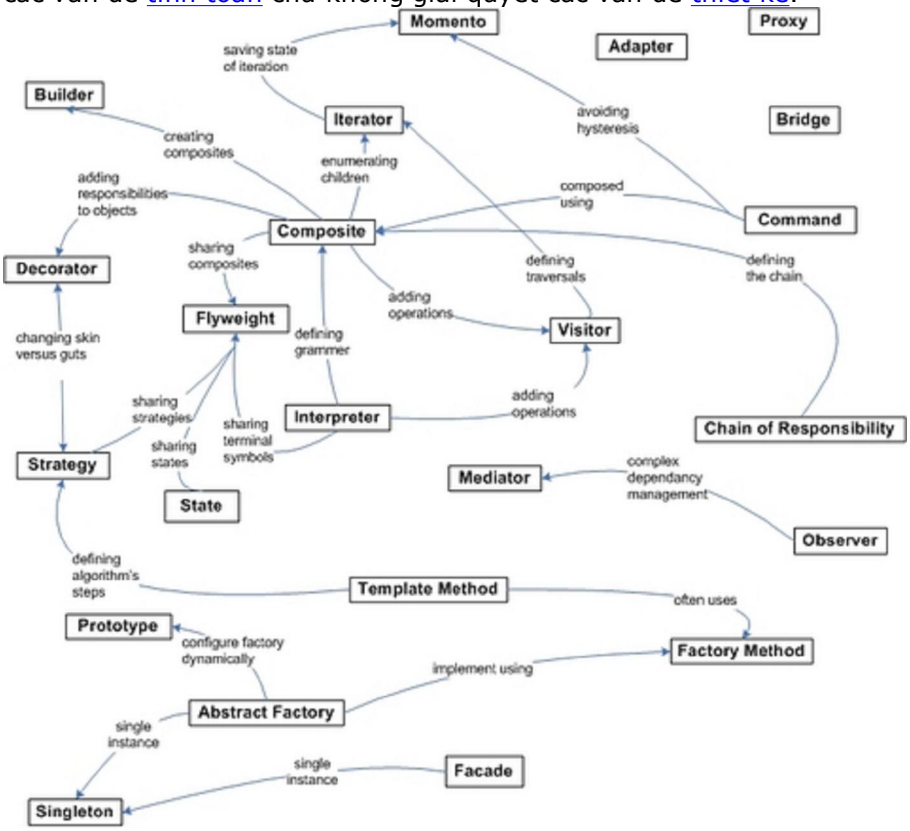
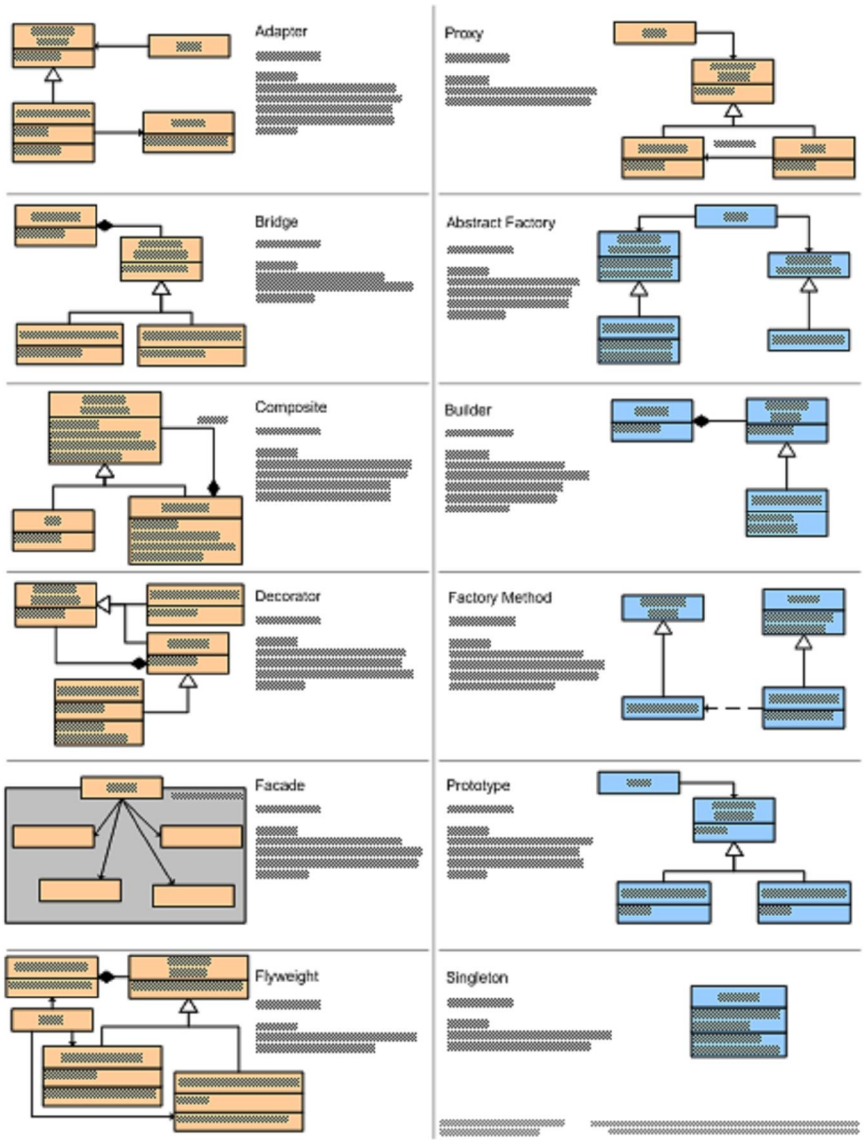
GIỚI THIỆU DESIGN PATTERN

Trong kỹ thuật phần mềm(software engineering), **design pattern** là giải pháp tổng quát có thể dùng lại cho các vấn đề phổ biến trong thiết kế phần mềm. Design pattern không phải là design cuối cùng có thể dùng để chuyển thành code. Nó chỉ là các gợi ý, mẫu mà chỉ ra cách giải quyết vấn đề trong các trường hợp. Các design pattern trong thiết kế hướng đối tượng thường chỉ ra mối quan hệ và tương tác giữa các lớp hay các đối tượng, chứ không chỉ ra các lớp, đối tượng cụ thể nào. Thuật toán không phải design patterns vì chúng chỉ qiải quyết các vấn đề tính toán chứ không giải quyết các vấn đề thiết kế.



Design pattern relationships



Ứng dụng

Design pattern giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm. Thiết kế phần mềm hiệu quả đòi hỏi phải cân nhắc các vấn đề sẽ nảy sinh trong quá trình hiện thực hóa (implementation). Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, đồng thời giúp code dễ đọc hơn.
Thông thường, chúng ta chỉ biết áp dụng một kĩ thuật thiết kế nhất định để giải quyết một bài toán nhất định. Các kĩ thuật này rất khó áp dụng với các vấn đề trong phạm vi rộng hơn. Design pattern cung cấp giải pháp ở dạng tổng quát.
Design pattern gồm các phần như Structure, Participants, Collaboration, .. Các phần này mô tả một *design motif*: là một *micro-architecture* nguyên mẫu mà các developer sẽ lấy và áp dụng vào thiết kế cụ thể của họ. Micro-architecture là tập hợp các thành phần (class, method..) và mối quan hệ giữa chúng. Developer sử dụng design pattern bằng cách đưa các micro-architecture vào trong thiết kế của họ, nghĩa là các micro-architecture trong thiết kế của họ có cấu trúc và cách tổ chức tương tự như trong design motif được chọn.

Hệ thống các mẫu design pattern hiện có 23 mẫu được định nghĩa trong cuốn "Design patterns Elements of Reusable Object Oriented Software". Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại.

Phân loại

Pattern được phân loại ra làm 3 nhóm chính sau đây:

• *Nhóm cấu thành (Creational Pattern)*: Gồm Factory, Abstract Factory, Singleton, Prototype, Builder... Liên quan đến quá trình khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface).

• *Nhóm cấu trúc tĩnh (Structural Pattern)*: Gồm Proxy, Adapter, Wrapper, Bridge, Facade, Flyweight, Visitor... Liên quan đến vấn đề làm thế nào để các lớp và đối tượng kết hợp với nhau tạo thành các cấu trúc lớn hơn.

• *Nhóm tương tác động (Behavioral Pattern)*: Gồm Observer, State, Command, Iterator... Mô tả cách thức để các lớp hoặc đối tượng có thể giao tiếp với nhau.

1 Stuctural Patterns:

- Nhóm cấu trúc tĩnh (Structural Pattern):

- Liên quan đến vấn đề làm thế nào để các lớp và đối tượng kết hợp với nhau tạo thành các cấu trúc lớn hơn.
- Cung cấp cơ chế xử lý những lớp không thể thay đổi , ràng buộc muộn và giảm kết nối giữa các thành phần (late binding and lower coupling) và cung cấp các cơ chế khác để thừa kế.
- Gồm :

STT	Tên	Mục đích
1	**Adapter (adapteur)**	Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp.
2	**Bridge (Pont)**	Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt ; mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau.
3	**Composite**	Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau. Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau -> khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì
4	**Decorator (Décorateur)**	Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).
5	**Facade (Façade)**	Cung cấp một interface thuần nhất cho một tập hợp các interface trong một "hệ thống con" (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn
6	**Flyweight (Poids mouche)**	Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng "cở nhỏ" (chẳng hạn paragraph, dòng, cột, ký tự…)
7	**Proxy (Procuration)**	Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy

2 Creational Patterns :

- Nhóm cấu thành (Creational Pattern): Liên quan đến quá trình khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface). Khắc phục các vấn đề khởi tạo đối tượng, hạn chế sự phụ thuộc platform

STT	Tên	Mục đích
1	**Abstract Factory (Fabrique Abstraite)**	Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng
2	**Builder (Monter)**	Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau
3	**Factory Method (Fabrication)**	Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con

4	**Prototype**	Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu nầy.
5	**Singleton**	Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó

3 Behavioral Patterns :

- Nhóm tương tác động (Behavioral Pattern) : Mô tả cách thức để các lớp hoặc đối tượng có thể giao tiếp với nhau.
- Che dấu hiện thực của đối tượng, che dấu giải thuật , hỗ trợ việc thay đổi cấu hình đối tượng một cách linh động

STT	Tên	Mục đích
1	**Chain of Responsibility(Chaîne de responsabilités)**	Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp; Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi nầy đến khi gặp được đối tượng xử lý nó.Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có có cơ hội xử lý request . liên kết các đối tượng nhận request thành 1 dây chuyền rồi "pass" request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.
	Command(Commande)	Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gởi đi như các đối tượng.Đóng gói request vào trong một Object , nhờ đó có thể nthông số hoá chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo…
2	**Interpreter(Interpreteur)**	Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
3	**Iterator(Itérateur)**	Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, …) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.
4	**Mediator(Médiateur)**	Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.
5	**Memento**	Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.
6	**Observer(Observateur)**	Định nghĩa sự phụ thuộc *một-nhiều* giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.
7	**State(Etat)**	Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi , ta có cảm giác như class của đối tượng bị thay đổi
8	**Strategy**	Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán.Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng
9	**Template method(Patron de méthode)**	Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.
10	**Visitor(Visiteur)**	Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.

