

# Spring'in Kullandığı Tasarım Desenleri

**Creational Patterns (Yaratımsal Kalıplar):** Bu tasarım deseni nesneleri doğrudan new operatörü kullanarak oluşturmak yerine nesne oluşturma mantığını gizleyerek sınıflardan nesne oluşturmaya alternatif çözümler sunar. Bu program akışında hangi nesneye ihtiyaç varsa onu oluşturmada esneklik ve kolaylık sağlar.

**Oluşturucu Tasarım Şablonları :** Factory , Factory Method , Abstract Factory , Singleton, Builder, Prototype Object Pool.

**Structural Patterns (Yapısal Kalıplar):** Bu tasarım deseni nesneler arasındaki ilişkinin yapısını düzenlemek için çözümler sunar.

**Yapısal Tasarım Şablonları :** Class Adapter ,Object Adapter ,Bridge ,Facade ,Composite ,Decorator, Virtual Proxy ,Dinamic Proxy, Protection Proxy, Remote Proxy, Flyweight

**Behavioral Patterns (Davranışsal Kalıplar):** Bu tasarım deseni çalışma zamanında nesneler arasındaki davranışlar için çözümler sunar.

**Davranışsal Tasarım Şablonları :** Command ,Iterator ,Memento ,State ,Observer ,Strategy ,Chain Of Responsibility ,Mediator ,Visitor ,Template Method ,Interpreter

**Java EE Tasarım Şablonları :** Model View Controller ,Front Controller ,Data Access Object ,Business Delegate ,Service Locator ,Intercepting Filter ,Business Objecta

**Diğer Tasarım Şablonları :** DataMapper ,RequestMapper ,ResponseMapper ,Active Record ,Message Channel ,Message Router , Registry , Null Object , Dependency Injection

Yazılım esnasında tekrar eden sorunları çözmek için kullanılan ve tekrar kullanılabilir yapıda kod yazılımını destekleyen, bir ya da birden fazla sınıftan oluşmuş modül ve program parçalarına tasarım şablonu (design pattern) ismi verilir. Tasarım kalıbı adı üstünde bir tasarımı tasarlarken tasarımın kullanılabilirliği, test edilebilirliği, verimliliği, kalitesi vb birçok şeye dikkat edilmesi gerekir. Yazılım dünyasında geçmişten bugüne oluşan bu kalıplar ,yazılımcıların çözdükleri problemlerin benzerlikleriyle ve sık tekrar etmesiyle ihtiyaç duydukları ve bu çözümleri belli kalıplar haline getirerek bulduğu kalıplara tasarım deseni diyoruz. Bunlar çözdükleri problemlere yaptıkları işlemlere göre geçmişten günümüze genel olarak isimlendirilerek bugünkü hallerini almışlar. Yaşanılan sorunları çözen kalıplar olarak ihtiyaca ve tasarıma göre kullanılırlar.

## Creational Patterns

**1.1. Tek Nesne ( Singleton ) Tasarım Deseni** Bu tasarım deseninde, bir sınıfın sistem içinde yalnızca bir tane nesnesi oluşturulabilir. Tek bir arayüz sunularak, bu nesneye yalnızca buradan erişim sağlanabilir. Bu desen kullanılarak, sistem içinde değeri değişmeyen, genel değişkenler bu oluşturulan tek nesneye konulabilir. Sistemde tek nesne yaratılabilme, **statik** değişken ve yordamlar sayesinde olur.

```
1 class Singleton
2 {
3     private static Singleton instance;
4     private Singleton()
5     {
6         ...
7     }
8     public static synchronized Singleton getInstance()
9     {
10         if (instance == null)
11             instance = new Singleton();
12         return instance;
13     }
14     ...
15     public void doSomething()
16     {
17 ...
18     }
19 }
```

### ***Fabrika Yordam ( Factory Method ) Tasarım Deseni***

Fabrika yordam tasarım deseni, nesne yaratma sorumluluğunun bir yordama verilmesidir. Yaratılan nesne, bir sınıf hiyerarşisindeki alt sınıflardan biridir. Hangi alt sınıfın yaratılacağı kararı fabrika yordam içinde verilir. Bu yordam ile belirli bir sınıf hiyerarşisindeki alt nesnelerden birinin yaratma sorumluluğu belirli bir arayüze verilerek sistemden soyutlanmış olur. Böylece nesneleri yaratma kodlarında, kod tekrarları önlenmiş olur. Sistem içinde sınıfların yaratılacağı yer tek olduğu için, ilgili mantıklar tek bir yerde toplanabilir.

Bu kalıbı tasarlamak için bir fabrika sınıfına ihtiyaç duymaktayız.

İlanımızın Tarla Tipinde ve Konut Tipinde oluşturabildiğimizi varsayarak böyle şartlandırarak nesne oluşumunda bağımlılığı azaltmış oluyoruz ve new ile yeni bir nesne üretmemiş oluyoruz.

