SPRING FRAMEWORKUN KULLANDIĞI DESIGN PATTERNLAR

Singleton Pattern

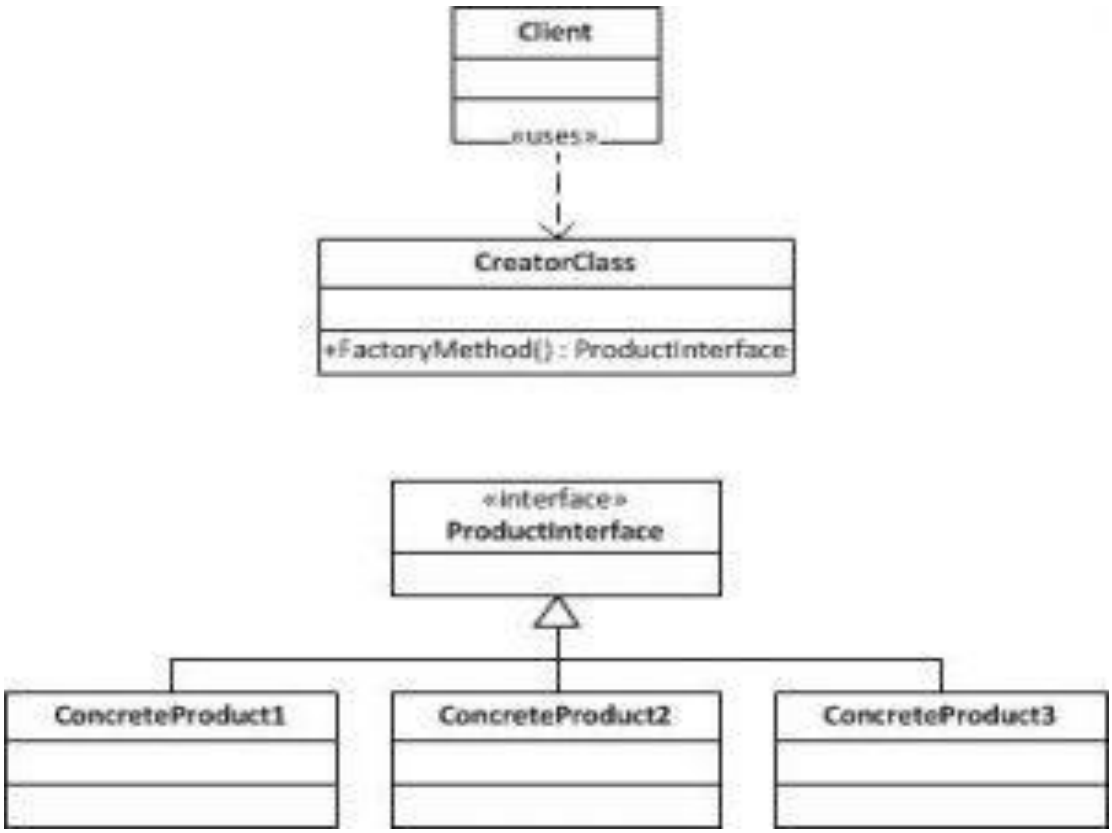
Singleton deseni, en basit tasarım modellerinden biridir. Bu desen, yalnızca tek bir nesnenin yaratıldığından emin olurken bir nesneyi oluşturmaktan sorumlu olan tek bir sınıfı içerir. Bu sınıf, sınıf nesnesinin somutlaştırılmasına gerek kalmadan doğrudan erişilebilen tek nesnesine erişmenin bir yolunu sağlar. Bunu da Java'da 'static' keywordünü kullanarak sağlarız.

SingleObject sınıfının constructorı bu sınıftan nesne yaratılamaması için private olarak tanımlıdır. Sınıf içinde bir tane aynı sınıftan instance bulunur ve bu instance yalnızca getInstance metodu tarafından erişilebilir. Tüm uygulamanızda bu sınıf üzerinden aynı tek nesneye erişebilirsiniz. Şimdi Java ile örneğini görelim.

```
public class SingleObject {

   //SingleObject sınıfından bir nesne oluştur
   private static SingleObject instance = new SingleObject();

   /*constructorın access modifierını private olarak tanımlayalım ki
bu sınıftan nesne oluşturulamasın*/
   private SingleObject(){}

   //Oluşturduğumuz nesneye erişim için getter
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //SingleObject sınıfında bulunan instance nesnesine erişelim
      SingleObject object = SingleObject.getInstance();

      object.showMessage();
   }}
```

Factory Method Design Pattern

Factory Method tasarım kalıbı , kalıtımsal ilişkileri olan nesnelerin üretilmesi amacıyla kullanılan patternlerden birisidir. Burada asıl olan bir metottur. Bu metodun üstlendiği iş ise istemcinin ihtiyacı olan asıl ürünlerin üretilmesini sağlamak.



