OBJECT ORIENTED SOFTWARE DEVELOPMENT

LAB II

Title : Implementation of Creation Design Patterns

By

Arjun Prasad Adhikari

Sixth Semster

Roll No. 6

Gandaki College of Engineering and Science

Dec 11 2020

Contents

List of Figures

1 Singleton Pattern

Singleton pattern is one of the simplest 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. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.

1.1 Class Diagram

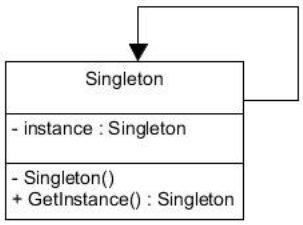

Figure 1: Class Diagram of Singleton Pattern

1.2 Source Code (Java)

1.2.1 OS Class as Singleton

```java
package singleton;

/* OS as singleton class. */

class OS {
    /* 1. Create a static object. */
    static OS obj = new OS();
    int value = 10;


    /* 2. The user shouldn't be able to
    create instance through constructor. */
    private OS() {
    }

    /* 3. Static method to return object
    which also return static object. */
    public static OS getInstance() {
        return obj;
    }

    /* 4. (Optional) Create a method to
    alter the value(value) so that we can
    observer the instance being shared
    among every objects.*/
    public void setValue(int value){
        this.value = value;
    }
}
```

1.2.2 Driver Class

```java
package singleton;

/* Class to demonstrate singleton behaviour of OS class. */

public class Singleton {
    public static void main(String[] args) {
        OS obj1 = OS.getInstance();
        OS obj2 = OS.getInstance();

        System.out.println(obj1.value);
        System.out.println(obj2.value);
```

```
        /* Changing the value property of obj1 object
        will also change the value property of obj2
        object. It happens because the instance is shared
        between every objects.
         */
        obj1.setValue(20);

        System.out.println(obj1.value);
        System.out.println(obj2.value);
    }
}
```

1.3 Output

```
10
10

20
20
```

2 Factory 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.

2.1 Class Diagram

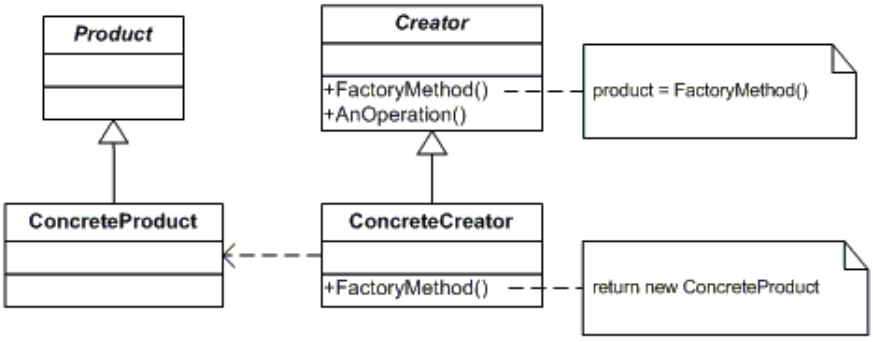

Figure 2: Class Diagram of Factory Pattern

2.2 Source Code (Java)

2.2.1 OS Interface

```
package factory;

/* OS interface. */

interface OS {
    void spec();
}
```

2.2.2 Android Class

```
package factory;

package factory;

/* Android class than implements OS interface. */

class Android implements OS {
    public void spec(){
        System.out.println("Most Powerful OS");
    }
}
```

2.2.3 IOS Class

```
package factory;

/* IOS class than implements OS interface. */

class IOS implements OS {
    public void spec(){
        System.out.println("Most Secure OS");
    }
}
```

2.2.4 Windows Class

```
package factory;

/* Windows class than implements OS interface. */

class Windows implements OS {
    public void spec(){
        System.out.println("Desktop OS");
```

```
    }
}
```

2.2.5 Factory Class - OSFactory

```
package factory;

class OSFactory {
    public static OS getInstance(String type){
        if (type.equals("ios")){
            return new IOS();
        } else if (type.equals("android")){
            return new Android();
        } else if (type.equals("windows")){
            return new Windows();
        } else {
            return null;
        }
    }
}
```

2.2.6 Driver Class

```
package factory;

package factory;

/* Driver code for Factory design pattern. */

import java.util.Scanner;

class FactoryDriver {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter operating system : ");
        String osType = sc.next();

        OS obj = OSFactory.getInstance(osType);
        if(obj != null){
            obj.spec();
        } else {
            System.out.println("Can't create object of given type.");
        }
        sc.close();
    }
}
```

