1906003172019

Tasarım Desenleri

Dr. Öğr. Üy. Önder EYECİOĞLU
Bilgisayar Mühendisliği

Hafta	İşlenecek Konu
1	Bölüm 1: Yazılım Tasarımı, Yazılım Örüntüleri ve UML
	1.1. Yazılım geliştirme süreci
	1.2. Yazılım Tasarımı ve Tasarım Örüntüleri
2	1.3. UML
	1.3.1. Sınıf Diyagramı (Class Diagram)
	1.4. Java Dökümantasyonu (Javadoc)
3	Bölüm 2: Nesnesel Kavramlar
	2.1. Modülerlik
	2.2. Soyutlama (abstraction)
	2.2.1. Genelleştirme ve Özelleştirme
4	2.3. Sınıf Tasarımı
	2.3.1. Bilgi Saklama
	2.4. Kalıt (inheritance)
5	2.5. Tür Değiştirme (Type Casting)
	2.6. Soyut Sınıf (Abstract Class)
	2.7. Soyut Metotlar
6	2.8. Polimorfizm (polimorphism)
	2.9. Arayüz (interface)
	2.10. Nesnenin Yaşam Döngüsü
	2.11. Sınıf ve Nesne Verileri
7	2.12. Java'da Diğer İlgili Konular
	2.12.1. Java Nesnelerinin Kopyalanması Veya Klonlanması
	2.12.2. Paket

Hafta	İşlenecek Konu
8	2.12.3. Java Kütüphaneleri
	2.12.4. Java Collections
	2.12.5. Java Generics
	2.12.6. Java Nesnelerinin Karşılaştırılması
9	Bölüm 3: Tasarım Örüntüleri
	3.1. Tasarım Örüntüsü Nedir?
	3.2. İncelenecek Tasarım Örüntüleri ve Tasarım Kategorileri
10	3.3. Gözlemci Örüntüsü (Observer Pattern)
	3.4. Dekoratör Örüntüsü (Decorator Pattern)
	3.5. Strateji Örüntüsü (Strategy Pattern)
	3.5.1. Soyut Fabrika Örüntüsü (Abstract Factory Pattern)
11	3.6. Tekli Örüntü (Singleton Pattern)
	3.7. Komut Örüntüsü (Command Pattern)
	3.8. Adaptör Örüntüsü (Adapter Pattern)
12	3.9. Fasat Örüntüsü (Façade Pattern)
	3.10. Kalıp Metodu Örüntüsü (Template Method Pattern)
	3.11. İterasyon Örüntüsü (Iterator Pattern)
13	3.12. Komposit Örüntüsü (Composite Pattern)
	3.13. Durum Örüntüsü (State Pattern)
	3.14. Proksi Örüntüsü (Proxy Pattern)
14	Genel Tekrar

- 1.3. UML
  - 1.3.1. Sınıf Diyagramı (Class Diagram)
- 1.4. Java Dökümantasyonu (Javadoc)

- Tekil Modelleme Dili (**UNIFIED MODELING LANGUAGE-UML**)
- Modelleme Gereksinimi
  - UML Tarihçesi
  - UML Diyagramları
    - Sınıf Diyagramları
    - Erişim
    - Sınıflar Arası İlişki
    - Kalıtım (Inheritence)
    - İçerme (Aggregations)

- Modelleme Gereksinimi
  - UML Diyagramları
    - Arayüzler
  - Sınıfların Sahip Olduğu Roller

UML, Yazılım Mühendisliği'nde nesne tabanlı sistemleri modellemede kullanılan açık standart olmuş bir görsel programlama dilidir. Analizden tasarım aşamasına dek oldukça etkili bir notasyon sağlar.

