# Nesneye Dayalı Programlama ve UML

Nesneye Dayalı Programlama nedir?

UML Nedir?

Sınıf Diyagramları

Nesneye Dayalı Programlamanın Temel Taşları

- Miras alma (Inheritance)
- Çok biçimlilik (Polymorphism)
- Veri saklama (Encapsulation)

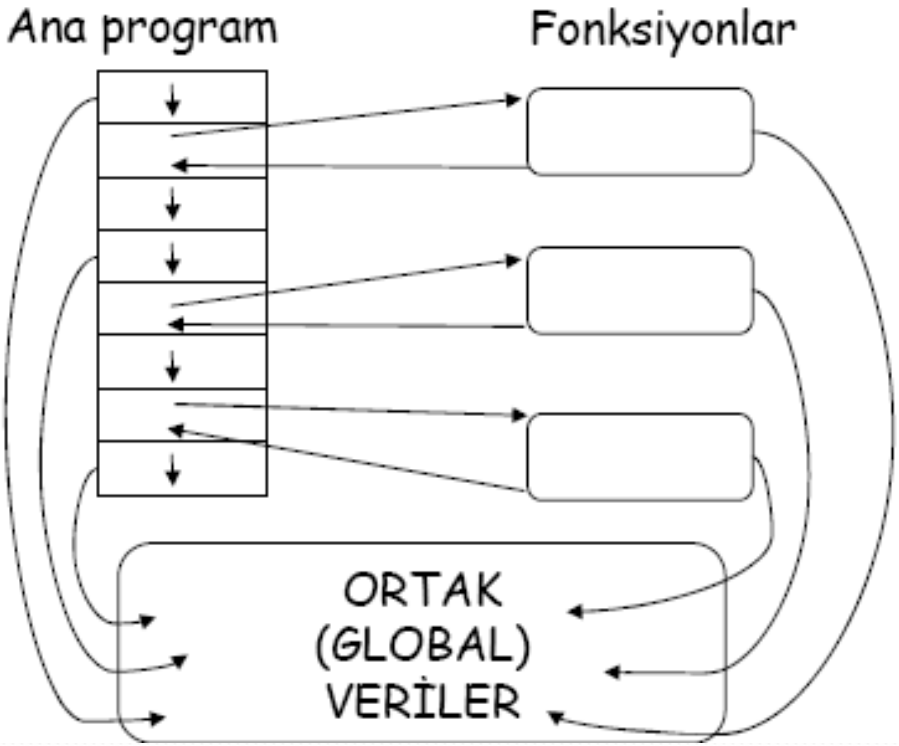
# Nesneye Dayalı Programlama ve UML

Yazılım sektöründe program geliştirme konusunda günümüze kadar bir çok yaklaşım denenmiştir. Bunların ilki programın baştan aşağıya sırası ile yazılıp çalıştırılmasıdır. Bu yaklaşımla BASIC dili kullanılarak bir çok program yazıldığını biliyoruz. Burada sorun programın akışı sırasında değişik kısımlara **goto** deyimi ile atlanmasıdır. Program kodu bir kaç bin satır olunca, kodu okumak ve yönetmek gerçekten çok büyük sorun oluyordu. GWBAsic, QBasic, Quick Basic gibi derleyiciler buna iyi örnekti.

İkinci yaklaşım ise işleve dayalı (procedürel) yaklaşımdır. Programlarda bir çok işin tekrar tekrar farklı değerleri kullanılarak yapıldığı farkedildi. Mesela herhangi bir programda iki tarih arasında ne kadar gün olduğunu bulmak birçok kez gerek olabilir. Bu durumda başlangıç ve bitiş tarihlerini alıp aradaki gün sayısını veren bir fonksiyon yazılabilir ve bu fonksiyon ihtiyaç duyulduğu yerde uygun parametrelerle çağrılıp istenen sonuç elde edilebilir. Yapısal yaklaşım Pascal ve C dillerinde uzun yıllar başari ile kullanılmıştır.

## İşleve Dayalı (*Procedural*) Programlama Yöntemi

Basic, Fortran, Pascal, C gibi programlama dillerinin desteklediği bu yöntemde öncelikle gerçeklenmek istenen sistemin yapması gereken iş belirlenir.
Büyük boyutlu ve karmaşık işler, daha küçük ve basit işlevlere (fonksiyon) bölünerek gerçeklenirler.
Gerçek dünyanın modellenmesi (soyutlama) fonksiyonlar ile yapılır.



## İşleve Dayalı Programlama Yönteminin Değerlendirmesi

- Karmaşık bir problemi daha basit fonksiyonlara bölmek programlama işini kolaylaştırmaktadır.

Ancak, yazılımların karmaşıklılığı sadece boyutlarına bağlı değildir. Birimler arası ilişkiler ve bilgi akışı karmaşıklığa neden olabilir.

- Gerçek dünyadaki sistemler sadece fonksiyonlardan oluşmaz. Sistemin gerçeğe yakın bir modelini bilgisayarda oluşturmak zordur.
- Tasarım aşamasında verilerin göz ardı edilip fonksiyonlara ağırlık verilmesi hatalar nedeniyle verilerin bozulma olasılığını arttırır. Veri gizleme (*data hiding*) olanağı kısıtlıdır.
- Programcılar kendi veri tiplerini yaratamazlar
- Programı güncellemek gerektiğinde, yeni öğeler eklemek ve eski fonksiyonları yeni eklenen unsurlar için de kullanmak zordur.

İşleve dayalı yöntemi de kullanarak kaliteli programlar yazmak mümkündür. Ancak nesneye dayalı yöntem kaliteli programların oluşturulması için programcılara daha çok olanak sağlamaktadır ve yukarıda açıklanan sakıncaları önleyecek mekanizmalara sahiptir.

## Nesneye Dayalı (*Object-Oriented*) Programlama Yöntemi

- Gerçek dünya nesnelerden oluşmaktadır.
- Çözülmek istenen problemi oluşturan nesneler, gerçek dünyadaki yapılarına benzer bir şekilde bilgisayarda modellenmelidir.
- Nesnelerin yapıları iki bölümden oluşmaktadır: **1**. *Nitelikler* (özellikler ya da durum bilgileri), **2**. *Davranışlar* (yetenekler, sorumluluklar)
- Nesneler belli bir **sorumluluğu** yerine getirmek üzere tasarlanırlar

Tasarım yapılırken sistemin işlevi değil, sistemi oluşturan **varlıklar** esas alınır. Bu nedenle tasarım yapılırken sorulması gereken soru,
"*Bu sistem ne iş yapar?*" değil,
"***Bu sistem hangi nesnelerden oluşur?***" olmalıdır.

Hangi unsurların nesne olarak modellenebilir:

- İnsan kaynakları ile ilgili bir programda; memur, işçi, müdür, genel müdür.
- Grafik programında; nokta, çizgi, çember, silindir.
- Matematiksel işlemler yapan programda; karmaşık sayılar, matris.
- Kullanıcı arayüzü programında; pencere, menü, çerçeve.

