YZM 3017
Yazılım Tasarımı ve Mimarisi

Prof. Dr. Hamdi Tolga KAHRAMAN
Arş. Gör. Mustafa Hakan BOZKURT
Arş. Gör. Sefa ARAS

TASARIM DESENLERİ

1.TASARIM DESENLERİ NEDİR,NEDEN KULLANILIR?

- Tasarım desenleri temel olarak, tasarım prensiplerinin bir amaç için sınıf seviyesinde birlikte kullanılmasıdır. Yazılan sınıf ve bu sınıflardan oluşturulan nesnelerinin birbirleri ile olan ilişkileri kalıpları oluşturur.

- Desenler, yazılım geliştiricilerin zaman içerisinde deneme/yanılma, iyi/kötü tecrübelerinin sonucu olarak oluşmuş yazılım geliştirme teknikleridir. Yazılım geliştirilirken sıklıkla karşılaşılan problemlere özel olarak geliştirilen bu teknikler, yazılım tasarımının en önemli parçalarından birini oluşturmaktadır.

TASARIM DESENLERİ

1.TASARIM DESENLERİ NEDİR,NEDEN KULLANILIR?

- Desenler, kullanım alanlarına göre farklılık gösterseler de ortak bir takım özelliklere sahiptirler.
- Bunlar;
  - tekrar eden bir probleme karşı geliştirilmişlerdir,
  - tekrar kullanılabilirdirler,
  - problem çözümü için yeniden uğraştırmazlar ki bu yazılım dünyasında tekerleği yeniden icat etmeye gerek yok diye bilinir ve en önemlisi
  - uzun yıllar elde edilen tecrübeler sonucu ortaya çıkmalarıdır.

TASARIM DESENLERİ

Konunun tarihçesinden bir kaç cümle ile bahsetmek gerekirse bilgisayar bilimlerinde tasarım kalıplarının popularite ve ivme kazanması 1994 te "gang of four" olarak bilinen 4 yazarın (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) eseri olan *Design Patterns : Elements of Reusable Object-Oriented Software* isimli kitap ile başlamıştır. Değişen ihtiyaçlar ile birlikte bir çok farklı kalıplar zaman içerisinde ortaya çıkmıştır, var olanlara ihtiyaçlar dahilinde farklı bakış açıları getirilmiştir.

Genel olarak tasarım kalıpları:

- Creational Patterns (Oluşturucu Kalıplar)
- Structural Patterns (Yapısal Kalıplar)
- Behavioral Patterns (Davranişşal Kalıplar)

olarak ayrılır.

TASARIM DESENLERİ

Gerçek hayata yakın kod örnekleri ile pattern in pratik uygulaması da yapılmaya çalışılacak. Bu genel başlıklar altında yer alan pattern ler şunlardır.

- **<u>Creational Patterns (Oluşturucu Tasarım Kalıpları)</u>**

- Factory
- **Abstract Factory**
- **Builder**
- Prototype
- Object Pool
- **Singleton**

TASARIM DESENLERİ

- **<u>Structural Patterns (Yapısal Tasarım Kalıpları)</u>**

- **Adapter**
- Bridge
- Composite
- **Decorator**
- **Facade**
- Flyweight
- Front Controller
- Module
- Proxy

TASARIM DESENLERİ

Behavioral Patterns (Davranışsal Tasarım Kalıpları)

- **Chain of Responsibility**
- **Command**
- Mediator
- Memento
- Observer
- Strategy
- Template
- **Visitor**

TASARIM DESENLERİ

- Tasarım desenleri ile algoritmaları karıştırmamak gerekir.
- Algoritmalar bir hesaplama problemini veya bir işlem problemini çözerler.
- Tasarım desenleri ise yazılım tasarımı ile ilgilidir.
- Desenlerin ilgilendiği iki önemli konu vardır; tekrar kullanılabilirlik ve esneklik.

SİNGLETON TASARIM DESENİ

- Singleton(Tek Nesne) tasarım deseni, bir sınıfın tek bir örneğini oluşturmak için kullanılır.
- Amaç, oluşturulan nesneye global erişim noktası sağlamaktır.
- Oluşturulan tek örnek, tüm isteklere kendi üzerinden cevap verir.
- Sistem çalıştığı sürece ikinci bir örnek oluşturulamaz.

SİNGLETON TASARIM DESENİ

- Sınıf örneği new operatörü ile alınamaz.
- Geliştirilen teknik ile sınıf ilk çağrıldığında statik örnek oluşturulur ve bundan sonraki çağrımlarda tüm istemcilere aynı örnek gönderilir.
- Burada dikkat edilmesi gereken nokta, çok istemcili sistemlerde ilk örnek oluşturulurken güvenli örnekleme yapılması ve oluşturulacak objenin kilitlenmesidir.

SİNGLETON TASARIM DESENİ

- Kilitleme yapılmaz ve güvenli hale getirilmezse elimizde birden fazla örnek olmuş olur.

- Bunun sebebi iki farklı thread'in birbirine çok yakın zamanlarda singleton sınıfını çağırması ve işlemci mimarisi gereği önce biri sonra diğeri çağrıldığında iki thread birbirinin sonucuna ulaşmadan yeni örneklendirme yapmasıdır.

SİNGLETON TASARIM DESENİ

- Kilitleme(lock) yapılırsa ilk örnek alındığında, ikinci bir thread istek yapsa bile obje kilitli olacağından diğer thread'in işinin bitmesini bekler.
- İşlem bitiminden sonra, istek yapıldığından, oluşturulmuş örnek geri gönderilir.
- Sisteminiz çoklu istemciye hizmet vermeyecekse bu önlemi almanıza gerek yoktur.

SİNGLETON TASARIM DESENİ

Singleton deseni aşağıdaki durumlarda ve koşullarla kullanılabilir.

- Bir sınıftan sadece bir nesne örneğinin olması öngörüldüğünde (Örn: Database erişim işlemleri gibi tekrar tekrar nesne oluşturulmaması gereken durumlarda)
- Sınıftan nesne oluşturması masraflı olduğu durumlarda kullanılır.
- Singleton uygulanacak sınıfın global bir erişim seviyesine sahip olması gerekir ve uygulamanın diğer tüm bileşenleri bu sınıfa erişilebilmelidir.
- Singleton olması istenen sınıfın yapıcı methodu parametresiz olması gerekir.

Singleton Tasarım Deseni

- Singleton deseni bir programın yaşam süresince belirli bir nesneden sadece bir örneğinin(instance) olmasını garantiler.
- Aynı zamanda bu desen, yaratılan tek nesneye ilgili sınıfın dışından global düzeyde mutlaka erişilmesini hedefler.
- Örneğin bir veritabanı uygulaması geliştirdiğinizi düşünelim.
- Her programcı mutlaka belli bir anda sadece bir bağlantı nesnesinin olmasını isteyecektir.
- Böylece her gerektiğinde yeni bir bağlantı nesnesi yaratmaktansa varolan bağlantı nesnesi kullanılarak sistem kaynaklarının daha efektif bir şekilde harcanması sağlanır.
- Bu örnekleri daha da artırmak mümkündür.
- Siz ne zaman belli bir anda ilgili sınıfın bir örneğine ihtiyaç duyarsanız bu deseni kullanabilirsiniz.

Singleton Tasarım Deseni

- Peki bu işlemi nasıl yapacağız.?
- Nasıl olacakta bir sınıftan sadece ve sadece bir nesne yaratılması garanti altına alınacak?
- Çözüm gerçekten de basit : statik üye elemanlarını kullanarak.

Singleton Tasarım Deseni

- Singleton tasarım desenine geçmeden önce sınıflar ve nesneler ile ilgili temel bilgilerimizi hatırlayalım.

- Hatırlayacağınız üzere bir sınıftan yeni bir nesne oluşturmak için yapıcı metot(constructor) kullanılır.

- Yapıcı metotlar C# dilinde new anahtar sözcüğü kullanılarak aşağıdaki gibi çağrılabilmektedir.

- **Sınıf nesne = new Sınıf();**

Singleton Tasarım Deseni

Sınıf nesne = new Sınıf();

- Bu şekilde yeni bir nesne oluşturmak için new anahtar sözcüğünün temsil ettiği yapıcı metoduna dışarıdan erişimin olması gerekir. Yani yapıcı metodun public olarak bildirilmiş olması gerekir.

- Ancak "Singleton" desenine göre belirli bir anda sadece bir nesne olabileceği için new anahtar sözcüğünün ilgili sınıf için yasaklanması gerekir yani yapıcı metodun protected yada private olarak bildirilmesi gerekir.

Singleton Tasarım Deseni

- Eğer bir metodun varsayılan yapıcı metodu(default constructor-parametresiz yapıcı metot) public olarak bildirilmemişse ilgili sınıf türünden herhangi bir nesnenin sınıfın dışında tanımlanması mümkün değildir.

Sınıf nesne = new Sınıf();

- Ancak bizim isteğimiz yalnızca bir nesnenin yaratılması olduğuna göre ilgili sınıfın içinde bir yerde nesnenin oluşturulması gerekir.

- Bunu elbette statik bir özellik (property) yada statik bir metotla yapacağız.

- Bu statik metot sınıfın kendi içinde yaratılan nesneyi geri dönüş değeri olarak bize gönderecektir.

Singleton Tasarım Deseni

- **Sınıf nesne = new Sınıf();**
- Peki bu nesne nerde ve ne zaman yaratılacaktır?

- Bu nesne statik metodun yada özelliğin içinde yaratılıp yine sınıfın private olan elemanına atanır.

- Tekil olarak yaratılan bu nesne her istendiğinde eğer nesne zaten yaratılmışsa bu private olan elemanın referansına geri dönmek yada nesneyi yaratıp bu private değişkene atamak gerekmektedir.

Singleton Deseninin 1. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new SingletonDeseni();

    private SingletonDeseni()
    {

    }

    public static SingletonDeseni Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

- Yukarıdaki sınıf örneğinde SingletonDeseni sınıfı belleğe yüklendiği anda statik olan SingletonDeseni nesnesi yaratılacaktır.
- Bu nesne yaratılışının new anahtar sözcüğü ile yapıldığına dikkat edin.
- Eğer siz Main() gibi bir metodun içinden bu nesneyi yaratmaya kalksaydınız derleme aşamasında hata alırdınız.
- Çünkü public olan herhangi bir yapıcı metot bulunmamaktadır.

Singleton Deseninin 1. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new SingletonDeseni();

    private SingletonDeseni()
    {

    }

    public static SingletonDeseni Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Ayrıca Main() gibi bir metodun içinden yaratılan bu nesneye

SingletonDeseni nesne = SingletonDeseni.Nesne;

şeklinde erişmeniz mümkündür. Böylece yukarıdaki deyimi her kullandığınızda size geri dönen nesne, sınıfın belleğe ilk yüklendiğinde yaratılan nesne olduğu garanti altına alınmış oldu. Dikkat etmeniz gereken diğer bir nokta ise nesneyi geri döndüren özelliğin yalnızca get bloğunun olmasıdır. Böylece bir kez yaratılan nesne harici bir kaynak tarafından hiç bir şekilde değiştirilemeyecektir.

Yazılım Tasarımı ve Mimarisi

```
public class SingletonDeseni
{
    private static  SingletonDeseni  nesne = new  SingletonDeseni();

    private SingletonDeseni()
    {

    }

    public  static  SingletonDeseni  Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Yukarıdaki sınıf örneğinde SingletonDeseni sınıfı belleğe yüklendiği anda statik olan SingletonDeseni nesnesi yaratılacaktır. Bu nesne yaratılışının new anahtar sözcüğü ile yapıldığına dikkat ediniz. Eğer Main() gibi bir metodun içinden bu nesneyi yaratmaya kalkılırsa derleme aşamasında hata alınır. Çünkü public olan herhangi bir yapıcı metot bulunmamaktadır. Önceki kodda

SingletonDeseni nesne = SingletonDeseni.Nesne();

şeklinde istenilen nesne zaten yaratılmış durumda olmaktadır. Oysa bu sınıfı daha efektif bir hale getirerek yaratılacak nesnenin ancak biz onu istediğimizde yaratılması sağlanabilir. Bu durumu uygulayan Singleton deseninin 2. versiyonunu aşağıda görebilirsiniz.

Singleton Deseninin 2. versiyonu

- Yukarıdaki SingletonDeseni sınıfını aşağıdaki gibi de yazmamız mümkündür.

- Dikkat ederseniz iki sınıfın tek farkı oluşturulan nesneye erişme biçimidir. İlk versiyonda nesneye özellik üzerinden erişilirken ikinci versiyonda metot üzerinden erişilmektedir. Değişmeyen tek nokta ise her iki erişim aracının da statik olmasıdır.

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new Singleton();

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        return nesne;
    }
}
```

Singleton Tasarım Deseni 1. ve 2. versiyonlar

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new SingletonDeseni();

    private SingletonDeseni()
    {

    }

    public static SingletonDeseni Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new Singleton();

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        return nesne;
    }
}
```

Yukarıdaki her iki versiyonda da biz yaratılan nesneyi

SingletonDeseni nesne = SingletonDeseni.Nesne;

yada

SingletonDeseni nesne = SingletonDeseni.Nesne();

şeklinde istediğimizde nesne zaten yaratılmış durumda olmaktadır. Oysa bu sınıfı daha efektif bir hale getirerek yaratılacak nesnenin ancak biz onu istediğimizde yaratılmasını sağlayabiliriz. Bu durumu uygulayan Singleton deseninin 3 versiyonunu olarak aşağıda görebilirsiniz.

Singleton Deseninin 3. versiyonu

- Yukarıdaki SingletonDeseni sınıfını aşağıdaki gibi de yazmamız mümkündür.
- Gördüğünüz üzere nesne ilk olarak sınıf belleğe yüklendiğinde değil de o nesneyi ilk defa kullanmak istediğimizde yaratılıyor. İlgili nesneyi her istediğimizde yeni bir nesnenin yaratılmaması içinde

```
if(nesne == null)
```

şeklinde bir koşul altında nesnenin yaratıldığına dikkat edin.

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne;

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        if(nesne == null)

         nesne = new SingletonDeseni();

         return nesne;
    }
}
```

Not : 3.versiyonda nesneyi yaratan bir metot olabileceği gibi 1. versiyondaki gibi sadece get bloğu olan özellikte olabilir.

Singleton Deseninin 4. versiyonu

- Her şeye rağmen 3 versiyonda bazı durumlar için tek bir nesnenin oluşmasını garanti etmemiş olabiliriz.
- Eğer çok kanallı(multi-thread) bir uygulama geliştiriyorsanız farklı kanalların aynı nesneyi tekrar yaratması olasıdır. Ancak eğer çok kanallı çalışmıyorsanız(çoğunlukla tek thread ile çalışırız) yukarıdaki sade ama öz olan 3 versiyondan birini kullanabilirsiniz.

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne;

    private static Object kanalKontrol = new Object;

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        if(nesne == null)
        {
            lock(kanalKontrol)
            {
                 if(nesne == null)
                {
                        nesne = new SingletonDeseni();
                }
            }
        }

         return nesne;
    }
}
```

Singleton Deseninin 4. versiyonu

- Ama eğer çok kanallı programlama modeli söz konusu ise farklı kanalların aynı nesneden tekrar yaratmasını engellemek için ekstra kontroller yapmanız gerekmektedir.
- bu yapacağımız kontrol performansı büyük ölçüde düşürmektedir.
- O halde çok kanallı uygulamalarda kullanabileceğimiz Singleton deseni:

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne;

    private static Object kanalKontrol = new Object;

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        if(nesne == null)
        {
            lock(kanalKontrol)
            {
                 if(nesne == null)
                {
                        nesne = new SingletonDeseni();
                }
            }
        }

         return nesne;
    }
}
```

- Lock özellikle çok kanallı programlama esnasında kullanılır.
- Lock içerisindeki işlemler bitmeden diğer işlemler yapılmaz.
- Thread uygulamalarında lock ile belirtilen işlemler önce yapılır diğerleri sıraya sokulur.
- Lock içerisindeki işlem bitmeden tekrardan Lock çalıştırılamaz.

Singleton Deseninin 4. versiyonu

- desendeki püf nokta lock anahtar sözcüğünün kullanımıdır.

- Eğer nesne ilk defa yaratılacaksa yani daha önceden nesne null değere sahipse lock anahtar sözcüğü ile işaretlenen blok kitlenerek başka kanalların bu bloğa erişmesi engellenir.

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne;

    private static Object kanalKontrol = new Object;

    private SingletonDeseni()
    {
    }

    public static Singleton Nesne()
    {
       if(nesne == null)
       {
           lock(kanalKontrol)
           {
                if(nesne == null)
               {
                      nesne = new SingletonDeseni();
               }
           }
       }

        return nesne;
    }
}
```

- Böylece kilitleme işlemi bittiğinde nesne yaratılmış olacağı için, kilidin kalkmasını bekleyen diğer kanal lock bloğuna girmiş olsa bile bu bloktaki ikinci if kontrolü nesnenin yeniden oluşturulmasını engelleyecektir.

- Böylece çok kanallı uygulamalar içinde tek bir nesnenin oluşmasını ve bu nesneye erişimi garanti altına alan Singleton desenini tasarlamış olduk.

Singleton Deseninin 5. versiyonu

- Son olarak lock anahtar sözcüğünü kullanmadan çok kanallı uygulamalar içinde tek bir nesneyi garanti altına alacak deseni yazalım. Aşağıda Singleton desenin 5. versiyonu bulunmaktadır.

Singleton Deseninin 5. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni  nesne = new SingletonDeseni ();

    private static SingletonDeseni()
    {

    }

    private SingletonDeseni()
    {

    }

    public static SingletonDeseni  Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

- Bu versiyonun birinci versiyondan tek farkı yapıcı metodunda statik olmasıdır. C# dilinde statik yapıcı metotlar bir uygulama domeninde ancak ve ancak bir nesne yaratıldığında yada statik bir üye eleman referans edildiğinde bir defaya mahsus olmak üzere çalıştırılır. Yani yukarıdaki versiyonda farklı kanalların(thread) birden fazla SingletonDeseni nesnesi yaratması imkansızdır. Çünkü static üye elemanlar ancak ve ancak bir defa çalıştırılır.

Singleton Deseninin 5. versiyonu

- Son versiyon basit ve kullanışlı görünmesine rağmen kullanımının bazı sakıncaları vardır. Örneğin Nesne Özelliği dışında herhangi bir statik üye elemanınız var ise ve ilk olarak bu statik üye elemanını kullanıyorsanız siz istemediğiniz halde SingletonDeseni nesnesi yaratılacaktır.
- Zira yukarıda da dediğimiz gibi bir statik yapıcı metot herhangi bir statik üye elemanı kullanıldığı anda çalıştırılır.
- Diğer bir sakıncalı durumda birbirini çağıran statik yapıcı metotların çağrılması sırasında çelişkilerin oluşabileceğidir. Örneğin her static yapıcı metot ancak ve ancak bir defa çalıştırılır dedik.
- Eğer çalıştırılan bir static metot diğer bir statik metodu çağırıyor ve bu statik metotta ilkini çağırıyorsa bir çelişki olacaktır.

Kısacası eğer kodunuzun çelişki yaratmayacağından eminseniz 5. deseni kullanmanız doğru olacaktır. Eğer çok kanallı uygulama geliştiriyorsanız 4. versiyonu, çok kanallı uygulama geliştirmiyorsanız da 3. versiyonu kullanmanız tavsiye edilmektedir.

Singleton Deseni Örnek

- Mac OS X işletim sistemlerindeki Panel(dock) eklentisini düşünülsün.
- Bu eklentinin üzerinde belli başlı öğeler dizilir. Dizilen öğelere tıklanınca öğenin tipine göre yeni pencereler açılır ve işlemi gerçekleştirilir.
- Mac OS X' teki gibi bir panelin geliştirileceği düşünülsün. Her bir öğe için yeni bir panel oluşturmak çok maliyetli ve gereksiz olacaktır? Mantıklı olan; bir adet panel olmalı ve bu panelin üzerinde öğeler olmalı.
- Panelin, öğelere tıklanınca ne olduğunu bilmesine gerek yoktur. Ayrıca işletim sistemi kaynak yönetimi maliyeti bakımından tek bir panel için kaynak ayırmak, düşük maliyet olarak karlı olacaktır. Tüm bunlar birleştirildiğinde panel singleton nesnesi olacaktır. Örneği kod yardımıyla açıklamak gerekirse panel(dock) sınıfı şu şekilde olacaktır.

Singleton Deseni Örnek

```
1   namespace Singleton
2   {
3       public class Dock
4       {
5           %Statik değişken
6           private static Dock _dock;
7           %Kilitleme Lock Objesi
8           private static object lockObject = new Object();
9           %Panel üzerindeki masaüstü öğesi
10          private Finder finder;
11          %Yapıcı metodu gizli yaparak new ile örneklendirmeyi önlüyoruz.
12          private Dock(){}
13          %Sınıftan Örnek Alma Singleton Kalıbı Oluşturma
14          public static Dock Instance()
15          {
16              %Hiç örnek alınmadıysa
17              if(_dock==null)
18              {
19                  %Çoklu istemcili sistemler için kilitleme mekanizması
20                  lock(lockObject)
21                  {
22                      %Kilitlemeden sonra örnekleme kontrolü
23                      if(_dock==null)
24                      _dock=new Dock();
25                  }
26              }
```

Singleton Deseni Örnek

```
27              %Panel geri döndür
28              return _dock;
29          }
30          %Panel üzerine tıklama
31          public void ClickDock()
32          {
33              Console.Writeline("Panel'e tıklandı.");
34              %Örneğimiz tek bir öğe üzerinden olduğu için bu şekilde yaptık.
35              %Tıklanan masaüstü öğesi iseyeni masaüstü öğesi oluştur.
36                  finder = new Finder();
37                  Console.Writeline("Finder:"+finder.Title);
38                  }
39          }
40      }
```

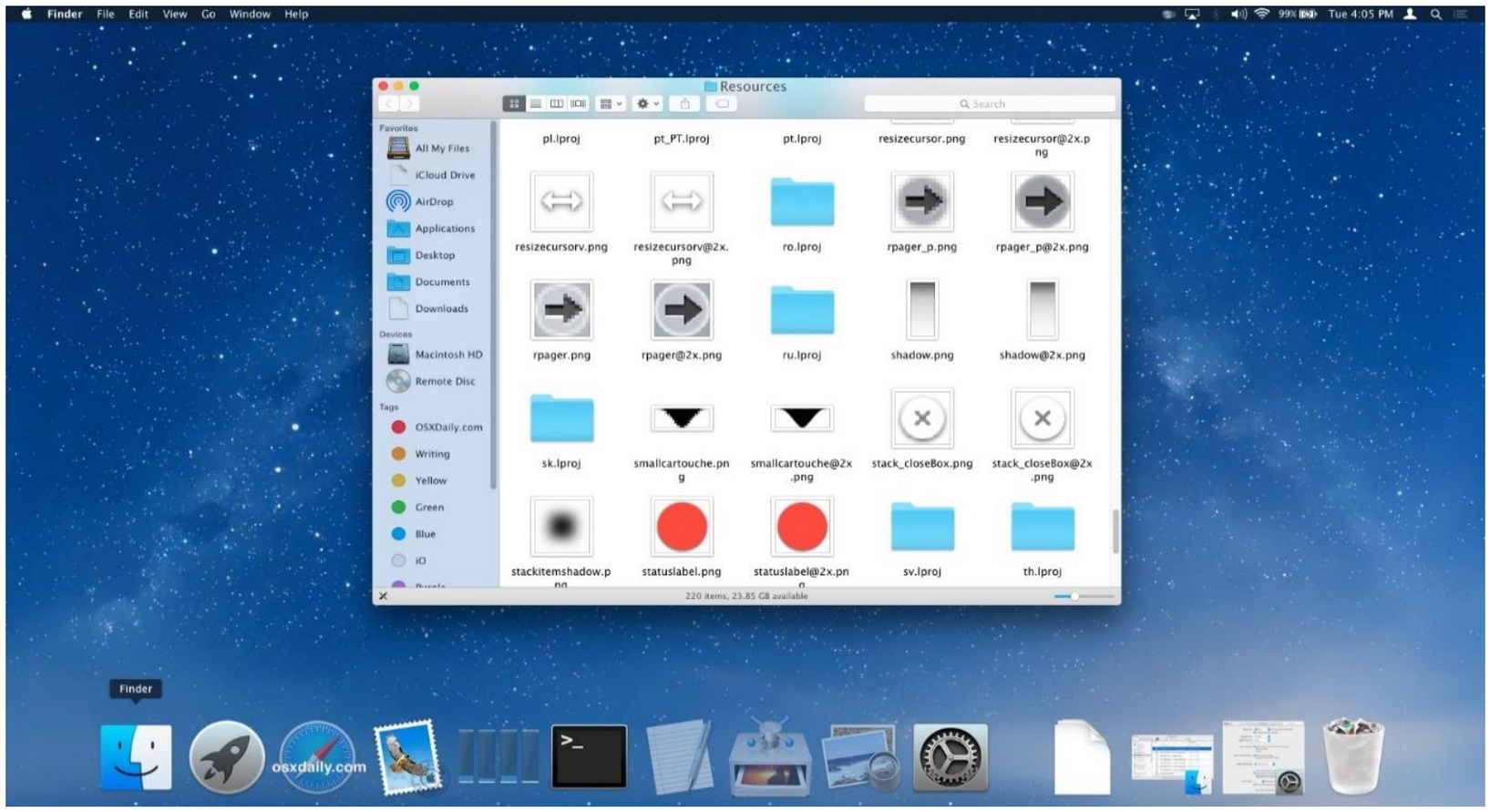
Singleton Deseni Örnek

Singleton panel sınıfı yorum satırları ile açıklandı. Panel'e tıklandığında finder öğesinin bir örneği oluşturulur. Bu sınıfın içeriği aşağıdadır. Başta da söylenildiği gibi panel sınıfı finder sınıfının ne yaptığı ile ilgilenmez ve bilmez.

