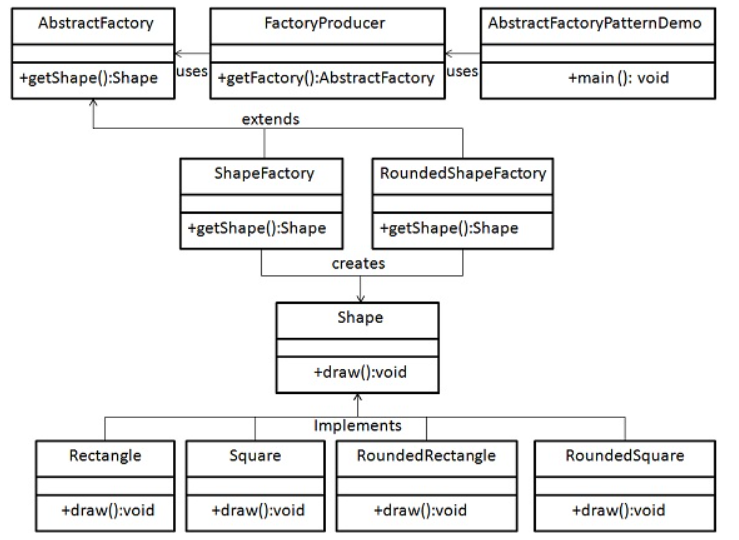
**Abstract Factory**

Mục đích: Cung cấp một cách hỗ trợ việc quản lý và tạo ra các đối tượng cùng nhóm. Như tên của pattern này thì nó giống như một nhà máy sản sinh ra các đối tượng.

Chúng ta sẽ tạo ra một interface Shape và một số class Implements nó. Ta tạo ra một abstract factory class AbstractFactory như bước tiếp theo. Factory class ShapeFactory được xác định, extends AbstractFactory. Một factory creator/generator class FactoryProducer được tạo ra.

AbstractFactoryPatternDemo, class demo của chúng ta sử dụng FactoryProducer để có được một đối tượng AbstractFactory. Nó sẽ chuyển thông tin (CIRCLE / RECTANGLE / SQUARE for Shape) đến AbstractFactory để có được loại đối tượng cần thiết.



## Step 1

Tạo một interface Shape.

**public** **interface** Shape {

**void** draw();

}

## Step 2

Tạo các concrete class implement interface.

**public** **class** RoundedRectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside RoundedRectangle::draw() method.");

}

}

------------------------------------------------------------------------------

**public** **class** RoundedSquare **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside RoundedSquare::draw() method.");

}

}

------------------------------------------------------------------------------

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Rectangle::draw() method.");

}

}

------------------------------------------------------------------------------

**public** **class** Square **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Square::draw() method.");

}

}

## Step 3

Tạo một Abstract class để có được nhà máy cho các đối tượng hình dạng bình thường và viền tròn.

**public** **abstract** **class** AbstractFactory {

**abstract** Shape getShape(String shapeType);

}

## Step 4

Tạo các Factory class extending AbstractFactory để tạo ra các đối tượng của concrete class dựa trên thông tin đã nhận.

**public** **class** ShapeFactory **extends** AbstractFactory {

@Override

**public** Shape getShape(String shapeType) {

**if** (shapeType.equalsIgnoreCase("RECTANGLE")) {

**return** **new** Rectangle();

} **else** **if** (shapeType.equalsIgnoreCase("SQUARE")) {

**return** **new** Square();

}

**return** **null**;

}

}

------------------------------------------------------------------------------

**public** **class** RoundedShapeFactory **extends** AbstractFactory {

@Override

**public** Shape getShape(String shapeType) {

**if** (shapeType.equalsIgnoreCase("RECTANGLE")) {

**return** **new** RoundedRectangle();

} **else** **if** (shapeType.equalsIgnoreCase("SQUARE")) {

**return** **new** RoundedSquare();

}

**return** **null**;

}

}

## Step 5

Tạo một class FactoryProducer để có được các nhà máy bằng cách chuyển một thông tin như Shape.

**public** **class** FactoryProducer {

**public** **static** AbstractFactory getFactory(**boolean** rounded) {

**if** (rounded) {

**return** **new** RoundedShapeFactory();

} **else** {

**return** **new** ShapeFactory();

}

}

}

## Step 6

Sử dụng FactoryProducer để có được AbstractFactory để có được các nhà máy của các concrete class bằng cách thông qua một thông tin như loại.

