

Creational Design Patterns (Yaratımsal/Varoluşsal Tasarım Desenleri/Kalıpları/Örüntüleri)

3 ana tasarım kalibinden bir tanesini oluşturur. Uygulama içerisinde oluşturulacak nesnenin farklı koşullara göre farklı şekilde oluşturulması ve yönetilmesi ile ilgilenmektedir. En temelde nesnenin doğrudan new anahtar sözcüğü üzerinden standart şekilde oluşturulmasına alternatif yöntemler sunar.

1) Singleton Pattern (Tekil Desen)

Bu tasarım deseni bir sınıfın globalden ulaşılabilen tek bir örnek (instance) oluşturulmasını ve sınıf nesnesine her ihtiyaç olduğunda oluşturulan bu tek nesnenin kullanılmasını temel alır. Bu şekilde bir nesne oluşumu için sınıfların varsayılan yapıçı methodlarının kullanımı uygun değildir. Varsayılan yapıçı methodlar her kullanıldıklarında yeni bir nesne dönecek şekilde tasarlanmıştır.

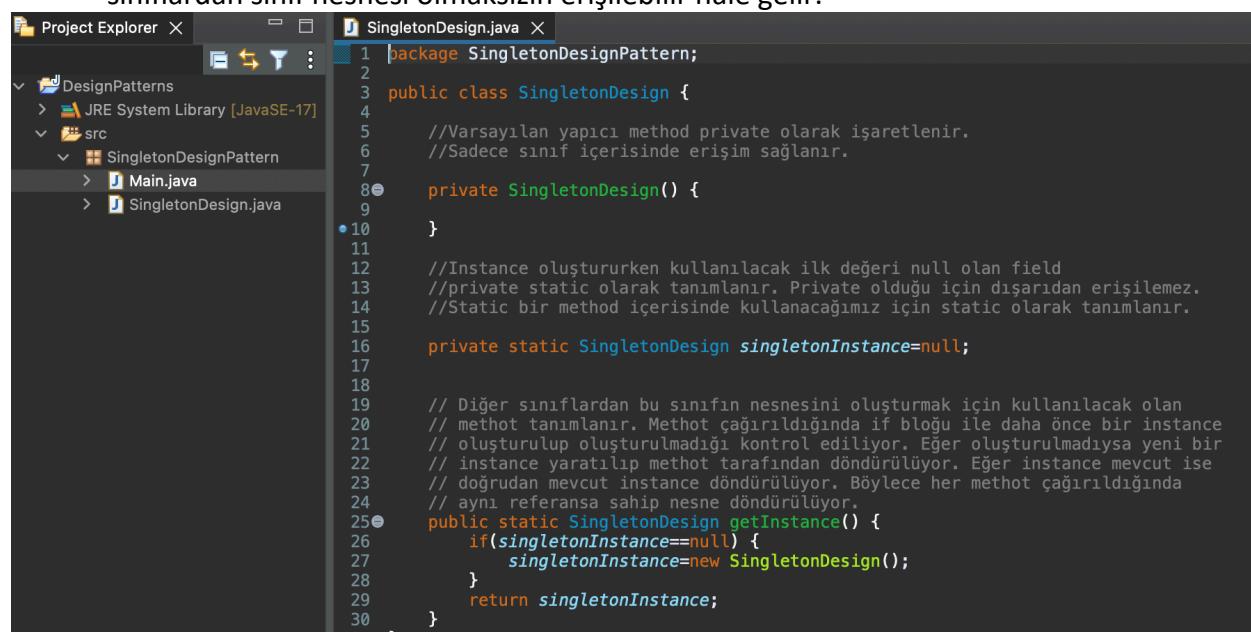
Tek Örnek Gerektiren Singleton Pattern kullanım alanları:

Veri tabanı bağlantıları, Port bağlantıları, Dosya işlemleri, Loglama işlemleri, Bildirimler, İş katmanı servisleri.

Çıktısı: Bir sınıfın sadece tek bir instance oluşumunu garanti eder. Global olarak bu örneğe ulaşım sağlanabilir.

