Spring Framework'te Tasarım Modelleri

Web uygulamaları oluşturmak için genelde Spring Framework kullanılıyor. Spring Framework'te kaç tasarım deseni olduğunu biliyor musunuz? Kodumuzun yazılım tasarım ilkelerine (SOLID, KISS, DRY, YAGNI) uymasını sağlayacak olan projemizde Design Pattern kullandığımızda kodumuzu okunması kolay, anlaşılması kolay, bakımı kolay yapar.

Tasarım Deseni Nedir?

Yazılım mühendisliğinde, bir yazılım tasarım deseni, yazılım tasarımında belirli bir bağlamda yaygın olarak ortaya çıkan bir soruna genel, yeniden kullanılabilir bir çözümdür. Doğrudan kaynak veya makine koduna dönüştürülebilecek bitmiş bir tasarım değildir. Bunun yerine, birçok farklı durumda kullanılabilecek bir sorunun nasıl çözüleceğine ilişkin bir açıklama veya şablondur. Tasarım kalıpları, programcının bir uygulama veya sistem tasarlarken yaygın sorunları çözmek için kullanabileceği resmileştirilmiş en iyi uygulamalardır. Tasarım Modeli kullanmak, kullandığınız dile bağlı değildir.

Proxy Tasarım Modeli

Spring, belirli bir hedef bean için proxy oluşturmak için JDK proxy'lerini (vekillenen hedef bile en az bir arabirim uyguladığında tercih edilir) veya CGLIB proxy'lerini (hedef nesne herhangi bir arabirim uygulamıyorsa) kullanır. Aksi şekilde yapılandırılmadıkça, Spring AOP çalışma zamanı log gerçekleştirir Farzedelim ki her metot girişini ve çıkışını günlüğe kaydetmek istiyoruz. Bu, her yöntemin başında ve sonunda günlük ifadeleri yazılarak sağlanabilir. Ancak bu, çok fazla kod çalışması gerektirecektir. Tüm yöntemler veya sınıflar arasında uygulanması gereken Güvenlik gibi çeşitli görevler vardır. Bunlar kesişen endişeler olarak bilinir. AOP, farklı yöntemlerde tekrarlanan ve normalde tamamen kendi modülüne, örneğin günlüğe kaydetme veya doğrulama gibi yeniden düzenlenemeyen herhangi bir kod türü olan kesişen endişeler sorununu ele alır.

Şimdi basit bir arayüz ve konfigürasyon oluşturalım.

```
public interface ExpensiveObject {
	void process();
}
```

Ve bu arayüzün büyük bir ilk konfigürasyonla uygulanması:

```
public class ExpensiveObjectImpl implements ExpensiveObject {
	public ExpensiveObjectImpl() {
		heavyInitialConfiguration();
	}
	@Override
	public void process() {
		LOG.info("processing complete.");
	}
	private void heavyInitialConfiguration() {
```

```
            LOG.info("Loading initial configuration...");
      }
}
```

Şimdi Proxy modelini kullanacağız ve istek üzerine nesnemizi başlatacağız:

```
public class ExpensiveObjectProxy implements ExpensiveObject {
      private static ExpensiveObject object;
      @Override public void process() {
            if (object == null) {
                  object = new ExpensiveObjectImpl();
            }
            object.process();
      }
}
```

Müşterimiz process() yöntemini çağırdığında, sadece işlemeyi görecek ve ilk yapılandırma her zaman gizli kalacaktır:

```
public static void main(String[] args) {
      ExpensiveObject object = new ExpensiveObjectProxy();
      object.process();
      object.process();
}
```

process() yöntemini iki kez çağırdığımızı unutmayın. Sahne arkasında, ayarlar bölümü yalnızca bir kez gerçekleşir – nesne ilk başlatıldığında. Sonraki her arama için, bu model ilk yapılandırmayı atlar ve yalnızca işleme gerçekleşir:

```
Loading initial configuration...
processing complete.
processing complete.
```

