

TASARIM KALIPLARI

Tasarım Kalıbının Bileşenleri

- İsmi (ki neyi kastettiğimiz hemen anlaşılır)
- Problem (kalıbın neden ortaya çıktığını mevzudur)
- Çözüm (Problemin çözümünü tanımlar)
- Sonuçlar (kazanılan ve kaybedilenler)

YARATIMSAL KALIPLAR (Creational Patterns)

Yaratımsal Kalıplar

- Nesne tabanlı programlamada en temel sıkıntı nesne yaratmaktır.
- Kötü oluşturulan bir modelde nesneler gelişigüzelikten kaynaklı karmaşıktır. İyi oluşturulmuş modelde nesneler arasındaki ilişki, tanımlı durumları ve tekrar kullanılabilirlik sebebiyle anlaşılabilir bir karmaşıklığa sahiptir.
- Yaratımsal kalıplar, yazılım nesnelerinin nasıl yaratılacağı ile ilgili kullandığı esnek yapı sayesinde gerekli nesneleri yaratır.
- Hangi nesnenin çağırılması gerektiğini izlemeden sistemin uygun nesneyi çağırmasını sağlayan tasarım kalıplarıdır.
- Uygulamaya esneklik sağlar.
- Amaç, iyi yazılımın içinde barındırdığı nesnelerin nasıl yaratıldığından bağımsız olarak tasarlanması gerekliliğidir.

Nesne Oluşturmada

- İş nesnelerini oluşturmak zordur.
 - Nesne nerde oluşturulacak?
 - Nesne nasıl oluşturulacak? (Nesne durumları ve diğer nesnelerle ilişkisi)
- Nesne oluşturmak, bir iş süreci ya da bir iş kuralı gibi özel sorumluluktur ve sistemde bu sorumluluk için özel alanlar olmalıdır.

- Nesneler yapılandırıcılar ile oluşturulur.
 - Nesne tüm parametreler yapılandırıcıya verilerek oluşturulabilir.
 - JavaBean yöntemi ile oluşturulabilir.

- Nesne oluştururken ki anti-pattern ler;
 - Nesne oluşturan kodların her yere yayılması
 - İstemcilerin nesneye olan bağımlılığının artması
 - Nesnelerin yanlış ilişki ve durumlarla oluşturulması

- Nesne oluşturulurken;
 - Oluşturulması basitleştirilmeli
 - Tekrar kullanılabilir olmalı
 - İstemcilerin, nesne oluşturulmasına bağımlılıkları az olmalı.
- Bu durumları sağlamak için nesne oluşturma kodun diğer kısımlarından soyutlanmalıdır. Nesneler sadece belli yerlerde oluşturulmalı ve gereksiz new kullanımı olmamalıdır.
- Ayrıca nesneler sadece onların durumunu çok iyi bilenler tarafından oluşturulmalıdır.

- **Tekil Kalıp - Yegane (Singleton Design Pattern):** Bir nesneden sadece bir tane oluşturmak.
- **Fabrika Metodu (Factory Method):** Nesne oluşturmayı soyutlamak.
- **Soyut Fabrika Metodu (Abstract Factory Pattern):** Nesne ailesi oluşturmayı soyutlamak.
- **İlk örnek Kalıp - Örnek (Prototype Design Pattern):** Bir örnek nesneden kopyalama ile yeni nesne üretmek.
- **Yapıcı Kalıp (Builder Design Pattern):** Karmaşık nesne oluşturma sürecini kurgulamak.

Tekli Kalıp (Yegane - Singleton)

- Nesnenin sadece bir defa oluşturulmasını öngören bir mekanizma kurulmak istenildiğinde etkin bir biçimde kullanılabilen bir tasarım deseni.
- Oluşturulan bir sınıftan sadece bir nesne yaratılacak şekilde bir kısıtlama yapabilme olanağı sağlar ve nesneye ilk kez ihtiyaç duyulana kadar yaratılmayabilir.
- En basit kalıptır.
- Global değişkenlerine göre daha sağlıklıdır.
- Static metotlardan daha genişleyebilir bir yapıdır.

- Amaç bu sınıftan her isteyeninin bir nesne oluşturamaması, var olan nesneyi kullanmasıdır.
- Singleton oluşturmak için;
 - Nesne oluşturma kontrol edilebilir
 - Enumeration kullanılabilir

Nesne Oluşturma ile

- Aşağıda ne kontrol edilebilir?
Yegane ygn=new Yegane();
 - Sınıf private olsa?
 - Varsayılan yapılandırıcı private olsa?

- Yapılandırıcının private olması sebebiyle çağırılıp nesne oluşturulamadı.
- Çözüm;
 - Sınıfın kendi içinde kendi nesnesini oluşturması ve bunu döndürecek bir metot olması.

```
private Yegane ygn=new Yegane();  
public Yegane getYgn(){  
    return ygn;  
}
```

- Sınıf kendi instance değişkeni olması sebebiyle bu da durumu sonsuz döngüye sokar;
- Çözüm;
 - Metodu static tanımlanması. Dolayısıyla nesnenin de static olması.

```
private static Yegane ygn=new Yegane();  
public static Yegane getYgn(){  
    return ygn;  
}
```

- Sonuç olarak diğer sınıfta new anahtar kelimesi kullanmadan nesne oluşturmuş oluruz.

