Yaratıcı Tasarım Desenleri

Yazılım mühendisliğinde bir Tasarım Modeli, yazılım tasarımında en sık karşılaşılan sorunlara yerleşik bir çözümü tanımlar. Deneyimli yazılım geliştiricileri tarafından uzun bir süre boyunca deneme yanılma yoluyla geliştirilen en iyi uygulamaları temsil eder. Design Patterns, Design Patterns: Elements of Reusable Object-Oriented Software kitabının 1994 yılında Erich Gamma, John Vlissides, Ralph Johnson ve Richard Helm (Gang of Four veya GoF olarak da bilinir) tarafından yayınlanmasından sonra popülerlik kazandı.

Yaratıcı Tasarım Kalıpları, nesnelerin oluşturulma şekliyle ilgilidir. Nesneleri kontrollü bir şekilde oluşturarak karmaşıklıkları ve kararsızlığı azaltırlar. Yeni operatör, nesneleri uygulamanın her yerine dağıttığı için genellikle zararlı olarak kabul edilir. Sınıflar birbirine sıkı sıkıya bağlı hale geldiğinden, zamanla bir uygulamayı değiştirmek zorlaşabilir. Yaratıcı Tasarım Modelleri, istemciyi gerçek başlatma sürecinden tamamen ayırarak bu sorunu ele alır. Bu makalede, dört tür Yaratıcı Tasarım Modelini tartışacağız:

1. Singleton – Uygulama boyunca bir nesnenin en fazla yalnızca bir örneğinin var olmasını sağlar
2. Factory Method – Oluşturulacak tam nesneyi belirtmeden ilgili birkaç sınıfın nesnelerini oluşturur
3. Abstract Factory – İlgili bağımlı nesnelerin ailelerini oluşturur
4. Builder – Adım adım yaklaşımı kullanarak karmaşık nesneler oluşturur

1. Singleton

Singleton Tasarım Modeli, Java Sanal Makinesi boyunca nesnenin yalnızca bir örneğinin var olmasını sağlayarak belirli bir sınıfa ait nesnelerin başlatılmasını kontrol etmeyi amaçlar. Bir Singleton sınıfı ayrıca, erişim noktasına yapılan sonraki her çağrının yalnızca o belirli nesneyi döndürmesi için nesneye benzersiz bir genel erişim noktası sağlar.

Singleton modeli GoF tarafından tanıtılmasına rağmen, orijinal uygulamanın çok iş parçacıklı senaryolarda sorunlu olduğu bilinmektedir. Yani burada, statik bir iç sınıftan yararlanan daha optimal bir yaklaşım izleyeceğiz:

```
public class Singleton  {
        private Singleton() {}
        private static class SingletonHolder  {
                public static final Singleton instance = new Singleton();
        }
        public static Singleton getInstance() {
                return SingletonHolder.instance;
        }
}
```

Burada, Singleton sınıfının örneğini tutan statik bir iç sınıf yarattık. Örneği, dış sınıf yüklendiğinde değil, yalnızca getInstance() yöntemini çağırdığında oluşturur. Bu, bir Singleton sınıfı için yaygın olarak kullanılan bir yaklaşımdır çünkü senkronizasyon gerektirmez, iş parçacığı için güvenlidir, tembel başlatmayı zorlar ve nispeten daha az ortak plakaya sahiptir. Ayrıca, yapıcının özel erişim değiştiricisine sahip olduğunu unutmayın. Bu, bir Singleton

oluşturmak için bir gerekliliktir, çünkü bir genel kurucu, herkesin ona erişebileceği ve yeni örnekler oluşturmaya başlayabileceği anlamına gelir.

