Fabrika Metod(Factory Method) Tasarım Deseni

Nedir?

Fabrika metod tasarım deseni, creational tasarım desenlerinden biridir. Bu tasarım deseni bir nesne yaratmak için arayüz sağlar, **fakat** hangi sınıftan nesne yaratılacağını, alt sınıfların belirlemesine olanak tanır.

Ne zaman Kullanılır?

Super sınıf ve alt sınıfların olduğu bir uygulamada, alt sınıfların yaratılma işlemini **client** yani istemci sınıfında yapılmasını **engellemek** için kullanılır.

Nasıl Kullanılır?

Üst sınıfta implement edilmemiş bir metod bulunacak. Bu metod alt sınıflar tarafından implement edilecek. Alt sınıfın yaratılacak nesneyi belirleme işlemi, implement edilmemiş bu metod sayesinde gerçekleştirilecektir.

Not: Üst sınıfta implement edilmemiş bir metod bulunacağı için bu sınıfın türü ya **abstract** tanımlanmalı ya da **interface** olmalıdır.

Faydaları Nedir?

1. Birbirine daha az bağımlı(loosely coupled) sınıflar oluşturmaya imkan tanıdığı ve Factory sınıfı ve onun alt sınıflarına nesne yaratma işlemi taşındığı için daha az karmaşık kod yazılır. Böyle bir kodun bakımı ise daha kolay olacaktır.
2. İstemci (Client) kod, sadece Product **interface** ile ilgilenir ve bu sayede somut(concrete) Product'lar, client kodu değiştirmeden rahatça eklenebilir.

Gerekenler

Türü **abstract** veya **interface** olan bir **süper(base)** fabrika sınıfı
En az bir tane **alt** fabrika sınıfı
En az bir tane Product(ürün) sınıfı
Test sınıfı

Örnek Kullanım Alanları

1. **java.util.Calendar#getInstance()**
2. **java.util.ResourceBundle#getBundle()**
3. **java.text.NumberFormat#getInstance()**
4. **java.nio.charset.Charset#forName()**

Örnek Uygulama

Base(Süper) Fabrika Sınıfı

```
1  /**
2   * AnimalStore
3   */
4  public abstract class AnimalStore {
5      protected Animal printProperties(String type) {
6          Animal animal = createAnimal(type);
7          String properties=type+" weight: "+animal.getWeight()+" height: "+animal.getHeight()+" age: "+animal.getAge();
8          System.out.println(properties);
9       return animal;
10     }
11
12     protected abstract Animal createAnimal(String type);
13
14 }
```

Dikkat edersek **createAnimal()** isimli bir metod, parametreye göre **Animal** türünden nesne üretecektir. Ayrıca **abstract** tanımlandığı için bu metodu alt sınıflar kendine göre implement etmeleri gerekmektedir. Bu metod **abstract** tanımlanmak **zorunda değildir**. Fakat normal kullanımı bu şekildedir. Ayrıca normal kullanımlarda, bu metodun visible tipi **protected** olmaktadır.

Not: **printProperties()** metodu içerisinde **createAnimal()** metodu kullanılmıştır. Fabrika metod tasarım deseni, bu şekilde kullanımı **zorunlu kılar**.

Alt Fabrika Sınıfları

```
1  /**
2   * BirdStore isimli alt fabrika sinifi
3   */
4  public class BirdStore extends AnimalStore {
5      @Override
6      protected Animal createAnimal(String type) {
7          if("Eagle".equalsIgnoreCase(type)){
8              return new Eagle();
9          }else if("Hawk".equalsIgnoreCase(type)){
10             return new Hawk();
11         }else{
12             throw new IllegalArgumentException();
```

```
        }
    }
}
```

```
/**
 * MammalStore isimli alt fabrika sinifi
 */
public class MammalStore extends AnimalStore {
    @Override
    public Animal createAnimal(String type) {
        if("Cow".equalsIgnoreCase(type)){
            return new Cow();
        }else if("Hourse".equalsIgnoreCase(type)){
            return new Hourse();
        }else{
            throw new IllegalArgumentException();
        }
    }
}
```

BirdStore ve **MammalStore** isimli iki tane alt sınıf yazıldı. Bu sınıflar **AnimalStore** isimli süper fabrika sınıfında bulunan **createAnimal()** metodunu kendileri implement etmektedir. Bu implementasyonun nasıl yapıldığından, üst sınıfın ve client sınıfın haberi olmamaktadır.

Abstract tanımlanmış Product Süper Sınıfı

```
/**
 * Animal isimli abstract tanimlanmis Product sinifi
 */
public abstract class Animal {
    private int age;
    private int weight;
    private int height;

    public int getAge() {
        return age;
    }

    public int getWeight() {
        return weight;
    }

    public int getHeight() {
        return height;
    }
}
```

Animal isimli sınıfımız **abstract** olarak tanımlandı. Bunun nedeni, bütün hayvanlarda ortak olan bazı özelliklerin olması, bazı özelliklerinin ise birbirinden farklı olmasıdır. Örneğin yaş, kilo, boy gibi özellikler tüm hayvanlar için ortak özellikler olmasına karşın, uçma özelliği sadece kuşlara özgüdür. **abstract** olarak tanımlanan bir sınıf, kendisinden türüyen sınıflara belirli özelliklerin aynen aktarılmasını, farklı özelliklerin ise bu sınıflar tarafından kendilerine özgü tanımlanmasına olanak tanır.

