Creational Patterns

Creational Patterns, nesne yaratma mekanizmasıyla ilgilenen, uygulamamızda duruma uygun şekilde nesne yaratmaya çalışan tasarım kalıplarıdır. Yani uygulamada nesnelerin oluşturulmasından sorumludur. Yazılımımızda nesnelerin standart şekilde oluşturulması tasarım ve performans sorunlarına yol açabilir. Bunun için Singleton, Factory, Abstract Factory, Builder ve Prototype gibi yaratımsal tasarım desenleri geliştirilmiştir. Şimdi bunları inceleyelim.

Singleton Pattern

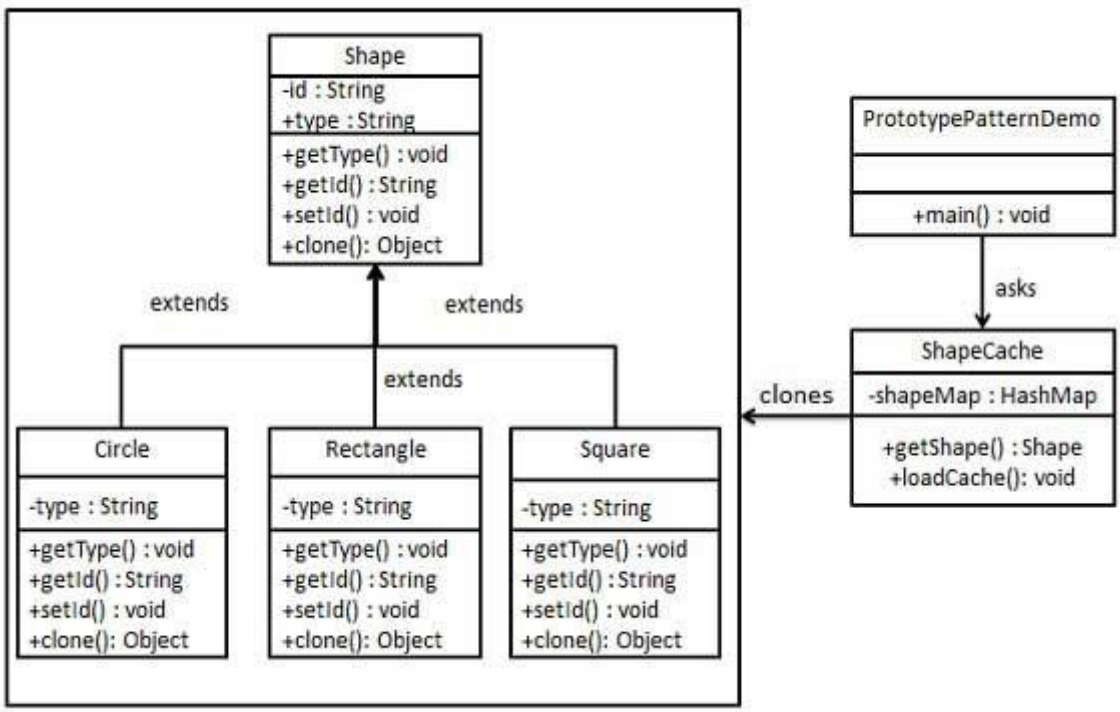
Singleton deseni, en basit tasarım modellerinden biridir. Bu desen, yalnızca tek bir nesnenin yaratıldığından emin olurken bir nesneyi oluşturmaktan sorumlu olan tek bir sınıfı içerir. Bu sınıf, sınıf nesnesinin somutlaştırılmasına gerek kalmadan doğrudan erişilebilen tek nesnesine erişmenin bir yolunu 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();
   }}
```

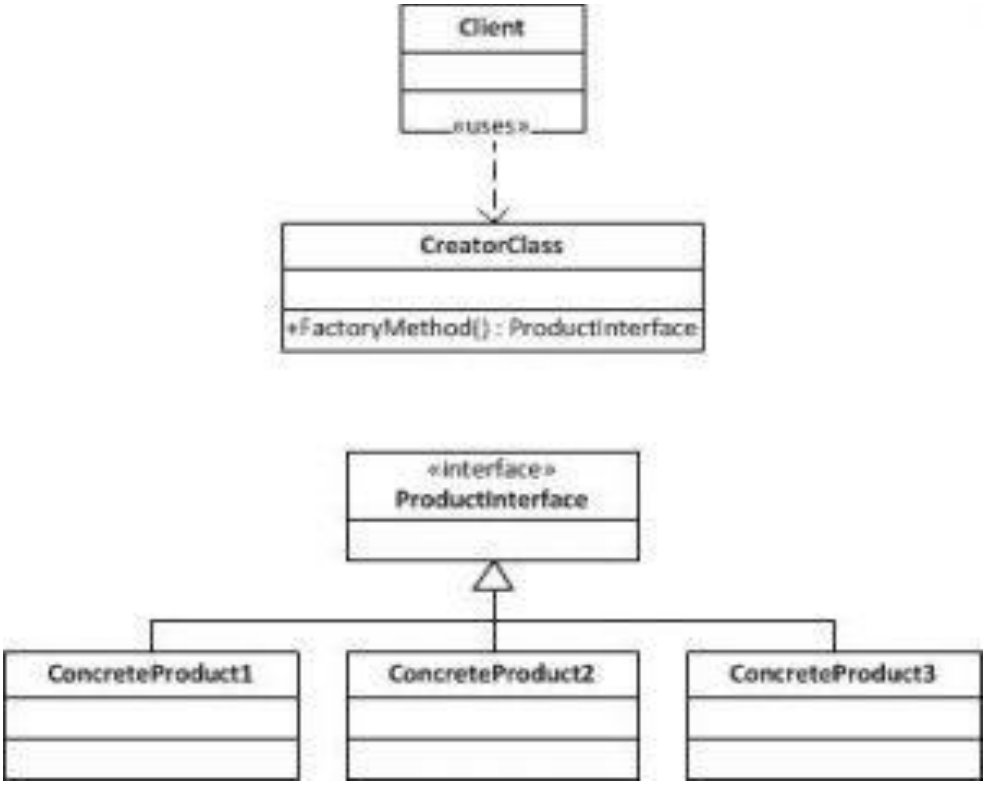
Prototype Pattern

Prototype Pattern, mevcut nesnenin bir klonunu oluşturmayı söyleyen bir prototip arayüzünün uygulanmasını içerir. Bu desenin, doğrudan nesne oluşturmanın pahalı olduğu durumlarda kullanılması önerilir. Örneğin, bir veritabanı işleminden sonra bir nesne oluşturulur ve bu nesne cachelenir(ön belleğe alınır). Daha sonraki isteklerde bu nesne üstünden clone oluşturulur. Veritabanı sorgularından kaynaklı yükü azaltmış oluruz.



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();
    }

}
```

