NESNE YÖNELİMLİ PROGRAMLAMA 2(Object Oriented Programming 2/OOP)

Öğr. Gör. Celil ÖZTÜRK
Marmara Üniversitesi
Teknik Bilimler Meslek Yüksekokulu

İçerik

- ✓ABSTRACT FACTORY TASARIM KALIBI
- ✓BUILDER TASARIM KALIBI
- ✓PROTOTYPE TASARIM KALIBI

Yaratımsal Kalıplar(**Creational Patterns**)

- Yaratımsal kalıplar, yazılım nesnelerinin nasıl yaratılacağı ile ilgilenen tasarım kalıplarıdır.
- Daha önceden belirlenen durumlara bağlı olarak, gerekli nesneleri yaratır.
- Uygulamada nesnelerin oluşturulmasından sorumlu yapılardır.
- Bu kalıplar nesneye yönelik programlanın en yaygın görevlerinden biri olan yazılım sistemindeki nesnelerin yaratılması hakkında yol göstermektedir.

Tasarım Kalıpları

Creational Patterns(Yaratımsal Kalıplar)

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

Abstract Factory(Soyut Fabrika) Kalıbı

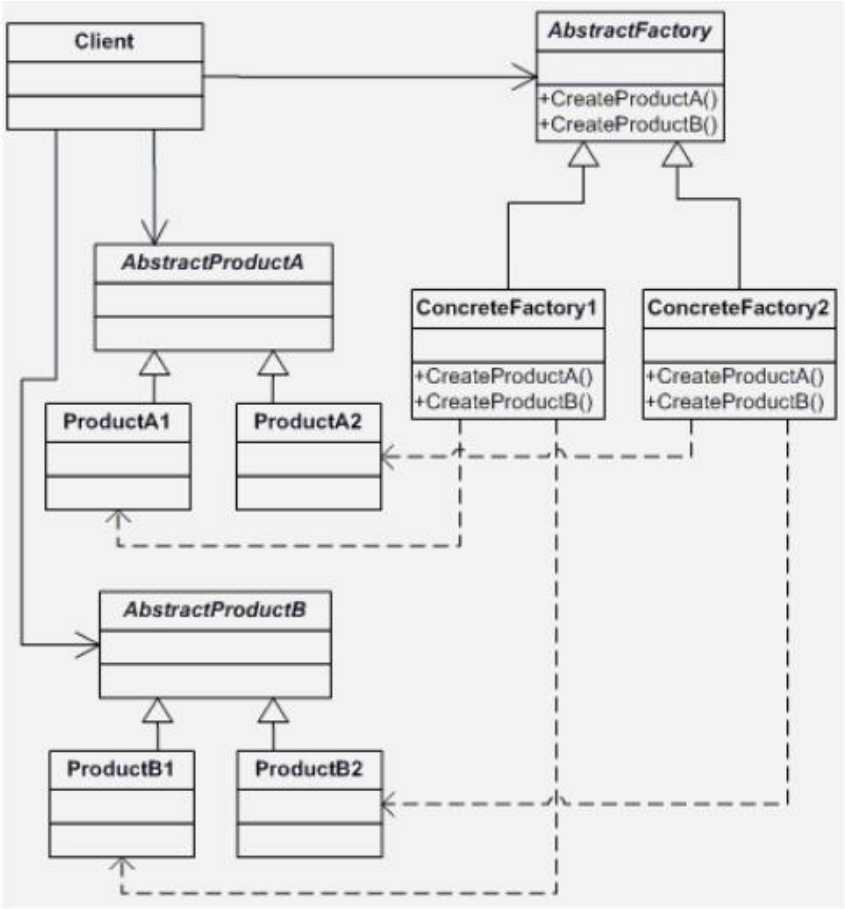
- Bu kalıp,birbirine benzer ürünlerin ortak bir ara katman üzerinden yönetilebilmesini sağlamaktadır.
- Kurulan bu yapı esneklik sağlar.
- Tek arayüz kullanarak bir nesne ailesinin farklı platformlarda yaratılmasını sağlar.
- Soyut fabrika tasarım kalıbının en belirgin özelliği, üretilecek nesnelerin birbirleri ile ilişkili olmasıdır.