**public** **class** AbstractFactoryPatternDemo {

**public** **static** **void** main(String[] args) {

// get shape factory

AbstractFactory shapeFactory = FactoryProducer.*getFactory*(**false**);

// get an object of Shape Rectangle

Shape shape1 = shapeFactory.getShape("RECTANGLE");

// call draw method of Shape Rectangle

shape1.draw();

// get an object of Shape Square

Shape shape2 = shapeFactory.getShape("SQUARE");

// call draw method of Shape Square

shape2.draw();

// get shape factory

AbstractFactory shapeFactory1 = FactoryProducer.*getFactory*(**true**);

// get an object of Shape Rectangle

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

// call draw method of Shape Rectangle

shape3.draw();

// get an object of Shape Square

Shape shape4 = shapeFactory1.getShape("SQUARE");

// call draw method of Shape Square

shape4.draw();

}

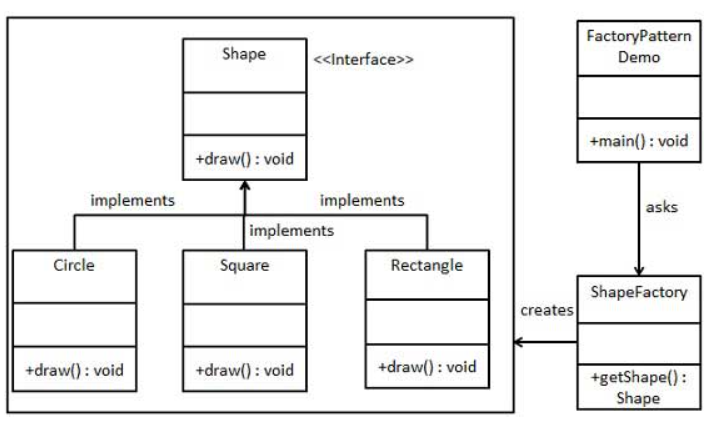
}

## Factory Method

Đây là pattern có thể nói là được sử dụng phổ biến nhất, với mục đích sử dụng để quản lý và khởi tạo các đối tượng một cách linh hoạt.

Chúng ta sẽ tạo một interface Shape và các class implement interface Shape. Một factory class ShapeFactory được xác định là bước tiếp theo.

FactoryPatternDemo, class demo của chúng ta sẽ sử dụng ShapeFactory để có được một đối tượng Shape. Nó sẽ chuyển thông tin (CIRCLE / RECTANGLE / SQUARE ) đến ShapeFactory để có được loại đối tượng cần thiết.



## Step 1

Tạo một interface.

**public** **interface** Shape {

**void** draw();

}

## Step 2

Tạo các concrete class implement interface trên.

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Rectangle::draw() method.");

}

}

------------------------------------------------------------------------------

**public** **class** Square **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Square::draw() method.");

}

}

------------------------------------------------------------------------------

**public** **class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Inside Circle::draw() method.");

}

}

## Step 3

Tạo một Factory để tạo ra các đối tượng của các concrete class dựa trên thông tin nhận được.

**public** **class** ShapeFactory {

// use getShape method to get object of type shape

**public** Shape getShape(String shapeType) {

**if** (shapeType == **null**) {

**return** **null**;

}

**if** (shapeType.equalsIgnoreCase("CIRCLE")) {

**return** **new** Circle();

} **else** **if** (shapeType.equalsIgnoreCase("RECTANGLE")) {

**return** **new** Rectangle();

} **else** **if** (shapeType.equalsIgnoreCase("SQUARE")) {

**return** **new** Square();

}

**return** **null**;

}

}

## Step 4

Sử dụng Factory để có được đối tượng của các concrete class dựa trên thông tin như loại.

