CREATIONAL DESIGN PATTERNS

Yaratımsal Tasarım Kalıpları, oluşturulacak nesnenin nasıl oluşturulacağı ile ilgili daha önceki deneyimlere dayanılarak oluşturulan kalıplardır. Yanlış şekilde oluşturulan nesneler tasarım karmaşasına veya yazılımın büyümesi ile birlikte birtakım yönetim problemlerine sebebiyet verebilir.

Yazılımda nesnelerin standart bir şekilde oluşturulmasının doğuracağı bazı sonuçlardan yukarıda bahsettik. Şimdi oluşan bu problemleri çözmek amacıyla geliştirilen Singleton, Factory, Abstract Factory, Builder ve Prototype tasarım kalıplarına biraz daha yakından bakalım ve nasıl implemente edildiğini anlamaya çalışalım.

1- Singleton

Singleton tasarım kalıbı en basit desenlerden biridir. Bu desen bizlere aynı nesneyi tekrar tekrar oluşturmak yerine tek bir oluşturup yazılımın geriye kalanında aynı nesne üzerinden gerçekleme yapma imkanı tanır. Faydaları; daha az nesne yaratıp Ram tüketimini etkin bir biçimde kullanmak ve yönetimleri daha kolay olan nesneler ile yazılımın karmaşasını en aza indirgemek gibi kavramlar vardır.

Bu kalıp oluşturulurken bir Singleton sınıfı yaratılır. Sınıf içerisinde bize nesneyi döndüren static keywordu ile oluşturulmuş bir method ve yine static ile oluşturulmuş bir nesne bulunur. İlgili sınıftan nesne üretmek için kullanılan method eğer daha önce bu nesne yaratılmışsa o nesneyi return eder. Fakat daha önce yaratılmamış ise nesneyi yaratır ve sonucu return eder.

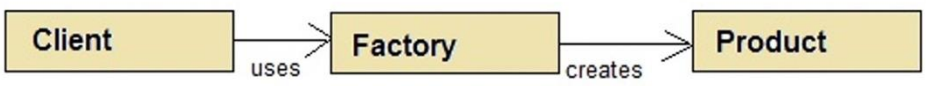
Aşağıdaki kod satırlarında Singleton nesnesi nasıl yaratılır inceleyelim

```
1   public class Singleton{
2
3   // Sınıfımızın Construcutor Methodu
4   private Singleton(){
5   }
6
7   // static değişkenimiz singleton class'ımızın instance'ı
8   private static Singleton singleton;
9
10  private static Singleton getInstance(){
11
12      If(singleton == null )
13          singleton = new Singleton();
14      return singleton;
15
16      }
17  }
18
```

2- Factory

Object Oriented Programming yaklaşımı ile günlük hayattan örnek vermek gerekirse, bazı basit nesnelerin üretim süreci kolay olmakta bunları evde kendimiz bile üretebiliriz. Fakat bazı nesnelerin üretim süreçleri daha karmaşık bir yapıya sahiptir. Bu nesnelerin üretimini bizim yerimize özel fabrikalar gerçekleştirir ve üretim aşamalarını biz kullanıcılardan gizleyerek bu süreci yönetirler.

Factory Pattern de tam olarak böyle bir yaklaşıma sahiptir. Oluşturulan nesneleri kullanacak sınıflar bu süreçten haberdar olmak detayları bilmek istemezler.