Tài liệu tham khảo

- PCWorld – ID: A0506_116 – Thực hiện: Phạm Đình Trường
- http://www.codeproject.com/KB/architecture/CSharpClassFactory.aspx

- Design Patterns – Phương Lan và một số tác giả – Nhà Xuất Bản Phương Đông
- [1] Design Patterns in C# and VB.NET – Gang of Four (GOF) http://www.dofactory.com/Patterns/Patterns.aspx
- [2] Head First Design Pattern – O'REILLY.http://www.oreilly.com
- http://www.oodesign.com
- http://exciton.cs.rice.edu

Editor and Poster: Đặng Thanh Tùng

1.Mẫu kiến tạo(Creational Pattern)

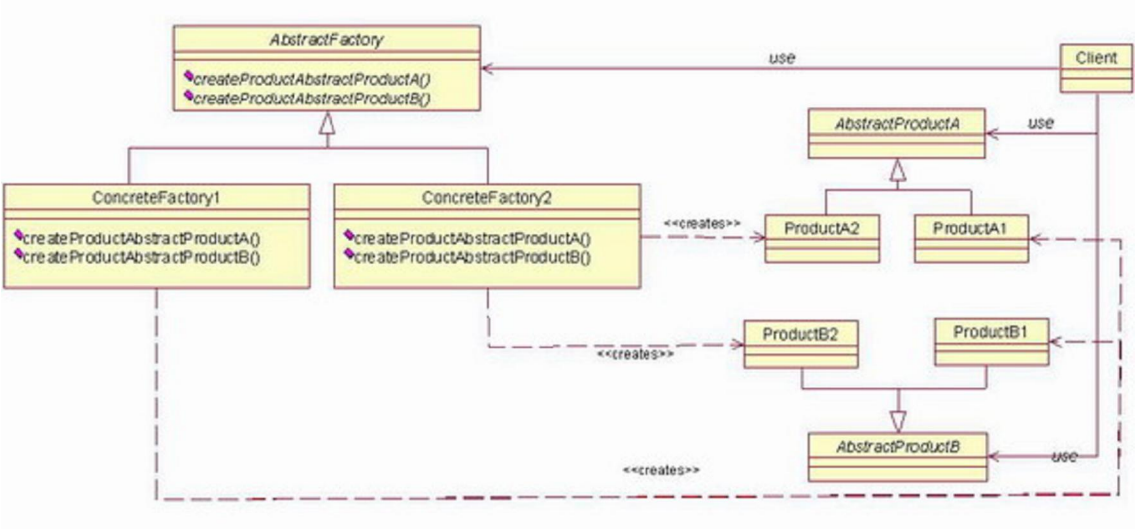
Những mẫu này hỗ trợ cho một trong những nhiệm vụ của lập trình hướng đối tượng – khởi tạo đối tượng trong hệ thống. Hầu hết các hệ thống hướng đối tượng phức tạp yêu cầu nhiều đối tượng được thể hiện theo thời gian, và các mẫu này hỗ trợ cho việc tạo các tiến trình bằng việc cung cấp các khả năng:
- Sự thể hiện chung – Điều này cho phép các đối tượng được tạo ra trong hệ thống không cần phải định nghĩa một đặc tả kiểu lớp trong mã nguồn
- Đơn giản – Một vài mẫu làm cho việc khởi tạo đối tượng trở nên dễ dàng, vì vậy lớp "gọi" khởi tạo đối tượng không phải viết mã nhiều cũng như phức tạp

1.1.Abstract Factory Method Pattern

- **Ý nghĩa**

Đóng gói một nhóm những lớp đóng vai trò "sản xuất" (Factory) trong ứng dụng, đây là những lớp được dùng để tạo lập các đối tượng. Các lớp sản xuất này có chung một giao diện lập trình được kế thừa từ một lớp cha thuần ảo gọi là "lớp sản xuất ảo"

- **Cấu trúc mẫu**



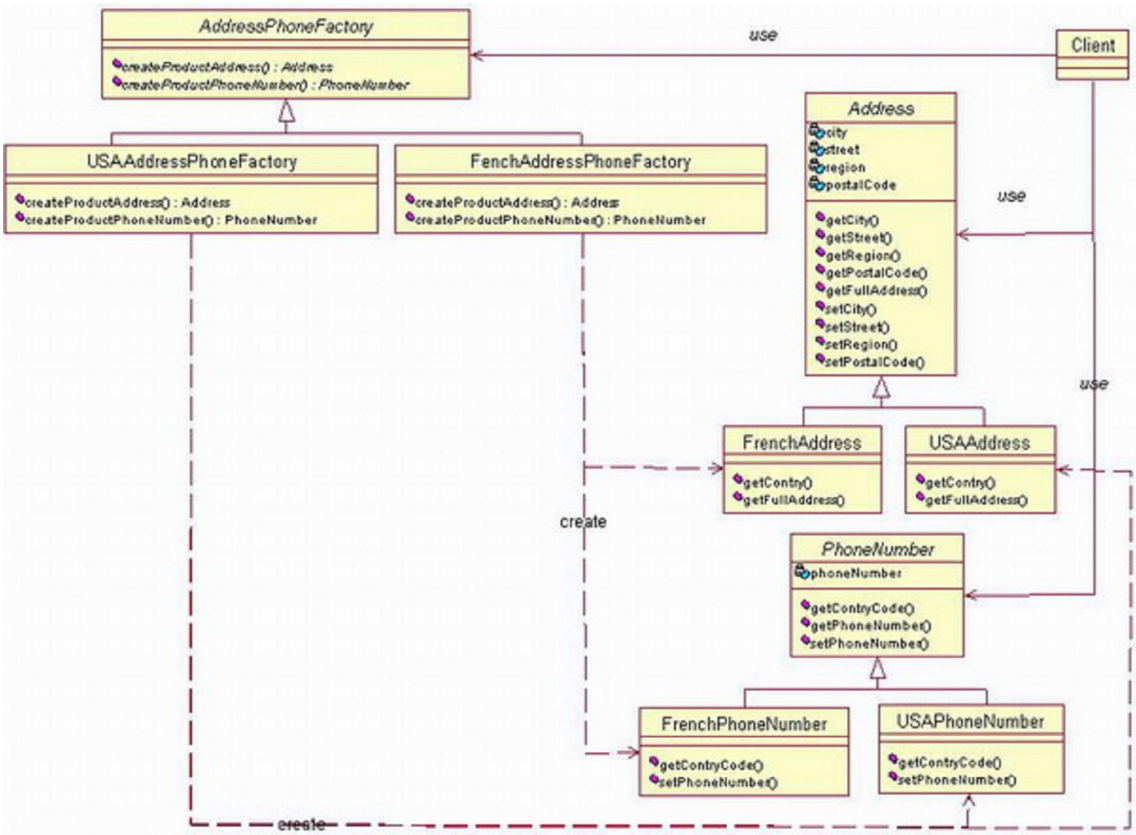
Trong đó:
o **AbstractFactory**: là lớp trừu tượng, tạo ra các đối tượng thuộc 2 lớp trừu tượng là: AbstractProductA và AbstractProductB
o **ConcreteFactoryX**: là lớp kế thừa từ AbstractFatory, lớp này sẽ tạo ra một đối tượng cụ thể
o **AbstractProduct**: là các lớp trừu tượng, các đối tượng cụ thể sẽ là các thể hiện của các lớp dẫn xuất từ lớp này.

- **Tình huống áp dụng**

o Phía trình khách sẽ không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.
o Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm.
o Các đối tượng cần phải được tạo ra như một tập hợp để có thể tương thích với nhau.
o Chúng ta muốn cung cấp một tập các lớp và chúng ta muốn thể hiện các ràng buộc, các mối quan hệ giữa chúng mà không phải là các thực thi của chúng(interface).

- **Ví dụ**

Giả sử ta cần viết một ứng dụng quản lý địa chỉ và số điện thoại cho các quốc gia trên thế giới. Địa chỉ và số địa thoại của mỗi quốc gia sẽ có 1 số điểm giống nhau và 1 số điểm khác nhau. Ta xây dựng sơ đồ lớp như sau:



Ta sẽ xây dựng các phương thức tạo Address, và PhoneNumber cụ thể trong các lớp USAAddressPhoneFactory, FrechAddressPhoneFactory.
Với phương thực createProductAddress() của lớp USAAddressPhoneFactory sẽ trả về đối tượng của lớp USAAddress
Với phương thực createProductAddress() của lớp FrechAddressPhoneFactory sẽ trả về đối tượng của lớp FrechAddress
Tương tự với PhoneNumber.

AddressFactory.java

```
public interface AddressFactory {

    public Address createAddress();

    public PhoneNumber createPhoneNumber();
}
```

Address.java

```
public abstract class Address {

    private String street;
    private String city;
    private String region;
    private String postalCode;
    public static final String EOL_STRING =
            System.getProperty("line.separator");
    public static final String SPACE = " ";

    public String getStreet() {
        return street;
    }

    public String getCity() {
        return city;
    }
```

```
    public String getPostalCode() {
        return postalCode;
    }

    public String getRegion() {
        return region;
    }

    public abstract String getCountry();

    public String getFullAddress() {
        return street + EOL_STRING + city + SPACE + postalCode + EOL_STRING;
    }

    public void setStreet(String newStreet) {
        street = newStreet;
    }

    public void setCity(String newCity) {
        city = newCity;
    }

    public void setRegion(String newRegion) {
        region = newRegion;
    }

    public void setPostalCode(String newPostalCode) {
        postalCode = newPostalCode;
    }
}
```

USAddressFactory.java

```
public class USAddressFactory implements AddressFactory {

    public Address createAddress() {
        return new USAddress();
    }

    public PhoneNumber createPhoneNumber() {
        return new USPhoneNumber();
    }
}
```

USAddress.java

```
public class USAddress extends Address {

    private static final String COUNTRY = "UNITED STATES";
    private static final String COMMA = ",";

    public String getCountry() {
        return COUNTRY;
    }

    public String getFullAddress() {
        return getStreet() + EOL_STRING + getCity() + COMMA + SPACE + getRegion() + SPACE +
getPostalCode() + EOL_STRING + COUNTRY + EOL_STRING;
    }
}
```

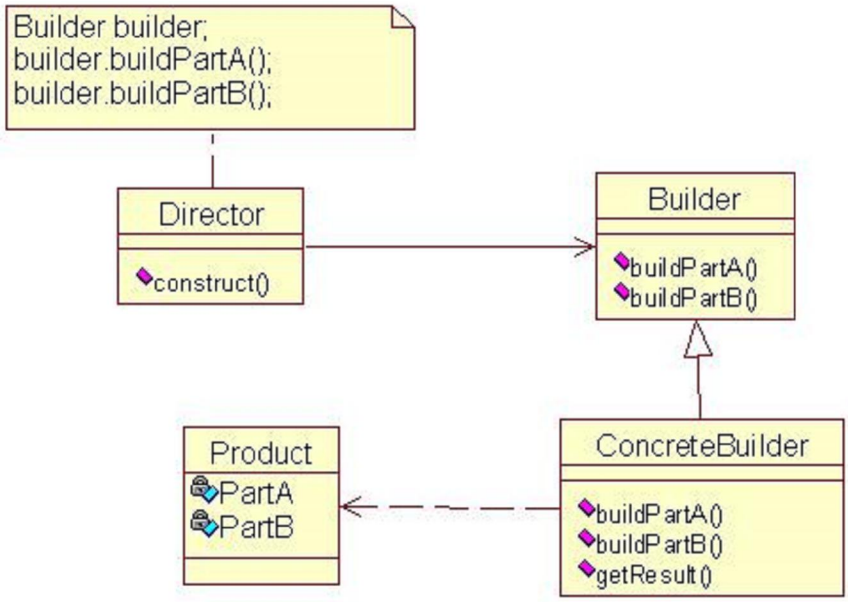
Tương tự cho lớp PhoneNumber và USAPhoneNumber

1.2.Builder Pattern

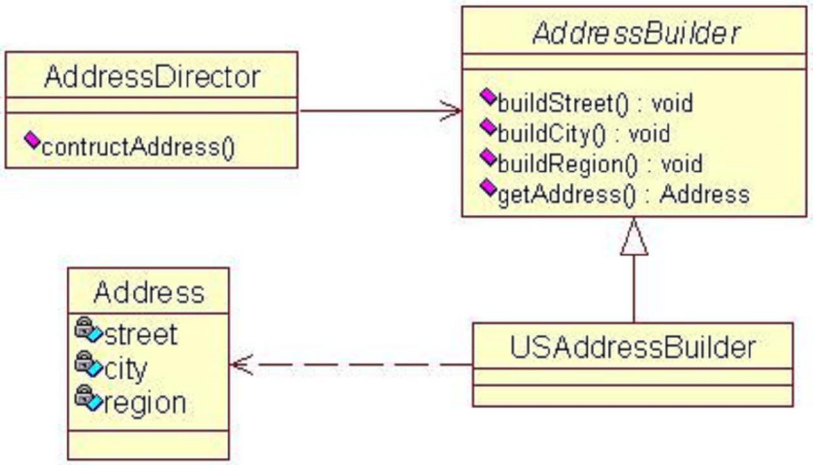
- **Ý nghĩa**

Phân tách những khởi tạo các thành phần của một đối tượng phức hợp, để có thể cùng một khởi tạo mà có thể tạo nên nhiều định dạng khác nhau.

- **Cấu trúc mẫu**



Trong đó:
o **Director**: là lớp điều khiển tạo ra một đối tượng Product
o **Builder**: là lớp trừu tượng cho phép tạo ra đối tượng Product từ các phương thức nhỏ khởi tạo từng thành phần của Product
o **ConcreteBuilder**: là lớp dẫn xuất của Builder, khởi tạo từng đối tượng cụ thể, lớp này sẽ khởi tạo đối tượng.
- **Tình huống áp dụng**
o Có cấu trúc bên trong phức tạp (đặc biệt là một biến là một tập các đối tượng liên quan với nhau)
o Có các thuộc tính phụ thuộc vào các thuộc tính khác
o Sử dụng các đối tượng khác trong hệ thống mà có thể khó khởi tạo hoặc khởi tạo phức tạp
- **Ví dụ**
Ta lại xét đối tượng Address, có các thành phần sau: Street, City và Region. Ta phân tách việc khởi tạo 1 đối tượng Address thành các phần : buildStreet, buildCity và buildRegion.



