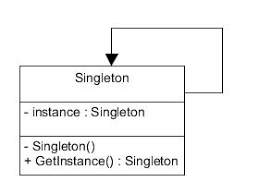
## Singleton Pattern là gì?

**Singleton** là 1 trong 5 design pattern của nhóm **Creational Design Pattern**.

**Singleton** đảm bảo chỉ duy nhất **một thể hiện (instance)** được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

[](https://gpcoder.com/wp-content/uploads/2018/09/singleton-pattern.png)

Sử dụng Singleton khi chúng ta muốn:

* Đảm bảo rằng chỉ có một instance của lớp.
* Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
* Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.

## Implement Singleton Pattern như thế nào?

Có rất nhiều cách để implement Singleton Pattern. Nhưng dù cho việc implement bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây cơ bản dưới đây:

* **private constructor** để hạn chế truy cập từ class bên ngoài.
* Đặt **private static final variable** đảm bảo biến chỉ được khởi tạo trong class.
* Có một method **public static** để **return instance** được khởi tạo ở trên.

## Những cách nào để implement Singleton Pattern

Dựa trên những nguyên tắc thiết kế Singleton ở trên, chúng ta có các cách implement singleton như sau:

### Eager initialization

**Singleton** Class được khởi tạo ngay khi được gọi đến. Đây là cách dễ nhất nhưng nó có một nhược điểm mặc dù instance đã được khởi tạo mà có thể sẽ không dùng tới.

Eager initialization là cách tiếp cận tốt, dễ cài đặt, tuy nhiên, nó dễ dàng bị phá vỡ bởi Reflection.

### Static block initialization

Cách làm tương tự như **Eager initialization** chỉ khác phần **static block** cung cấp thêm lựa chọn cho việc handle exception hay các xử lý khác.

Lazy Initialization

Là một cách làm mang tính mở rộng hơn so với 2 cách làm trên và hoạt động tốt trong môi trường đơn luồng (single-thread).

Cách này đã khắc phục được nhược điểm của cách **Eager initialization**, chỉ khi nào **getInstance()** được gọi thì instance mới được khởi tạo. Tuy nhiên, cách này chỉ sử dụng tốt trong trường hợp đơn luồng (single-thread), trường hợp nếu có nhiều luồng (multi-thread) cùng chạy và cùng gọi hàm getInstance() tại cùng một thời điểm thì có thể có nhiều hơn 1 thể hiện của instance. Để khắc phục nhược điểm này chúng ta sử dụng **Thread Safe Singleton**.

Một nhược điểm nữa của **Lazy Initialization** cần quan tâm là: đối với thao tác create instance quá chậm thì người dùng có phải chờ lâu cho lần sử dụng đầu tiên.

### Thread Safe Singleton

Cách đơn giản nhất là chúng ta gọi phương thức **synchronized** của hàm **getInstance()** và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.

Biến **volatile** trong Java có tác dụng thông báo sự thay đổi giá trị của biến tới các thread khác nhau nếu biến này đang được sử dụng trong nhiều thread.

Cách này có nhược điểm là một phương thức **synchronized** sẽ chạy rất chậm và tốn hiệu năng, bất kỳ Thread nào gọi đến đều phải chờ nếu có một Thread khác đang sử dụng. Có những tác vụ xử lý trước và sau khi tạo thể hiện không cần thiết phải block. Vì vậy chúng ta cần cải tiến nó đi 1 chút với **Double Check Locking Singleton**.

### Double Check Locking Singleton

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hổ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo **volatile** cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

Bill Pugh Singleton Implementation

Với cách làm này bạn sẽ tạo ra **static nested class** với vai trò 1 Helper khi muốn tách biệt chức năng cho 1 class function rõ ràng hơn. Đây là cách thường hay được sử dụng và có hiệu suất tốt (theo các chuyên gia đánh giá.

Khi Singleton được tải vào bộ nhớ thì SingletonHelper chưa được tải vào. Nó chỉ được tải khi và chỉ khi phương thức getInstance() được gọi. Với cách này tránh được lỗi cơ chế khởi tạo instance của Singleton trong Multi-Thread, performance cao do tách biệt được quá trình xử lý. Do đó, cách làm này được đánh giá là cách triển khai Singleton nhanh và hiệu quả nhất.