Abstract Factory(Soyut Fabrika) Kalıbı

- Üretim sınıfında, üretimin yapılacağı fabrikanın hangi fabrika olduğu veya üretilen nesnelerin hangi tür olduğu ile ilgilenilmez. Soyutlama yapılmış olur.
- Aynı arayüz veya soyut sınıf kullanarak yeni nesneleri kalıba eklemek kolaylaşır.

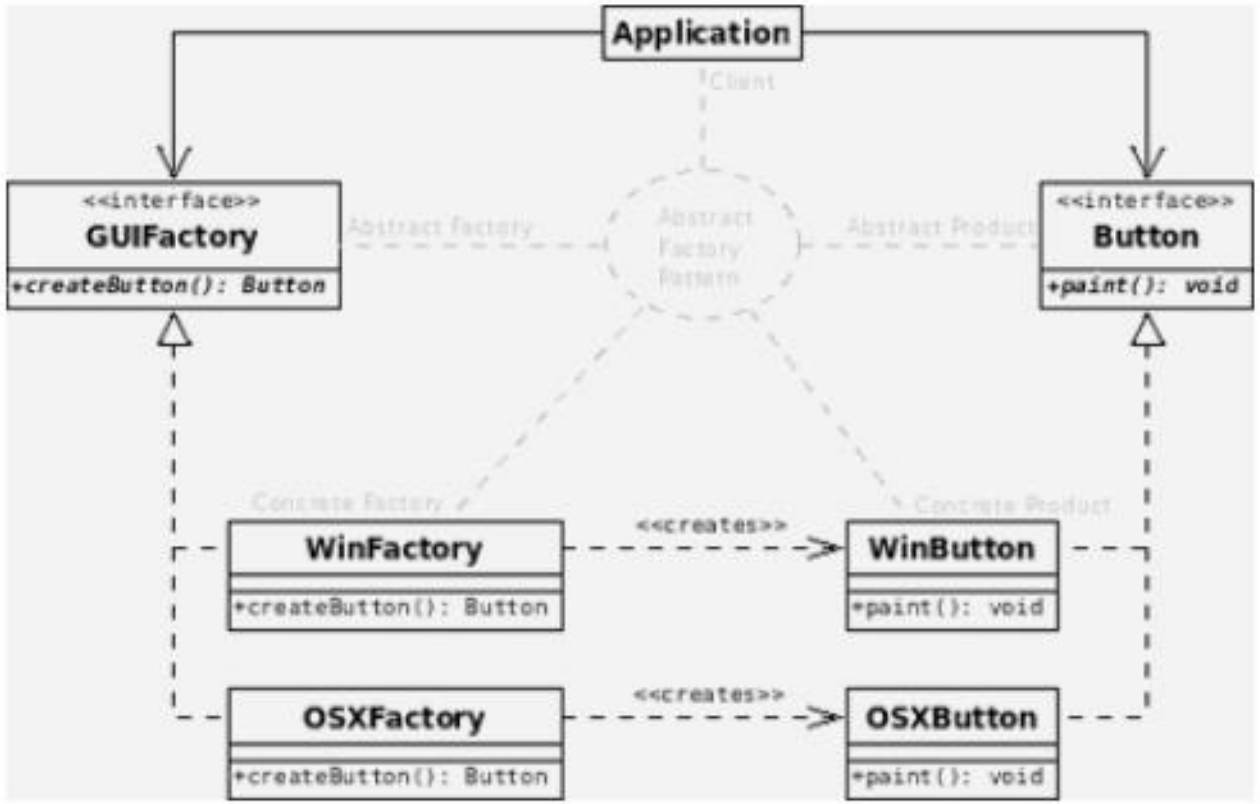
Abstract Factory(Soyut Fabrika) Kalıbı

Abstract Factory UML Gösterimi



https://bidb.itu.edu.tr/seyir-defteri/blog/2013/09/08/soyut-fabrika-tasar%C4%B1m-kal%C4%B1b%C4%B1-(abstract-factory-design-pattern)