Singleton Pattern Kullanım Adımları:

- Varsayılan yapıçı method private tanımlanarak diğer sınıflardan erişimi kısıtlanır.
- İlk nesnenin atanacağı değişken private static olarak tanımlanır. İlk değeri null olmalıdır.
- İlk nesnenin yaratılacağı method public static olarak işaretlenir. Bu sayede diğer sınıflardan sınıf nesnesi olmaksızın erişilebilir hale gelir.



The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays a project named 'DesignPatterns' containing a 'src' folder with a 'SingletonDesignPattern' package, which contains 'Main.java' and 'SingletonDesign.java'. On the right, the code editor window shows the 'SingletonDesign.java' file with the following code:

```
1 package SingletonDesignPattern;
2
3 public class SingletonDesign {
4
5     //Varsayılan yapıçı method private olarak işaretlenir.
6     //Sadece sınıf içerisinde erişim sağlanır.
7
8     private SingletonDesign() {
9
10    }
11
12     //Instance oluştururken kullanılacak ilk değeri null olan field
13     //private static olarak tanımlanır. Private olduğu için dışarıdan erişilemez.
14     //Static bir method içerisinde kullanacağımız için static olarak tanımlanır.
15
16     private static SingletonDesign singletonInstance=null;
17
18
19     // Diğer sınıflardan bu sınıfın nesnesini oluşturmak için kullanılacak olan
20     // method tanımlanır. Methot çağırıldığında if bloğu ile daha önce bir instance
21     // oluşturulup oluşturulmadığı kontrol ediliyor. Eğer oluşturulmadiysa yeni bir
22     // instance yaratılıp method tarafından dönürtülüyor. Eğer instance mevcut ise
23     // doğrudan mevcut instance dönürtülüyor. Böylece her methot çağırıldığında
24     // aynı referansa sahip nesne dönürtülüyor.
25     public static SingletonDesign getInstance() {
26         if(singletonInstance==null) {
27             singletonInstance=new SingletonDesign();
28         }
29         return singletonInstance;
30     }
31 }
```

The screenshot shows the Eclipse IDE interface with the 'Project Explorer' view on the left displaying a Java project named 'DesignPatterns'. Inside the 'src' folder, there are two files: 'Main.java' and 'SingletonDesign.java'. The 'Main.java' file is open in the central editor, showing the following code:

```
1 package com.singletondesignpattern;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         // İlgili sınıfın yapıcı methodu private olduğu için yapıcı
8         // methodu kullanarak nesne yaratamıyoruz.
9         SingletonDesign singletonInstance2=new SingletonDesign();
10
11        // Sınıf içerisindeki örnek oluşturma metoduna static işaretli olduğu için
12        // sınıfın nesnesine ihtiyaç duymadan ulaşabiliyoruz. Method sayesinde ilk
13        // ve kalıcı methodumuzu oluşturuyoruz.
14        SingletonDesign singletonInstance3=SingletonDesign.getInstance();
15
16        // Aynı methodu farklı bir değişken ismi ile tekrar çağrıdığımızda önceden
17        // oluşturmuş olduğumuz nesneyi döndürecektir.
18        SingletonDesign singletonInstance4=SingletonDesign.getInstance();
19
20        // İki değişkenin de aynı referansa sahip olması durumunda true dönecektir.
21        System.out.println(singletonInstance3==singletonInstance4);
22    }
23
24 }
```

The 'Console' tab at the bottom shows the output: <terminated> Main [Java Application] /Users/mariami/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64 true

2) Prototype Pattern (Örnek Tasarım Kalıbı)

Temel olarak mevcut bir nesnenin bire bir kopyasına veya çok yakın bir versiyonuna sürekli ihtiyaç duyulduğunda, bu kopyaların new anahtar sözcüğü ile yaratılmasının maliyetinden kurtulmak amacıyla kullanılan ve clone() methoduna dayanan bir tasarım desenidir. Bahsedilen maliyet sınıfın yapıcı methoduna verilmesi gereken parametre değerleridir. Teker teker bu parametreleri belirtmeye gerek kalmadan nesne oluşturulabiliriz. Kopyalama sırasında deep copy yöntemi uygulanır. Çoğunlukla oyun programlama'da kullanılır.

Çıktısı: Nesne Maliyetlerini Düşürür.