### Phá vỡ cấu trúc Singleton Pattern bằng Reflection

**Reflection** có thể được dùng để phá vỡ Pattern của **Eager Initialization** ở trên.

Tương tự **Eager Initialization**, implement theo **Bill Pugh Singleton** cũng bị break bởi Reflection.

### Enum Singleton

Khi dùng **enum** thì các params chỉ được khởi tạo 1 lần duy nhất, đây cũng là cách giúp bạn tạo ra Singleton instance.

Lưu ý:

* Enum có thể sử dụng như một Singleton, nhưng nó có nhược điểm là không thể extends từ một lớp được, nên khi sử dụng cần xem xét vấn đề này.
* Hàm **constructor** của **enum** là **lazy**, nghĩa là khi được sử dụng mới chạy hàm khởi tạo và nó chỉ chạy duy nhất một lần. Nếu muốn sử dụng như một eager singleton thì cần gọi thực thi trong một **static block** khi start chương trình.

So sánh giữa 2 cách sử dụng **enum initialization** và **static block initialization** **method**, enum có một điểm rất mạnh khi giải quyết về vấn đề **Serialization/ Deserialization**.

### Serialization and Singleton

Đôi khi trong các hệ thống phân tán (distributed system), chúng ta cần implement interface **Serializable** trong lớp Singleton để chúng ta có thể lưu trữ trạng thái của nó trong file hệ thống và truy xuất lại nó sau.

Như trong ví dụ trên, **Deserialize** đối tượng của **SerializedSingleton** khác với đối tượng gốc. Tuy nhiên vấn đề này không xảy ra khi sử dụng **enum**.

Thực tế thì vẫn có cách khắc phục khi sử dụng class SerializedSingleton là implement một phương thức **readResolve()**. Nhưng khi chúng ta thật sự gặp vấn đề và cần sử dụng Serialize/ Deserialize, thì nên sử dụng **enum** sẽ đơn giản hơn.

## Sử dụng Singleton Pattern khi nào?

Dưới đây là một số trường hợp sử dụng của Singleton Pattern thường gặp:

* Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …
* Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,…
* Đã được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop.

## Tổng kết

Có rất nhiều cách implement cho Singleton, mình thường sử dụng **BillPughSingleton** vì có hiệu suất cao, sử dụng **LazyInitializedSingleton** cho những ứng dụng chỉ làm việc với ứng dụng **single-thread** và sử dụng **DoubleCheckLockingSingleton** khi làm việc với ứng dụng **multi-thread**. Tùy theo trường hợp cụ thể, bạn hãy chọn cho mình cách implement phù hợp.

## Factory Method Pattern là gì?

**Factory Method Design Pattern**hay gọi ngắn gọn là **Factory Pattern** là một trong những Pattern thuộc nhóm **Creational Design Pattern**. Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

**Factory Pattern** đúng nghĩa là một **nhà máy**, và nhà máy này sẽ “**sản xuất**” các đối tượng theo yêu cầu của chúng ta.

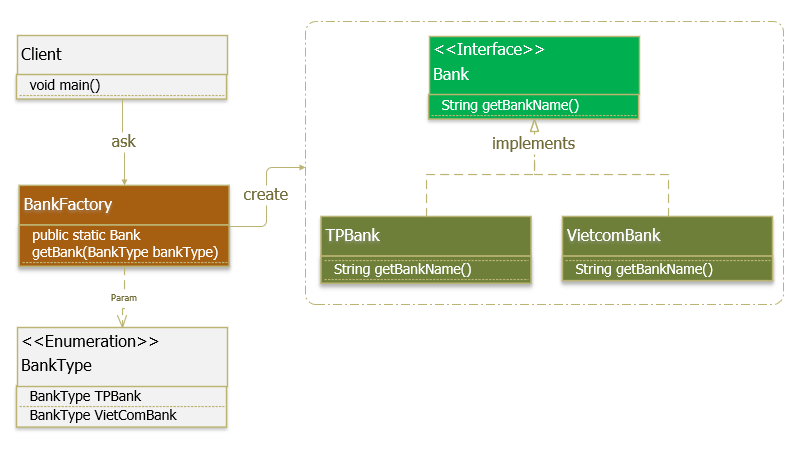
Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