2.3 Output

```
Enter operating system : android
Most Powerful OS

Enter operating system : ios
Most Secure OS

Enter operating system : windows
Desktop OS

Enter operating system : other
Can't create object of given type.
```

3 Abstract Factory Pattern

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

3.1 Class Diagram

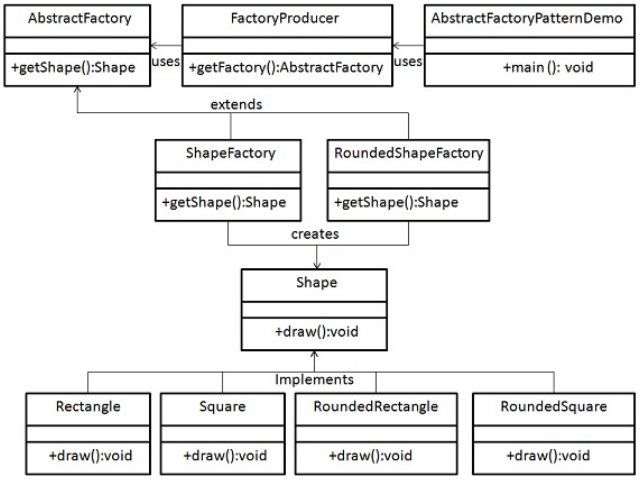

Figure 3: Class Diagram of Abstract Factory Pattern

3.2 Source Code (Java)

3.2.1 Shape Interface

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

3.2.2 RoundedSquare Class

```
public class RoundedSquare implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedSquare::draw() method.");
   }
}
```

3.2.3 RoundedRectangle Class

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}
```

3.2.4 Rectangle Class

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

3.2.5 AbstractFactory abstract Class

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

3.2.6 ShapeFactory Class

```
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;
    }
}
```

3.2.7 RoundedShapeFactory Class

```
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;
    }
}
```

3.2.8 FactoryProducer Class

```
public class FactoryProducer {
    public static AbstractFactory getFactory(boolean rounded){
        if(rounded){
            return new RoundedShapeFactory();
        }else{
            return new ShapeFactory();
        }
    }
}
```

3.2.9 Driver Class

```
public class AbstractFactoryDriver {
    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();

   }
}
```

3.3 Output

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

4 Prototype Pattern

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design provides one of the best ways to create an object. This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

4.1 Class Diagram

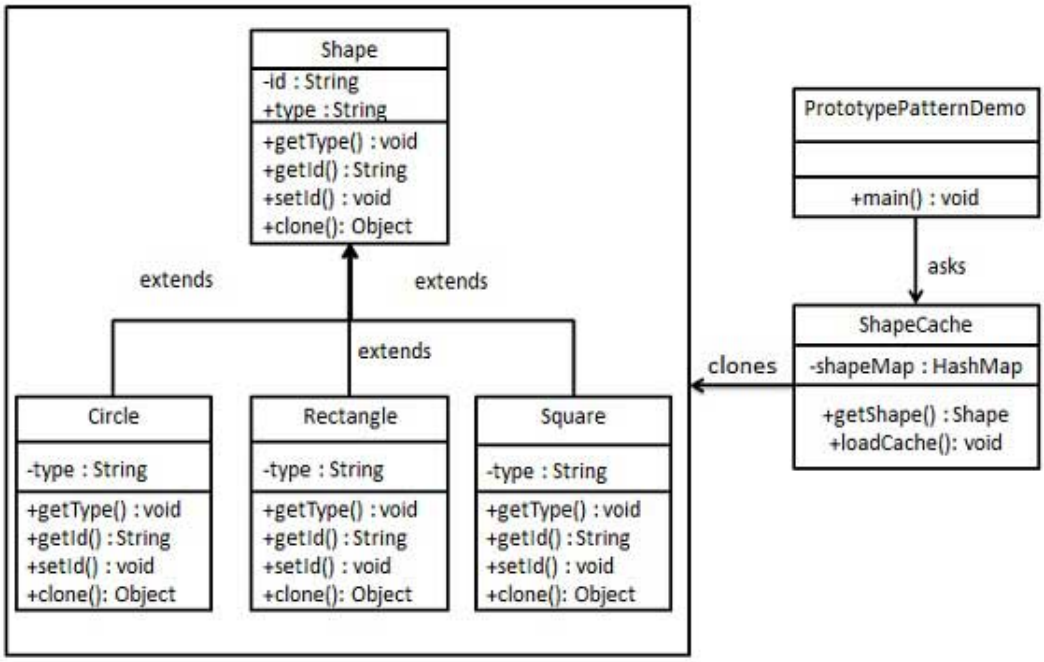

Figure 4: Class Diagram of Prototype Pattern

4.2 Source Code (Java)

4.2.1 Shape Class

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

4.2.2 Rectangle Class

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
```

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

4.2.3 Square Class

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

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

4.2.4 Circle Class

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

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

4.2.5 ShapeCache Class

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
```

```
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

4.2.6 Driver Class

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

4.3 Output

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

5 Builder Pattern

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. A Builder class builds the final object step by step. This builder is independent of other objects.

