



YMT 312-Yazılım Tasarım Ve Mimarisi Tasarımı Şablonu-NYP

Dr. Öğr. Üyesi Fatih ÖZYURT

Fırat Üniversitesi Yazılım Mühendisliği Bölümü

Bölüm-4

Yazılım Tasarımının Önemi

- Tasarlanmış (designed) bir dünyada yaşıyoruz.
- Tasarım ekonomik olarak öneme sahiptir ve yaşam kalitemizi doğrudan etkiler.
- Yazılım son derece yaygın hale gelmektedir.
- Yazılım tasarımının kalitesinin önemli sonuçları olmaktadır ve yazılım tasarımcıları bunların farkında olmalı, bunları ciddiye almalıdır.

Tasarım Şablonu

- Yazılım esnasında **tekrar eden sorunları** çözmek için kullanılan ve **tekrar kullanılabilir** yapıda kod yazılımını destekleyen, bir ya da birden fazla sınıftan oluşmuş **modül** ve **program** parçalarına tasarım şablonu (design pattern) ismi verilir.
- Tasarım şablonları aşağıda yer alan ortak özelliklere sahiptirler:
 - Edinilen tecrübeler sonunda ortaya çıkmışlardır
 - Tekerin tekrar icat edilmesini önlerler
 - Tekrar kullanılabilir kalıplardır.
 - Ortak kullanılarak daha büyük problemlerin çözülmesine katkı sağlarlar
 - Devamlı geliştirilerek, genel bir çözüm olmaları için çaba sarfedilir.

Tasarım Kavramları

Soyutlama (abstraction):

- **Soyutlama (abstraction):** Detayları gizleyerek yukarıdan bakabilme imkanı sağlar.



Soyutlama kavramı veri, işlev ve yapısal açılar için geçerlidir. Örneğin bir kapı nesne olarak ele alındığında onun kulpu, rengi menteşeleri, malzemesi gibi detayları düşünmeden kapıyı bir ev mimarisi içinde değerlendirebiliriz. Aksi taktirde diğer detaylara yoğunlaşan bir tasarımcı 'oda' düzeyinde görsel canlandırmalara hakim olamaz.

Tasarım Kavramları

İyileştirme (enhancement):

- **İyileştirme (enhancement):** Soyutlama düzeyinde irdeleme bittikten sonra, daha alt seviyelere inilerek tanımlamalarda ayrıntı, bazen de düzeltme yapılarak **tasarımın daha kesinlik kazanması sağlanır.**

Kulp	Menteşe
Renk	Malzeme

Tasarım Kavramları

Modülerlik (modularity):

- **Modülerlik (modularity):** Sistemi istenen kalite faktörleri ışığında parçalara ayırıştırma sonucu elde edilir. Bir işlev için sistemin tümü değil, ayrılmış bir kısmı üzerinde çalışma yapabilme olanağı sağlar.

Yuvarlak Köşeli Uzun Kısa ...	Yay Dil Vidalar ...
Ahşap Metal Cam ...	Beyaz Metalik Kahverengi ...

Modülerlik

- Bütün karmaşıklığın tek bir modülde toplanması yerine, anlaşılabilir ve dolayısıyla projenin zihinsel kontrol altında tutulması için sistem bir çok modüle ayrılır.
- Modüller, isimleri olan tanımlanmış işlevleri bulunan ve hedef sistemi gerçekleştirmek üzere tümleştirilen birimlerdir.



Tasarım Şablonu Neden Kullanılır?

Her tasarım şablonunun belirli bir ismi vardır ve bu isim kullanıldığı zaman hangi tasarım şablonundan bahsedildiği hemen anlaşılır.

- Yazılım ekibinin kullanacağı ortak bir kelime hazinesi oluşur.
- Programcılar takım içinde tasarım şablonlarının isimlerini kullanarak, hangi sorunlar üzerinde çalıştıklarını kolaylıkla anlatabilirler.
- Tasarım şablonlarını tanımayan programcılar için duydukları tasarım şablonlarını öğrenmeye yönlendirecek bir motivasyon kaynağı oluşturur.
- Nesneler seviyesinde sorunları çözmek her zaman kolay olmayabilir, lakin tasarım kalıpları seviyesinde düşünüldüğü zaman, problem çözüm işlemi kolaylaşır.