Abstract Factory(Soyut Fabrika) Kalıbı



- GUI ve Button Arayüzleri tanımlanmıştır.
- Arayüzleri uygulayan(implement eden) WinFactory,OSXFctory ve WinButton, OSX button sınıfları oluşturulmuştur.
- Applicatin sınıfı yaratılmış ve belirtilen dosyadan yaratılacak fabrikanın hangi tip olacağı belirlenir.
- Programın ekran çıktısı fabrika tiğine göre Win yada OSX olacaktır.

Abstract Factory(Soyut Fabrika) Kalıbı

```
interface GUIFactory
{
    public Button createButton();
}
```

```
class WinFactory implements GUIFactory
{
    public Button createButton()
    {
        return new WinButton();
    }
}
```

```
class OSXFactory implements GUIFactory
{
    public Button createButton()
    {
        return new OSXButton();
    }
}
```

Abstract Factory(Soyut Fabrika) Kalıbı

```
interface Button
{
    public void paint();
}
```

```
class WinButton implements Button
{
    public void paint()
    {
        System.out.println("WinButton");
    }
}
```

```
class OSXButton implements Button
{
    public void paint()
    {
        System.out.println("OSXButton");
    }
}
```

Abstract Factory(Soyut Fabrika) Kalıbı

```
class Application
{
    public Application(GUIFactory factory)
    {
        Button button = factory.createButton();
        button.paint();
    }
}
```

- Application Runner sınıfında, belirtilen dosyadan yaratılacak fabrikanın tipi okunur ve kullanılacak fabrika belirlenir.

- Application sınıfının yapıcı metodunda, okunan fabrika tipine göre oluşturulacak buton, override edilmiş **createButton** metodu kullanılarak oluşturulur.

```
public class ApplicationRunner
{
    public static void main(String[] args)
    {
        new Application(createOsSpecificFactory());
    }

    public static GUIFactory
createOsSpecificFactory()     {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0)
        {
            return new WinFactory();
        }
        else
        {
            return new OSXFactory();
        }
    } }
```

Abstract Factory ve Factory Farkları

- Factory tasarım kalıbında tekbir ürün ailesine ait tek bir arayüz mevcutken, abstract factory'de farklı ürün aileleri için farklı arayüzler mevcuttur.
- Factory Tasarım Kalıbında, ilişkisel olan birden fazla nesnenin üretimini ortak bir arayüz aracılığıyla tek bir sınıf üzerinden yapılacak bir talep ile gerçekleştirmek ve nesne üretim anında istemcinin üretilen nesneye olan bağımlılığını sıfıra indirmeyi hedeflemektedir.
- Abstract Factory Tasarım Kalıbında, ilişkisel olan birden fazla nesnenin üretimini tek bir arayüz tarafından değil her ürün ailesi için farklı bir arayüz tanımlayarak sağlamaktadır.