```
1  -    namespace Singleton
2  -    {
3           public class Finder
4           {
5           %Pencere menü yazısı
6           public string Title{ get; set; }
7           %İşletim sistemi pencere katmanı simülasyonu
8  -        private WindowSystem _winSys = new WindowSystem();
9           %Yapıcı Metod
10 -        public Finder()
11 -        {
12              %Yeni oluşturulan pencereye pencere çizimi sonucunda başlık atanır.
13              this.Title=this._winSys.WindowDraw();
14          }
15      }
16
17      }
```

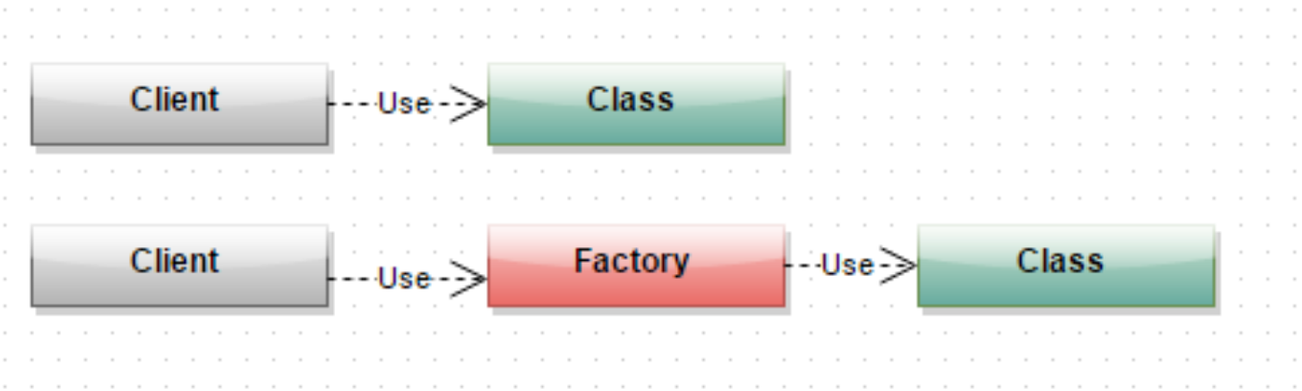
Singleton Deseni Örnek

Görüldüğü gibi, Singleton sınıfı üzerinden alt sistemler çağrıldı. Çağırılan alt sistemler başka alt sistemleri de çağırabilir. Kod incelenirse masaüstü öğesi(finder), pencere oluşturabilmek için işletim sistemi fonksiyonlarına ihtiyaç duyar ve onları çağırır. Tüm bu işlemleri yöneten sınıf örneği, singleton deseni ile bir tane olmuş olup kontrolü merkezi bir yerden sağlanmıştır.



FACTORY TASARIM DESENİ

Yaratımsal (Oluşturucu) kategoride yer alan diğer tasarım deseni factory desenidir. Aynı arayüze veya soyut sınıfı uygulayan sınıfların nesnelerinin oluşturulmasından sorumludur. Bu sınıflardan nesne oluşturma işlemi sınıfın uyguladığı **arayüz veya soyut sınıf** aracılığıyla fabrika sınıfları tarafından oluşturulur.



Yazılım Tasarımı ve Mimarisi

Temel sınıf nesnesini/nesnelerini oluşturan bir factory (fabrika) sınıf vardır. Nesneler temel sınıflar üzerinden değil, factory sınıf üzerinden türetilir.

Client code (istemci kod, bir diğer deyişle sınıfı kullanacak olan kod), factory sınıfa nesne türetme isteği yapar. Factory sınıf ise istemciye uygun bir nesne döndürür. Factory sınıfın temel görevi, istemcinin istediği nesneyi hangi sınıftan nasıl türeteceğini ondan soyutlamaktır.

Factory deseni ile bir sınıfın nesne oluşturma ve fonksiyonellikleri birbirinden ayrılmış olur. İstemci kod doğrudan asıl sınıfın fonksiyonelliğine erişemez.

Asıl sınıfın fonksiyonellikerine erişmek için factory methodu veya sınıfı aracılığıyla asıl sınıfından bir instance oluşturur ve bu şeklide fonksiyonelliklere erişebilmiş olur.

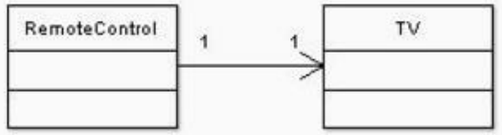
İstemci ve işi yapacak olan sınıf fonksiyonellikleri birbirinden ayrıldığı için gevşek bağımlılık sağlanmış olur.

Yazılım Tasarımı ve Mimarisi

Neden Factory?

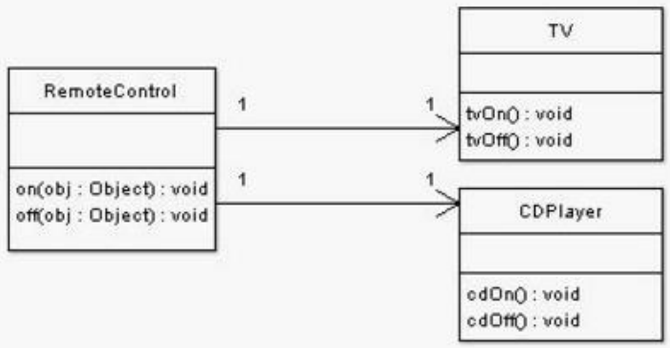
1. Factory tasarım deseni, nesnelerin yaratılma sürecini istemci sınıftan soyutlamak amacıyla oluşturulmuştur.
2. Fabrika sınıfı sayesinde alt sınıfların nesnelerin yaratılması ve bu nesneler üzerinden üye elemanların (işlevlerin) kullanılma karmaşıklığından istemci sınıfı soyutlanır.
3. Fabrika sınıfı özellikle dinamik yapıda olan ve değişime çok fazla gerek duyulan sınıfların yaratılmasında gereklidir.
4. Bir sınıf eklemekten ziyade varolan sınıfın değişmesi, değişimin sınıfın dışarıya açılan yüzü üzerinde çok fazla yaşanması durumlarında fabrika tasarım deseni faydalı olur.
5. Fabrika sınıfı nesnelerin yaratılmasından sorumlu olduğu için nesne modelleri olan sınıfların tasarımında oluşabilecek değişimler fabrika sınıflarında yapılacak değişimler ile sağlanmış/karşılanmış olur. Bu sayede değişim talebi ilgili sınıf ve fabrika sınıfı ile sınırlı tutulmaya çalışılır.

Yazılım Tasarımı ve Mimarisi

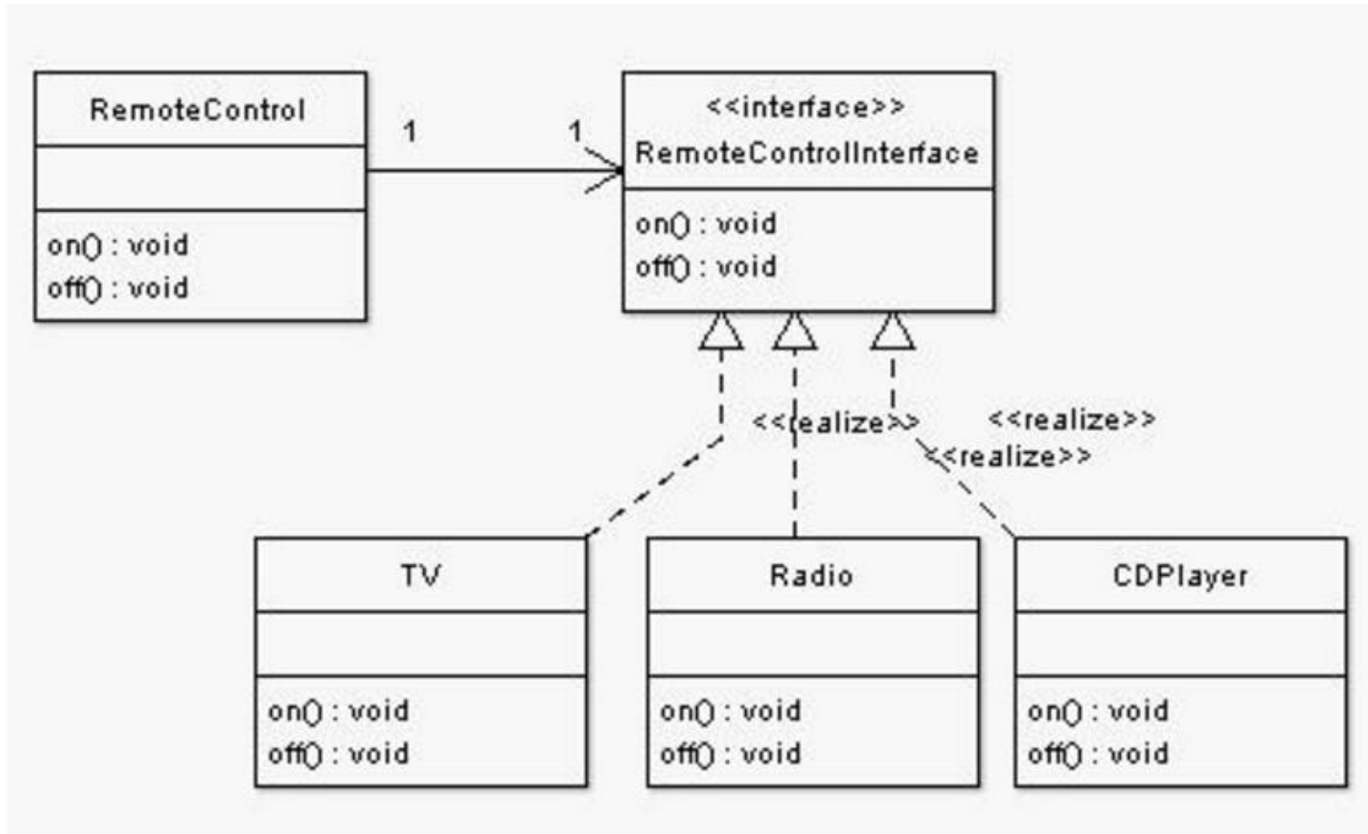


Resim 1 RemoteControl ve TV sınıfları

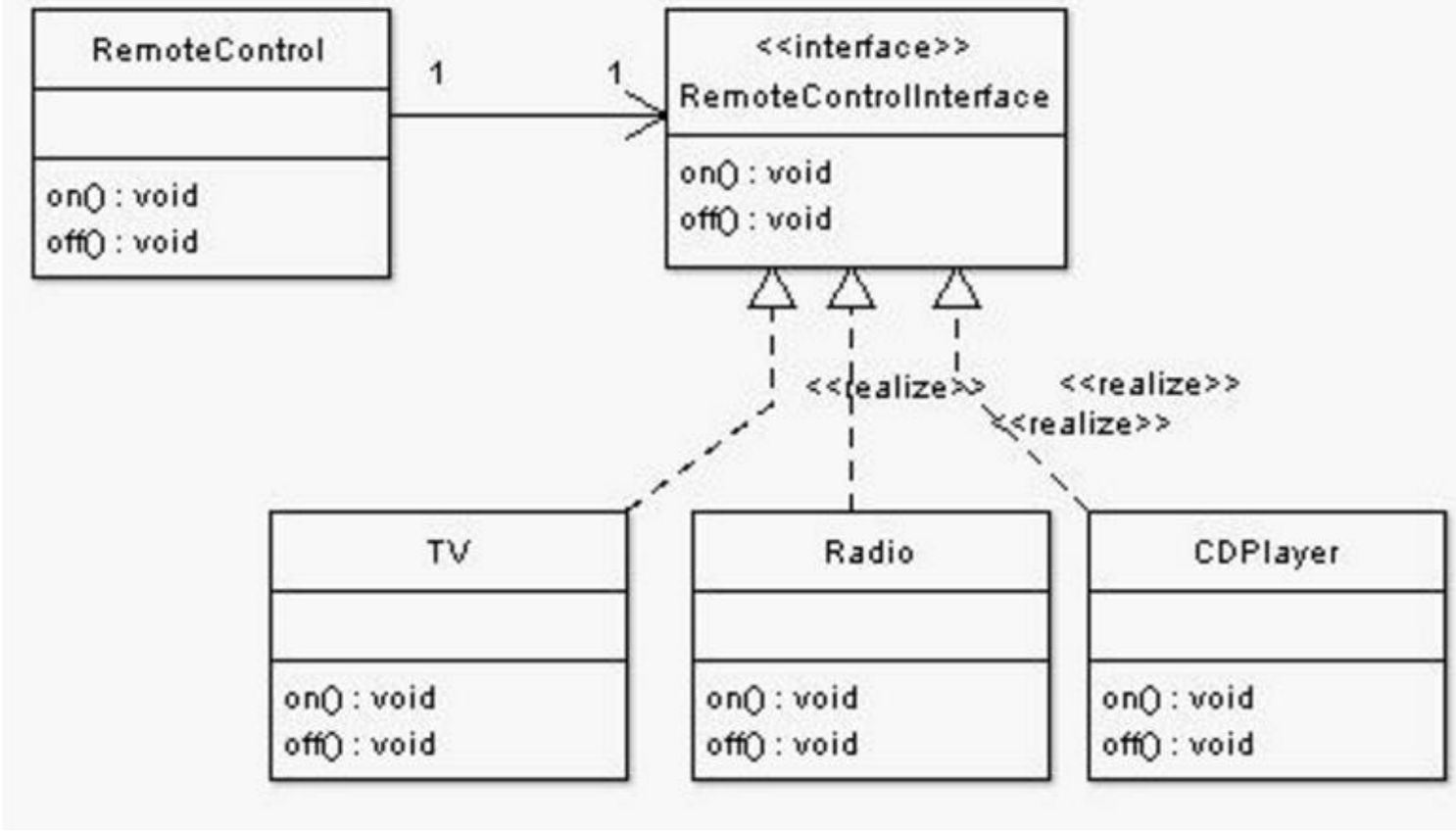
Resim 1 de görüldüğü gibi RemoteControl sınıfı TV sınıfını kullanarak işlevini yerine getirmektedir. Eğer RemoteControl sınıfını TV haricinde başka bir aleti kontrol etmek için kullanmak istersek, örneğin CDPlayer Resim 2 deki gibi değişiklik yapmamız gerekebilir. Bu noktada esnek bağ prensibini unutarak, resim 2 de yer alan çözümün bizim için yeterli olduğunu düşünelim.



Resim 2 RemoteControl sınıfı TV ve CDPlayer sınıflarından olan nesneleri kontrol etmektedir.



- Örneğin yukarıdaki çözümleme sınıfların iç yapılarında (tasarımlarında) ve birbirleriyle olan ilişkilerinde bir değişiklik olmadan yeni sınıfların eklenebildiği esnek bir yapıdır.




```
public class RemoteControl
{
    /**
     * Delegasyon islemi için RemoteControlInterface
     * tipinde bir sınıf degiskeni tanimliyoruz.
     * Tüm islemler bu nesnenin metodlarina
     * delege edilir.
     */
    private RemoteControlInterface remote;

    /**
     * Sinif konstuktörü. Bir nesne oluşturma islemi
     * esnasında kullanilacak RemoteControlInterface
     * implementasyonu parametre olarak verilir.
     *
     * @param _remote RemoteControlInterface
     */
    public RemoteControl(RemoteControlInterface _remote)
    {
        this.remote = _remote;
    }

    /**
     * Aleti acmak
     * için kullanilan metot.
     *
     */
    public void on()
    {
        remote.on();
    }


    /**
     * Aleti kapatmak
     * için kullanilan metot.
     */
    public void off()
    {
        remote.off();
    }

}
```

- Ancak yukarıdaki sınıfların üye eleman sayılarında, üye elemanlarının parametrik yapılarında, sınıflar arası ilişkilerde, sınıflardan nesne üretmenin maliyetinin arttığı durumlarda fabrika sınıfı kullanılması elzem hale dönüşür.

Faydaları Nedir?

1. Birbirine daha az bağımlı(loosely coupled) sınıflar oluşturmaya imkan tanıdığı ve Factory sınıfı ve onun alt sınıflarına nesne yaratma işlemi taşındığı için daha az karmaşık kod yazılır. Böyle bir kodun bakımı ise daha kolay olacaktır.

2. İstemci (Client) kod, sadece Product interface ile ilgilenir ve bu sayede somut(concrete) Product'lar, client kodu değiştirmeden rahatça eklenebilir.

Gerekenler

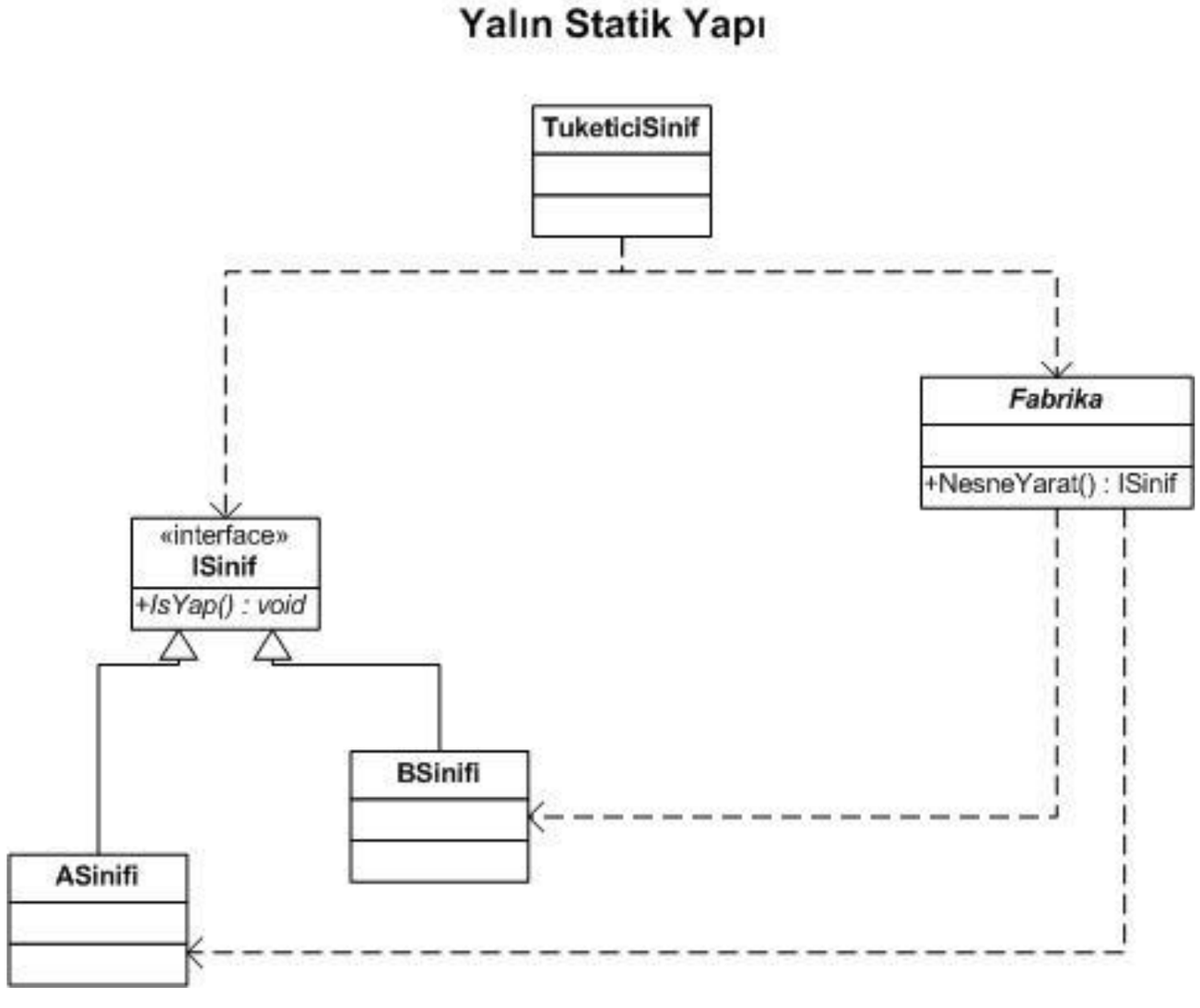
- Türü abstract veya interface olan bir süper(base) fabrika sınıfı
- En az bir tane alt fabrika sınıfı
- En az bir tane Product(ürün) sınıfı
- Test sınıfı

Yazılım Tasarımı ve Mimarisi

Örnek 1:

Bu örnekte bir factory sınıfı üzerinden soyutlama yapılır. Bir soyut sınıfın, hangi somut alt sınıfı ile gerçekleneceğini bilmeden (daha doğrusu bu mantığı saklayarak) kullanımını ifade eder. Nesnenin yaratılış mantığı ise fabrika sınıfı içerisinde saklıdır.

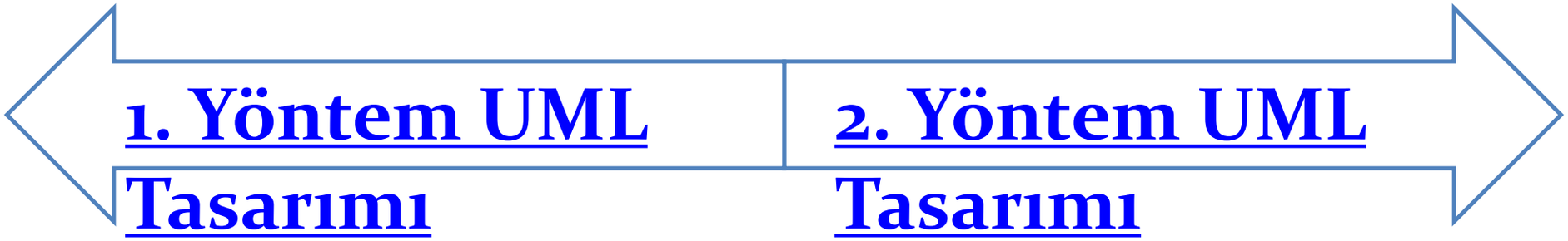
UML



Yazılım Tasarımı ve Mimarisi

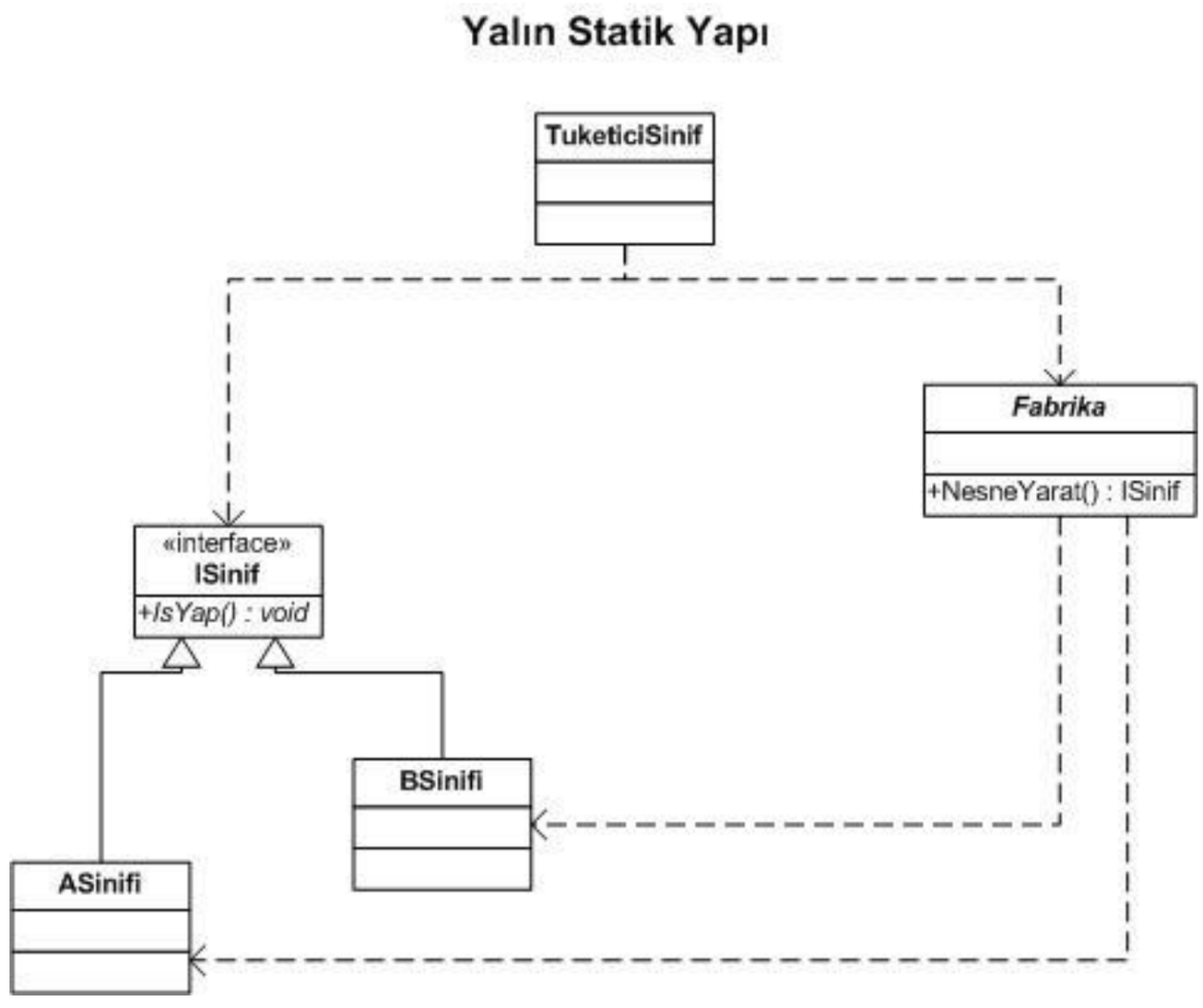
Factory desenin temel olarak 2 farklı şeklide uygulama yöntemi vardır. Birinci yöntemde nesnelerin oluşturulması için **Creator** veya **Factory** soneki almış bir sınıf içerisinde tek bir method aracılığıyla oluşturulur.

Diğer yöntemde ise her nesne için ayrı bir factory sınıf yazılır. İki yönteminde yaptığı iş aynı kurgusu farklıdır. Fakat ikinci yöntem tercih edildiğinde yeni eklemeler daha rahat olacaktır.



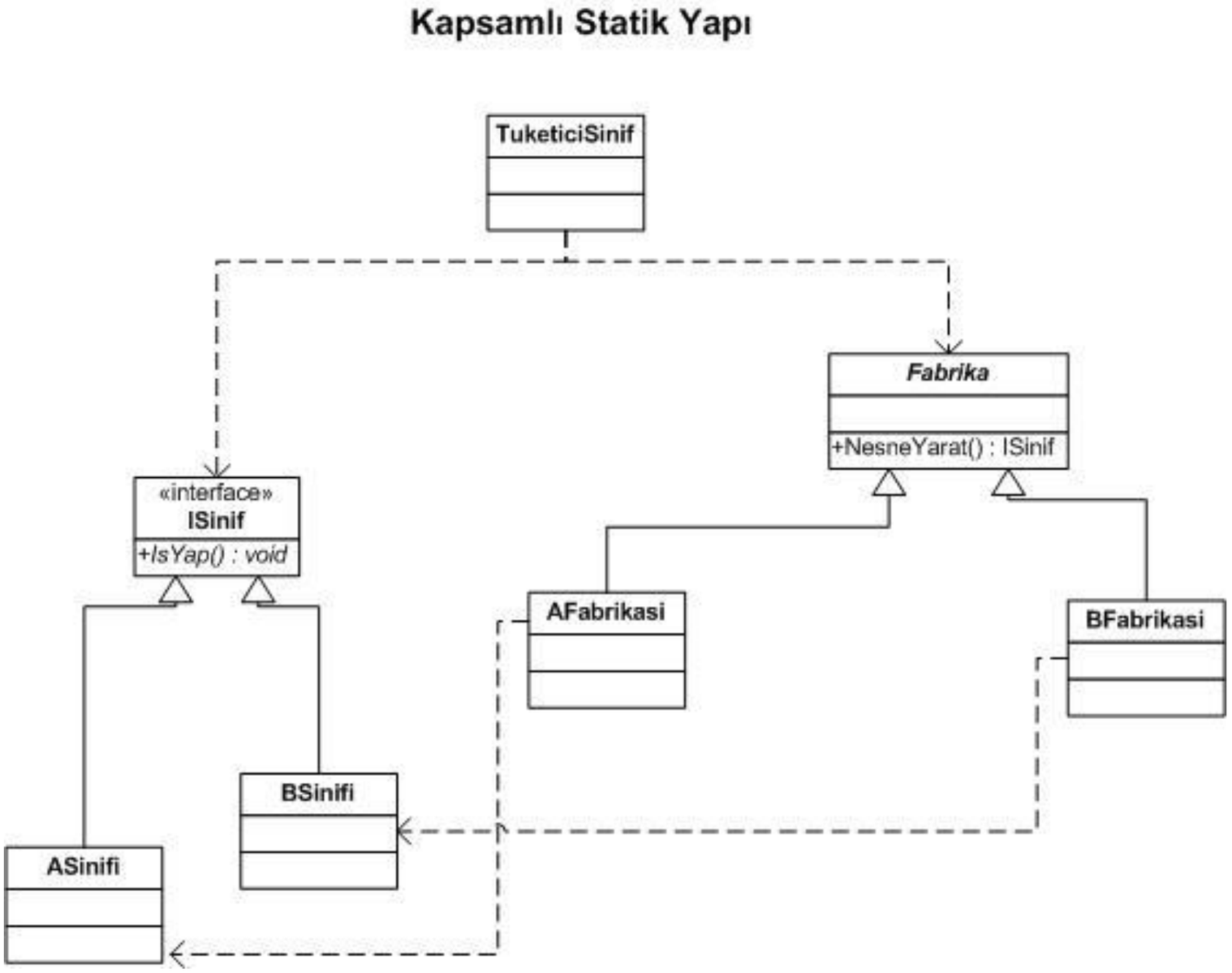
Yazılım Tasarımı ve Mimarisi

- 1. Yöntem UML Tasarımı (Factory)



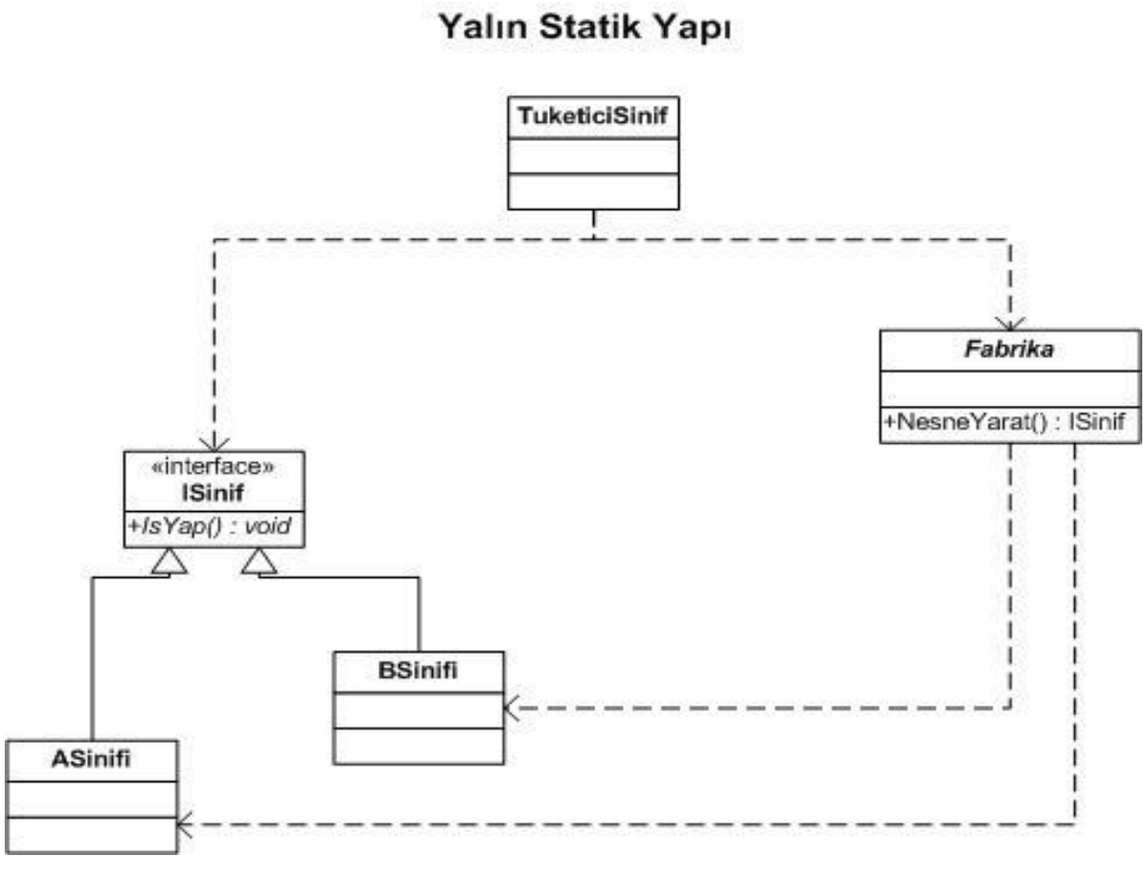
Yazılım Tasarımı ve Mimarisi

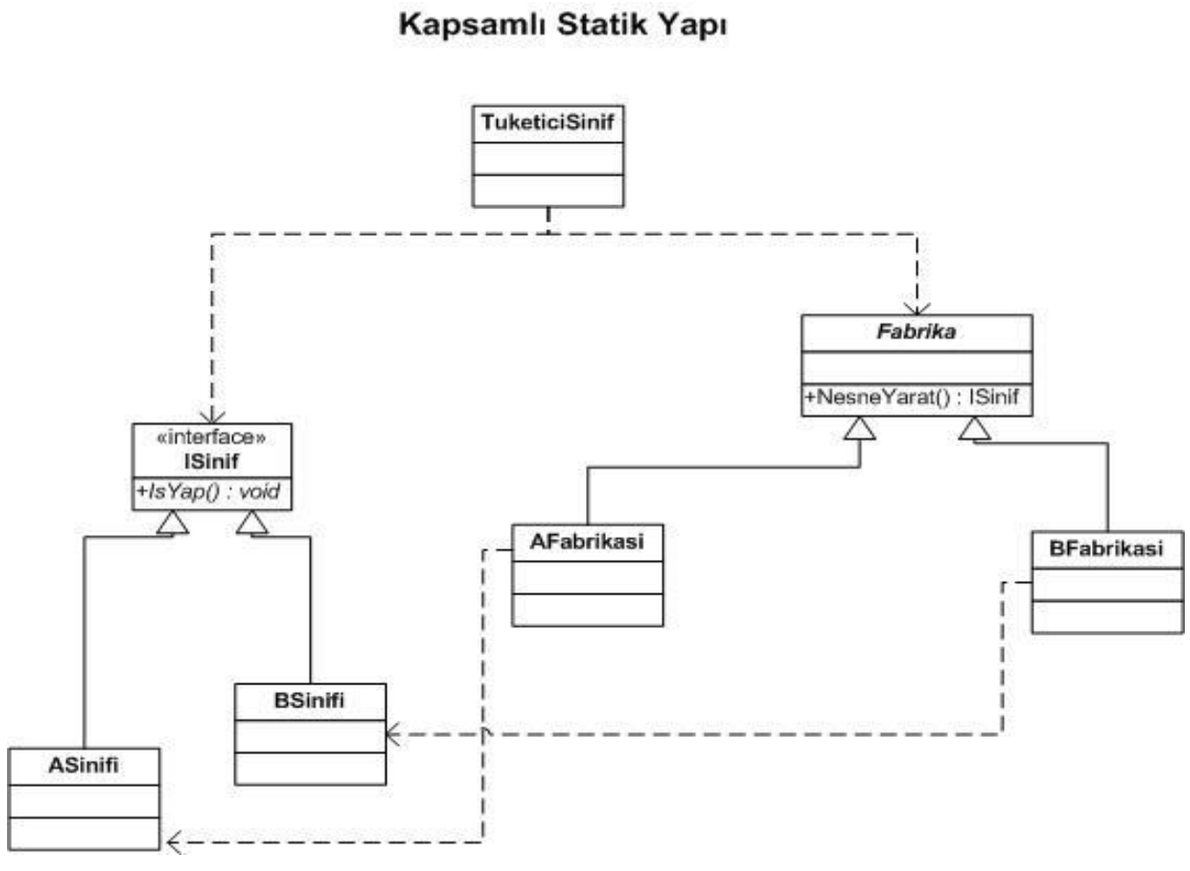
- 1. Yöntem UML Tasarımı (Factory)



Yazılım Tasarımı ve Mimarisi

- İki yöntem



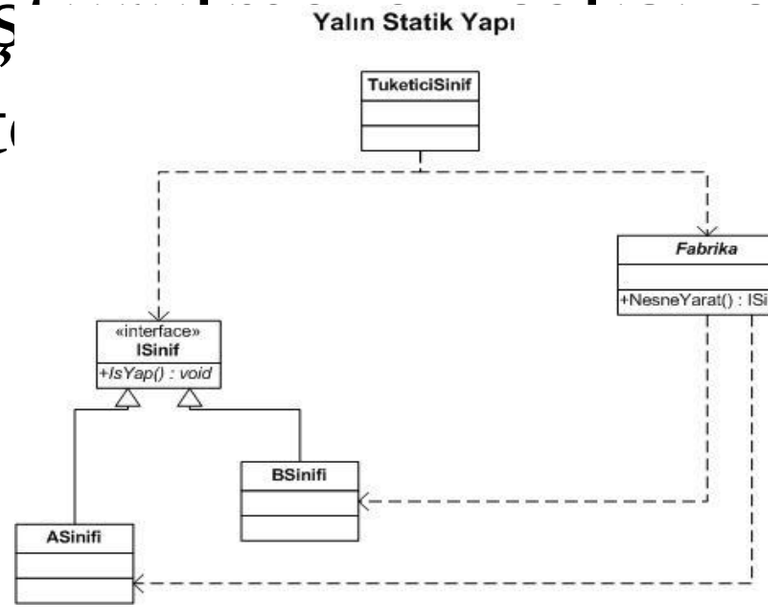


Yazılım Tasarımı ve Mimarisi

Factory Method 1.

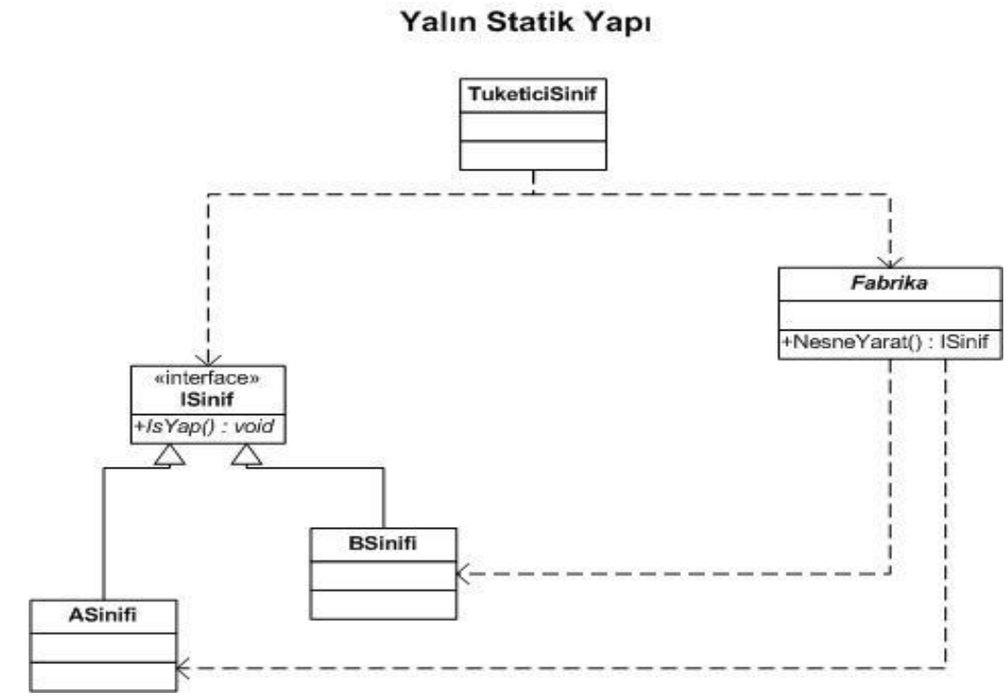
Aşağıda ilk yöntem için yazılan kod örneği görülüyor. Burada dokümanları okumak için altyapı oluş[illegible]. Burada *generic* kullanım şekli de görülmekt[illegible]