Singleton Tasarım Modeli

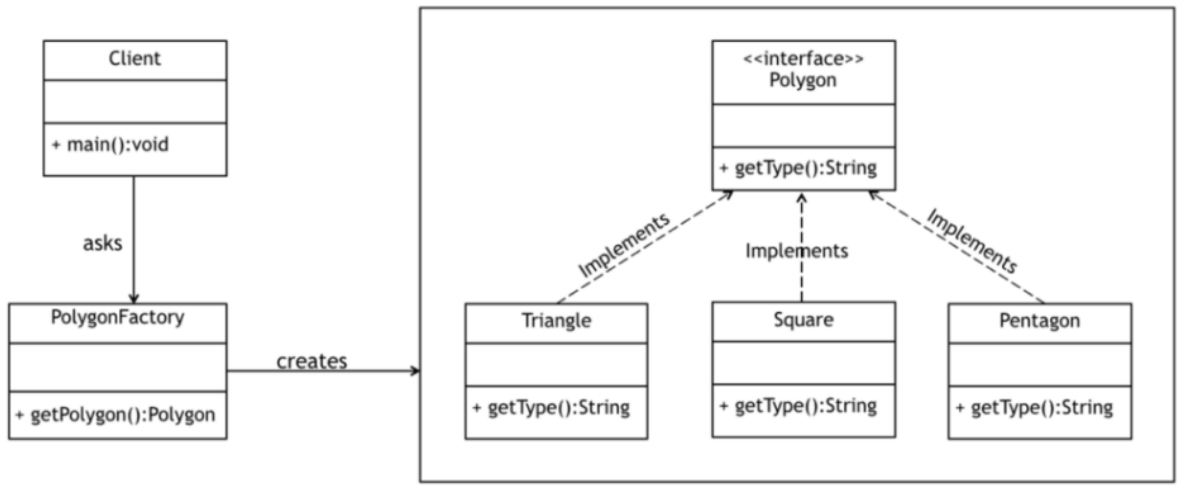
Spring config dosyalarında tanımlanan bean, varsayılan olarak tekildir. İlkbaharda bir singleton bean'i ve singleton deseni oldukça farklıdır. Singleton modeli, sınıf yükleyici başına belirli bir sınıfın yalnızca bir örneğinin oluşturulacağını söylüyor. Bir Spring singletonunun kapsamı "konteynır başına bean başına" olarak tanımlanır. Spring IoC konteyneri başına tek bir nesne örneğine yönelik bean tanımının kapsamıdır. Spring'deki varsayılan kapsam Singleton'dur.

```
public final class ClassSingleton {
      private static ClassSingleton INSTANCE;
```

```
	private String info = "Initial info class";
	private ClassSingleton() { }
	public static ClassSingleton getInstance() {
		if(INSTANCE == null) {
			INSTANCE = new ClassSingleton();
		}
		return INSTANCE;
	}
}
```

Factory Tasarım Modeli

Bu model, BeanFactory ve ApplicationContext kullanarak bean yüklemek için Spring tarafından kullanılır. BeanFactory ve ApplicationContent arasındaki fark nedir? Kısacası, BeanFactory, yapılandırma çerçevesi ve temel işlevsellik sağlar ve ApplicationContext daha fazla kuruluşa özel işlevsellik ekler. ApplicationContext, bir BeanFactory'nin tüm işlevlerini içerir. Bu nedenle, iyi bir nedeniniz olmadıkça bir ApplicationContext kullanmalısınız.



Önce Polygon arayüzünü oluşturalım:

```
public interface Polygon {
	String getType();
}
```

Daha sonra, bu arabirimi uygulayan ve Çokgen türünde bir nesne döndüren Kare, Üçgen vb. gibi birkaç uygulama oluşturacağız. Artık taraf sayısını argüman olarak alan ve bu arayüzün uygun uygulamasını döndüren bir fabrika oluşturabiliriz:

```
public class PolygonFactory {
```

```
	public Polygon getPolygon(int numberOfSides) {
		if(numberOfSides == 3) {
			return new Triangle();
		}
		if(numberOfSides == 4) {
			return new Square();
		}
		if(numberOfSides == 5) {
			return new Pentagon();
		}
		if(numberOfSides == 7) {
			return new Heptagon();
		} else if(numberOfSides == 8) {
			return new Octagon();
		} return null;
	}
}
```

Decorator Tasarım Modeli

Java kitaplıklarında dekoratör, özellikle akışlarla ilgili kodlarda oldukça standarttır. Bazı örnekler, java.io.InputStream, OutputStream, Reader, Writer ve java.util.Collections'ın tüm alt sınıfları (checkedX(), synchronizedX(), unmodifiedX() yöntemi), … Kısacası, Dekoratör Kalıbı "Nesnelere dinamik olarak sorumluluklar ekleyin" ilkesine dayanır.

