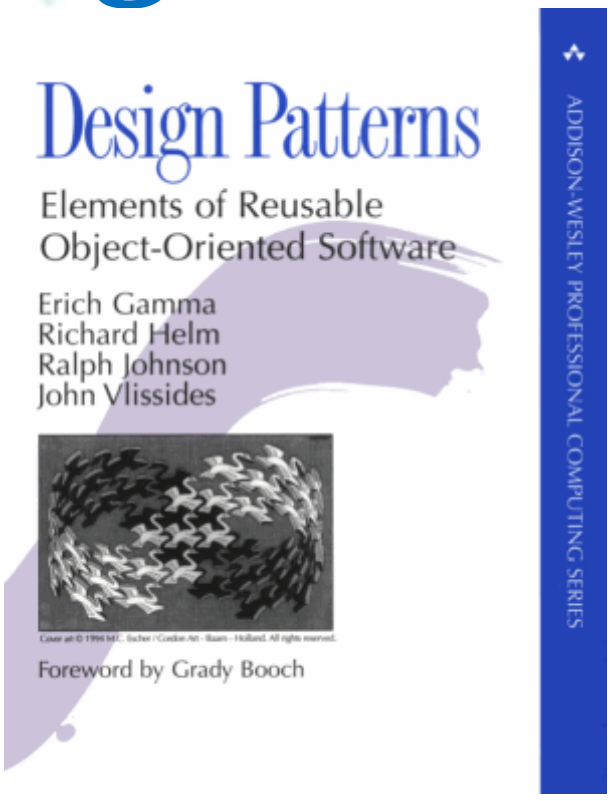
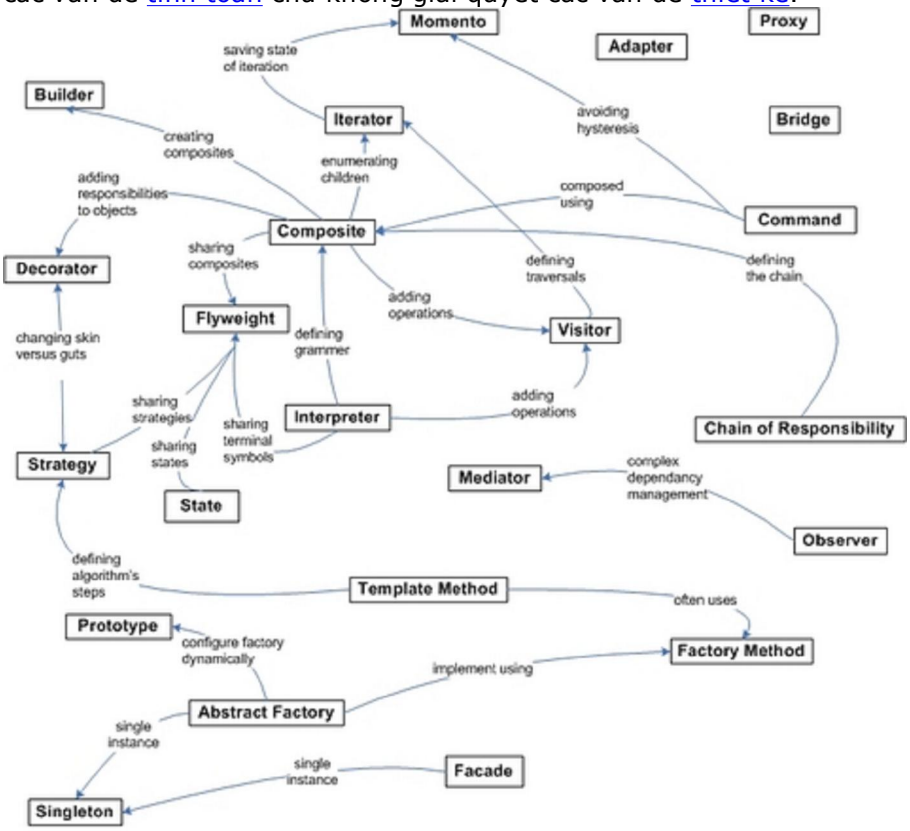
Hệ thống các mẫu Design Pattern

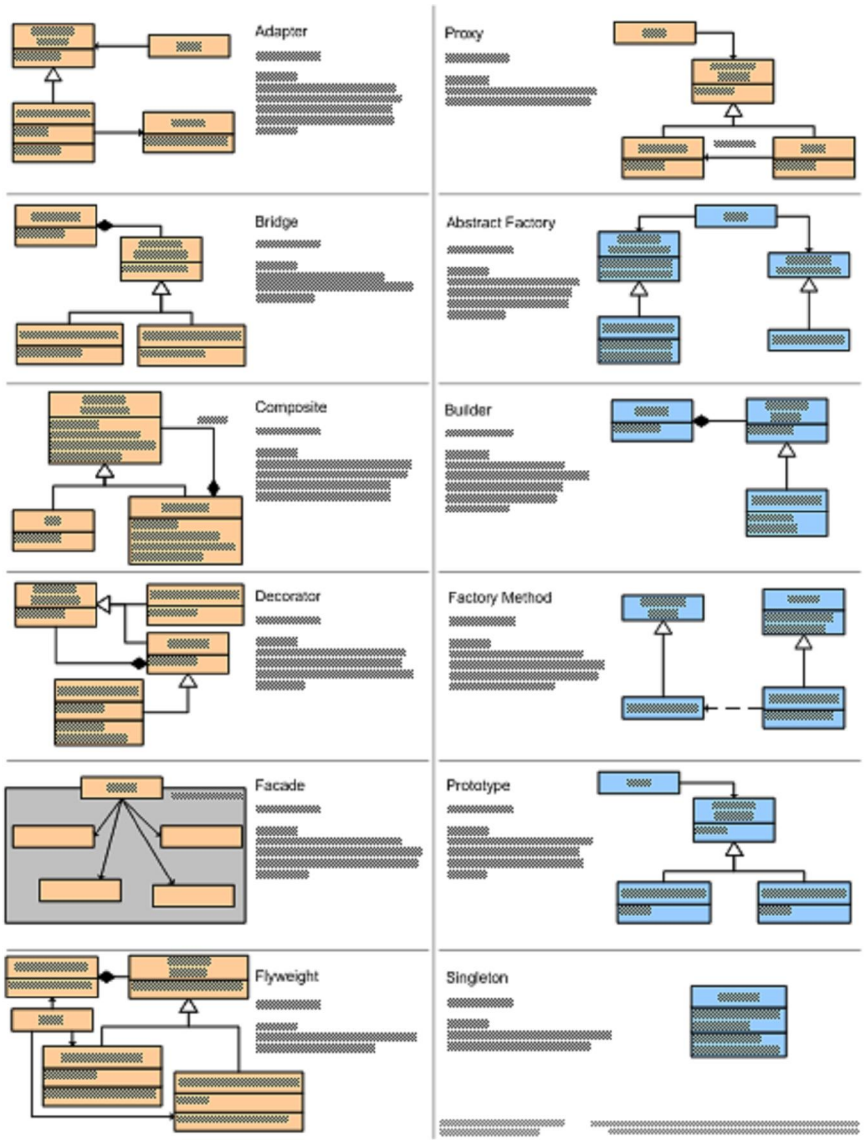


GIỚI THIỆU DESIGN PATTERN

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)

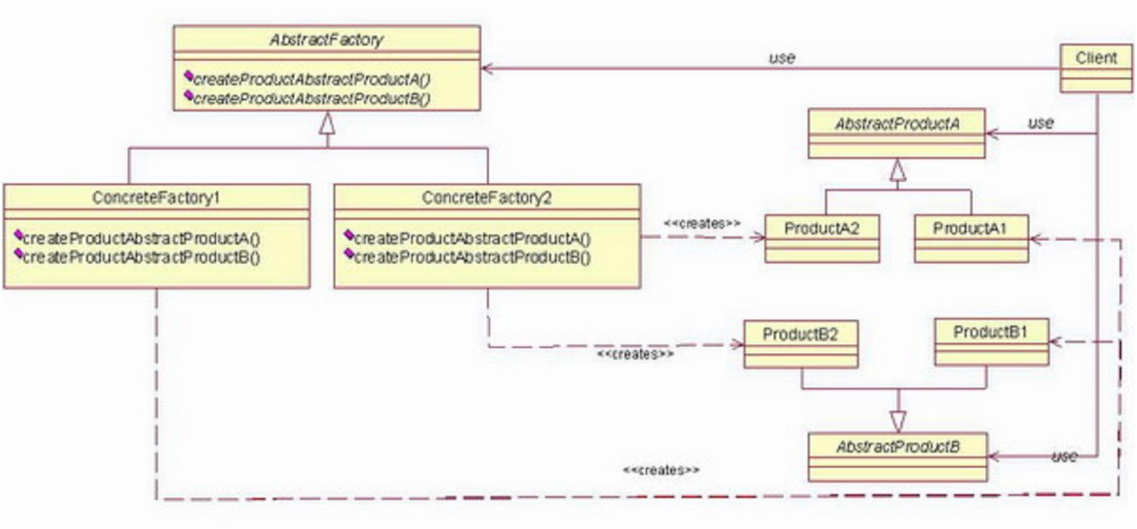