Tasarım Şablonları Nasıl Kullanılmalıdır?

Eğer sürekli mevcut kodu deęiřtirerek, **uygulamaya yeni özellikler** kazandırmak zorundaysanız, belki bir tasarım şablonu kullanma vakti gelmiş olabilir.

Tasarım şablonlarını uygulamak için mümkün olan her fırsatı deęerlendirmek ya da fırsat aramak yerine, öncelikle tasarım şablonu uygulayışının ne kadar gerekli olduęu sorgulanmalıdır.



Tasarım Şablon Çeşitleri

Oluşturucu tasarım şablonları

1. Factory
2. Factory Method
3. Abstract Factory
4. Singleton
5. Builder
6. Prototype
7. Object Pool

Tasarım Şablon Çeşitleri

Yapısal tasarım şablonları

- 1) Class Adapter
- 2) Object Adapter
- 3) Bridge
- 4) Facade
- 5) Composite
- 6) Decorator

Davranışsal tasarım şablonları

1. Command
2. Iterator
3. Memento
4. State
5. Observer
6. Strategy
7. Chain Of Responsibility
8. Mediator
9. Visitor
10. Template Method
11. Interpreter

Nesneye Yönelik Programlama - Object Oriented Programming

Geniş tabanlı uygulamalarda oluşan kodun mantıklı bir çerçevede organize edilebilmesi için sadece metotların kullanımı yeterli değildir.

Kodsal ilişkileri ifade etmek ve belli kalıplar oluşturabilmek için metotlar haricinde başka bir mekanizmaya daha ihtiyaç duymaktayız. Böyle bir mekanizmayı nesneye yönelik programlama (object oriented programming) tekniği sunmaktadır.

Örnek:

Müşterimiz ürün satmak amacıyla bir e-satış sistemine ihtiyaç duymaktadır. Müşteri ile yaptığımız görüşmelerde aşağıdaki terimlerin kullanıldığını fark ettik:

1. Müşteri
2. Ürün
3. Sipariş
4. Sepet
5. Ödeme
6. İade
7. Adres

```
// Kod 1
```

```
public class Musteri{  
    private String isim;  
    private String soyad;  
}
```

Bir nesneyi nesne yapan bilgiler sınıf bünyesinde sınıf değişkenlerinde tutulmaktadır. Örneğin kod 1 de bir müşteriyi temsil etmek amacıyla İsim ve Soyad gibi değişkenler tanımlanmıştır. Bu sınıftan bir müşteri **nesnesini** şu şekilde oluşturabiliriz:

```
// Kod 2
```

```
Musteri müsteri = new Musteri();
```

Sınıflar programcılar tarafından kullanılan dili genişletmek için oluşturulan veri tipleridir. Örneğin int ya da String nasıl bir veri tipi ise ve bir değişken tanımlamak için kullanılabiliyorsa, Musteri de bir veri tipidir ve müşteri nesnelerin

```
// Kod 3
```

```
Musteri musteril = new Musteri();  
Müşteri musteril2 = musteril;
```

Sınıflar değişkenler haricinde metotlara sahip olabilirler.

Bu metotlar sınıf değişkenlerine olan erişimi sağlamak ve sınıf değişkenlerinin sahip oldukları değerleri değiştirmek için kullanılabilirler. Bu metotlar bünyesinde ayrıca iş mantığı implemente edilir.

Kod 4 de yer alan örnekte isim değişkenine olan erişim getIsim(), isim değişkeni üzerindeki değişiklikler setIsim() metodu aracılığı ile yapılmaktadır.

// Kod 4

```
public class Musteri{  
    private String isim;  
    private String soyad;  
  
    public String getIsim(){  
        return this.isim;  
    }  
  
    public void setString(String isim){  
        this.isim = isim;  
    }  
}
```


Burada nesneye yönelik programlamanın temel bir prensibini görmekteyiz.

Nesnelerin iç dünyalarını oluşturan sınıf değişkenlerine doğrudan erişim mümkün olmamalıdır.