Trong đó:
o AddressDirector: là lớp tạo ra đối tượng Address
o AddressBuilder: là lớp trừu tượng cho phép tạo ra 1 đối tượng Address bằng các phương thức khởi tạo từng thành phần của Address
o USAddressBuilder: là lớp tạo ra các Address. USAddressBuilder sẽ tạo ra địa chỉ theo chuẩn của USA
Address.java

```
class Address {

    private String street;
    private String city;
    private Stringregion ;
```

```
    /**
     * @return the city
     */
    public String getCity() {
        return city;
    }

    /**
     * @param city the city to set
     */
    public void setCity(String city) {
        this.city = city;
    }

    /**
     * @return the region
     */
    public String getRegion() {
        return region;
    }

    /**
     * @param region the region to set
     */
    public void setRegion(String region) {
        this.region = region;
    }

    /**
     * @return the street
     */
    public String getStreet() {
        return street;
    }

    /**
     * @param street the street to set
     */
    public void setStreet(String street) {
        this.street = street;
    }
}
```

AddressBuilder.java

```
abstract class AddressBuilder {

    abstract public void buildStreet(String street) {
    }

    abstract public void buildCity(String city) {
    }

    abstract public void buildRegion(String region) {
    }
}
```

USAddressBuilder.java

```
class USAddressBuilder extends AddressBuilder {

    private Address add;

    public void buildStreet(String street) {
        add.setStreet(street);
    }
```

```
    public void buildCity(String city) {
        add.setCity(city);
    }

    public void buildRegion(String region) {
        add.setRegion(region);
    }

    public Address getAddress() {
        return add;
    }
}
```

AddressDirector.java

```
class AddressDirector {

    public void Contruct(AddressBuilder builder, String street, String city, String region) {
        builder.buildStreet(street);
        builder.buildCity(city);
        builder.buildRegion(region);
    }
}
```

Client.java

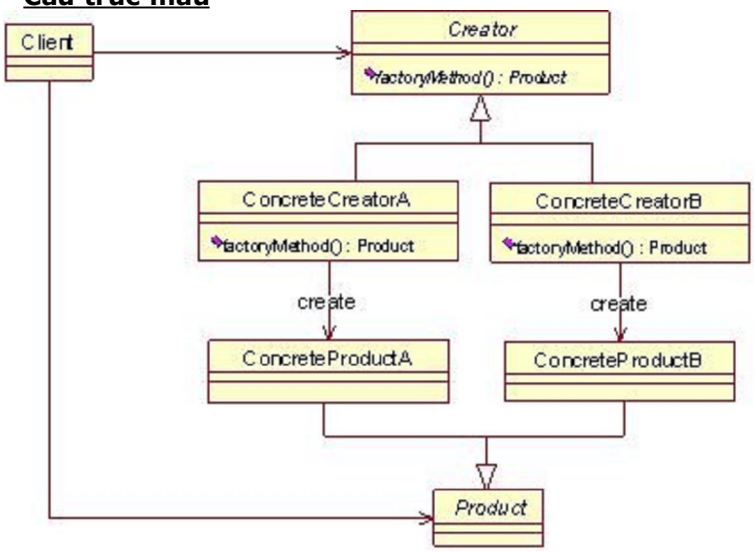
```
class Client {

    AddressDirector director = new AddressDirector();
    USAddressBuilder b = new USAddressBuilder();

    director.Contruct (b, "Street 01", "City 01", "Region 01");
    Address add = b.getAddress();
}
```

1.3.Factory Method

- **Ý nghĩa**

Định nghĩa một phương thức chuẩn để khởi tạo đối tượng, như là một phần của phương thức tạo, nhưng việc quyết định kiểu đối tượng nào được tạo ra thì phụ thuộc vào các lớp con

- **Cấu trúc mẫu**



Trong đó:

o **Creator** là lớp trừu tượng, khai báo phương thức factoryMethod() nhưng không cài đặt

o **Product** cũng là lớp trừu tượng

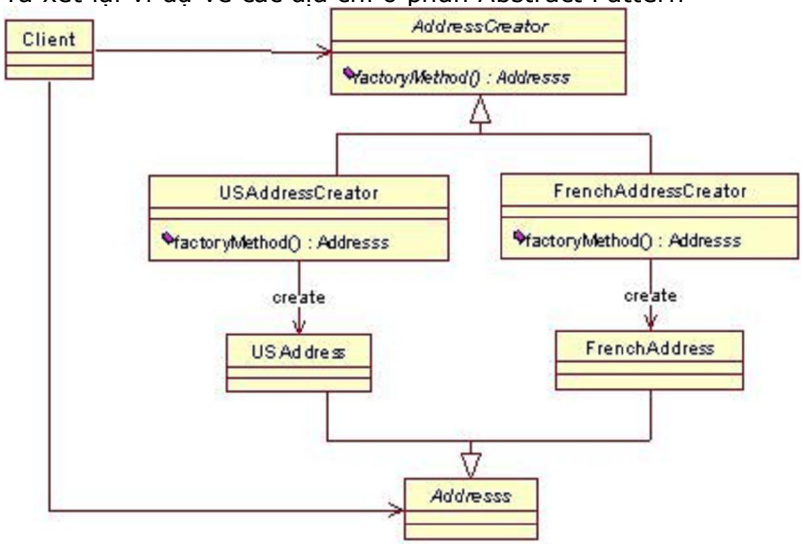
o **ConcreteCreatorA** và **ConcreteCreatorB** là 2 lớp kế thừa từ lớp Creator để tạo ra các đối tượng riêng biệt

o **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

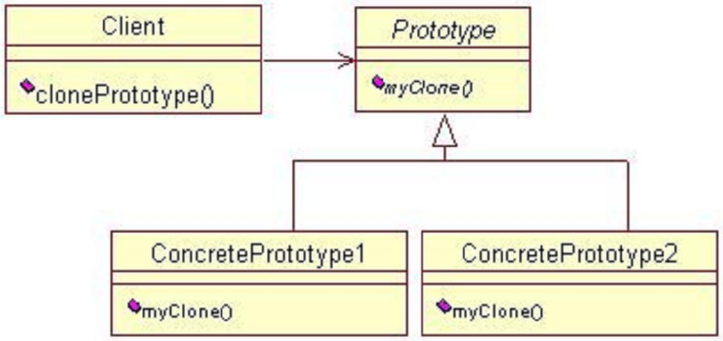
- **Tình huống áp dụng**

o Khi bạn muốn tạo ra một framework có thể mở rộng, có nghĩa là nó cho phép tính mềm dẻo trong một số

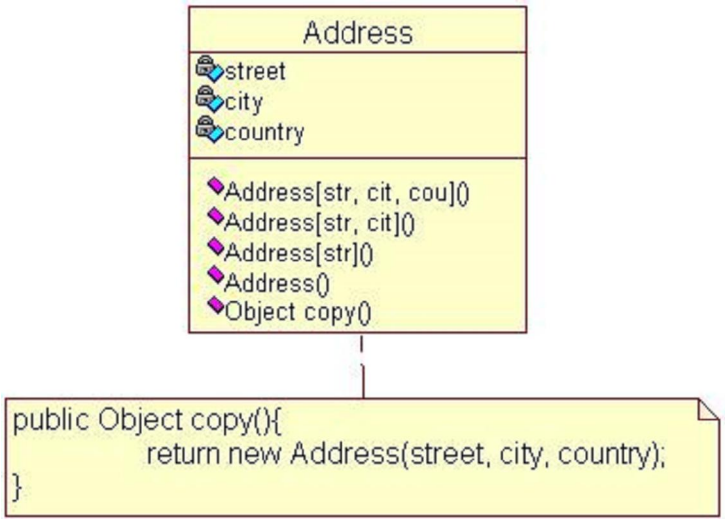
quyết định như chỉ ra loại đối tượng nào được tạo ra
o Khi bạn muốn 1 lớp con, mở rộng từ 1 lớp cha, quyết định lại đối tượng được khởi tạo
o Khi bạn biết khi nào thì khởi tạo một đối tượng nhưng không biết loại đối tượng nào được khởi tạo
o Bạn cần một vài khai báo chồng phương thức tạo với danh sách 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
- **Ví dụ**
Ta xét lại ví dụ về các địa chỉ ở phần Abstract Pattern



1.4.Prototype
- **Ý nghĩa**
Giúp khởi tạo đối tượng bằng cách copy một đối tượng khác đã tồn tại (đối tượng này là "prototype" – nguyên mẫu).
- **Cấu trúc mẫu**



Trong đó:
o **Prototype** là lớp trừu tượng cài đặt phương thức myClone() là phương thức copy bản thân đối tượng đã tồn tại.
o **ConcretePrototype1** và **ConcretePrototype2** là các lớp kế thừa lớp Prototype.
- **Tình huống áp dụng**
o Khi bạn muốn khởi tạo một đối tượng bằng cách sao chép từ một đối tượng đã tồn tại
- **Ví dụ**

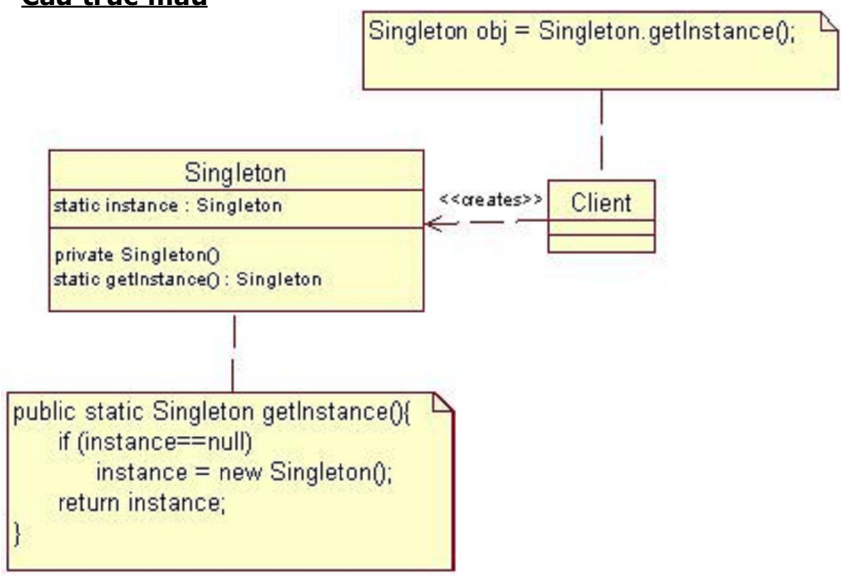


1.5.Singleton

- **Ý nghĩa**

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

- **Cấu trúc mẫu**



Trong đó:

o **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

- **Tình huống áp dụng**

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

- **Ví dụ**

```
public class Singleton {

    private String _strName;
    private static Singleton instance;

    private Singleton(String name) {
        _strName = name;
    }

    public static Singleton getInstance(String name) {
        if (instance == null) {
            instance = new Singleton(name);
        }
        return instance;
    }

    public void printName() {
        System.out.println(this._strName);
```

```
        }
}
```

2.BEHAVIORAL PATTERNS

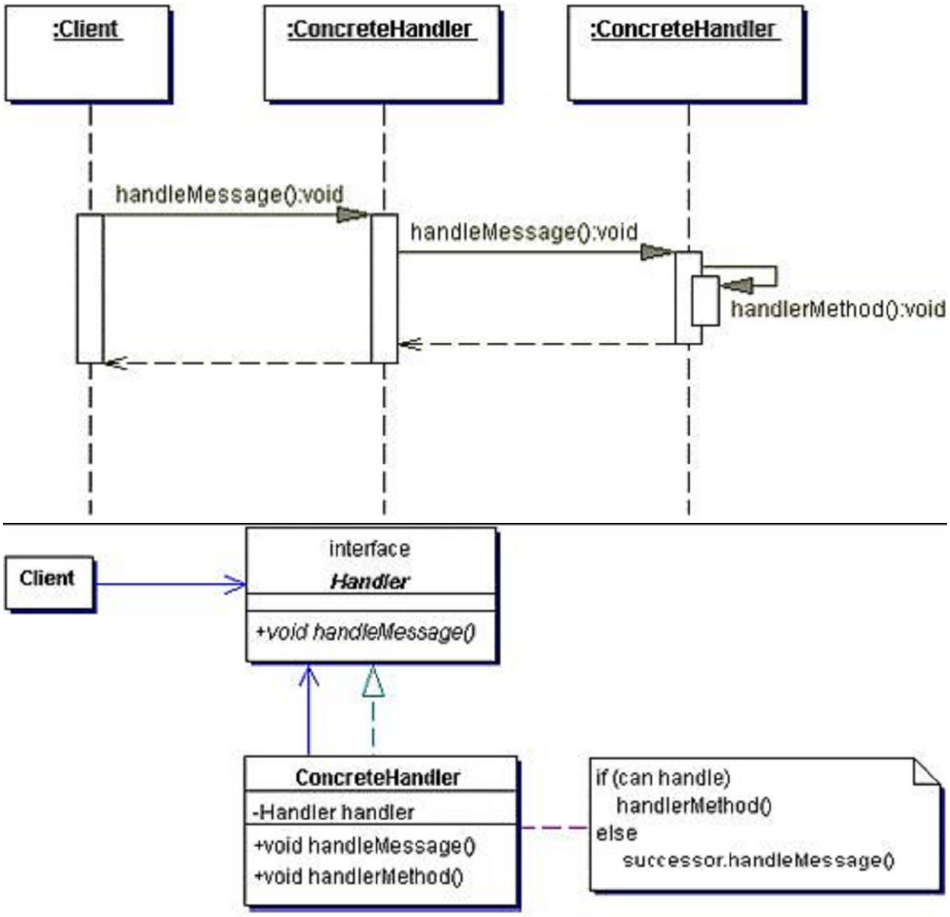
Mẫu Behavioral có liên quan đến luồng điều khiển của hệ thống. Một vài cách của tổ chức điều khiển bên trong một hệ thống để có thể nâng mang lại các lợi ích cả về hiệu suất lẫn khả năng bảo trì hệ thống đó

