**Design Pattern**

[1. Design pattern là gì? 2](#_Toc475691079)

[2. Tại sao cần áp dụng Design Pattern trong việc thiết kế, sản xuất phần mềm? 2](#_Toc475691080)

[3. Đặc tính của Design Pattern 2](#_Toc475691081)

[4. Các loại Design Pattern 3](#_Toc475691082)

[4.1. Nhóm khởi tạo (Creational Pattern) 3](#_Toc475691083)

[4.2. Nhóm cấu trúc tĩnh (Structural Pattern) 3](#_Toc475691084)

[4.3. Nhóm tương tác động (Behavioral Pattern) 4](#_Toc475691085)

[5. Các Design Pattern thông dụng 5](#_Toc475691086)

[5.1. Singleton 5](#_Toc475691087)

[5.2. Abstract Factory Pattern 8](#_Toc475691088)

[5.3. Factory Pattern 9](#_Toc475691089)

[5.4. Adapter Pattern 10](#_Toc475691090)

[5.5. Façade pattern 11](#_Toc475691091)

[5.6. Interpreter Pattern 12](#_Toc475691092)

[5.7. Iterator Pattern 13](#_Toc475691093)

[5.8. Strategy Pattern 13](#_Toc475691094)

[**TÀI LIỆU THAM KHẢO** 14](#_Toc475691095)

1. Design pattern là gì?

Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

* *Design patterns là tập các giải pháp cho vấn đề phổ biến trong thiết kế các hệ thống máy tính. Đây là tập các giải pháp đã được công nhận là tài liệu có giá trị, những người phát triển có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự.*

1. Tại sao cần áp dụng Design Pattern trong việc thiết kế, sản xuất phần mềm?

* Design pattern cung cấp giải pháp ở dạng tổng quát, 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, dễ dàng nâng cấp, bảo trì về sau.
* Một lợi thế lớn để sử dụng một mẫu thiết kế là lập trình viên khác sẽ có thể dễ dàng nhận ra nó (đặc biệt là nếu bạn sử dụng quy ước đặt tên tốt).

1. Đặc tính của Design Pattern

* Dễ bảo trì, quản lý và sửa đổi.
* Có thể tái sử dụng trong nhiều project.
* Cung cấp giải pháp giúp xác định được kiến trúc hệ thống.
* Nắm bắt được các kinh nghiệm kỹ thuật phần mềm.
* Là các giải pháp tốt được kiểm chứng được xây dựng dựa trên kiến thức và kinh nghiệm của các chuyên gia phát triển phần mềm.

1. Các loại Design Pattern
   1. Nhóm khởi tạo (Creational Pattern)

Khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface).

|  |  |  |
| --- | --- | --- |
| **STT** | **Tên** | **Mô tả** |
| 1 | Abstract Factory | Cung cấp 1 interface cho việc tạo lập các đối tượng (có liên hệ mật thiết với nhau) mà không cần quy định lớp khi hay xác định lớp cụ thể tạo mỗi đối tượng. |
| 2 | Builder | 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 | 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 | Quy định loại các đối tượng cần tạo bằng cách dùng 1 đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này. |
| 5 | Singleton | Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó. |

* 1. Nhóm cấu trúc tĩnh (Structural Pattern)

Dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

|  |  |  |
| --- | --- | --- |
| **STT** | **Tên** | **Mô tả** |
| 1 | Adapter | 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 | 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. |
| 4 | Decorator | Cho phép thêm chức năng mới cho một đối tượng hiện tại mà không làm thay đổi cấu trúc của nó. |
| 5 | Facade | 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 | Được sử dụng chủ yếu để giảm số lượng của các đối tượng được tạo ra, giảm bộ nhớ và tăng hiệu suất. |
| 7 | Proxy | Cung cấp các 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. |

* 1. Nhóm tương tác động (Behavioral Pattern)

Nhóm này dùng trong thực hiện các hành vi của đối tượng.

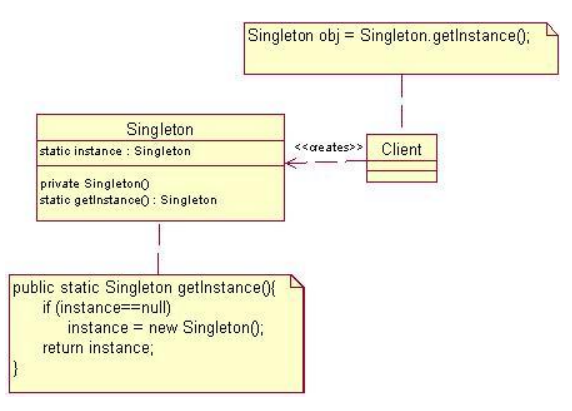
|  |  |  |
| --- | --- | --- |
| **STT** | **Tên** | **Mô tả** |
| 1 | Chain of Responsibility | Khắc phục việc ghép cặp giữa bộ gởi và bộ phận thông điệp. |
| 2 | Command | Là mẫu thiết kế hướng dữ liệu. |
| 3 | Interpreter | Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ. |
| 4 | Iterator | Truy xuất các phần tử của một đối tượng thành dạng tập hợp tuần tự (list, array, ..) mà không phụ thuộc vào biểu diễn bên trong của các phần tử. |
| 5 | Mediator | Định nghĩa đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau. |
| 6 | Memento | Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà không vi phạm việc bao bọc dữ liệu. |
| 7 | Observe | Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo |
| 8 | State | Cho phép đối tượng thay đổi hành vi khi trạng thái bên trong nó thay đổi. |