**public** **class** FactoryPatternDemo {

**public** **static** **void** main(String[] args) {

ShapeFactory shapeFactory = **new** ShapeFactory();

// get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

// call draw method of Circle

shape1.draw();

// get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

// call draw method of Rectangle

shape2.draw();

// get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

// call draw method of square

shape3.draw();

}

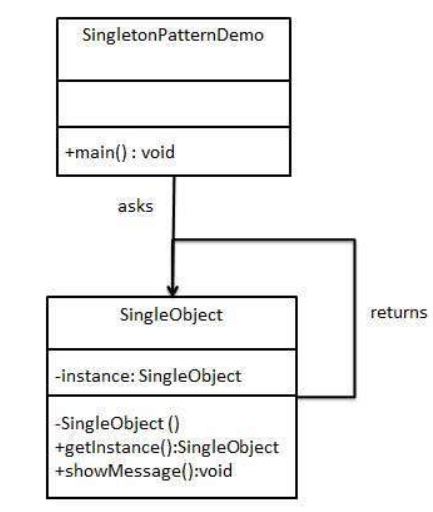
}

**Singleton**

1. Đảm bảo rằng một class chỉ có duy nhất một instance
2. Và cung cấp một cách truy cập tới instance đó mà không cần khởi tạo.

Chúng ta sẽ tạo một class SingleObject. Class đã có private constructor và một static instance của chính nó.

Class SingleObject cung cấp một phương thức static để lấy được static instance của nó ở bên ngoài. SingletonPatternDemo, class demo của chúng ta sẽ sử dụng class SingleObject để lấy một đối tượng SingleObject.



## Step 1

Tạo một class Singleton.

**public** **class** SingleObject {

// create an object of SingleObject

**private** **static** SingleObject *instance* = **new** SingleObject();

// make the constructor private so that this class cannot be

// instantiated

**private** SingleObject() {

}

// Get the only object available

**public** **static** SingleObject getInstance() {

**return** *instance*;

}

**public** **void** showMessage() {

System.***out***.println("Hello World!");

}

}

## Step 2

Lấy đối tượng duy nhất ra từ class Singleton.

**public** **class** SingletonPatternDemo {

**public** **static** **void** main(String[] args) {

// illegal construct

// Compile Time Error: The constructor SingleObject() is not visible

// SingleObject object = new SingleObject();

// Get the only object available

SingleObject object = SingleObject.*getInstance*();

// show the message

object.showMessage();

}

}

**Adapter**

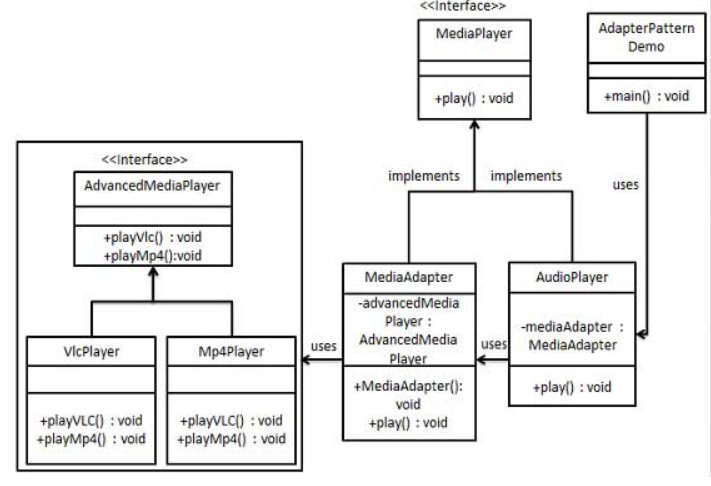
Adapter Pattern là pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi giao diện của một hay nhiều lớp có sẵn thành một giao diện khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các giao diện khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua giao diện trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết. Adapter Pattern còn gọi là Wrapper Pattern do cung cấp một giao diện “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có giao diện không tương thích với lớp đang viết.

Chúng ta có một interface MediaPlayer và một class AudioPlayer implement interface đó. AudioPlayer có thể phát các tệp âm thanh định dạng mp3 theo mặc định.

Chúng ta có một interface khác: AdvancedMediaPlayer và các class implement giao diện đó. Các class này có thể phát các tệp định dạng vlc và mp4.

Chúng ta muốn làm cho AudioPlayer chơi được các định dạng khác. Để đạt được điều này, chúng ta đã tạo ra một adapter class MediaAdapter implement MediaPlayer  và sử dụng các đối tượng AdvancedMediaPlayer để chơi các định dạng cần thiết.