## Cài đặt Factory Pattern như thế nào?

Một Factory Pattern bao gồm các thành phần cơ bản sau:

* **Super Class**: môt supper class trong Factory Pattern có thể là một **interface**, **abstract class** hay một **class** thông thường.
* **Sub Classes**: các sub class sẽ implement các phương thức của **supper class** theo nghiệp vụ riêng của nó.
* **Factory Class**: một class chịu tránh nhiệm khởi tạo các đối tượng **sub class** dựa theo tham số đầu vào. Lưu ý: lớp này là [**Singleton**](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)hoặc cung cấp một **public static method** cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

[](https://gpcoder.com/wp-content/uploads/2018/09/design-patterns-factory-method-diagram.png)

## Sử dụng Factory Pattern khi nào?

Factory Pattern được sử dụng khi:

* Chúng ta có một super class với nhiều class con và dựa trên đầu vào, chúng ta cần trả về một class con. Mô hình này giúp chúng ta đưa trách nhiệm của việc khởi tạo một lớp từ phía người dùng (client) sang lớp Factory.
* Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

## Lợi ích của Factory Pattern là gì?

Lợi ích của Factory Pattern:

* Factory Pattern giúp giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với Interface thay thì các implement. Giúp chuơng trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.
* Mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.
* Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy. Người dùng không biết logic thực sực được khởi tạo bên dưới phương thức factory.
* Dễ dạng quản lý life cycle của các Object được tạo bởi Factory Pattern.
* Thống nhất về naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

## Abstract Factory Pattern là gì?

**Abstract Factory pattern** là một trong những **Creational pattern**. Nó là phương pháp tạo ra một Super-factory dùng để tạo ra các Factory khác. Hay còn được gọi là Factory của các Factory. Abstract Factory Pattern là một Pattern cấp cao hơn so với Factory Method Pattern.

Trong Abstract Factory pattern, một interface có nhiệm vụ tạo ra một Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Mỗi Factory được tạo ra có thể tạo ra các object bằng phương pháp giống như Factory pattern.

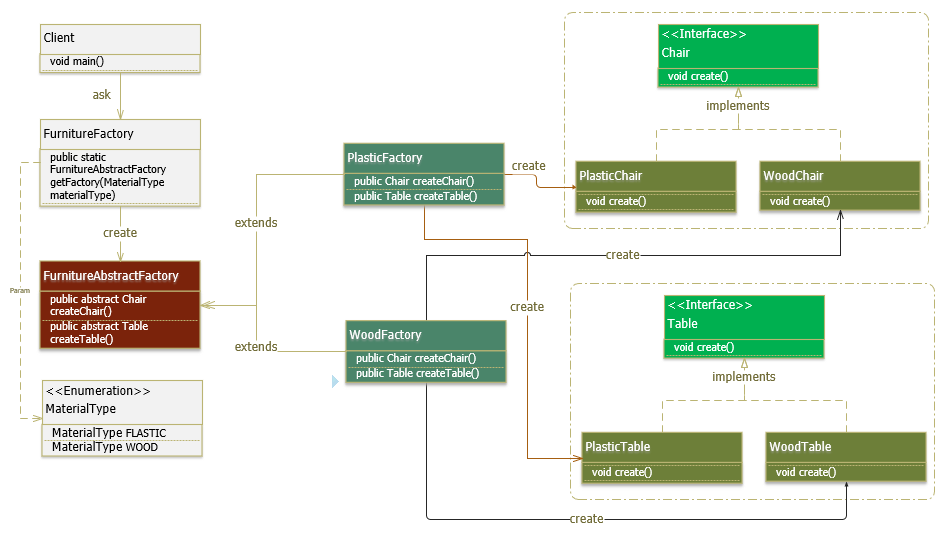
Hãy tưởng tượng, Abstract factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

Sử dụng Abstract Factory khi code của bạn cần làm việc với các biến thể của các product liên quan, nhưng không muốn phụ thuộc vào concrete class của những product đó

## Cài đặt Abstract Factory Pattern như thế nào?

Một Abstract Factory Pattern bao gồm các thành phần cơ bản sau:

* **AbstractFactory**: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
* **ConcreteFactory**: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
* **AbstractProduct**: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
* **Product**: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại AbstractProduct.
* **Client**: là đối tượng sử dụng AbstractFactory và các AbstractProduct.