Prototype Pattern Kullanım Adımları:

- İlgili sınıf ve fieldları tanımlanır,
- Field getter ve setterleri ve bu fieldların değer atama işlemlerini içeren sınıf yapıcı methodu oluşturulur.
- Sınıf Clonable interface'ini implemente eder.
- Sınıf içerisinde clone methodunu ezilerek kullanıldığındá deep copy yapacak şekilde güncellenir.
- Sınıfın ilk nesnesi new anahtar kelimesi ile yaratılır. Diğer nesneler bu sınıfın override edilen clone methodu kullanılarak kopyalama yöntemi ile hızlıca yaratılır. Yeni nesnenin güncellenmesi gereken field değerleri setter methodları kullanılarak güncellenebilir.

The screenshot shows the Eclipse IDE interface. The Project Explorer view on the left displays a project named 'DesignPatterns' containing two packages: 'PrototypeDesignPattern' and 'SingletonDesignPattern'. Both packages have 'Main.java' and 'Tree.java' files. The 'Tree.java' file is currently open in the code editor on the right. The code implements the Prototype design pattern for a 'Tree' class, including a constructor, field getters/setters, and a clone method that performs a deep copy.

```
1 package PrototypeDesignPattern;
2
3 //Oluşturulan sınıf Cloneable arayüzüni implement eder
4 public class Tree implements Cloneable{
5
6     //Sınıfin fieldları oluşturulur.
7     private int height;
8     private int width;
9     private int age;
10
11    //Field değerlerini içeren yapıcı method olusturulur.
12    public Tree(int height, int width, int age) {
13        super();
14        this.height = height;
15        this.width = width;
16        this.age = age;
17    }
18
19    // Field getter ve setter metotları oluşturulur.
20    public int getHeight() {return height;}
21    public void setHeight(int height) {this.height = height;}
22    public int getWidth() {return width;}
23    public void setWidth(int width) {this.width = width;}
24    public int getAge() {return age;}
25    public void setAge(int age) {this.age = age;}
26
27    //Clone methodunu tree objesinin deep copy versiyonunu döndürecek
28    //şekilde güncelliyoruz.
29    @Override
30    public Tree clone() {
31
32        Tree tree = null;
33
34        try {
35            tree = (Tree) super.clone();
36        } catch (CloneNotSupportedException e) {
37            System.out.println("Tree clone'u yaratılamadı.");
38            e.printStackTrace();
39        }
40        return tree;
41    }
42
43    }
44 }
```

The screenshot shows the Eclipse IDE interface. The Project Explorer view on the left displays the same 'DesignPatterns' project structure. The 'Main.java' file is currently open in the code editor on the right. It contains a main method that creates a 'Tree' object, clones it, and prints the result to the console, demonstrating that the clone operation creates a deep copy.

```
1 package PrototypeDesignPattern;
2
3 public class Main {
4
5    public static void main(String[] args) {
6
7        //Tree nesnesi yarattık.
8        Tree tree1=new Tree(1,2,15);
9        //Clone methodunu kullanarak yeni bir tree değişkenine
10       //nesne ataması yaptı.
11       Tree tree2=tree1.clone();
12       // Eşitlik durumunu incelediğimizde false dönmektedir.
13       // Bu da iki farklı nesneye ait referans olduğunu kanıtlar.
14       System.out.println(tree1==tree2);
15
16    }
17
18 }
```

Below the code editor, the Console view shows the output: <terminated> Main (1) [Java Application] /Users/mariami/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64 false

3)Factory Method Pattern (Fabrika Tasarım Kalıbı)

Belirli koşullara bağlı olarak benzer nesnelerin üretilmesi gereken durumlarda kullanılabilir. Basit şekilde düşünüldüğünde if-else veya switch-case gibi karar kontrol mekanizmaları kullanılabilir. Ancak sürekli bu yapıları kullanmak kod açısından maliyetli olacaktır.