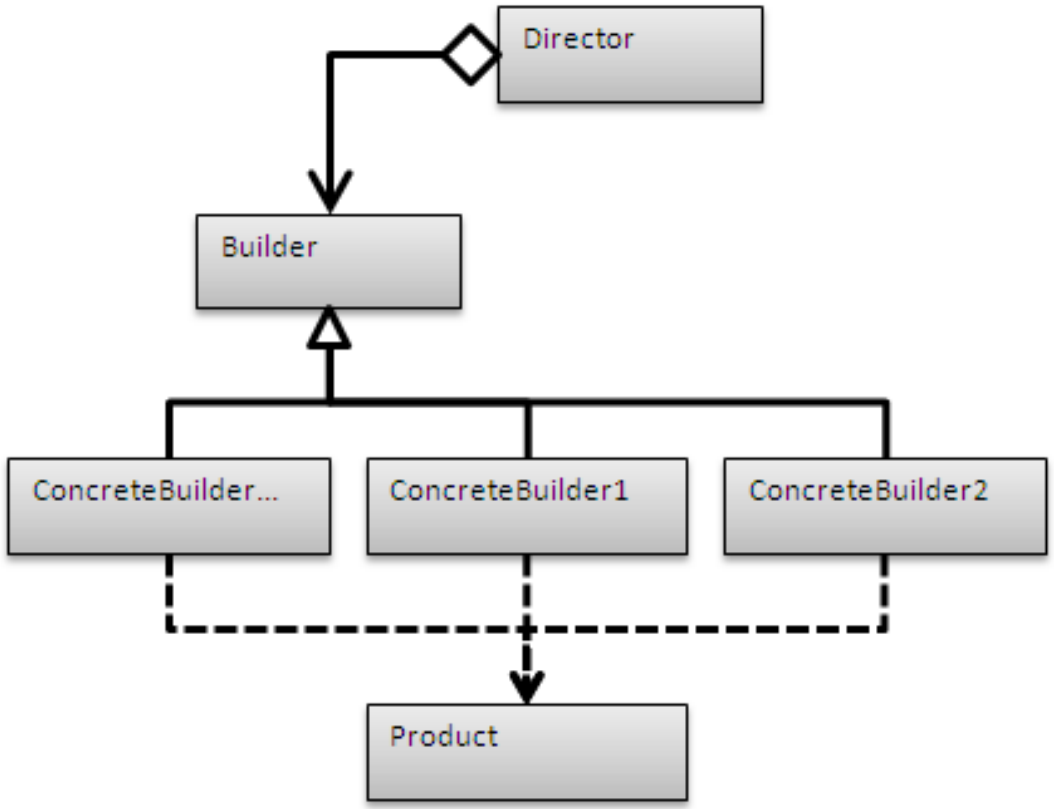
Builder Tasarım Kalıbı

- Builder tasarım kalıbı, karmaşık yapıdaki nesnelerin oluşturulmasında, istemcinin sadece nesne tipini belirterek üretimi gerçekleştirebilmesini sağlamak için kullanılır.
- İstemcinin kullanmak istediği gerçek ürünün birden fazla sunumunun olabileceği düşünülür. Farklı sunumların üretimi Builder adı verilen nesnelerin sorumluluğundadır.
- Builder ile, farklı ve karmaşık üretim süreçleri istemciden tamamen soyutlanabilir.