```
1  namespace FactoryMethod.Sample2
2  {
3      class Program
4      {
5          static void Main(string[] args)
6          {
7              // normal kullanım
8              PDFReader pdfreader = new PDFReader();
9              pdfreader.Read();
10             pdfreader.Extract();
11
12             // 1. kullanım
13             DocumentReaderFactory readerFac = new DocumentReaderFactory();
14             IDocumentReader pdfReader = (PDFReader)readerFac.Get("PDF");
15             pdfReader.Read();
16             pdfReader.Extract();
17
18             // 1.1 generic version
19
20             IDocumentReader pdfReader1 = (PDFReader)DocumentFactory.Get();
21             IDocumentReader wordReader1 = (MsWordReader)DocumentFactory.Get();
22
23             Console.ReadLine();
24         }
25     }
```



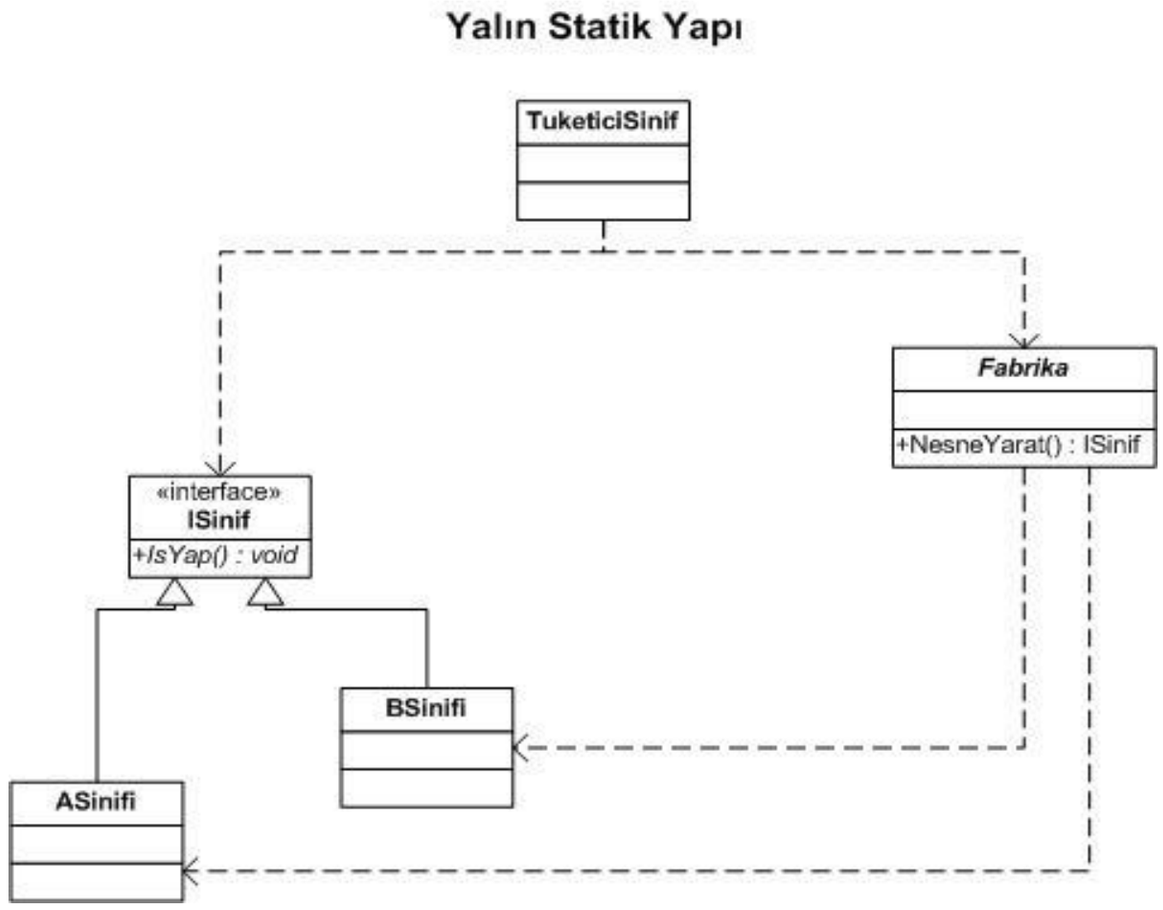
Yazılım Tasarımı ve Mimarisi

```
26
27      public interface IDocumentReader
28      {
29          void Read();
30          void Extract();
31      }
32
33      public class PDFReader
34          : IDocumentReader
35      {
36          public void Read()
37          {
38              throw new NotImplementedException();
39          }
40
41          public void Extract()
42          {
43              throw new NotImplementedException();
44          }
45      }
46
47      public class MsWordReader
48          : IDocumentReader
49      {
50          public void Read()
51          {
52              throw new NotImplementedException();
53          }
```



Yazılım Tasarımı ve Mimarisi

```
54
55          public void Extract()
56          {
57              throw new NotImplementedException();
58          }
59      }
60
61      // 1
62      public class DocumentReaderFactory
63      {
64          // burada string tipi enum da kullanabiliriz.
65          public IDocumentReader Get(string readerType)
66          {
67              switch (readerType)
68              {
69                  case "PDF":
70                      return new PDFReader();
71                  case "MsWord":
72                      return new MsWordReader();
73                  default:
74                      return new PDFReader();
75                  // örneğin burası için NULL object deseni kullanılarak nul tanımlı bir sınıf yazılabilir.
76                  // veya exception fırlatılır throw new Exception("Invalid DocumentReader type");
77              }
78          }
79      }
```



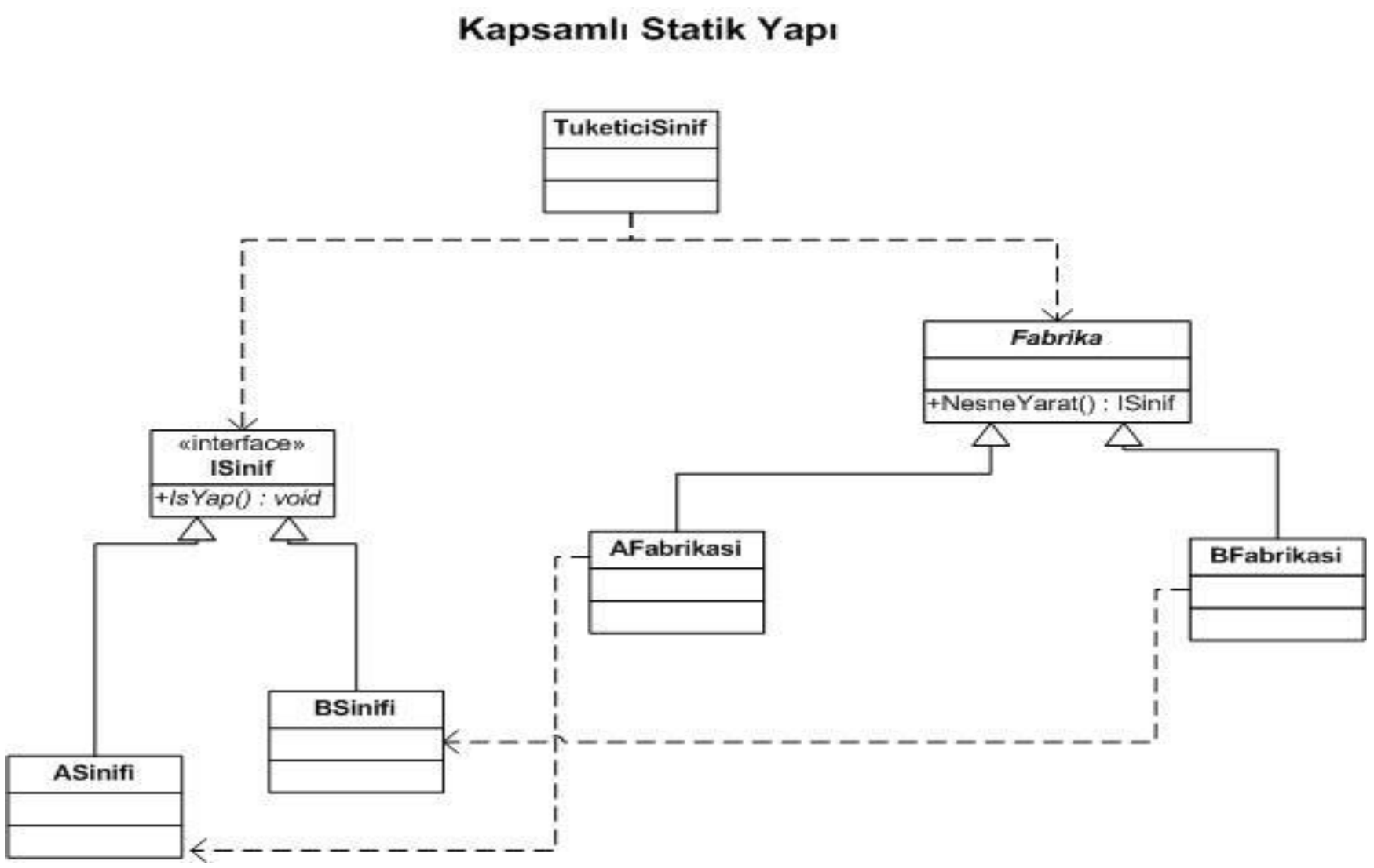
Yazılım Tasarımı ve Mimarisi

```
81      // 1.1 generic version
82      class DocumentFactory
83          where T : IDocumentReader, new()
84      {
85          private static IDocumentReader opr = null;
86
87          public static IDocumentReader Get()
88          {
89              return opr = new T();
90          }
91      }
92  }
```

Yazılım Tasarımı ve Mimarisi

Factory Sınıfı

Bu yöntemde ise ayrı sınıflar için factory sınıfları yazıyoruz. Yani PDFReader ve MsWordReader sınıfları için PDFReaderFactory ve MsWordReaderFactory sınıfları tanımlandı. Kullanım kısmında görüleceği gibi nesne örnekleri için bu factory sınıflarını kullanılıyor.



Yazılım Tasarımı ve Mimarisi

```
namespace FactoryMethod.Sample2
{
    class Program
    {
        static void Main(string[] args)
        {
            // 2. kullanım
            PDFReaderFactory pdfFactory = new PDFReaderFactory();
            PDFReader pdfReader2 = (PDFReader)pdfFactory.CreateReader();
            pdfReader2.Read();
            pdfReader2.Extract();

            //3. kullanım
             // 3.0
            DocumentFactory docFac = new DocumentFactory();
            PDFReader pdfReader3 = (PDFReader)docFac.Get(new PDFReaderFactory());
            pdfReader3.Read();
            pdfReader3.Extract();

            // 3.1
            DocumentProcessor pro = new DocumentProcessor();
            pro.Process(new PDFReaderFactory());

            Console.ReadLine();
        }
    }
    // daha önce kullanılan mevcut sınıflar yukarıda yer alıyor
```

Yazılım Tasarımı ve Mimarisi

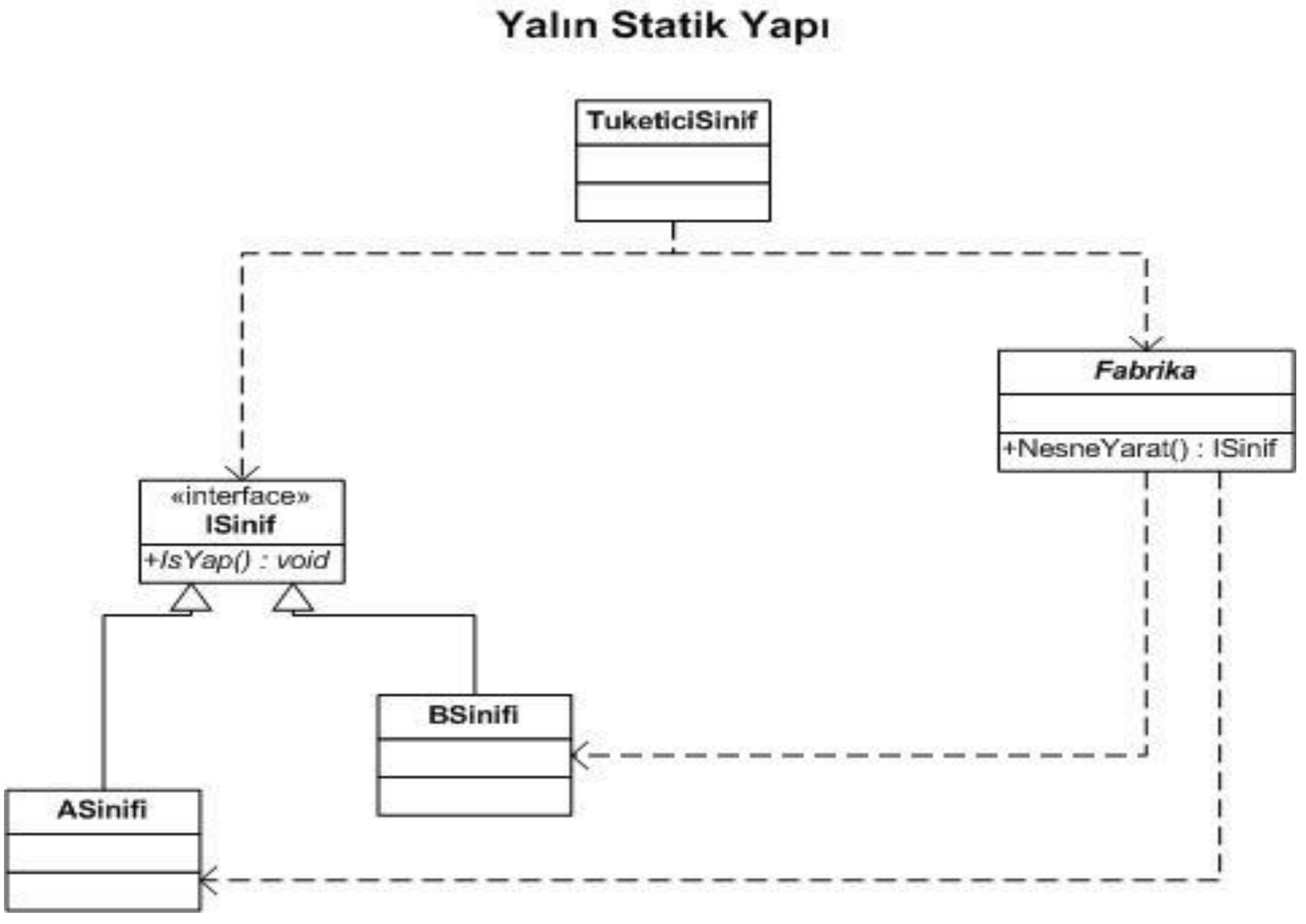
```
29      // 2
30      public interface IDocumentReaderFactory
31      {
32          IDocumentReader CreateReader();
33      }
34
35      public class PDFReaderFactory
36          : IDocumentReaderFactory
37      {
38
39          public IDocumentReader CreateReader()
40          {
41              return new PDFReader();
42          }
43      }
44
45      public class MsWordReaderFactory
46          : IDocumentReaderFactory
47      {
48          public IDocumentReader CreateReader()
49          {
50              return new MsWordReader();
51          }
52      }
```

Yazılım Tasarımı ve Mimarisi

```
53
54      // 3.0
55      public class DocumentFactory
56      {
57          public IDocumentReader Get(IDocumentReaderFactory factory)
58          {
59              return factory.CreateReader();
60          }
61      }
62
63      // 3.1
64      public class DocumentProcessor
65      {
66          public void Process(IDocumentReaderFactory factory)
67          {
68              IDocumentReader reader = factory.CreateReader();
69              reader.Read();
70              reader.Extract();
71          }
72      }
73  }
```

Yazılım Tasarımı ve Mimarisi

Örnek:



Yazılım Tasarımı ve Mimarisi

Aşağıdaki örnekteki elemanların önceki slaytta belirtilen UML diyagramında kullanılan kavramlarla eşleşmesi şu şekildedir:

ASinifi :CsvAktarici
BSinifi : ExcelAktarici
Fabrika : AktariciFabrikasi
Tuketici : Console

Bu yapıya ait kodlar aşağıdaki gibi olmaktadır:

Yazılım Tasarımı ve Mimarisi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;

namespace TasarimDesenleri.Gof.Fabrika
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length == 0)
            {
                return;
            }

            DataTable tablo = TabloOlustur();

            string format = args[0];
            bool basliklariIcersin = bool.Parse(args[1]);

            IAktarici aktarici = AktariciFabrikasi.NesneYarat(format);
            aktarici.Aktar(tablo, basliklariIcersin);
        }

        private static DataTable TabloOlustur()
        {
            throw new NotImplementedException();
        }
    }
```

Yazılım Tasarımı ve Mimarisi

```
32
33      public interface IAktarici
34      {
35          void Aktar(DataTable tablo, bool basliklariIcersin);
36      }
37
38      public class CsvAktarici : IAktarici
39      {
40          public void Aktar(DataTable tablo, bool basliklariIcersin)
41          {
42              //TODO: CSV dosyasına yaz.
43          }
44      }
45
46      public class ExcelAktarici : IAktarici
47      {
48          public void Aktar(DataTable tablo, bool basliklariIcersin)
49          {
50              //TODO: Excel dosyasına yaz.
51          }
52      }
53      public class AktariciFabrikasi
54      {
55          public const string CSV = "csv";
56          public const string EXCEL = "excel";
57
```

Yazılım Tasarımı ve Mimarisi

```
53      public class AktariciFabrikasi
54      {
55          public const string CSV = "csv";
56          public const string EXCEL = "excel";
57
58          public static IAktarici NesneYarat(string format)
59          {
60              if (format == CSV)
61              {
62                  return new CsvAktarici();
63              }
64              else if (format == EXCEL)
65              {
66                  return new ExcelAktarici();
67              }
68              else
69              {
70                  throw new InvalidOperationException("format desteklenmiyor");
71              }
72          }
73      }
74  }
```

ABSTRACT FACTORY TASARIM DESENİ

- Gerçek hayatta varlıkların çoğu zaman ilişkili oldukları gruplar ya da aileler vardır. Örneğin
  - «gardırop ve masa» mobilya grubundan,
  - «kazak ve gömlek» ise giysi grubundan birer varlıktır.
- Varlıklar arasında böyle bir gruplama yapılmasının nedeni; birbirleriyle olan mantıksal ilişkileridir.

