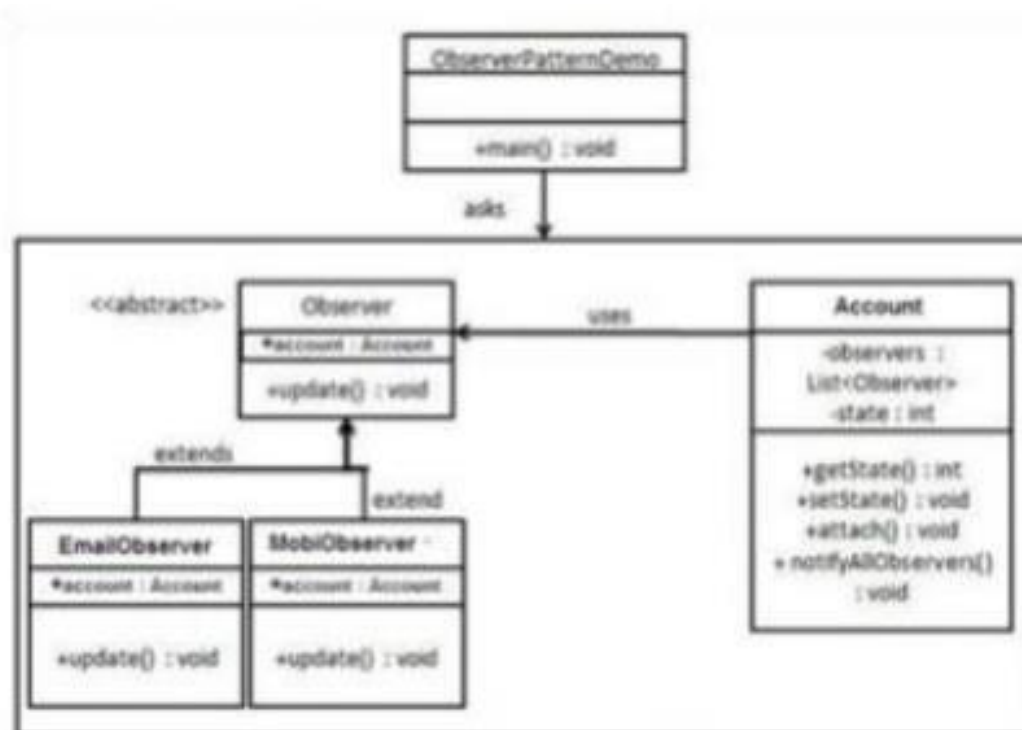
- Observable – interface hoặc abstract class xác định các hoạt động để gắn hoặc gắn lại observer cho client, còn được gọi là Account
- ConcreteObservable – concrete Observable class. Nó duy trì trạng thái của đối tượng, khi có một thay đổi trong trạng thái xuất hiện, nó sẽ thông báo cho các Observer khác kèm theo
- Observer – interface hoặc abstract class định nghĩa các operation được sử dụng để thông báo cho đối tượng này.
- ConcreteObserverA, ConcreteObserverB – lớp cụ thể được kế thừa từ lớp Observer

□ Áp dụng

- Khi một trừu tượng (abstraction) có hai phía cạnh (aspect), cái nó phụ thuộc vào cái kia
- Đóng gói các khía cạnh này trong các đối tượng riêng biệt cho phép thay đổi và tái sử dụng chúng một cách độc lập
- Khi có một thay đổi được đến một đối tượng yêu cầu thay đổi các đối tượng khác.

Observer

- ❑ Ví dụ: Đối tượng được quan sát sẽ là đối tượng Account, khi nó có thay đổi trạng thái thì các đối tượng quan sát là MobiPhone và Email sẽ thực hiện hành động tương ứng là gửi SMS và gửi email thông báo



Observer

```
public class Account {  
    private List<Observer> observers = new ArrayList<Observer>();  
    private int state;  
    public int getState() {  
        return state;  
    }  
    public void setState(int state) {  
        this.state = state;  
        notifyAllObservers();  
    }  
    public void attach(Observer observer) {  
        observers.add(observer);  
    }  
    public void notifyAllObservers() {  
        for (Observer observer : observers) {  
            observer.update();  
        }  
    }  
}
```

Observer

```
public abstract class Observer {  
    protected Account account;  
    public abstract void update();  
}
```

```
public class EmailObserver extends  
Observer{  
    public EmailObserver(Account  
account){  
        this.account = account;  
        this.account.attach(this);  
    }  
    @Override  
    public void update() {  
        // nội dung của Email  
    }  
}
```