# UML

Herşeyden önce UML bir programlama dili değildir. UML(Unified Modeling Language) daha çok Nesneye Dayalı Programlama için kullanabileceğimiz bir modelleme dilidir. Grady Blooch, James Rumbaugh ve Ivar Jacobson tarafından geliştirilmiştir. UML'den önce yukarıda adı geçen kişilerin her birinin kendi geliştirdiği metodolojiler vardı. Grady Booch'un "Booch Metodolojisi", James Rumbaugh'un "OMT(Object Modeling Technique)" ve Ivar Jacobson'un " OOSE (Object Oriented Software Engineering) " metodolojileri vardı. Rational firması tarafından bu görsel modelleme dilleri birleştirilerek 1995'te UML geliştirilmiş ve 1997 yılında Object Management Group (OMG) tarafından bir standart olarak kabul edilmiştir.

www.omg.org

UML 1.0 (1997)

UML 2.0 (2004)

# Nesneye Dayalı Programlama ve UML

Yazılım teknolojisi geliştikçe yazılan programların karmaşıklığı ve zorluğu giderek artmaktadır. Donanım ve yazlımın iç içe girdiği, büyük ağ sistemlerinin giderek arttığı bir dönemde doğaldır ki biz programcıların yazacağı programlarda büyüyecektir.Yazacağımız programlar çok karmaşık olacağı için kod organizasyonu yapmamız zor olacaktır. Hele birçok programcının çalışacağı projelerde bu nerdeyse imkansız hale gelmiştir. Bu yüzden standart bir modelleme ve analiz diline ihtiyaç duyarız. Programımızın analiz ve dizayn aşamasında modellemeyi güzel yaparsak ileride doğabilecek birçok problemin çıkmasına engel olmuş oluruz. UML daha çok nesneye dayalı programlama dilleri için uygundur. Problemlerimizi parçalara ayırabiliyorsak, ve parçalar arasında belirli ilişkiler sağlayabiliyorsak UML bizim için biçilmiş kaftan gibidir. Mesela bir ATM siteminde müşteriyi, banka memurunu ve ATM makinasını ayrı parçalar halinde düşünebiliriz. Müşteri ATM makinasından para çeker, banka memuru ATM makinasına para yükler.Ama banka memuru ile müşteri arasında doğrudan bir ilişki yoktur.Bu tür ilişkiler UML 'de çeşitli diyagramlarla gösterilir. Bu diyagramlar hakkında geniş ve detaylı bilgiyi daha sonraki yazılarımızda ele alacağız.

Bu kısa girişten sonra UML 'in tarihine bakalım.UML 'in doğuşu son yıllarda yazılım endüstrisindeki en büyük gelişmelerden biri olarak kabul edilebilir. UML 1997 yılında yazılımın, diyagram şeklinde ifade edilmesi için bir standartlar komitesi tarafından oluşturuldu. Daha önce hemen hemen her daldaki mühendislerin standart bir diyagram çizme aracı vardı. Ve şimdi de programcıların UML 'si var.

# Nesneye Dayalı Programlama ve UML

UML ile hazırlanmış bir yazılım hem daha az maliyetli hem daha etkili ve daha uzun ömürlü olur. UML ile dokümantasyonu yapılmış bir programın sonradan düzenlenmesi daha kolay olur. Bütün bunlar UML kullanmamız için yeterli sebeplerdir diye düşünüyorum.Kısaca ,UML 'nin faydalarını maddeler halinde sıralarsak;

**1-)** Öncelikle programımız kodlanmaya başlamadan önce geniş bir analizi ve tasarımı yapılmış olacğından kodlama işlemi daha kolay olur. Çünkü programdan ne beklediğimizi ve programlama ile neler yapacağımızı profesyonel bir şekilde belirleriz UML ile.

**2-)** Programımızda beklenmedik bir takım mantıksal hataları (bug) minimuma indirgemiş oluruz.

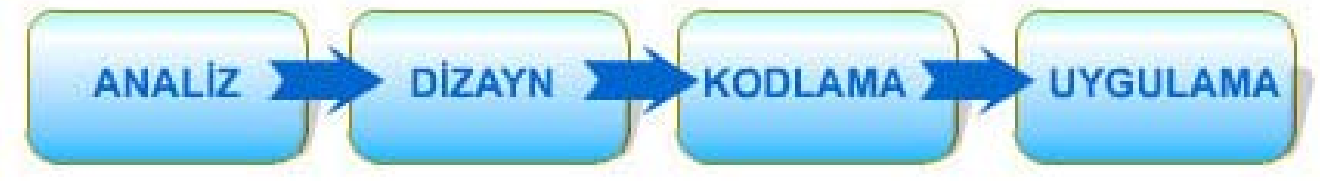
**3-)** Tasarım aşaması düzgün yapıldıysa tekrar kullanılabilen kodların sayısı artacaktır. Buda program geliştirme maliyetini büyük ölçüde düşürecektir.

**4-)** UML diagramları programımızın tamamını kapsayacağı için bellek kullanımını daha etkili hale getirebiliriz.

**5-)** Programımızın kararlılığı artacaktır. UML ile dokümanlandırılmış kodları düzenlemek daha az zaman alacaktır.

**6-)** Ortak çalışılan projelerde programcıların iletişimi daha kolay hale gelir.Çünkü UML ile programımızı parçalara ayırdık ve parçalar arasında bir ilişki kurduk.

Bir sistemin geliştirilmesi kabaca aşağıdaki aşamalardan geçmektedir. İlk iki aşamada UML büyük ölçüde rol oynar.



# Nesneye Dayalı Programlama ve UML

Şimdi kısa ve öz bir şekilde UML komponentlerinden(diagramlar) bahsedelim: Nesneler arasında ilişki kurmak için UML bir takım grafiksel elemanlara sahiptir.Bu elemanları kullanarak diyagramlar oluşturacağız. Bu makalede sadece bu diagramların ne işe yaradığını göreceğiz. UML temel olarak 9 diyagram türünden oluşur.

**CLASS DIAGRAM**

Gerçek dünyada eşyaları nasıl araba, masa, bilgisayar şeklinde sınıflandırıyorsak yazılımda da birtakım benzer özelliklere ve fiillere sahip gruplar oluştururuz. Bunlara "Class"(sınıf) denir. Geliştirici açısından önemli olan "Class Diagramları" hakkında daha sonra detaylı bir makalemiz olacak.

**OBJECT DIAGRAM**

Bir nesne(object) sınıfın (class) bir örneğidir. Bu tür diyagramlarda sınıfın yerine gerçek nesneler kullanılır.

**STATE DIAGRAM**

Gerçek nesnelerin herhangi bir zaman içindeki durumunu gösteren diyagramlardır.Mesela, Ali nesnesi insan sınıfının gerçek bir örneği olsun. Ali 'nin doğması, büyümesi, gençliği ve ölmesi State Diagram 'larıyla gösterilir.

**SEQUENCE DIAGRAM**

Class ve Object diyagramları statik bilgiyi modeller.Halbuki gerçek zamanlı sistemlerde zaman içinde değişen interaktiviteler bu diyagramlarla gösterilemez. Bu tür zamanla değişen durumları belirtmek için sequence diyagramları kullanılır.

# Nesneye Dayalı Programlama ve UML

**ACTIVITY DIAGRAM**

Bir nesnesinin durumu zamanla kullanıcı tarafından ya da nesnenin kendi içsel işlevleri tarafından değişebilir.Bu değişim sırasını activity diyagramlarıyla gösteririz.

**USE CASE DIAGRAM**

Programımızın davranışının bir kullanıcı gözüyle incelenmesi Use Case diyagramlarıyla yapılır. Gerçek dünyada insanların kullanacağı bir sistemde bu diyagramlar büyük önem taşırlar.