- Yazılım ve donanımların bir arada düşünülmesi gereken,
- Zor ve karmaşık programların,
- Özellikle birden fazla yazılımcı tarafından kodlanacağı durumlarda,
- Standart sağlamak amacıyla endüstriyel olarak geliştirilmiş grafiksel bir dildir.
- ~~Programlama dili~~-Diyagram çizme ve ilişkisel modelleme dili

- UML yazılım sisteminin önemli bileşenlerini tanımlamayı, tasarlamayı ve dokümantasyonunu sağlar
- Yazılım geliştirme sürecindeki tüm katılımcıların (kullanıcı, iş çözümleyici, sistem çözümleyici, tasarımcı, programcı,...) gözüyle modellenmesine olanak sağlar,
- UML
- Yazılımın geniş bir analizi ve tasarımı yapılmış olacağından kodlama işlemi daha kolay ve doğru olur
- Hataların en aza inmesine yardımcı olur
- Geliştirme ekibi arasındaki iletişimi kolaylaştırır
- Tekrar kullanılabilir kod sayısını artırır
- Tüm tasarım kararları kod yazmadan verilir
- Yazılım geliştirme sürecinin tamamını kapsar
- “resmin tamamını” görmeyi sağlar
- gösterimi nesneye dayalı yazılım mühendisliğine dayanır.

Karmaşık sistemleri daha iyi anlayabilmek için modeller yaparız. Oluşturulan model sayesinde karmaşık bir gerçeği daha basit bir dille ifade etme şansımız olur. Modeller, karmaşık sistem ya da yapıların gereksiz ayrıntılarını filtreleyerek, problemlerin anlaşılabilirliğini arttırırlar, buna kısaca soyutlama (abstraction) denmektedir.

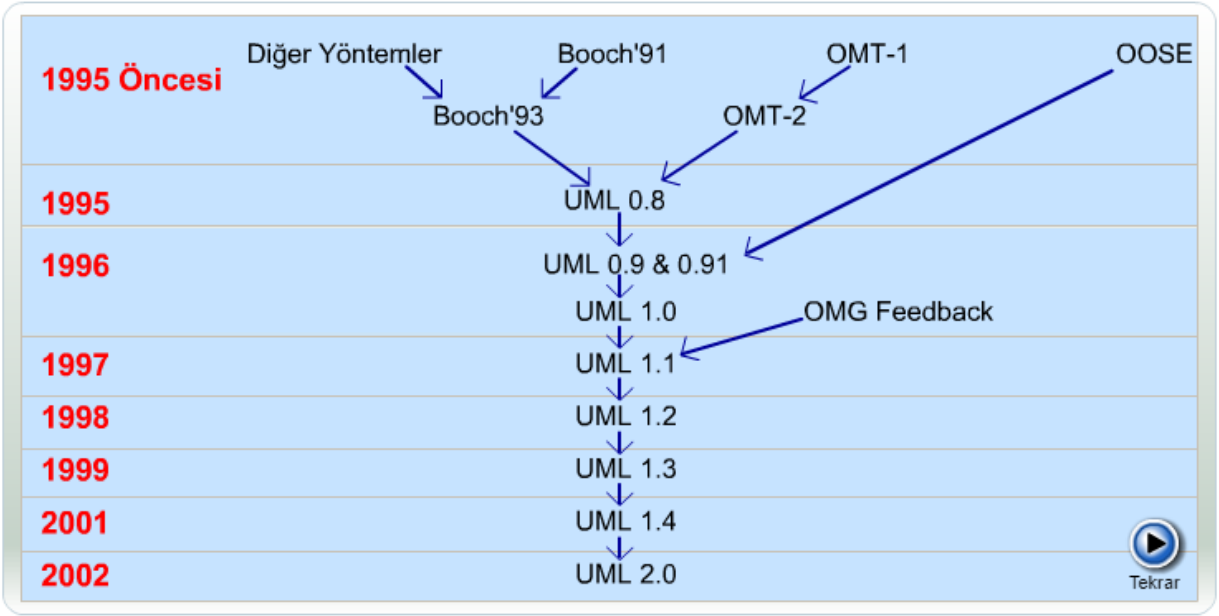
- program geliştirici ya da tasarımcı, bir model oluşturarak proje elemanlarının detaylarda boğulmadan, birbirleriyle nasıl etkileştiği konusunda bir büyük resim sağlar.
- Modellemenin gerekliliğini.