Builder Tasarım Kalıbı

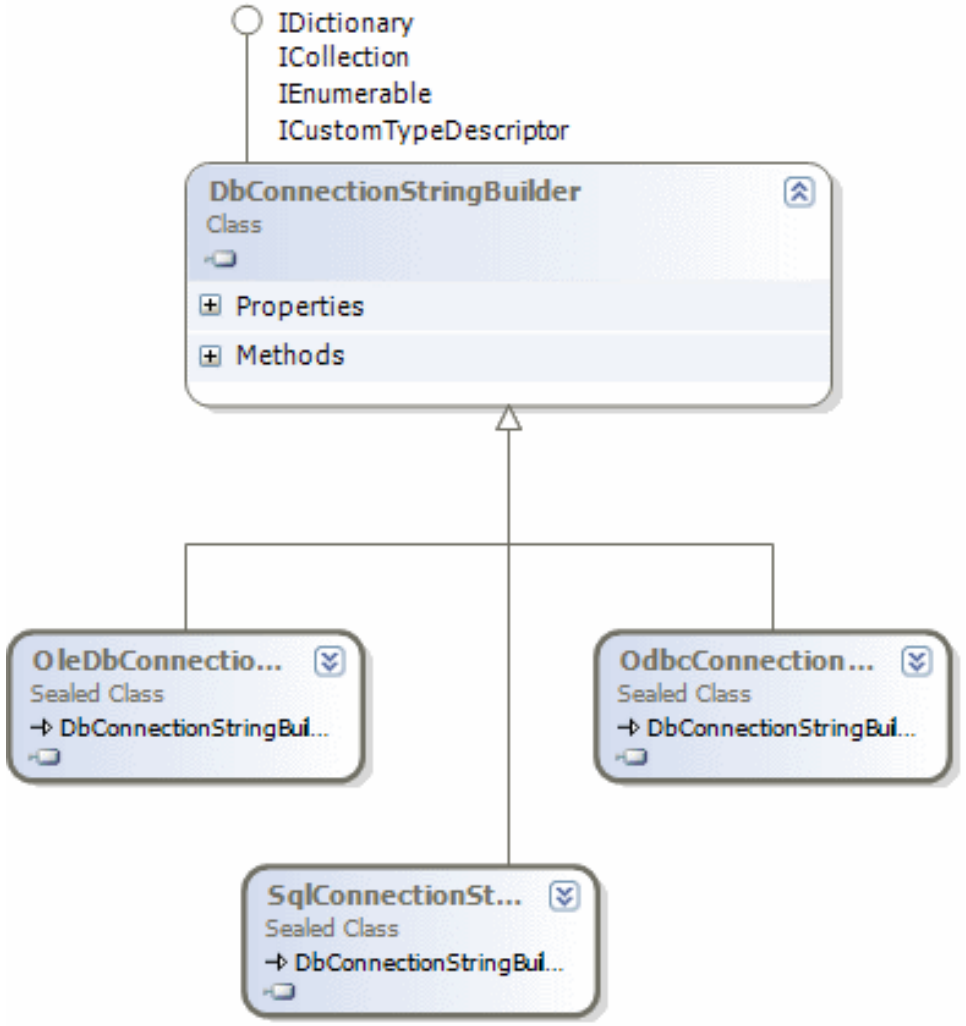
- **Builder: Product** nesnesinin oluşturulması için gerekli soyut arayüzü sunar.
- **ConcreteBuilder: Product** nesnesini oluşturur. Product ile ilişkili temel özellikleride tesis eder ve **Product'** ın elde edilebilmesi için(istemci tarafından) gerekli arayüzü sunar.
- **Director: Builder** arayüzünü kullanarak nesne örneklemesini yapar.
- **Product:** Üretim sonucu ortaya çıkan nesneyi temsil eder. Dahili yapısı(örneğin temel özellikleri) **ConcreteBuilder** tarafından inşa edilir.

Builder Tasarım Kalıbı



https://www.buraksenyurt.com/post/Tasarc4b1m-Desenleri-Builder

Builder Tasarım Kalıbı



https://www.buraksenyurt.com/post/Tasarc4b1m-Desenleri-Builder

Builder Tasarım Kalıbı/Örnek 1

```
// Product class
    public class Pizza
    {
        public string PizzaTipi { get; set; }
        public string Hamur { get; set; }
        public string Sos { get; set; }

        public override string ToString()
        {
            return String.Format("{0} {1} {2}", PizzaTipi, Hamur, Sos);
        }
    }


 // Builder class
    public abstract class PizzaBuilder
    {
        protected Pizza _pizza;

        public Pizza Pizza
        {
            get { return _pizza; }
        }

        public abstract void SosuHazirla();
        public abstract void HamuruHazirla();
    }
```

Builder Tasarım Kalıbı/Örnek 1

```
// ConcreteBuilder class
    public class BaharatliPizzaBuilder
        : PizzaBuilder
    {
        public BaharatliPizzaBuilder()
        {
            _pizza = new Pizza { PizzaTipi = "Baharatlı Baharatlı" };
        }
        public override void SosuHazirla()
        {
            _pizza.Sos = "Acı sos, pepperoni, atom biber";
        }

        public override void HamuruHazirla()
        {
            _pizza.Hamur = "İnce Kenar, Kaşarlı";
        }
    }
```

```
// Builder class
    public abstract class PizzaBuilder
    {
        protected Pizza _pizza;

        public Pizza Pizza
        {
            get { return _pizza; }
        }

        public abstract void SosuHazirla();
        public abstract void HamuruHazirla();
    }
```