Fabrika metodunun özelliği istemciden gelen talebe göre uygun olan ürünün üretilip istemciye verilmesidir. Tek bir sınıf ve metodun bunu üstlenebilmesi için polimorfik özelliği olan bir tipe ihtiyacımız var. Yani bir parent class ve bu parent classtan türeyen subclasslar. Bu yüzden productların interface olarak bir atası tasarlanır. Yani bizim creatorClassımız bir productu yani IProduct'ın taşıyabilceği türden bir referansı geriye döndürecektir.

Aşağıdaki örnekte Screen parent sınıfından 3 adet sub class oluşturulmuştur ve bu alt sınıfların üretiminden sorumlu bir creator class (factory) tanımlanmıştır.

```
public abstract class Screen  {
//Abstract parent class
      public abstract void Draw();
 }

//Concrete Product
 class WinScreen : Screen {
      public override void Draw() {
        System.out.println("Windows Ekranı");
     }
 }
//Concrete Product
 class WebScreen : Screen  {
     public override void Draw() {
       System.out.println("Web Ekranı");
     }
 }
//Concrete Product
 class MobileScreen : Screen  {
     public override void Draw()  {
       System.out.println("Mobile Ekranı");
     }
 }
```

```
enum ScreenModel  {
   Windows,
   Web,
   Mobile
 }
//Creator Class
 class Creator {
    public Screen ScreenFactory(ScreenModel screenModel) {
        Screen screen = null;
      switch (screenModel)  {
          case ScreenModel.Windows:
             screen = new WinScreen();
          break;

          case ScreenModel.Web:
            screen = new WebScreen();
          break;

          case ScreenModel.Mobile:
            screen = new MobileScreen();
          break;
       }
        return screen;
    }
 }
```

```
 class Program
{

    static void Main(string[] args)
    {
        Creator creator = new Creator();

        Screen screenWindows =
creator.ScreenFactory(ScreenModel.Windows);
        Screen screenWeb =
creator.ScreenFactory(ScreenModel.Web);
        Screen screenMobile =
creator.ScreenFactory(ScreenModel.Mobile);

        screenWindows.Draw();
        screenWeb.Draw();
        screenMobile.Draw();
    }

}
```

Proxy Pattern

Proxy kelime anlamı olarak **temsilci** anlamına gelir. Her ne kadar kelimenin türkçe karşılığı deseni direkt olarak tanımlıyor olsada biz yinede basitçe tanımlamak istersek; Proxy tasarım deseni, oluşturduğumuz bir **Proxy** sınıf üzerinden, başka bir sınıfı temsil etmemizi sağlar. Böylelikle **Proxy** sınıf, esas sınıfa müdahale etmeden istediği işlem veya işlemler bütününü gerçekleştirebilir.

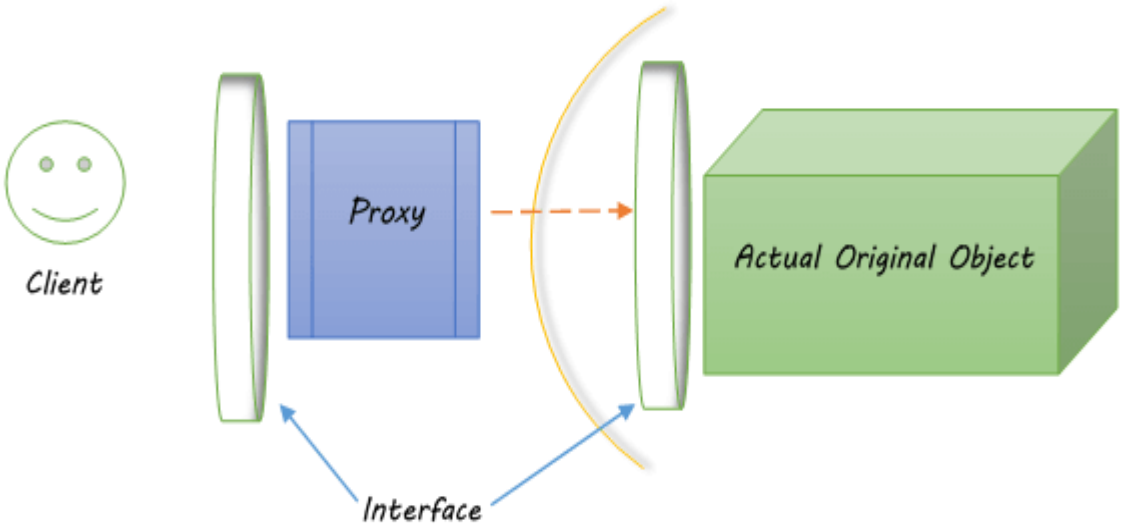


Figure 1 : Proxy Pattern Illustration

Proxy tasarım deseninin kullanıldığı birçok durum var. Popüler kullanımları ise:

1. **Virtual Proxy:** Ağır sistem kaynağı kullanan sınıflarda kullanılır. Uygulama başlar başlamaz; sınıfı oluşturmak yerine, sadece ihtiyaç olduğunda hedef sınıfı oluşturur.
2. **Protection Proxy:** Client'ın, sadece belirli durumlarda sınıfı çalıştırması gerekiyorsa uygulanır.