Những mẫu này hỗ trợ cho một trong những nhiệm vụ của lập trình hướng đối tượng – khởi tạo đối tượng trong hệ thống. Hầu hết các hệ thống hướng đối tượng phức tạp yêu cầu nhiều đối tượng được thể hiện theo thời gian, và các mẫu này hỗ trợ cho việc tạo các tiến trình bằng việc cung cấp các khả năng:
- 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**



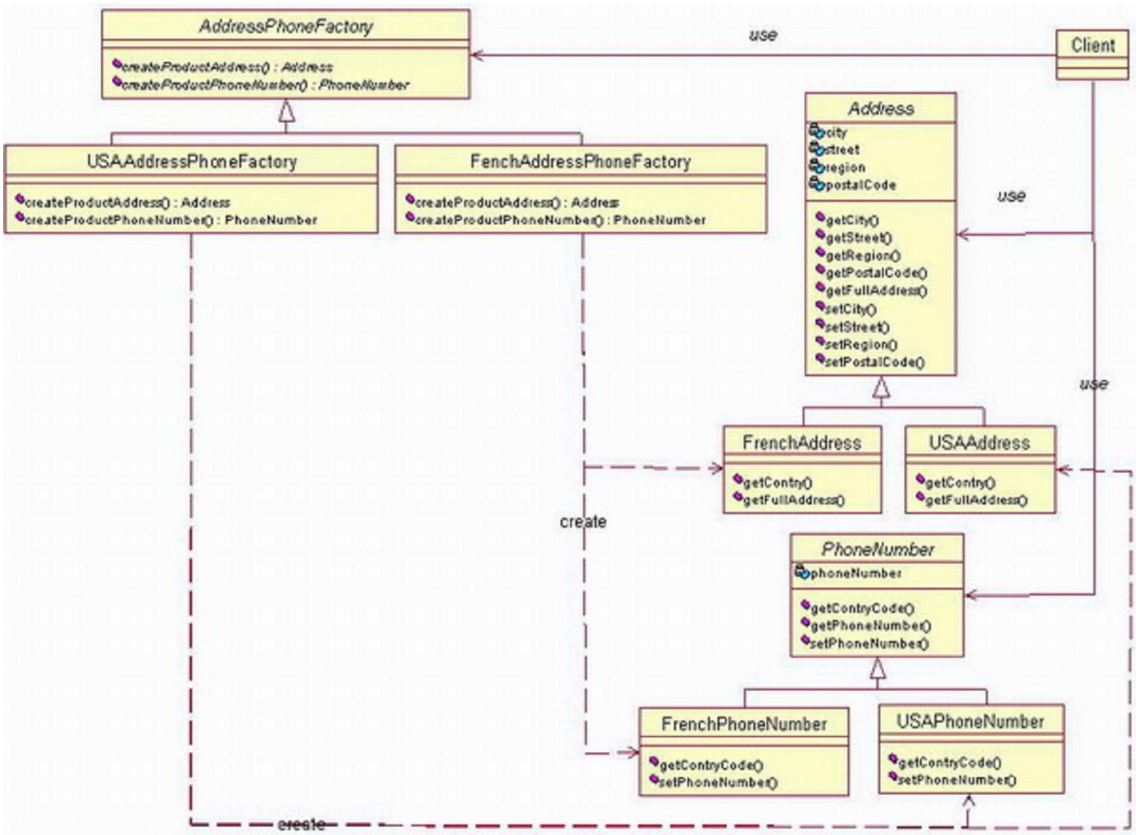
Trong đó:
o **AbstractFactory**: là lớp trừu tượng, tạo ra các đối tượng thuộc 2 lớp trừu tượng là: AbstractProductA và AbstractProductB
o **ConcreteFactoryX**: là lớp kế thừa từ AbstractFatory, lớp này sẽ tạo ra một đối tượng cụ thể