**COLLABORATION DIAGRAM**

Bir sistemin amacının yerine gelmesi için sistemin bütün parçaları işlerini yerine getirmesi gerekir. Bu işler genellikle birkaç parçanın beraber çalışmasıyla mümkün olabilir. Bu tür ilişkileri göstermek için Collaboration Diyagramları gösterilir.

**COMPONENT DIAGRAM**

Özellikle birden çok geliştiricinin yürüttüğü projelerde sistemi component dediğimiz parçalara ayırmak, geliştirmeyi kolaylaştırır.Sistemi öyle modellememiz gerekir ki her geliştirici ötekinden bağımsız olarak çalışabilsin.Bu tür modellemeler Component Diyagramlarıyla yapılır.

**DEPLOYMENT DIAGRAM**

Bu tür diyagramlarla sistemin fiziksel incelenmesi yapılır. Mesela bilgisayarlar arasındaki baglantılar, programın kurulacağı makinalar ve sistemimizdeki bütün aletler Deployment Diyagramında gösterilir.

## Nesne (object)

- ✓Gerçek dünyada, ayrı ayrı tanımlanabilen herşey bir nesnedir.
- ✓Modelde, her nesnenin bir kimliği, durumu, ve davranışı vardır.

## Sınıf (class)

- ✓Gerçek dünyada, benzer karakteristik ve davranışlara sahip nesneler bir sınıf (class) ile temsil edilir.
- ✓Modelde, bir sınıf, nesneler tarafından paylaşılan durum ve davranışları temsil eder.

## NESNE

Kimlik (*identity*)

Nesneyi birtek (*unique*) olarak tanımlar ve onu diğer nesnelerden ayırır

Durum (*state*)

Özellikler (*fields* veya *attributes*) ile belirtilir

Davranış (*behavior*)

Metotlar (*methods* veya *operations*): nesnenin durum bilgilerine erişebilen ve değiştirebilen işlemler.
Metot, metot adı, aldığı parametre türleri, ve döndürdüğü tür ile tanımlanır. Herhangi bir değer döndürmeyen metotlar *void* ile belirtilir.

## NESNE Örneği

Kimlik:

Öğrenci123

Durum:

ad: “Ali Yılmaz”

öğrenciNo: “0401.....”

yıl: 2007

Metotlar:

dersEkle()

dersSil()

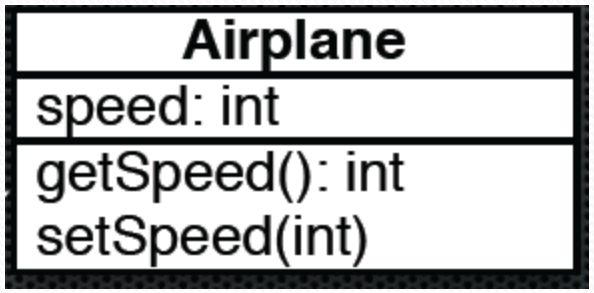
danışmanAta()

**Sınıf (*class*) aşağıdakileri tanımlar:**

Alanlar (*fields*): Nesne özelliklerini tanımlayan değişkenler, adları ve türleri ile.

Metotlar (*methods*): Metot adları,döndürdüğü tür, parametreleri, ve metotu gerçekleştiren program kodu

UML Diyagramı




```
1  public class Airplane {
2
3      private int speed;
4
5      public Airplane(int speed) {
6          this.speed = speed;
7      }
8
9      public int getSpeed() {
10         return speed;
11     }
12
13     public void setSpeed(int speed) {
14         this.speed = speed;
15     }
16
17 }
```

# UML Sınıf Tanımlamaları

**Alanlar :**

Kod → private long maas

UML → private maas : long

**Metotlar :**

Kod → public double maasHesapla()

UML → public maasHesapla(): double

## ERİŞİM

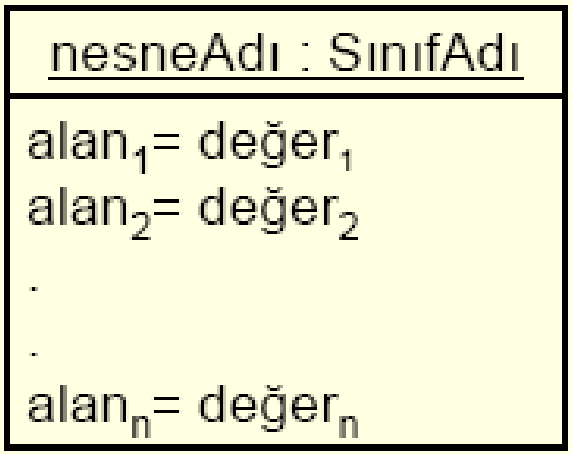
**Public:** diğer sınıflar erişebilir. UML'de + sembolü ile gösterilir.

**Protected:** aynı paketteki (*package*) diğer sınıflar ve bütün alt sınıflar (*subclasses*) tarafında erişilebilir. UML'de # sembolü ile gösterilir.

**Package:** aynı paketteki (*package*) diğer sınıflar tarafında erişilebilir. UML'de ~ sembolü ile gösterilir.

**Private:** yalnızca içinde bulunduğu sınıf tarafından erişilebilir (diğer sınıflar erişemezler). UML'de - sembolü ile gösterilir.

# UML'de Nesne Gösterimi

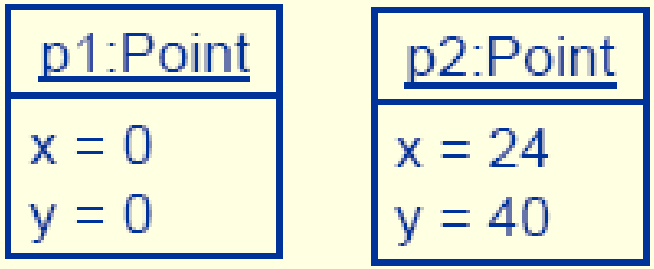



← Nesne ve Sınıf Adı; altı çizili

← Alanlar ve aldıkları değerler

# UML'de Nesne Gösterimi

UML




Java

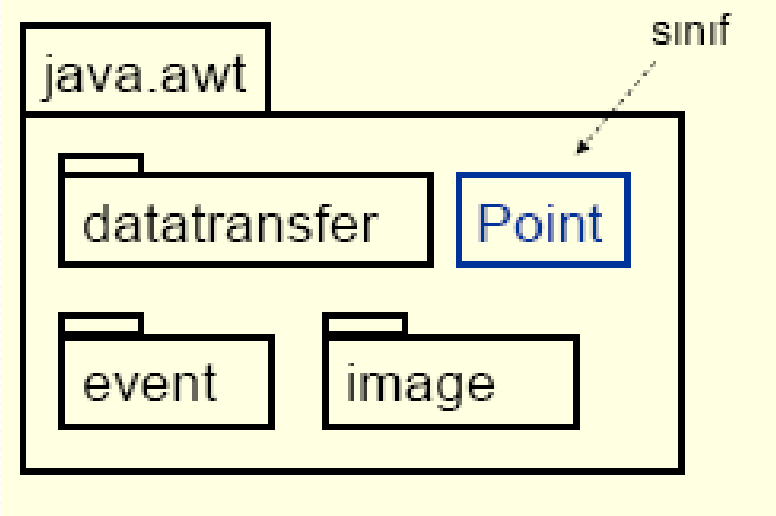
```
Point p1 = new Point();
p1.x = 0;
p1.y = 0;

Point p1 = new Point();
p1.x = 24;
p1.y = 40;
```

# UML'de Paket Gösterimi

Birbirleriyle ilişkili sınıflar bir paket (package) içine yerleştirilirler.