ABSTRACT FACTORY TASARIM DESENİ

- Şüphesiz aynı aile ya da gruba ilişkin varlıkları üreten de çeşitli fabrikalar vardır. Örneğin mobilyaları üreten , elektronik eşyaları ya da giysileri üreten fabrikalar gibi.

- Fakat burada önemli olan nokta; o varlıkları kullanan kullanıcıların söz konusu varlığın üretiminden fabrikası sayesinde tümüyle soyutlanmış olmasıdır.

- Biz bir varlığa ihtiyaç duyduğumuzda onu belirli bir fabrikanın ürettiğini biliriz. Sözgelimi televizyon almak istersek televizyon fabrikasının bunu ürettiğini biliriz ama hangi markanın fabrikasının hangi yöntemle nasıl üretim yaptığı detayı bizi ilgilendirmez.

Abstract Factory Tasarım Deseni Nasıl Çalışır?

- "Creational" grubundaki desenler bir yada daha çok nesnenin çeşitli şekillerde oluşturulması ile ilgili desenlerdir.

- Bu kategoride ele alınan "Abstract Factory" ise birbirleriyle ilişkili yada birbirlerine bağlı olan nesnelerin oluşturulmasını en etkin bir şekilde çözmeyi hedefler.

- Bu hedefe ulaşmak için soyut sınıflardan (abstract class) veya arayüzlerden (interface) yoğun bir şekilde faydalanmaktadır.

Abstract Factory Tasarım Deseni Nasıl Çalışır?

- "Abstract Factory" deseninin ana teması **belirli sınıfların içerdiği ortak arayüzü** soyut bir sınıf yada arayüz olarak tasarlamaktır.

- Böylece nesneleri üreten sınıfın, hangi nesnenin üretileceği ile pek fazla ilgilenmesi gerekmez.

- İlgilenmesi gereken nokta oluşturacağı nesnenin hangi arayüzleri desteklediği yada uyguladığıdır. Bahsi geçen mekanizmalarla deseni oluşturduğumuz anda çalışma zamanında hangi nesnenin oluşturulması gerektiğini bilmeden nesnelerin oluşturulmasını yönetebiliriz.

Abstract Factory Tasarım Deseni Nasıl Çalışır?

- Abstract Factory (Soyut Fabrika) tasarım deseni tek bir arayüzden değil, her ürün ailesi için farklı bir arayüzden nesne oluşturur.

- Nesne oluşumunu arayüzler üzerinden gerçekleştirdiği için çok esnek ve genişletilebilir yapıya sahiptir.

- Bütün nesne oluşturma işlemi arayüzler üzerinden gerçekleşir. Ürünler, ürün arayüzüne bağlanır.

Abstract Factory Tasarım Deseni Nasıl Çalışır?

- Soyut fabrika tasarım deseni, somut ürün oluşturucu sınıf üzerinden alt sınıfları oluşturur.

- **İstemci, hiçbir şekilde somut sınıfları bilmez. Soyut sınıflar üzerinden tür dönüşümleri ile nesneleri oluşturur.**

- **Bu sayede if-switch gibi hangi nesnenin üretileceğine karar verecek bir yapıya da gerek kalmaz.**

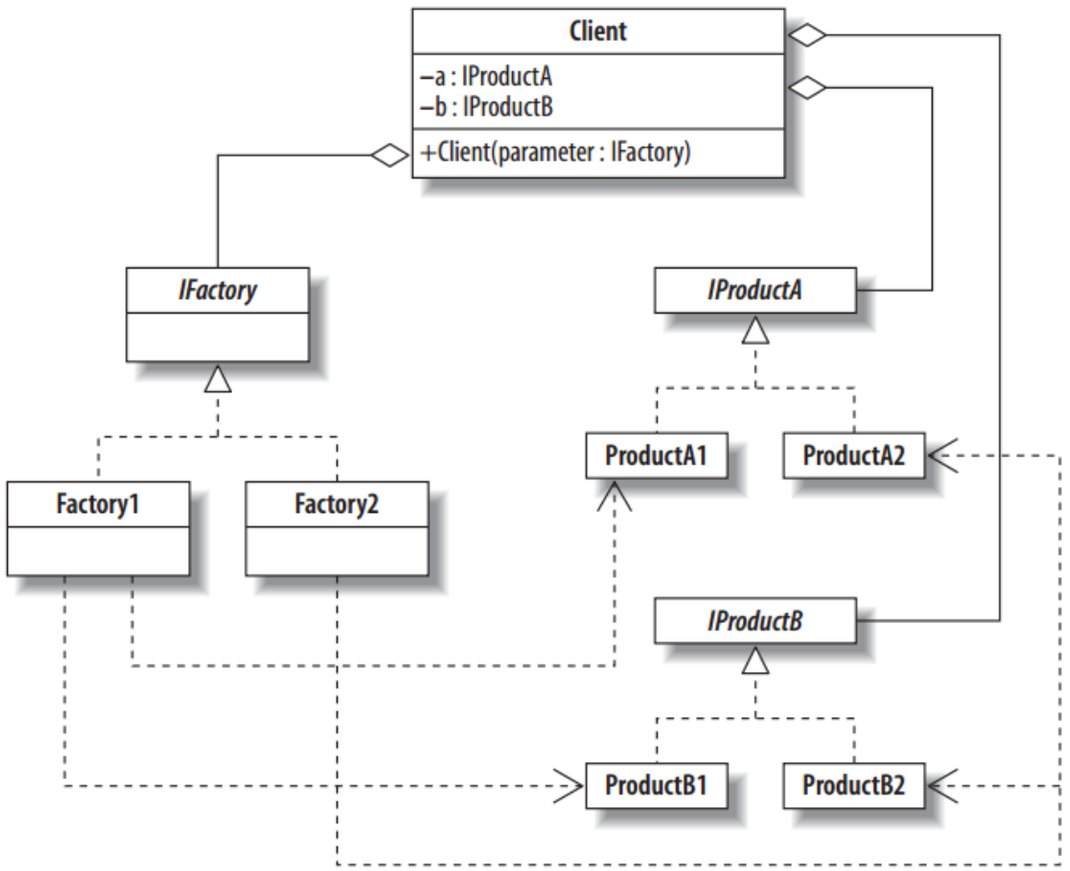
- Nesne oluşturucular, soyut fabrika arayüzüne bağlanır. Nesne oluşturucuların içinden hangi ürünlerin oluşturulacağına karar verilir. Soyut fabrika parametre olarak somut nesne oluşturucu alır ve aldığı parametreye göre ürünler oluşturulmuş olur.

Nasıl Çalışır?

- Eğer bir nesne oluşturacaksanız ve tam olarak hangi nesnenin oluşturulacağına bir switch yada if deyimi ile karar veriyorsanız muhtemelen her nesneyi oluşturduğunuzda aynı switch yapısını kullanmak zorunda kalacaksınız.
- Bu tür tekrarları önlemek için "Abstract Factory" deseninden faydalanılabilir. Bu elbette ki nesnelerin ortak bir arayüzü uygulamış olma zorunluluğunun getirdiği bir faydadır.

Abstract Factory Tasarım Deseni UML Diyagramı

- Soyut fabrika tasarım deseni, somut ürün oluşturucu sınıf üzerinden alt sınıfları oluşturur.
- İstemci, hiçbir şekilde somut sınıfları bilmez.
- Soyut sınıflar üzerinden tür dönüşümleri ile nesneleri oluşturur. Bu sayede if-switch gibi hangi nesnenin üretileceğine karar verecek bir yapıya da gerek kalmaz.
- Nesne oluşturucular, soyut fabrika arayüzüne bağlanır.
- Nesne oluşturucuların içinden hangi ürünlerin oluşturulacağına karar verilir. Soyut fabrika parametre olarak somut nesne oluşturucu alır ve aldığı parametreye göre ürünler oluşturulmuş olur.

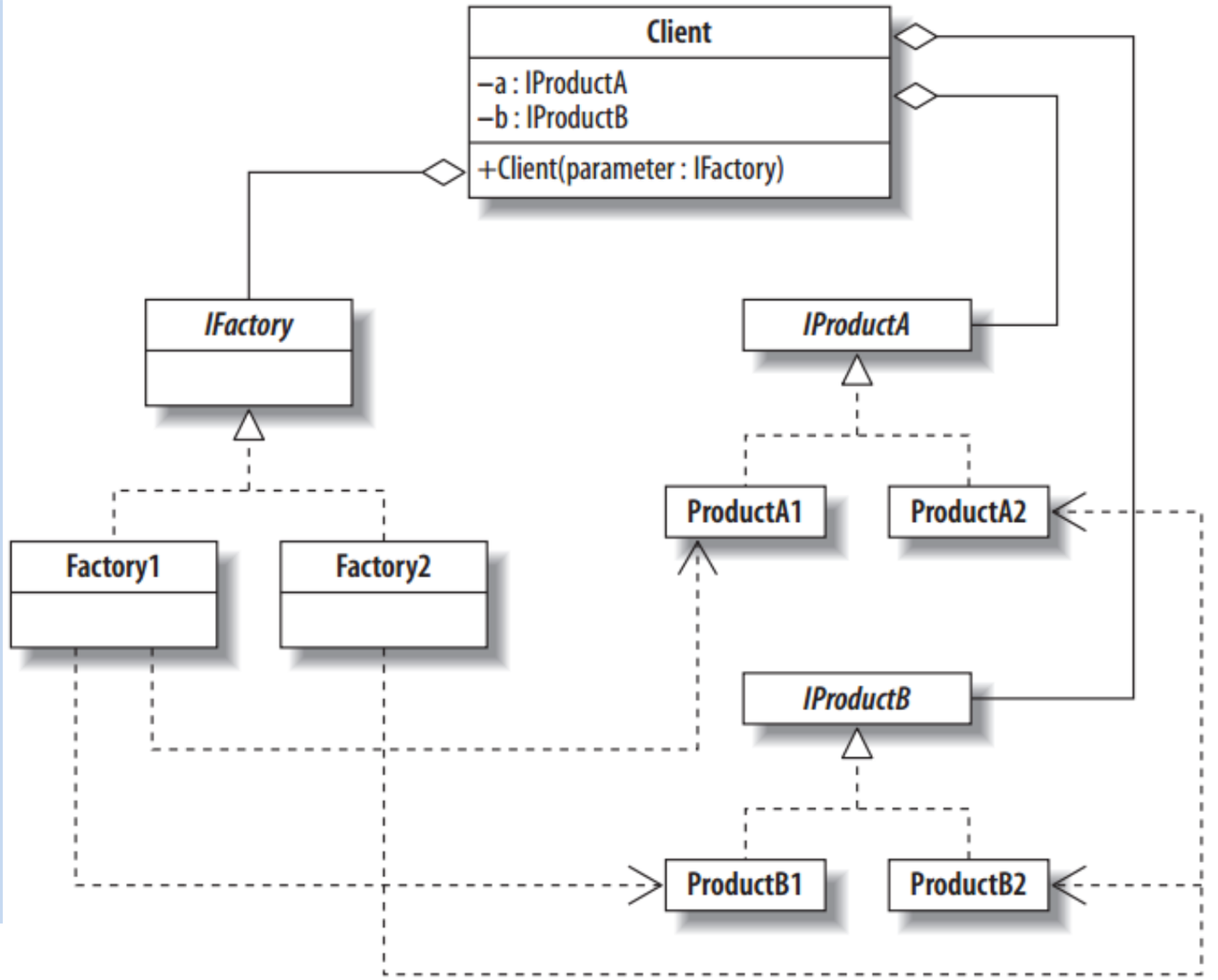


- **Abstract Factory ;** Factory sınıfları için soyut bir tip sağlar ve içerisinde yine abstract product sınıfları için methodlar bulunur. (**IFactory**)
- **Concrete Factory ;** İçerisinde nesne ailesine ilişkin concrete product nesnelerinin üretiminde sorumlu metotları içerir. Asıl fabrika sınıflarıdır. Bu sınıf abstract factory sınıfını gerçekleştirirler. (**Factory1** , **Factory2**)

Abstract Factory Tasarım Deseni UML Diyagramı

- Abstract factory tasarım deseninde aynı soyut sınıfı veya interface i uygulayan sınıfların örneğini veren fabrika sınıfları olur.
- Bu nesnelerin örnekleri bu fabrika sınıflarından elde edilir.
- Ve client da kullanılan bu nesnelerin yönetildiği base bir fabrika sınıfı yazılır.

Abstract factory tasarım desenindeki temel nesne grupları şöyledir;



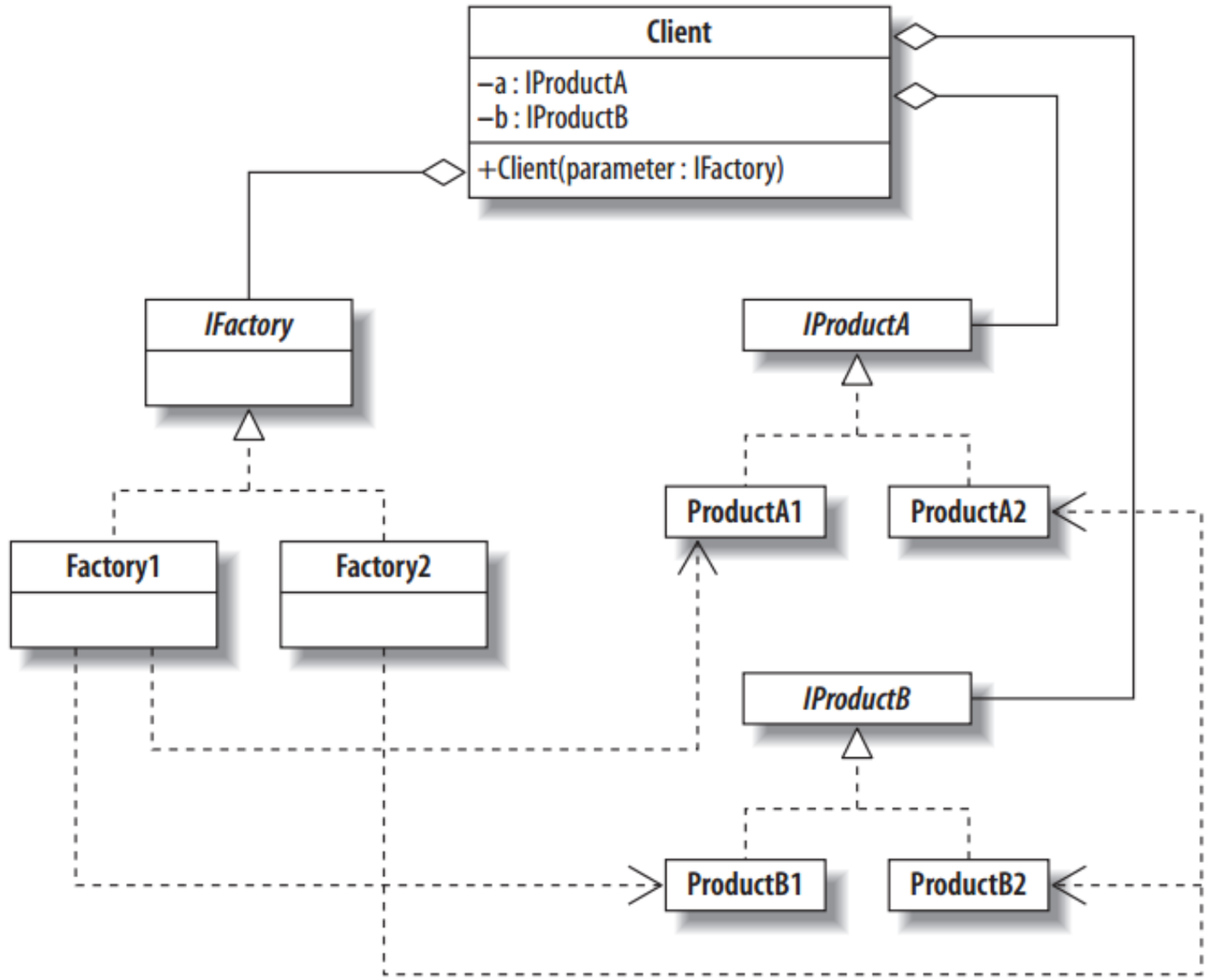
- **Abstract Factory ;** Factory sınıfları için soyut bir tip sağlar ve içerisinde yine abstract product sınıfları için methodlar bulunur. (**IFactory**)
- **Concrete Factory ;** İçerisinde nesne ailesine ilişkin concrete product nesnelerinin üretiminde sorumlu metotları içerir. Asıl fabrika sınıflarıdır. Bu sınıf abstract factory sınıfını gerçekleştirirler. (**Factory1** , **Factory2**)

Abstract Factory Tasarım Deseni UML Diyagramı

Abstract Product; birbiriyle ilişkisi bulunan sınıfları tek bir tip altında tanımlayabilmek için kullanılır.

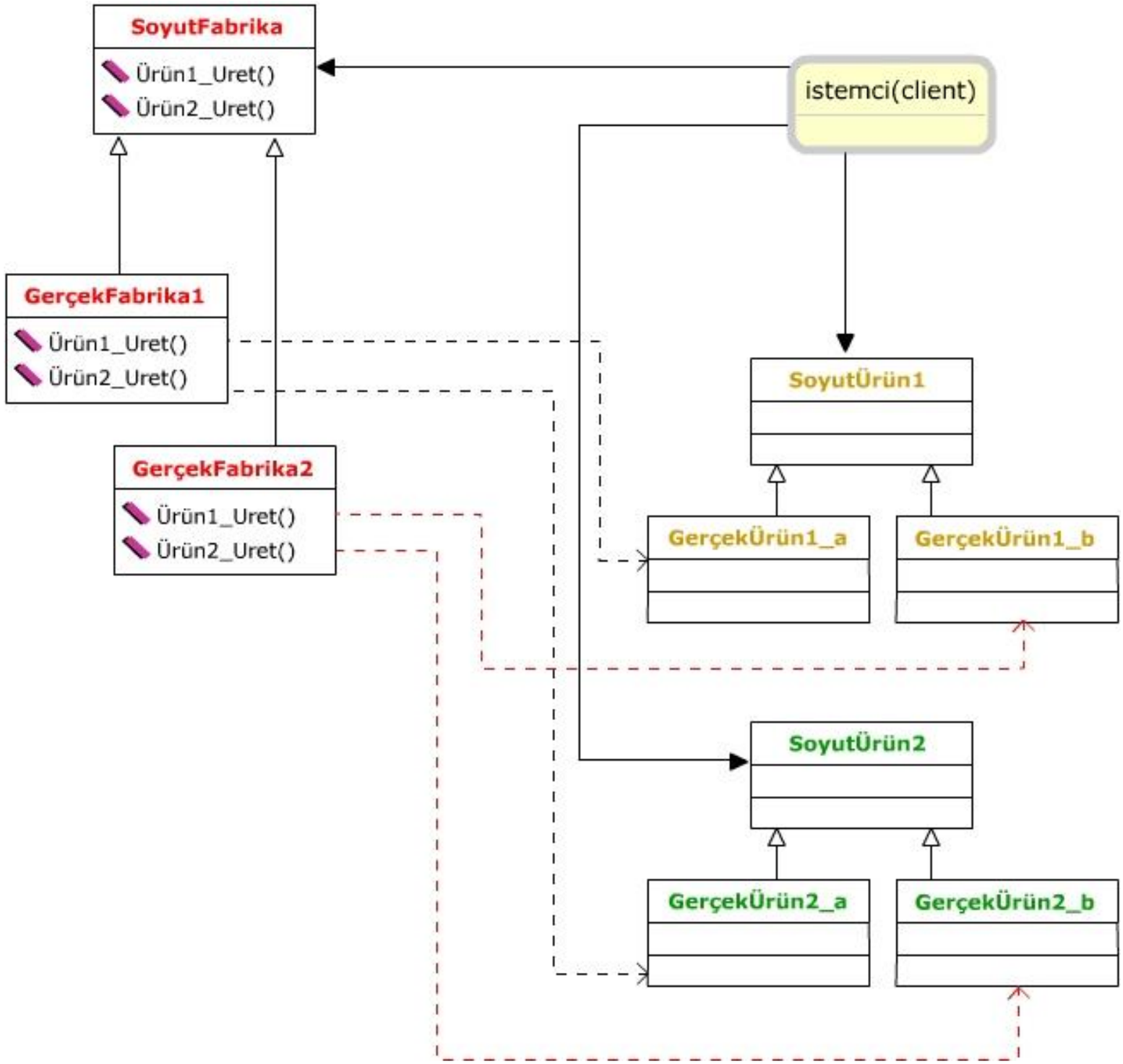
- Concrete product sınıfları tarafından uygulanır/implement edilir.
- İnterface ve abstract class olarak yazılabilirler. (**IProductA** , **IProductB**)

Concrete Product; Birbiriyle ilişkisi bulunan nesneler ve asıl üretilmesi istenilen nesnelerdir. **(ProductA1 , ProductA2 , ProductB1 , ProductB2)**



Abstract Factory Tasarım Deseni

Aşağıdaki şekil "abstract factory" tasarım deseninin yapısal UML diyagramını göstermektedir. Şemadaki her bir şekil desendeki bir sınıfı modellemektedir. Ayrıca desendeki sınıflar arasındaki ilişkilerde detaylı bir şekilde gösterilmiştir.



Abstract Factory Tasarım Deseni

- Aralarında ilişki bulunan nesnelerin üretiminden sorumlu olan soyut fabrikaların tasarlandığı tasarım deseni olarak düşünülebilir.

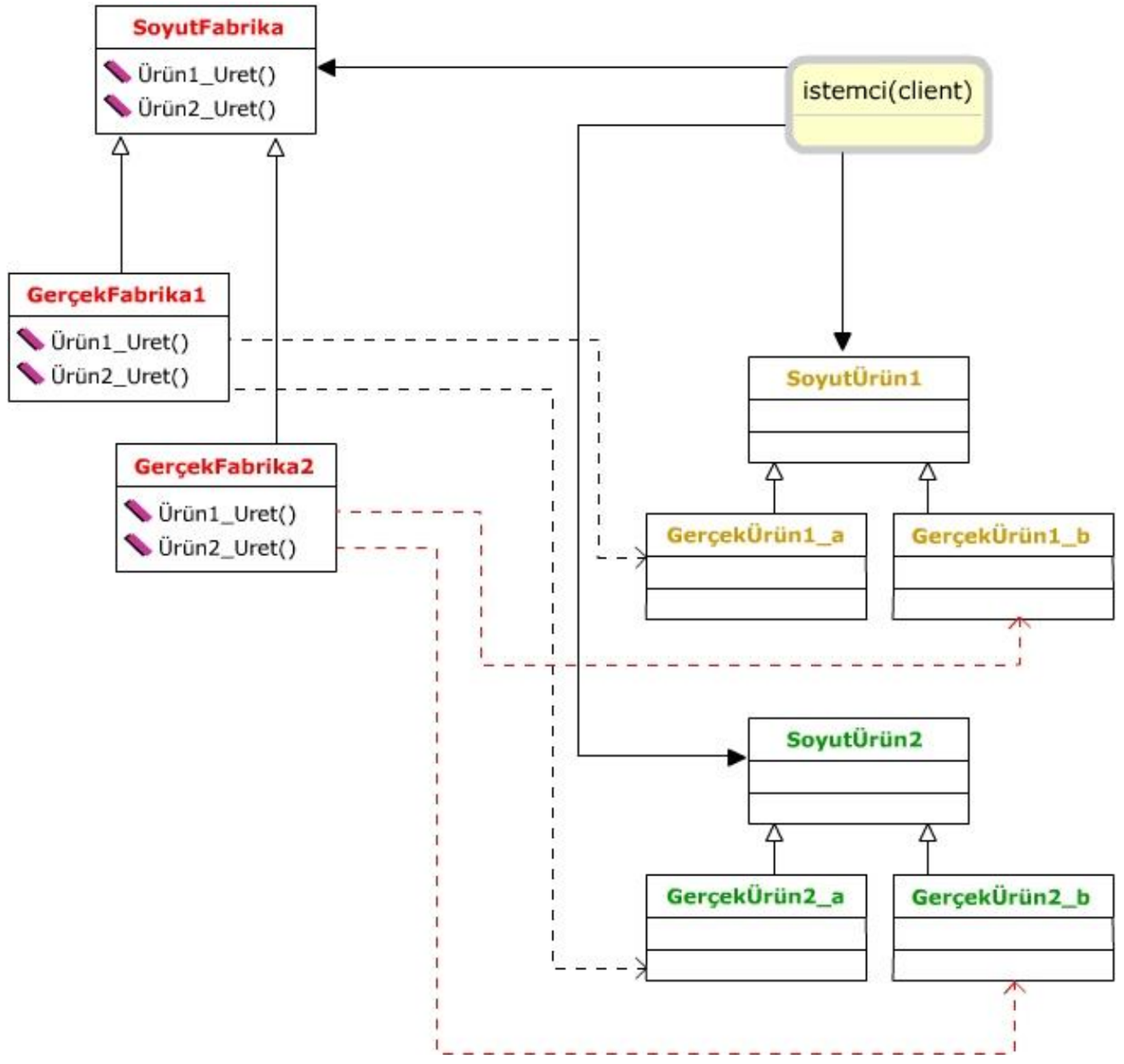
- Abstract Factory tasarım deseni sayesinde böyle durumlarda bu nesneler her oluşturulduğunda **if** gibi karar mekanizması yazmaya gerek kalmaz.