2.1.Chain of Responsibility

- Ý nghĩa

Mẫu này thiết lập một chuỗi bên trong một hệ thống, nơi mà các thông điệp hoặc có thể được thực hiện ở tại một mức nơi mà nó được nhận lần đầu hoặc là được chuyển đến một đối tượng mà có thể thực hiện điều đó

- Mô hình mẫu



Trong đó:

o **Handler**: là một giao tiếp định nghĩa phương thức sử dụng để chuyển thông điệp qua các lần thực hiện tiếp theo.

o **ConcreteHandler**: là một thực thi của giao tiếp Handler. Nó giữ một tham chiếu đến một Handler tiếp theo. Việc thực thi phương thức handleMessage có thể xác định làm thế nào để thực hiện phương thức và gọi một handlerMethod, chuyển tiếp thông điệp đến cho Handler tiếp theo hoặc kết hợp cả hai

- Trường hợp ứng dụng

o Có một nhóm các đối tượng trong một hệ thống có thể đáp ứng tất cả các loại thông điệp giống nhau

o Các thông điệp phải được thực hiện bởi một vài các đối tượng trong hệ thống

o Các thông điệp đi theo mô hình "thực hiện – chuyển tiếp", một vài sự kiện có thể được thực hiện tại mức mà chúng được nhận hoặc tại ra, trong khi số khác phải được chuyển tiếp đến một vìa đối tượng khác

- Ví dụ mẫu

Hệ thống quản lý thông tin các nhân có thể được sử dụng để quản lý các dự án như là các liên hệ.

Ta hình dung một cấu trúc cây như sau; đỉnh là một dự án, các đỉnh con là các tác vụ của dự án đó, và cứ như vậy, mỗi đỉnh con tác vụ lại có một tập các đỉnh con tác vụ khác.

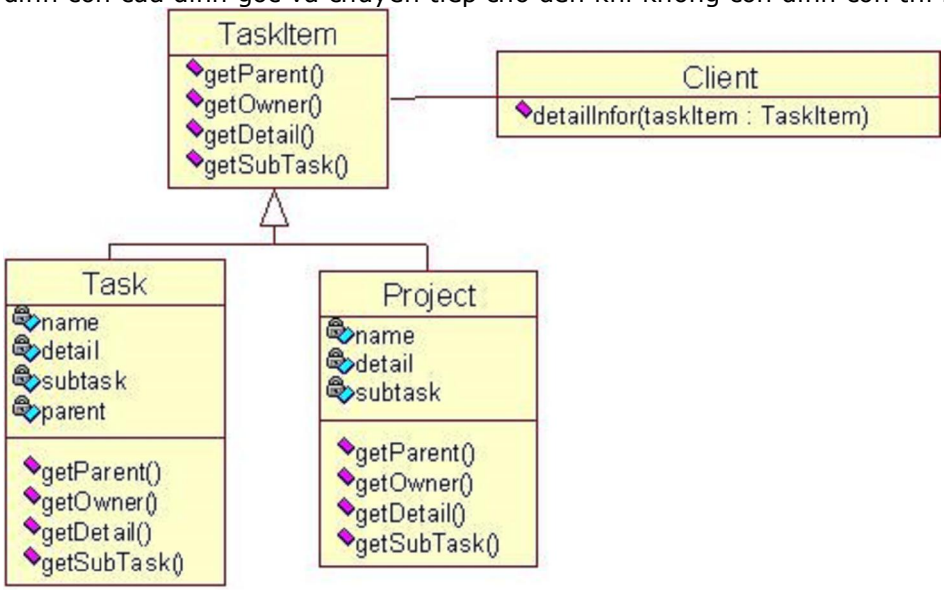
Để quản lý cấu trúc này ta thực hiện như sau:

-Ở mỗi đỉnh ta lưu các thông tin như sau: tên tác vụ, đỉnh cha, tập các đỉnh con

-Ta xét thông điệp sau: duyệt từ đỉnh gốc (project cở sở) in ra các thông tin

-Như vậy với thông điệp này, việc in thông tin ở một đỉnh là chưa đủ, ta phải chuyển tiếp đến in thông tin các

đỉnh con của đỉnh gốc và chuyển tiếp cho đến khi không còn đỉnh con thì mới dừng



Giao tiếp TaskItem định nghĩa các phương thức cho project cơ sở và các tác vụ

```
public interface TaskItem {

    public TaskItem getParent();

    public String getDetails();

    public ArrayList getProjectItems();
}

//Lớp Project thực thi giao tiếp TaskItem, nó là lớp đại diện cho các đỉnh gốc trên cùng của
cây
public class Project implements TaskItem {

    private String name;
    private String details;
    private ArrayList subtask = new ArrayList();

    public Project() {
    }

    public Project(String newName, String newDetails) {
        name = newName;
        details = newDetails;
    }

    public String getName() {
        return name;
    }

    public String getDetails() {
        return details;
    }

    public ProjectItem getParent() {
        return null;
//vì project là ở mức cơ sở, đỉnh gốc trên cùng nên không có cha
    }

    public ArrayList getSubTask() {
        return subtask;
    }

    public void setName(String newName) {
        name = newName;
    }
```

```
    public void setDetails(String newDetails) {
        details = newDetails;
    }

    public void addTask(TaskItem element) {
        if (!subtask.contains(element)) {
            subtask.add(element);
        }
    }

    public void removeProjectItem(TaskItem element) {
        subtask.remove(element);
    }
}

//Lớp Task thực thi giao tiếp TaskItem, nó đại diện cho các tác vụ, các đỉnh không phải ở gốc
của cây
public class Task implements TaskItem {

    private String name;
    private ArrayList subtask = new ArrayList();
    private String details;
    private TaskItem parent;

    public Task(TaskItem newParent) {
        this(newParent, "", "");
    }

    public Task(TaskItem newParent, String newName, String newDetails, ) {
        parent = newParent;
        name = newName;
        details = newDetails;
    }

    public String getDetails() {
        if (primaryTask) {
            return details;
        } else {
            return parent.getDetails() + EOL_STRING + "\t" + details;
        }
    }

    public String getName() {
        return name;
    }

    public ArrayList getSubTask() {
        return subtask;
    }

    public ProjectItem getParent() {
        return parent;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void setParent(TaskItem newParent) {
        parent = newParent;
    }

    public void setDetails(String newDetails) {
        details = newDetails;
    }
```

```
    public void addSubTask(TaskItem element) {
        if (!subtask.contains(element)) {
            subtask.add(element);
        }
    }

    public void removeSubTask(TaskItem element) {
        subtask.remove(element);
    }
}

//Lớp thực thi test mẫu
public class RunPattern {

    public static void main(String[] arguments) {
        Project project = new Project("Project 01", "Detail of Project 01");
    //Khởi tạo, thiết lập các tác vụ con …
    detailInfor(project);
}


private static void detailInfor (TaskItem item){
System.out.println("TaskItem: " + item);
        System.out.println(" Details: " + item.getDetails());
        System.out.println();
        if (item.getSubTask() != null) {
            Iterator subElements = item.getSubTask().iterator();
            while (subElements.hasNext()) {
                detailInfor((TaskItem) subElements.next());
            }
        }
    }
}
```

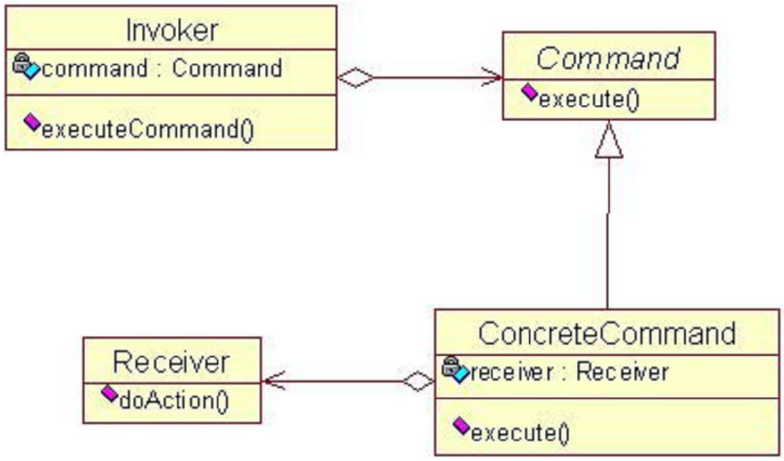
Gọi thông điệp detailInfor(item) và thông điệp này được chuyển tiếp nhiều lần qua nhiều đối tượng thực thi

2.2.Command Pattern

- Ý nghĩa

Gói một mệnh lệnh vào trong một đối tượng mà nó có thể được lưu trữ, chuyển vào các phương thức và trả về một vài đối tượng khác

- Cấu trúc mẫu



Trong đó:

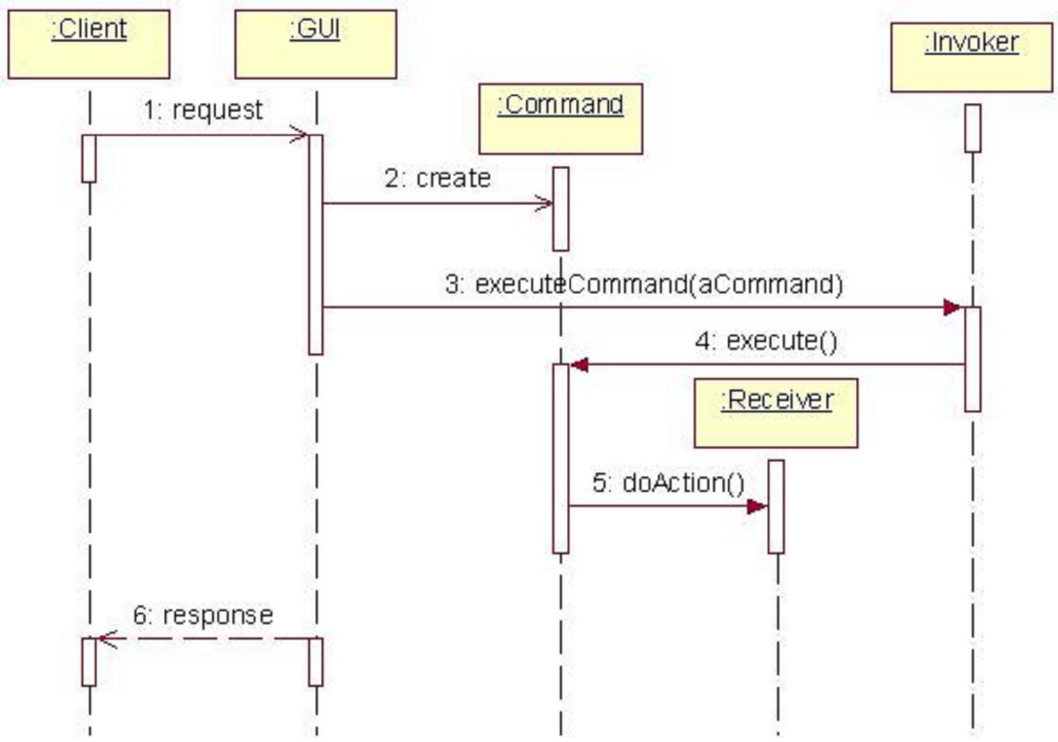
o **Command**: là một giao tiếp định nghĩa các phương thức cho Invoker sử dụng

o **Invoker**: lớp này thực hiện các phương thức của đối tượng Command

o **Receiver**: là đích đến của Command và là đối tượng thực hiện hoàn tất yêu cầu, nó có tất cả các thông tin cần thiết để thực hiện điều này

o **ConcreteCommand**: là một thực thi của giao tiếp Command. Nó lưu giữa một tham chiếu Receiver mong muốn