Abstract Factory Pattern

Soyut Fabrika tasarım deseni, creational tasarım desenlerinden biridir. Bu tasarım deseni birbiriyle alakalı veya bağımlı nesnelerin somut sınıflarını belirtmeden, yaratılması için gereken bir arayüz sağlar. Ayrıca bu desene fabrikaların fabrikası(factories of the factory) da denir.

Ne zaman Kullanılır?

Product'ları yaratan fabrika sınıfından somut nesne yaratma işlemini çıkarmak istiyorsak, bu tasarım desenini kullanmak gereklidir. Ayrıca temel fabrika deseninde bulunan if-else yapısından da kurtulmak istiyorsanız soyut fabrika tasarım desenini kullanabilirsiniz.

Nasıl Kullanılır?

Her bir Product alt sınıfları için bir fabrika sınıfı oluşturmak gereklidir. Bu oluşturulacak fabrika sınıfları ise türü interface veya abstract olan bir süper fabrika sınıfından türümelidir.

Faydaları Nedir?

1. Client sınıfına, bir abstract arayüz kullanmasını sağlayarak, gerçekte üretilecek ilişkili Product sınıflarını bilmeden veya önemsemeden oluşturulmasına olanak tanır.
2. if-else yapısından kurtararak daha anlaşılır kod yazmayı sağlar.

```
/*
 * Bu bizim super sinifimiz. Dikkat edin interface olarak tanimlandi. Normal sinif veya soyut ta olabilirdi
 */
public interface Shape {
   public double getArea();
   public double getSize();
}
```

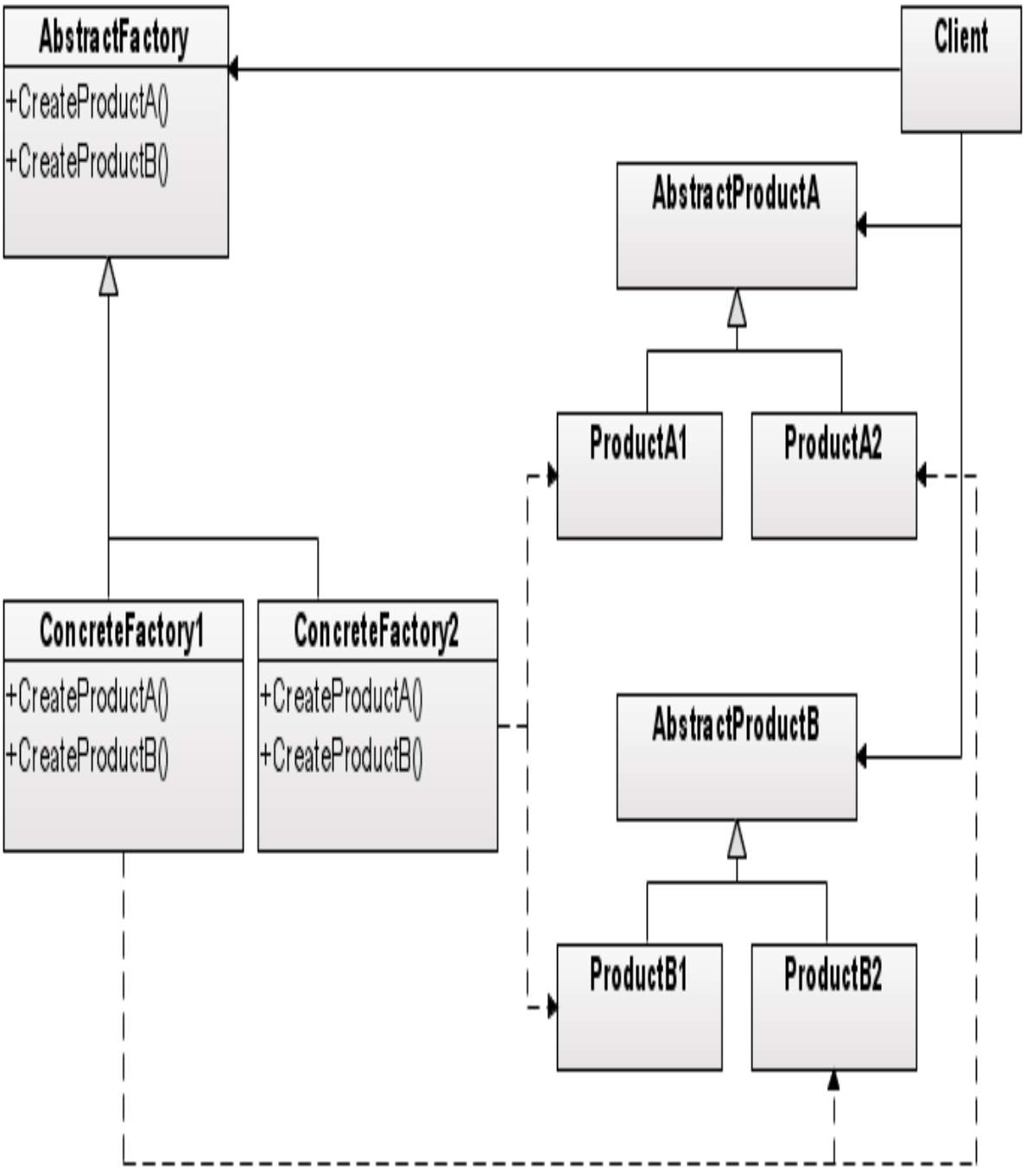
```
/**
```

```
 * Basit bir Circle sinifi
 */
public class Circle implements Shape {
    private double radius;

    @Override
    public double getArea() {
        return Math.PI*radius*radius;
    }

    @Override
    public double getSize() {
        return 2*Math.PI*radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
}




/**
```

```