Yegane ygn=Yegane. getYgn();

- Bunun sıkıntısı static olması sebebiyle sınıf yüklenirken initialize edileceği için kullanmayacağımız durumlar için sıkıntılıdır.
- Çözüm;
 - Gecikmeli çağırma. Nesnenin null olup olmaması kontrol edilerek, sadece new işlemi get metodu içine alınır.
if(ygn==null)
 ygn=new Yegane();

- Gecikmeli çağırmanın sıkıntısı ise çoklu çağırma da (multi-threaded) problemlere yol açar. Null kontrolü birden fazla kanalın yapması, birden fazla nesne oluşmasına sebep olur.
- Çözüm;
 - Senkronizasyon yapmak. (C# da lock)

```
Synchronized (Yegane.class){  
    if(ygn==null)  
        ygn=new Yegane();  
}
```

- Bunun sıkıntısı metodun çalışması bihayli yavaşladı. Çünkü her metot farklı thread ı bekleyeceği için.
- Çözüm;
 - Senkronizasyon işleminden önce nesnenin oluşup oluşmadığı kontrol edilir (Double check Locking).

```
if(ygn==null)
{
    Synchronized (Yegane.class){
        if(ygn==null)
            ygn=new Yegane();
    }
}
```

Enumeration

- Enumeration ile singleton asla kırılmaz. (Serialization, Reflection)
- Enum'daki amaç sabit bir şekilde nesne oluşturmak.

```
public enum SingletonEnum {  
    INSTANCE;  
  
    int value;  
  
    public int getValue() {  
        return value;  
    }  
  
    public void setValue(int value) {  
        this.value = value;  
    }  
}
```

```
public class EnumDemo {  
  
    public static void main(String[] args) {  
        SingletonEnum singleton =  
            SingletonEnum.INSTANCE;  
  
        System.out.println(singleton.getValue());  
        singleton.setValue(2);  
        System.out.println(singleton.getValue());  
    }  
}
```

- Singleton nesnenin güncellenmesi gerekiyorsa bu işlem, durumunu değişmez (final/readonly) yaparak ya da sadece bir istemci tarafından güncellenebilir yaparak sağlanabilir.
- Singleton nesne hem uygulamanın her yerinden ulaşılabilir durumdadır hem de uygulama boyunca bellekte kalmaya devam eder.
- Geç yükleme tercih edildiğinde çok kanallı ortamlarda problem çıkması sebebiyle bunları gidermek için gereken karmaşık süreçler sıkıntılıdır.
- Tüm kurucuları private olduğu için miras alınamazlar ama miras alabilirler.

- Aşağıdaki kalıplar için Singleton kalıbı kullanılır;
 - Abstract Factory
 - Builder
 - Prototype

Fabrika Yöntemi (Factory Method)

- Nesnenin nasıl yaratılacağını kalıtım yoluyla alt sınıflara bırakıp nesne yaratımı için tek ara yüz kullanarak, ara yüzle nesne yaratım işlevlerini temelde birbirinden ayırmaya yarayan yaratımsal tasarım kalıbıdır.
- Yani nesne oluşturmak için arayüz tanımlanır fakat hangi sınıfın nesnesinin oluşturulacağına alt sınıflar karar verir.
- Nesneyi oluşturmak, kullanmaktan daha karmaşık olabilir.

Neden bu kalıba ihtiyaç var?

- Clientların nesnenin sadece arayüzlerini değil nasıl oluşturulacağını da bilmesi bağımlılığı artırır.
- İş nesnelere kodun farklı yerlerinde ihtiyaç olacağı için copy-paste kaçınılmazdır.
- Çok parametrelili yapılandırıcılar oluşmasına sebep olabilir.

- Bu sorunların üstesinden gelebilmek için;
- Nesne oluşturmalarını soyutlayarak bu işi client yerine factory metotlarda (Bir nesnenin oluşturulmasından sorumlu metot) yapılmasını sağlar
- Bir arayüzle tanımlanır ve alt sınıflarda gerçekleştirilir.
- Her alt sınıf başka nesne oluşturur.

- Öncelikle bir interface oluşturulur;

```
public interface Shape {
```

```
    void draw();
```

```
}
```

- Sonra sırayla sınıflar oluşturulur;

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectengle");  
    }  
}
```

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square");  
    }  
}
```

- Factory oluşturulur;

```
public class ShapeFactory {  
    public Shape getShape(String shapeType) {  
        if (shapeType == null) {  
            return null;  
        }  
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {  
            return new Rectangle();  
        }  
        else if (shapeType.equalsIgnoreCase("SQUARE")) {  
            return new Square();  
        }  
        return null;  
    }  
}
```

- Main kısmı;

```
public static void main(String[] args) {  
    ShapeFactory shapeFactory = new  
ShapeFactory();  
  
    Shape shape1 =  
shapeFactory.getShape("RECTANGLE");  
    shape1.draw();  
  
    Shape shape2 =  
shapeFactory.getShape("SQUARE");  
    shape2.draw();  
}
```

- Nesneler farklı metotlarla oluşturuluyorsa tek interface işimizi çözmez.
- Interface yerine staticler ile somut sınıflar yaparsak anti pattern olur.
- Bu pattern'in en temel problemi, her yeni sınıf için yeni bir Factory alt sınıfına hatta ayrı arayüzler olması durumunda yeni bir arayüze ihtiyaç olmasıdır.
- Bu patterndaki else if karmaşasının önüne geçmek için abstract factory pattern'i ortaya çıkmıştır.

Kaynakça

- <https://en.wikipedia.org/>
- <https://bidb.itu.edu.tr/>
- <https://www.tutorialspoint.com/>
- Akin Kaldıroğlu ders notları