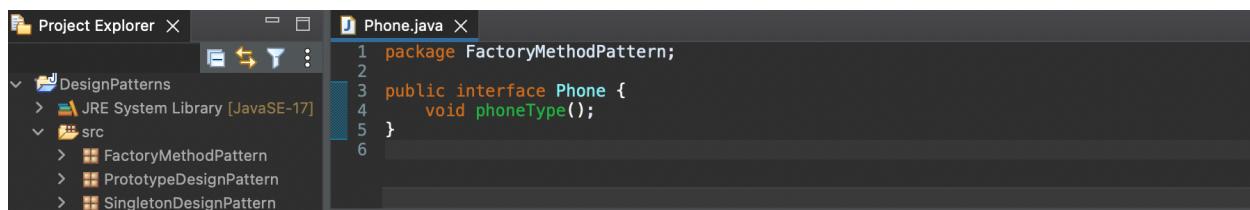
Bu yöntemde temelde bir interface veya abstract sınıf bulunmaktadır. Nesnesi üretilecek sınıflar bu soyut sınıflardan kalıtım alır. Benzer nesne üretimlerini bu şekilde kalıtımsal yöntemle üretiyor olmak performansı artıracaktır.

Arayüz/Soyut sınıf kullanılmasının en önemli sebebi bu yapıların referans tutucu görevi görebilmeleridir. Bu sayede farklı sınıfların nesnelerini tek bir arayüz/soyut sınıf değişken türü üzerinden oluşturabiliriz.

Cıktısı: Nesne Maliyetlerini Düşürür.

Factory Method Pattern Kullanım Adımları:

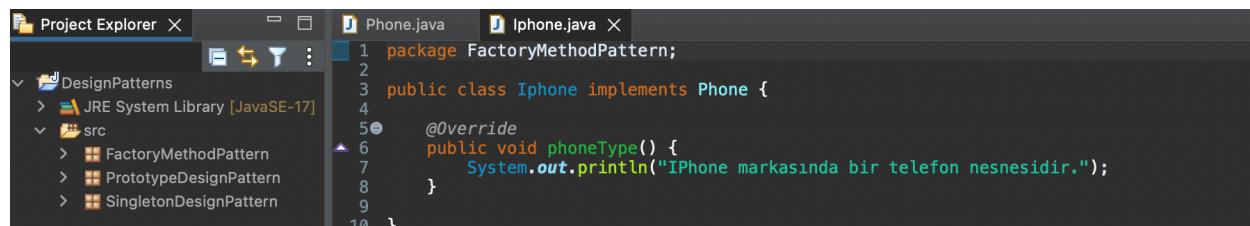
- Implemente edilecek/Kalıtım alınacak arayüz/soyut sınıf oluşturulur. Alt sınıflarda kesinlikle bulunması istenilen method veya elemanlar varsa bu yapı içerisinde tanımlanmalıdır.



The screenshot shows the Eclipse IDE's Project Explorer on the left and a code editor on the right. The Project Explorer lists a project named 'DesignPatterns' with a 'src' folder containing 'FactoryMethodPattern', 'PrototypeDesignPattern', and 'SingletonDesignPattern'. The code editor displays the 'Phone.java' file with the following content:

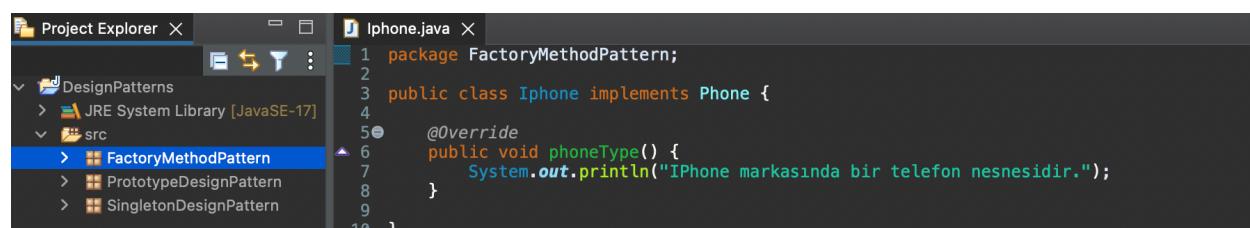
```
1 package FactoryMethodPattern;
2
3 public interface Phone {
4     void phoneType();
5 }
```

- İlgili alt sınıflar oluşturulur ve tanımlanan arayüzü implemente ederler.



