Name: Resmi Mariyil
UID: 179270

Pattern Design

A design patterns are well-proved solution for solving the specific problem/task. Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Basically, design patterns are categorized into two parts:

Core Java (or JSE) Design Patterns and JEE Design Patterns.

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1. Creational Design Pattern
    - Factory Pattern
    - Abstract Factory Pattern
    - Singleton Pattern
    - Prototype Pattern
    - Builder Pattern.

2. Structural Design Pattern
    - Adapter Pattern
    - Bridge Pattern
    - Composite Pattern
    - Decorator Pattern
    - Facade Pattern
    - Flyweight Pattern
    - Proxy Pattern

3. Behavioral Design Pattern
    - Chain Of Responsibility Pattern
    - Command Pattern
    - Interpreter Pattern
    - Iterator Pattern

- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Visitor Pattern

Factory Method Pattern

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface. The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

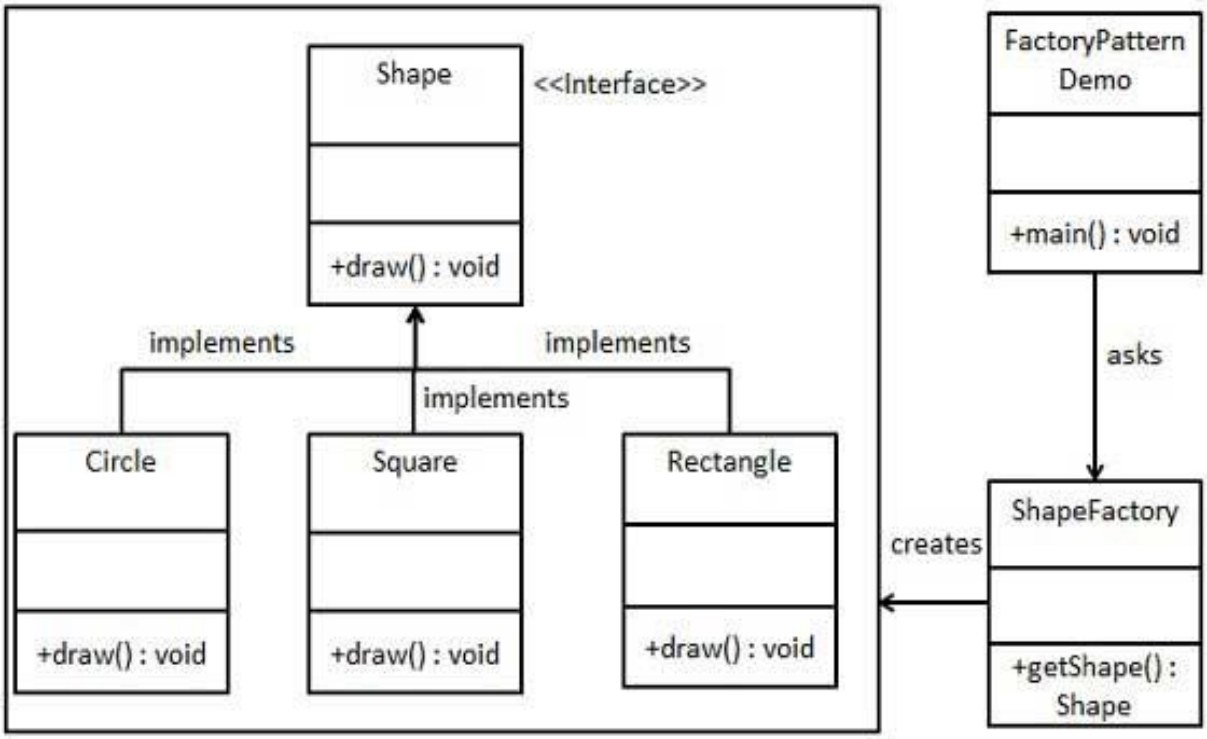
Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

Implementation

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.



Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
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

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
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();
   }
}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Flyweight Pattern

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

Advantage of Flyweight Pattern

- It reduces the number of objects.
- It reduces the amount of memory and storage devices required if the objects are persisted

Usage of Flyweight Pattern

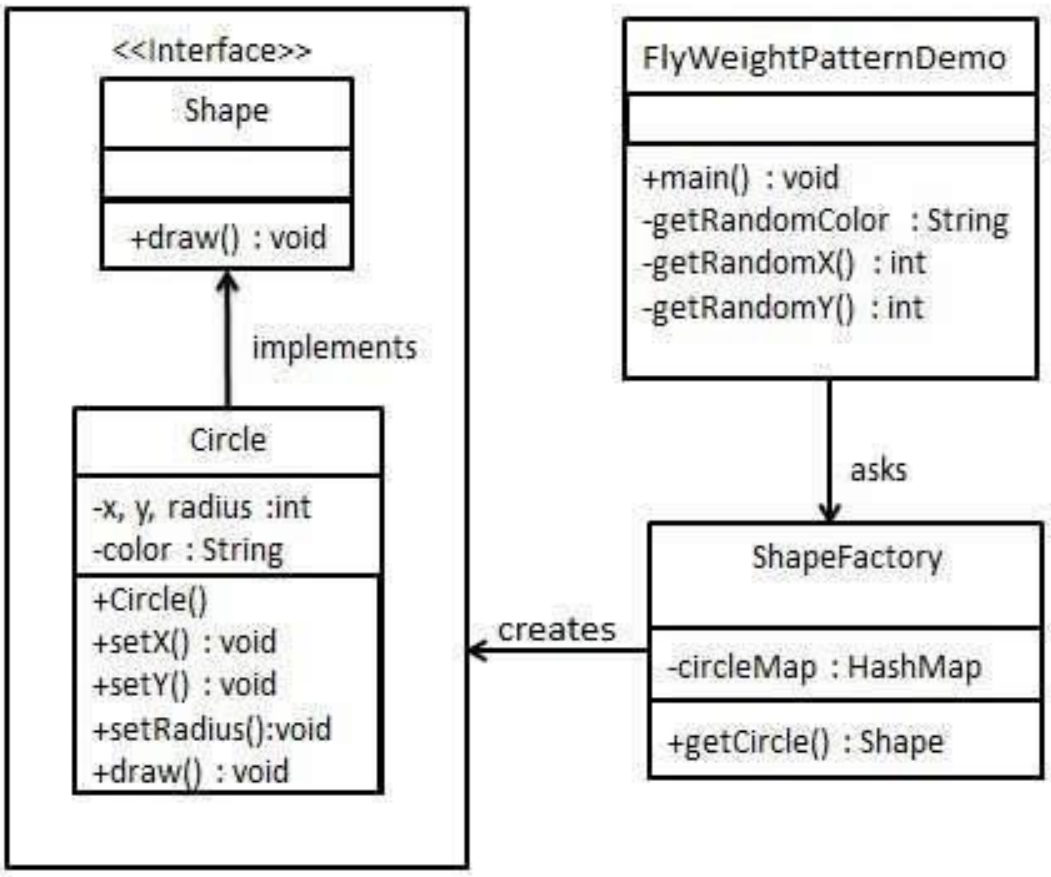
- When an application uses number of objects
- When the storage cost is high because of the quantity of objects.
- When the application does not depend on object identity.

Implementation

We are going to create a Shape interface and concrete class
Circle implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

ShapeFactory has a HashMap of Circle having key as color of the Circle object. Whenever a request comes to create a circle of particular color to ShapeFactory, it checks the circle object in its HashMap, if object of Circle found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

FlyWeightPatternDemo, our demo class, will use ShapeFactory to get a Shape object. It will pass information (red / green / blue/ black / white) to ShapeFactory to get the circle of desired color it needs.



Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete class implementing the same interface.

Circle.java

```
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;

   public Circle(String color){
      this.color = color;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public void setRadius(int radius) {
      this.radius = radius;
   }

   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
   }
}
```

Step 3

Create a factory to generate object of concrete class based on given information.

ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {

   // Uncomment the compiler directive line and
   // javac *.java will compile properly.
   // @SuppressWarnings("unchecked")
   private static final HashMap circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

Step 4

Use the factory to get object of concrete class by passing an information such as color.