1. Problem çözümü için bir yapı sağlamak
2. Birçok farklı çözüm yolu araştırmak için denemeler yapmak
3. Kompleksliği yönetmek için soyut bir fikir sağlamak
4. Problemlerin çözümü için zamanı azaltmak
5. Gerçekleştirim maliyetini düşürmek
6. Hata yapma riskini düşürmek

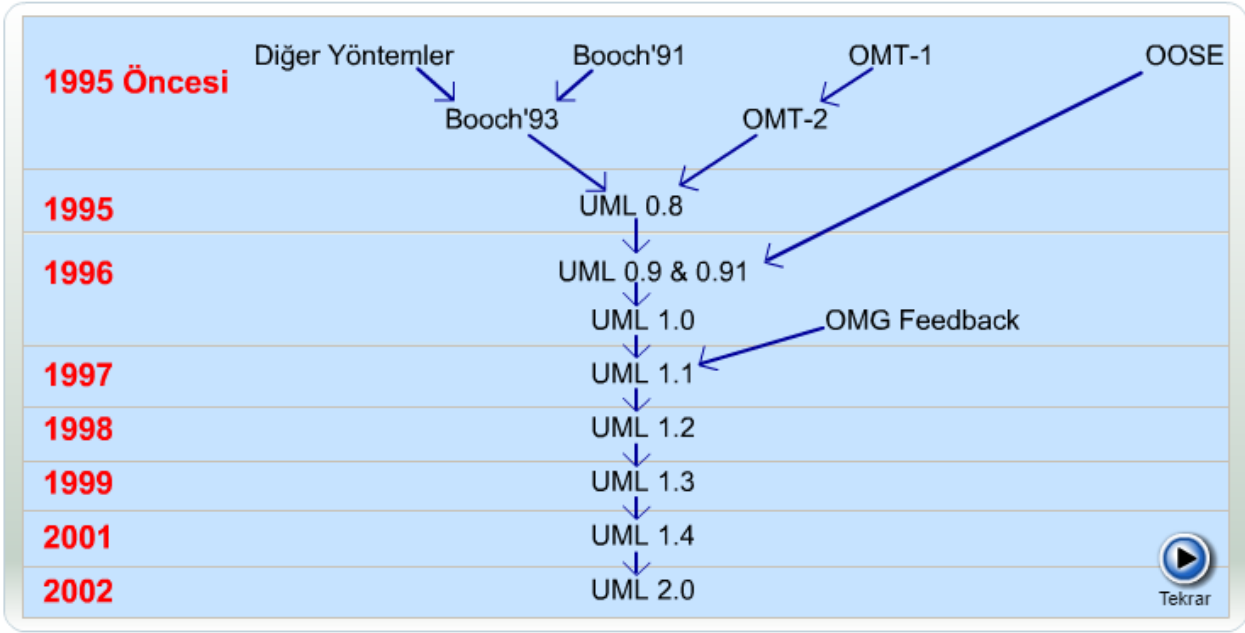
90'lı yıllarda farklı notasyonlara sahip metodolojiler aynı şeyleri farklı yöntemlerle ifade ediyorlardı. Her biri bazı sistemler için mükemmelken, bazıları için işe yaramaz durumdaydı ve hemen hemen hepsi Yazılım Yaşam Döngüsünün (Software Life Cycle) bazı adımlarını tanımlamakta yetersiz kalıyordu.

Bunlardan en yaygın olan üç tanesi: OMT (Object Modelling Technology), Booch ve OOSE (Object Oriented Software Engineering) metotlarıydı.