AudioPlayer sử dụng adapter class MediaAdapter chuyển qua nó loại tệp âm thanh mong muốn mà không cần biết các class thực tế có thể chơi các định dạng đó. Demo class AdapterPatternDemo sẽ sử dụng class AudioPlayer để chơi các định dạng khác nhau.



## Step 1

Tạo các interface Media Player và Advanced Media Player.

**public** **interface** MediaPlayer {

**public** **void** play(String audioType, String fileName);

}

------------------------------------------------------------------------------

**public** **interface** AdvancedMediaPlayer {

**public** **void** playVlc(String fileName);

**public** **void** playMp4(String fileName);

}

## Step 2

Tạo các concrete class implement *AdvancedMediaPlayer*.

**public** **class** VlcPlayer **implements** AdvancedMediaPlayer {

@Override

**public** **void** playVlc(String fileName) {

System.***out***.println("Playing vlc file. Name: " + fileName);

}

@Override

**public** **void** playMp4(String fileName) {

// do nothing

}

}

------------------------------------------------------------------------------

**public** **class** Mp4Player **implements** AdvancedMediaPlayer {

@Override

**public** **void** playVlc(String fileName) {

// do nothing

}

@Override

**public** **void** playMp4(String fileName) {

System.***out***.println("Playing mp4 file. Name: " + fileName);

}

}

## Step 3

Tạo adapter class implement *MediaPlayer*.

**public** **class** MediaAdapter **implements** MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

**public** MediaAdapter(String audioType) {

**if** (audioType.equalsIgnoreCase("vlc")) {

advancedMusicPlayer = **new** VlcPlayer();

} **else** **if** (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer = **new** Mp4Player();

}

}

@Override

**public** **void** play(String audioType, String fileName) {

**if** (audioType.equalsIgnoreCase("vlc")) {

advancedMusicPlayer.playVlc(fileName);

} **else** **if** (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer.playMp4(fileName);

}

}

}

## Step 4

Tạo concrete class implement *MediaPlayer*.

**public** **class** AudioPlayer **implements** MediaPlayer {

MediaAdapter mediaAdapter;

@Override

**public** **void** play(String audioType, String fileName) {

// inbuilt support to play mp3 music files

**if** (audioType.equalsIgnoreCase("mp3")) {

System.***out***.println("Playing mp3 file. Name: " + fileName);

}

// mediaAdapter is providing support to play other file formats

**else** **if** (audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")) {

mediaAdapter = **new** MediaAdapter(audioType);

mediaAdapter.play(audioType, fileName);

}

**else** {

System.***out***.println("Invalid media. " + audioType + " format not supported");

}

}

}

## Step 5

Sử dụng AudioPlayer để phát các loại định dạng âm thanh khác nhau.

**public** **class** AdapterPatternDemo {

**public** **static** **void** main(String[] args) {

AudioPlayer audioPlayer = **new** AudioPlayer();

audioPlayer.play("mp3", "beyond the horizon.mp3");

audioPlayer.play("mp4", "alone.mp4");

audioPlayer.play("vlc", "far far away.vlc");

audioPlayer.play("avi", "mind me.avi");

}

}

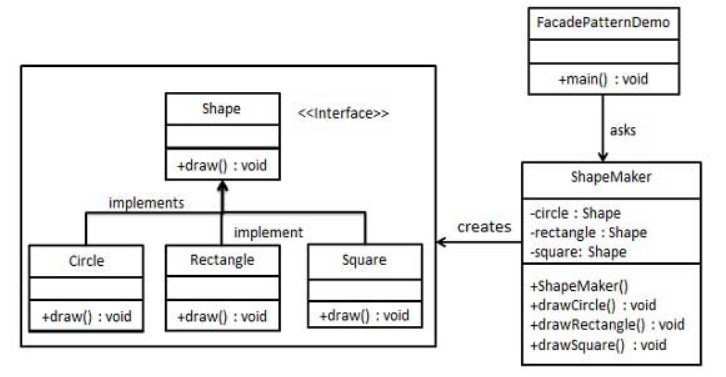
**Facade**

Cung cấp một giao diện thống nhất cho một tập hợp các giao diện trong một hệ thống con. Facade định nghĩa một giao diện cấp cao hơn làm cho hệ thống con dễ sử dụng hơn.

Bao bọc một hệ thống con phức tạp với một giao diện đơn giản.