FlyweightPatternDemo.java

```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

Step 5

Verify the output.

```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

Chain of Responsibility Pattern

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Advantage of Chain of Responsibility Pattern

- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Usage of Chain of Responsibility Pattern:

- When more than one object can handle a request and the handler is unknown.
- When the group of objects that can handle the request must be specified in dynamic way.

Observer Pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category. The observer pattern is also known as Dependents or Publish-Subscribe.

Benefits of Observer pattern:

- It describes the coupling between the objects and the observer.
- It provides the support for broadcast-type communication.

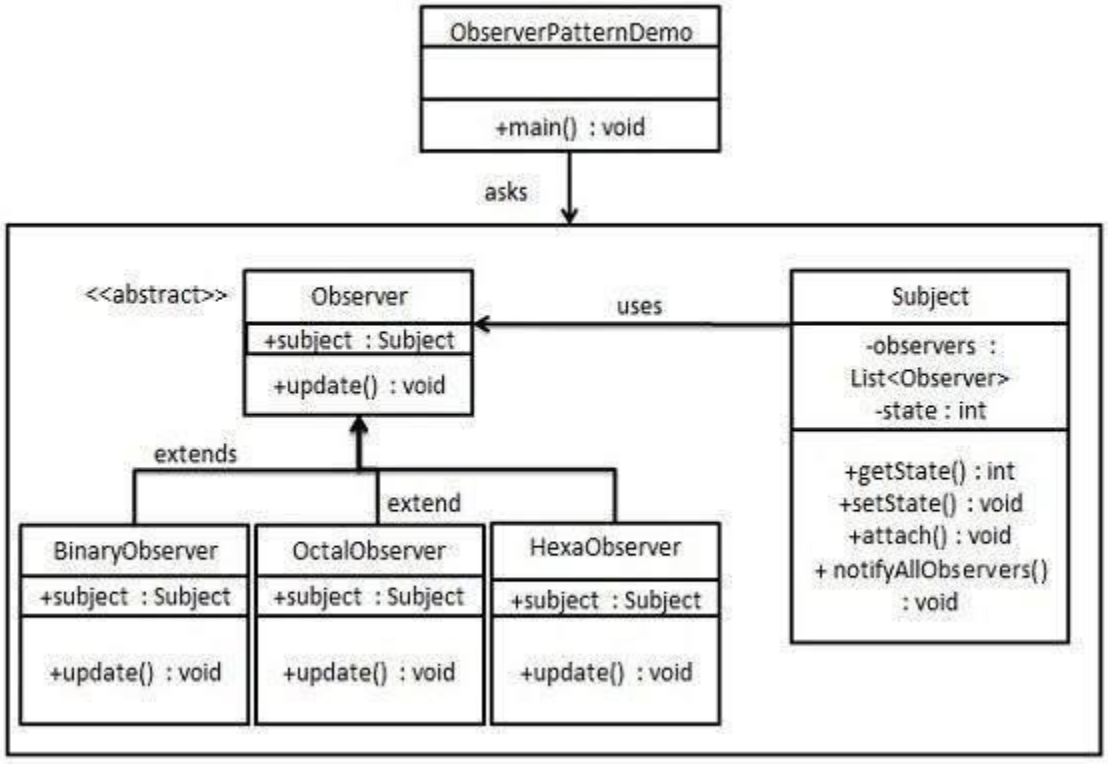
Usage of Observer pattern:

- When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class Observer and a concrete class Subject that is extending class Observer.

ObserverPatternDemo, our demo class, will use Subject and concrete class object to show observer pattern in action.



Step 1

Create Subject class.

Subject.java

```java
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

Create Observer class.

Observer.java

```java
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

Step 3

Create concrete observer classes

BinaryObserver.java

```
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() ) );
   }
}
```

OctalObserver.java

```
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() ) );
   }
```

HexaObserver.java

```
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

Use Subject and concrete observer objects.

ObserverPatternDemo.java

```
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);
   }
}
```

Step 5

Verify the output.

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```