|  |  |  |
| --- | --- | --- |
| 9 | Strategy | Một hành vi của class hoặc thuật toán của nó có thể thay đổi thời gian chạy. |
| 10 | Template Method | 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 |
| 11 | Visitor | 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. |
| 12 | MVC pattern | Là một mô hình thiết kế hỗ trợ cho việc phân chia chức năng của hệ thống thành từng mảng khác nhau để quản lý. |

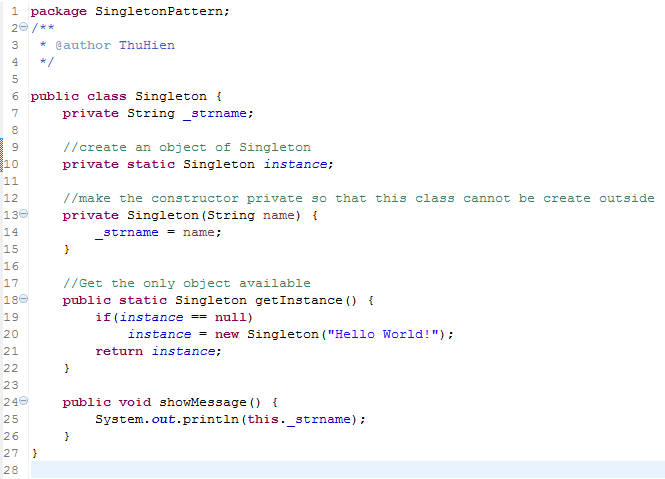
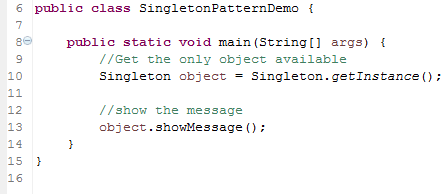
1. Các Design Pattern thông dụng
   1. Singleton

Mẫu Singleton bảo đảm chỉ có một thể hiện (instance) duy nhất trong một lớp và thể hiện này có thể được truy xuất mọi lúc mọi nơi trong chương trình.

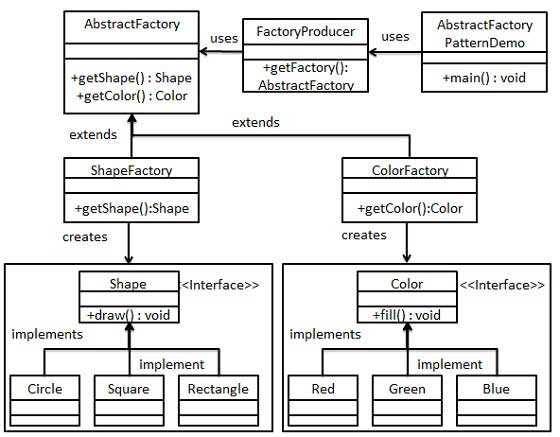
Nói cách khác, mẫu đảm bảo lớp được khởi tạo chỉ một lần và tất cả các yêu cầu đưa tới một và chỉ một đối tượng. Hơn nữa, đối tượng chỉ được tạo ra cho đến khi nó thực sự cần thiết.



Trong đó:

* Định nghĩa một thuộc tính instance trong lớp singleton dùng private và static.
* Định nghĩa tất cả các constructor thành protected hoặc private để người dùng không thể tạo thực thể trực tiếp từ bên ngoài lớp.
* Phương thức getInstance() trong lớp singleton dùng public và static.
* Khởi tạo khi yêu cầu (chỉ khi gọi phương thức getInstance() mới khởi tạo): nếu thể hiện (instance) là giá trị rỗng thì trả về 1 thể hiện (instance) mới. Dùng để báo xem lớp đó đã có thể hiện hay chưa.
* Nói cách khác, phương thức *getInstance()*: nếu chưa có đối tượng thì nó sẽ tạo đối tượng mới, còn nếu có rồi thì nó trỏ vào đối tượng đã có rồi, như vậy thì bảo đảm chương trình luôn tồn tại 1 đối tượng.
* Clients chỉ dùng phương thức getInstance() để tạo đối tượng của lớp singleton.
* **Ví dụ:** Tạo 1 lớp “Singleton”
* Lấy 1 đối tượng chỉ từ lớp Singleton.
  1. Abstract Factory Pattern

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 (Interface) gọi là “lớp sản xuất ảo”.