Chúng ta sẽ tạo ra một interface Shape  và các class implement interface đó. Một facade class ShapeMaker được xác định là bước tiếp theo.

Class ShapeMaker sử dụng các class trên để uỷ quyền cho người dùng gọi tới các class này. Demo class FacadePatternDemo sẽ dùng class ShapeMaker để hiển thị kết quả.



## Step 1

Tạo một interface.

**public** **interface** Shape {

**void** draw();

}

## Step 2

Tạo các concrete class implement interface đó.

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Rectangle::draw()");

}

}

------------------------------------------------------------------------------

**public** **class** Square **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Square::draw()");

}

}

------------------------------------------------------------------------------

**public** **class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Circle::draw()");

}

}

## Step 3

Tạo một facade class.

**public** **class** ShapeMaker {

**private** Shape circle;

**private** Shape rectangle;

**private** Shape square;

**public** ShapeMaker() {

circle = **new** Circle();

rectangle = **new** Rectangle();

square = **new** Square();

}

**public** **void** drawCircle() {

circle.draw();

}

**public** **void** drawRectangle() {

rectangle.draw();

}

**public** **void** drawSquare() {

square.draw();

}

}

## Step 4

Sử dụng facade để vẽ các loại hình dạng khác nhau.

**public** **class** FacadePatternDemo {

**public** **static** **void** main(String[] args) {

ShapeMaker shapeMaker = **new** ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

shapeMaker.drawSquare();

}

}

**Decorator**

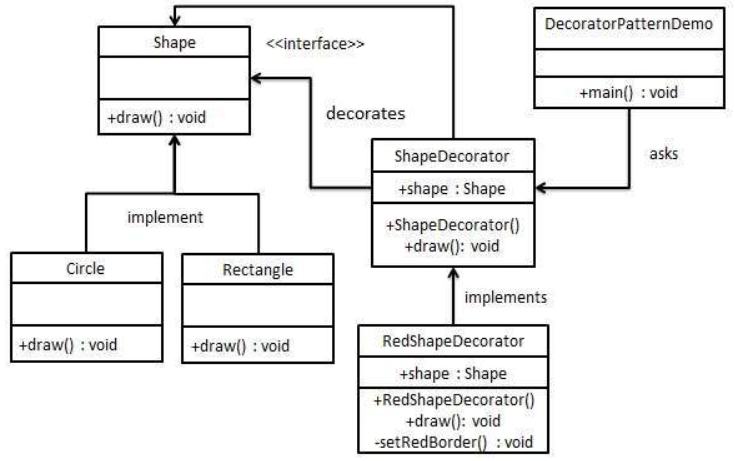
Decorator pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một class bao bọc (wrap) cho class hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

Decorator pattern sử dụng [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/#Cai_dat_Adapter_Pattern_nhu_the_nao) thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.

Chúng ta sẽ tạo một interface Shape  và các class implement nó. Sau đó chúng ta tạo một abstract decorator class ShapeDecorator implement interface Shape và có đối tượng của Shape như là biến instance của nó.

RedShapeDecorator là class implement ShapeDecorator.

Demo class DecoratorPatternDemo sẽ dùng RedShapeDecorator để trang trí các đối tượng Shape.



## Step 1

Tạo một interface.

**public** **interface** Shape {

**void** draw();

}

## Step 2

Tạo các concrete class implement interface đó.

**public** **class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Shape: Rectangle");

}

}

------------------------------------------------------------------------------

**public** **class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Shape: Circle");

}

}

## Step 3

Tạo abstract decorator class implement interface Shape.

**public** **abstract** **class** ShapeDecorator **implements** Shape {

**protected** Shape decoratedShape;

**public** ShapeDecorator(Shape decoratedShape) {

**this**.decoratedShape = decoratedShape;

}

**public** **void** draw() {

decoratedShape.draw();

}

}

## Step 4

Tạo concrete decorator class extend class *ShapeDecorator*.

**public** **class** RedShapeDecorator **extends** ShapeDecorator {

**public** RedShapeDecorator(Shape decoratedShape) {

**super**(decoratedShape);

}

@Override

**public** **void** draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

**private** **void** setRedBorder(Shape decoratedShape) {

System.***out***.println("Border Color: Red");

}

}

## Step 5

Sử dụng *RedShapeDecorator* để trang trí các đối tượng *Shape*.

