**LOW LEVEL DESIGN**

**Overview**

-Là đơn vị thiết kế thứ 5 trong CNPM

-Trái ngược và tương phản với High Level Design (Thiết kế cấp độ cao)

-Trong công nghệ hiện nay có 7 mấu thiết kế cơ bản: Singleton, Strategy, State, Facade, Decorator, MVC, và MVP.

-Mục tiêu

+Học về bản chất của mẫu thiết kế cấp độ thấp (Low Level Design) cho hệ thống phần mềm thực thi, thực hiện.

+Học cách thực hành tốt nhất để áp dụng mô hình thiết kế này.

+Tìm ra làm thế nào và khi nào dùng các mẫu thiết kế đó.

**Introduction**

Điều này đưa lại một số phản hồi để tránh gặp phải những thiết kế nghèo nàn sau này. Nhưng trước khi có thể thực hiện các thiết kế cao, cần phải suy nghĩ về những thiết kế ở mức độ thấp hơn.

Đó là thay vì suy nghĩ về ý định thiết kế của bản thân, cần phải di chuyển sang một không gian cơ học hơn. Bởi một khi các nhà phát triển bắt đầu mã hóa, cần nghĩ về việc thiết kế đó sẽ được thực hiện như thế nào bởi họ.

Vì vậy, để làm được điều này, thường giữ ba nguyên tắc trong tâm trí mình để có thể vận hành thiết kế cấp cao vào một cái gì đó mà có thể chuyển trực tiếp thành mã.

Đầu tiên là muốn tóm lược những thay đổi.

Tiếp theo, muốn đảm bảo rằng mình đang lập trình để giao diện.

Và cuối cùng, muốn ủng hộ những thành phần kế thừa.

vậy, hãy cùng nói về từng vấn đề lần lượt nào.

Một điều cần ghi nhớ về các nguyên tắc thiết kế cấp thấp này là có một số lượng đáng kinh ngạc của sự chồng chéo với mức độ cao trong nguyên tắc thiết kế mà đã nói về trước đó.

Đây thực sự là các cách để vận hành những nguyên tắc cấp cao trong thực tế.

Vì vậy, khi nghĩ về việc tóm lược lại những thay đổi thì cái thực sự muốn làm là muốn được thiết kế hệ thống của mình theo cách như vậy. Cái mà nó có thể được dễ dàng mở rộng trong tương lai.

Và quay ngược trở lại vấn đề rằng là, điều này giúp các lỗi xảy ra bên trong các hệ thống đó được giới thiệu vào chương trình của được dễ dàng giải quyết trong không gian có thể xác định được, đúng chứ?

không muốn những sai lầm của mình trở nên rải rác trong hệ thống của .

Trong các hệ thống ngày trước, hằng số duy nhất thực sự có thể dựa vào

là cái mà hệ thống của sẽ phải thay đổi để thích ứng với nhu cầu trong tương lai.

Vì vậy, khi đang thiết kế hệ thống của mình , thực sự muốn suy nghĩ về việc xây dựng các quan điểm trừu tượng thích hợp để các nhà phát triển trong tương lai có thể thêm vào đó các chức năng mới càng dễ dàng càng tốt.