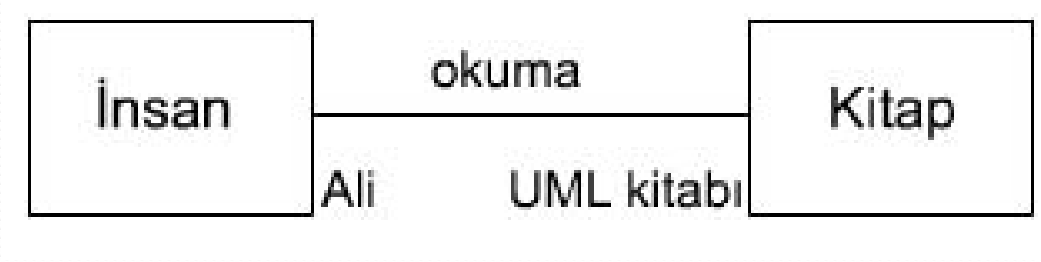
Paket isimler küçük harflerle yazılır



# ASSOCATION (Sınıflar arası ilişki)

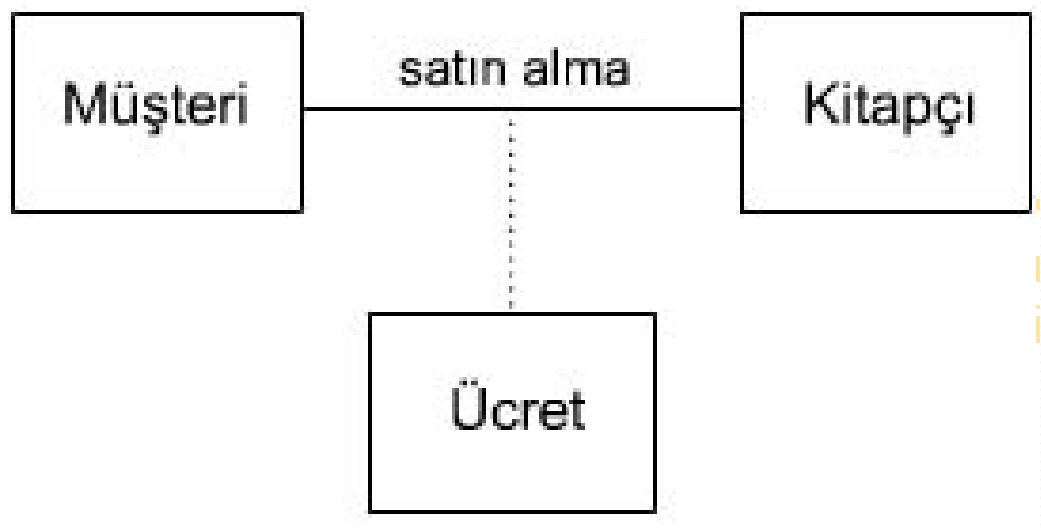
Sınıflar arasındaki ilişkiyi göstermek için iki sınıf arasına düz bir çizgi çekilir.İlişkiyi gösteren çizginin üzerine ilişkinin türü yazılır.Mesela Kitap ve İnsan sınıfları olsun.Kitap ile insan sınıfı arasında "okuma" ilişkisi vardır.Bunu sınıf diyagramında aşağıdaki gibi gösteririz.

Bir İnsan sınıfı gerçek nesnesi olan "Ali" ile kitap sınıfı gerçek nesnesi olan "UML kitabı" arasında "okuma" ilşkisi vardı.Kısaca şöyle deriz. Ali, UML kitabı okur.Tabi gerçek bir sistemde ilşkiler bu kadar basit olmayabilir. Bazı durumlarda ikiden fazla sınıf arasında ilişki olabilir, o zaman da her sınıf arasındaki ilşkiyi tanımlamamız gerekir.Bazı durumlarda ise belirtilen ilişkinin bir kurala uyması gerekebilir.Bu durumda ilişki çizgisinin yanına "constraints"(ilişki kuralı) yazılır.



# Nesneye Dayalı Programlama ve UML

Bazı durumlarda sınıflar arasındaki ilişki, bir çizgiyle belirtebileceğimiz şekilde basit olmayabilir.Bu durumda ilişki sınıfları kullanılır.İlişki sınıfları bildigimiz sınıflarla aynıdır.Özellik ve işlev elemanları olabilir.Sınıflar arasındaki ilişki eğer bir sınıf türüyle belirleniyorsa UML ile gösterimi aşağıdaki şekildeki gibi yapılır.




Görüldüğü gibi Müşteri ile Kitapçı sınıfı arasında "satın alma" ilişkisi vardır.Fakat müşteri satın alırken Ücret ödemek zorundadır.Bu ilişkiyi göstermek için Ücret sınıfı ilişki ile kesikli çizgi ile birleştirilir.

# Nesneye Dayalı Programlama ve UML

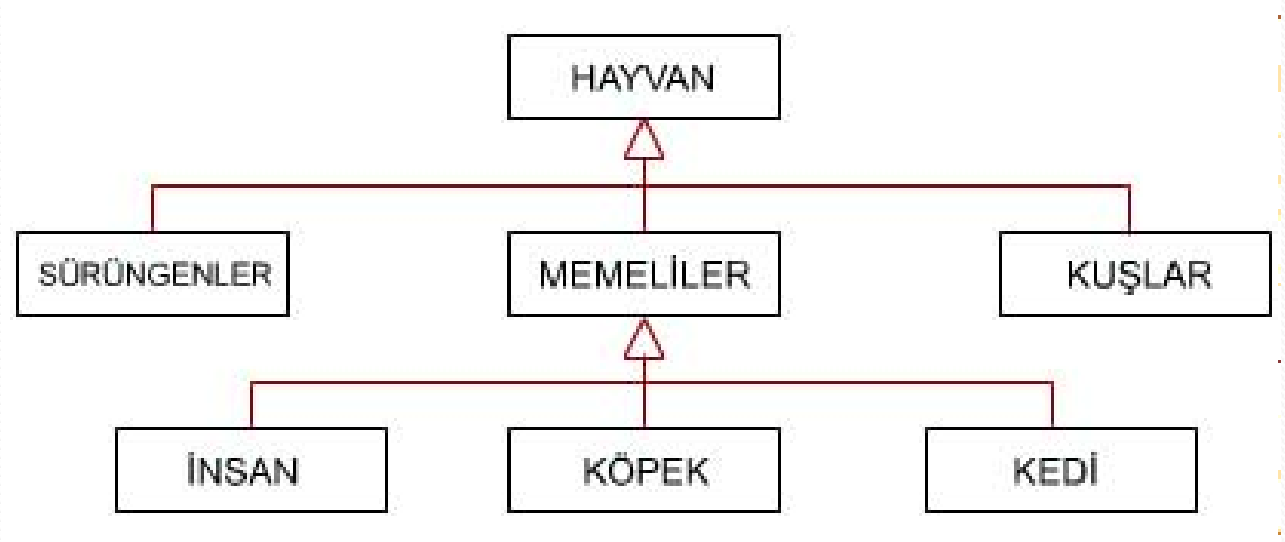
Şu ana kadar gördüğümüz ilişkiler bire-bir ilişkilerdi.İlişkiler bire-bir olmak zorunda değildir.Bir sınıf, n tane başka bir sınıf ile ilişkiliyse buna bire-çok ilişiki denir.Mesela Yüzbaşı ile Er arasında bire-yüz bir ilişki vardır.Diyagramda bunu gösterirken Yüzbaşı sınıfına 1 Er sınıfına ise 100 yazacağız.Gösterimi aşağıdaki gibidir.