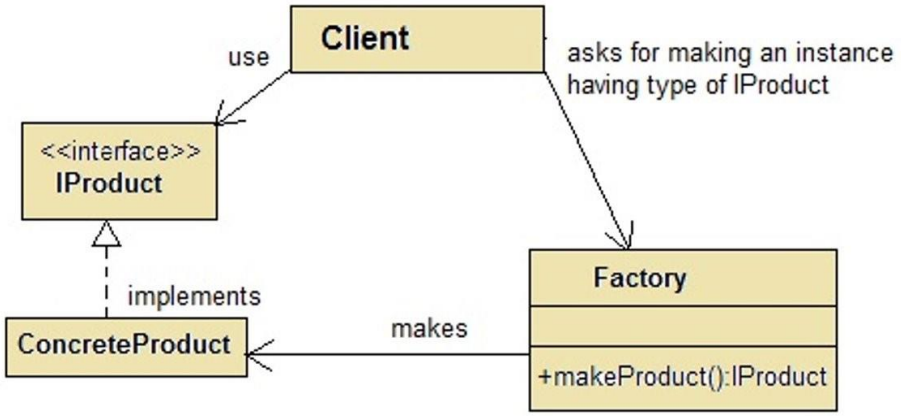
Client olan nesne Factory nesnesini kullanarak ihtiyacı olan Product nesnesini elde eder.

Nesne oluşturma sürecinin Factory sınıfına aktarılması ile birlikte Client sınıfı bu işlemden soyutlanmış olur. Bu sayede Client sadece kendi rolüne odaklanır.

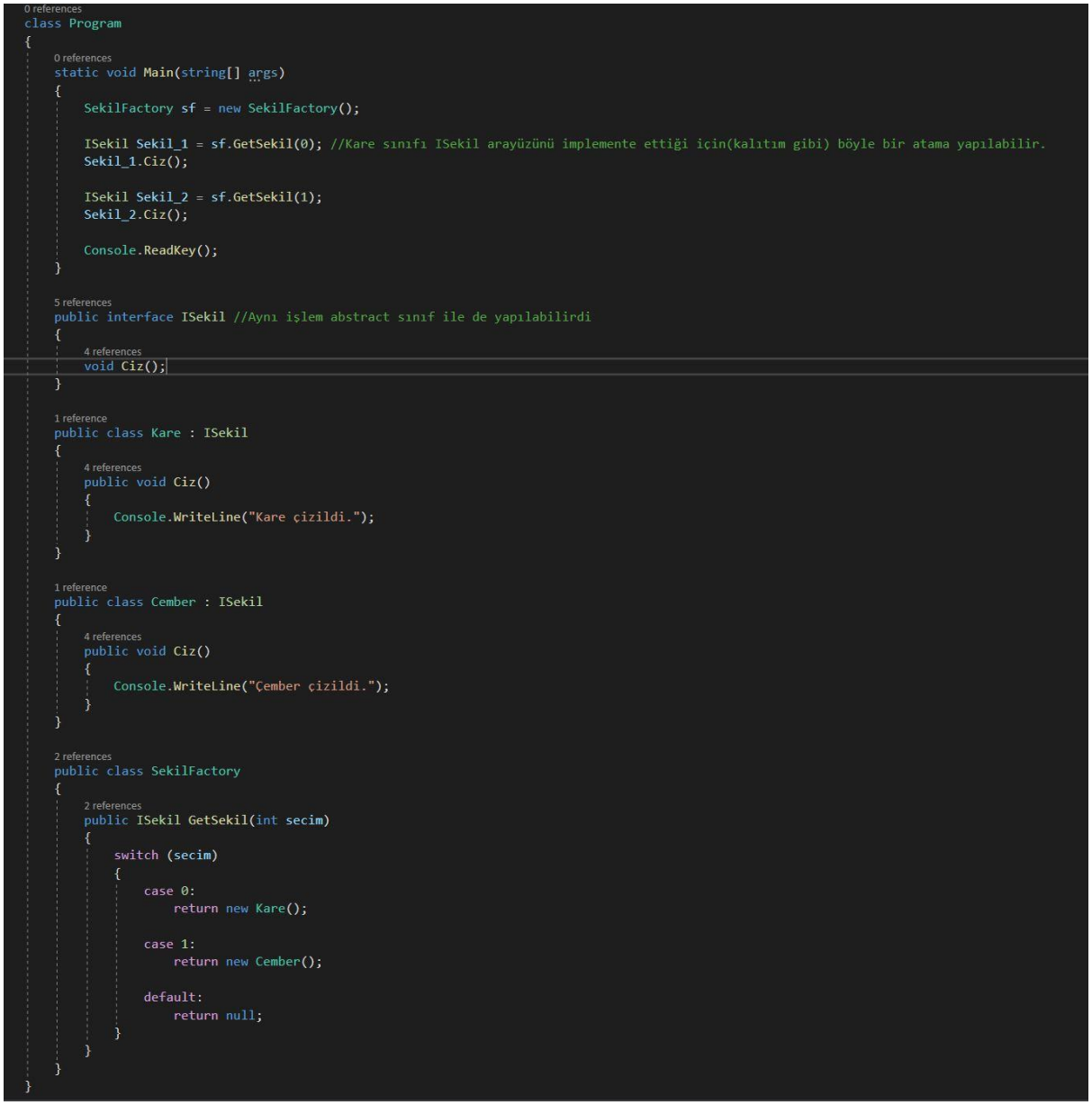
Factory Pattern genel olarak:

İstenen tipte nesne oluşturma sürecini Client'ın bu konuda detay bilgi sahibi olmadan gerçekleştirilmesini sağlar.

Yeni oluşturulan nesneye bir interface ile referans edilerek ulaşılmasını sağlar.



Factory design patern i bir de kod üzerinde görelim;



```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        SekilFactory sf = new SekilFactory();

        ISekil Sekil_1 = sf.GetSekil(0); //Kare sınıfı ISekil arayüzünü implemente ettiği için(kalıtım gibi) böyle bir atama yapılabilir.
        Sekil_1.Ciz();

        ISekil Sekil_2 = sf.GetSekil(1);
        Sekil_2.Ciz();

        Console.ReadKey();
    }

    5 references
    public interface ISekil //Aynı işlem abstract sınıf ile de yapılabilirdi
    {
        4 references
        void Ciz();
    }

    1 reference
    public class Kare : ISekil
    {
        4 references
        public void Ciz()
        {
            Console.WriteLine("Kare çizildi.");
        }
    }

    1 reference
    public class Cember : ISekil
    {
        4 references
        public void Ciz()
        {
            Console.WriteLine("Çember çizildi.");
        }
    }

    2 references
    public class SekilFactory
    {
        2 references
        public ISekil GetSekil(int secim)
        {
            switch (secim)
            {
                case 0:
                    return new Kare();

                case 1:
                    return new Cember();

                default:
                    return null;
            }
        }
    }
}
```

3- Abstract Factory

Bu tasarım kalıbından bahsederken Factory kalıbına deyinmemek olmaz. Factory kalıbında tek bir ürün üzerinde üretim sağlayan fabrika benzetmesi bulunurken, Abstract Factory kalıbını birçok ürün üreten bir fabrika olarak düşünebiliriz.

Factory design pattern’de tek bir ürün ailesine ait tek bir arayüz mevcutken,abstract factory’de farklı ürün aileleri için farklı arayüzler mevcuttur.

Şimdi kod tarafında bu deseni nasıl oluşturabiliriz inceleyelim.

```
class Program {
    static void Main(string[] args) {
        //ABD
        AbdAdresFactory af1 = new AbdAdresFactory();


        Adres adr1 = af1.AdresYarat();
        adr1.GetUlke();


        TelefonNumarasi tn1 = af1.TelefonNumarasiYarat();
        tn1.GetUlkeKodu();


        //HOLLANDA
        AbstractFactory af2 = new HollandaAdresFactory();


        Adres adr2 = af2.AdresYarat();
        adr2.GetUlke();


        TelefonNumarasi tn2 = af2.TelefonNumarasiYarat();
        tn2.GetUlkeKodu();


        //LIST COMPLETED
        Console.WriteLine("List completed");


        Console.ReadKey();
    }



    public abstract class AbstractFactory {
        public abstract Adres AdresYarat();
        public abstract TelefonNumarasi TelefonNumarasiYarat()
    }


    public class AbdAdresFactory: AbstractFactory {
        public override Adres AdresYarat() {
            return new AbdAdres();
        }