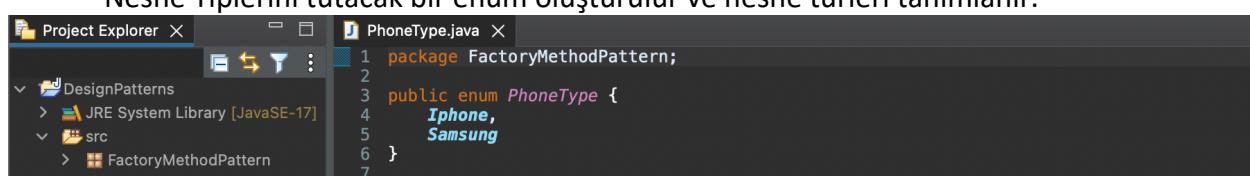
The screenshot shows the Eclipse IDE's Project Explorer on the left and a code editor on the right. The Project Explorer lists the same project structure as before. The code editor displays the 'Iphone.java' file with the following content:

```
1 package FactoryMethodPattern;
2
3 public class Iphone implements Phone {
4
5     @Override
6     public void phoneType() {
7         System.out.println("IPhone markasında bir telefon nesnesidir.");
8     }
9
10 }
```



The screenshot shows the Eclipse IDE's Project Explorer on the left and a code editor on the right. The Project Explorer lists the project structure, and the 'FactoryMethodPattern' folder is currently selected. The code editor displays the 'Iphone.java' file with the same content as the previous screenshot.

- Nesne Tiplerini tutacak bir enum oluşturulur ve nesne türleri tanımlanır.



The screenshot shows the Eclipse IDE's Project Explorer on the left and a code editor on the right. The Project Explorer lists the project structure, and the 'FactoryMethodPattern' folder is selected. The code editor displays the 'PhoneType.java' file with the following content:

```
1 package FactoryMethodPattern;
2
3 public enum PhoneType {
4     Iphone,
5     Samsung
6 }
```

- Nesnelerimizi ilgili seçime göre oluşturacak olan methodu içeren sınıf ve method tanımlanır. Method parametre olarak istenilen nesne türünü almalıdır.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer shows a project named 'DesignPatterns' with a 'src' folder containing 'FactoryMethodPattern', 'PrototypeDesignPattern', and 'SingletonDesignPattern'. On the right, the editor window displays the 'PhoneCreator.java' code:

```

1 package FactoryMethodPattern;
2
3 public class PhoneCreator {
4
5     public Phone factoryMethod(PhoneType phoneType) {
6         Phone phone=null;
7         switch(phoneType) {
8             case Iphone:
9                 phone=new Iphone();
10            break;
11             case Samsung:
12                 phone=new Samsung();
13                 break;
14         }
15     return phone;
16 }
17
18 }

```

- Son olarak nesne oluşturma methodunu içeren sınıfın nesnesi oluşturulur. Bu nesne üzerinden method'a istenilen ürün enum değeri belirtilir ve nesne oluşturulur.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer shows the same 'DesignPatterns' project structure. On the right, the editor window displays the 'Main.java' code:

```

1 package FactoryMethodPattern;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         PhoneCreator phoneCreator=new PhoneCreator();
7
8         Phone phone1=phoneCreator.factoryMethod(PhoneType.Iphone);
9         Phone phone2=phoneCreator.factoryMethod(PhoneType.Samsung);
10
11         phone1.phonetype();
12         phone2.phonetype();
13     }
14 }

```

The 'Console' tab at the bottom shows the output of the application's execution:

```

<terminated> Main (2) [Java Application] /Users/mariami/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86
IPhone markasında bir telefon nesnesidir.
Samsung markasında bir telefon nesnesidir.

```

4)Abstract Factory Pattern (Soyut Fabrika Tasarım Deseni)

Factory Method Pattern benzer nesneleri tek bir arayüz/soyut sınıf üzerinden tanımlar ve tek bir nesne üretici sınıf kullanılarak nesne üretir. Abstract Factory Pattern ise Factory Method Pattern'den farklı olarak benzer nesneleri üretmek için farklı nesne oluşturucu sınıflar ve farklı arayüzler/soyut sınıflar kullanır. Bu tasarım kalıbı çoğunlukla birden fazla ürün ailesi ile çalışmak zorunda olunduğunda kullanılır.

