Title

Creational Design Patterns

Creational Design Patterns, nesne (object) oluşturma yöntemine odaklanır ve bir sınıfın (class) somutlaştırılması (instantiation) sırasında bir karar vermemiz gerekliliğinde kullanılır. Bu desenler, nesneleri düzenli bir şekilde oluşturarak karmaşıklıkları azaltıp, netliği arttırmamızı sağlar.

6 tip Creational Design Patterns bulunmaktadır:

1. Factory Method Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern
6. Object Pool Pattern

Bunlardan Singleton, Factory Method, Abstract Factory ve Builder tasarım desenlerini inceliyor olacağız.
Hadi başlayalım :)

- SINGLETON -
Singleton Design Pattern; Java Sanal Makinesi (JVM) boyunca nesnenin yalnızca bir örneğinin (instance) var olmasını sağlayarak belirli bir sınıftaki nesnelerin (object) başlatılmasını kontrol etmeyi amaçlar.
Bir Singleton sınıfı ayrıca, erişim noktasına (access point) yapılan sonraki her çağrının yalnızca o belirli nesneyi döndürmesi (return) için nesneye benzersiz bir genel erişim noktası sağlar.

Örnek vermek gerekirse:

```
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 class'ının instance'ını tutan static bir inner-class yaratmk. Instance outer-class yüklendiğinde değil, yalnızca getInstance() method'u çağırıldığında oluşturuluyor.

Bu Singleton class için yaygın olarak kullanılan bir yaklaşımdır çünkü senkronizasyon gerektirmez, thread safe'dir ve lazy initialization'ı zorlar.

Ayrıca, constructor method'un private erişim belirleyiciye sahip olduğunu gözden kaçırmayın. Bu bir Singleton oluşturmak için gerekliliktir. Çünkü bir public constructor herkesin ona erişebileceği ve yeni örnekler oluşturmaya başlayabileceği anlamına gelir.

- FACTORY METHOD -
Fabrika Design Pattern'ı, Java'da en çok kullanılan tasarım desenlerinden biridir.

GoF'a (Gang of Four) göre, bu desen "bir nesne oluşturmak için bir interface tanımlar, ancak inner-classların hangi classın başlatılacağına karar vermesine izin verir.

Örnek vermek gerekirse:

Önce Polygon interface'ini oluşturalım:

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

Daha sonra, bu interface'i implemente eden ve Polygon türünde bir nesne return eden Square, Triangle vb. gibi birkaç implementasyon class'ı oluşturacağız.

Artık kenar sayısını parametre olarak alan ve sonucu girilen değere göre hesaplayıp return eden bir Factory oluşturabiliriz.

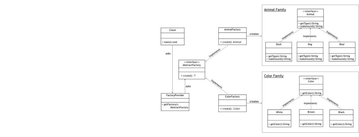
```
public class PolygonFactory {

    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;
    }
}
```

- ABSTRACT FACTORY -
Yukarıda Factory Design Pattern'ın tek bir aileyle ilgili nesneler oluşturmak için nasıl kullanılabileceğini gördük.

Buna karşılık, Abstract Factory Design Pattern; ilgili veya bağımlı nesnelerin ailelerini oluşturmak için kullanılır. Bazen fabrikaların fabrikası (factory of factories) olarak da adlandırılır.

Örnek vermek gerekirse:

İlk önce Animal interface'inden bir aile oluşturacağız ve daha sonra bunu Abstract Factory olarak kullanacağız.



Animal interface'i:

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

ve Duck implementasyonu:

```
public class Duck implements Animal {

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

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

Diğer implement edecek classları da aynı şekilde oluşturabiliriz. Bu kısmı size bırakıyorum.

Artık gerekli classları hazır hale getirdiğimizi düşünerek onlar için bir AbstractFactory interface'i oluşturabiliriz:

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

Daha sonra Factory Method Design Pattern kullanarak bir AnimalFactory class'ı implemente edeceğiz:

```
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;
    }
}
```

Aynı tasarım desenini kullanarak Color interface'i içinde bir Factory implemente edebiliriz.

Daha sonra, FactoryProvider class'ı oluşturacak ve getFactory() method'unu kullanarak girilen parametreye göre ColorFactory veya AnimalFactory instance'ı oluşturulmasını sağlayacağız:

```
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;
    }
}
```

- BUILDER -
Builder Design Pattern; nispeten karmaşık nesnelerin oluşturulmasıyla başa çıkmak için tasarlanmış bir Creational Design Pattern'dir.

Nesne oluşturmanın karmaşıklığı arttığında, Builder tasarım deseni, nesneyi oluşturmak için başka bir nesne (builder) kullanarak örneklerne sürecini ayrıştırabilir.

Örnek vermek gerekirse:

BankAccount class'ı aynı zamanda içerisinde bir static inner-class barındırır:

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

    // constructors/getters ...

    public static class BankAccountBuilder {
        // builder code ...
    }
}
```

Tüm property'ler için erişim belirleyicilerin private olduğuna dikkat edin çünkü dışarıdaki nesnelerin onlara ulaşmasını istemiyoruz.
Ayrıca constructor da private olmalıdır. Böylece sadece bu class'a atanan Builder ona erişebilecek.

BankAccountBuilder constructor'ını static inner-class içerisinde tanımlıyoruz:

```
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. Outer-class'ın içerdiği property'lerin aynılarını burada da set etti.

Sonuç olarak build() methodu outer-class'ın private constructor'ını çağırır ve kendisini parametre olarak iletir.

Return edilen BankAccount, BankAccountBuilder tarafından gelen parametrelere göre somutlaştırılacaktır:

```
BankAccount newAccount = new BankAccount
    .BankAccountBuilder("Jon", "22748022275")
    .withEmail("jon@example.com")
    .wantNewsletter(true)
    .build();
```

Umarım bu makale Java geliştiricilerine bir nebze de olsa katkı sağlamıştır.

Başka bir yazıda görüşmek üzere.

İstek ve önerileriniz için bana hatahankilsac@gmail.com adresinden ulaşabilirsiniz.