```

```
47
48          public override TelefonNumarasi TelefonNumarasiYarat() {
49              return new AbdTelefonNumarasi();
50          }
51      }
52
53
54      public class HollandaAdresFactory: AbstractFactory {
55          public override Adres AdresYarat() {
56              return new HollandaAdres();
57          }
58
59
60          public override TelefonNumarasi TelefonNumarasiYarat() {
61              return new HollandaTelefonNumarasi();
62          }
63      }
64
65
66      public abstract class Adres {
67          public abstract void GetUlke();
68      }
69
70
71      public class AbdAdres: Adres {
72          public override void GetUlke() {
73              Console.WriteLine("Amerika Birleşik Devletleri");
74          }
75      }
76
77
78      public class HollandaAdres: Adres {
79          public override void GetUlke() {
80              Console.WriteLine("Hollanda");
81          }
82      }
83
84
85      public abstract class TelefonNumarasi {
86          public abstract void GetUlkeKodu();
87      }
88
89
90      public class AbdTelefonNumarasi: TelefonNumarasi {
91          public override void GetUlkeKodu() {
92              Console.WriteLine("Ulke kodu:01");
93          }
94      }
95
96
97      public class HollandaTelefonNumarasi: TelefonNumarasi {
98          public override void GetUlkeKodu() {
99              Console.WriteLine("Ulke kodu:02");
100         }
101     }
102
```

4- Builder

Nesne yönelimli programlamada oluşturduğumuz classlara bir de constructor method yani kurucu metotlar yazarız. Kurucu metotlar classlar üretildiğinde kurucu metoduna gönderilen parametler ile oluşur.

Örnek bir kurucu metod şu şekilde tanımlanır.

```
1   public class Person {
2
3       private String name, surname, address;
4
5       public Person(String name, String surname, String address) {
6           this.name = name;
7           this.surname = surname;
8           this.address = address;
9       }
10  }
11
```

Kalıbın kullanılma sebebini açıklayalım. Classlar bazen büyüyebilir kurucu metotlarına başka parametler eklememiz gerekebilir veya class ilk kez yaratılırken hangi parametler gönderileceğini bilemeyiz. Tam da bu nokta imdadımıza Builder tasarım kalıbı yetişir.

Şimdi hep birlikte bir Builder kalıbı oluşturalım.

```java
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);
        }

    }
```

5- Prototype

Prototype tasarım deseni sayesinde önceden oluşturulmuş nesnelerin prototipini oluşturabiliyoruz. Bunun bizlere ne gibi bir faydası olacak gibi sorulara cevap olarak da şunu söyleyebiliriz; new keywordu ile aynı nesneyi üretmemiz bizler için maliyetli olabilir. Burada maliyetten kasıt, parametreli constructer vs. olabilir. Ram tüketimini etkin bir şekilde kullanmak için bize daha cazip gelecektir.

Şimdi hep birlikte bu tasarım desenine ait bir örnek verelim.

```java
abstract class Color implements Cloneable
{

    protected String colorName;

    abstract void addColor();

    public Object clone()
    {
        Object clone = null;
        try
        {
            clone = super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
        return clone;
    }
}

class blueColor extends Color
{
    public blueColor()
    {
        this.colorName = "blue";
    }

    @Override
    void addColor()
    {
        System.out.println("Blue color added");
    }

}

```

```
51
52  class ColorStore {
53
54      private static Map<String, Color> colorMap = new HashMap<String, Color>();
55
56      static
57      {
58          colorMap.put("blue", new blueColor());
59          colorMap.put("black", new blackColor());
60      }
61
62      public static Color getColor(String colorName)
63      {
64          return (Color) colorMap.get(colorName).clone();
65      }
66  }
67
68
69  // Driver class
70  class Prototype
71  {
72      public static void main (String[] args)
73      {
74          ColorStore.getColor("blue").addColor();
75          ColorStore.getColor("black").addColor();
76          ColorStore.getColor("black").addColor();
77          ColorStore.getColor("blue").addColor();
78      }
79  }
```