- Abstract factory deseninin uygulanabilmesi için bu desendeki nesnelerin aynı abstract class veya interface i uygulamış olması gerekir.

- Bu desen üretilen nesnelerin uyguladıkları arayüzler ile ilgilendiğinden desene yeni bir nesne eklemek kolaydır.

Abstract Factory Tasarım Deseni

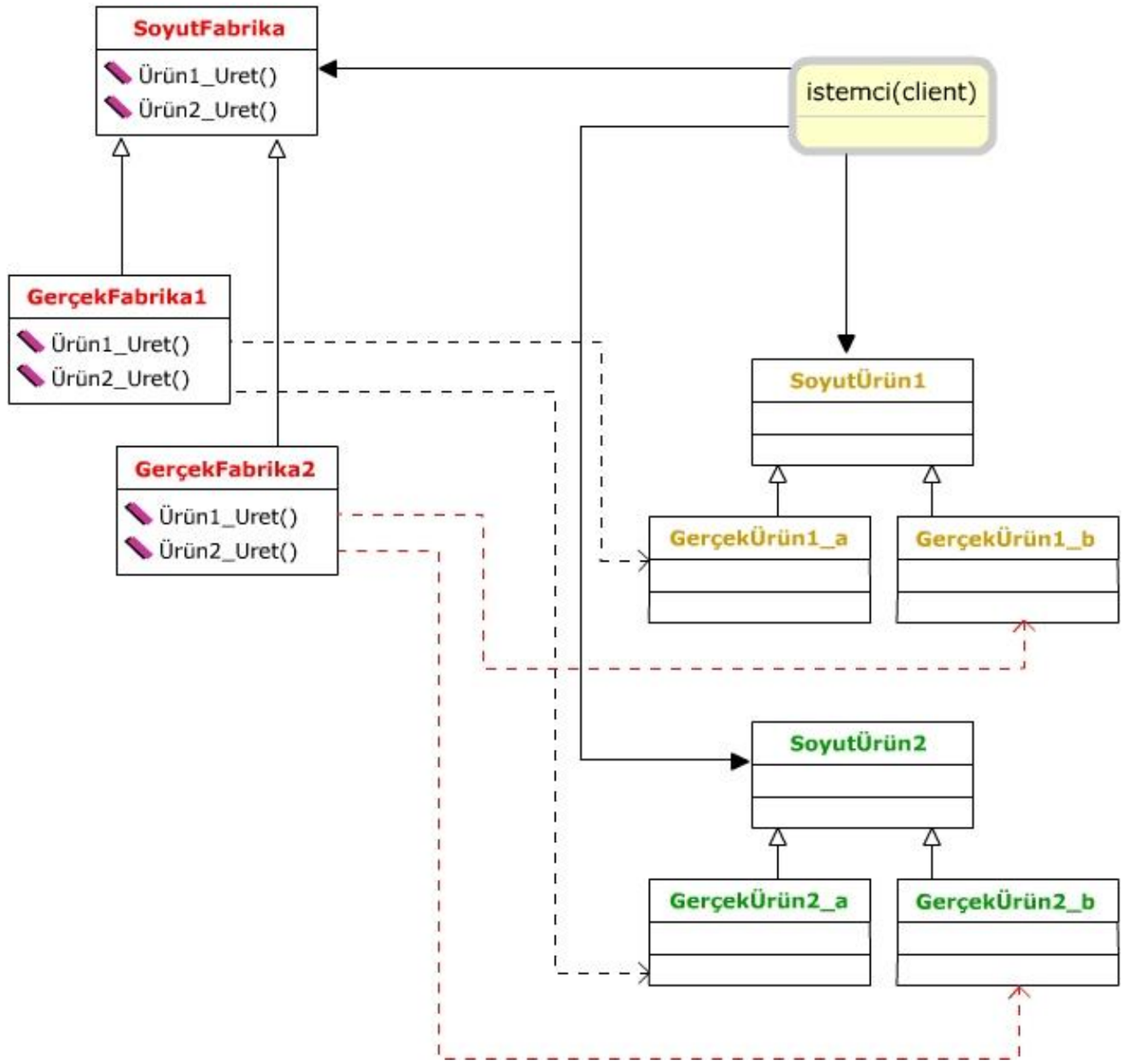
- Yandaki şemayı kısaca açıklamakta fayda var.
- Şemadan da görüleceği üzere "abstract factory" deseninde 3 ana yapı vardır.
- İlk yapı, nesnelerin oluşturulmasından sorumlu soyut ve gerçek fabrikalar, ikinci yapı soyut fabrikadan türeyen gerçek fabrikaların ürettiği ürünleri temsil eden soyut ve gerçek ürün sınıflar,
- son yapı ise herhangi bir ürünü, kendisine parametre olarak verilen soyut fabrikaları kullanarak üreten istemci(client) sınıfıdır.



Abstract Factory Tasarım Deseni

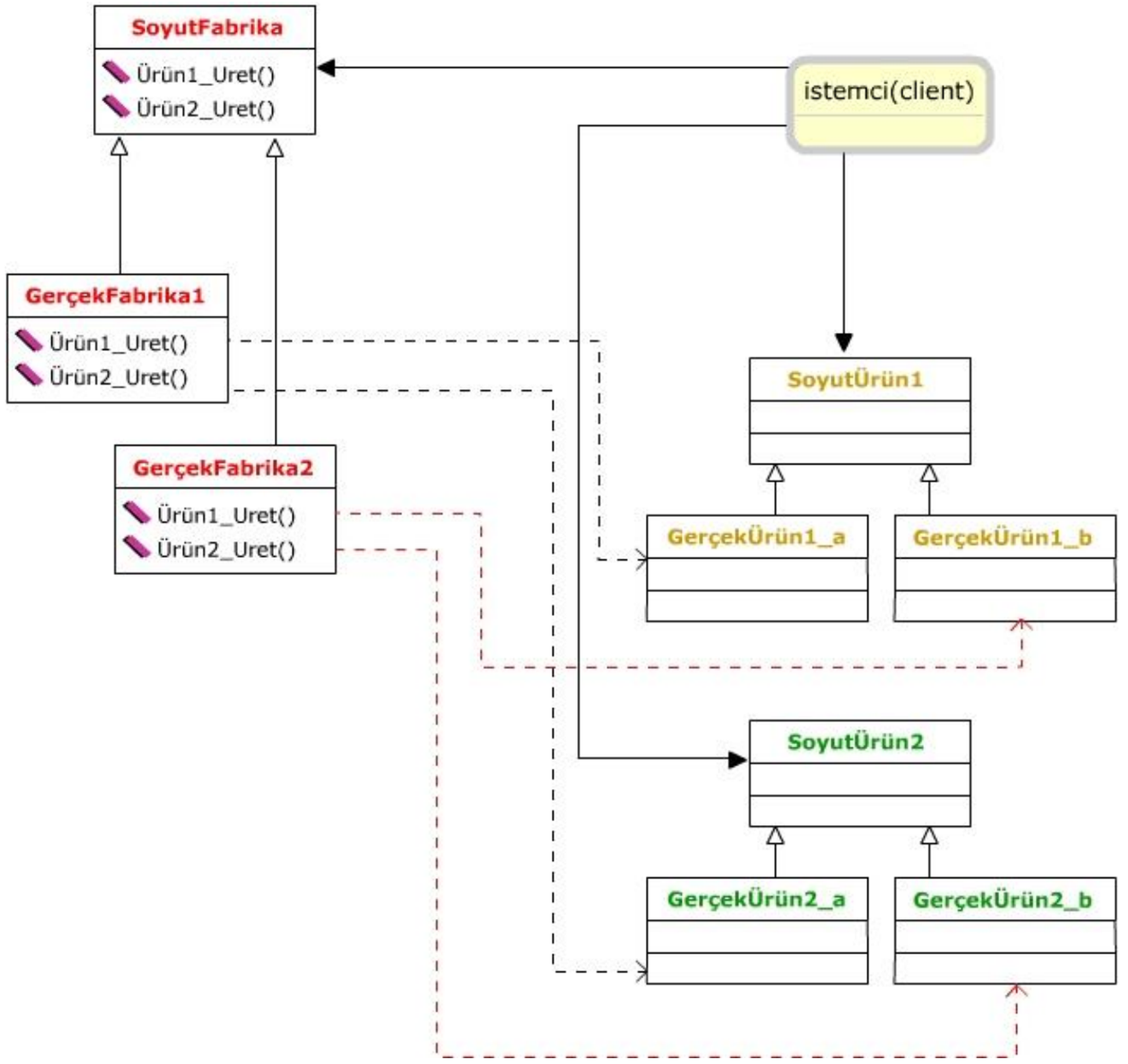
- SoyutFabrika sınıfı gerçek fabrikaların uygulaması gereken arayüzü temsil eder.

- Bu sınıf, bütün metotları soyut olan sınıf olabileceği gibi bir arayüz de olabilir. Uygulamanızın ihtiyacına göre dilediğinizi kullanabilirsiniz.



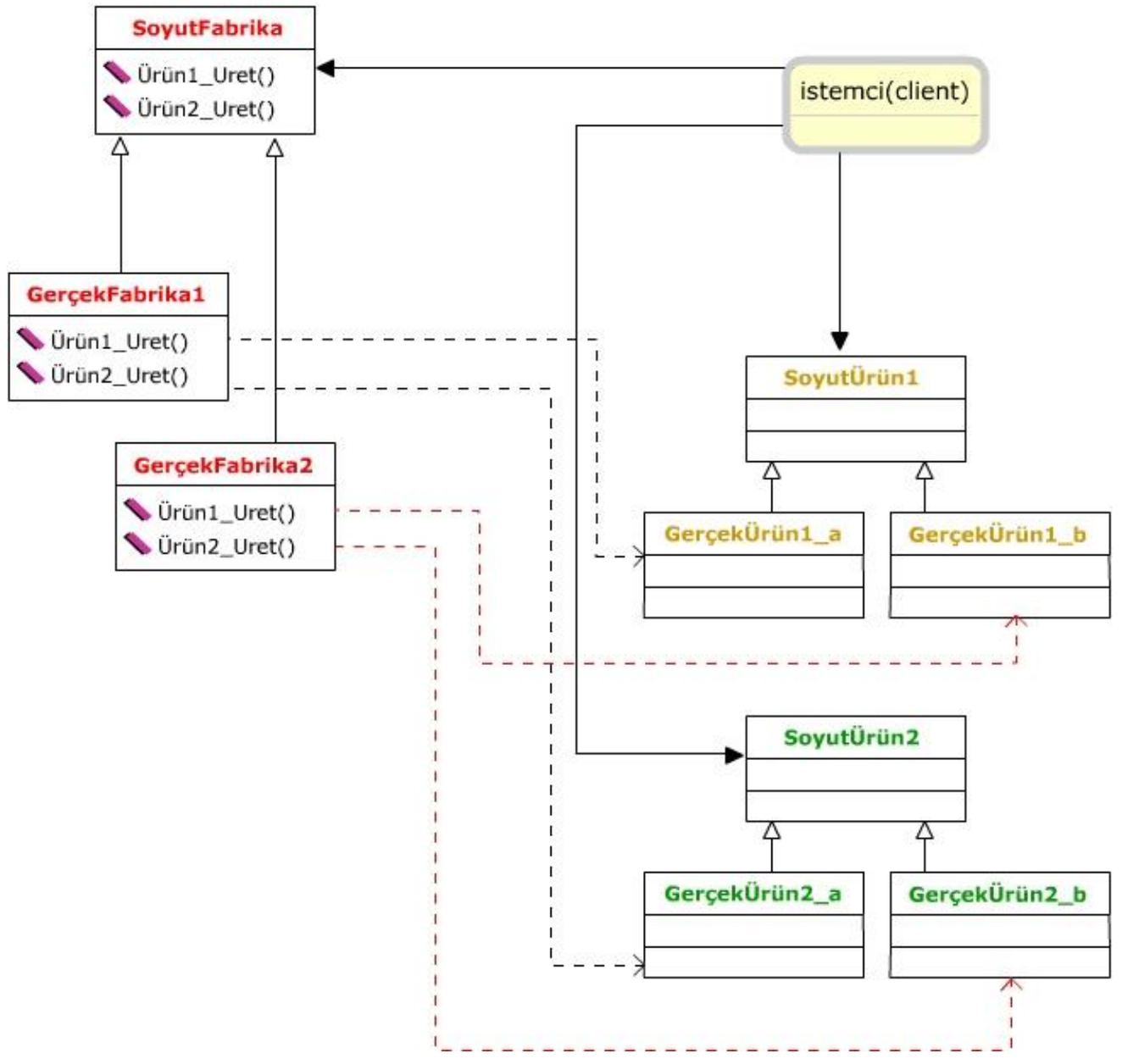
Abstract Factory Tasarım Deseni

- SoyutFabrika sınıfında ürün1 ve ürün2'nin üretilmesinden sorumlu iki tane metot bulunmaktadır.
- Dolayısıyla bütün gerçek fabrikaların hem ürün1'i hemde ürün‘2yi ürettiği kabul edilmektedir.
- Her bir ürünün ortak özelliklerini belirlemek ve ana yapıda toplamak için SoyutUrun1 ve SoyutUrun2 sınıfları oluşturulur.



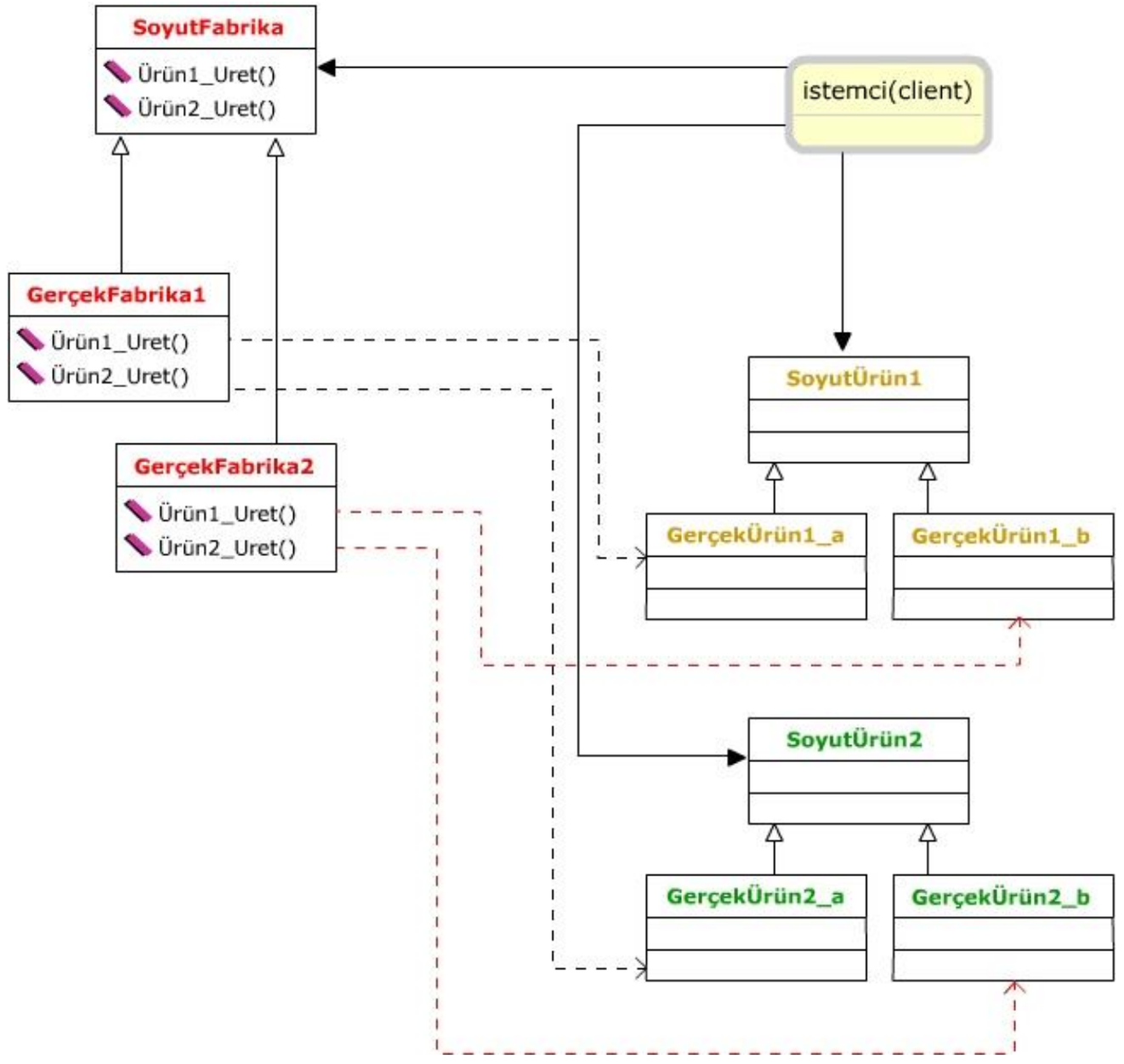
Abstract Factory Tasarım Deseni

- Bu sınıflarda herhangi bir ürüne özel bilgi bulunmamaktadır.
- Asıl bilgi bu soyut ürünlerden türeyen GercekUrun sınıflarında bulunmaktadır.
- Her bir fabrikanın ürettiği ürünleri modelleyen sınıflarda yandaki şekilde gösterilmiştir.
- Asıl önemli mesele ise gerçek fabrikaların üretimden sorumlu metotlarının ne şekilde geri döneceğidir.
- Şemadan da görüleceği üzere bu metotlar üreteceği ürünün soyut sınıfına dönmektedir. Yani üretim sonucunda geri dönen gerçek ürün nesnesi değildir.



Abstract Factory Tasarım Deseni

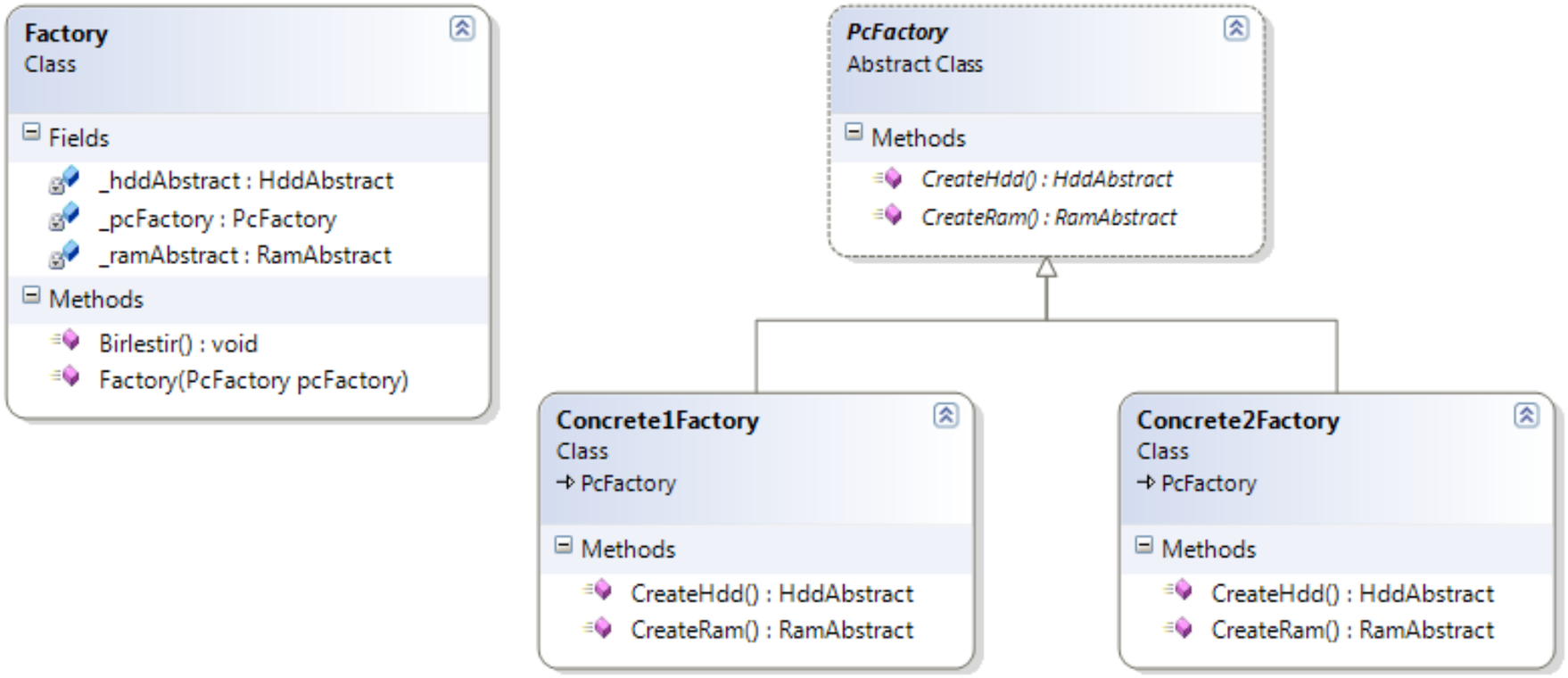
- Şemada Client olarak gösterilen sınıfın yapısı ise şu şekildedir :
- Client sınıfı yapıcı metoduna bir soyut fabrika nesnesi alır. Ve soyut fabrikanın üretimden sorumlu metotlarını kullanarak soyut ürünleri üretir.
- Dikkat ederseniz Client sınıfı hangi gerçek fabrikanın üretim yaptığından ve üretilen ürünün gerçek özelliklerinden haberi yoktur.
- Client sadece soyut fabrikanın içerdiği temel özelliklerin farkındadır. Bunu şemadaki kalın ve kesikli oklardan görmek mümkündür.



Abstract Factory Tasarım Deseni Örnek

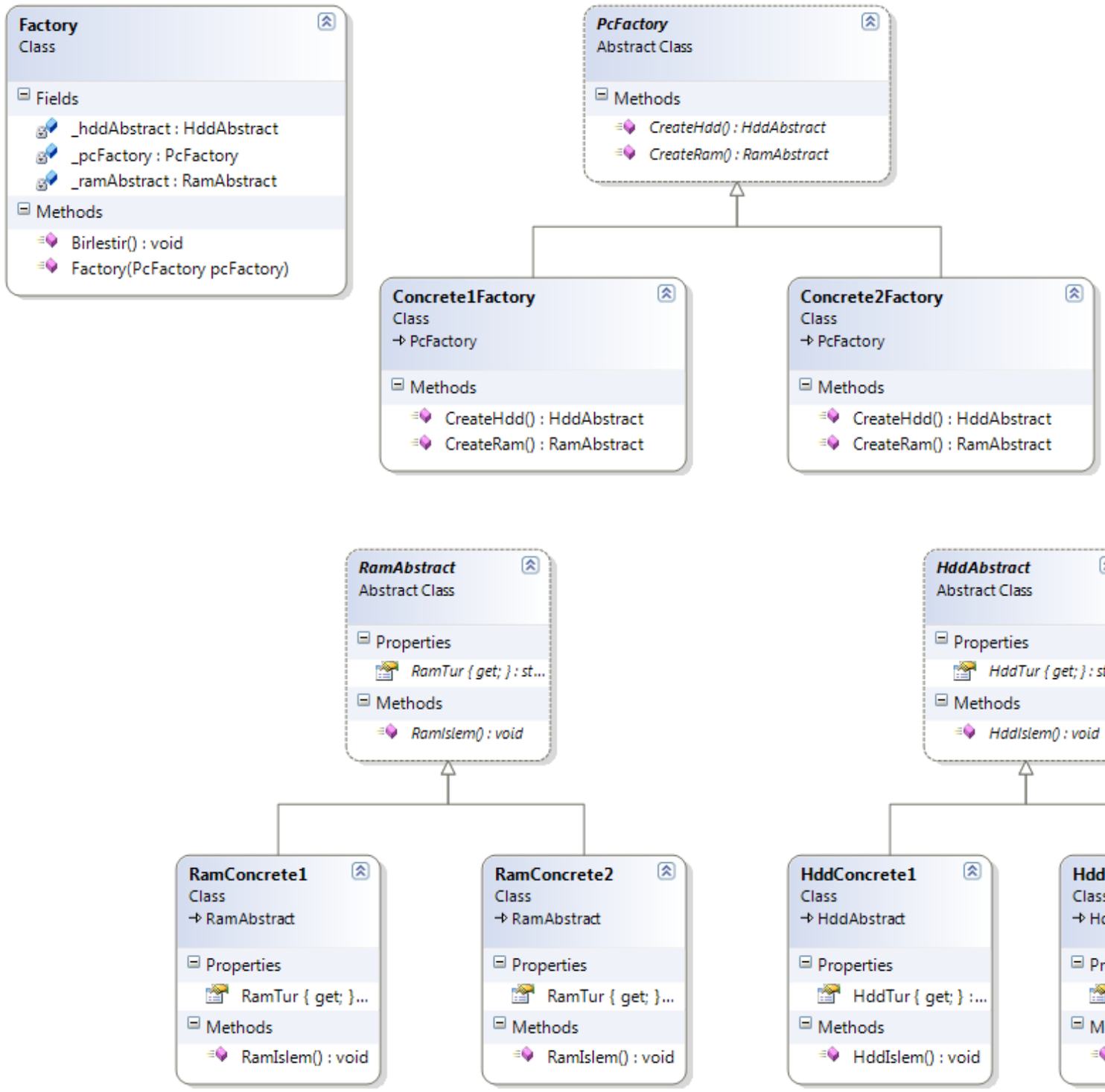
- İki farklı bilgisayar firması bilgisayarları için RAM ve HDD üretmektedir. Bu süreci modelleyen yazılım sınıflarını ve UML sınıf diyagramlarını abstract factory tasarım deseni ile gerçekleştiriniz.