[](https://gpcoder.com/wp-content/uploads/2018/09/design-patterns-abstract-factory-diagram.png)

## Lợi ích của Abstract Factory Pattern là gì?

* Các lợi ích của Factory Pattern cũng tương tự như Factory Method Pattern như: cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, …
* Giúp tránh được việc sử dụng điều kiện logic bên trong **Factory Pattern**. Khi một **Factory Method** lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình **Abstract Factory** để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).
* Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.
* Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều product khác nhau.

## Builder Pattern là gì?

**Builder pattern** là một trong những **Creational pattern**. Builder pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đôi tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng đôc lập với các đối tượng khác.

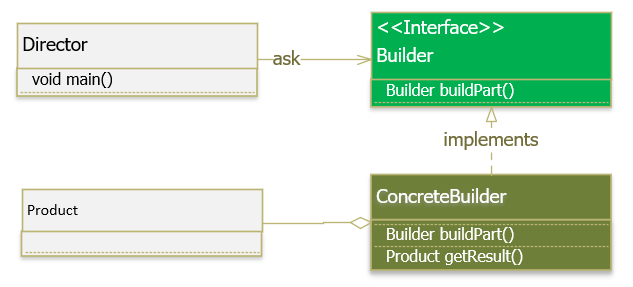
Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.

Có ba vấn đề chính với  Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:

* Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.
* Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
* Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

Chúng ta có thể xử lý những vấn đề này với một số lượng lớn các tham số bằng việc cung cấp một hàm khởi tạo với những tham số bắt buộc và các method getter/ setter để cài đặt các tham số tùy chọn. Vấn đề với hướng tiếp cận này là trạng thái của Object sẽ không nhất quán cho tới khi tất cả các thuộc tính được cài đặt một cách rõ ràng. Nếu cần xây dựng một đối tượng Immutable thì cách này cũng không thể thực hiện được.

## Cài đặt Builder Pattern như thế nào?

[](https://gpcoder.com/wp-content/uploads/2018/09/design-patterns-builder-diagram.png)

Một builder gồm các thành phần cơ bản sau:

* **Product** : đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
* **Builder** : là abstract class hoặc interface khai báo phương thức tạo đối tượng.
* **ConcreteBuilder** : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
* **Director**/ Client: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

## Lợi ích của Builder Pattern là gì?

* Hỗ trợ, loại bớt việc phải viết nhiều constructor.
* Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính (propery) bắt buộc để tạo một object từ 4 hoặc 5 propery.
* Giảm bớt số lượng constructor, không cần truyền giá trị null cho các tham số không sử dụng.
* Ít bị lỗi do việc gán sai tham số khi mà có nhiều tham số trong constructor: bởi vì người dùng đã biết được chính xác giá trị gì khi gọi phương thức tương ứng.
* Đối tượng được xây dựng an toàn hơn: bởi vì nó đã được tạo hoàn chỉnh trước khi sử dụng.
* Cung cấp cho bạn kiểm soát tốt hơn quá trình xây dựng: chúng ta có thể thêm xử lý kiểm tra ràng buộc trước khi đối tượng được trả về người dùng.
* Có thể tạo đối tượng immutable.

## Nhược điểm của Builder Pattern là gì?

Builder Pattern có nhược điểm là duplicate code khá nhiều: do cần phải copy tất cả các thuộc tính từ class Product sang class Builder.

Tăng độ phức tạp của code (tổng thể) do số lượng class tăng lên.

## Sử dụng Builder Pattern khi nào?

* Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (requried), một số không bắt buộc (optional).
* Khi có quá nhiều hàm constructor, bạn nên nghĩ đến Builder.
* Muốn tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng.
* Muốn kiểm soát quá trình xây dựng.
* Khi người dùng (client) mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.

## So sánh Builder Pattern với Factory/ Abstract Factory Pattern

Factory Pattern cũng có thể được sử dụng để xây dựng một đối tượng phức tạp, vậy sự khác biệt của nó với mô hình Builder Pattern là gì?