Luồng thực thi của mẫu Command như sau:



o Client gửi yêu cầu đến GUI của ứng dụng
o Ứng dụng khởi tạo một đối tượng Command thích hợp cho yêu cầu đó (đối tượng này sẽ là các ConcreteCommand)
o Sau đó ứng dụng gọi phương thức executeCommand() với tham số là đối tượng Command vừa khởi tạo
o Invoker khi được gọi thông qua phương thức executeCommand() sẽ thực hiện gọi phương thức execute() của đối tượng Command tham số
o Đối tượng Command này sẽ gọi tiếp phương thức doAction() của thành phần Receiver của nó, được khởi tạo từ đầu, doAction() chính là phương thức chính để hoàn tất yêu cầu của Client

- Trường hợp áp dụng

o Hỗ trợ undo, logging hoặc transaction
o Thực hiện hàng đợi lệnh và thực hiện lệnh tại các thời điểm khác nhau
o Hạn chế sự chặt chẽ của yêu cầu với đối tượng thực hiện hoàn tất yêu cầu đó

- Ví dụ mẫu

```
public interface Command {

    public void execute();
}

public class ConcreteCommand implements Command {

    private Receiver receiver;

    public void setReceiver(Receiver receiver) {
        this.receiver = receiver;
    }

    public Receiver getReceiver() {
        return this.receiver;
    }

    public void execute() {
        receiver.doAction();
    }
}

public class Receiver {

    private String name;
```

```
    public Receiver(String name) {
        this.name = namel
    }

    public void doAction() {
        System.out.print(this.name + "fulfill request !");
    }
}

public class Invoker {

    public void executeCommand(Command command) {
        command.execute();
    }
}

public class Run {

    public static void main(String[] agrs) {
        Command command = new ConcreteCommand();
        command.setReceiver(new Receiver("NguyenD"));

        Invoker invoker = new Invoker();
        Invoker.executeCommand(command);
    }
}
```

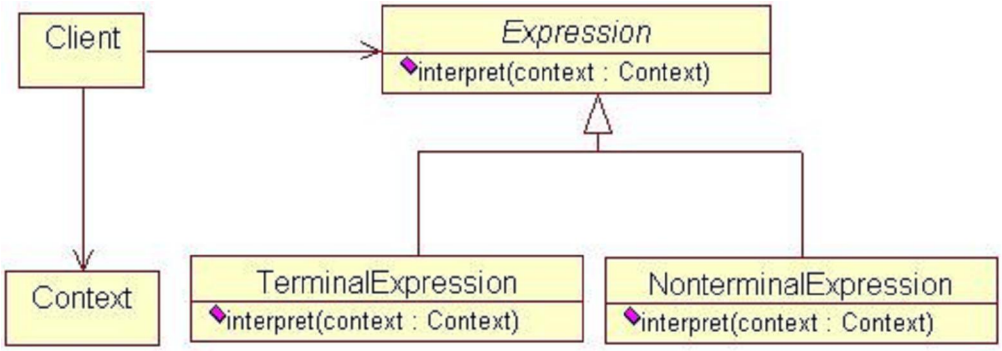
2.3.Interpreter Pattern

- **Ý nghĩa**

o Ý tưởng chính của Interpreter là triển khai ngôn ngữ máy tính đặc tả để giải quyết nhanh một lớp vấn đề được định nghĩa. Ngôn ngữ đặc tả thường làm cho vấn đề được giải quyết nhanh hơn ngôn ngữ thông thường từ một cho đến vài trăm lần

o Ý tưởng tương tự như vậy biểu diễn các biểu thức tính toán theo cú pháp Ba Lan

- **Mô hình mẫu**



Trong đó:

o **Expression**: là một giao tiếp mà thông qua nó, client tương tác với các biểu thức

o **TerminalExpression**: là một thực thi của giao tiếp Expression, đại diện cho các nốt cuối trong cây cú pháp

o **NonterminalExpression**: là một thực thi khác của giao tiếp Expression, đại diện cho các nút chưa kết thúc trong cấu trúc của cây cú pháp. Nó lưu trữ một tham chiếu đến Expression và triệu gọi phương thức diễn giải cho mỗi phần tử con

o **Context**: chứa thông tin cần thiết cho một vài vị trí trong khi diễn giải. Nó có thể phục vụ như một kênh truyền thông cho các thể hiện của Expression

o **Client**: hoặc là xây dựng hoặc là nhận một thể hiện của cây cú pháp ảo. Cây cú pháp này bao gồm các thể hiện của TerminalExpression và NoterminalExpression để tạo nên câu đặc tả. Client triệu gọi các phương thức diễn giải với ngữ cảnh thích hợp khi cần thiết

- **Trường hợp ứng dụng**

o Có một ngôn ngữ đơn giản để diễn giải vấn đề

o Các vấn đề lặp lại có thể được diễn giải nhanh bằng ngôn ngữ đó

- **Ví dụ mẫu**

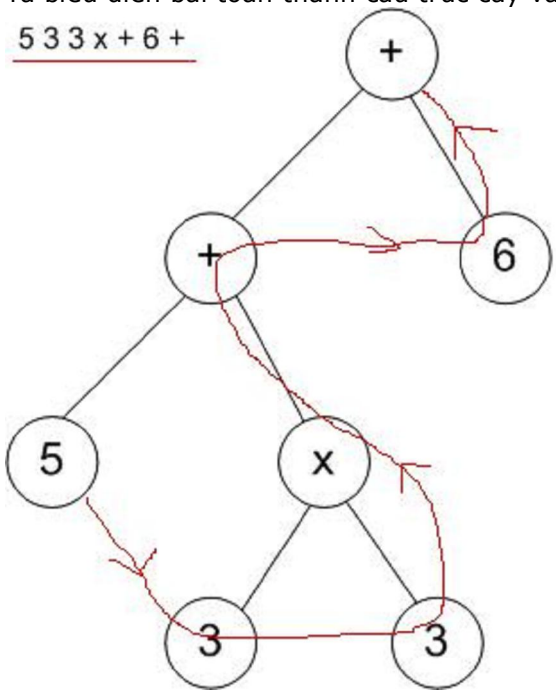
Xét biểu thức 5 + 3 x 3 + 6, với bài tóan này ta có thể chia thành các bài các bài tóan nhỏ hơn

-Tính 3 x 3 = a

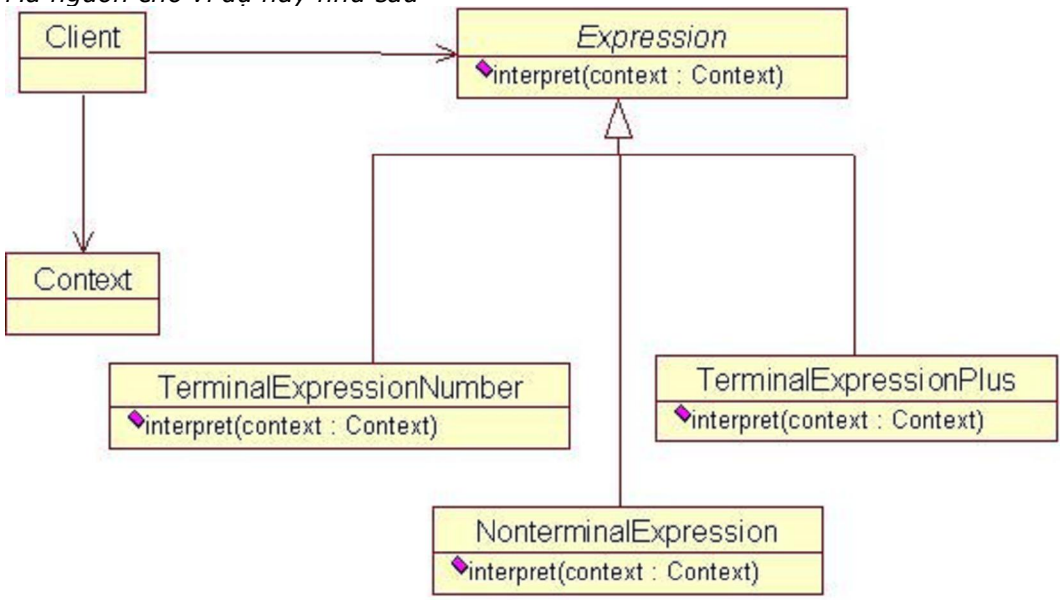
-Sau đó tính 5 + a = b

-Sau đó tính b + 6

Ta biểu diễn bài tóan thành cấu trúc cây và duyệt cây theo Ba Lan (hay Ba Lan đảo gì đó không còn nhớ nữa)



Mã nguồn cho ví dụ này như sau




```
import java.util.*;

public class Context extends Stack<Integer> {
}

public interface Expression {

    public void interpret(Context context);
}

public class TerminalExpressionNumber implements Expression {

    private int number;

    public TerminalExpressionNumber(int number) {
        this.number = number;
    }

    public void interpret(Context context) {
        context.push(this.number);
```

```
    }
}

public class TerminalExpressionPlus implements Expression {

    public void interpret(Context context) {
    //Cong 2 phan tu phia tren dinh Stack
        context.push(context.pop() + context.pop());
    }
}

public class TerminalExpressionMutil implements Expression {

    public void interpret(Context context) {
//Nhan 2 phan tu phia tren dinh Stack
        context.push(context.pop() * context.pop());
    }
}

public class NonterminalExpression implements Expression {

    private ArrayList<Expression> expressions;//tham chieu den mang Exoression con

    public ArrayList<Expression> getExpressions() {
        return expressions;
    }

    public void setExpressions(ArrayList<Expression> expressions) {
        this.expressions = expressions;
    }

    public void interpret(Context context) {
        if (expressions != null) {
            int size = expressions.size();
            for (Expression e : expressions) {
                e.interpret(context);
            }
        }
    }
}

public class Client {

    public static void main(String[] agrs) {
        Context context = new Context();

        // 3 3 *
        ArrayList<Expression> treeLevel1 = new ArrayList<Expression>();
        treeLevel1.add(new TerminalExpressionNumber(3));
        treeLevel1.add(new TerminalExpressionNumber(3));
        treeLevel1.add(new TerminalExpressionMutil());

        // 5 (3 3 *) +
        ArrayList<Expression> treeLevel2 = new ArrayList<Expression>();
        treeLevel2.add(new TerminalExpressionNumber(5));
        Expression nonexpLevel1 = new NonterminalExpression();
        ((NonterminalExpression) nonexpLevel1).setExpressions(treeLevel1);
        treeLevel2.add(nonexpLevel1);
        treeLevel2.add(new TerminalExpressionPlus());

        // (5 (3 3 *) +) 6 +
        ArrayList<Expression> treeLevel3 = new ArrayList<Expression>();
        Expression nonexpLevel2 = new NonterminalExpression();
        ((NonterminalExpression) nonexpLevel2).setExpressions(treeLevel2);
        treeLevel3.add(nonexpLevel2);
        treeLevel3.add(new TerminalExpressionNumber(6));
        treeLevel3.add(new TerminalExpressionPlus());
```

```
        for (Expression e : treeLevel3) {
            e.interpret(context);
        }

        if (context != null) {
            System.out.print("Ket qua: " + context.pop());
        }
    }
}
```

3.Structural Pattern

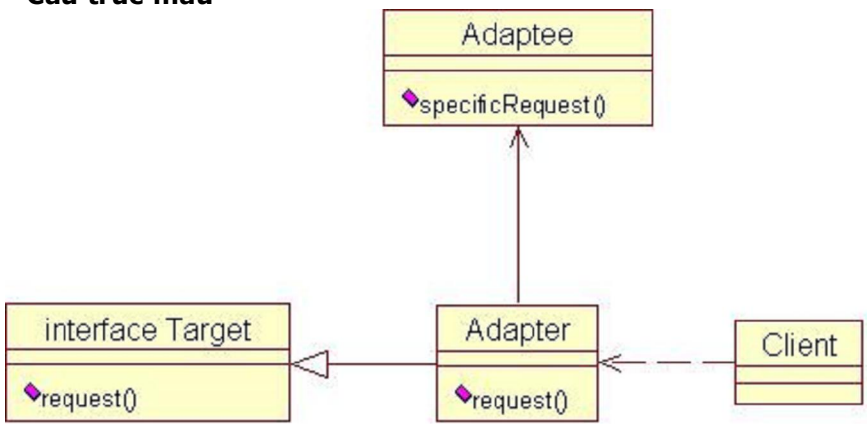
Các mẫu Structural diễn tả một cách có hiệu quả cả việc phân chia hoặc kết hợp các phần tử trong một ứng dụng. Những cách mà các mẫu Structural áp dụng vào ứng dụng rất rộng: ví dụ, mẫu Adapter có thể làm cho hai hệ thống không tương thích có thể giao tiếp với nhau, trong khi mẫu Façade cho phép bạn làm đơn giản hóa một giao tiếp để sử dụng mà không cần gỡ bỏ tất cả các tùy biến đã có trong hệ thống

3.1. Adapter Pattern

- Ý nghĩa

Tạo một giao diện trung gian để gắn kết vào hệ thống một lớp đối tượng mong muốn nào đó.