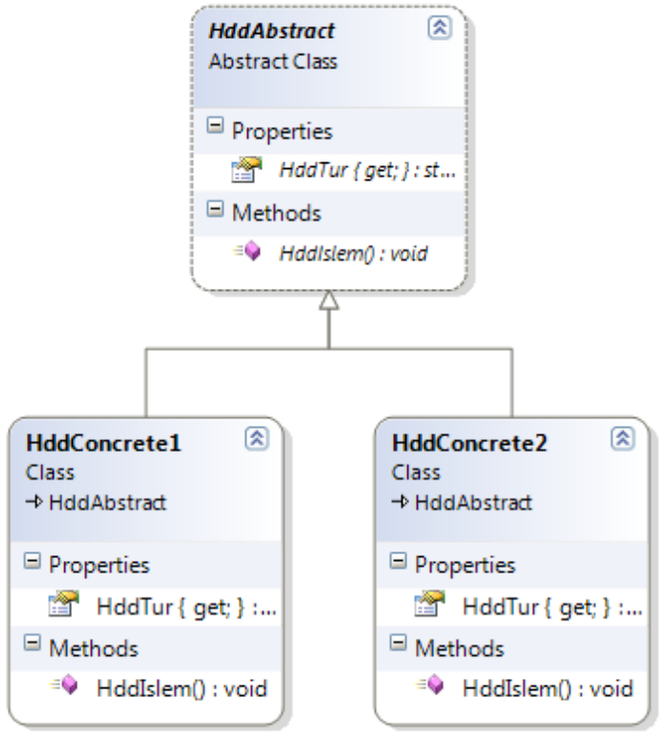
Abstract Factory Tasarım Deseni UML Diyagramı



Abstract Factory Tasarım Deseni UML Diyagramı

Her ürünümüz için (ram ve hdd) arayüz oluşturup(RamAbstract ve HddAbstract abstract sınıfları) gerçek ürünlerimizde(Concrete1 ve Concrete2 ile biten gerçek sınıflarımız)de bu arayüzleri uygulayalım. Böylece farklı türlerdeki ürünlerimiz için ortak bir yapı oluşturmuş olduk.

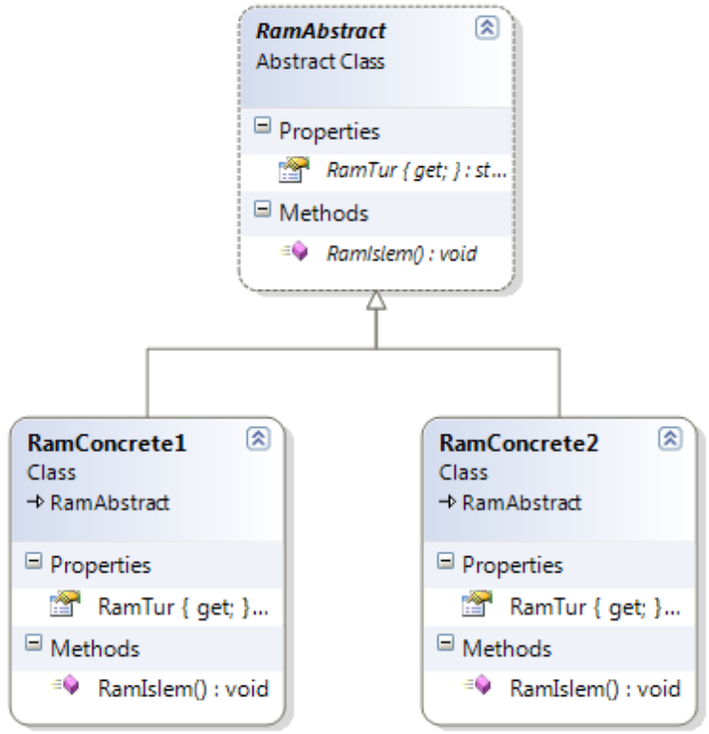





```
public abstract class HddAbstract
{
    public abstract void HddIslem();
    public abstract string HddTur { get; }
}
```

```
public class HddConcrete1:HddAbstract
{
    public override void HddIslem()
    {
        Console.WriteLine("HddConcrete1 birleştirildi.");
    }
    public override string HddTur
    {
        get { return "Bu hdd türü: HddConcrete1"; }
    }
}
```

```
public class HddConcrete1:HddAbstract
{
    public override void HddIslem()
    {
        Console.WriteLine("HddConcrete1 birleştirildi.");
    }
    public override string HddTur
    {
        get { return "Bu hdd türü: HddConcrete1"; }
    }
}
```



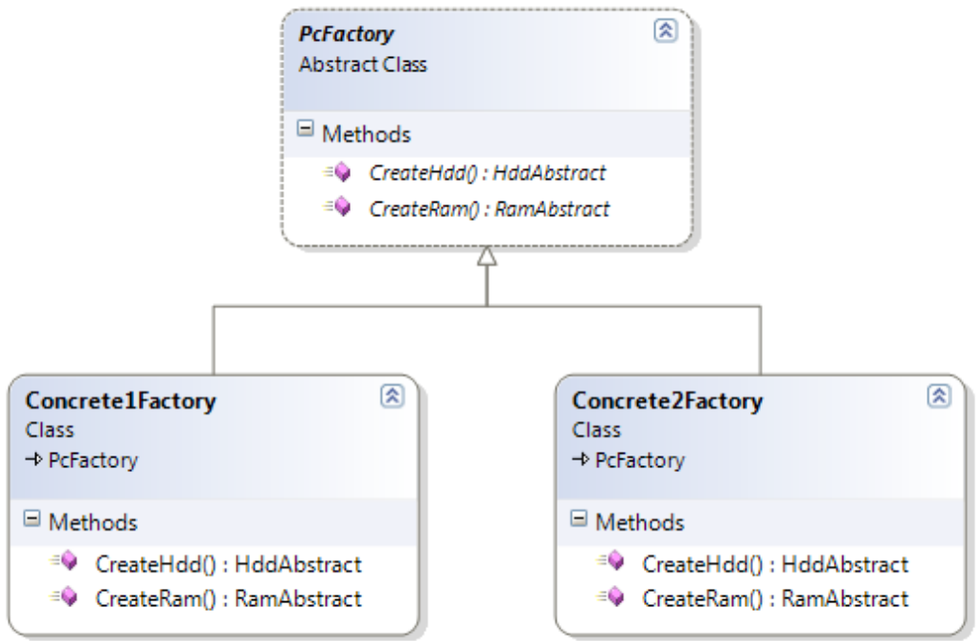

```
public abstract class RamAbstract
{
    public abstract void RamIslem();
    public abstract string RamTur { get; }
}
```

```
public class RamConcrete1 : RamAbstract
{
    public override void RamIslem()
    {
        Console.WriteLine("RamConcrete1 birleştirildi");
    }
    public override string RamTur
    {
        get { return "Bu ram türü: RamConcrete1"; }
    }
}
```

```
public class RamConcrete2 : RamAbstract
{
    public override void RamIslem()
    {
        Console.WriteLine("RamConcrete2 birleştirildi");
    }
    public override string RamTur
    {
        get { return "Bu ram türü: RamConcrete2"; }
    }
}
```

- Senaryomuza göre her pc türü için bu sınıflar farklı oluşturulacaktır.

- Bunu yönetmek için bu sınıfların üretildiği başka bir yapı daha(PcFactory abstract classı) oluşturalım.




```
public abstract class PcFactory
{
    public abstract HddAbstract CreateHdd();
    public abstract RamAbstract CreateRam();
}
```

```
public class Concrete1Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete1();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete1();
    }
}
```

```
public class Concrete2Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete2();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete2();
    }
}
```

- Senaryomuza göre her pc türü için bu sınıflar farklı oluşturulacaktır.

- sınıfların üretildiği (PcFactory abstract classı) arayüzde; RamAbstract ve HddAbstract arayüzünden türeyen sınıfların örneklerini oluşturan abstract metotlar tanımlayalım(CreateHdd ve CreateRam).

- Tabiki bu arayüzü kullanacak sınıflar ekleyip, bu metotlarda; ilgili concrete sınıflarını oluşturan kodlamayı yapıyoruz.

```
public abstract class PcFactory
{
    public abstract HddAbstract CreateHdd();
    public abstract RamAbstract CreateRam();
}
```

```
public class Concrete1Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete1();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete1();
    }
}
```

```
public class Concrete2Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete2();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete2();
    }
}
```

i. Yani Concrete2Factory sınıfı; RamConcrete2 sınıfını ve HddConcrete2 sınıfını oluşturacak,

ii. Concrete1Factory sınıfı; RamConcrete1 sınıfını ve HddConcrete1 sınıfını oluşturacak.

```
public abstract class PcFactory
{
    public abstract HddAbstract CreateHdd();
    public abstract RamAbstract CreateRam();
}
```

```
public class Concrete1Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete1();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete1();
    }
}
```

```
public class Concrete2Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete2();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete2();
    }
}
```

i. Son olarak da bütün bu yapıyı yöneteceğimiz Factory sınıfını oluşturalım.

ii. Bu sınıfın içinde concrete arayüzünden türetilen sınıflarımızı temsil eden tanımlamaları ve bu concrete sınıflarının üretiminden sorumlu olan PcFactory arayüzünden türetilen sınıfımızı temsil eden bir tanımlama yapalım.

```
public class Factory
{
    private PcFactory _pcFactory;
    private HddAbstract _hddAbstract;
    private RamAbstract _ramAbstract;
    public Factory(PcFactory pcFactory)
    {
        _pcFactory = pcFactory;
        _hddAbstract = _pcFactory.CreateHdd();
        _ramAbstract = _pcFactory.CreateRam();
    }
    public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

→

```
public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

i. Görüldüğü üzere hangi türden nesnelerin üretileceğini Factory sınıfını oluştururken constructor da verdiğimiz Concrete1Factory veya Concrete2Factory sınıfları ile belirttik. RamConcrete ve HddConcrete nesnelerinin üretimini bu sınıflara bıraktığımız için karar vermemiz gereken tek şey Concrete1Factory türünden mi Concrete2Factory türünden mi Ram ve Hdd sınıflarının üretileceğidir.

```
public class Factory
{
    private PcFactory _pcFactory;
    private HddAbstract _hddAbstract;
    private RamAbstract _ramAbstract;
    public Factory(PcFactory pcFactory)
    {
        _pcFactory = pcFactory;
        _hddAbstract = _pcFactory.CreateHdd();
        _ramAbstract = _pcFactory.CreateRam();
    }
    public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

→

```
public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

Sonradan senaryomuza Ultrabook eklememiz gerekir ise sadece Ultrabook için sınıfları oluşturmamız ve ilgili yerlerde kullanmamız gerekir. AbstractFactory deseni generic kullanılarak daha etkin bir şekilde de kullanılabilir. Gerçek kodlamadan örnek vermemiz gerekir ise birden fazla türde örneğin hem SqlServer hem Oracle veritabanı kullanacak bir uygulama yazarken abstract factory tasarım deseninden faydalanılabilinir.

```
classProgram
{
    static void Main(string[] args)
    {
        Factory f = new Factory(new Concrete1Factory());
        f.Birlestir();
        Console.WriteLine();
        f = new Factory(new Concrete2Factory());
        f.Birlestir();
        Console.ReadKey();
    }
}
```

```
public class Factory
{
    private PcFactory _pcFactory;
    private HddAbstract _hddAbstract;
    private RamAbstract _ramAbstract;
    public Factory(PcFactory pcFactory)
    {
        _pcFactory = pcFactory;
        _hddAbstract = _pcFactory.CreateHdd();
        _ramAbstract = _pcFactory.CreateRam();
    }
    public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

```
public void Birlestir()
    {
        Console.WriteLine(_hddAbstract.HddTur);
        _hddAbstract.HddIslem();
        Console.WriteLine(_ramAbstract.RamTur);
        _ramAbstract.RamIslem();
    }
}
```

```
public abstract class PcFactory
{
    public abstract HddAbstract CreateHdd();
    public abstract RamAbstract CreateRam();
}
```

```
public class Concrete1Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete1();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete1();
    }
}
```

```
public class Concrete2Factory : PcFactory
{
    public override HddAbstract CreateHdd()
    {
        return new HddConcrete2();
    }
    public override RamAbstract CreateRam()
    {
        return new RamConcrete2();
    }
}
```

```
public abstract class HddAbstract
{
    public abstract void HddIslem();
    public abstract string HddTur { get; }
}
```

```
public abstract class RamAbstract
{
    public abstract void RamIslem();
    public abstract string RamTur { get; }
}
```

```
public class HddConcrete1:HddAbstract
{
    public override void HddIslem()
    {
        Console.WriteLine("HddConcrete1 birleştirildi.");
    }
    public override string HddTur
    {
        get { return "Bu hdd türü: HddConcrete1"; }
    }
}
```

```
public class HddConcrete1:HddAbstract
{
    public override void HddIslem()
    {
        Console.WriteLine("HddConcrete1 birleştirildi.");
    }
    public override string HddTur
    {
        get { return "Bu hdd türü: HddConcrete1"; }
    }
}
```

```
public class RamConcrete1 : RamAbstract
{
    public override void RamIslem()
    {
        Console.WriteLine("RamConcrete1 birleştirildi");
    }
    public override string RamTur
    {
        get { return "Bu ram türü: RamConcrete1"; }
    }
}
```

```
public class RamConcrete2 : RamAbstract
{
    public override void RamIslem()
    {
        Console.WriteLine("RamConcrete2 birleştirildi");
    }
    public override string RamTur
    {
        get { return "Bu ram türü: RamConcrete2"; }
    }
}
```

Abstract Factory Tasarım Deseni

Desenin C# ile Gerçekleştirilmesi

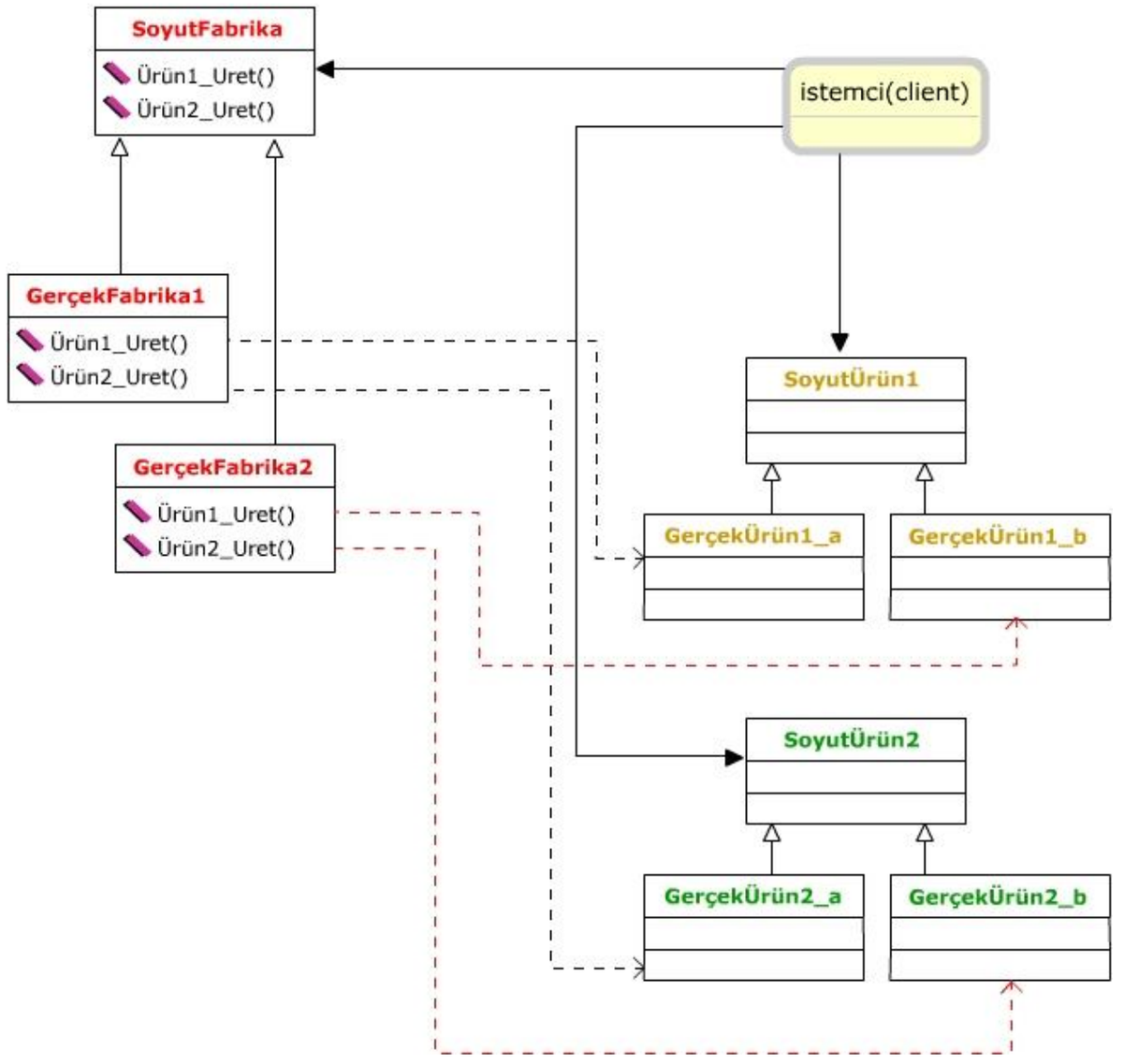
```
using System; namespace DesignPattern
{
    abstract class SoyutArabaFabrikasi
    {
        abstract public SoyutArabaKasasi KasaUret();
        abstract public SoyutArabaLastigi LastikUret();
    }

    class MercedesFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new MercedesE200();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new MercedesLastik();
        }
    }
class FordFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new FordFocus();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new FordLastik();
        }
    }
 }
```

- Yukarıdaki yapısal örneği verdikten sonra gerçek bir örnek ile bu deseni nasıl gerçekleştirebileceğimizi inceleyelim. Bu örnekte araba kasası ve araba lastiği üreten farklı iki firmanın üretimi modellenmektedir.



Abstract Factory Tasarım Deseni

Desenin C# ile Gerçekleştirilmesi

```
using System; namespace DesignPattern
{
    abstract class SoyutArabaFabrikasi
    {
        abstract public SoyutArabaKasasi KasaUret();
        abstract public SoyutArabaLastigi LastikUret();
    }

    class MercedesFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new MercedesE200();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new MercedesLastik();
        }
    }
class FordFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new FordFocus();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new FordLastik();
        }
    }
}
```

- örnekte SoyutArabaFabrikasi sınfı iki metot içermektedir. Bu metotlar SoyutArabaFabrikasi sınıfından türeyecek sınıfların uygulaması gereken metotlardır. Çünkü metotlar abstract olarak bildirilmiştir. Bu metotlar gerçek fabrika sınıflarının araba kasası ve araba lastiği üretmesi gerektiğinin belirtisidir. Zira görüldüğü üzere SoyutArabaFabrikasi sınıfından türeyen MercedesFabrikasi ve FordFabrikasi kendilerine has lastikleri ve kasaları üretmek için soyut fabrika sınıfının metotlarını kullanmaktadır. Bu metotlar geri dönüş değeri olarak soyut ürün sınıflarını temsil eden sınıfları döndürmektedirler.

Abstract Factory Tasarım Deseni

Desenin C# ile Gerçekleştirilmesi

```
using System; namespace DesignPattern
{
    abstract class SoyutArabaFabrikasi
    {
        abstract public SoyutArabaKasasi KasaUret();
        abstract public SoyutArabaLastigi LastikUret();
    }

    class MercedesFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new MercedesE200();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new MercedesLastik();
        }
    }
class FordFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new FordFocus();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new FordLastik();
        }
    }
}
```

- Örneğin KasaUret() metodu her bir fabrika için farklı ürün üretmesine rağmen her bir ürün SoyutArabaKasasi sınıfındaki metotları uyguladığı için gerçek ürünler birbirleyile ilişkili hale gelir. Mercedes fabrikası KasaUret() metodu ile MercedesE200 ürününü döndürmesine rağmen Ford fabrikası aynı metotla FordFocus ürününü döndürmektedir. Ancak her iki fabrikanın da ürettiği ürün SoyutArabaKasasi sınıfından türediği için herhangi bir çelişki olmamaktadır.

Abstract Factory Tasarım Deseni

```
using System; namespace DesignPattern
{

   abstract class SoyutArabaKasasi
   {
      abstract public void LastikTak(SoyutArabaLastigi a );
   }

   abstract class SoyutArabaLastigi
   {
   }

   class MercedesE200 : SoyutArabaKasasi
   {
      public override void LastikTak(SoyutArabaLastigi lastik)
      {
         Console.WriteLine( lastik + " lastikli MercedesE200");
      }
   }


class FordFocus : SoyutArabaKasasi
{
    public override void LastikTak(SoyutArabaLastigi lastik)
    {
        Console.WriteLine( lastik + " lastikli FordFocus");
    }

}


class MercedesLastik : SoyutArabaLastigi
{

}

class FordLastik : SoyutArabaLastigi
{

}
```

- SoyutArabaKasasi sınıfındaki LastikTak() sınıfı fabrikadan üretilen ürünlerin birbirleriyle karıştırılmadan esnek bir şekilde nasıl ilişkilendirildiğini gösterilmektedir.
- Bu metot parametre olarak gerçek lastik ürünü yerine soyut lastik ürünü alır. Dolayısıyla herhangi bir fabrikadan üretilen lastik ürünü bu metoda parametre olarak geçirilebilir.

Abstract Factory Tasarım Deseni

```
using System; namespace DesignPattern
{
    class FabrikaOtomasyon
   {
      private SoyutArabaKasasi ArabaKasasi;
      private SoyutArabaLastigi ArabaLastigi;

      public FabrikaOtomasyon( SoyutArabaFabrikasi fabrika )
      {
         ArabaKasasi = fabrika.KasaUret();
         ArabaLastigi = fabrika.LastikUret();
      }

      public void LastikTak()
      {
         ArabaKasasi.LastikTak( ArabaLastigi );
      }
   }

   class UretimBandi
   {
      public static void Main()
      {
         SoyutArabaFabrikasi fabrika1 = new MercedesFabrikasi();
         FabrikaOtomasyon fo1 = new FabrikaOtomasyon( fabrika1 );
         fo1.LastikTak();

         SoyutArabaFabrikasi fabrika2 = new FordFabrikasi();
         FabrikaOtomasyon fo2 = new FabrikaOtomasyon( fabrika2 );
         fo2.LastikTak();
      }
   }
}
```

- FabrikaOtomasyon sınıfı kendisine verilen bir soyut fabrika nesnesi üzerinden kasa ve lastik üretir ve üretilen lastiği, lastiğin gerçek türünü bilmeden üretilen araba kasası ile ilişkilendirir.
- Dikkat ederseniz bu sınıf üretimin yapılacağı fabrikanın hangi fabrika olduğu ve üretilen ürünlerin gerçekte hangi ürünler olduğu ile ilgilenmez.

Abstract Factory Tasarım Deseni

```
using System; namespace DesignPattern
{
    class FabrikaOtomasyon
    {
        private SoyutArabaKasasi ArabaKasasi;
        private SoyutArabaLastigi ArabaLastigi;

        public FabrikaOtomasyon( SoyutArabaFabrikasi fabrika )
        {
            ArabaKasasi = fabrika.KasaUret();
            ArabaLastigi = fabrika.LastikUret();
        }

        public void LastikTak()
        {
            ArabaKasasi.LastikTak( ArabaLastigi );
        }
    }

    class UretimBandi
    {
        public static void Main()
        {
            SoyutArabaFabrikasi fabrika1 = new MercedesFabrikasi();
            FabrikaOtomasyon fo1 = new FabrikaOtomasyon( fabrika1 );
            fo1.LastikTak();

            SoyutArabaFabrikasi fabrika2 = new FordFabrikasi();
            FabrikaOtomasyon fo2 = new FabrikaOtomasyon( fabrika2 );
            fo2.LastikTak();
        }
    }
}
```

- Son olarak tasarladığımız bütün bu sınıfları test edecek UretimBandı sınıfını inceleyelim. Bu sınıf içerisinde ürünleri üretilecek fabrikanın soyut nesnesi oluşturulur ve FabrikaOtomasyonu nesnesine parametre olarak verilir.
- SoyutFabrika nesnesini alan FabrikaOtomasyonu bu nesnenin standart üretimden sorumlu metotlarını kullanarak kasa ve lastik üretir. Ardından SoyutArabaKasasi sınıfının LastikTak() metodunu kullanarak kasa ve lastik ürünlerini ilişkilendirir.

```
abstract class SoyutArabaFabrikasi
  {
     abstract public SoyutArabaKasasi KasaUret();
     abstract public SoyutArabaLastigi LastikUret();
  }
```

```
class MercedesFabrikasi : SoyutArabaFabrikasi
  {
     public override SoyutArabaKasasi KasaUret()
     {
        return new MercedesE200();
     }

     public override SoyutArabaLastigi LastikUret()
     {
        return new MercedesLastik();
     }
  }
```

```
class FordFabrikasi : SoyutArabaFabrikasi
  {
     public override SoyutArabaKasasi KasaUret()
     {
        return new FordFocus();
     }

     public override SoyutArabaLastigi LastikUret()
     {
        return new FordLastik();
     }
  }
```

```
abstract class SoyutArabaKasasi
  {
     abstract public void LastikTak(SoyutArabaLastigi a );
  }
```

```
class MercedesE200 : SoyutArabaKasasi
  {
     public override void LastikTak(SoyutArabaLastigi lastik)
     {
        Console.WriteLine( lastik + " lastikli MercedesE200");
     }
  }
```

```
class FordFocus : SoyutArabaKasasi
  {
     public override void LastikTak(SoyutArabaLastigi lastik)
     {
        Console.WriteLine( lastik + " lastikli FordFocus");
     }

  }
```

```
abstract class SoyutArabaLastigi
  {
  }
```

```
class MercedesLastik : SoyutArabaLastigi
  {

  }
```

```
class FordLastik : SoyutArabaLastigi
  {

  }
```

```
class FabrikaOtomasyon
  {
     private SoyutArabaKasasi ArabaKasasi;
     private SoyutArabaLastigi ArabaLastigi;

     public FabrikaOtomasyon( SoyutArabaFabrikasi fabrika )
     {
        ArabaKasasi = fabrika.KasaUret();
        ArabaLastigi = fabrika.LastikUret();
     }

     public void LastikTak()
     {
        ArabaKasasi.LastikTak( ArabaLastigi );
     }
  }
```

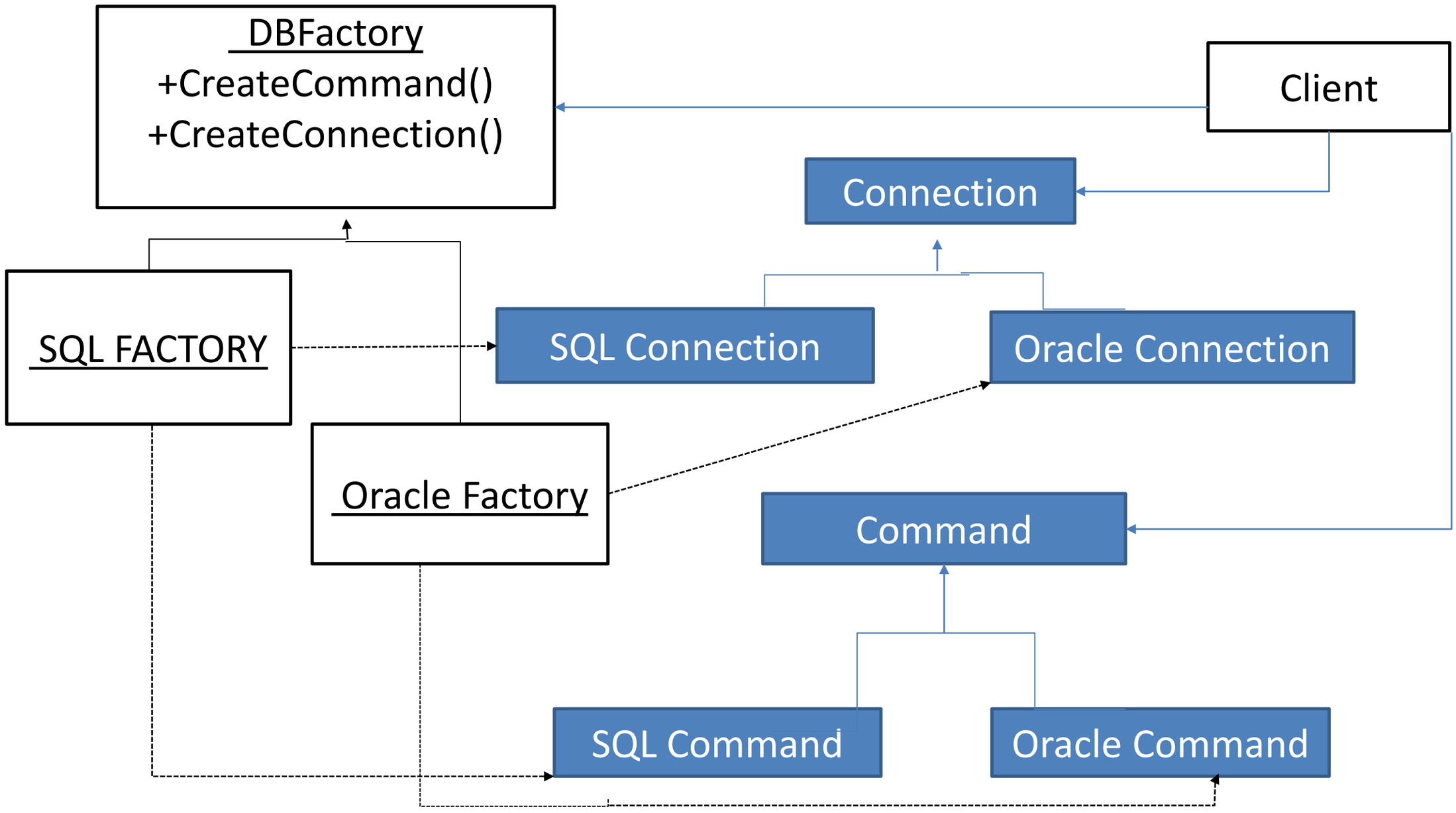
```
class UretimBandi
  {
     public static void Main()
     {
        SoyutArabaFabrikasi fabrika1 = new MercedesFabrikasi();
        FabrikaOtomasyon fo1 = new FabrikaOtomasyon( fabrika1 );
        fo1.LastikTak();

        SoyutArabaFabrikasi fabrika2 = new FordFabrikasi();
        FabrikaOtomasyon fo2 = new FabrikaOtomasyon( fabrika2 );
        fo2.LastikTak();
     }
  }
```

➢Son olarak tasarladığımız bütün bu sınıfları test edecek UretimBandı sınıfını inceleyelim. Bu sınıf içerisinde ürünleri üretilecek fabrikanın soyut nesnesi oluşturulur ve FabrikaOtomasyonu nesnesine parametre olarak verilir.

➢SoyutFabrika nesnesini alan FabrikaOtomasyonu bu nesnenin standart üretimden sorumlu metotlarını kullanarak kasa ve lastik üretir. Ardından SoyutArabaKasasi sınıfının LastikTak() metodunu kullanarak kasa ve lastik ürünlerini ilişkilendirir.

Yazılım Tasarımı ve Mimarisi



Yazılım Tasarımı ve Mimarisi

Örnekte çeşitli veritabanları üzerinde işlemler yapmak isteyen bir istemci nesne söz konusu olup bu nesnenin somut ürün ve somut fabrikalar konusunda herhangi bir bilgisi bulunmamaktadır. İstemci FactoryUtil sınıfının GetFactory() fonksiyonuna argüman olarak geçtiği veritabanı ismine göre farklı factory elde etmekte ve factory üzerinden yine soyut ürünleri kullanarak(kullandığını zannederek) istediği işlemleri yapmaktadır. Örneği inceleyiniz.