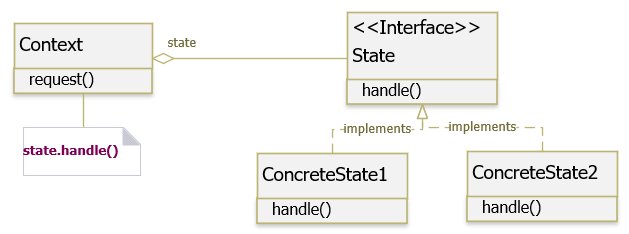
Sự khác biệt lớn duy nhất giữa Builder Pattern và Factory Pattern cung cấp cho bạn nhiều quyền kiểm soát hơn đối với quá trình tạo đối tượng.

Trong **Builder Pattern**, đối tượng được xây dựng từng bước (step by step). Builder Pattern có nhiều bước nhỏ, mỗi bước sẽ có các đơn vị logic nhỏ kèm theo trong đó. Cũng sẽ có một chuỗi (sequence) liên quan. Nó sẽ bắt đầu từ bước 1 và sẽ đi lên tối đa bước n và bước cuối cùng là trả về đối tượng. Nhưng trong Factory Pattern, bạn sẽ không thấy được đối tượng phức tạp được tạo như thế nào, nó không có từng bước xây dựng đối tượng.

## State Pattern là gì?

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

## Cài đặt State Pattern như thế nào?

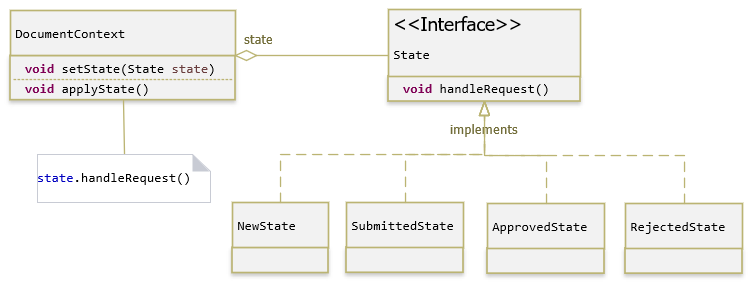
[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-state-diagram.png)

Các thành phần tham gia State Pattern:

* **Context** : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
* **State** : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
* **ConcreteState** : cài đặt các phương thức của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

Một vài điểm cần ghi nhớ khi áp dụng pattern này:

* Một đối tượng nên thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.
* Mỗi State nên được xác định độc lập.
* Thêm các trạng thái mới sẽ không làm ảnh hưởng đến các trạng thái hoặc chức năng khác.

[](https://gpcoder.com/wp-content/uploads/2019/01/design-patterns-state-example.png)

* Đầu tiên chúng ta sẽ tạo 1 base inteface để nhận yêu cầu xử lý. Lớp này gọi là **State**.
* Tiếp theo, ứng với mỗi giá trị trong enum, chúng ta sẽ tạo một class mới và implement các phương thức của State.
* Cuối cùng, chúng ta tạo một class **Context**. Class này chứa thông tin State hiện tại và nhận yêu cầu xử lý trực tiếp từ Client.

## Lợi ích của State Pattern là gì?

Lợi ích:

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Single_responsibility_principle_SRP) : tách biệt mỗi State tương ứng với 1 class riêng biệt.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) : chúng ta có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.
* Giữ hành vi cụ thể tương ứng với trạng thái.
* Giúp chuyển trạng thái một cách rõ ràng.

## Sử dụng State Pattern khi nào?

* Khi hành vi của đối tượng phụ thuộc vào trạng thái của nó và nó phải có khả năng thay đổi hành vi của nó lúc run-time theo trạng thái mới.
* Khi nhiều điều kiện phức tạp buộc đối tượng phụ thuộc vào trạng thái của nó.

## Adapter Pattern là gì?

Adapter Pattern (Người chuyển đổi) là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

Adapter Pattern còn gọi là **Wrapper Pattern** do cung cấp một interface “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có interface không tương thích với lớp đang viết.