The image consists of two screenshots of an IDE, likely IntelliJ IDEA, showing Java code. The top screenshot displays the `IlanFactory.java` file. It contains a package declaration `package hafta1.model;` and a public class `IlanFactory`. Inside the class, there is a public static method `getIlan(String tip)`. The method uses an if-else statement to return a new instance of `Tarla` if the tip is "Tarla" (ignoring case) and a new instance of `Konut` if the tip is "Konut" (ignoring case). If neither condition is met, it returns null. The bottom screenshot shows two files side-by-side. The left file is `Tarla.java`, which extends `Ilan` and has a public static method `getIlan()` that returns the string "Tarla sectin". The right file is `Konut.java`, which also extends `Ilan` and has a public static method `getIlan()` that returns the string "Konut Sectin". Both files are in the `hafta1.model` package.

```
package hafta1.model;

public class IlanFactory {

    public static Ilan getIlan(String tip) {

        if ("Tarla".equalsIgnoreCase(tip))
            return new Tarla();
        else if ("Konut".equalsIgnoreCase(tip))

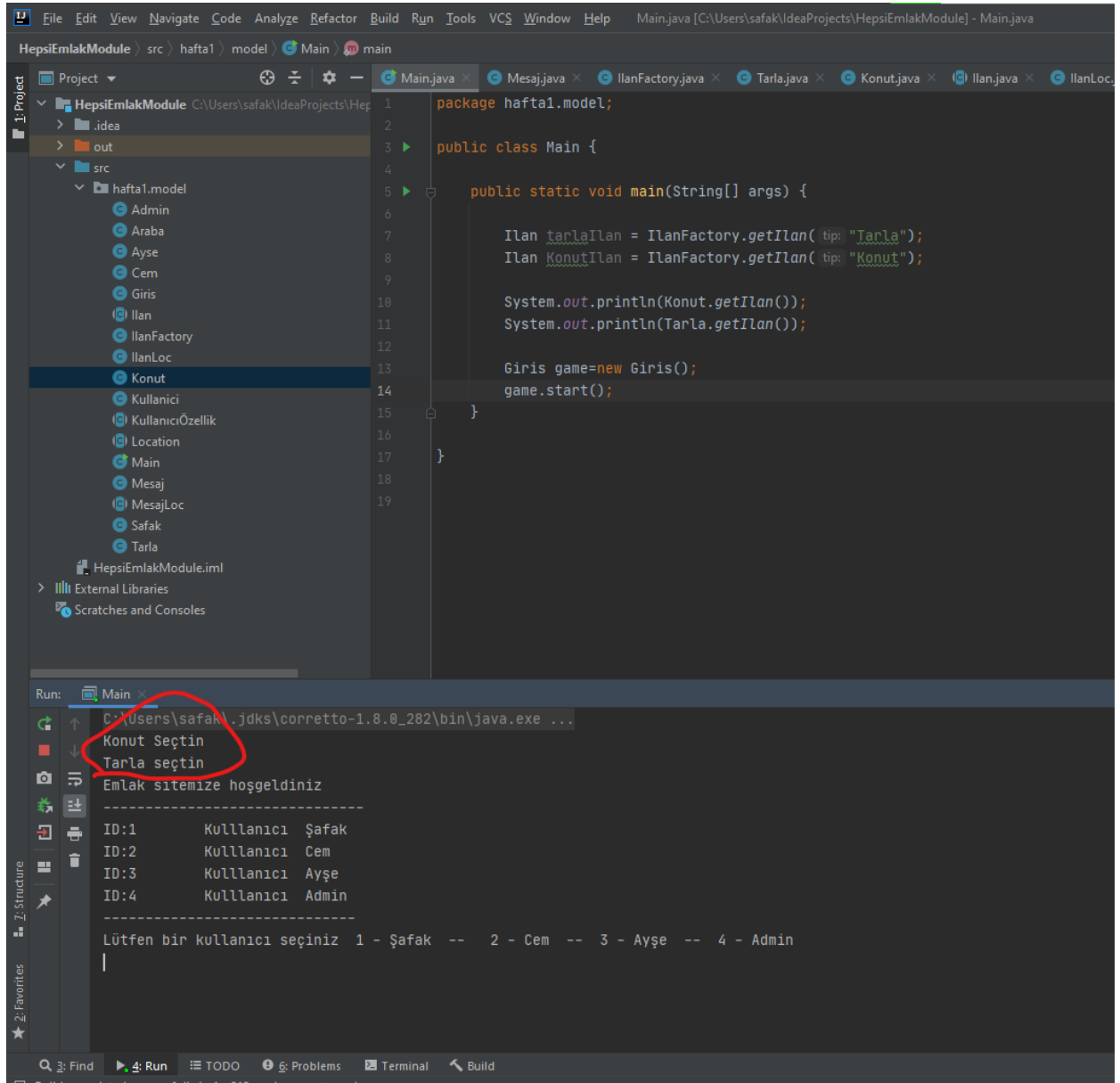
            return new Konut();

        return null;
    }
}
```

```
1 package hafta1.model;
2
3 public class Tarla extends Ilan {
4
5
6     @ public static String getIlan() {
7
8         return "Tarla sectin";
9     }
10 }
11
```

```
1 package hafta1.model;
2
3 public class Konut extends Ilan{
4
5     @ public static String getIlan() {
6
7         return "Konut Sectin";
8     }
9 }
10
```

Tarla ve konut classlarımız şekildeki özelleştirerek ayırıyoruz.



Ve sonunda Ilan sınıfımızdan neşen oluşturup Fabrikamızdan yardım alarak new kullanmadan istediğimiz nesneyi istediğimiz şekilde oluşturuyoruz.