En temel ilişkiler aşağıdaki gibi listelenebilir:

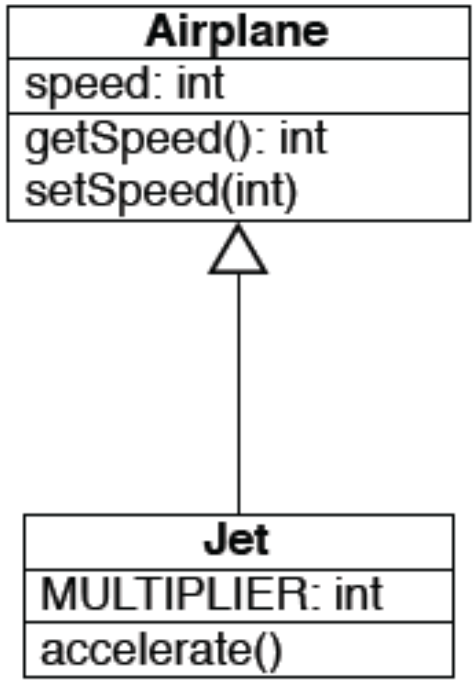
-> Bire-bir
-> Bire-çok
-> Bire-bir veya daha fazla
-> Bire-sıfır veya bir
-> Bire-sınırlı aralık (mesela:bire-[0,20] aralığı)
-> Bire-n (UML de birden çok ifadesini kullanmak için '*' simgesi kullanılır.)
-> Bire-Beş yada Bire-sekiz

# KALITIM (INHERITANCE)

Eğer eşyalar arasında genellemeler yapabiliyorsak genellemeyi yaptığımız eşyalarda ortak özelliklerin olduğunu biliriz. Mesela, "Hayvan" diye bir sınıfımız olsun.Memeliler, Sürüngenler, Kuşlar da diğer sınıflarımız olsun.Memeliler, Sürüngenler ve Kuşlar sınıfının farklı özellikleri olduğu gibi hepsinin Hayvan olmasından dolayı birtakım ortak özellikleri vardır. Bu yüzden Memeliler, Sürüngenler ve Kuşlar birer hayvandır deriz. Yani kısacası Memeliler, Sürüngenler ve Kuşlar, Hayvan sınıfından türemiş ve herbirinin kendine özgü özellikleri vardır deriz. Nesne yönelimli programlamada buna kalıtım (Inheritance) denir. UML 'de kalıtım aşağıdaki şekilde olduğu gibi gösterilir.



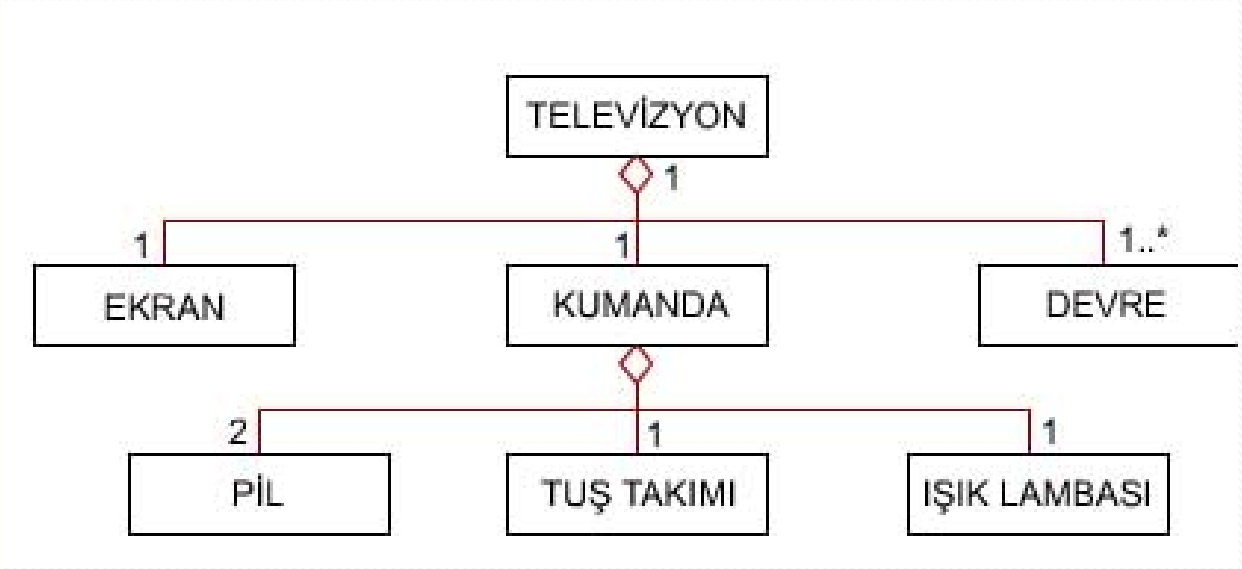
# Kalıtım'a Örnek




```
1  public class Jet extends Airplane {
2
3      private static final int MULTIPLIER = 2;
4
5      public Jet(int id, int speed) {
6          super(id, speed);
7      }
8
9      public void setSpeed(int speed) {
10         super.setSpeed(speed * MULTIPLIER);
11     }
12
13     public void accelerate() {
14         super.setSpeed(getSpeed() * 2);
15     }
16
17 }
18
```

# İÇERME (AGGREGATIONS)

Bazı sınıflar birden fazla parçadan oluşur.Bu tür özel ilişkiye "Aggregation" denir.Mesela ,bir TV 'yi ele alalım.Bir televizyon çeşitli parçalardan oluşmuştur.Ekran,Uzaktan Kumanda,Devreler vs.. Bütün bu parçaları birer sınıf ile temsil edersek TV bir bütün olarak oluşturulduğunda parçalarını istediğimiz gibi ekleyebiliriz. Aggregation ilişkisini 'bütün parça' yukarıda olacak şekilde ve 'bütün parça'nın ucuna içi boş elmas yerleştirecek şekilde gösteririz.Örnek bir şekil aşağıdaki gibidir.



