Spring frameworkünün kullandığı design patternlar neler?

Spring framework tarafından kullanılan en önemli dört design patternler şu şekildedir:

1. MVC Pattern

MVC Design Pattern bir sistemin aşağıdaki bileşenlerini birbirinden ayırır:

Model: Model, basitçe uygulamanın genel durumu ya da uygulamayı oluşturan bileşenlerin genel durumu hakkındaki bilgileri içerir.

View: Modelin yorumu ve görsel temsilidir.

Control: Control, aşağıdaki örnek de görüleceği gibi hem Model de View üzerinde çalışır. Genel olarak Model'e gelen veriyi kontrol eder ve bu veriye göre View'i günceller.

Şimdi öncelikle Model olarak bir Student sınıfı oluşturalım.

```
public class Student {
   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Ardından, View kısmında, StrundetView, yalnızca öğrenci bilgilerini ekrana yazdırsın.

```
public class StudentView {
  public void printStudentDetails(String studentName, String studentRollNo){
    System.out.println("Student: ");
    System.out.println("Name: " + studentName);
    System.out.println("Roll No: " + studentRollNo);
  }
}
```

Son olarak da Controller oluşturalım. Beklentimiz, Controller'ın Model'den datayı alıp View'i güncelleyerek, View'in ekrana öğrencileri yazdırmasını sağlamaktır.

```
public class StudentController {
  private Student model;
  private StudentView view;

  public StudentController(Student model, StudentView view){
    this.model = model;
    this.view = view;
  }

  public void setStudentName(String name){
    model.setName(name);
  }

  public String getStudentName(){
    return model.getName();
  }

  public void setStudentRollNo(String rollNo){
    model.setRollNo(rollNo);
  }

  public String getStudentRollNo(){
    return model.getRollNo();
  }

  public void updateView(){
    view.printStudentDetails(model.getName(), model.getRollNo());
  }
}
```

Gördüğümüz üzere, Controller parametre olarak Student ve StudentView objeleri alıyor ve daha sonra set metotlarında Model'den gelen objeleri alıp get metotlarıyla View'i güncelliyor. Sonuç olarak,

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

      //fetch student record based on his roll no from the database
      Student model  = retriveStudentFromDatabase();

      //Create a view : to write student details on console
      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);

      controller.updateView();

      //update model data
      controller.setStudentName("John");

      controller.updateView();
   }

   private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
}
```

Öncelikle retriveStudentFromDatabase() ile bir öğrenci objesi oluşturup control.updateView() metodu ile View'i güncelledi. Ardından sadece Controller objesi ile yalnızca isim değerini Model'den alıp View'i bir kez daha güncelledi. Çıktı şu şekildedir:

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

2. Factory Pattern

Factory Pattern ile obje, yaratılma süreci istemciye gösterilmeden oluşturulur ve ortak interface ile yeni yaratılan objeyi kullanılır. Örnek olarak draw() metoduna sahip, Shape adında bir interface yazalım.

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

Daha sonra, Shape interface'ini implemente eden üç tane concrete sınıf oluşturalım.

```
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.");
   }
}
```

Şimdi de, belli şartlara göre concrete sınıflardan obje üretecek sınıfı yazalım.

```
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;
  }
}
```

Son olarak da uygulayalım.

```
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();
  }
}
```

Burada bir shapeFactory objesi oluşturuldu ve ShapeFactory sınıfındaki getShape metoduyla ayrı ayrı üç tane obje yaratıldı. Bu objeleri yaratırken ayrı ayrı kare, dikdörtgen ya da çember sınıfları açık şekilde kullanıldı mı? Hayır. shapeFactory sınıfı bu işi halletti. Çıktı olarak gelen sonuç,

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

3. Singleton Pattern

Singleton Pattern bir obje yaratmaktan sorumlu olan tek bir sınıf içerir ve yalnızca tek bir obje yaratıldığını garantiye alır. Ayrıca bahsi geçen sınıf, tek objesine global bir erişim sağlar öyle ki objeyi initialize etmeden bu erişim gerçekleştirilebilir. Örnek olarak, öncelikle bir SingleObject sınıfı yazalım.

```
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!");
   }
}
```

Burada obje initialize edilemesin diye constructor private olarak yazılır. SingleObject sınıfı için bir obje yaratılır ve daha sonra bu obje dış dünyaya döndürülür.