- Cấu trúc mẫu



Trong đó:

o **Tagret** là một interface định nghĩa chức năng, yêu cầu mà Client cần sử dụng

o **Adaptee** là lớp chức các chức năng mà Target cần sử dụng để tạo ra được chức năng mà Target cần cung cấp cho Client

o **Adapter** thực thi từ Target và sử dụng đối tượng lớp Adaptee, Apdater có nhiệm vụ gắn kết Adaptee vào Target để có được chức năng mà Client mong muốn

- Trường hợp ứng dụng

o Muốn sử dụng 1 lớp có sẵn nhưng giao tiếp của nó không tương thích với yêu cầu hiện tại

o Muốn tạo 1 lớp có thể sử dụng lại mà lớp này có thể làm việc được với những lớp khác không liên hệ gì với nó, và là những lớp không cần thiết tương thích trong giao diện.

- Ví dụ mẫu

- Xét ví dụ: ta có một hệ thống PhoneTarget cần thực hiện một chức năng gì đó, trong đó có một phương thức trả về số điện thoại trong một chuỗi đầu vào
- Trước đó ta đã có một lớp có một chức năng là lấy các kí tự số trong một chuỗi
- Giờ ta muốn sử dụng chức năng lấy kí tự số vào hệ thống lấy số điện thoại

```
public interface PhoneTarget{
    public String getPhoneNumber(String message);//lấy số điện thoại trong 1 chuỗi
}

public GetNumberAdaptee{
    public String getNumber(String str){/*...*/}
    //lấy ra dạng số trong 1 chuỗi
    //…
}

public Adapter implements PhoneTarget {
    public String getPhoneNumber(String message) {
        GetNumberAdaptee obj = new GetNumberAdaptee;
        String str = obj.getNumber(message);
        return "84" + str;
```

```
    }
}
```

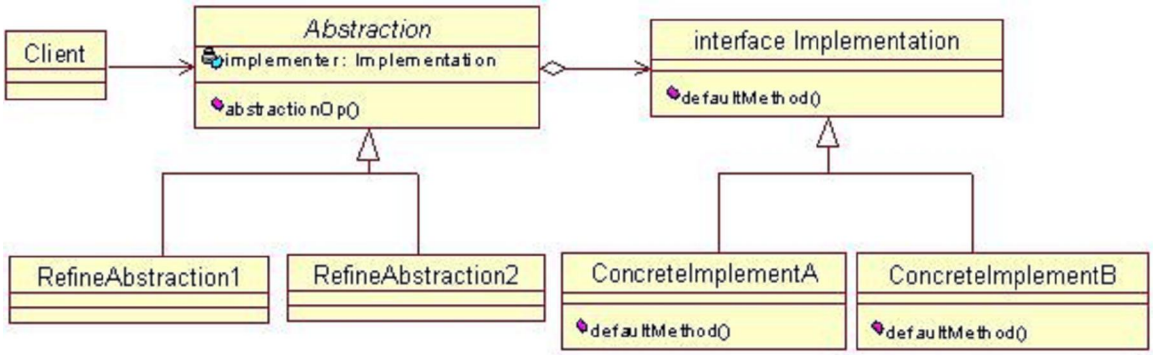
3.2. Bridge Pattern

- Ý nghĩa

Một thành phần trong OOP thường có 2 phần: phần ảo – định nghĩa các chức năng và phần thực thi – thực thi các chức năng được định nghĩa trong phần ảo. Hai phần này liên hệ với nhau qua quan hệ kế thừa. Những thay đổi trong phần ảo dẫn đến các thay đổi trong phần thực thi.

Mẫu Bridge được sử dụng để tách thành phần ảo và thành phần thực thi riêng biệt, do đó các thành phần này có thể thay đổi độc lập và linh động. Thay vì liên hệ với nhau bằng quan hệ kế thừa hai thành phần này liên hệ với nhau thông qua quan hệ "chứa trong".

- Cấu trúc mẫu



Trong đó:

o **Abstraction**: là lớp trừu tượng khai báo các chức năng và cấu trúc cơ bản, trong lớp này có 1 thuộc tính là 1 thể hiện của giao tiếp Implementation, thể hiện này bằng các phương thức của mình sẽ thực hiện các chức năng abstractionOp() của lớp Abstraction

o **Implementation**: là giao tiếp thực thi của lớp các chức năng nào đó của Abstraction

o **RefineAbstraction**: là định nghĩa các chức năng mới hoặc các chức năng đã có trong Absrtaction.

o **ConcreteImplement**: là các lớp định nghĩa tường minh các thực thi trong lớp giao tiếp Implementation

- Trường hợp ứng dụng

o Khi bạn muốn tạo ra sự mềm dẻo giữa 2 thành phần ảo và thực thi của một thành phần, và tránh đi mối quan hệ tĩnh giữa chúng

o Khi bạn muốn những thay đổi của phần thực thi sẽ không ảnh hưởng đến client

o Bạn định nghĩa nhiều thành phần ảo và thực thi.

o Phân lớp con một cách thích hợp, nhưng bạn muốn quản lý 2 thành phần của hệ thống một các riêng biệt

- Ví dụ mẫu

```
class MyAbstraction {

    private MyImplementation myImp;

    public void method01() {
        myImp.doMethod1();
    }

    public String method2() {
        myImp.doMethod2();
    }
}

interface

    class MyImplementation {

        public void doMethod1();

        public String doMethod2();
    }

    class RefineAbstraction1 extends MyAbstraction {
//các định nghĩa riêng, tường minh
    }

    class ConcreteImpleA
```

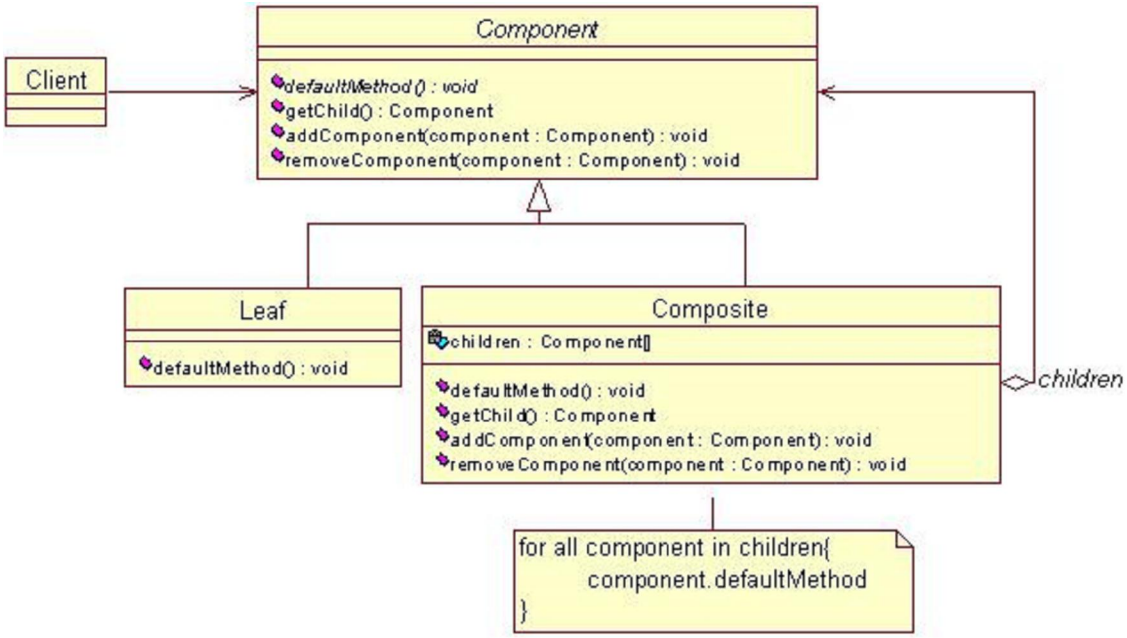
```
     extend  MyImplement {
//các định nghĩa riêng, tường minh
    }

    class RefineAbstraction2


extends MyAbstraction{
//các định nghĩa riêng, tường minh
}

class ConcreteImpleB extend MyImplement{
//các định nghĩa riêng, tường minh
}
```

3.3. Composite Pattern

- Ý nghĩa

Mẫu này nhằm gom các đối tượng vào trong một cấu trúc cây để thể hiện được cấu trúc tổng quát của nó. Trong khi đó cho phép mỗi phần tử của cấu trúc cây có thể thực hiện một chức năng theo một giao tiếp chung

- Mô hình mẫu



Trong đó:

o **Component**: là một giao tiếp định nghĩa các phương thức cho tất cả các phần của cấu trúc cây. Component có thể được thực thi như một lớp trừu tượng khi bạn cần cung cấp các hành vi cho tất cả các kiểu con. Bình thường, các Component không có các thể hiện, các lớp con hoặc các lớp thực thi của nó, gọi là các nốt, có thể có thể hiện và được sử dụng để tạo nên cấu trúc cây

o **Composite**: là lớp được định nghĩa bởi các thành phần mà nó chứa. Composite chứa một nhóm động các Component, vì vậy nó có các phương thức để thêm vào hoặc loại bổ các thể hiện của Component trong tập các Component của nó. Những phương thức được định nghĩa trong Component được thực thi để thực hiện các hành vi đặc tả cho lớp Composite và để gọi lại phương thức đó trong các nốt của nó. Lớp Composite được gọi là lớp nhánh hay lớp chứa

o **Leaf**: là lớp thực thi từ giao tiếp Component. Sự khác nhau giữa lớp Leaf và Composite là lớp Leaf không chứa các tham chiếu đến các Component khác, lớp Leaf đại diện cho mức thấp nhất của cấu trúc cây

- Trường hợp ứng dụng

o Khi có một mô hình thành phần với cấu trúc nhánh – lá, toàn bộ – bộ phận, …

o Khi cấu trúc có thể có vài mức phức tạp và động

o Bạn muốn thăm cấu trúc thành phần theo một cách qui chuẩn, sử dụng các thao tác chung thông qua mối quan hệ kế thừa

- Ví dụ mẫu

Trở lại ví dụ về dự án, một Project(Composite) có nhiều tác vụ Task(Leaf), ta cần tính tổng thời gian của dự án thông qua thời gian của tất cả các tác vụ

```
public interface TaskItem {

    public double getTime();
}

public class Project implements TaskItem {

    private String name;
    private ArrayList subtask = new ArrayList();

    public Project() {
    }

    public Project(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    public ArrayList getSubtasks() {
        return subtask;
    }

    public double getTime() {
        double totalTime = 0;
        Iterator items = subtask.iterator();
        while (items.hasNext()) {
            TaskItem item = (TaskItem) items.next();
            totalTime += item.getTime();
        }
        return totalTime;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void addTaskItem(TaskItem element) {
        if (!subtask.contains(element)) {
            subtask.add(element);
        }
    }

    public void removeTaskItem(TaskItem element) {
        subtask.remove(element);
    }
}

public class Task implements TaskItem {

    private String name;
    private double time;

    public Task() {
    }

    public Task(String newName, double newTimeRequired) {
        name = newName;
        time = newTimeRequired;
    }

    public String getName() {
        return name;
    }

    public double getTime() {
```

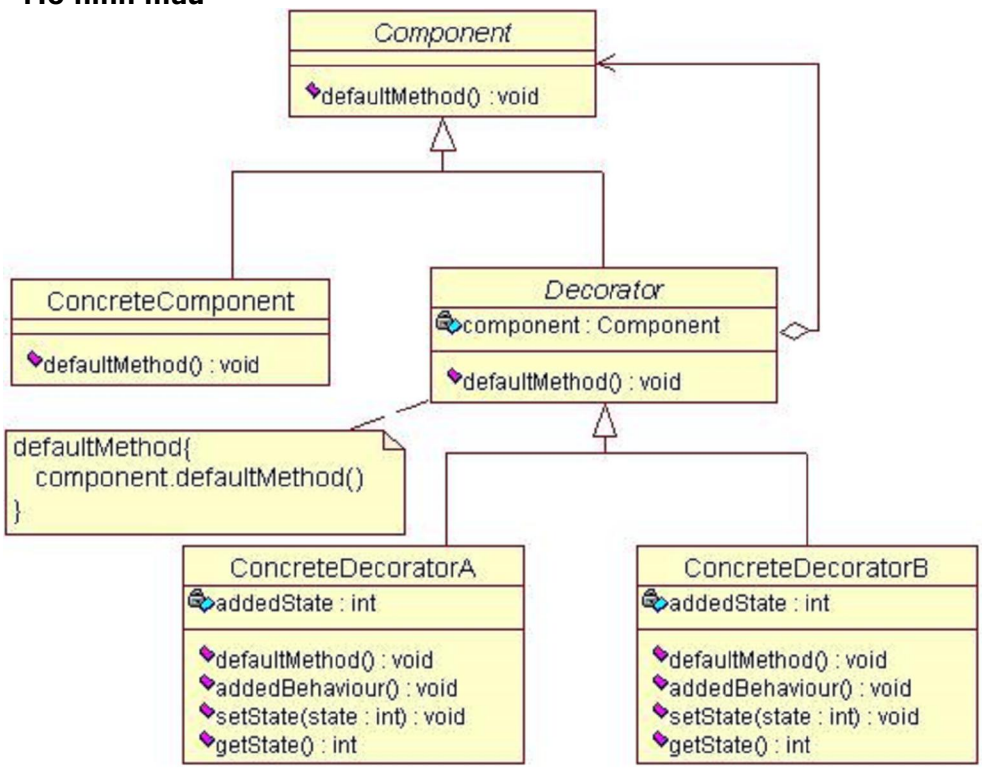
```
        return time;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void setTime(double newTimeRequired) {
        time = newTimeRequired;
    }
}
```

3.4. Decorator Pattern

- **Ý nghĩa**

Bổ sung trách nhiệm cho đối tượng tại thời điểm thực thi. Đây 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.