**public** **class** DecoratorPatternDemo {

**public** **static** **void** main(String[] args) {

Shape circle = **new** Circle();

Shape redCircle = **new** RedShapeDecorator(**new** Circle());

Shape redRectangle = **new** RedShapeDecorator(**new** Rectangle());

System.***out***.println("Circle with normal border");

circle.draw();

System.***out***.println("\nCircle of red border");

redCircle.draw();

System.***out***.println("\nRectangle of red border");

redRectangle.draw();

}

}

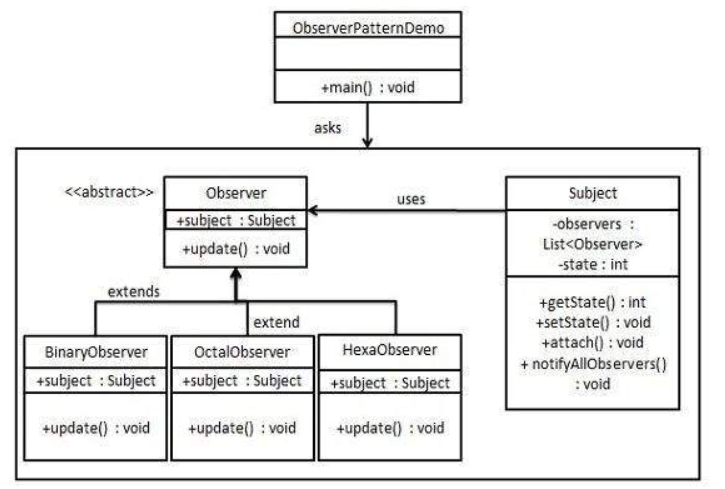
**Observer**

Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, Observer Pattern sẽ được gỡ khỏi hệ thống.

Observer pattern sử dụng 3 actor class: Subject, Observer and Client. Subject là một đối tượng có phương thức để đính kèm và tách các observer vào một đối tượng Client. Chúng ta tạo một abstract class Observer và một class Subject  extend class Observer.

Demo class ObserverPatternDemo sẽ dùng Subject và đối tượng của class để hiển thị observer pattern trong hành động.



## Step 1

Tạo class Subject.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Subject {

**private** List<Observer> observers = **new** ArrayList<Observer>();

**private** **int** state;

**public** **int** getState() {

**return** state;

}

**public** **void** setState(**int** state) {

**this**.state = state;

notifyAllObservers();

}

**public** **void** attach(Observer observer) {

observers.add(observer);

}

**public** **void** notifyAllObservers() {

**for** (Observer observer : observers) {

observer.update();

}

}

}

## Step 2

Tạo class Observer.

**public** **abstract** **class** Observer {

**protected** Subject subject;

**public** **abstract** **void** update();

}

## Step 3

Tạo các concrete class observer.

**public** **class** BinaryObserver **extends** Observer {

**public** BinaryObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("Binary String: " + Integer.*toBinaryString*(subject.getState()));

}

}

------------------------------------------------------------------------------

**public** **class** OctalObserver **extends** Observer {

**public** OctalObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("Octal String: " + Integer.*toOctalString*(subject.getState()));

}

}

------------------------------------------------------------------------------

**public** **class** HexaObserver **extends** Observer {

**public** HexaObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("Hex String: " + Integer.*toHexString*(subject.getState()).toUpperCase());

}

}

## Step 4

Dùng *Subject* và đối tượng các concrete class.

**public** **class** ObserverPatternDemo {

**public** **static** **void** main(String[] args) {

Subject subject = **new** Subject();

**new** HexaObserver(subject);

**new** OctalObserver(subject);

**new** BinaryObserver(subject);

System.***out***.println("First state change: 15");

subject.setState(15);

System.***out***.println("Second state change: 10");

subject.setState(10);

}

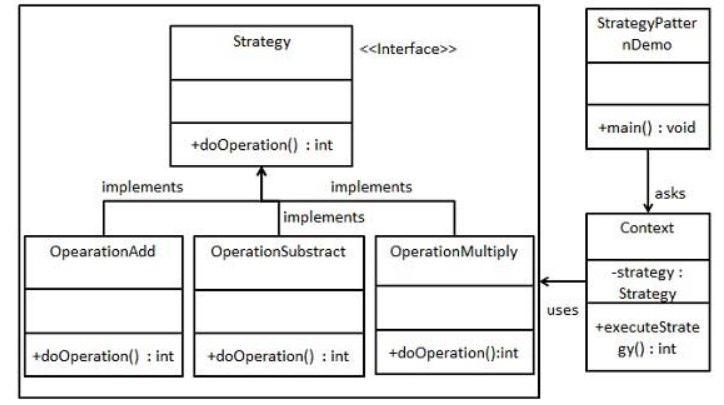
}

**Strategy**

Strategy Pattern là giúp 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 chúng ta 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.