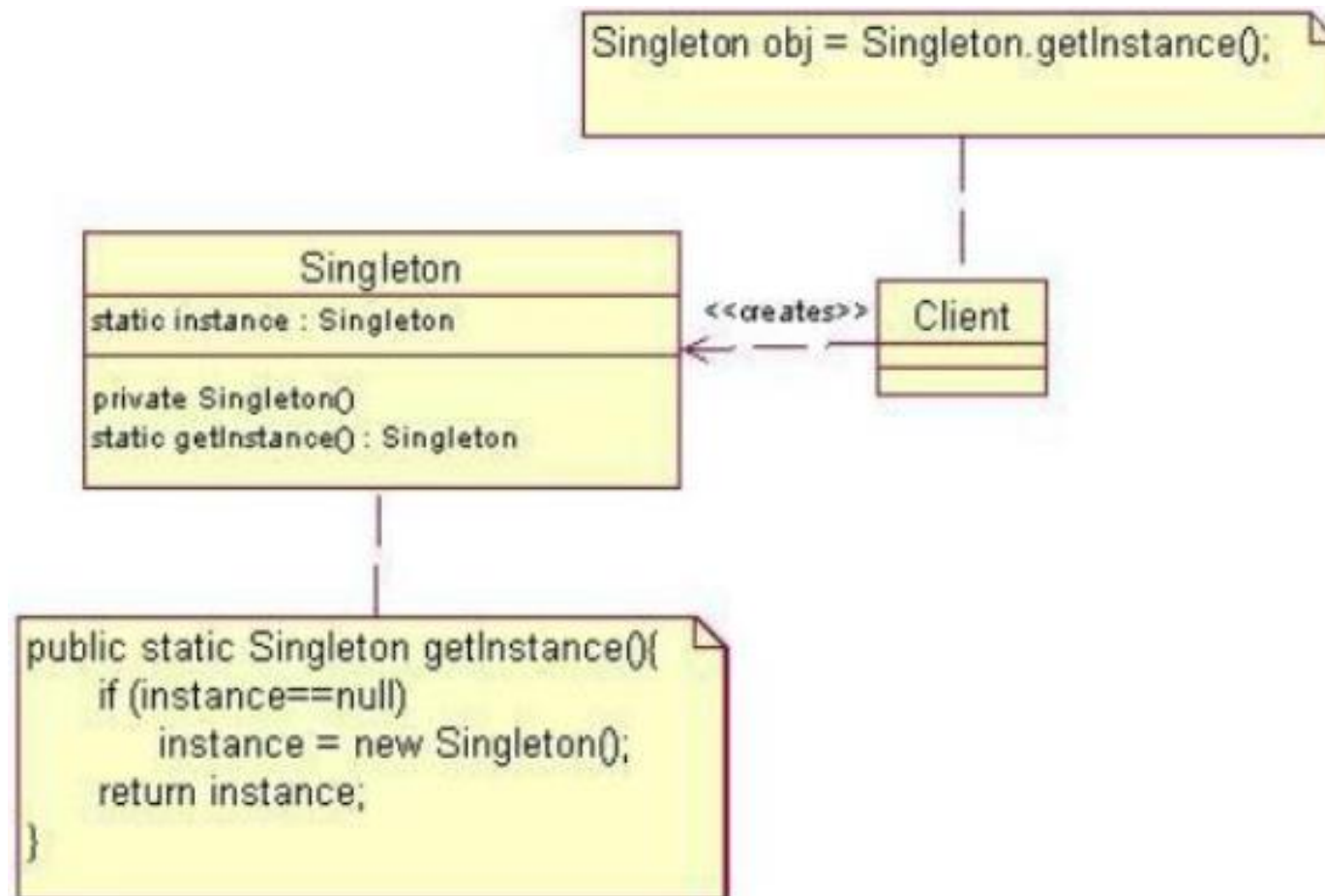
```
public class MobiObserver extends  
Observer{  
    public MobiObserver(Account  
account){  
        this.account = account;  
        this.account.attach(this);  
    }  
    @Override  
    public void update() {  
        // nội dung của Mobi  
    }  
}
```

□ Giới thiệu

- Mẫu này được thiết kế để đảm bảo cho một lớp chỉ có thể tạo duy nhất một thể hiện của nó

Singleton

□ Cấu trúc mẫu



Singleton

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (instance == null)  
            instance = new Singleton;  
        return instance;  
    }  
    ...  
}
```

```
//client  
SingleObject object = SingleObject.getInstance();  
...  