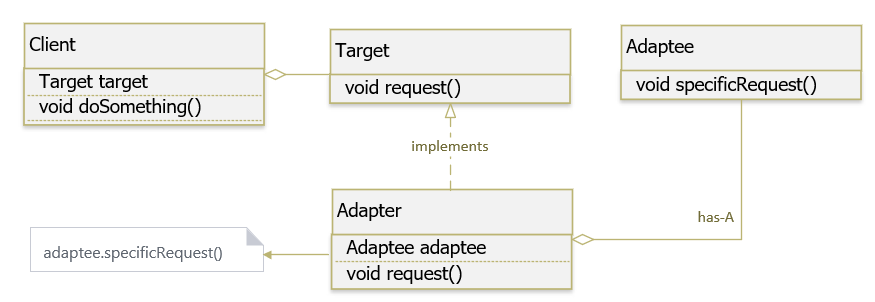
## Cài đặt Adapter Pattern như thế nào?

Một Adapter Pattern bao gồm các thành phần cơ bản sau:

* **Adaptee**: định nghĩa interface không tương thích, cần được tích hợp vào.
* **Adapter**: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
* **Target**: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
* **Client**: lớp sử dụng các đối tượng có interface Target.

Có hai cách để thực hiện Adapter Pattern dựa theo cách cài đặt (implement) của chúng:

* **Object Adapter – Composition**(Chứa trong): trong mô hình này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.

[](https://gpcoder.com/wp-content/uploads/2018/10/design-patterns-object-adapter-diagram.png)

* **Class Adapter – Inheritance**(Kế thừa) : trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.

So sánh Class Adapter với Object Adapter:

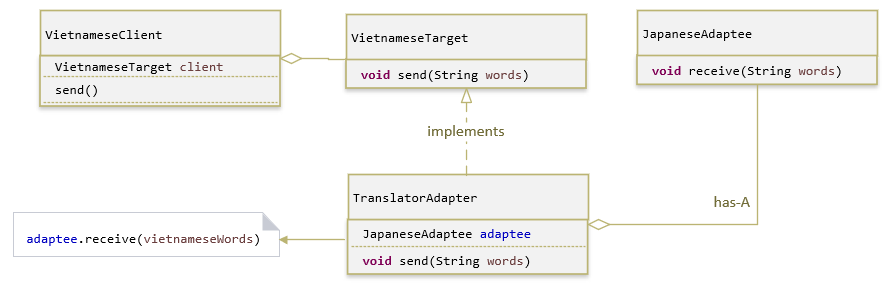
* Sự khác biệt chính là Class Adapter sử dụng Inheritance (kế thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition (chứa trong) để kết nối Adapter và Adaptee.
* Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.

Tại sao Object Adapter lại tốt hơn?

* Nó sử dụng Composition để giữ một thể hiện của Adaptee, cho phép một Adapter hoạt động với nhiều Adaptee nếu cần thiết.

### Ví dụ Adapter Pattern với ứng dụng Translation

* **Client**: người Việt sẽ là Client trong ví dụ này,vì anh ta cần gửi một số message cho người Nhật.
* **Target**: đây là nội dung message được Client cung cấp cho thông dịch viên (Translator / Adapter).
* **Adapter**: thông dịch viên (Translator) sẽ là Adapter, nhận message tiếng Việt từ Client và chuyển đổi nó sang tiếng Nhật trước khi gởi cho người Nhật.
* **Adaptee**: đây là interface hoặc class được người Nhật sử dụng để nhận message được chuyển đổi từ thông dịch viên (Translator).

[](https://gpcoder.com/wp-content/uploads/2018/10/design-patterns-adapter-diagram-translator-example.png)

## Lợi ích của Adapter Pattern là gì?

Việc sử dụng Adapter Pattern đem lại các lợi ích sau:

* Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác và giao tiếp với nhau.
* Tăng khả năng sử dụng lại thư viện với interface không thay đổi do không có mã nguồn.

Bên cạnh những lợi ích trên, nó cũng nó một số khuyết điểm nhỏ sau:

* Tất cả các yêu cầu được chuyển tiếp, do đó làm tăng thêm một ít chi phí.
* Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (chain) trước khi đến được yêu cầu thực sự.