- **Mô hình mẫu**



Trong đó:

o **Component**: là một interface chứa các phương thức ảo (ở đây là defaultMethod)

o **ConcreteComponent**: là một lớp kế thừa từ Component, cài đặt các phương thức cụ thể (defaultMethod được cài đặt tường minh)

o **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.

o **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

- **Trường hợp ứng dụng**

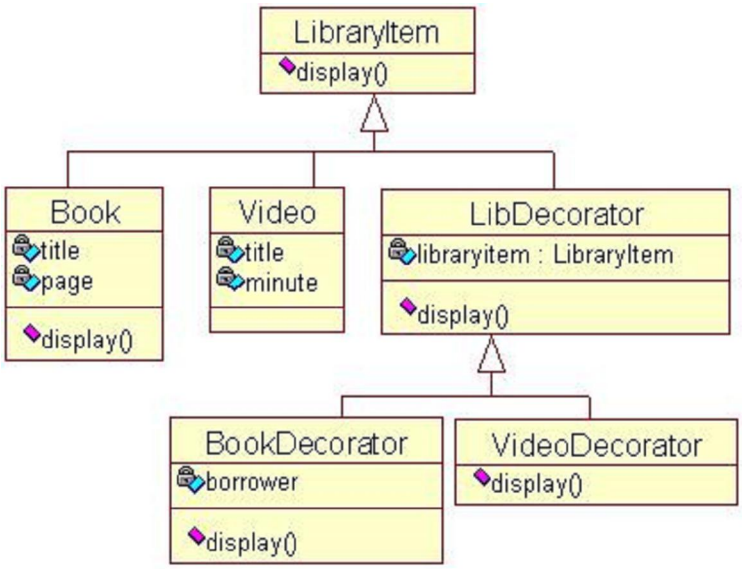
o Khi bạn muốn thay đổi động mà không ảnh hưởng đến người dùng, không phụ thuộc vào giới hạn các lớp con

o Khi bạn muốn thành phần có thể thêm vào hoặc rút bỏ đi khi hệ thống đang chạy

o Có một số đặc tính phụ thuộc mà bạn muốn ứng dụng một cách động và bạn muốn kết hợp chúng vào trong một thành phần

- **Ví dụ**

Gia sử 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 LibraryItem 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 không phải lúc nào cũng muốn hiển thị), hoặc muốn xem đoạn video(không thường xuyên).



Giao tiếp LibraryItem định nghĩa phương thức display() cho tất cả các tài liệu của thư viện

```
public interface LibraryItem {

    public void display(); // đây là defaultMethod
}

//Các lớp tài liệu
public class Book implements LibraryItem {

    private String title;
    private int page;

    public Book(String s, int p) {
        title = s;
        page = p;
    }

    public void display() {
        System.out.println("Title: " + title);
        System.out.println("Page number: " + page);
    }
}

public class Video implements LibraryItem {

    private String title;
    private int minutes;

    public Video(String s, int m) {
        title = s;
        minutes = m;
    }

    public void display() {
        System.out.println("Title: " + title);
        System.out.println("Time: " + minutes);
    }
}

//Lớp ảo Decorator thư viện
public abstract class LibDecorator implements LibraryItem {

    private LibraryItem libraryitem;

    public LibDecorator(LibraryItem li) {
        libraryitem = li;
```

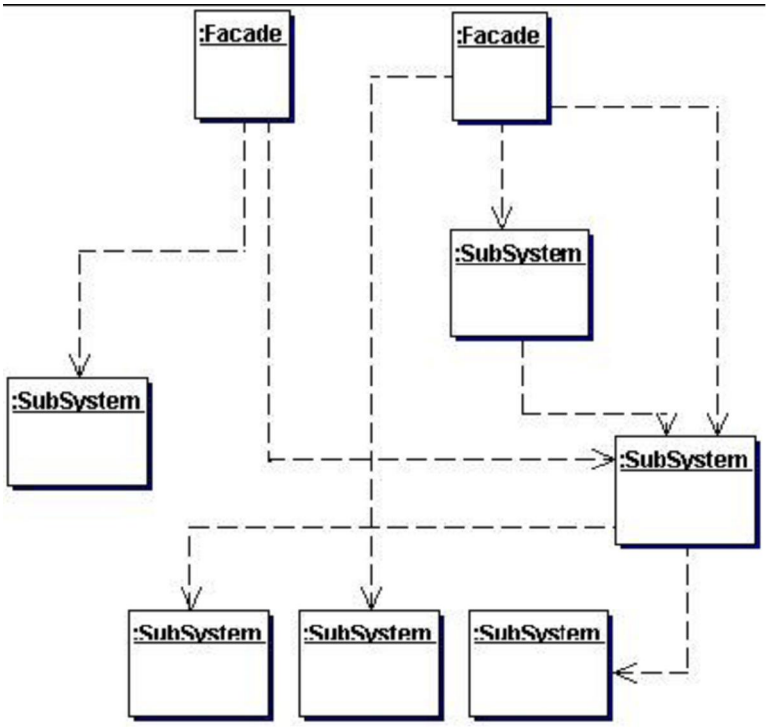
```
    }

    public void display() {
        libraryitem.display();
    }
}

//Các lớp Decorator cho mỗi tài liệu thư viện cần bổ sung trách nhiệm ở thời điểm run-time
public class BookDecorator extends LibDecorator {

    private String borrower;

    public BookDecorator(LibraryItem li, String b) {
        super(li);
        borrower = b;
    }

    public void display() {
        super.display();
        System.out.println("Borrower: " + borrower);
    }
}

public class VideoDecorator extends LibDecorator {

    public VideoDecorator(LibraryItem li) {
        super(li);
    }

    public void display() {
        super.display();
        play();
//phương thức play video
    }
}
```

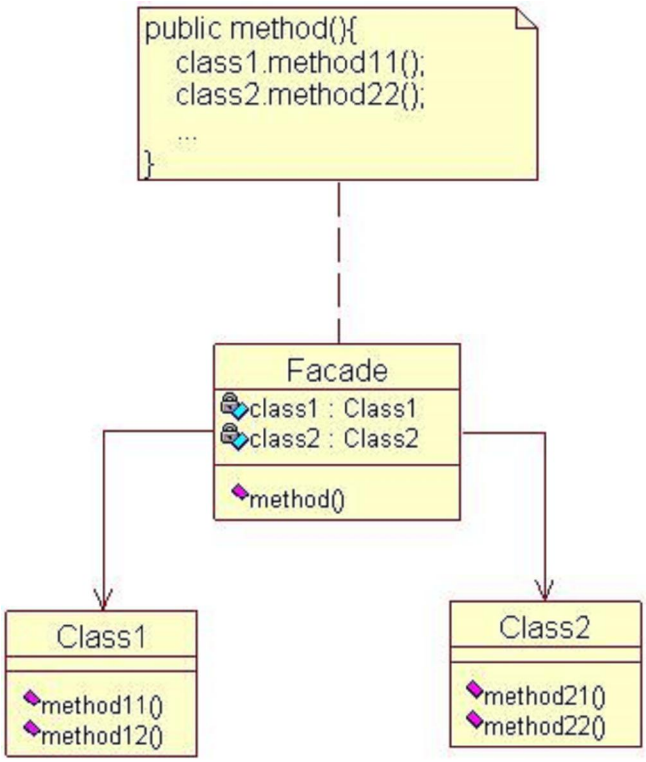
3.5. Facade Pattern

- Ý nghĩa

Cung cấp một giao tiếp hợp nhất của một tập các giao tiếp trong hệ thống con. Façade định nghĩa một giao tiếp mức cao hơn để làm cho hệ thống con dễ sử dụng

- Mô hình mẫu





Trong đó

o **Class1** và **Class2** là các lớp đã có trong hệ thống

o **Façade** là lớp sử dụng các phương thức của Class1 và Class2 để tạo ra một giao diện mới, thường được sử dụng, đỡ phức tạp hơn khi sử dụng riêng Class1 và Class2

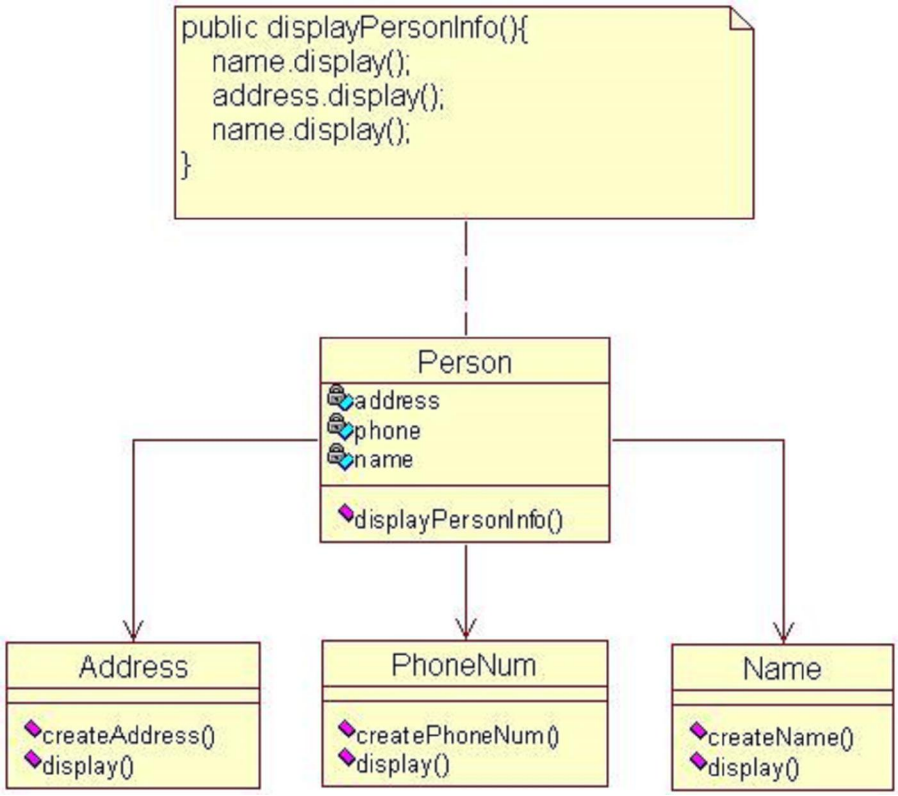
- Trường hợp ứng dụng

o Làm cho một hệ thống phức tạp dễ sử dụng hơn bằng cách cung cấp một giao tiếp đơn gian mà không cần loại bỏ các lựa chọn phức tạp

o Giảm bớt sự ràng buộc giữa client và các hệ thống con

- Ví dụ mẫu

Giả sử hệ thống cũ đã có các lớp: Địa chỉ, Số điện thoại, Tên tuổi về thông tin của một người, giờ ta muốn quản lý các thông tin trên của một người, ta tận dụng lại hệ thống cũ, xây một lớp Người sử dụng lại các lớp ở trên.