Çıktısı: Nesne maliyetlerini düşürür

Abstract Factory Pattern Kullanım Adımları:

- Üretilcek nesnelerin soyut sınıfları oluşturulur.

```

Project Explorer X
DesignPatterns
  JRE System Library [JavaSE-17]
  src
    Computer.java
      package AbstractFactoryDesignPattern;
      public interface Computer {
      }

    Phone.java
      package AbstractFactoryDesignPattern;
      public interface Phone {
      }
  
```

- Soyut sınıfları implemente edecek olan nesneleri üretilecek olan somut nesne sınıfları oluşturulur. Nesnelerin olduğunu konsol üzerinden gözlemleyebilmek adına yapıci methodlara string değerler girdik.

```

Project Explorer X
DesignPatterns
  JRE System Library [JavaSE-17]
  src
    AbstractFactoryDesignPattern
    FactoryMethodPattern
    PrototypeDesignPattern
    SingletonDesignPattern
    AppleComputer.java
      package AbstractFactoryDesignPattern;
      public class AppleComputer implements Computer{
        AppleComputer() {
          System.out.println("Apple Computer nesnesi oluşturuldu.");
        }
      }

    SamsungComputer.java
      package AbstractFactoryDesignPattern;
      public class SamsungComputer implements Computer {
        public SamsungComputer() {
          System.out.println("Samsung Computer nesnesi oluşturuldu.");
        }
      }

    ApplePhone.java
      package AbstractFactoryDesignPattern;
      public class ApplePhone implements Phone {
        ApplePhone() {
          System.out.println("Apple Phone nesnesi oluşturuldu.");
        }
      }

    SamsungPhone.java
      package AbstractFactoryDesignPattern;
      public class SamsungPhone implements Phone {
        SamsungPhone() {
          System.out.println("Samsung Phone nesnesi oluşturuldu.");
        }
      }
  
```

- Ürün ailelerini oluşturacak olan fabrika sınıflarının arayüzü oluşturulur.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays a package named 'DesignPatterns' containing several sub-packages: JRE System Library [JavaSE-17], src, AbstractFactoryDesignPattern, FactoryMethodPattern, PrototypeDesignPattern, and SingletonDesignPattern. On the right, the code editor shows the 'ProductFactory.java' file:

```
1 package AbstractFactoryDesignPattern;
2
3 public interface ProductFactory {
4     Phone createPhone();
5     Computer createComputer();
6 }
7
8
```

-Nesneleri üretecek olan ürün aile sınıfları oluşturulur. Ana fabrika arayüzüünü implemente ederler.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays the same 'DesignPatterns' project structure. On the right, two code editors are shown side-by-side: 'SamsungProduct.java' and 'AppleProduct.java'. Both files implement the 'ProductFactory' interface.

SamsungProduct.java:

```
1 package AbstractFactoryDesignPattern;
2
3 public class SamsungProduct implements ProductFactory{
4     @Override
5     public Phone createPhone() {
6         return new SamsungPhone();
7     }
8
9     @Override
10    public Computer createComputer() {
11        return new SamsungComputer();
12    }
13 }
14
```

AppleProduct.java:

```
1 package AbstractFactoryDesignPattern;
2
3 public class AppleProduct implements ProductFactory {
4     @Override
5     public Phone createPhone() {
6         return new ApplePhone();
7     }
8
9     @Override
10    public Computer createComputer() {
11        return new AppleComputer();
12    }
13 }
14
```

- Son olarak main methodu üzerinden ürün aile sınıfları kullanılarak istenilen nesneler oluşturulur.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays the 'DesignPatterns' project structure. On the right, the code editor shows the 'Main.java' file:

```
1 package AbstractFactoryDesignPattern;
2
3 public class Main {
4     public static void main(String[] args) {
5         ProductFactory appleProduct=new AppleProduct();
6         ProductFactory samsungProduct=new SamsungProduct();
7
8         appleProduct.createComputer();
9         appleProduct.createPhone();
10
11         samsungProduct.createComputer();
12         samsungProduct.createPhone();
13     }
14 }
15
```

Below the code editor, the 'Console' view shows the output of the program execution:

```
<terminated> Main (3) [Java Application] /Users/mariami/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx-x86_64
Apple Computer nesnesi oluşturuldu.
Apple Phone nesnesi oluşturuldu.
Samsung Computer nesnesi oluşturuldu.
Samsung Computer nesnesi oluşturuldu.
```