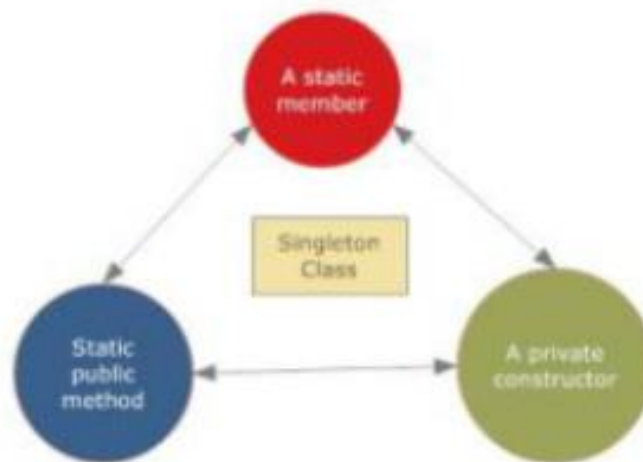
```


Singleton

□ Trong đó

- Singleton cung cấp một phương thức tạo private , duy trì một thuộc tính tĩnh để tham chiếu đến một thể hiện của lớp Singleton này, và nó cung cấp thêm một phương thức tĩnh trả về thuộc tính tĩnh này

Singleton implementation



Singleton

□ Áp dụng

- Khi muốn lớp chỉ có 1 thể hiện duy nhất và nó có hiệu lực ở mọi nơi

Singleton

❑ Ví dụ:

- Bằng cách sử dụng mẫu Singleton, lớp Teacher đã được khởi tạo chỉ một lần và sau đó mỗi đối tượng Student nhận được một tài liệu tham khảo của instance đó

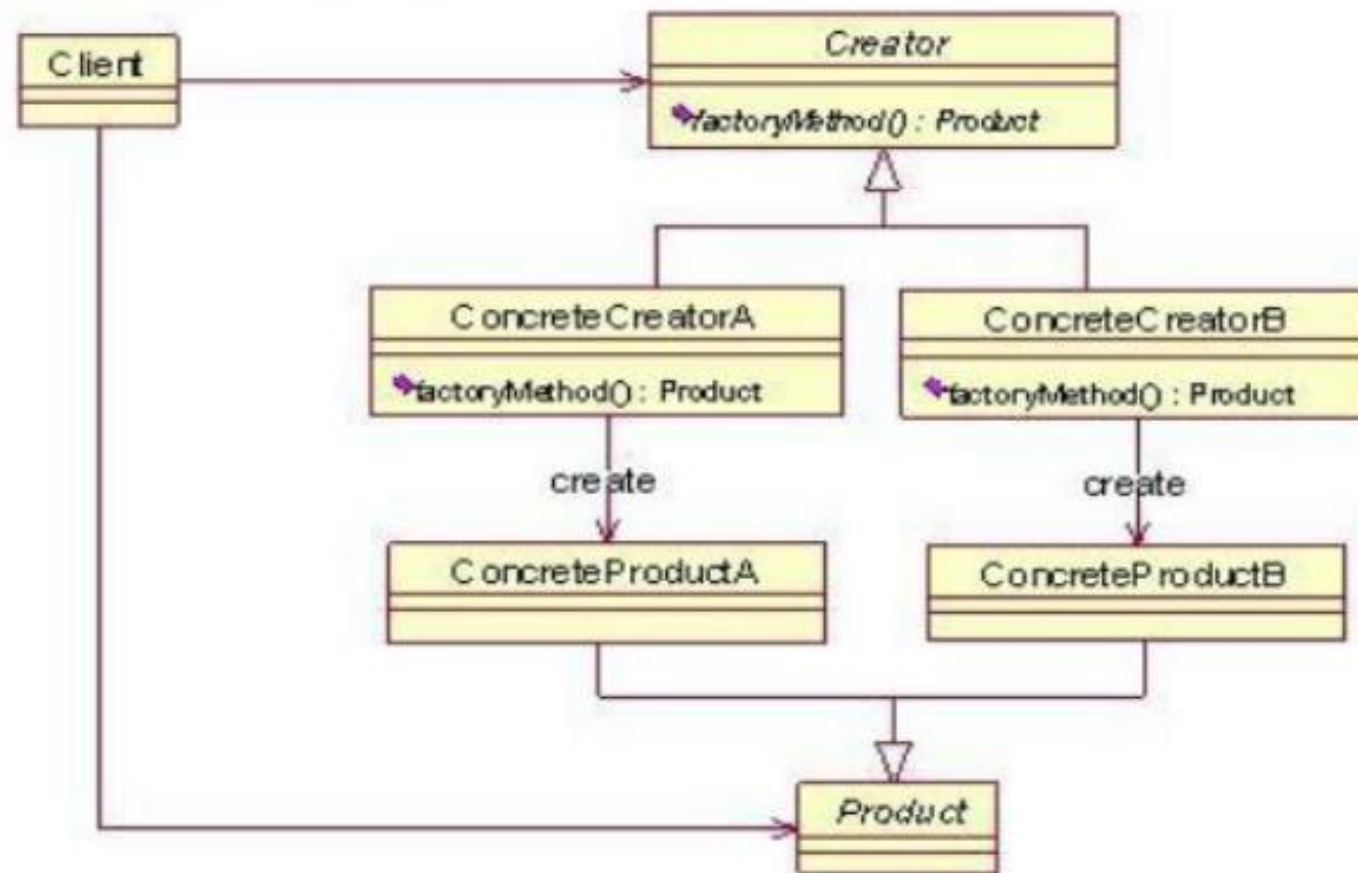


❑ Giới thiệu:

- Bản chất của mẫu thiết kế Factory là “ Định nghĩa một giao diện (interface) cho việc tạo một đối tượng, nhưng để các lớp con quyết định lớp nào sẽ được tạo. “Factory method” giao việc khởi tạo một đối tượng cụ thể cho lớp con

Factory

❑ Cấu trúc mẫu:



Factory