2. Factory Method

Fabrika Tasarım Kalıbı veya Fabrika Yöntemi Tasarım Kalıbı, Java'da en çok kullanılan tasarım kalıplarından biridir. GoF'a göre, bu model "bir nesne oluşturmak için bir arabirim tanımlar, ancak alt sınıfların hangi sınıfın başlatılacağına karar vermesine izin verir. Fabrika yöntemi, bir sınıfın somutlaştırmayı alt sınıflara ertelemesine izin verir". Bu model, bir tür sanal kurucu oluşturarak istemciden belirli bir fabrika sınıfına bir sınıfı başlatma sorumluluğunu devreder. Bunu başarmak için, bize nesneleri sağlayan ve gerçek uygulama ayrıntılarını gizleyen bir fabrikaya güveniyoruz. Oluşturulan nesnelere ortak bir arayüz kullanılarak erişilir.

Öncelikle bir abstract sınıf ya da interface oluşturarak ana bir metod veya değişkenleri tanımlıyoruz.

```
public abstract class Ilan {
	void print() {
	}
}
```

Sonrasında fabrika yöntemi ile üretmek istediğimiz sınıfları implement ya da kalıtarak yapmak istediğimiz işlemleri gerçekleştiriyoruz.

```
public class IlanDaire extends Ilan{
	@Override
	public void print() {
		System.out.println("IlanDaire from factory");
	}
}

public class IlanArsa extends Ilan{
	@Override
	public void print() {
		System.out.println("IlanArsa from factory");
	}
}
```

Daha sonra fabrika sınıfımızı ayarlayarak gerekli ayarlamaları yaparak verilen bilgi doğrultusunda sınıflarımızı oluşturuyoruz.

```
public class IlanFactory {
	public Ilan getType(String type) {
		if(type == null) {
			return null;
		}
		if(type.equals("Arsa")) {
			return new IlanArsa();
		}
		if(type.equals("Daire")) {
			return new IlanDaire();
		}
```

```
            return null;
        }
}
```

Main sınıfı içerisinde de gerçeklemeleri örneklendirerek aşağıdaki çıktıları elde ederiz.

```
public class main {

        public static void main(String[] args) {
                System.out.println("\n\n" + "Factory Design Pattern Usage Example");
                IlanFactory ilanFactory = new IlanFactory();
                Ilan ilan = ilanFactory.getType("Arsa");
                ilan.print();
                Ilan ilan2 = ilanFactory.getType("Daire");
                ilan2.print();
        }
}
```

Çıktı:
Factory Design Pattern Usage Example
IlanArsa from factory
IlanDaire from factory

3. Abstract Factory

Önceki bölümde, Fabrika Yöntemi tasarım deseninin tek bir aileyle ilgili nesneler oluşturmak için nasıl kullanılabileceğini gördük. Buna karşılık, Soyut Fabrika Tasarım Modeli, ilgili veya bağımlı nesnelerin ailelerini oluşturmak için kullanılır. Bazen fabrikaların fabrikası olarak da adlandırılır.

İlk önce Animal sınıfından bir aile oluşturacağız ve daha sonra bunu Soyut Fabrikamızda kullanacağız.

```
public interface Animal {
        String getAnimal();
        String makeSound();
}

public class Duck implements Animal {
        @Override
        public String getAnimal() {
                return "Duck";
        }

        @Override
        public String makeSound() {
                return "Squeks";
        }
}
```

Ayrıca Animal interface'in (Köpek, Ayı vb. gibi) daha somut uygulamalarını tam olarak bu şekilde oluşturabiliriz. Soyut Fabrika, bağımlı nesnelerin aileleriyle ilgilenir. Artık birden fazla aileyi hazır hale getirdiğimize göre, onlar için bir AbstractFactory arayüzü oluşturabiliriz:

```
public interface AbstractFactory<T> {
	T create(String animalType) ;
}
```

Ardından, önceki bölümde tartıştığımız Fabrika Yöntemi tasarım modelini kullanarak bir AnimalFactory uygulayacağız:

```
public class AnimalFactory implements AbstractFactory<Animal> {
	@Override
	public Animal create(String animalType) {
		if ("Dog".equalsIgnoreCase(animalType)) {
			return new Dog();
		} else if ("Duck".equalsIgnoreCase(animalType)) {
			return new Duck();
		}
		return null;
	}
}
```