5)Builder Pattern (Oluşturucu Tasarım Deseni)

Birden fazla field içeren bir sınıfımız olduğunu düşünelim. Bu sınıfın nesnesini oluşturmak için yapıçı method yoluyla field parametrelerini göndermemiz gerekmektedir. Eğer nesne oluşturma sırasında bütün fieldlara atama yapmak istemiyorsak bütün kombinasyonları göz önüne alarak yapıçı methodu overload etmemiz gerekmektedir. Builder pattern bu soruna bir çözüm olarak sunulmaktadır. Tek bir yapıçı method yardımıyla o an ataması yapılmak istenilen fieldlar kullanılarak nesne oluşturulabilir.

Cıktısı: Kod karmaşıklığı asgari düzeye indirilir. Nesne üretim maliyeti düşürülür.

Builder Pattern Kullanım Adımları:

- Nesnesi oluşturulacak ana sınıfımız tanımlanır. İçerisine builder pattern’ı kullanmamızda yardımcı olacak olan static inner Builder sınıfı tanımlanır. Ana sınıfta bulunacak fieldlar bu sınıf içerisinde de tanımlanır.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer shows a package named 'DesignPatterns' containing 'src' and 'BuilderDesignPattern'. Inside 'src', there are 'AbstractFactoryDesignPattern' and 'BuilderDesignPattern'. On the right, the code editor displays 'Computer.java' with the following content:

```
1 package BuilderDesignPattern;
2
3 public class Computer{
4
5     public static class Builder{
6
7         private String ram,ssd,cpu;
8
9     }
10
11     public Builder ram(String ram) {
12         this.ram=ram;
13         return this;
14     }
15     public Builder ssd(String ssd) {
16         this.ssd=ssd;
17         return this;
18     }
19     public Builder cpu(String cpu) {
20         this.cpu=cpu;
21         return this;
22     }
23
24     public Computer build() {
25         return new Computer(this);
26     }
27 }
```

- Her field için dönüş tipi Build sınıfı olan ve parametre olarak ilgili field değerini alan methodlar tanımlanır. Metot içerisinde alınan parametre sınıfın field değerine atanır ve sonuç olarak oluşturulan nesnenin döndürülmesi sağlanır.

The screenshot shows the Eclipse IDE interface with the code editor focused on 'Computer.java'. The left sidebar lists other Java files: Computer.java, Main.java, FactoryMethodPattern, PrototypeDesignPattern, and SingletonDesignPattern. The code editor shows the implementation of the 'Builder' inner class:

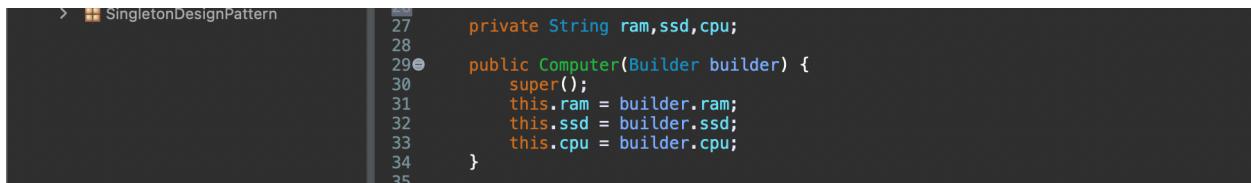
```
9     public Builder ram(String ram) {
10         this.ram=ram;
11         return this;
12     }
13     public Builder ssd(String ssd) {
14         this.ssd=ssd;
15         return this;
16     }
17     public Builder cpu(String cpu) {
18         this.cpu=cpu;
19         return this;
20     }
21
22     public Computer build() {
23         return new Computer(this);
24     }
25 }
```

- Son olarak Build sınıfı içerisinde nesnesini yaratmak istediğimiz ana sınıf tipinde referans döndüren build methodu tanımlanır. Bu method geriye parametre olarak yukarıda oluşturduğumuz builder nesnelerini alan yeni bir ana sınıf nesnesi döndürür.