Aşağıdaki yapı nesneye yönelik programlamaya aykırı bir durumdur, çünkü sınıf değişkenleri public olarak tanımlanmıştır ve bu değişkenler doğrudan erişim ile değiştirilebilirler.

```
// Kod 5
```

```
public class Musteri{  
    public String isim;  
    public String soyad;  
}
```

```
Musteri muster1 = new Musteri();  
muster1.isim = "Ali";
```

Nesneler iç dünyalarını yani sahip oldukları değişken değerlerini koruma hakkına sahiptirler. Bu değişkenlere olan erişim kontrollü bir şekilde yapılmalıdır. Kod 6 da bu erişimin sınıf metotları aracılığı ile nasıl yapıldığını görmekteyiz.

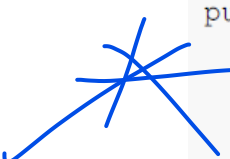
```
// Kod 6
```

```
Musteri musteri = new Musteri();  
String isim = musteri.getIsim();  
String yeniIsim = "Ahmet";  
musteri.setIsim(yeniIsim);
```

Interface Nedir?

```
public interface Tasit {  
  
    public String getMarka();  
  
}
```

```
public class Audi implements Tasit{  
  
    public String getMarka() {  
        return "Audi A4";  
    }  
  
}
```



```
public class TasitTest2{  
  
    public static void main(String args[]){  
        Tasit tasit = new Audi();  
        System.out.println(tasit.getMarka());  
    }  
  
}
```

Interface Nedir?

1. Bir interface sadece dış dünyaya sunulacak hizmetleri tanımlar.
2. Bu hizmetlerin nasıl yerine getirileceğine interface sınıfını implemente eden altsınıflar karar verir.
3. Interface kullanıcısı genelde hangi altsınıf üzerinden gereken hizmeti aldığını bilmez, bilmek zorunda değildir.
4. Bu sayede „loose coupling“ olarak tabir edilen, servis sağlayıcı ve kullanıcı arasında gevşek bir bağımlılık oluşturulmuş olur.

Abstract Nedir?

```
public abstract class BinekOto {  
    private String marka;  
    private int ueretimYili;  
    private int vitesAdedi;  
  
    public abstract String getMarka();  
  
    public int getUeretimYili(){  
        return this.ueretimYili;  
    }  
}
```

```
public class Ford extends BinekOto{  
  
    public String getMarka() {  
        return "Ford Focus";  
    }  
}
```

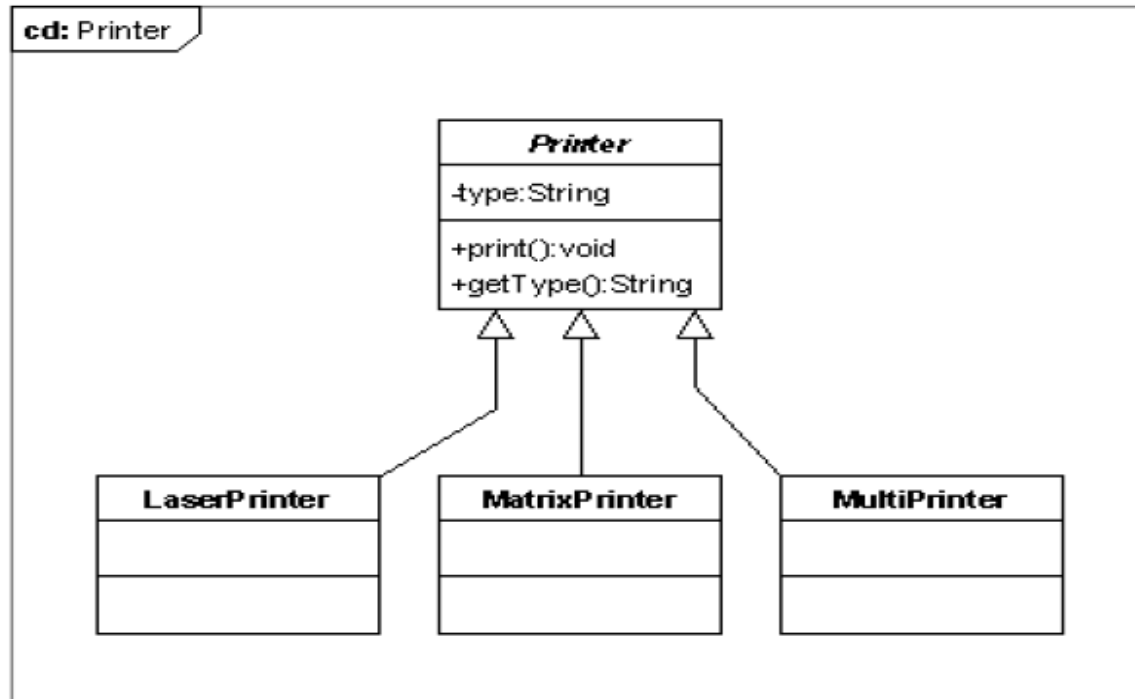
Abstract Nedir?