3. **Logging Proxy:** İstenilen belirli işlemlerden önce veya sonra, log göndermek için kullanır.
4. **Caching Proxy:** Sürekli olarak gönderilen aynı tipte isteklerin aynı sonucu geri dönderdiği durumlar kullanılır.

Proxy tasarım desenini 3 madde ile özetlemek istersek;

1. Bu tasarım deseni ile kod tekrarını önleriz. Böylelikle performans artışı da sağlamış oluruz.
2. Kompleks sınıfları, client in bilmesine zorlamadan yönetebilirsiniz.
3. **Proxy** tasarım deseni, **Open/Closed (SOLID)** presibine uyar. Mevcut sınıfı değiştirmeden, yeni proxy ler ekleyebilirsiniz.

Template Pattern

Template tasarım deseni, behavioral tasarım desenlerinden biridir. Bir algoritmanın adımlarını tanımlamayı sağlar ve alt sınıfların bir veya daha fazla adımların implementasyonunu yapmasını olanak tanır. Örneğin, bir ev inşa etmek istiyoruz. Ev inşa etmek için şu adımlar gereklidir: bina temel atılması, bina sütunları, bina duvarları ve pencereler. Burada dikkat edilecek olan husus, adımların sırayla yapılmasının zorunlu olmasıdır. Pencereler takıldıktan sonra temel atılamaz, önce temel atılır sonra bina sütunları yapılır, daha sonra duvarlar yapılır en son ise pencereler yapılır. Template metod tasarımı ile bu adımların sırasıyla yapılması zorunlu hale getirilmiştir.

Ne zaman Kullanılır?

Benzer adımlardan oluşan sınıfları tasarlarken, aynı kodları her sınıfta tekrar tekrar kullanmayı engellemek için kullanılır.

Nasıl Kullanılır?

Türü abstract olan bir sınıf yaratılır. Bu sınıf, içerisinde aynı adımlara sahip olan işlemleri temsil eder. Her işlem için bir metod eklenir. İşlemlerin sırasıyla yapılmasını zorunlu yapabilmek için ise template metodu eklenir. Template metodu işlem adımlarının zorunlu yapılması için final türüne sahip olur. Bu sayede alt sınıflar template metodu override edemezler. Template metod içerisinde, işlem adımları için oluşturulmuş metodlar sırasıyla çağrılır.

```
/**
 * Soyut sinifi
 */
public abstract class Education {
   /**
```

```
     * template metodu
     * final tanimlandi bu sayede alt siniflar override edemeyecek.
     */
    final void template(){
        elementary();
        secondary();
        highSchool();
        collage();
        //Hook kullanimi.
        //Goruldugu gibi algoritmayi alt sinifin degistirebilmesi saglandi
        if(hasMaster()){
            master();
        }
    }
    public abstract void elementary();
    //final tanimlanarak alt siniflarin override edebilmesi engellendi
    //Bu sayede alt siniflar metodu degistiremeyecek
    public final void secondary(){
        System.out.println("İkisi de Niksar Anadolu Ortaokulu'na gitti");
    }
    protected abstract void collage();
    protected abstract void highSchool();
    //Hook metodu. Default implementasyona sahip
    protected boolean hasMaster(){   return true;    }   protected abstract void master();}
/**
 * Somut sinif
 */
public class Ahmet extends Education {
    @Override
```

```
    public void elementary() {
        System.out.println("Ahmet Büyük Ata İlkokulu'na gitti");
    }

    @Override
    protected void collage() {
        System.out.println("Ahmet Gazi Üniversitesi'nde okudu");
    }

    @Override
    protected void highSchool() {
        System.out.println("Ahmet Niksar Anadolu Lisesi'nde okudu");
    }

    @Override
    protected boolean hasMaster() {
        return false; //Ahmet yüksek lisans yapmadığı için false donderdik
    }

    @Override
    protected void master() {
        System.out.println("Ahmet yüksek lisans yaptı");
    }
}
/**
 * Somut sinif
 */
public class Ayse extends Education {
    @Override
```

```
    public void elementary() {
        System.out.println("Ayşe Danişment Gazi İlkokulu'na gitti");
    }
    @Override
    protected void collage() {
        System.out.println("Ayşe Anadolu Üniversitesi'ne gitti");
    }

    @Override
    protected void highSchool() {
        System.out.println("Ayşe Ankara Fen Lisesi'ni gitti");
    }



    @Override
    protected void master() {
        System.out.println("Ayşe yüksek lisans yaptı");
    }
}
```

```
/**
 * Client sinifimiz
 */
public class Test {
```

```
    public static void main(String[] args) {
        Education ahmet=new Ahmet();
        ahmet.template();

        Education ayse=new Ayse();
        ayse.template();
    }
}
```