Builder Tasarım Kalıbı/Örnek 1

```
// ConcreteBuilder Class
    public class DortMevsimPizzaBuilder
        : PizzaBuilder
    {
        public DortMevsimPizzaBuilder()
        {
            _pizza = new Pizza { PizzaTipi = "4 Mevsim" };
        }
        public override void SosuHazirla()
        {
            _pizza.Sos = "Biber, Domates, Peynir, Salam, Sosis";
        }

        public override void HamuruHazirla()
        {
            _pizza.Hamur = "Kalın, fesleğenli";
        }
    }
```

```
// Builder class
    public abstract class PizzaBuilder
    {
        protected Pizza _pizza;

        public Pizza Pizza
        {
            get { return _pizza; }
        }

        public abstract void SosuHazirla();
        public abstract void HamuruHazirla();
    }
```

Builder Tasarım Kalıbı/Örnek 1

```
// Director Class
    public class VedenikliKamil
    {
        public void Olustur(PizzaBuilder vBuilder)
        {
            vBuilder.SosuHazirla();
            vBuilder.HamuruHazirla();
        }
    }
```

- İstemcinin amacı bir tip pizza almaktır.
- İstemci ConcreteBuilder nesne örneğini seçerek üretimi gerçekleştirir.
- Nesne seçim işlemi Director sınıfı içindeki Oluştur metoduna parametre olarak gönderilir.
- Sonraki aşamada istemcinin istedğini pizza üretilerek elde edilir.

```
// Client class
    class Program
    {
        static void Main(string[] args)
        {
            PizzaBuilder vBuilder;

            VedenikliKamil kamil= new VedenikliKamil();
            vBuilder = new BaharatliPizzaBuilder();

            kamil.Olustur(vBuilder);
            Console.WriteLine(vBuilder.Pizza.ToString());

            vBuilder = new DortMevsimPizzaBuilder();
            kamil.Olustur(vBuilder);
            Console.WriteLine(vBuilder.Pizza.ToString());
        }
    }
```

Builder Tasarım Kalıbı/Örnek 2

- Sınıflar üzerinden nesneler yaratılır ve bunun için Constructorlar kullanılır.
- Sınıftaki field sayısı fazla olur ise birden fazla constructora ihtiyaç duyulabilir.
- Ad,soyad ve adres alanları olan bir sınıfta,

-bir nesne oluşturmak için 3 alanında olması gereklidir.

-sadece ad ve soyad olan alanları kullanan bir nesne yaratmak için yeni bir overload Constructor yazılması gerekir.

Yukarıda bahsedilen durumların çoğaldığı örneklerde field sayısına göre çok fazla Constructor yazmaya ihtiyaç duyulabilir. Bu durumların çözümünde ise **Builder DP** kullanılabilir.

Builder Tasarım Kalıbı/Örnek 2

```
public class Person {

private String name, surname, address;

public Person(Builder builder) {
this.name = builder.name;
this.surname = builder.surname;
this.address = builder.address;
}

public String getName() {
return name;
}

public String getSurname() {
return surname;
}

public String getAddress() {
return address;
}
```

```
public static class Builder{

   private String name, surname, address;

  public Builder(){ }

   public Builder name(String name){
        this.name = name;
       return this;
}

   public Builder surname(String surname){
       this.surname = surname;

        return this;
}

   public Builder address(String address){
      this.address = address;
      return this;
    }

  public Company build(){
        return new Company(this);
   }

} }
```

Builder Tasarım Kalıbı/Örnek 2

- Person person = new Person.Builder().name("Tuğrul").surname("Bayrak").address("Türkiye").build();

Prototype Tasarım Deseni

- Yazılım geliştirmede bellek ve çalışma zamanı gibi durumlar göz önüne alındığında nesneler yüksek maliyetli olabilir.
- Prototype tasarım kalıbı, mevcut nesnenin bir klonunu oluşturmayı söyleyen bir prototip arayüzünün uygulanmasını içerir ve yukarıdaki sorunların çözümü olarak kullanılabilir.
- Prototype tasarım kalıbı Örnek Nesne adıyla da kullanılır.