1. BinekOto sınıfında getMarka() soyut olarak tanımlandığı için altsınıflarda mutlaka getMarka() isminde implementasyonu yapılmış bir metodun bulunması gerekmektedir.
2. Zorunda olmadan BinekOto sınıfından getUretimYili() metodunu kullanabilir.

* Abstract Class ve Interface Arasındaki Farklar *

ABSTRACT CLASS	INTERFACE
f Constructor içerebilir.	Constructor içeremez.
x Static üyeler içerebilir.	Static üyeler içeremez.
OK Farklı tiplerde access modifier (erişim belirleyicisi) içerebilir. public, private, protected gibi.	f Farklı tiplerde access modifier içeremez. Interface'te tanımlanan her metod default olarak public kabul edilir.
x Sınıfın ait olduğu kimliği belirtmek için kullanılır. (is-a ilişkisi)	* Sınıfın yapabileceği kabiliyetleri belirtmek için kullanılır. (can-do ilişkisi)
f Bir sınıf sadece bir tane abstract sınıfı inherit edebilir.	f Bir sınıf birden fazla interface'i inherit edebilir.
x Eğer birçok sınıf aynı türden ve ortak davranışlar sergiliyorsa abstract sınıfı base class olarak kullanmak doğru olacaktır.	x Eğer birçok sınıf yalnızca ortak metodları kullanıyor ise interface'ten türetilmeleri doğru olacaktır.
Abstract sınıf metod, fields, contents vb. üyeleri içerebilir.	f Interface yalnızca metod imzalarını içerebilir.
Türetilen sınıflar abstract sınıfı tamamen veya kısmi implemente edebilir.	Türetilen sınıflar interface'i tamamen implement etmek zorundadır.
x Metod imzaları veya implementasyonları içerebilir.	* Yalnızca metod imzalarını içerebilir.