İlk olarak, bir ChristmasTree arayüzü ve onun uygulamasını oluşturacağız:

```
public interface ChristmasTree {
	String decorate();
}
```

Bu arayüzün uygulanması şöyle görünecektir:

```
public class ChristmasTreeImpl implements ChristmasTree {
	@Override public String decorate() {
```

```
            return "Christmas tree";
        }
}
```

Şimdi bu ağaç için soyut bir TreeDecorator sınıfı oluşturacağız. Bu dekoratör, ChristmasTree arabirimini uygulayacak ve aynı nesneyi tutacaktır. Aynı arabirimden uygulanan yöntem, arabirimimizden yalnızca decor() yöntemini çağırır:

```
public abstract class TreeDecorator implements ChristmasTree {

    private ChristmasTree tree;

    // standard constructors

    @Override

    public String decorate() {

        return tree.decorate();

    }

}
```

Şimdi bir dekorasyon öğesi oluşturacağız. Bu dekoratörler, soyut TreeDecorator sınıfımızı genişletecek ve bizim gereksinimimize göre decor() yöntemini değiştirecek:

```
public class BubbleLights extends TreeDecorator {

    public BubbleLights(ChristmasTree tree) {

        super(tree);

    }

    public String decorate() {

        return super.decorate() + decorateWithBubbleLights();

    }

    private String decorateWithBubbleLights() {

        return " with Bubble Lights";

    }

}
```

Bu durum için aşağıdakiler doğrudur:

```
@Test
```

```
public void whenDecoratorsInjectedAtRuntime_thenConfigSuccess() {

	ChristmasTree tree1 = new Garland(new ChristmasTreeImpl());

	assertEquals(tree1.decorate(), "Christmas tree with Garland");

	ChristmasTree tree2 = new BubbleLights( new Garland(new Garland(new
	ChristmasTreeImpl())));

	assertEquals(tree2.decorate(), "Christmas tree with Garland with Garland
	with Bubble Lights");

}
```

İlk tree1 nesnesini yalnızca bir Çelenk ile süslediğimizi, diğer ağaç2 nesnesini ise bir BubbleLights ve iki Çelenk ile süslediğimizi unutmayın. Bu model bize çalışma zamanında istediğimiz kadar dekoratör ekleme esnekliği sağlar.

Strategy Tasarım Modeli

Strateji kalıbı, sınıfın kendisini değiştirmeden çalışma zamanında dahili algoritmayı değiştirerek bir sınıfın davranışını değiştirmektir. Spring ile Strateji Modelinin kullanımı hakkında çok fazla tartışma var. Genellikle, Springs IoC kapsayıcısının hangi stratejiyi kullanacağına karar verdiği Bağımlılık Enjeksiyonu ile birlikte kullanılan Strateji Modelini görürsünüz. Harika bir örnek olarak farklı veri kaynakları. Yerel kalkınma için bir H2 veri kaynağı kullanmak bir stratejidir. MySQL'i üretim için kullanmak başka bir stratejidir. Hangisinin çalışma zamanında kullanılacağı Spring IoC kapsayıcısına bağlıdır.

Noel, Paskalya veya Yeni Yıl olmasına bağlı olarak bir satın alma işlemine farklı türde indirimler uygulama zorunluluğumuz olduğunu varsayalım. İlk olarak, stratejilerimizin her biri tarafından uygulanacak bir İndirimci arayüzü oluşturalım:

```
public interface Discounter {

	BigDecimal applyDiscount(BigDecimal amount);

}
```

O zaman diyelim ki Paskalya'da %50, Noel'de %10 indirim uygulamak istiyoruz. Arayüzümüzü bu stratejilerin her biri için uygulayalım:

```
public static class EasterDiscounter implements Discounter {

	@Override

	public BigDecimal applyDiscount(final BigDecimal amount) {

		return amount.multiply(BigDecimal.valueOf(0.5));

		}

	}
```

```
public static class ChristmasDiscounter implements Discounter {

	@Override

	public BigDecimal applyDiscount(final BigDecimal amount) {

		return amount.multiply(BigDecimal.valueOf(0.9));

	}

}
```

Son olarak, bir testte bir strateji deneyelim:

```
Discounter easterDiscounter = new EasterDiscounter();

BigDecimal discountedValue = easterDiscounter
.applyDiscount(BigDecimal.valueOf(100));

assertThat(discountedValue) .isEqualByComparingTo(BigDecimal.valueOf(50));
```

Bu oldukça iyi çalışıyor, ancak sorun şu ki, her strateji için somut bir sınıf oluşturmak zorunda kalmak biraz acı verici olabilir. Alternatif, anonim iç türleri kullanmak olabilir, ancak bu yine de oldukça ayrıntılı ve önceki çözümden çok daha kullanışlı değil:

```
Discounter easterDiscounter = new Discounter() {

	@Override

	public BigDecimal applyDiscount(final BigDecimal amount) {

		return amount.multiply(BigDecimal.valueOf(0.5));

	}

};
```