Abstract factory 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.

* 1. Factory Pattern

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.

* 1. Adapter Pattern

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

Nói cách khác, mẫu adapter chuyển đổi giao diện thành một giao diện khác mà phù hợp với yêu cầu. Giúp kết nối các lớp có giao diện không tương thích để làm việc với nhau.

**Ví dụ 1:** Có 1 công ty muốn mở rộng công ty bằng cách chuyển địa điểm làm việc. Có người sẽ đề xuất ý kiến là mua mới toàn bộ các cơ sở vật chất để phù hợp vối môi trường mới. Nhưng có đề xuất là ta sẽ sử dụng lại các cơ sở vật chất sẵn có và sẽ mua thêm các cơ sở vật chất mới nếu cần.

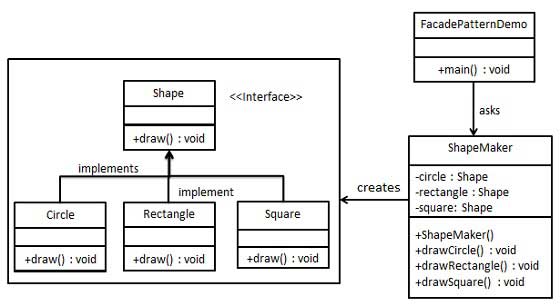
**Ví dụ 2:** Ban đầu ta có class AudioPlayer implement MediaPlayer chỉ chơi được định dạng mp3. Và 1 interface AdvancedMediaPlayer chơi được định dạng mp4, vlc. Về sau, ta muốn mở rộng AudioPlayer chơi được cả định dạng mp4, vlc.

* Cần tạo ra một MediaAdapter implement MediaPlayer và sử dụng các đối tượng của AdvancedMediaPlayer để giải quyết vấn đề trên.
  1. Façade pattern

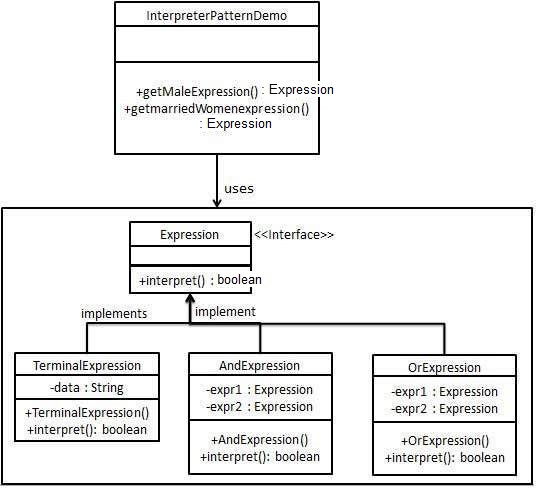
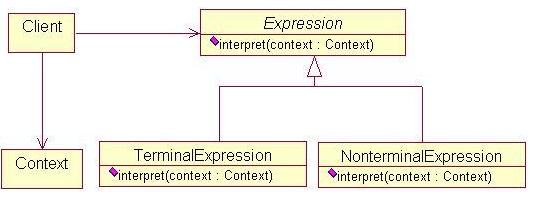
Khi việc thiết kế các pattern sinh ra rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Ta sẽ sử dụng Façade Pattern để giải quyết vấn đề đó.

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

**Lợi ích khi dùng Facade pattern:**

* Người dùng không cần biết đến sự phức tạp bên trong hệ thống con mà dễ dàng sử dụng hệ thống con vì chỉ cần giao tiếp với hệ thống con thông qua một giao diện chung đơn giản.
* Nâng cao khả năng độc lập của hệ thống con do cho phép nâng cấp đơn thể trong hệ thống con mà không cần phải sửa lại mã lệnh từ phía người dùng.
  1. Interpreter Pattern

Interpreter Pattern giúp người lập trình có thể "xây dựng" những đối tượng "động" bằng cách đọc mô tả về đối tượng (file XML hoặc gì đó) rồi sau đó "xây dựng" đối tượng đúng theo mô tả đó.