Prototype Tasarım Deseni

```
public class Uye implements Cloneable {

    private List<String> uyeListesi;

    public Uye() {
        uyeListesi = new ArrayList<String>();
    }
    public Uye(List<String> liste) {
        this.uyeListesi = liste;
    }

    public void uyeEkle() {

        uyeListesi.add("Burak");
        uyeListesi.add("Ahmet");
        uyeListesi.add("Mehmet");
    }
```

Devamı

```
    public List<String> getUyeListesi() {
        return uyeListesi;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        List<String> uyeListesi = new ArrayList<String>();
        for (String s : this.getUyeListesi()) {
            uyeListesi.add(s);
        }
        return new Uye(uyeListesi);
    }
}
```

Prototype Tasarım Deseni

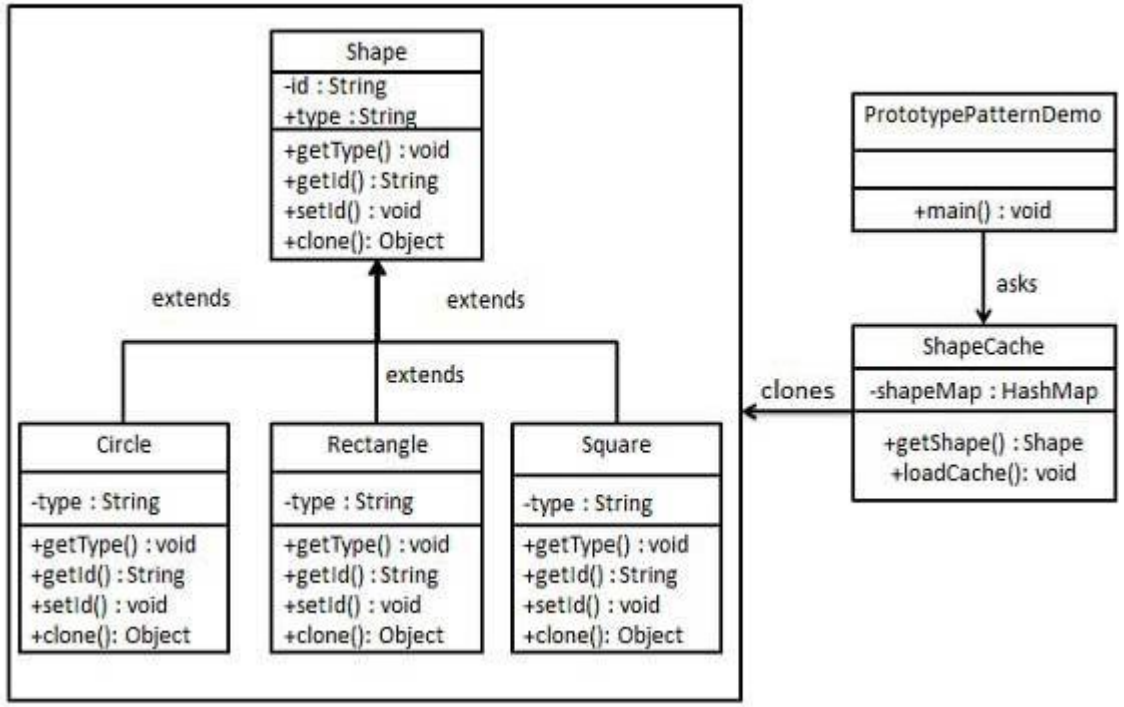
- Uye sınıfının klonlar oluşturularak çalışacağını Cloneable sınıfını implement ederek belirtiyoruz.
- Nesneleri klonlamak için Clone metodunu Overrride ediyoruz.
- Her yeni nesnede 3 eleman olan nesneyi klonluyoruz.

```
uyeler List: [Burak, Ahmet, Mehmet]
yeniUye List: [Burak, Ahmet, Mehmet, Ayşe]
yeniUye2 List: [Burak, Mehmet]
```