Benzer şekilde, aynı tasarım desenini kullanarak Color arayüzü için bir fabrika uygulayabiliriz. Tüm bunlar ayarlandığında, getFactory() yöntemine sağladığımız argümana bağlı olarak bize AnimalFactory veya ColorFactory uygulamasını sağlayacak bir FactoryProvider sınıfı oluşturacağız:

```
public class FactoryProvider {
	public static AbstractFactory getFactory(String choice){
		if("Animal".equalsIgnoreCase(choice)){
			return new AnimalFactory();
		} else if("Color".equalsIgnoreCase(choice)){
			return new ColorFactory();
		}
		return null;
	}
}
```

4. Builder

Builder Tasarım Modeli, nispeten karmaşık nesnelerin yapımıyla başa çıkmak için tasarlanmış başka bir yaratıcı modeldir. Nesne oluşturmanın karmaşıklığı arttığında, Oluşturucu deseni, nesneyi oluşturmak için başka bir nesne (builder) kullanarak örnekleme sürecini ayırabilir. Bu oluşturucu daha sonra basit bir adım adım yaklaşım kullanarak benzer birçok başka temsili oluşturmak için kullanılabilir.

GoF tarafından sunulan orijinal Builder Design Pattern, soyutlamaya odaklanır ve karmaşık nesnelerle uğraşırken çok iyidir, ancak tasarım biraz karmaşıktır. Joshua Bloch, Etkili Java kitabında, oluşturucu modelinin temiz, yüksek oranda okunabilir (çünkü akıcı tasarım kullandığından) ve müşterinin bakış açısından kullanımı kolay olan geliştirilmiş bir sürümünü tanıttı. Bu örnekte, bu versiyonu tartışacağız. Bu örnekte, statik bir iç sınıf olarak bir oluşturucu içeren BankAccount adlı yalnızca bir sınıf vardır:

```
public class BankAccount {
	private String name;
	private String accountNumber;
	private String email;
```

```
        private boolean newsletter;
        // constructors/getters
        public static class BankAccountBuilder {
                // builder code
        }
}
```

Dış nesnelerin onlara doğrudan erişmesini istemediğimiz için, alanlardaki tüm erişim değiştiricilerinin özel olarak bildirildiğini unutmayın. Yapıcı ayrıca özeldir, böylece yalnızca bu sınıfa atanan Oluşturucu ona erişebilir. Yapıcıda ayarlanan tüm özellikler, argüman olarak sağladığımız oluşturucu nesnesinden çıkarılır. BankAccountBuilder'ı statik bir iç sınıfta tanımladık:

```
public static class BankAccountBuilder {
        private String name;
        private String accountNumber;
        private String email;
        private boolean newsletter;
        public BankAccountBuilder(String name, String accountNumber) {
                this.name = name;
                this.accountNumber = accountNumber;
        }

        public BankAccountBuilder withEmail(String email) {
                this.email = email;
                return this;
        }

        public BankAccountBuilder wantNewsletter(boolean newsletter) {
                this.newsletter = newsletter;
                return this;
        }

        public BankAccount build() {
                return new BankAccount(this);
        }
}
```

Dikkat edin, dış sınıfın içerdiği aynı alan kümesini ilan ettik. Herhangi bir zorunlu alan, iç sınıfın yapıcısı için bağımsız değişkenler olarak gereklidir, kalan isteğe bağlı alanlar ise ayarlayıcı yöntemleri kullanılarak belirtilebilir. Bu uygulama, ayarlayıcı yöntemlerin oluşturucu nesnesini döndürmesini sağlayarak akıcı tasarım yaklaşımını da destekler. Son olarak, build yöntemi dış sınıfın özel kurucusunu çağırır ve kendisini argüman olarak iletir. İade edilen BankAccount, BankAccountBuilder tarafından ayarlanan parametrelerle somutlaştırılacaktır. Oluşturucu modelinin hızlı bir örneğini çalışırken görelim:

```
BankAccount newAccount = new BankAccount
  .BankAccountBuilder("Jon", "22738022275")
  .withEmail("jon@example.com")
  .wantNewsletter(true)
  .build();
```