```
public interface Product {  
    void create();  
}
```

```
public class ConcreteProductA implements Product {  
    ...  
    @Override  
    public void create() {  
        // cách tạo ConcreteProductA  
    }  
}
```

```
public class ConcreteProductB implements Product {  
    ...  
    @Override  
    public void create() {  
        // cách tạo ConcreteProductB  
    }  
}
```

Factory

```
public interface Creator{  
    Product factoryMethod();  
}
```

```
public class ConcreteCreatorA implements Creator {  
    ...  
    @Override  
    public Product factoryMethod() {  
        // Trả về ConcreteProductA  
    }  
}
```

```
public class ConcreteCreatorB implements Creator {  
    ...  
    @Override  
    public Product factoryMethod() {  
        // Trả về ConcreteProductB  
    }  
}
```

Factory

□ Trong đó

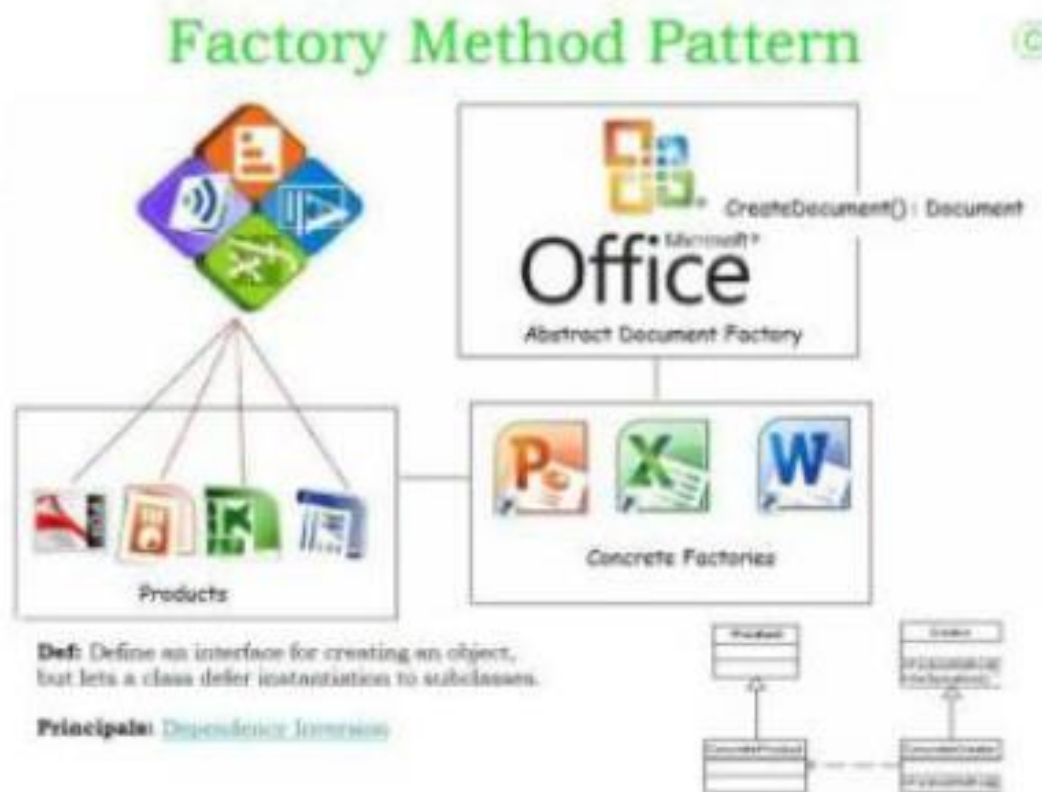
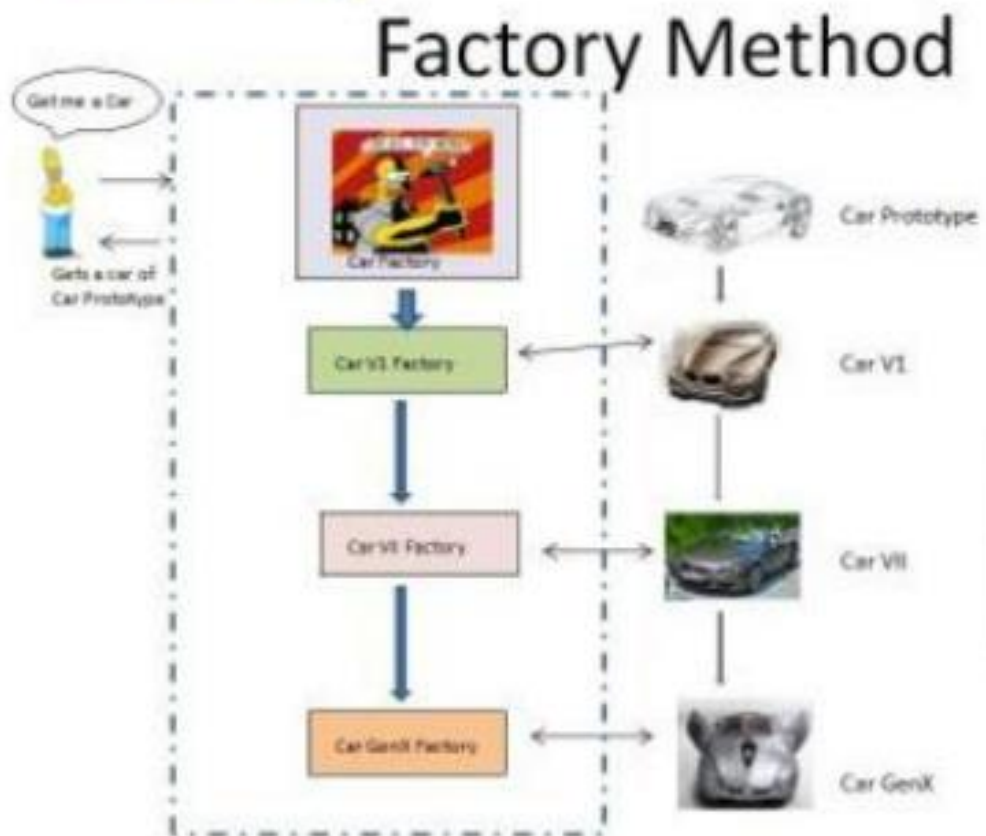
- Creator là lớp trừu tượng, khai báo phương thức factoryMethod() nhưng không cài đặt