****Metadata (ngôn ngữ mô tả) --> [Interpreter Pattern] --> Đối tượng tương ứng.

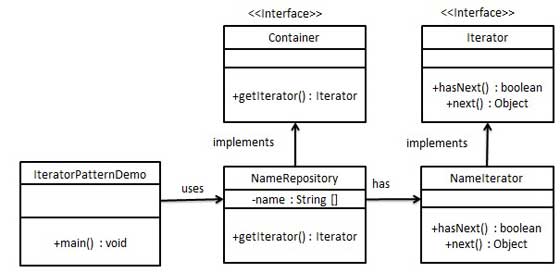
* 1. Iterator Pattern

Iterator Pattern là design pattern thường xuyên được sử dụng trong môi trường lập tình Java và .Net. Mô hình này được sử dụng để có được một cách thức truy cập các phần tử của một đối tượng theo cách tuần tự mà không cần phải biết đại diện cơ bản của nó.

Hàng ngày, bạn phải làm việc với nhiều loại tập hợp *(như cấu trúc cây, cây nhị phân, mảng, bảng băm, danh sách mảng,…)*. Cách thức mà các tập hợp này lưu trữ đối tượng của nó rất khác nhau, và nếu bạn muốn truy cập dữ liệu của những đối tượng này, bạn phải học các kỹ thuật khác nhau cho từng loại tập hợp.

* Mẫu Iterator xuất hiện, bạn có thể sử dụng một giao diện interface được xác định rõ ràng để truy cập tới từng phần tử của tập hợp. Sử dụng những phương pháp này, bạn có thể truy xuất tới các phần tử trong tập hợp theo cách cơ bản nhất.

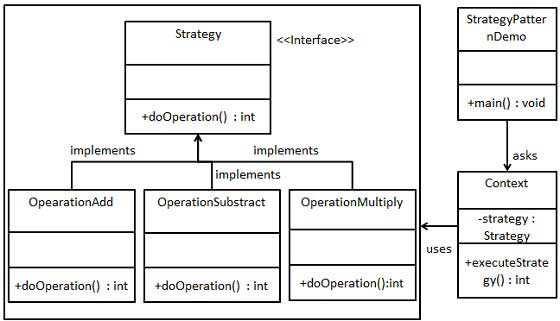
Java hỗ trợ một giao diện Iterator trong java.util.Iterator, được định nghĩa với 3 phương thức sau:

* hasNext(): Trả về giá trị True nếu vẫn còn phần tử trong tập hợp và trả về false trong trường hợp ngược lại
* next(): Trả về phần tử kế tiếp trong tập hợp
* remove(): Cho phép gỡ bỏ một phần tử trong tập hợp.
* Ta có 1 mảng chứa tên của các nhân viên trong công ty, yêu cầu xuất tất cả tên của các nhân viên trong công ty ra.
  1. Strategy Pattern

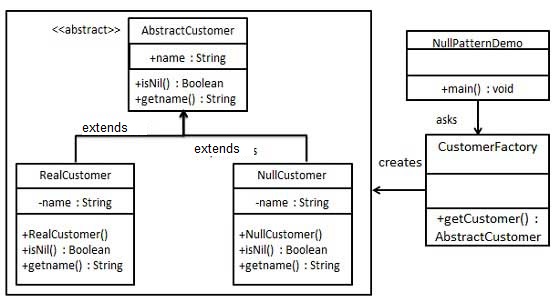
Ý nghĩa thực sự của Strategy pattern (mẫu chiến lược) là tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà bạn thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

Strategy pattern cho thấy đôi khi nó sẽ được áp dụng tốt cho mục đích hướng chức năng. Và nó đặc biệt quan trọng khi muốn thực hiện công việc nâng cấp, bảo trì cho các đoạn mã dễ thay đổi của bạn một cách riêng biệt với toàn bộ mã của chương trình, hoặc khi bạn muốn thay đổi thuật toán sử dụng khi chương trình được thực thi.

**Nên sử dụng strategy pattern cho những trường hợp sau:**

* Bạn có một đoạn code dễ thay đổi, và bạn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
* Bạn muốn tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
* Bạn muốn thay đổi thuật toán sử dụng khi chạy chương trình.
  1. Null Object Pattern

Null Object Pattern cung cấp một null object để thay thế cho trường hợp một instance bị NULL. Thay vì sử dụng một lệnh IF để check một null value, Null Object sẽ phản ánh một mối liên hệ không phải thực hiện - không làm gì cả.

Khi các khối lệnh IF vô nghĩa tồn tại trong ứng dụng thì bạn nên nghĩ tới việc sử dụng Null Object Design Pattern. Nó là giải pháp hay giúp cho source code của bạn đơn giản, tinh tế hơn rất nhiều với việc check null theo cách truyền thống.

# **TÀI LIỆU THAM KHẢO**

<http://diendan.congdongcviet.com/threads/t76502::tong-hop-series-bai-dich-thiet-ke-mau-design-patterns-for-dummies.cpp>

<http://laptrinh.vn/d/4346-design-pattern-la-gi-huong-dan-design-pattern.html>

<https://www.tutorialspoint.com/design_pattern/index.htm>

<https://sourcemaking.com/design_patterns>

<http://www.oodesign.com/>