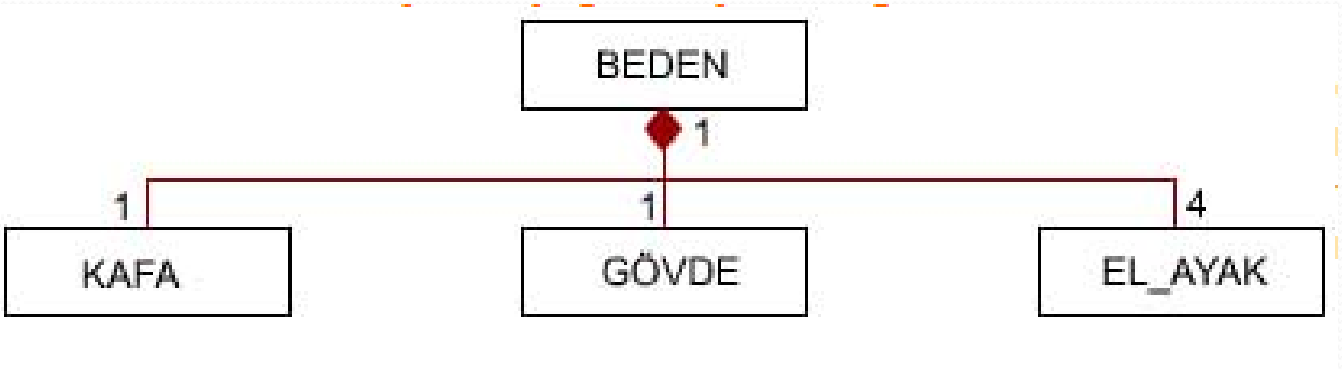
# Nesneye Dayalı Programlama ve UML

Üstteki şekilden de görüldüğü üzere bir TV sistemi 1 EKRAN,1 KUMANDA, birden çok DEVRE 'den oluşmaktadır. TV' nin bir parçası olan KUMANDA ise 2 PİL, 1 TUŞ TAKIMI ve 1 IŞIK LAMBASI 'ndan oluşmaktadır. İçi boş elma ile gösterilen ilşkilerde herbir parça ayrı bir sınıftır ve tek başlarına anlam ifade ederler. Parça bütün arasında çok sıkı bir ilişki yoktur. TV nesnesi yaratıldığında bir ekran veya bir kumanda nesnesi daha sonradan oluşturularak TV ye takılır. Ama bazı durumlarda bütün nesneyi yarattığımızda parçalarının da yaratılmasını isteriz. Mesela bir insan bedenini analizini yapalım.Bir insan vücudu baş, gövde, el ve ayaklardan oluşur. Bir insan vücudunu düşündüğümüzde tümüyle düşünürüz.

Sadece "Beden" nesnesini oluşturup sonradan bedene el, ayak, baş takmak çok mantıksız olurdu. Bu tür ilişkilerin gösterilmesine ise "COMPOSITE ASSOCATION" denir. Bu ilişki diğerine göre daha sıkıdır. Bu tür ilişkilerde bütün nesne yaratıldığında parçalar da anında yaratılır. Bazı durumlarda, takılacak parçalar duruma göre değişebilir.Belirli koşullarda Kumanda, bazı durumlarda da Ekran olmayacaksa bu tür durumlar koşul ifadeleri ile birlikte noktalı çizgilerle belirtilir.Bu konuyu daha sonraki makalelerimizde detaylı bir şekilde ele alacağız. Bu durumda takılacak parçalar "constraint(koşul)" ile belirtilir.

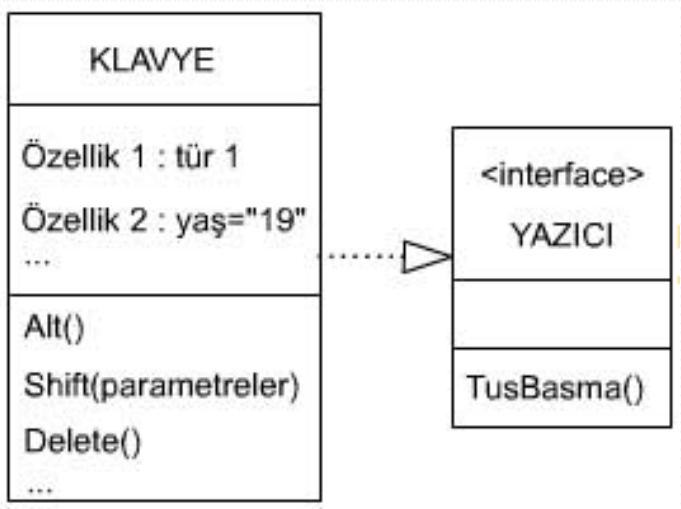
# Nesneye Dayalı Programlama ve UML

Gerçek bir BEDEN nesnesi oluştuğunda mutlaka ve mutlaka 1 KAFA, 1 GÖVDE ve 4 EL_AYAK nesnesi yaratılacaktır.Gördüğünüz gibi sıkı bir parça-bütün ilişkisi mevcuttur.



# ARAYÜZ(INTERFACE)

Bazı durumlarda bir sınıf sadece belirli işlemleri yapmak için kullanılır. Herhangi bir sınıfla ilişkisi olmayan ve standart bazı işlemleri yerine getiren sınıfa benzer yapılara arayüz(interface) denir.Arayüzlerin özellikleri yoktur. Yalnızca bir takım işleri yerine getirmek için başka sınıflar tarafından kullanılırlar. Mesela, bir "TuşaBasma" arayüzü yaparak ister onu "KUMANDA" sınıfında istersek de aşağıdaki şekilde görüldüğü gibi "KLAVYE" sınıfında kullanabiliriz. Sınıf ile arayüz arasındaki ilişkiyi kesik çizgilerle ve çizginin ucunda boş üçgen olacak şekilde gösteririz. Sınıf ile arayüz arasındaki bu ilişkiye gerçekleme(realization) denir. Sınıfla, arayüz arasında UML gösterimi açısından fazla bir fark yoktur.Tek fark arayüzde özellik(attribute) yoktur.Diğer bir fark ise arayüz adlarını yazarken adın üstüne <interface> yazısını eklemektir.Aşağıda bir arayüz-sınıf ilişkisi mevcuttur.



## Polymorphism (Çok biçimlilik)

**Çok biçimlilik bir nesnenin davranış şekillerinin duruma göre değiştirilebilmesidir.**" Aynen, bulunduğu ortamın şartlarına göre renklerini mükemmel bir biçimde ayarlayan bukalemun gibi.