Chúng ta sẽ tạo một interface Strategy  xác định một hành động và các strategy class implement interface đó. Context là một class sử dụng một Strategy.

Demo class StrategyPatternDemo sẽ dùng Context và các đối tượng strategy để chứng minh sự thay đổi trong hành vi của Context dựa trên strategy mà nó triển khai hoặc sử dụng.



## Step 1

Tạo một interface.

**public** **interface** Strategy {

**public** **int** doOperation(**int** num1, **int** num2);

}

## Step 2

Tạo các concrete class implement interface đó.

**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

Tạo class *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 sự thay đổi trong hành vi khi nó đổi *Strategy*.

**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());

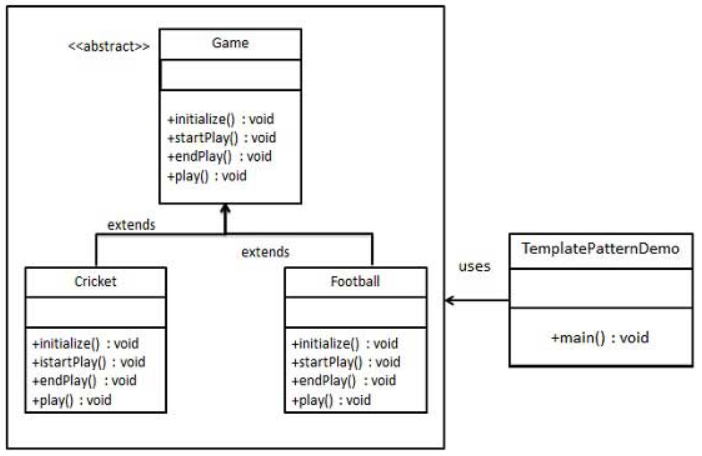
System.***out***.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

**Template method**

**Template Method Pattern** là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Template method giúp cho chúng ta tạo nên một bộ khung (template) cho một vấn đề đang cần giải quyết. Trong đó các đối tượng cụ thể sẽ có cùng các bước thực hiện, nhưng trong mỗi bước thực hiện đó có thể khác nhau. Điều này sẽ tạo nên một cách thức truy cập giống nhau nhưng có hành động và kết quả khác nhau.



## Step 1

Tạo một abstract class với một phương thức template là final.

**public** **abstract** **class** Game {

**abstract** **void** initialize();

**abstract** **void** startPlay();

**abstract** **void** endPlay();

// template method

**public** **final** **void** play() {

// initialize the game

initialize();

// start game

startPlay();

// end game

endPlay();

}

}

## Step 2

Tạo các concrete class extend class trên.

**public** **class** Cricket **extends** Game {

@Override

**void** endPlay() {

System.***out***.println("Cricket Game Finished!");

}

@Override

**void** initialize() {

System.***out***.println("Cricket Game Initialized! Start playing.");

}

@Override

**void** startPlay() {

System.***out***.println("Cricket Game Started. Enjoy the game!");

}

}

------------------------------------------------------------------------------

**public** **class** Football **extends** Game {

@Override

**void** endPlay() {

System.***out***.println("Football Game Finished!");

}

@Override

**void** initialize() {

System.***out***.println("Football Game Initialized! Start playing.");

}

@Override

**void** startPlay() {

System.***out***.println("Football Game Started. Enjoy the game!");

}

}

## Step 3

Sử dụng phương thức template của *Game* play() để thể hiện một cách xác định chơi trò chơi

**public** **class** TemplatePatternDemo {

**public** **static** **void** main(String[] args) {

Game game = **new** Cricket();

game.play();

System.***out***.println();

game = **new** Football();

game.play();

}

}