The screenshot shows the Eclipse IDE interface with the code editor focused on 'Computer.java'. The code shows the final part of the 'Builder' class implementation:

```
21
22     public Computer build() {
23         return new Computer(this);
24     }
25 }
```

- Ana sınıf içerisine fieldlar tanımlanır. Ana sınıfın yapıcı methodu Builder nesnesi alacak şekilde oluşturulur. İçeriği builder nesnesinden gelecek field değerleri ana sınıfın field değerlerine atanacak şekilde düzenlenir.

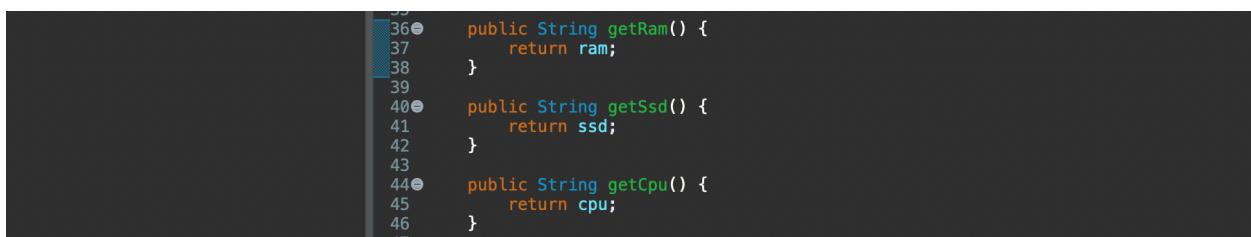


```

27     private String ram,ssd,cpu;
28
29●  public Computer(Builder builder) {
30     super();
31     this.ram = builder.ram;
32     this.ssd = builder.ssd;
33     this.cpu = builder.cpu;
34 }
35

```

- Ana sınıf içerisine fieldların get methodları yazılır. Set methodlarına ihtiyaç yoktur çünkü atamalar Build sınıfı nesnesi yardımı ile yapılır.

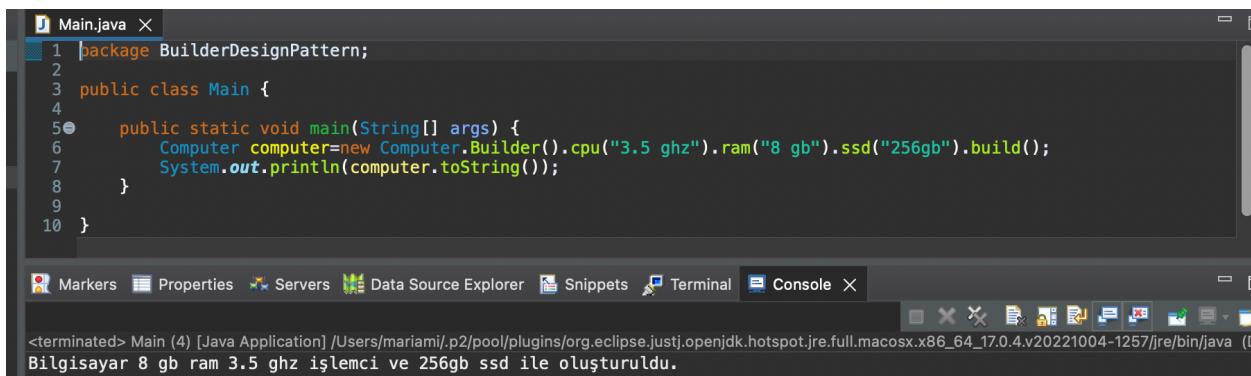


```

36●  public String getRam() {
37     return ram;
38 }
39
40●  public String getSsd() {
41     return ssd;
42 }
43
44●  public String getCpu() {
45     return cpu;
46 }
47

```

- Aşağıdaki şekilde nesne oluşturulurken verilmek istenilen field değerleri belirtilerek yeni nesne üretilir.



```

1 package BuilderDesignPattern;
2
3 public class Main {
4
5●   public static void main(String[] args) {
6     Computer computer=new Computer.Builder().cpu("3.5 ghz").ram("8 gb").ssd("256gb").build();
7     System.out.println(computer.toString());
8   }
9
10 }

```

Console output:

```

<terminated> Main (4) [Java Application] /Users/mariami/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.4.v20221004-1257/jre/bin/java (D
Bilgisayar 8 gb ram 3.5 ghz işlemci ve 256gb ssd ile oluşturuldu.

```