Creational Design Patterns

Yazılım mühendisliğinde bir Design Pattern, 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 .

Bu yazıda, yaratıcı tasarım kalıplarını ve türlerini keşfedeceğiz. Ayrıca bazı kod örneklerine bakacağız ve bu modellerin tasarımımıza uyduğu durumları tartışacağız.

Creational Design Patternler Nelerdir?

Creational Design Patternler, 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.

Creational Design Patternler, istemciyi gerçek başlatma sürecinden tamamen ayırarak bu sorunu ele alır.

Bu makalede, dört tür Creational Design Pattern'i 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

Şimdi bu kalıpların her birini ayrıntılı olarak tartışalım.

Singleton Design Pattern

Singleton Design Pattern, JVM boyunca nesnenin yalnızca bir örneğinin var olmasını sağlayarak belirli bir sınıftaki 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 Model Örneği

Singleton modeli GoF tarafından tanıtılmasına rağmen, orijinal uygulamanın multi threadli 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 . Örneğin , dış sınıf yüklendiğinde değil, yalnızca getInstance() yöntemini çağırdığında oluşturur.

Bu, bir Singleton sınıfı için yaygın olarak kullanılan bir yaklaşımdır çünkü senkronizasyon gerektirmez, thread safedir, lazy initializationı zorlar ve nispeten daha az basmakalıptır.

Ayrıca, constructor private access modifier'a sahip olduğunu unutmayın. Bu bir Singleton oluşturmak için bir gerekliliktir, çünkü bir public constructor, herkesin ona erişebileceği ve yeni örnekler oluşturmaya başlayabileceği anlamına gelir.

Singleton Design Patterni Ne Zaman Kullanılır?

- Oluşturulması pahalı olan kaynaklar için (veritabanı bağlantı nesneleri gibi)
- Performansı arttırmak için tüm loggerları Singleton olarak tutmak iyi bir uygulamadır.
- Uygulama için yapılandırma ayarlarına erişim sağlayan sınıflar
- Paylaşılan modda erişilen kaynakları içeren sınıflar

Factory Method Design Pattern

Factory Method Desing Pattern'i, Java'da en çok kullanılan Design Pattern'lerden biridir.

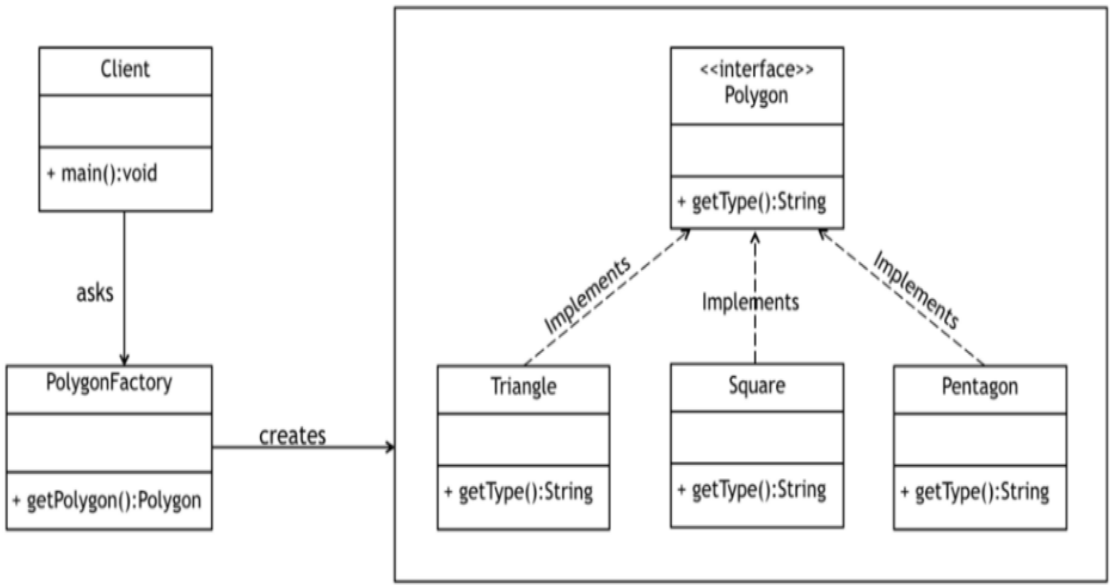
GoF'a göre, bu model "bir nesne oluşturmak için bir arabirim tanımlar, ancak alt sınıfların hangi sınıfın başlatılacağına karar vermesine izin verir. Factory Methodu, bir sınıfın somutlaştırmayı alt sınıflara ertelemesine izin verir".

Bu model, bir tür sanal constructor oluşturarak istemciden belirli bir factory sınıfına bir sınıfı başlatma sorumluluğunu devreder.

Bunu başarmak için, bize nesneleri sağlayan ve gerçek uygulama ayrıntılarını gizleyen bir factorye güveniyoruz. Oluşturulan nesnelere ortak bir interface kullanılarak erişilir.

Factory Method Design Pattern Model Örneği

Bu örnekte, birkaç somut sınıf tarafından uygulanacak bir Polygon interface oluşturacağız. Bu aileden nesneleri getirmek için bir PolygonFactory kullanılacaktır:



Önce Polygon interfaceini oluşturalım:

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

Artık kenar sayısını argüman olarak alan ve bu interfacein uygun implementasyonunu döndüren bir factory 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;
    }
}
```

İstemcinin, nesneyi doğrudan başlatmak zorunda kalmadan bize uygun bir Polygon vermesi için bu factorye nasıl güvenebileceğine dikkat edin.

Factory Method Design Patterni Ne Zaman Kullanılır?

- Bir interfacein implementasyonunun veya bir abstrac sınıfın sık sık değişmesi beklendiğinde
- Mevcut uygulama yeni değişikliği rahatça karşılayamadığında
- Başlatma işlemi nispeten basit olduğunda ve constructor yalnızca bir avuç parametre gerektirdiğinde

Abstract Factory Design Pattern

Abstract Factory Design Pattern, ilgili veya bağımlı nesnelerin ailelerini oluşturmak için kullanılır. Bazen factorylerin factorysi olarak da adlandırılır.

Builder Design Pattern

Builder Design Pattern, nispeten karmaşık nesnelerin yapımıyla başa çıkmak için tasarlanmış başka bir creational patterndir.

Nesne oluşturmanın karmaşıklığı arttığında, Builder Design Pattern, nesneyi oluşturmak için başka bir nesne (bir builder) kullanarak örnekleme sürecini ayırabilir.

Bu builder daha sonra basit bir adım adım yaklaşım kullanarak benzer birçok başka temsili oluşturmak için kullanılabilir.

Builder Design Pattern Model Örneği

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.

Constructor ayrıca özeldir, böylece yalnızca bu sınıfa atanan builder ona erişebilir. Constructor'da ayarlanan tüm özellikle, argüman olarak sağladığımız builder 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 oluşturduk. Herhangi bir zorunlu alan, iç sınıfın constructorı için bağımsız değişkenler olarak gereklidir, kalan isteğe bağlı alanlar ise ayarlayıcı yöntemleri kullanılarak belirtilebilir.

Bu uygulama, ayarlayıcı yöntemlerin builder 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 .

Builder modelinin hızlı bir örneğini çalışırken görelim:

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

Builder Design Patterni Ne Zaman Kullanılır?

- Bir nesne oluşturma süreci, çok sayıda zorunlu ve isteğe bağlı parametre ile son derece karmaşık olduğunda
- Builder parametrelerinin sayısındaki artış, çok sayıda constructor listesine yol açtığında
- Müşteri, oluşturulan nesne için farklı temsiller beklediğinde