- Product cũng là lớp trừu tượng
- ConcreteCreatorA và ConcreteCreatorB là 2 lớp kế thừa từ lớp Creator để tạo ra các đối tượng riêng biệt
- ConcreteProductA và ConcreteProductB là các lớp kế thừa của lớp Product, các đối tượng của 2 lớp này sẽ do 2 lớp ConcreteCreatorA và ConcreteCreatorB tạo ra.

□ Áp dụng

- Khi chúng ta có một lớp “cha” và nhiều lớp “con”, và dựa trên dữ liệu được cung cấp, chúng ta trả về một đối tượng là một trong những lớp “con”
- Khi muốn tạo ra một framework có thể mở rộng, có nghĩa là cho phép tính mềm dẻo trong một số quyết định như chỉ ra loại đối tượng nào được tạo ra
- Khi muốn 1 lớp con, mở rộng từ 1 lớp cha, quyết định lại đối tượng được khởi tạo
- Khi ta biết khi nào thì khởi tạo một đối tượng nhưng không biết loại đối tượng nào được khởi tạo
- Khi cần một vài khai báo chồng phương thức tạo với danh sách các tham số như nhau, điều mà Java không cho phép. Thay vì điều đó ta sử dụng các Factory Method với các tên khác nhau

Factory

□ Ví dụ







T3H

IT-Institute

THANK YOU