Creational Design Patterns

Singleton Design Pattern

Singleton Design Pattern, aynı nesneyi çağırıp kullanabileceğimiz durumlarda, performans kaybı yaşamak yerine, hali hazırda oluşmuş bir nesneyi kullanma amacı taşıyan bir tasarımdır. Oluşturulan her yeni nesne hafızada yeni bir alanı doldurur. Aynı obje ihtiyacımızı gideriyor ise, onu tekrar oluşturarak gereksiz performans kaybı yaşarız. Singleton Pattern, bu gereksiz hafıza kaybını önleyerek, method aracılığı ile obje üzerinden classa ulaşmayı sağlar. Gerekli objeyi oluşturacağımız constructorı private yaparak, bu nesnenin oluşabilmesinin önüne geçilir ve nesne, bir method yardımı ile sadece bir kez oluşturulur. Sonraki isteklerde, bu oluşturulan nesne, bu nesneyi return edecek method çağrılarak kullanılır.

```java
package com.company;

public class Database {

    private static Database database =new Database();

    private Database() {}

    public static Database getDatabase() { return database; }

    public void showMessage() { System.out.println("Logged to database."); }
}
```

Yukarıda kod bloğunda görüldüğü gibi, bir database objesi oluşturduk ve Database den obje oluşturmamıza yarayan constructor ı private yaptık, artık bu class dan bir obje oluşturulamaz. Geriye oluşturduğumuz database objesini return eden fonksiyonu yazarak, oluşturduğumuz nesnemize global erişimi sağladık.

```java
package com.company;

public class Main {

    public static void main(String[] args) {

        //Burada yeni bir Database objesi oluşturmak yerine,
        //get methodu yardımı ile zaten varolan objeyi kullanıyorum.
        Database applySingleton=Database.getDatabase();

        //kullandığım obje ile showMessage methoduna erişiyorum.
        applySingleton.showMessage();
    }
}
```

Database classında yazdığımız showMessage() methodunu çağırabilmek için, Database classından bir nesneye ihtiyacımız var. Aşağıdaki kod parçasında görüldüğü gibi, private olduğu için yeni nesne oluşturamıyoruz, var olana erişim sağlayabiliyoruz.

```java
        //yeni bir nesne oluşturamıyoruz.
        Database database2=new Database();
```

Bu durumda, ihtiyacımızı karşılayacak olan nesnemizi, getDatabase() methodu yardımı ile çağırdık ve showMessage() methoduna erişebildik.

Prototype Design Pattern

Prototip Design Pattern, elimizde olan nesnelerin klonlanmasını sağlayan bir tasarımdır. Bu yasarımın amacı, elimizde olan nesneleri hızlı bir şekilde üretmektir. Singleton Design Pattern' i anlatırken, her

yeni objenin bir maliyeti olduğundan bahsetmiştim. Prototype Pattern' in de değindiği nokta tam olarak bu diyebiliriz. Prototype Design Pattern, "new" keywordünü kullanmadan yani yeni bir nesne üretmeden, var olan nesneyi klonlayarak yeni bir nesne elde eder.

Aşağıdaki kod bloğunda görüldüğü gibi, "Prototype" adında bir interfaceimiz var ve içinde clone() adında bir methodumuz bulunuyor. User sınıfımız, Prototype' ı implemente ederek clone() methodunda yeni bir user nesnesi oluşturuyor. Main classda da bir user oluşturup, ikinci bir user oluştururken clone() methodundan yararlanıyoruz.

```java
package com.company;

public class User implements Prototype {

    String email;
    String password;

    public User() {
    }

    public User(String email, String password) {
        this.email = email;
        this.password = password;
        signUp();
    }

    @Override
    public Prototype clone() { return new User(email,password); }

    private void signUp() { System.out.println(email+" : Giriş başarılı"); }
}
```

```java
package com.company;

public interface Prototype {

    Prototype clone();

}
```

```java
package com.company;

public class Main {

    public static void main(String[] args) {

        User user=new User( email: "seyma@hotmail.com", password: "123456");
        User user1= (User) user.clone();
    }
}
```

Factory Design Pattern

Birbirinden kalıtılabilir nesneleri oluşturmak için Factory Design Pattern kullanırız.

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

```
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 Design Pattern

Factory Design Pattern' da tek bir ürün ve alt sınıflarına ait tek bir arayüz vardır. Abstract Design Pattern' da ise, farklı ürün ve alt sınıfları için farklı arayüzler mevcuttur.

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

```
public class Duck implements Animal {

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

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

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

```java
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;
    }

}
```

```java
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 Design Pattern

Bir sınıf için birden fazla sayıda field olduğunda ve bu fieldların her biri, nesne ilk oluşturulduğunda ihtiyaç dışı ise bu fieldları constructor içinde kullanmayabiliriz. Birden fazla constructor oluşturup, ihtiyaca göre nesne oluşturabiliriz. Fakat, fieldların sayısını oldukça fazla sayıda düşünürsek birden fazla constructor oluşturmak kafa karıştırıcı olur ve kirli bir görünüme sebebiyet verir. Bu istenen bir durum değildir. Builder Design Pattern tam da bu noktada ihtiyacımızı gideren, sorunumuzu çözen bir pattern. Builder' a göre, bir Builder sınıfı oluşturup, fieldları bu sınıfa verip, encapsule edip, ana sınıfımızda da constructora Builder objesi verebiliriz. Böylece objeyi oluştururlen encapsulation yardımı ile fieldlara ulaşırız.

```java
package com.company;

import java.util.Date;

public class TodoList {

    String title;
    String content;
    String userId;

    public TodoList(TodoBuilder todoBuilder){
        this.title=todoBuilder.title;
        this.content=todoBuilder.content;
        this.userId=todoBuilder.userId;
    }

}
```

```java
package com.company;

import java.util.Date;

public class TodoBuilder {
    String title;
    String content;
    String userId;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }


    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }
}
```

```java
package com.company;

public class Main {

    public static void main(String[] args) {

        TodoBuilder todoBuilder=new TodoBuilder();
        todoBuilder.title="It is a title";
        todoBuilder.content="Publish on Medium";
        todoBuilder.userId="E8974";
        TodoList todoList=new TodoList(todoBuilder);

    }
}
```

Kaynaklar

1) https://www.baeldung.com/creational-design-patterns
2) https://www.journaldev.com/1827/java-design-patterns-example-tutorial
3) https://www.javatpoint.com/design-patterns-in-java
4) https://www.geeksforgeeks.org/

Şeyma Tezcan