o **AbstractProduct**: là các lớp trừu tượng, các đối tượng cụ thể 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;
    }
}
```

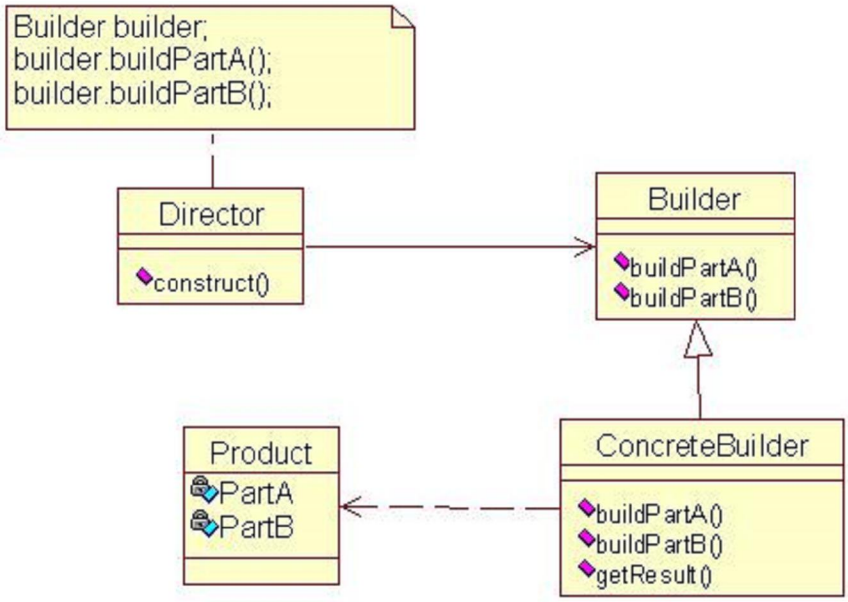
Tương tự cho lớp PhoneNumber và USAPhoneNumber

1.2.Builder Pattern

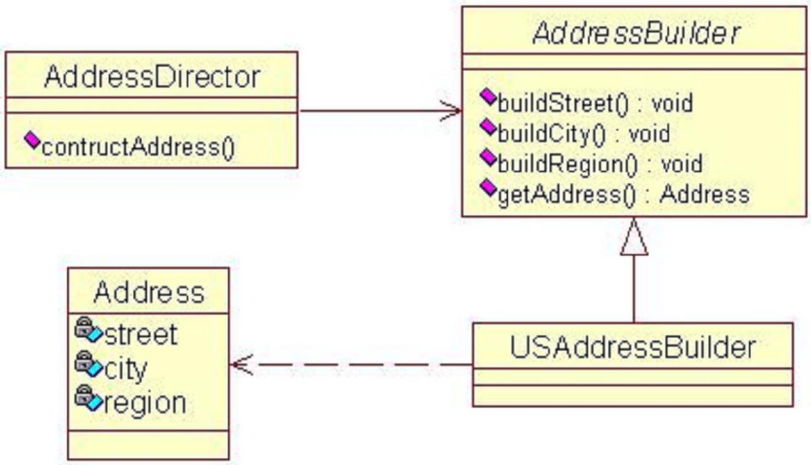
- **Ý nghĩa**

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);
    }
```