5.1 Class Diagram

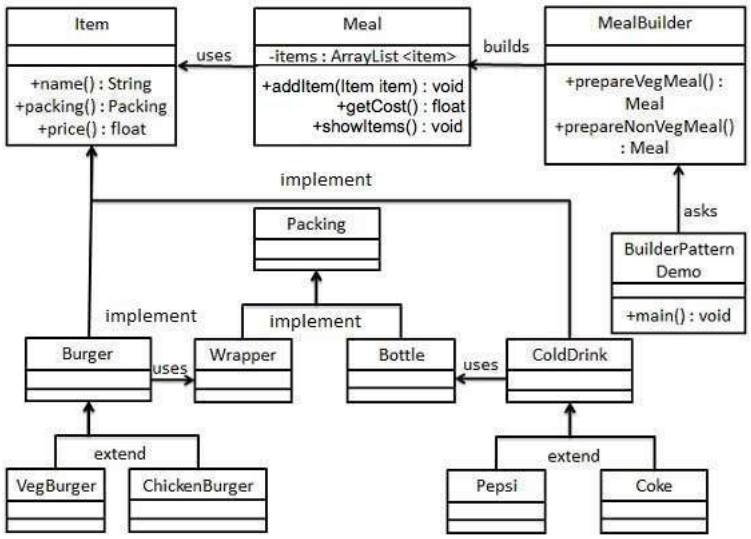

Figure 5: Class Diagram of Builder Pattern

5.2 Source Code (Java)

5.2.1 Item Interface

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

5.2.2 Packing Interface

```
public interface Packing {
   public String pack();
}
```

5.2.3 Wrapper Class

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

5.2.4 Bottle Class

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

5.2.5 Burger abstract Class

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

5.2.6 ColdDrink abstract Class

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

5.2.7 VegBurger Class

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

5.2.8 ChickenBurger Class

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

5.2.9 Coke Class

```
public class Coke extends ColdDrink {
```

```
    @Override
    public float price() {
        return 30.0f;
    }

    @Override
    public String name() {
        return "Coke";
    }
}
```

5.2.10 Pepsi Class

```
public class Pepsi extends ColdDrink {

    @Override
    public float price() {
        return 35.0f;
    }

    @Override
    public String name() {
        return "Pepsi";
    }
}
```

5.2.11 Meal Class

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item){
        items.add(item);
    }

    public float getCost(){
        float cost = 0.0f;

        for (Item item : items) {
            cost += item.price();
        }
        return cost;
    }
```

```
   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

5.2.12 MealBuilder Class

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

5.2.13 Driver Class

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

5.3 Output

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

6 TypeSafe Enum Pattern

The enums are type-safe means that an enum has its own namespace, we can't assign any other value other than specified in enum constants. Additionally, an enum is a reference type, which means that it behaves more like a class or an interface. As a programmer, we can create methods and variables inside the enum declaration.

6.1 Class Diagram

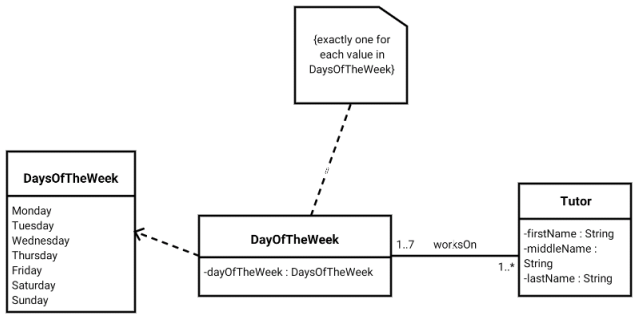

Figure 6: Class Diagram of TypeSafeEnum Pattern

6.2 Source Code (Java)

6.2.1 Suit Class

```
public class Suit {
    private final String name;
    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
    private Suit(String name) {
        this.name = name;
    }
    public String toString() {
        return name;
    }
}
```

6.2.2 Suit Enum

```
public enum Suit {
    CLUBS("clubs"), DIAMONDS("diamonds"), HEARTS("hearts"), SPADES("spades");
    private final String name;
    private Suit(String name) {
        this.name = name;
    }
}
```