**(**Dịch lại trong khóa học**)**

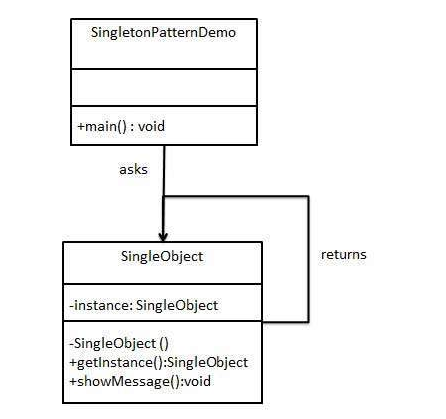
**Singleton**

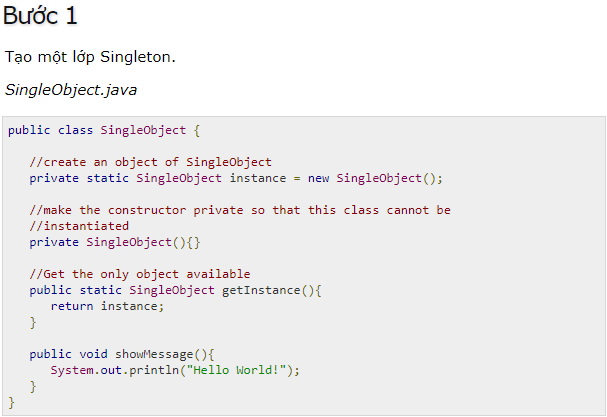
-Mẫu Singleton là một trong những mẫu thiết kế đơn giản nhất trong Java. Kiểu thiết kế kiểu mẫu này được tạo ra theo mô hình tạo mẫu vì mẫu này cung cấp một trong những cách tốt nhất để tạo ra một đối tượng.

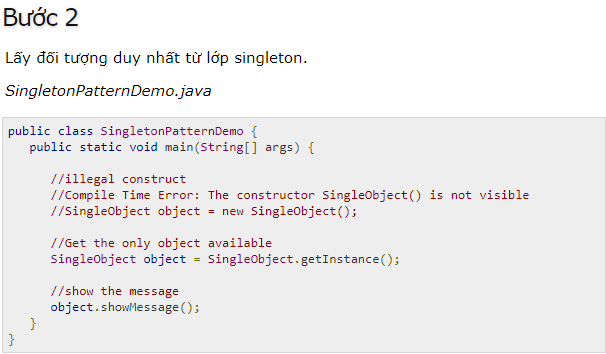
-Mô hình này liên quan đến một lớp duy nhất có trách nhiệm tạo ra một đối tượng trong khi đảm bảo rằng chỉ có một đối tượng được tạo ra. Lớp này cung cấp một cách để truy cập đối tượng duy nhất của nó mà có thể được truy cập trực tiếp mà không cần phải instance đối tượng của lớp.

-Chúng ta sẽ tạo ra một lớp *SingleObject* . *SingleObject* lớp có constructor của nó như là private và có static của chính nó.

*SingleObject* lớp cung cấp một phương thức static để được tĩnh đến thế giới bên ngoài. *SingletonPatternDemo* , lớp giới thiệu của chúng tôi sẽ sử dụng lớp *SingleObject* để có được một đối tượng *SingleObject*







Bước 3:

Xác minh đầu ra

**Nguồn: https://www.tutorialspoint.com/design\_pattern/singleton\_pattern.htm**

**Strategy**

-Để có thể thay đổi hành vi của một nền tảng ứng dụng trên những yêu cầu mới là một nhiệm vụ tiến hóa chung,và một trong đó là mô hình thiết kế chiến lược đã được thiết kế rõ ràng.vấn đề mà mẫu thiết kế chiến lược đang cố gắng để giải quyết là chúng ta thường không biết hết những thay đổi có thể có trong thời gian kế tiếp.vì vậy,chúng ta muốn cung cấp một cơ chế đóng gói thuật toán để cho phép mở rộng trong tương lai.

* Trong mô hình Chiến lược, một hành vi lớp hoặc thuật toán của nó có thể được thay đổi trong thời gian chạy. Kiểu thiết kế kiểu này thuộc kiểu hành vi.
* Trong mô hình Chiến lược, chúng ta tạo các đối tượng đại diện cho các chiến lược khác nhau và một đối tượng ngữ cảnh có hành vi khác nhau theo đối tượng chiến lược của nó. Đối tượng chiến lược thay đổi thuật toán thực thi của đối tượng ngữ cảnh.