```
1   public class FactoryUtil
2   {
3       public static DBFactory GetFactory(string db_name)
4       {
5           if(db_name == "SQL")
6           {
7               return new SQLFactory();
8           }
9           if(db_name == "Oracle")
10          {
11              return new OracleFactory();
12          }
13          throw new Exception("error");
14      }
15  }
```

Yazılım Tasarımı ve Mimarisi

```
16  //ABSTRACT FACTORY
17  public abstract class DBFactory
18  {
19      public abstract Connection CreateConnection();
20      public abstract Command CreateCommand();
21  }
22  //Concreate Factory'ler
23  public class SQLFactory:DBFactory
24  {
25      public override Command CreateCommand()
26      {
27          return new SQLCommand();
28      }
29      public override Connection CreateConnection()
30      {
31          return new SQLConnection();
32      }
33  }
34  public class OracleFactory:DBFactory
35  {
36      public override Command CreateCommand()
37      {
38          return new OracleCommand();
39      }
40      public override Connection CreateConnection()
41      {
42          return new OracleConnection();
43      }
44  }
```

Yazılım Tasarımı ve Mimarisi

```
45  ////////////////////////////////////////////////////////////////
46  //Soyut Ürünler
47  ////////////////////////////////////////////////////////////////
48  public abstract class Connection
49  {
50      public abstract void Connect();
51      protected String connectionString;
52
53      public abstract String ConnectionString
54      {
55          get;
56          set;
57      }
58  }
59  public abstract class Command
60  {
61      public abstract void Execute();
62      protected String query;
63      public abstract String Query
64      {
65          get;
66          set;
67      }
68  }
```

Yazılım Tasarımı ve Mimarisi

```
69  ////////////////////////////////////////////////////////////////
70  //Somut Ürünler
71  ////////////////////////////////////////////////////////////////
72  public class SQLConnection : Connection
73  {
74      public override void Connect()
75      {
76          Console.WriteLine("SQL'e baglandi");
77      }
78      public override string ConnectionString
79      {
80          get
81          {
82              return base.connectionString;
83          }
84          set
85          {
86              base.connectionString=value;
87
88          }
89      }
90  }
```

Yazılım Tasarımı ve Mimarisi

```
91  public class OracleConnection:Connection
92  {
93      public override void Connect()
94      {
95          Console.WriteLine("Oracle'a baglandi");
96      }
97      public override string ConnectionString
98      {
99          get
100         {
101             return base.connectionString;
102         }
103         set
104         {
105             base.connectionString=value;
106
107         }
108     }
109 }
```

Yazılım Tasarımı ve Mimarisi

```
110     public class SQLCommand:Command
111     {
112         public override void Execute()
113         {
114             Console.WriteLine("T_SQL calisti");
115         }
116         public override string Query
117         {
118             get
119             {
120                 return base.query;
121             }
122             set
123             {
124                 base.query=value;
125 
126             }
127         }
128     }
```

Yazılım Tasarımı ve Mimarisi

```
129   public class OracleCommand:Command
130   {
131       public override void Execute()
132       {
133           Console.WriteLine("PL/SQL calisti");
134       }
135       public override string Query
136       {
137           get
138           {
139               return base.query;
140           }
141           set
142           {
143               base.query=value;
144
145           }
146       }
147   }
```

Yazılım Tasarımı ve Mimarisi

```
148  ////////////////////////////////////////////////////////////
149  //İstemci
150  ////////////////////////////////////////////////////////////
151  public class Client
152  {
153      public static void Main(String[] args)
154      {
155          DBFactory db=FactoryUtil.GetFactory("SQL");
156          Connection cnn=db.CreateConnection();
157          cnn.Connect();
158          Command cmd=db.CreateCommand();
159          cmd.Query="SELECT * FROM Tablo";
160          cmd.Execute();
161
162          db=FactoryUtil.GetFactory("Oracle");
163          cnn=db.CreateConnection();
164          cnn.Connect();
165
166          cmd=db.CreateCommand();
167          cmd.Query="SELECT * FROM Tablo";
168          cmd.Execute();
169      }
170  }
```

Abstract Factory Tasarım Deseni

"abstract factory" deseninin temel özelliklerini kısaca:

- "Abstract Factory", nesneleri oluşturan bir sınıftır. Oluşturulan bu nesneler birbirleriyle ilişkili olan nesnelerdir. Diğer bir deyişle aynı arayüzü uygulamış olan nesnelerdir.

- Üretilen nesnelerin kendisiyle ilgilenilmez. İlgilenilen nokta oluşturulacak nesnelerin sağladığı arayüzlerdir. Dolayısıyla aynı arayüzü uygulayan yeni nesneleri desene eklemek çok kolay ve esnektir.

- Bu desende üretilecek nesnelerin birbirleriyle ilişkili olması beklenir.

Abstract Factory Tasarım Deseni

- Genel olarak soyut fabrika deseni ;
- İlişkisi veya bağlılığı bulunan nesnelerin üretimi efektif ve kolay bir şeklide gerçekleşir.
- İstemci doğrudan somut sınıfı kullanmaz. Asıl ürüne erişmek istediğinde onun için tanımlı olan arayüz ile konuşur. Böylelikle ürün ailesindeki değişiklikler istemci kodunda herhangi bir değişikliğe ihtiyaç duymaz. Aynı zamanda daha test edilebilir bir yapı ortaya çıkmış olur.
- Asıl ürünlerden soyutlanmış sınıflar veya arayüzler üzerinden kullanıldığı için kolayca başka nesne grupları gerekli implementasyonlar yapıldığı takdirde desen içerisine ve üretime dahil edilmiş olur.

Abstract Factory Tasarım Deseni

- Birbiri ile ilişkileri olan nesneleri üretimini soyutlamak için sorumlu işlem metotlarında basit if-else, switch ifadelerini kullanarak da gerçekleştirebiliriz. Fakat bu tarz bir kodlama değişim ve ekleme işlemleri için sıkı kod yazılmasını beraberinde getirebileceği gibi işlem metotların sorumluluk alanının dışına çıkılmış olunur.
- Üretimde if-switch gibi alt seviye kodlar kullanılmadığı için değişimlere kolayca ayak uydurur.

Yazılım Tasarımı ve Mimarisi

BUİLDER TASARIM DESENİ

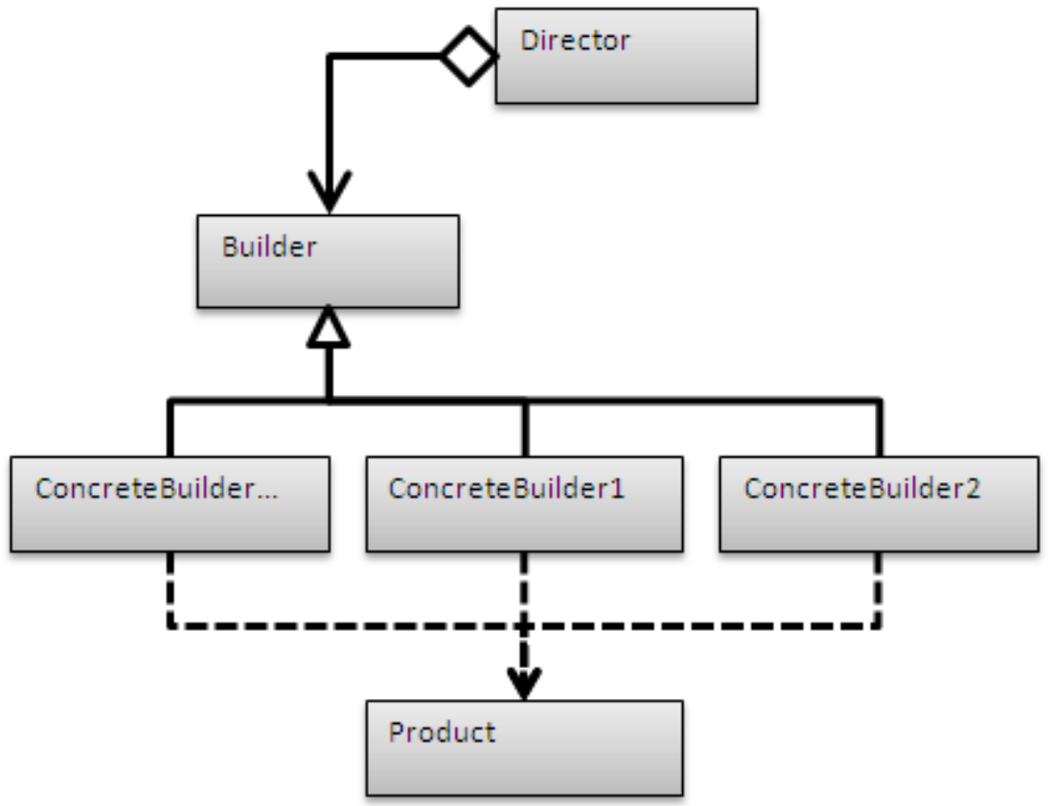
- Günlük hayatta bazı varlıkların pek çok alt parçanın birleşiminden oluştuğunu görülür. Örneğin bir bilgisayar ilk bakışta tek bir varlık gibi görünse de aslında klavye, ekran, kasa, fare gibi ana parçalardan oluşturulmuş, hatta onlarda kendi içlerinde başka alt parçalardan oluşturulmuştur.

- Yani bilgisayar tipik bir kompozit varlıktır ve bu durumun yazılım olarak modellenişine Builder deseni bir çözüm getirmektedir. Şüphesiz akla ilk gelen kompozit varlıklar bilgisayar gibi somut veya mekanik şeyler olsa da soyut varlıklar da kompozit olabilir. Örneğin bir sipariş nesnesinin, müşteri bileşeni, teslimat ve ürün bileşenleri olabilir.

Builder Patern'inin Uygulanması ile İlgili Esaslar

Bu paternin tam olarak yaptığı şey; aynı kompleks ürünün farklı parçalarla oluşturulup farklı sunumlarının elde edilebilmesini sağlamaktır.

Yazılım Tasarımı ve Mimarisi



Builder Tasarım Deseni

- "Builder" deseni adından da anlaşılacağı üzere bir nesnenin oluşturulması ile ilgilidir.

- Bu desende kullanılan yapılar "Abstract Factory" deseninde kullanılan yapılar ile çok benzerdir. Bu iki desen arasında çok küçük farklılıklar vardır.

- "Builder" deseni birden fazla parçadan oluşan kompleks yapıdaki bir nesnenin oluşturulmasını ve bu kompleks nesnenin oluşturulma safhalarını istemci modülünden tamamen gizlemek için kullanılır. Kompleks nesnenin yaratılması istemci modülünden tamamen yalıtıldığı için nesnenin yaratılması ile ilgili işlemler farklı versiyonlarda tamamen değiştirilebilir. Bu durum, istemci programın çalışmasını hiç bir şekilde etkilemeyecektir.

Builder Tasarım Deseni

- Burada dikkat edilmesi gereken nokta ise şudur : bu desen kompleks nesneyi oluşturan yapıların gerçek nesneden tamamen bağımsız bir yapıda olduğu durumlarda kullanıldığı zaman esneklik getirecektir.

- Dolayısıyla her bir farklı parçanın kompleks nesnede kullanılması, kompleks nesnenin işlevini değiştirmeyeceği gibi sadece görünümünü yada tipini değiştirecektir.

- Builder deseninin Abstract Factory deseninden farkına gelince; "Abstract Factory" deseninde soyut fabrika olarak bilinen yapının metotları fabrikaya ait nesnelerin yaratılmasından bizzat sorumludur.

Builder Tasarım Deseni

- Builder deseninde ise aynı mekanizma biraz daha farklı işlemektedir. Builder deseninde istemci modülü, nesnelerin ne şekilde oluşturulacağına soyut fabrika yapısına benzer bir yapı olan Builder yapısının metotları ile karar verir ve istemci oluşturulan bu nesneyi Builder sınıfından tekrar talep eder.

- Farklı donanım ürünlerinin bir araya getirilerek bilgisayar sistemlerinin oluşturulduğu bir teknik servisi göz önünde bulunduralım. Bir müşteri bu servise gelerek çeşitli özellikleri içinde barındıran bir bilgisayar talep eder. İsteği alan servis temsilcisi bu isteği teknik birimlere iletir ardından teknik eleman istenilen özelliklerde bir bilgisayarın oluşması için gerekli donanımları raflarından alır ve birleştirir.

Builder Tasarım Deseni

- Bizi ilgilendiren nokta müşterinin bilgisayarı oluşturan parçaların birleştirilmesinden tamamen soyutlandığıdır.

- Dikkat ederseniz verilen özelliklerde bilgisayarın oluşmasında müşterinin hiç bir etkisi olmamıştır. Eğer müşteri son anda vazgeçip farklı özelliklerde bir bilgisayar istemiş olsaydı yine bir etkisi olmayacaktı. Bu durumda eski bilgisayarın ilgili parçaları değiştirilip yeni isteğe uygun parçalar takılacaktı.

- Burada da kompleks bir yapı olan bilgisayar sisteminin kendisini oluşturan parçalardan (donanım) tamamen bağımsız bir yapıda olduğunu görüyoruz. Herhangi bir diskin yada monitörün değişmesi bilgisayarı bilgisayar yapmaktan çıkarmayacak sadece bilgisayarın tipini ve görüntüsünü değiştirecektir. Bu durum bizim yukarıda açıkladığımız builder deseninin amacına tam olarak uymaktadır.

Builder Tasarım Deseni

- Builder desenin de temel olarak 4 yapı bulunmaktadır. Bu yapılar sırasıyla şöyledir

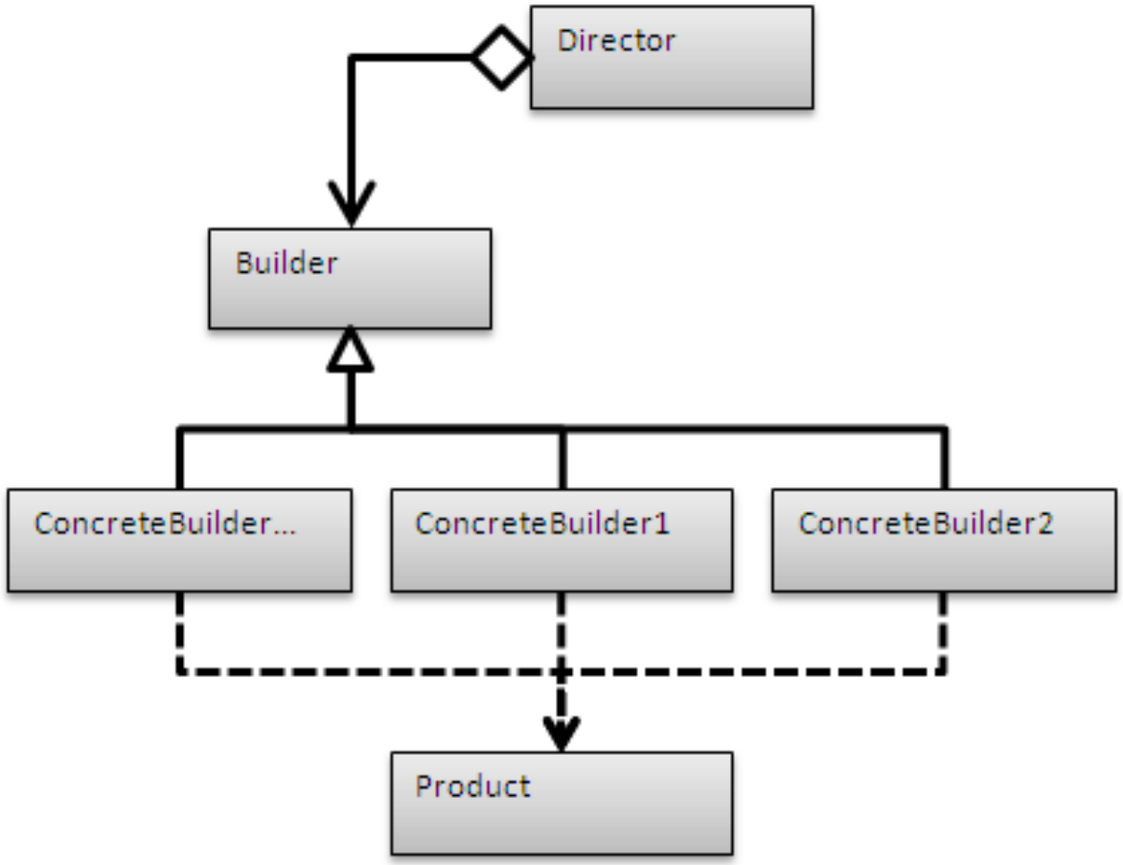
- **"Builder" Arayüzü** : Bu yapı içinde gerçek/somut(concrete) Builder sınıflarının uygulaması gereken özellikleri ve metotları barındırır. "Builder" abstract bir sınıf olabileceği gibi bir arayüz de olabilir. En çok tercih edilen yapı arayüz olmasına rağmen uygulamanızın ihtiyacına göre abstract sınıflarıda kullanabilirsiniz.

- Gerçek örneğimizde bu yapı bir bilgisayar sisteminin minimum olarak uygulaması gereken özellikleri temsil eder. Örneğin kullanıcılara satılacak her bilgisayar sistemi mutlaka HDD,RAM gibi yapıları içerisinde bulundurmalıdır. Dolayısıyla birbirinden farklı bilgisayar sistemlerinin bu tür yapıları mutlak olarak içerisinde barındırabilmesi için zorlayıcı bir etkene diğer bir değişle abstract üye elemanlara yada arayüzlere ihtiyacımız vardır.

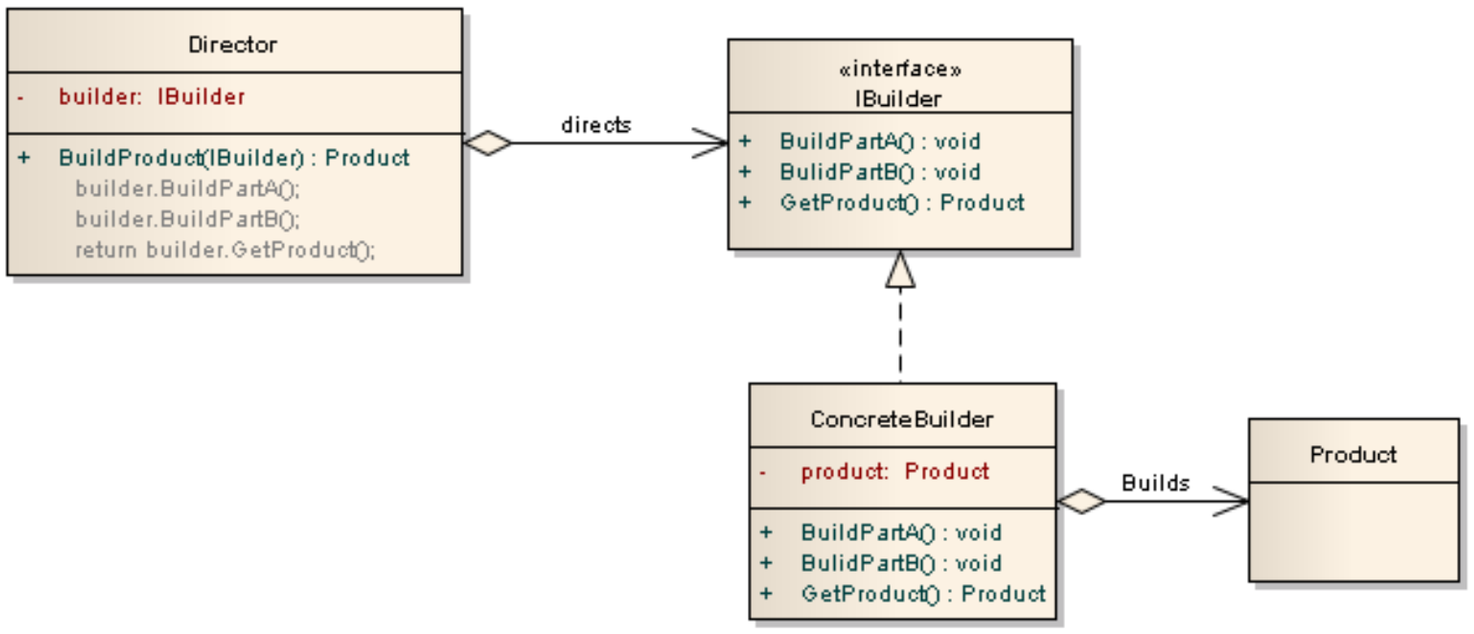
Builder Tasarım Deseni

- Builder desenindeki diğer 3 yapı sırasıyla şöyledir:

- **"Concrete (Somut) Builder" Yapısı** : Gerçek bir bilgisayar sistemini temsil eden ana yapımızdır. Farklı farklı donanımların bir araya getirilmesi ile çok sayıda sistem oluşturulabilir. Oluşturulabilecek her sistem gerçek bir Builder yapısına denk düşmektedir.
- **Product (Ürün)** : Gerçek bir bilgisayar sistemini ve bu sistemde hangi özelliklerin bulunduğunu somut olarak yapısında barındıran bir modeldir. Bu noktada bir önceki Concrete Builder yapısının parçaları oluşturduğunu, bu parçaların birleşmesinden oluşan kompleks nesnenin de Product olduğunu belirtmekte yarar var.
- **Director (Yönetici)** : Genellikle istemci ile yani örneğimizdeki müşteri ile Builder (parçaları birleştirip ürün yapan birimler) yapıları arasında köprü görevinde bulunan yapı olarak bilinir. Kısacası bilgisayar sisteminin oluşması için raflardan donanımları alıp onları birleştiren teknik eleman Director görevindedir. Tabi Director yapısının iş yaparken Builder arayüzünün kısıtlarına göre çalışması gerektiğini de belirtmek gerekir.

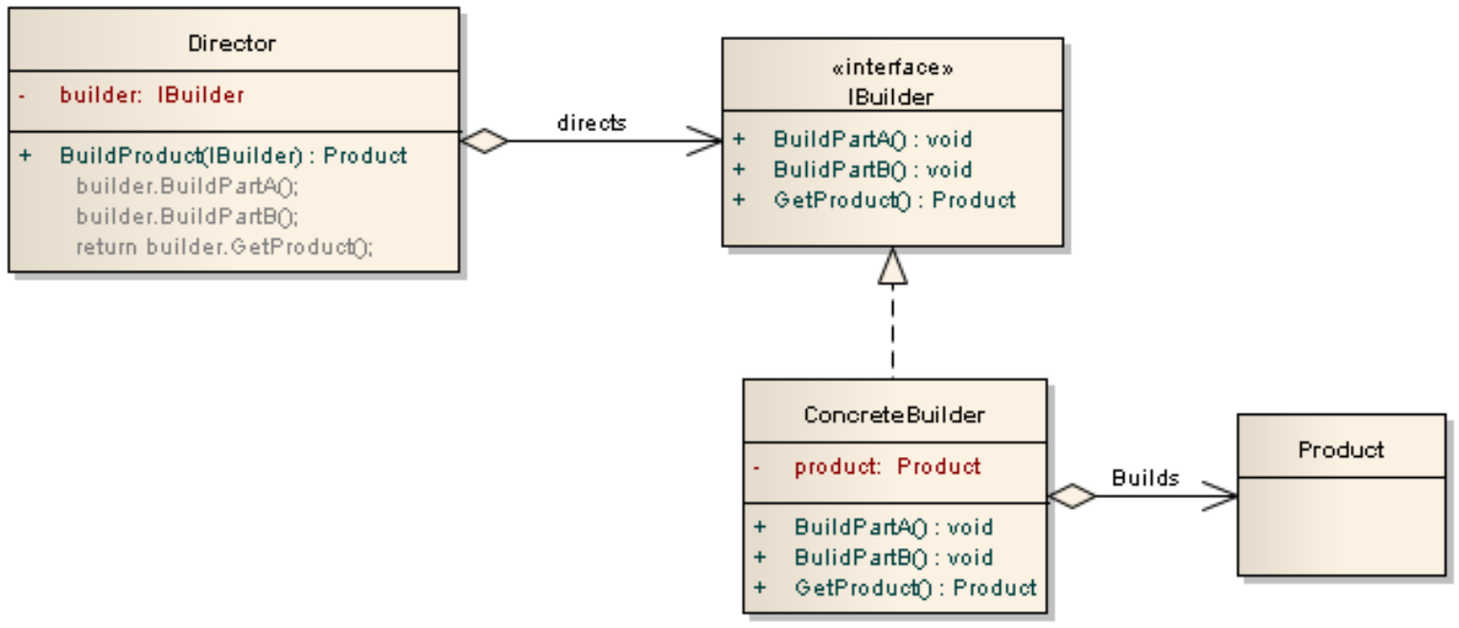


- **Builder: Product** nesnesinin oluşturulması için gerekli soyut arayüzü sunar.
- **ConcreteBuilder: Product** nesnesini oluşturur. Product ile ilişkili temel özellikleri de tesis eder ve **Product'** ın elde edilebilmesi için (istemci tarafından) gerekli arayüzü sunar.
- **Director: Builder** arayüzünü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



- Yukarıdaki UML diyagramı kısaca özetlenecek olursa : Builder arayüzünde bulunan metotlar yardımıyla gerçek builder sınıflarında kompleks nesnenin parçaları birleştirilerek istenilen nesne oluşturulur.

- Kompleks sistemi oluşturan parçalar farklı sayılarda olabildiği halde minumum gereklilikleri de sağlamak zorundadır. Bu minumum gereksinimler Builder arayüzünde bildirilmiştir.



- Director sınıfındaki bir metot, her sistem için standart olan Builder arayüzündeki metotları kullanarak hangi sistemin oluşturulduğundan bağımsız olarak kompleks nesnenin oluşturulması sağlanır.

- Yani Director hangi tür bir bilgisayar sisteminin oluşturulacağı ile ilgilenmez.

- Director *sınıfı sadece kendisine gönderilen parametre sayesinde Builder arayüzünün metotlarından haberdardır.*

- *Son* olarak istemci, Director sınıfının ilgili metotlarını kullanarak gerçek ürünün oluşmasını sağlar. Bunun için elbette Director sınıfının ilgili metoduna hangi ürünün oluşturulacağını bildirmesi gerekir.

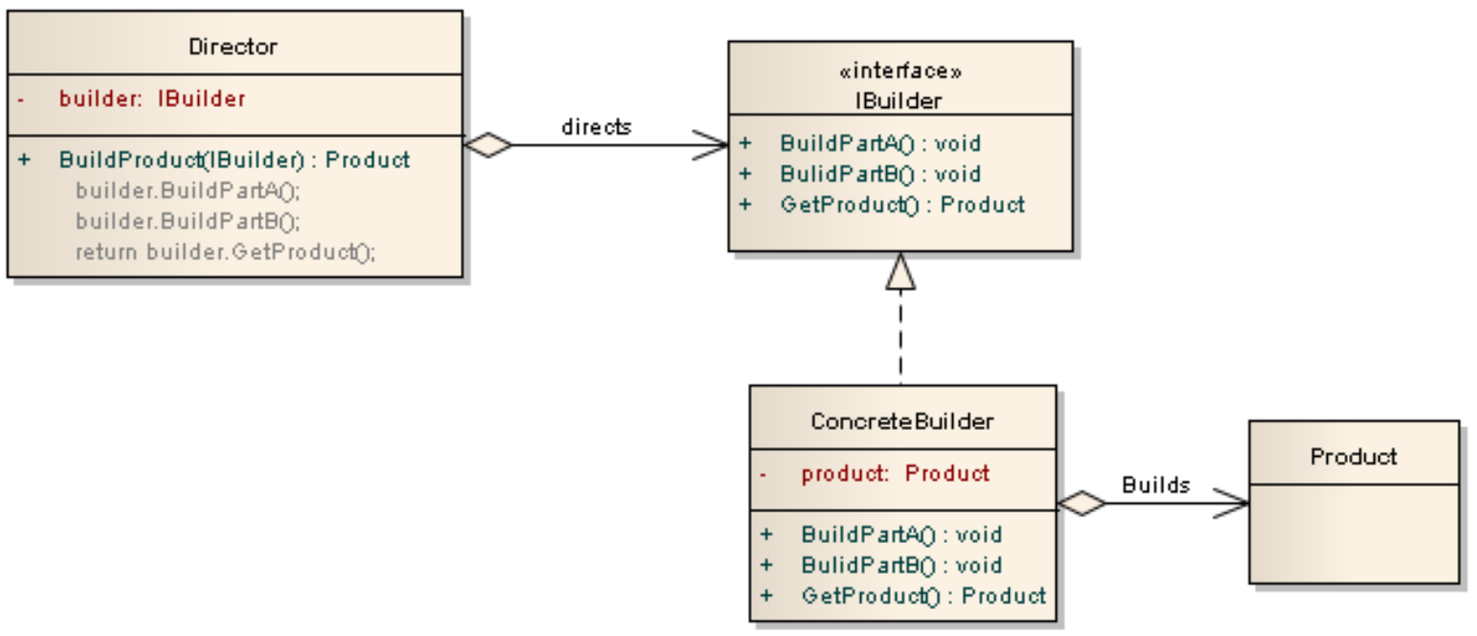
Ne Zaman Kullanılır?

- Builder tasarım deseni, kompleks bir nesneyi adım adım oluşturmak istediğimiz zaman kullanılan bir tekniktir.

- Eğer nesne birden fazla parçadan oluşuyorsa bu tasarım desenini kullanarak nesneden bir örnek oluşturabiliriz.

- Oluşturulması hedeflenen nesne birden fazla parçadan(nesneden) oluştuğu için nesneyi oluşturan parçalar yeni istekler geldikçe değişebilir ve artık ana nesnemizi oluşturmak çok karmaşık bir hale gelebilir.

- İşte builder tasarım deseni buna benzer karmaşıklıkları engellemek, genişleyebilirliği sağlamak için kullanılır.

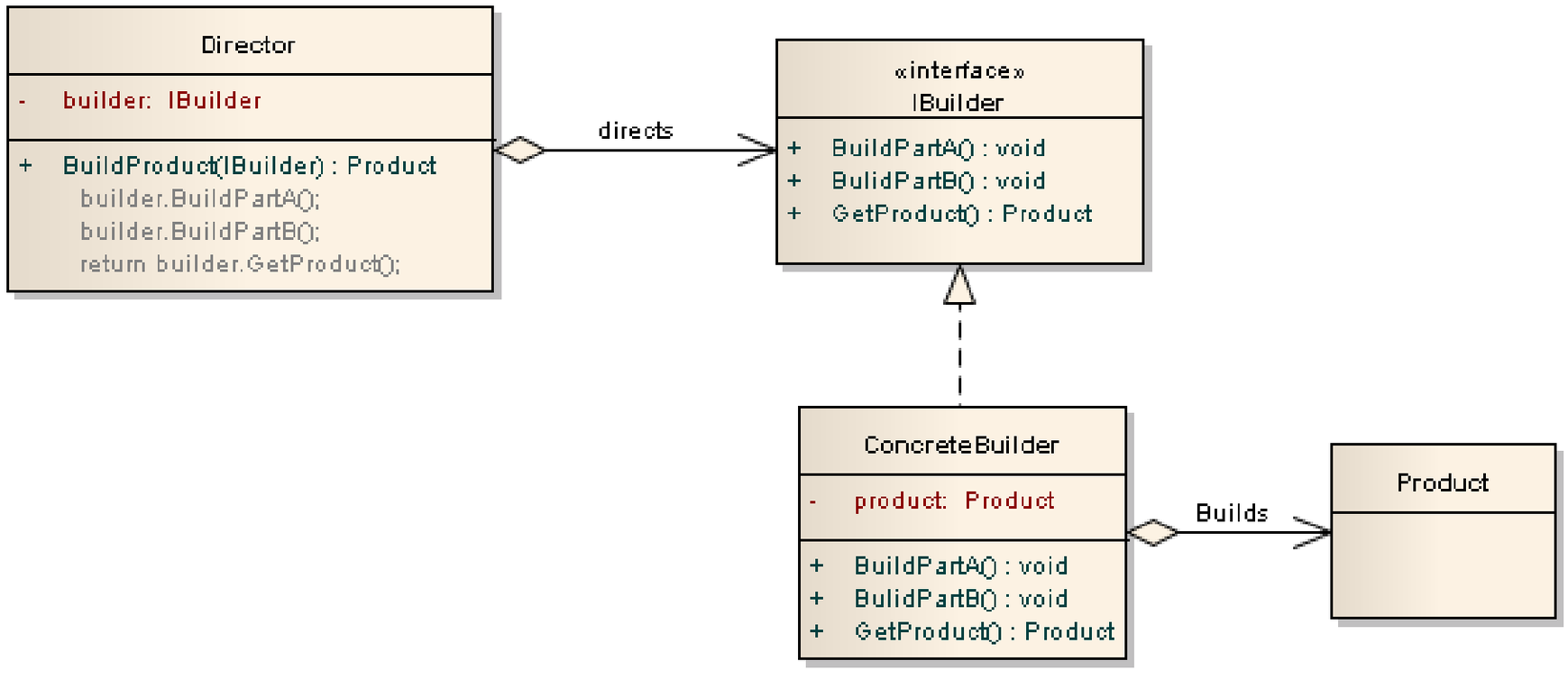


Ne Zaman Kullanılır?

- **Builder** deseni, karmaşık yapıdaki nesnelerin oluşturulmasında, istemcinin sadece nesne tipini belirterek üretimi gerçekleştirebilmesini sağlamak için kullanılmaktadır.

- Örneğin; eğer bir sınıfı oluşturduğumuzda yapıcı sayısının fazla olduğu görülürse bu deseni kullanmak gerektiği düşünülebilir.Fazla sayıda olan yapıcılar kod tekrarı yaptırdığı gibi sırası da akılda kalıcı olmadığından tercih edilmez.

- Encapsulation işlemi düşünülsün. Eğer bir nesnenin oluşturulması encapsüle edilmek istenirse kullanılabilir. Yani dışarıdan gizlemek için de builder deseni kullanılabilir.

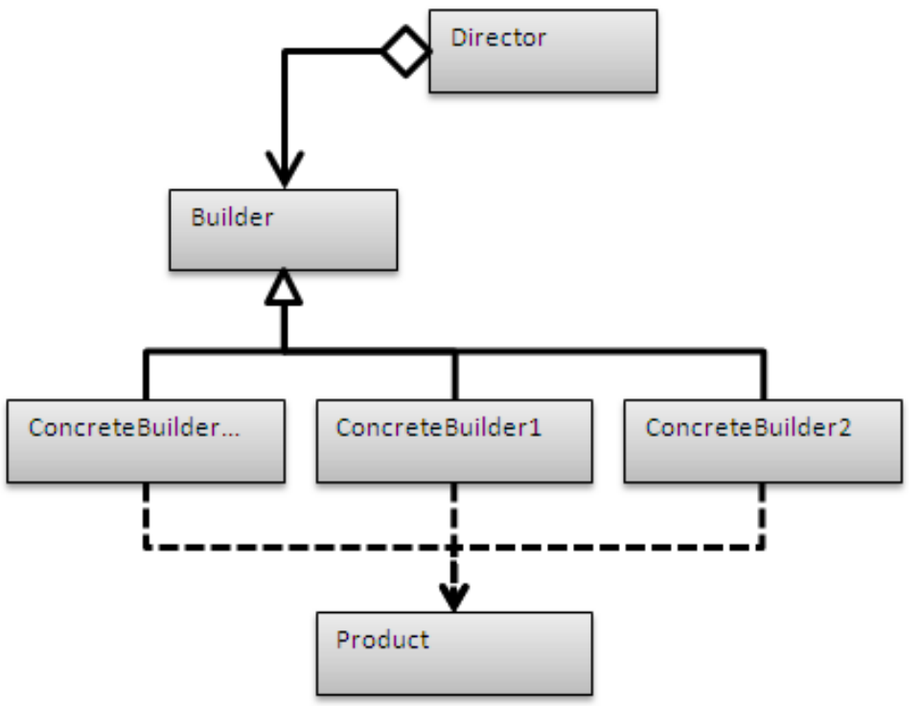


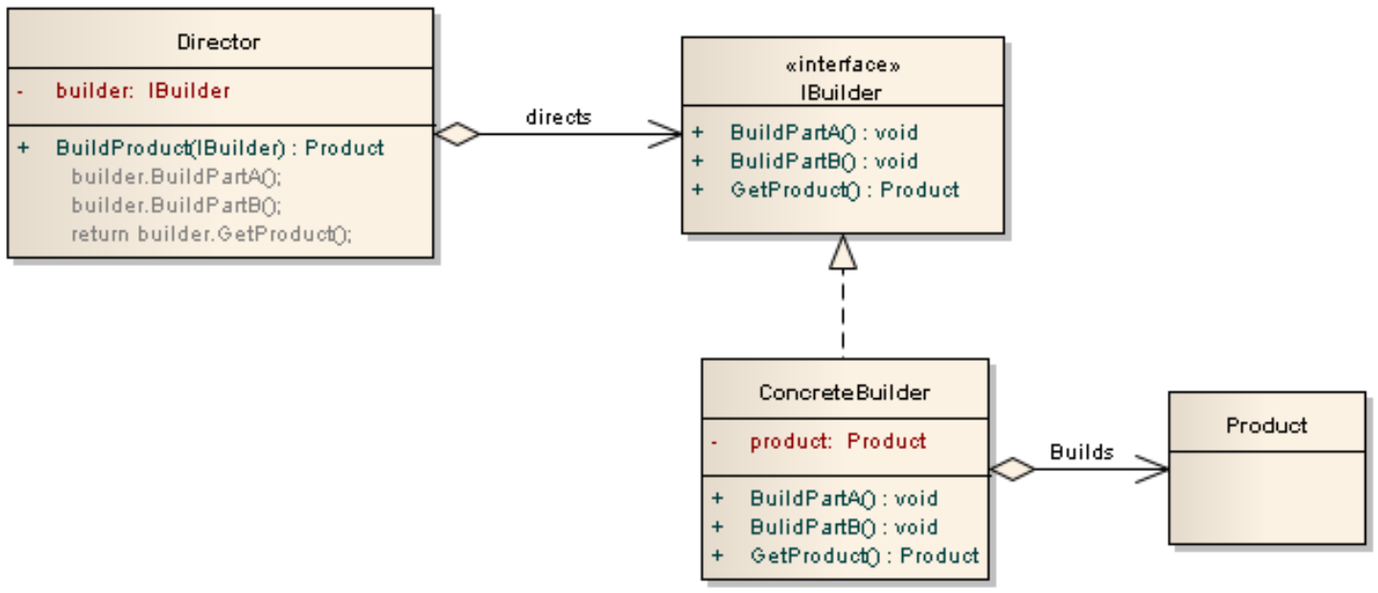
Builder deseninin Abstract Factory deseninden farkı;

- "Abstract Factory" deseninde soyut fabrika olarak bilinen yapının metotları fabrikaya ait nesnelerin yaratılmasından bizzat sorumludur.

- Builder deseninde ise aynı mekanizma biraz daha farklı işlemektedir. Builder deseninde istemci modülü, nesnelerin ne şekilde oluşturulacağına soyut fabrika yapısına benzer bir yapı olan Builder yapısının metotlarıyla karar verir ve istemci oluşturulan bu nesneyi Builder sınıfından tekrar talep eder.

- Builder deseni tek bir ürünü oluşturmaya yönelikken Abctract Factory belli bir ürün ailesini oluşturmayı hedefler.





Yazılım Tasarımı ve Mimarisi

Örnek: bilgisayar toplama örneğinin builder deseni ile tasarımı:

```
using System; namespace BuilderPattern
{
    public interface IBilgisayarToplayicisi
    {
        Bilgisayar Bilgisayar{get;}

        void CDROM_Olustur();
        void HDD_Olustur();
        void Monitor_Olustur();
        void RAM_Olustur();
    }
```

```
public class GoldPC : IBilgisayarToplayicisi
{
    private Bilgisayar mBilgisayar;

    public Bilgisayar Bilgisayar
    {
        get{return mBilgisayar;}
    }

    public GoldPC()
    {
        mBilgisayar = new Bilgisayar("Gold-PC");
    }

    public void CDROM_Olustur()
    {
        mBilgisayar["cdrom"] = "52X GoldStar";
    }

    public void HDD_Olustur()
    {
        mBilgisayar["hdd"] = "60 GB Seagate";
    }

    public void Monitor_Olustur()
    {
        mBilgisayar["monitor"] = "17' Hyundai";
    }

    public void RAM_Olustur()
    {
        mBilgisayar["ram"] = "512 MB DDR Kingston";
    }
}
```

```
public class SilverPC : IBilgisayarToplayicisi
{
    private Bilgisayar mBilgisayar;

    public Bilgisayar Bilgisayar
    {
        get{return mBilgisayar;}
    }

    public SilverPC()
    {
        mBilgisayar = new Bilgisayar("Silver-PC");
    }

    public void CDROM_Olustur()
    {
        mBilgisayar["cdrom"] = "48X Creative";
    }

    public void HDD_Olustur()
    {
        mBilgisayar["hdd"] = "30 GB Maxtor";
    }

    public void Monitor_Olustur()
    {
        mBilgisayar["monitor"] = "15' Vestel";
    }

    public void RAM_Olustur()
    {
        mBilgisayar["ram"] = "256 MB SD Kingston";
    }
}
```

```
public class Bilgisayar
{
    private string mBilgisayarTipi;
    private System.Collections.Hashtable mParcalar = new System.Collections.Hashtable();

    public Bilgisayar(string BilgisayarTipi)
    {
        mBilgisayarTipi = BilgisayarTipi;
    }

    public object this[string key]
    {
        get
        {
            return mParcalar[key];
        }
        set
        {
            mParcalar[key] = value;
        }
    }

    public void BilgisayariGoster()
    {
        Console.WriteLine("Bilgisayar Tipi : " + mBilgisayarTipi);
        Console.WriteLine("---> CD-ROM Model : " + mParcalar["cdrom"]);
        Console.WriteLine("---> Hard Disk Model : " + mParcalar["hdd"]);
        Console.WriteLine("---> Monitör Model : " + mParcalar["monitor"]);
        Console.WriteLine("---> RAM Model : " + mParcalar["ram"]);
    }
}
```

```
public class TeknikServis
{
    public void BilgisayarTopla(IBilgisayarToplayicisi bilgisayarToplayicisi)
    {
        bilgisayarToplayicisi.CDROM_Olustur();
        bilgisayarToplayicisi.HDD_Olustur();
        bilgisayarToplayicisi.Monitor_Olustur();
        bilgisayarToplayicisi.RAM_Olustur();
    }
}
```

```
using System; namespace BuilderPattern
{
    public interface IBilgisayarToplayicisi
    {
        Bilgisayar Bilgisayar{get;}

        void CDROM_Olustur();
        void HDD_Olustur();
        void Monitor_Olustur();
        void RAM_Olustur();
    }
```

Kaynak koddan da görüleceği üzere en temel yapı IBilgisayarToplayicisi arayüzüdür (Interface). Bu arayüzde bir Bilgisayar ürününü temsil etmek için bir özellik ve bilgisayarı oluşturan parçaları oluşturmak için gereken metotlar bulunmaktadır. Bu metotların bu arayüzden türeyen bütün sınıflar tarafından uygulanması gerekmektedir.

```
public class GoldPC : IBilgisayarToplayicisi
{
    private Bilgisayar mBilgisayar;

    public Bilgisayar Bilgisayar
    {
        get{return mBilgisayar;}
    }

    public GoldPC()
    {
        mBilgisayar = new Bilgisayar("Gold-PC");
    }

    public void CDROM_Olustur()
    {
        mBilgisayar["cdrom"] = "52X GoldStar";
    }

    public void HDD_Olustur()
    {
        mBilgisayar["hdd"] = "60 GB Seagate";
    }

    public void Monitor_Olustur()
    {
        mBilgisayar["monitor"] = "17' Hyundai";
    }

    public void RAM_Olustur()
    {
        mBilgisayar["ram"] = "512 MB DDR Kingston";
    }
}
```

Örnekte bir GoldPC ve SilverPC bilgisayar modelleri için gereken bileşenleri oluşturmak için 4 adet metot bulunmaktadır.

Burada en önemli nokta Bilgisayar tipinde bir özelliğinde bulunması. Bu özellik istendiği zaman oluşturulan Bilgisayar ürününü istemciye vermektedir. Zaten dikkat edilirse her bir sisteme ilişkin özellikler Bilgisayar sınıfındaki Hashtable nesnesinde saklanmıştır. Ayrıca her Bilgisayar nesnesinin ayrıca bir tip bilgisi saklanmaktadır.

```
public class TeknikServis
{
    public void BilgisayarTopla(IBilgisayarToplayicisi bilgisayarToplayicisi)
    {
        bilgisayarToplayicisi.CDROM_Olustur();
        bilgisayarToplayicisi.HDD_Olustur();
        bilgisayarToplayicisi.Monitor_Olustur();
        bilgisayarToplayicisi.RAM_Olustur();
    }
}
```

- TeknikServis isimli sınıf Builder desenindeki Director yapısına denk düşmektedir.

- Bu sınıftaki BilgisayarTopla metodu kendisine gönderilen bilgisayar toplayıcısı arayüzü referansına ait metotları kullanarak Bilgisayar nesnesinin parçalarını kendisi oluşturmaktadır.

Yazılım Tasarımı ve Mimarisi

Son olarak yukarıdaki yapıları kullanan bir istemci programı yazılıp desen test edilsin.

```
using System; namespace BuilderPattern
{
    class Class1
    {
        static void Main(string[] args)
        {
            TeknikServis teknikservis = new TeknikServis();

            IBilgisayarToplayicisi BT1 = new GoldPC();
            IBilgisayarToplayicisi BT2 = new SilverPC();

            teknikservis.BilgisayarTopla(BT1);
            teknikservis.BilgisayarTopla(BT2);

            BT1.Bilgisayar.BilgisayariGoster();
            Console.WriteLine("-------------");
            BT2.Bilgisayar.BilgisayariGoster();
        }
    }
}
```

Programı çalıştırıldığında aşağıdaki ekran görüntüsünü elde edilir.

```
Bilgisayar Tipi : Gold-PC
---> CD-ROM Model : 52X GoldStar
---> Hard Disk Model : 60 GB Seagate
---> Monitör Model : 17' Hyundai
---> RAM Model : 512 MB DDR Kingston
------------
Bilgisayar Tipi : Silver-PC
---> CD-ROM Model : 48X Creative
---> Hard Disk Model : 30 GB Maxtor
---> Monitör Model : 15' Vestel
---> RAM Model : 256 MB SD Kingston
```

ödev

- Üç farklı otomobilin üretimini sağlayan builder deseninin UML diyagramını çiziniz ve kodlayınız.
- Otomobil özellik ve işlevleri: Marka, Model, Vites, Koltuk, Tavan

```
//Product Class
class Araba
{
    public string Marka { get; set; }
    public string Model { get; set; }
    public double KM { get; set; }
    public bool Vites { get; set; }
    public override string ToString()
    {
        return $"{Marka} marka araba {Model} modelinde {KM} kilometrede {Vites} vites olarak üretilmiştir.";
    }
}
```

Burada oluşturmak istediğimiz ürün araba olacağı için Product nesnemiz ***Araba*** sınıfı olacaktır. Arabayı ürettirecek olan Opel, Ford, Mercedes vs. gibi birbirlerinden farklı özellik teşkil edecek olan tüm sınıflarımız ConcreteBuilder sınıfımız olacaktır.

```
//Builder
abstract class IArabaBuilder
{
    protected Araba araba;
    public Araba Araba
    {
        get { return araba; }
    }

    public abstract void SetMarka();
    public abstract void SetModel();
    public abstract void SetKM();
    public abstract void SetVites();
}
```

Burada dikkat etmeniz gereken husus, Builder sınıfımızda Araba referansını protected olarak işaretlememizdir. Bunun sebebi, bu Builder Class'ın uygulanacağı Derived Class'lardan bu fielda erişilebilmesi içindir.
Ardından ConcreteBuilder nesnelerimizi oluşturalım.

ConcreteBuilder nesnelerimizi oluşturalım.

```
//ConcreteBuilder Class
class OpelConcreteBuilder : IArabaBuilder
{
    public OpelConcreteBuilder()
    {
        araba = new Araba();
    }
    public override void SetKM() => araba.KM = 100;
    public override void SetMarka() => araba.Marka = "Opel";
    public override void SetModel() => araba.Model = "Astra";
    public override void SetVites() => araba.Vites = true;
}
//ConcreteBuilder Class
class ToyotaConcreteBuilder : IArabaBuilder
{
    public ToyotaConcreteBuilder()
    {
        araba = new Araba();
    }
    public override void SetKM() => araba.KM = 150;
    public override void SetMarka() => araba.Marka = "Toyota";
    public override void SetModel() => araba.Model = "Corolla";
    public override void SetVites() => araba.Vites = true;
}
//ConcreteBuilder Class
class BMWConcreteBuilder : IArabaBuilder
{
    public BMWConcreteBuilder()
    {
        araba = new Araba();
    }
    public override void SetKM() => araba.KM = 25;
    public override void SetMarka() => araba.Marka = "BMW";
    public override void SetModel() => araba.Model = "X Bilmem Kaç";
    public override void SetVites() => araba.Vites = true;
}
```

,Director Class'ı oluşturalım

```
//Director Class
class ArabaUret
{
    public void Uret(IArabaBuilder Araba)
    {
        Araba.SetKM();
        Araba.SetMarka();
        Araba.SetModel();
        Araba.SetVites();
    }
}
```

Şimdi bir Client tarafından araba talebinde bulunabiliriz

```
//Client
class Program
{
    static void Main(string[] args)
    {
        IArabaBuilder araba = new OpelConcreteBuilder();
        ArabaUret uret = new ArabaUret();
        uret.Uret(araba);
        Console.WriteLine(araba.Araba.ToString());

        araba = new ToyotaConcreteBuilder();
        uret.Uret(araba);
        Console.WriteLine(araba.Araba.ToString());

        Console.Read();
    }
}
```

Yazılım Tasarımı ve Mimarisi

KAYNAKLAR

1. C# ile Tasarım Desenleri ve Mimarileri, Pusula Yayıncılık, Ocak 2015, Ali Kaya ve Engin Bulut.
2. Yazılım Mimarının El Kitabı, C++ Java ve C# ile Uml ve Dizayn Paternleri, Pusula Yayıncılık, Eylül 2014, Aykut Taşdelen
3. http://kodcu.com/2015/01/singleton-tasarim-deseni/
4. Sefer Algan http://www.csharpnedir.com/articles/read/?id=134
5. osxdaily.com
6. http://kutluarasli.blogspot.com.tr/2009/08/fabrika-tasarm-deseni-factory.html
7. http://kodcu.com/2015/01/factory-tasarim-deseni/
8. http://www.codesenior.com/tutorial/Fabrika-Metod-Factory-Method-Tasarim-Deseni
9. Sefer Algan http://www.csharpnedir.com/articles/read/?id=146
10. Sefer Algan http://www.csharpnedir.com/articles/read/?id=154