## Encapsulation (Kapsülleme)

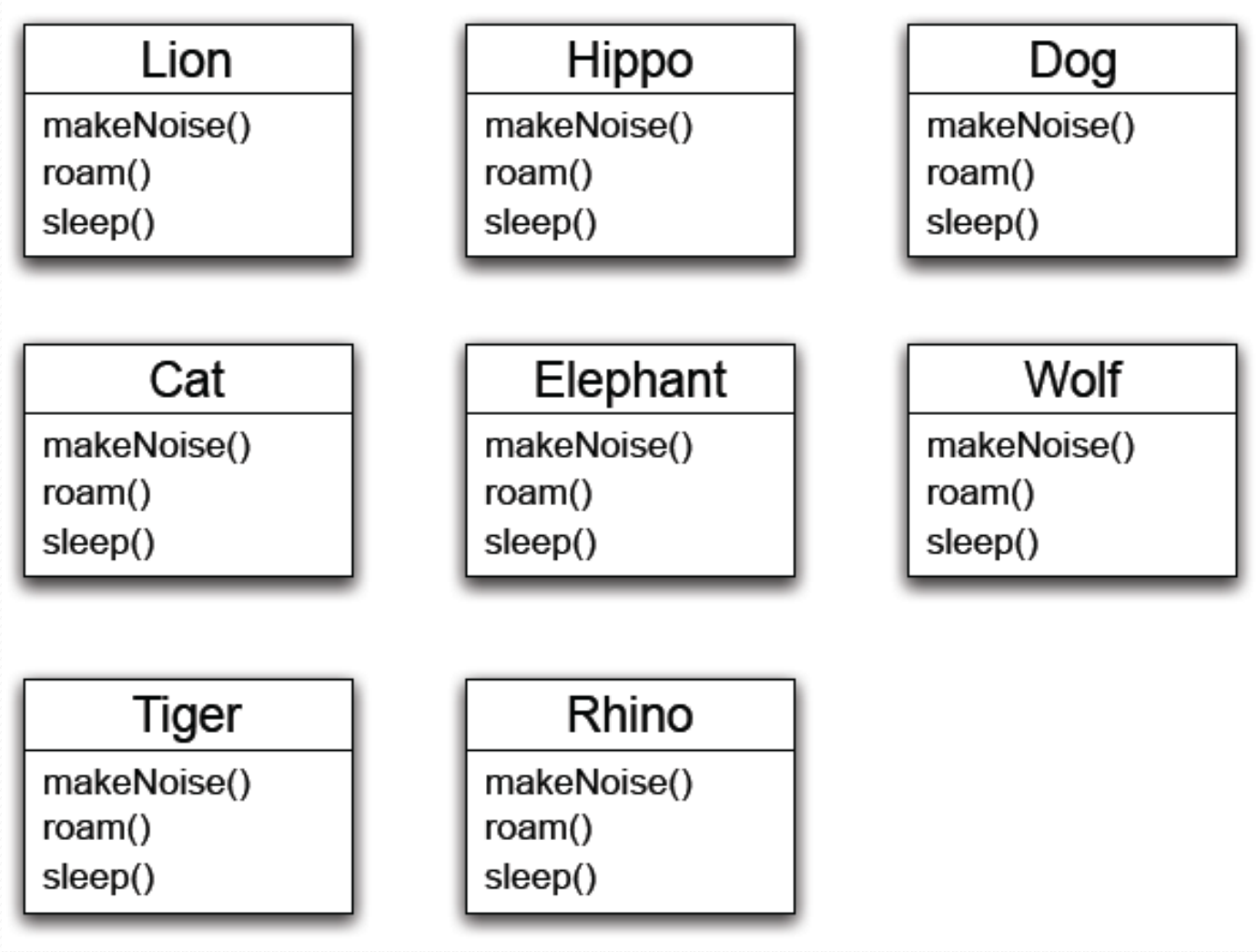
Çalışma detaylarını sunumdan gizleme. CepTelefonu sınıfı için tanımlanan Marka, Model gibi public özellikler CepTelefonu sınıfının görünen yüzüdür. Dışardan bakan sadece bu public alanları ve metodları görebilir. Ama sınıfın içine girdiğimizde seriNo, frekans gibi başka private (özel) özellikler görüyoruz. Bunlar tamamen implementation (CepTelefonu'nun çalışması) ile ilgili şeyler. Dışarıdan bilinmesi gerekmiyor. Hele hele dışarıdan değiştirilmemeliler yoksa telefon bozulabilir.

-Sınıflarımızda dışarıdan bilinmesi gerekmeyen özellikleri ve metodları gizlemek için private deyimini kullanıyoruz.

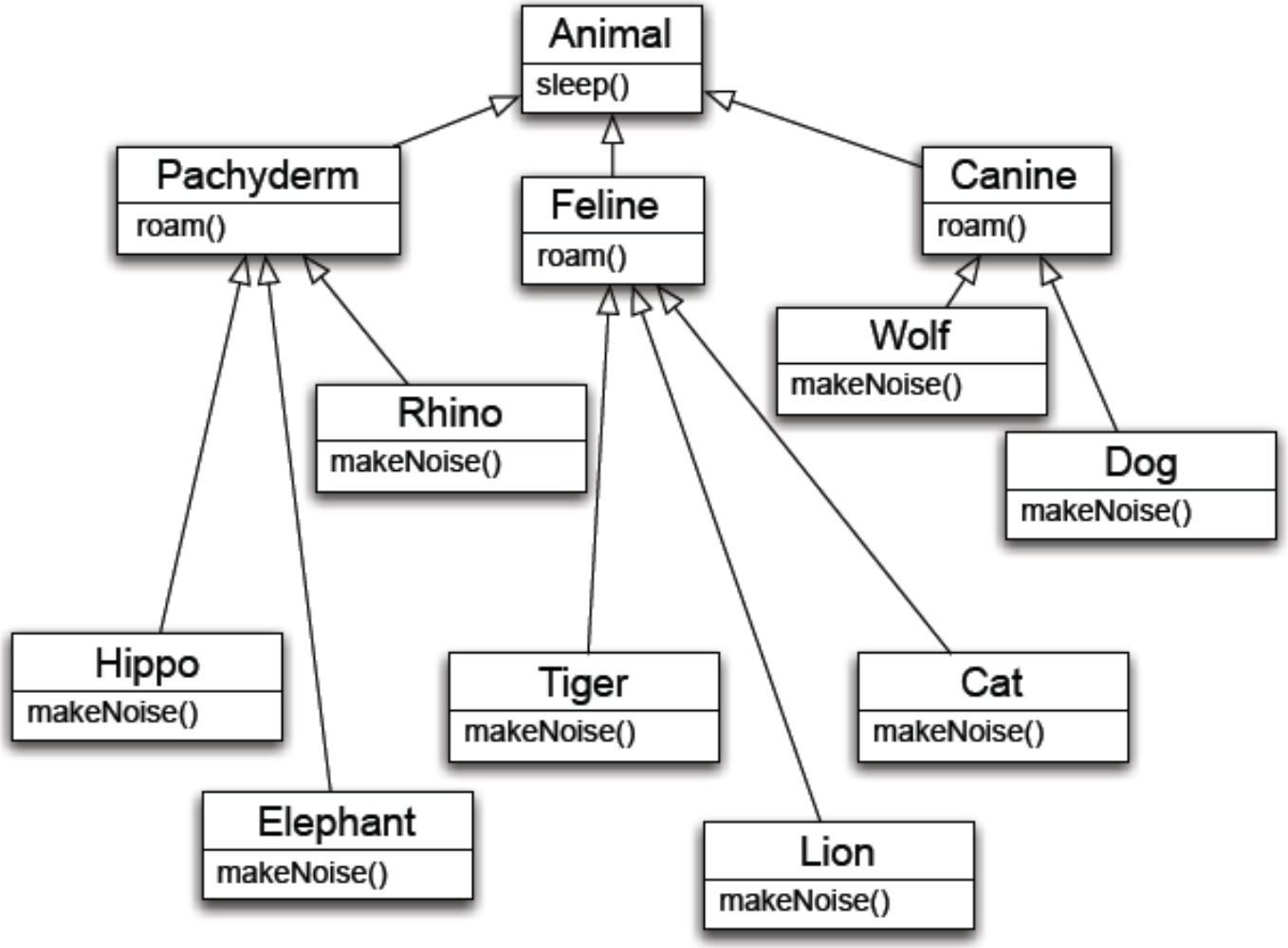
- Private özellikler sadece sınıfın içinden görülebilen, değiştirilebilen özelliklerdir.

- Bilinmesinde ve/veya/yahut/ya da değiştirilmesinde sakınca olmayan özellik ve metodları ise public yapıyoruz.