Eğer bu sınıf **somut** sınıf olsa idi, tüm özellikler aynen alt sınıflara aktarılacaktı. Bunun sonucunda ise uçma özelliğine sahip olmayan bir hayvan sınıfı içinde uçma metodu yer alacaktı.

Eğer bu sınıf **interface** olarak tanımlanmış olsa idi, her metodu alt sınıflar tekrar tekrar **implement** etmek zorunda kalacaklardı.

Not: abstract olarak tanımlanan sınıflar direkt olarak **new** ile yaratılamazlar. Bu sayede **loosely coupled** uygulamalar geliştirmek daha da kolaylaşır.

Alt Product Sınıfları(Mammal Grubuna Ait Olanlar)

```
/**
 * İnek sinifi
 */
public class Cow extends Animal {
    @Override
    public int getAge() {
        return 4;
    }

    @Override
    public int getHeight() {
        return 100;
    }

    @Override
    public int getWeight() {
        return 300;
    }
}
```

```
/**
 * At sinifi
 */
public class Hourse extends Animal {
```

```
    @Override
    public int getAge() {
        return 6;
    }

    @Override
    public int getHeight() {
        return 160;
    }

    @Override
    public int getWeight() {
        return 400;
    }
}
```

Alt Product Sınıfları(Bird Grubuna Ait Olanlar)

```
/**
 * Kartal sinifi
 */
public class Eagle extends Animal {
    @Override
    public int getAge() {
        return 50;
    }

    @Override
    public int getHeight() {
        return 100;
    }

    @Override
    public int getWeight() {
        return 5;
    }
}
```

```
/**
 * Sahin sinifi
 */
public class Hawk extends Animal {
    @Override
    public int getAge() {
        return 50;
    }

    @Override
    public int getHeight() {
        return 100;
    }

    @Override
    public int getWeight() {
        return 5;
    }
}
```

Test Sınıfı

```
/**
 * Test sinifi
 */
public class TestFactoryMethod {
    public static void main(String[] args) {
        AnimalStore mammalStore=new MammalStore();
        mammalStore.printProperties("cow");
        AnimalStore birdStore=new BirdStore();
        birdStore.printProperties("eagle");
    }
}
```

Dependency Inversion Principle

Örnekte görüldüğü gibi süper sınıf olan **AnimalStore** sınıfı, **abstract** olarak tanımlanmış **Animal** sınıfına bağlıdır. **Cow, Hourse, Eagle** ve **Hawk** sınıfları da **Animal** sınıfına bağlıdır. Yani **üst seviye(high level)** component ile **alt seviye(low level)** component arasında **abstract** tanımlanmış bir sınıf vardır. Yukarıdaki örnekte **high level** component **AnimalStore** sınıfı, **low level** component ise **Cow, Hourse, Eagle** ve **Hawk** sınıflarıdır. Eğer böyle bir tanımlama olmasaydı, **AnimalStore** sınıfı **Cow, Hourse, Eagle** ve **Hawk** sınıflarına bağımlı olacaktı. Bu sınıflardan herhangi biri değiştiği zaman **AnimalStore** sınıfı da bu değişimden etkilenecekti. **High level** ile **low level** arasında bir **abstraction** olmasından dolayı, **low level** deki değişimden **high level** etkilenmeyecektir. İşte bu prensibe, **dependency inversion principle** denilmektedir.

Bu prensibe göre aşağıdaki **şartların** gerçekleşmesi gerekmektedir:

1. Hiçbir **değişken** concrete(somut) bir sınıfın referansını **tutmamalıdır**. Çünkü **new** kullanırsak, somut bir sınıfa bağımlı oluruz. Bundan dolayı fabrika tasarım deseni kullanmak gereklidir.