 * Basit bir Rectangle sinifi
 */
public class Rectangle implements Shape {
    private double width;
    private double height;

    @Override
    public double getArea() {
        return width*height;
    }

    @Override
    public double getSize() {
        return 2*(width+height);
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }
}
/**
```

```
 * Abstract fabrika sinifimiz. Bu sinif süper fabrika sinifi
 */
public interface ShapeAbstractFactory {
   public Shape createShape();
}

/**
 * Circle sinifi icin fabrika sinifi
 */
public class CircleFactory implements ShapeAbstractFactory {
   @Override
   public Shape createShape() {
      return new Circle();
   }
}

/**
 * Rectangle sinifi icin fabrika sinifi
 */
public class RectangleFactory implements ShapeAbstractFactory {
   @Override
   public Shape createShape() {
      return new Rectangle();
   }
}



/**
```

```
 * Somut fabrika siniflarinin turune gore Shape nesneleri uretilmesini saglar
 */
public class ShapeFactory {
   public static Shape getShape(ShapeAbstractFactory factory){
      return factory.createShape();
   }
}



/**
 * Test sinifi.
 */
public class TestFactory {
   public static void main(String[] args) {
      Shape rectangle = ShapeFactory.getShape(new RectangleFactory());
      ((Rectangle) rectangle).setWidth(13);
      ((Rectangle) rectangle).setHeight(5);

      Shape circle = ShapeFactory.getShape(new CircleFactory());
      ((Circle) circle).setRadius(4);

      System.out.println("Rectangle area: "+rectangle.getArea()+" and size:
"+rectangle.getSize());
      System.out.println("Circle area: "+circle.getArea()+" and size: "+circle.getSize());
   }
}
```