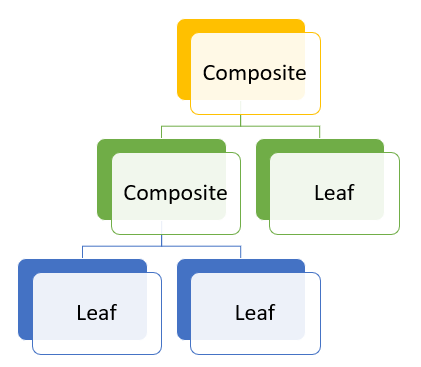
## Sử dụng Adapter Pattern khi nào?

Có thể dùng Adapter Pattern trong những trường hợp sau:

* Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.
* Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
* Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
* Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
* Khi cần đảm bảo nguyên tắc [**Open/ Close**](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) trong một ứng dụng.

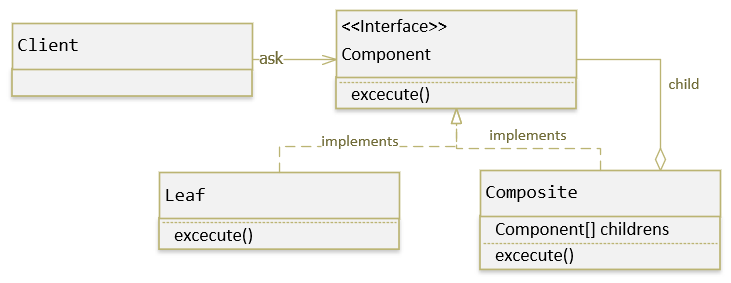
## Composite Pattern là gì?

**Composite** là một mẫu thiết kế thuộc nhóm cấu trúc (**Structural Pattern**). Composite Pattern là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.

[](https://gpcoder.com/wp-content/uploads/2018/11/Composite.png)

Composite Pattern được sử dụng khi chúng ta cần **xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object**. Composite pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng 1 object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

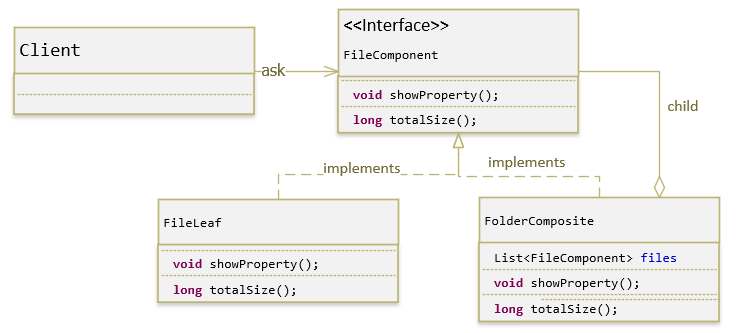
## Cài đặt Composite Pattern như thế nào?

[](https://gpcoder.com/wp-content/uploads/2018/11/design-patterns-composite-diagram.png)

Một Composite Pattern bao gồm các thành phần cơ bản sau:

* **Base Component**: là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
* **Leaf**: là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.
* **Composite :** lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.
* **Client**: sử dụng Base Component để làm việc với các đối tượng trong Composite.

**Ví dụ:** Cài đặt Composite Pattern về chương trình quản lý một hệ thống tập tin ở trên.

[](https://gpcoder.com/wp-content/uploads/2018/11/design-patterns-composite-example.png)

* Trước hết chúng ta định nghĩa một Inteface Component (FileComponent) có các phương thức chung cho cả folder và file. Để đơn giản, tôi chỉ tạo 2 phương thức showProperty() và totalSize(). Hai phương thức này sẽ cung cấp thông tin về file và tổng kích thước của nó.
* Tiếp theo, chúng ta sẽ tạo một class Leaf cài đặt các phương thức của Component (FileLeaf). Một class Composite chứa tập hợp các Leaf và cài đặt các phương thức của Component (FolderComposite).
* Cuối cùng, chúng ta sẽ tạo một class Client gọi các phương thức của FileComponent và FolderComposite. Cách gọi các phương thức của 2 class này hoàn toàn giống nhau do cùng implement một Component (FileComponent).

## Lợi ích của Composite Pattern là gì?

* Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.