Şimdi bunu uygularsak,

```
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();
   }
}
```

Burada standart yöntemle obje yaratmaya çalışırsak hata alırız çünkü şu an constructor private durumda. Dolayısıyla objemizi, SingleObject sınıfının instance objesini kullanarak elde ediyoruz. Burada “static” keyword’ü sayesinde bunlar sınıfa ait olduğundan,

sınıf ismiyle getInstance() metoduna ve instance objesine ulaşabiliriz. Dolayısıyla main metodun içinde, getInstance() metodunu çağırarak instance objesine, bunu "object" objesine atayarak ulaştığımızda ve showMessage() metodunu çağırdığımızda şu çıktıyı elde edebiliriz,

Hello World!

4. Template Pattern

Template Pattern'de, abstract bir sınıf metotlarının çalıştırılabilmesi ve çalıştırılma şekli için tanımlanmış durumları ya da template metotları gösterir. Burada, alt sınıflar metotları override edebilir ancak metotların çağrılma yöntemi ya da template metodu abstract sınıfta tanımlandığı gibi kalır.

Örnek olarak, Game adlı, içinde template metot barındıran bir abstract sınıf yazalım.

```
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();
  }
}
```

Burada, play() metodu template metottur ve "final" olarak tanımlanmıştır ki override edilemesin.

Şimdi de Game sınıfını extend eden ve metotlarını override eden Cricket ve Football adlı iki sınıf oluşturalım.

```
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 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!");
  }
}
```


Son olarak da Game sınıfının template metodunu kullanarak sonuca bakalım.

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

    Game game = new Cricket();
    game.play();
    System.out.println();
    game = new Football();
    game.play();
  }
}
```

Çıktı şu şekilde olur,

```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

Görüldüğü üzere, metotları override ettik ama objeden çağrıyı daha önceden belirlenmiş ve değiştirilemeyen template metottan yaptık.

5. Prototype Pattern

Protoype pattern, objenin klonlanmasına olanak sağlar. Örneğin, sıfırdan bir obje oluşturmak, zahmetli veritabanı işlemleri vb, sebeplerden ötürü zor olacaksa bu patterne başvurulabilir.

Örnek olarak, öncelikle bir abstract Shape sınıfı oluşturalım.

```
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;
    }
}
```

Şimdi de bu sınıfı extend eden üç tane concrete sınıf yazalım.

```
public class Rectangle extends Shape {

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

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

```
public class Square extends Shape {

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

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

```
public class Circle extends Shape {

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

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

Burada type, “protected” tanımlandığı için alt sınıflardan ona erişebildik.

Bir sonraki adımda, klonlanacak objelerin concrete sınıflarını bir Hashtable'da tutacak sınıfı yaratalım.

```
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);
   }
}
```

Son olarak da ShapeCache sınıfını kullanarak HashTable'daki objelerin klonlarını oluşturup ekrana yazdıralım.

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

Çıktı, beklendiği üzere şöyledir,

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

6. Proxy Pattern

Structural Design Patterns, bu yapıları esnek ve verimli tutarken objelerin ve sınıfların daha büyük yapılar halinde nasıl birleştirileceğini açıklar. Proxy Pattern de bir Structural Design Pattern'dir. Proxy Pattern'de, orijinal objeyi içeren bir obje yaratılır ve orijinal objenin işlevini dış dünyaya döndürmek amaçlanır.

Örnek olarak bir Image interface'i ve bu interface'i implement eden bir concrete sınıf yazalım.

```
public interface Image {
  void display();
}
```

```
public class RealImage implements Image {

  private String fileName;

  public RealImage(String fileName){
    this.fileName = fileName;
    loadFromDisk(fileName);
  }

  @Override
  public void display() {
    System.out.println("Displaying " + fileName);
  }

  private void loadFromDisk(String fileName){
    System.out.println("Loading " + fileName);
  }
}
```

Şimdi de orijinal objeyi içeren ve onun işlevini dış dünyaya veren sınıfı oluşturalım.

```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

Son olarak uygularsak,

```
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

Çıktı şu şekilde olur,

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

İlk çağrıda realImage objesi boş olduğundan yeni bir obje oluşturdu ve RealImage sınıfının constructor'unda tanımlı olan loadFromDisk() özelliğini, ardından da display() metodunu çağırdı. İkinci çağrıda, realImage objesi artık boş olmadığından yeni bir obje oluşturmadı ve yalnızca display() metodunu çağırdı.