Neden Abstract ve Interface Sınıflar Yeterli Değildir

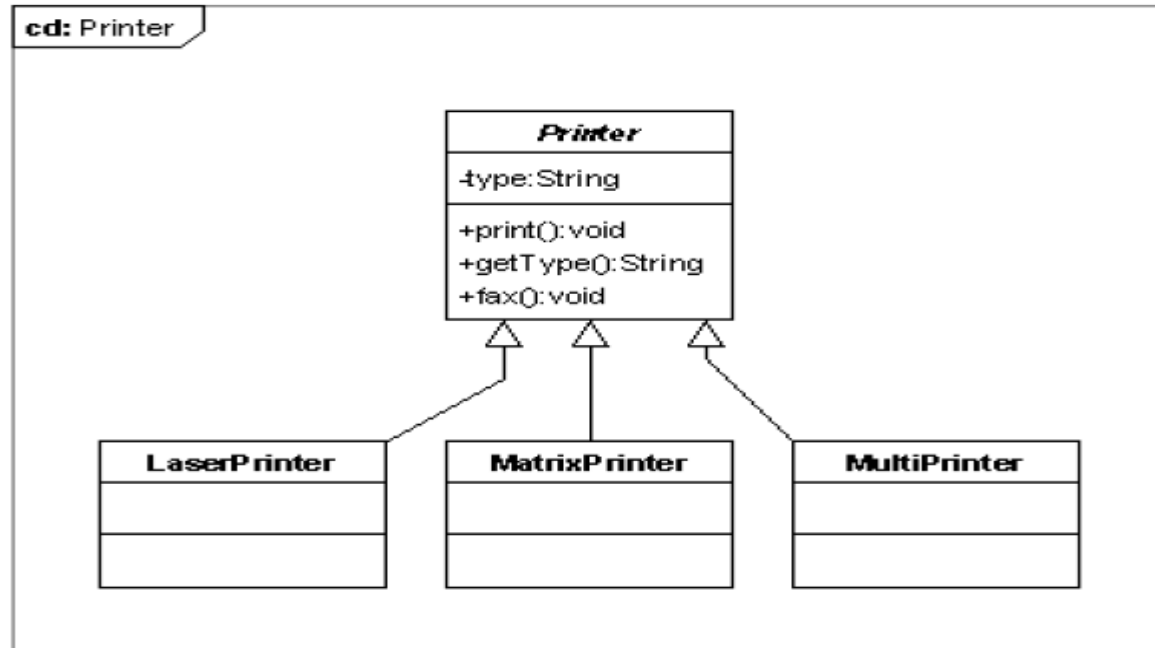


Printer isminde genel anlamda bir yazıcıyı modelleyen bir soyut sınıf oluşturuyoruz. Amacımız değişik tipteki yazıcılar için bir üstsınıf oluşturmak ve ortak olan özellikleri bu sınıfın bünyesinde toplamak.

Devam...

1. Yazılım yaparken amacımız, yazdığımız kodun ilerde bakımının kolay ve kolaylıkla değiştirilebilir ve genişletilebilir halde olmasını sağlamaktır.
2. Zamanla değişik yapıda ve modelde yazıcı sistemlerinin piyasaya sürüleceği bir gerçektir. Peki yaptığımız model bu gelişmeleri bünyesine katabilecek yapıda mıdır?
3. Modelimizde MultiPrinter isminde, faks gönderme fonksiyonuna sahip bir yazıcı bulunmaktadır.
4. Faks gönderebilmek için fax() isminde bir metodun oluşturulması gerekiyor.

Devam...



Altsınıflara tek tek fax() metodunu eklemek ve implemente etmek istemediğimiz için Printer sınıfına fax() isminde bir metot ekliyoruz. Implemente ettiğimiz fax() metodunu tüm altsınıflar kullanabilir.

Bu durumda faks gönderme özelliğine sahip olmayan LaserPrinter ya da MatrixPrinter fax() gönderme metoduna sahip olarak, faks gönderme özelliğine kavuşuyorlar.

Devam...

ÇÖZÜM!!!!

Bu sorunu fax() metodunu altsınıflarda, yazıcının yapısına göre tekrar implemente ederek çözebiliriz:

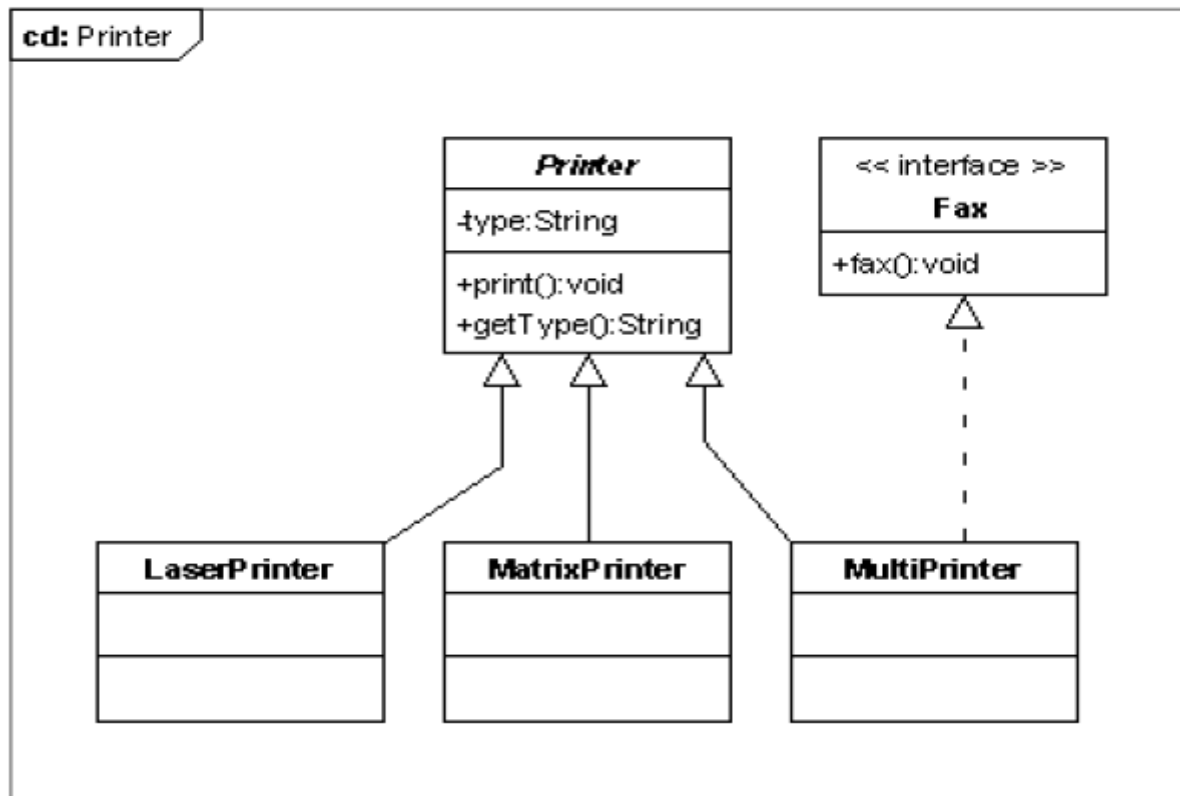
```
public class LaserPrinter extends Printer{
    @Override
    public void fax()
    {
        // LaserPrinter fax gönderemediği için bu metot
        // bos kalmak zorundadır.
    }
}
```

```
public class MultiPrinter extends Printer{
    @Override
    public void fax()
    {
        System.out.println("Fax sent with " + getType());
    }
}
```

Sadece fax() göndermek için her alt sınıf içinde değişiklik yapmamız gerekecek!!!

Interface sınıf kullanarak çözebilir miyiz?

Peki bu sorunu bir interface sınıf kullanarak çözebilir miyiz?



Fax() metodu sadece bazı yazıcılar tarafından desteklendiği için Fax isminde bir interface sınıfı oluşturarak, faks gönderebilecek yazıcıların bu sınıfı implemente etmesini sağlıyoruz. Böylece gerçekten faks gönderebilen yazıcılar bu özelliğe kavuşuyor.

```
public interface Fax {  
    void fax();  
}  
  
public class MultiPrinter extends Printer implements Fax{  
    public void fax()  
    {  
        System.out.println("Fax sent with " + getType());  
    }  
}
```

Peki sizce bu iyi bir çözüm müdür? Bizi bekleyen sorunları kestirebiliyor musunuz? Faks gönderme işlemini değiştirmek zorunda kaldığımızı düşünelim, bu durumda sistemde ne gibi değişiklikler yapılması gerekiyor.

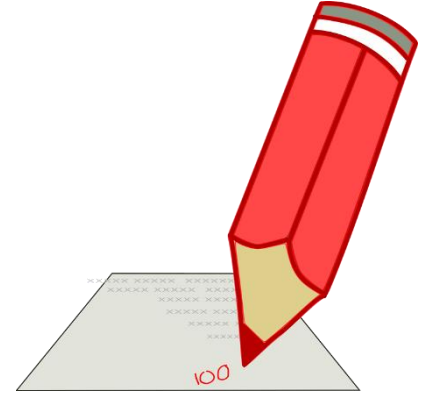
Interface sınıfı kullandığımız zaman karşılaştığımız ilk sorun, her altsınıfın interface sınıfında yer alan metotları implemente etmesi gerçeğidir. Elliye yakın altsınıfın bulunduğu bir sistem düşünün.

Fax() metodunu değiştirmek zorunda kaldığımızda, bu elli sınıfın fax() metodunu değiştirmek zorunda kalacağız.

Gereksiz yere aynı kodu çoğaltıp, program içinde kullanmış oluyoruz. oluyoruz. Bu sistemin bakımını çok zorlaştırır ve mutlaka ve mutlaka sakınılması gereken bir durumdur!!!!

Ödev

UML hakkında araştırma yapınız.



Sorularınız