## Sử dụng Composite Pattern khi nào?

* Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
* Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

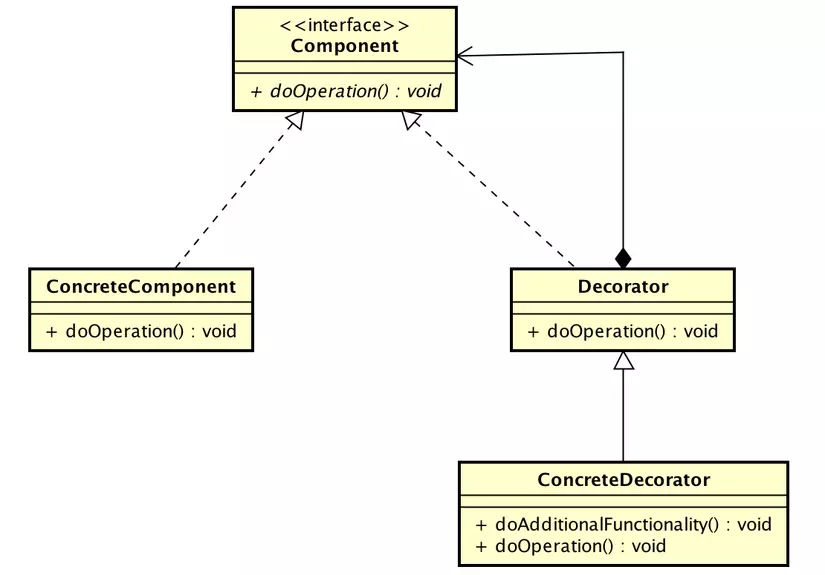
# Hiểu biết cơ bản về Decorator pattern

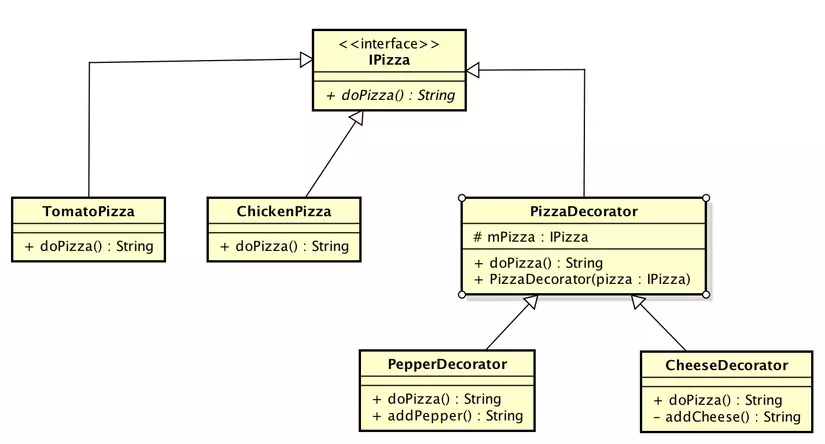
Đôi khi chúng ta cần mở rộng một phương thức trong đối tượng, và cách thông thường là chúng ta sẽ kế thừa đối tượng đó. Việc này không phải sai, nhưng trong một vài trường hợp sẽ làm cho mã nguồn trở lên phức tạp hơn chúng ta mong muốn. Đó là lý do chính cho việc ra đời của mẫu thiết kế **Decorator**, là một cách để mở rộng các phương thức một cách linh động.

Mẫu thiết kế này sẽ linh động thay đổi tính chất (functionality) đã có trong một đối tượng khi chương trình đang chạy (runtime) mà không ảnh hưởng đến các tình chất đã tồn tại của các đối tượng khác.

Mở rộng theo cách linh động là chúng ta sẽ cung cấp một cơ cấu mà cơ cấu này cho phép chúng ta thay đổi một đối tượng đã tồn tại nhưng không làm ảnh hưởng đến các đối tượng khác của cùng lớp đó.

Thêm nữa, có nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Mặt khác, mẫu thiết kế **Decorator** sẽ giảm cho bạn một lượng lớn thời gian viết code.





Pattern được áp dụng khi:

* Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ
* Cần mở rộng dự án với ít sự thay đổi nhất

Top of Form

Bottom of Form