Ekran Çıktısı

```
public class App {

    public static void main(String[] args) throws
CloneNotSupportedException {

        Uye uyeler = new Uye();
        uyeler.uyeEkle();

        Uye yeniUye = (Uye) uyeler.clone();
        Uye yeniUye2 = (Uye) uyeler.clone();

        List<String> list = yeniUye.getUyeListesi();
        list.add("Ayşe");

        List<String> list1 = yeniUye2.getUyeListesi();
        list1.remove("Ahmet");

        System.out.println("uyeler List: " + uyeler.getUyeListesi());
        System.out.println("yeniUye List: " + list);
        System.out.println("yeniUye2 List: " + list1);
    }
}
```

Prototype Tasarım Deseni



tutorialspoint.com

Prototype Tasarım Deseni

```
public class Rectangle extends Shape {

public Rectangle(){
type = "Rectangle";
}

@Override
public void draw() {
System.out.println("Inside Rectangle::draw() method.");
}
}
```

```
import java.util.Hashtable; //*******!!

public abstract class Shape implements Cloneable {

private String id;
protected String type;

abstract void draw();

public String getType(){
return type;
}

public String getId() {
return id;
}
public void setId(String id) {

this.id = id;
}

public Object clone() {
Object clone = null;
try {

clone = super.clone();

} catch (CloneNotSupportedException e) {
e.printStackTrace();
}

return clone;
}
}
```

Prototype Tasarım Deseni

```
public class Circle extends Shape
public Circle(){
type = "Circle";
}

@Override
public void draw() {
System.out.println("Inside Circle::draw() method.");
}
}
```

```
public class Square extends Shape {

public Square(){
type = "Square";
}

@Override
public void draw() {
System.out.println("Inside Square::draw() method.");
}
}
```

Prototype Tasarım Deseni

```
public class ShapeCache {

private static Hashtable<String, Shape> shapeMap = new
Hashtable<String, Shape>();

public static Shape getShape(String shapeId) {
Shape cachedShape = shapeMap.get(shapeId);
return (Shape) cachedShape.clone();
}
```

```
public static void loadCache() {

Circle circle = new Circle();
circle.setId("1");
shapeMap.put(circle.getId(),circle);

Square square = new Square();
square.setId("2");
shapeMap.put(square.getId(),square);

Rectangle rectangle = new Rectangle();
rectangle.setId("3");
shapeMap.put(rectangle.getId(), rectangle);
}
```

Prototype Tasarım Deseni

```
public class PrototypePatternDemo {

public static void main(String[] args) {
ShapeCache.loadCache();
Shape clonedShape = (Shape) ShapeCache.getShape("1");

System.out.println("Shape : " + clonedShape.getType());
Shape clonedShape2 = (Shape)
ShapeCache.getShape("2");
System.out.println("Shape : " + clonedShape2.getType());

Shape clonedShape3 = (Shape)
ShapeCache.getShape("3");
System.out.println("Shape : " + clonedShape3.getType());

}
}
```

Kaynaklar

- Java ve Java Teknolojileri, *Tevfik KIZILÖREN* – Kodlab Yayınları
- Yazılım Mühendisliği CBU-Dr. Öğr. Üyesi Deniz Kılınç Yazılım Mimarisi ve Tasarımı Ders Notları
- http://cagataykiziltan.net/tr/tasarim-kaliplari-design-patterns/1-creational-tasarim-kaliplari/2364-2/
- https://bidb.itu.edu.tr/seyir-defteri/blog/2013/09/08/soyut-fabrika-tasar%C4%B1m-kal%C4%B1b%C4%B1-(abstract-factory-design-pattern)
- https://www.buraksenyurt.com/post/Tasarc4b1m-Desenleri-Builder
- https://tugrulbayrak.medium.com/builder-pattern-2f6fb1dbf4a0
- https://tugrulbayrak.medium.com/creational-patterns-singleton-prototype-beabbcabdde6
- https://yasinmemic.medium.com/prototype-design-pattern-nedir-37dc82983bef
- https://blog.burakkutbay.com/design-patterns-prototype-pattern-nedir.html/