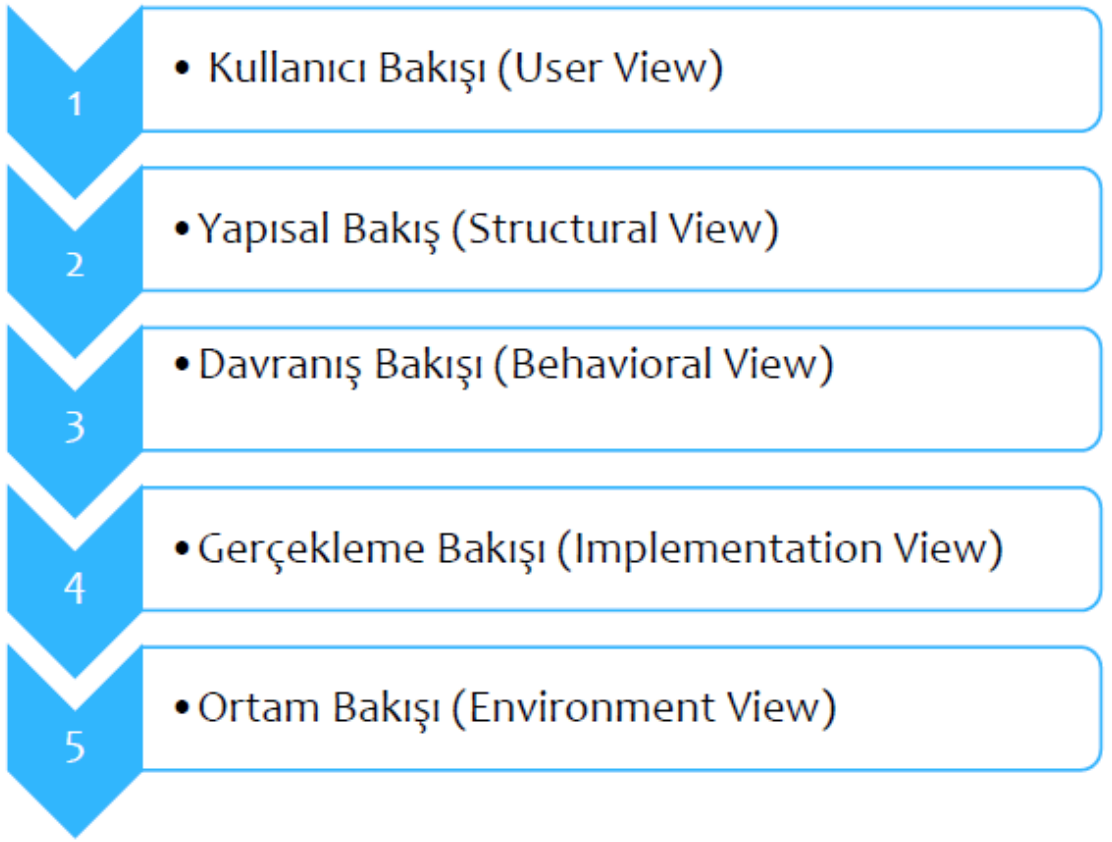
Metot savaşlarının sonu, notasyon açısından UML'nin ortaya konmasıyla geldi. Booch, OMT ve OOSE metotlarının yaratıcıları bir grup oluşturarak kendi yöntemlerinin olumlu taraflarını birleştirerek, komple bir yazılım projesinde sistemi modellemede kullanılabilecek eksiksiz bir modelleme dili geliştirmeye başladılar