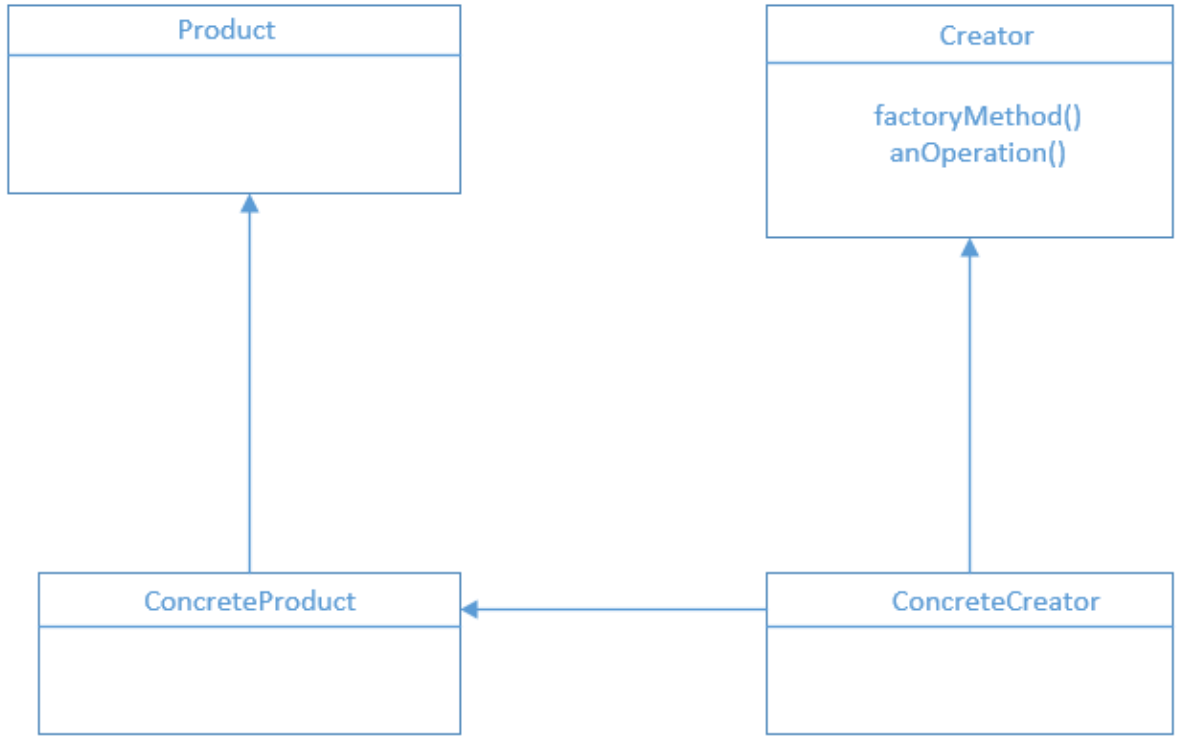
2. Hiçbir **sınıf** concrete(somut) bir sınıftan **türememelidir**. Çünkü bir somut sınıftan bir sınıf türetildiğinde yani alt sınıf olduğunda, bu alt sınıf, üst(süper) sınıfa bağımlı hale gelir. Bunun yerine üst sınıf mümkün olduğu kadar **interface** veya **abstract** tanımlanmalıdır.

3. Süper sınıftaki bir metod, alt sınıflar tarafından **override edilmemelidir**. Çünkü implement edilmiş bir metodu **override** edersek, **base** yani süper sınıf gerçek anlamda **abstraction** sınıf **olmaz**. Bir metod süper sınıfta implement edilmişse bunun anlamı alt sınıflar tarafından bu metod paylaşılmaktadır. Yani ortak özelliktir. Ortak özelliği değiştirmek pek mantıklı değildir.

Görüldüğü gibi dependency inversion prensibinin **katı kurallarını** uygulamak neredeyse imkansızdır. Bundan dolayı, kod yazarken mümkün olduğu kadar bu kurallara uymak gerekmektedir. Uymadığımız takdirde etkilerinin neler olabileceğini bilerek kod yazmalıyız.

Not: Çok zor değişen concrete bir sınıfa sahipsek, bu sınıftan nesne yaratmak problem olmaz. Örneğin, **String** sınıfını kullanırken aslında **new** ile bu sınıftan nesne yaratıp kullanıyoruz. Fakat **String** sınıfının değişmesi neredeyse imkansızdır. Bundan dolayı bir problem olmamaktadır. Fakat bu şekilde kullanım, **dependency inversion** prensibini ihlal etmektedir.

Fabrika Metod Tasarım Deseni'nin Şematik Gösterimi



Product ismi ile belirtilen sınıf alt product'ların **süper** sınıfıdır. Bu sınıf **abstract** veya **interface** olmalıdır. Örneğimizdeki **Animal** sınıfına denk gelir.

Creator ismi ile belirtilen sınıf **Product** sınıfına ait özelliklerin kullanıldığı sınıftır. Bu özellikler **anOperation()** ismiyle belirtilen metod aracılığı ile yapılır. **factoryMethod()** ise **abstract** tanımlanmıştır. Alt sınıflar **factoryMethod()** sınıfını kendileri implement edecektir. **Creator**, örneğimizdeki **AnimalStore** sınıfına, **anOperation()** metodu **printProperties()** metoduna, **factoryMethod()** metodu ise **createAnimal()** metoduna denk gelir.

ConcreteCreator ismi ile belirtilen sınıf, **factoryMethod()** metodunu implement eden **alt** sınıfları temsil eder. Ayrıca bir veya birden fazla concrete product nesnesi üretmekten sorumludur. **Sadece** bu sınıf, üretilecek yani **new** ile yaratılacak nesneleri bilir. Örneğimizdeki **BirdStore** ve **MammalStore** sınıfları **ConcreteCreator** sınıflarıdır.

ConcreteProduct ismi ile belirtilen sınıf ise kendisinden nesne üretilecek sınıf veya sınıfları temsil eder. Örneğimizdeki **Cow, Hourse, Hawk** ve **Eagle** sınıfları **ConcreteProduct'tır.**