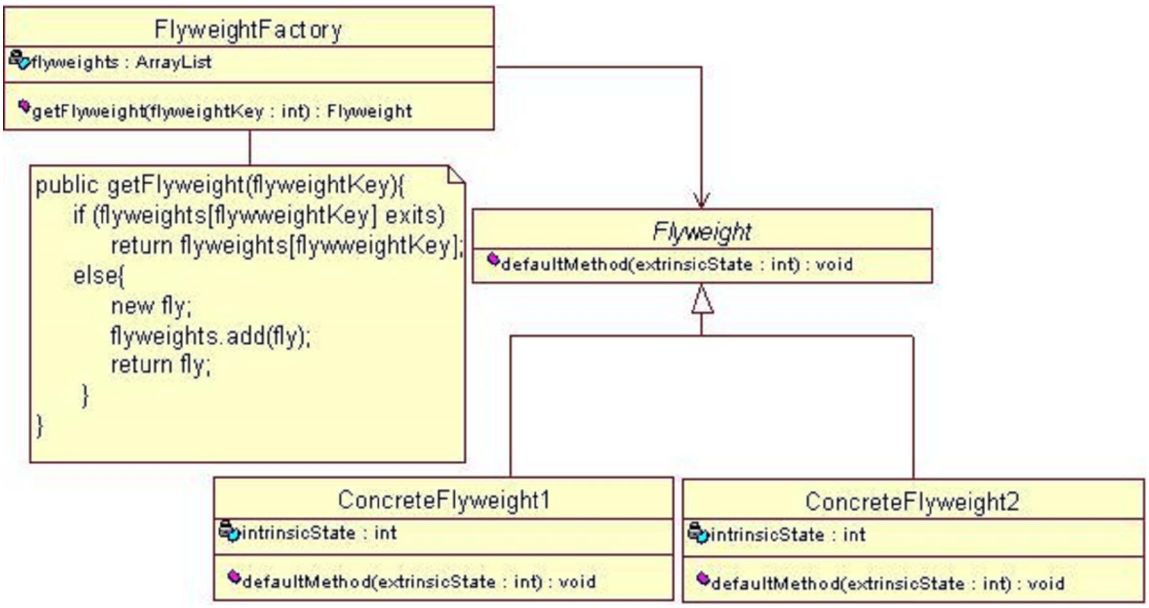


3.6. Flyweight Pattern

- **Ý nghĩa**

Làm phương tiện dùng chung để quản lý một cách hiệu quả một số lượng lớn các đối tượng nhỏ có các đặc điểm chung, mà các đối tượng nhỏ này lại được sử dụng tuỳ thuộc vào hoàn cảnh, điều kiện ngoài.

- **Mô hình mẫu**



Trong đó:

o **FlyweightFactory**: tạo ra và quản lý các đối tượng Flyweight

o **Flyweight** là một giao tiếp định nghĩa các phương thức chuẩn

o **ConcreteFlyweightX**: là các lớp thực thi của Flyweight, các thể hiện của các lớp này sẽ được sử dụng tuỳ thuộc vào điều kiện ngoài.

- **Trường hợp sử dụng**

o Ứng dụng sử dụng nhiều đối tượng giống hoặc gần giống nhau

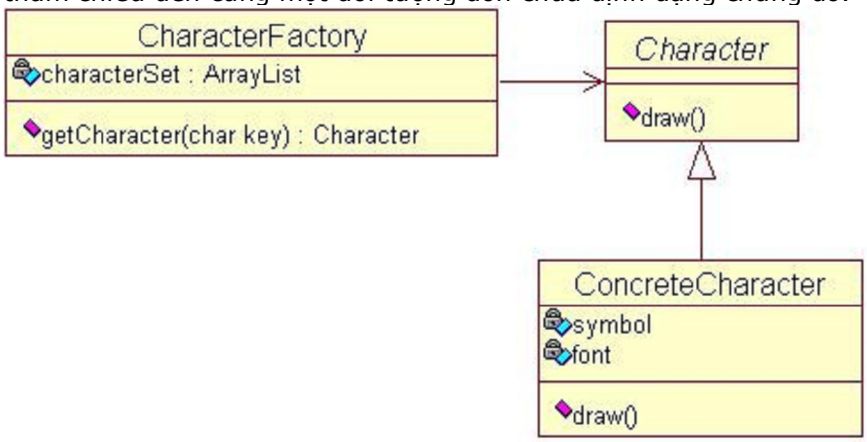
o Với các đối tượng gần giống nhau, những phần không giống nhau có thể tách rời với các phần giống nhau để cho phép các phần giống nhau có thể chia sẻ

o Nhóm các đối tượng gần giống nhau có thể được thay thế bởi một đối tượng chia sẻ mà các phần không giống nhau đã được loại bỏ

o Nếu ứng dụng cần phân biệt các đối tương gần giống nhau trong trạng thái gốc của chúng

- **Ví dụ mẫu**

Một ví dụ cổ điển của mẫu flyweight là các kí tự được lưu trong một bộ xử lí văn bản (word processor). Mỗi kí tự đại diện cho một đối tượng mà có dữ liệu là loại font (font face), kích thước font, và các dữ liệu định dạng khác. Bạn có thể tưởng tượng là, với một tài liệu (document) lớn với cấu trúc dữ liệu như thế này thì sẽ bộ xử lí văn bản sẽ khó mà có thể xử lí được. Hơn nữa, vì hầu hết dữ liệu dạng này là lặp lại, phải có một cách để giảm việc lưu giữ này – đó chính là mẫu Flyweight. Mỗi đối tượng kí tự sẽ chứa một tham chiếu đến một đối tượng định dạng riêng rẽ mà chính đối tượng này sẽ chứa các thuộc tính cần thiết. Điều này sẽ giảm một lượng lớn sự lưu giữ bằng cách kết hợp mọi kí tự có định dạng giống nhau trở thành các đối tượng đơn chỉ chứa tham chiếu đến cùng một đối tượng đơn chứa định dạng chung đó.



Giao tiếp Character định nghĩa phương thức vẽ một kí tự

```
public interface Character {

    public void draw();
}

//Lớp kí tự thực thi từ giao tiếp Character
//Sở dĩ ta định nghĩa Character và ConcreteCharacter riêng là vì trong cấu trúc sẽ có một phần nữa: phần không giống nhau giữa các đối tượng Character (mà ta không bàn tới)
public class ConcreteCharacter implements Character {

    private String symbol;
    private String font;

    public ConcreteCharacter(String s, String f) {
        this.symbol = s;
        this.font = f;
    }

    public void draw() {
        System.out.println("Symbol " + this.symbol + " with font " + this.font);
    }
}

//Lớp khởi tạo các kí tự theo symbol và font, mỗi lần khởi tạo nó sẽ lưu các đối tượng này vào vùng nhớ riêng của nó, nếu đối tượng ký tự symbol + font đã có thì nó sẽ lấy ra chứ không khởi tạo lại

public class CharacterFactory {
private Hashtable pool = new Hashtable<String, Character>();

public int getNum() {
return pool.size();
}

public Character get(String symbol, String fontFace) {
Character c;
String key = symbol + fontFace;
if ((c = (Character)pool.get(key)) != null) {
            return c;
```

```
        } else {
            c = new ConcreteCharacter(symbol, fontFace);
            pool.put(key, c);
            return c;
        }
    }
}

//Lớp Test

import java.util.*

public class Test {
public static void main(String[] agrs) {
        CharacterFactory characterfactory = new CharacterFactory();
        ArrayList<Character> text = new ArrayList<Character>();
        text.add(0, characterfactory.get("a", "arial"));
        text.add(1, characterfactory.get("b", "time"));
        text.add(2, characterfactory.get("a", "arial"));
        text.add(0, characterfactory.get("c", "arial"));
        for (int i = 0; i < text.size(); i++) {
            Character c = (Character) text.get(i);
            c.draw();
        }
    }
}
```

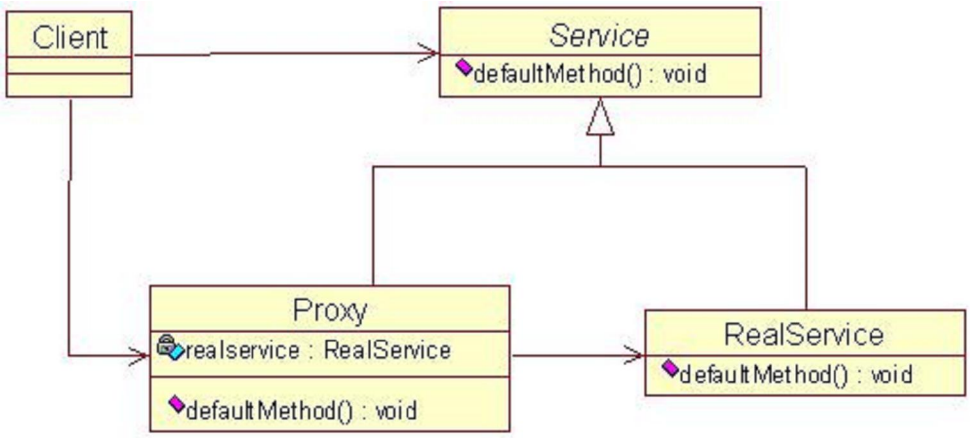
Như vậy 'a' + 'arial' gọi 2 lần như chỉ khởi tạo có 1 lần mà thôi

3.7. Proxy Pattern

- **Ý nghĩa**

Đại diện một đối tượng phức tạp bằng một đối tượng đơn giản, vì các mục đích truy xuất, tốc độ và bảo mật

- **Mô hình mẫu**



Trong đó:

o **Service**: là giao tiếp định nghĩa các phương thức chuẩn cho một dịch vụ nào đó

o **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

o **Proxy**: kế thừa Service và sử dụng đối tượng của RealService

- **Trường hợp ứng dụng**

o Sử dụng mẫu Proxy khi bạn cần một tham chiếu phức tạp đến một đối tượng thay vì chỉ một cách bình thường

o Remote proxy – sử dụng khi bạn cần một tham chiếu định vị cho một đối tượng trong không gian địa chỉ(JVM)

o Virtual proxy – lưu giữ các thông tin thêm vào về một dịch vụ thực vì vậy chúng có thể hoãn lại sự truy xuất vào dịch vụ này

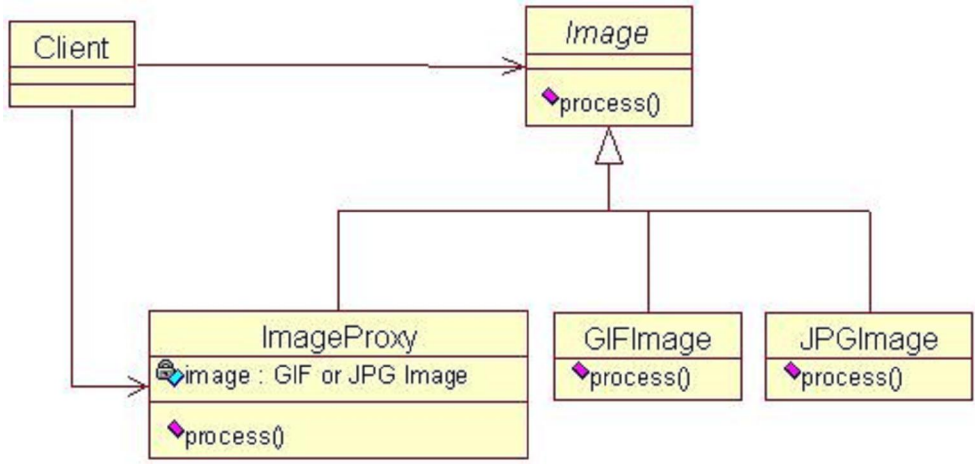
o Protection proxy – xác thực quyền truy xuất vào một đối tượng thực

- **Ví dụ mẫu**

Ví dụ lớp Image là một interface định nghĩa các phương thức xử lý ảnh, nó có các lớp con là GIFImage và JPGImage.

Theo hướng đối tượng thì thiết kế như thế có vẻ hợp lý, Client chỉ cần sử dụng lớp Image là đủ, còn tuỳ thuộc

vào loại ảnh sẽ có các phương thức khác nhau
Nhưng trong trường hợp, trên ứng dụng web chẳng hạn, một lúc ta đọc lên hàng trăm ảnh các loại và ta còn muốn xử lý tuỳ vào một điều kiện nào đó (ví dụ chỉ xử lý khi là ảnh JPG hoặc GIF). Nếu ta đặt điều kiện IF Image (sau đó sẽ tùy điều kiện này rồi xử lý riêng) thì không hợp lý, nếu đặt trong Client, nếu mỗi lần cần thay đổi IF ta lại sửa Client => không hợp lý khi Client là một ứng dụng lớn.
Ta sử dụng Proxy, lớp ImageProxy chỉ là lớp đại diện cho Image, kế thừa từ Image và sử dụng các lớp GIFImage hay JPGImage. Khi cần thay đổi ta chỉ cần thay đổi trên Proxy mà không cần tác động đến Client và Image.




```
public interface Image {

    public void process();
}

public class JPGImage implements Image {

    public void process() {
        System.out.print("JPG Image");
    }
}

public class GIFImage implements Image {

    public void process() {
        System.out.print("GIF Image");
    }
}

public class ImageProxy implements Image {

    private Image image;

    public void process() {
        if (image == null) {
            image = new JPGImage();//tạo đối tượng ảnh JPG, chỉ mang tính minh họa
        }
        image.process();

}
```

(Theo Opera NguyenD)}
Ở đây ta sẽ xử lý khi Image là ảnh JPG trong trường hợp muốn thay đổi ta sẽ thay đổi ở ImageProxy và client sẽ không bị ảnh hưởng
*** Đây là ví dụ cho trường hợp Virtual Proxy**
Tài liệu tham khảo:
http://www.dofactory.com/Patterns/Patterns.aspx
http://www.dofactory.com/Framework/Framework.aspx