AbstractFactory
+CreateProductA()
+CreateProductB()
Client
AbstractProductA
ProductA1
ProductA2
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
AbstractProductB
ProductB1
ProductB2

Builder Design Pattern Nedir

Birçok parametresi tanımlanmış bir sınıf düşünelim. Sınıflara dayalı bir uygulamada bu sınıfları farklı sınıflarda nesnelere dönüştürür ve kullanmaya başlarız. Bu nesneleri çağırırken aslında bir constructor yapısı oluştururuz. Ve bu constructor yapısında zorunlu olmayan parametreleri esnetebileceğimiz birbirinden farklı birçok constructor oluşturmamız gerekebilir. Bir örnek üzerinden gidecek olursak; Aşağıdaki gibi bir Person sınıfımız olsun.

```
@Getter
@Setter
public class Person {

    private Integer age;
    private String firstName;
    private String lastName;

    public Person(Integer age, String firstName, String lastName) {
        this.age = age;
        this.firstName = firstName;
        this.lastName = lastName;
    }

     public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Yukarıdaki sınıfımıza göre bir Person nesnesi yaratmak istediğimizde bize iki seçenek sunulmuştur. (age, firstName, lastName) veya (firstName, lastName) parametreleri alan iki constructordan birini tercih etmemiz gerekmektedir.

Bu durumda ya yeni constructor oluşturabilir yada argüman almayan bir constructor oluşturup tek tek değer atamak istediğimiz alanlarımızı set edebiliriz.

Person sınıfımıza Builder Pattern'i aşağıdaki gibi uygulayabiliriz. Burada aslında esnek bir constructor yapısı oluşturduk. Ve kod akışı olarak Person sınıfımıza ait static builder methodunu çağırarak nesnemizi oluşturmaktayız. Alanlarımızı bu nesne üzerinden set ederken son olarak bu nesnemize ait build methodu ile Person nesnemizi oluşturmaktayız.

```
@Getter
@Setter
public class Person {

   private Integer age;
   private String firstName;
   private String lastName;

   Person(Integer age, String firstName, String lastName) {
      this.age = age;
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public static PersonBuilder builder() {
      return new PersonBuilder();
   }
```

```
    public static class PersonBuilder {
        private Integer age;
        private String firstName;
        private String lastName;

        PersonBuilder() {
        }

        public PersonBuilder age(Integer age) {
            this.age = age;
            return this;
        }

        public PersonBuilder firstName(String firstName) {
            this.firstName = firstName;
            return this;
        }

        public PersonBuilder lastName(String lastName) {
            this.lastName = lastName;
            return this;
        }

        public Person build() {
            return new Person(age, firstName, lastName);
        }
    }
}
```