**Örnek :** Hayvanlar (Kalıtım kullanmadan)



# Hayvanlar (Kalıtım kullanarak)



# Karşılaştırma

Kalıtım olmadan : 9 dosya, 200 satır kod

Kalıtımla birlikte: 13 dosya, 167 satır kod

yaklaşık % 15 tasarruf (basit bir örnek için)

## Use Cases (Kullanım Şekilleri) Diyagramları

Sistem gereksinimleri UML Use Case (kullanım şekilleri) diyagramları ile belirtilir

Yazılım geliştirme için gerekli değildir, fakat gereksinimler ve nesnesel modeller arasında en önemli bağlantıdır

Use Case: Kullanım şekli

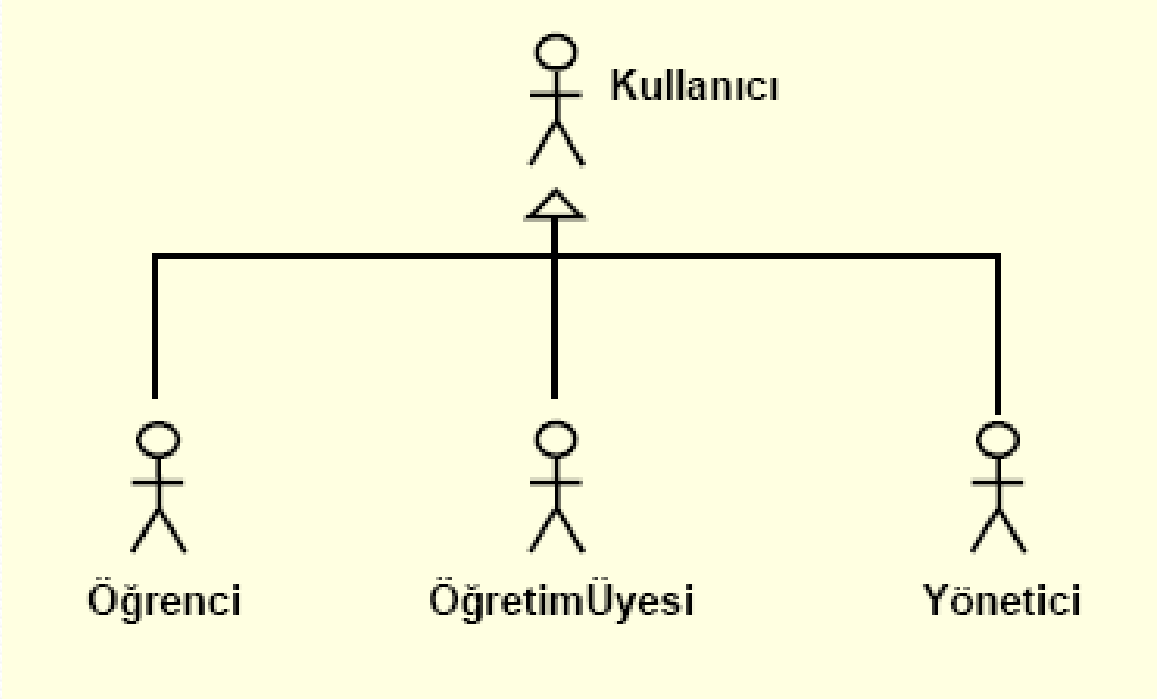
Bir sistem fonksiyonunun dışarıdan gözlemlenen davranışı
Sistemle, sistem dışı aktörler (kullanıcı veya diğer sistemler gibi) arasında etkileşimler..
Sistem “ne” yapıyorla ilgili, “nasıl” yapıyorla ilgili değil..

# Nesneye Dayalı Programlama ve UML

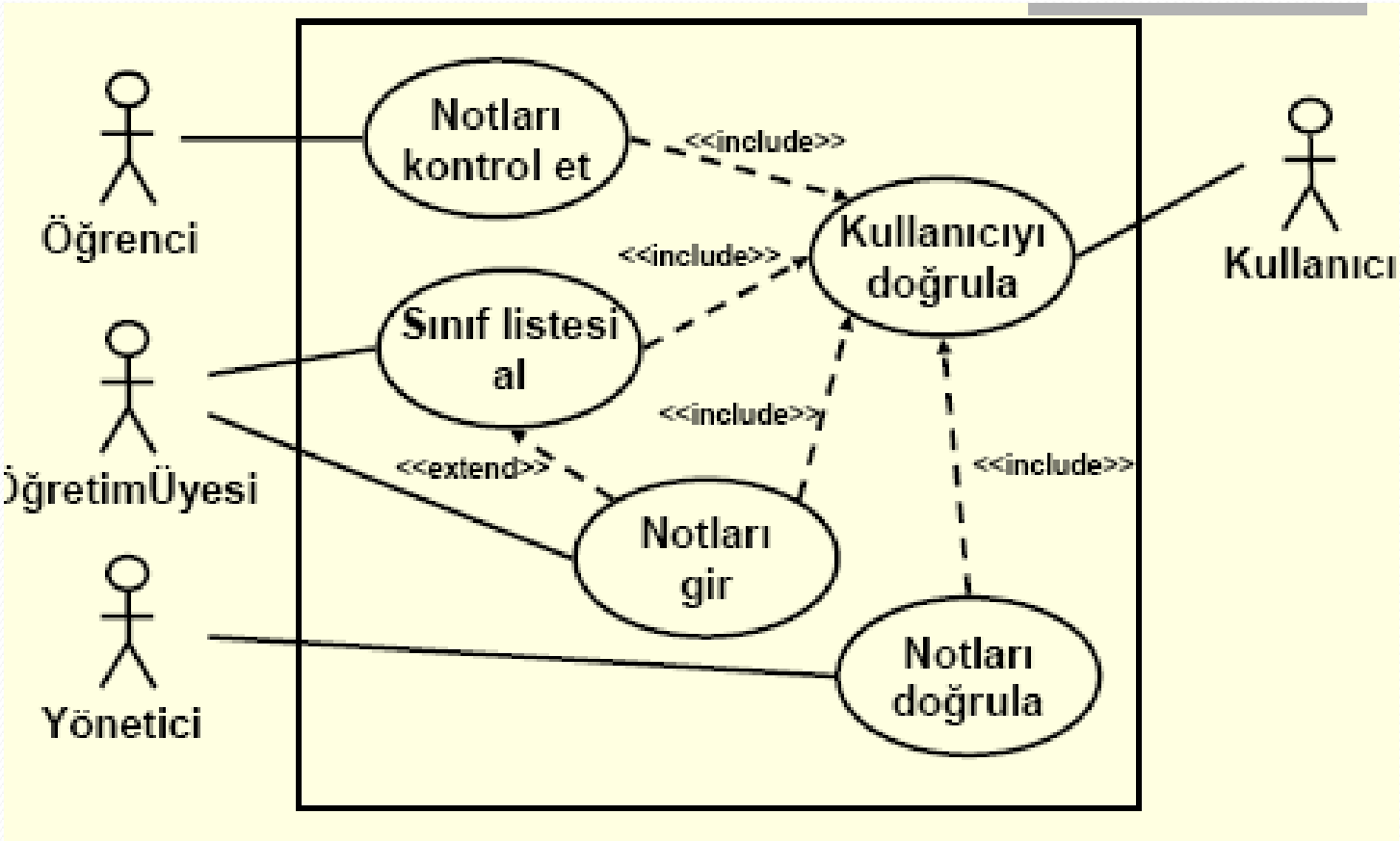

Aktör


Kullanım şekli (Use case)

# Nesneye Dayalı Programlama ve UML



# Nesneye Dayalı Programlama ve UML



# Kaynaklar

1) Head First Object-Oriented Analysis & Design
2) www.csharpnedir.com
3) Y.Doç.Dr. Feza BUZLUCA ders notları
4) Doç.Dr. Erdoğan DOĞDU ders notları