Step 1

Khởi tạo interface

public interface Strategy {

public int doOperation(int num1, int num2);

}

Step 2

Khởi tạo lớp cụ thể thực hiện giống như interface

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

Step 3

Khởi tạo lớp Context

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

Step 4

Sử dụng *Context* để thấy được những thay đổi trong hành vi khi thay đổi

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

Step 5

Kết quả

10 + 5 = 15

10 - 5 = 5

10 \* 5 = 50

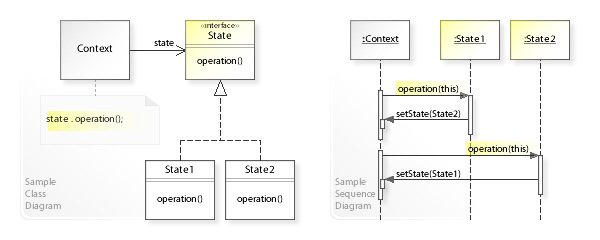
**State(Trạng thái)**

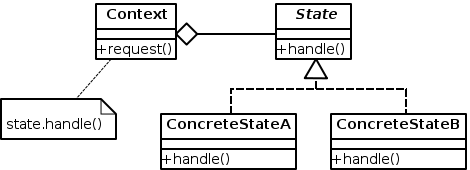
-Mẫu thiết kế State cho phép chương trình của chúng ta nhiều cách thể hiện linh hoạt, đối lập với mẫu thiết kế Strategy cho chương trình của chúng ta thế hiện tĩnh(static).

-Vấn đề mà mô hình trạng thái đang cố gắng giải quyết là chúng ta thường muốn sự thể hiện đa dạng của chương trình dựa trên những trạng thái nội bộ của chương trình.

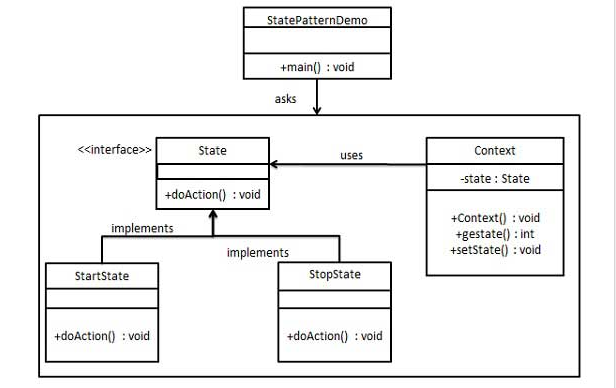
-Các trạng thái được đóng gói trong một đối tượng trạng thái độc lập, thay đổi khi trạng thái nội bộ chương trình thay đổi, cho phép các đối tượng này chuyển tiếp giữa chúng, dễ dàng hơn khi thêm một trạng thái mới vào chương trình mà không cần sửa đổi những trạng thái đã có trước đó(hướng đối tượng).

-Ví dụ về cấu trúc mô hình trạng thái:





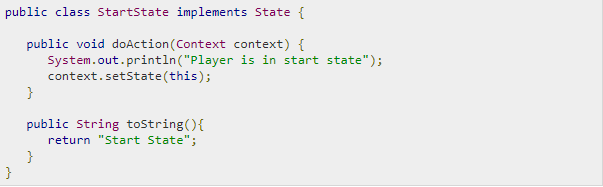
-Ví dụ: Một mô hình Trạng thái như hính dưới

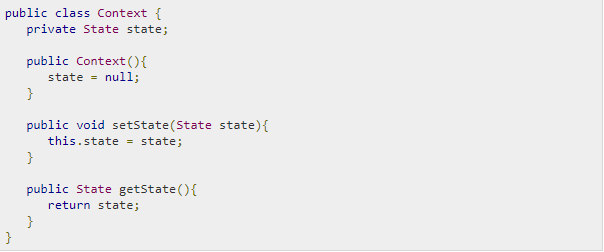


-Bước 1: Tạo giao diện State

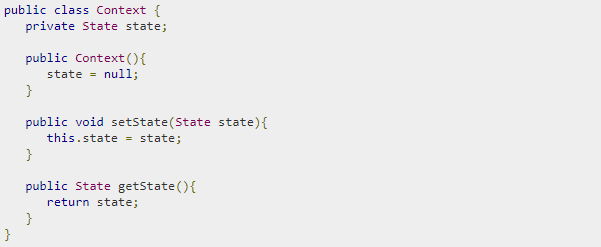


-Bước 2: Tạo lớp StartState và StopState theo giao diện State





-Bước 3: Tạo lớp Context



-Bước 4: Tạo phương thức và lớp chính