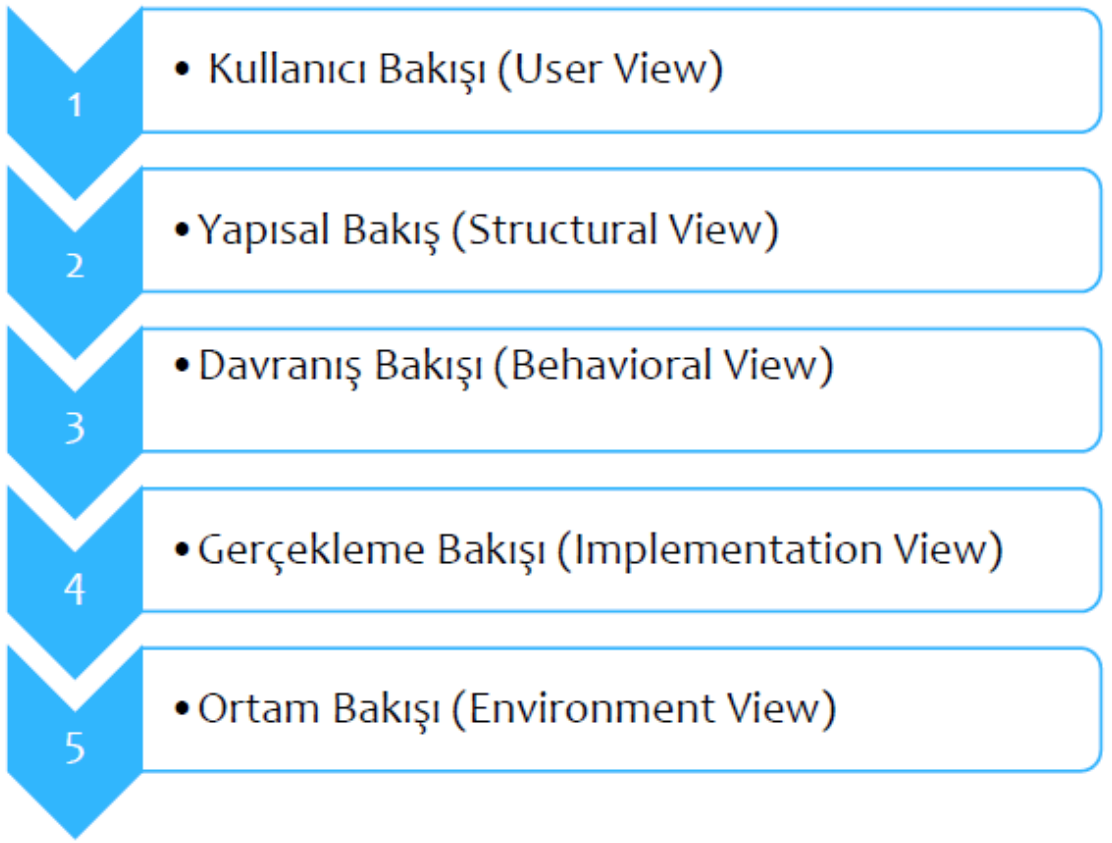
- Microsoft, Oracle, HP gibi büyük firmaların da katıldığı bir UML konsorsiyumu kuruldu ve bu şirketler UML'yi modellemelerinde kullanmaya başladılar. İlk resmi sürüm (ver 0.8), Ekim 1995'te tanıtıldı.
- Kullanıcıdan ve OOSE'nin yaratacısı Ivor Jacobson'dan gelen geri beslemeler ile Temmuz 1996'da sürüm 0.9 v Ekim 1996'da sürüm 0.91 çıkarıldı.
- Kar gütmeyen, bilgisayar endüstrisi standartlarını oluşturan bir organizasyon olan OMG (Object Management Group) tarafından açık standart olarak geliştirilmeye başlandı ve Temmuz 1997'de Sürüm 1.0 çıkarıldı.
- Herhangi bir şirkete veya kişiye ait bir modelleme dili olmayan UML, şu an Rational, MS Visio, Together Soft, vb. birçok modelleme aracı tarafından kullanılmaktadır.



- Yazılımın yaşam döngüsü içinde farklı görev gruplarının projeye ve sisteme farklı bakışları vardır. Müşteriyi hangi işin hangi sırayla yapılacağı, sisteme neler verip sistemden neler alacağı veya işler arası ilişkileri ilgilendirirken bir fonksiyonun detayları ilgilendirmemektedir.
- Çözümleyici açısından bir nesnenin özellikleri, fonksiyonları ve alacağı parametreler yeterli iken, tasarımcı açısından parametrelerin veri tipleri veya fonksiyonun ne kadar bir sürede cevap üretmesi gerektiği, bir nesnenin ne zaman etkin olacağı, yaşam süresi gibi bilgiler de önemli olmaktadır. Teknik yazar ise sistemin nasıl davranacağı ve ürünün işleyişi ile ilgilenmektedir. Bu sebeplerle UML çeşitli bakış açılarını ifade eden diyagramlar içermektedir.



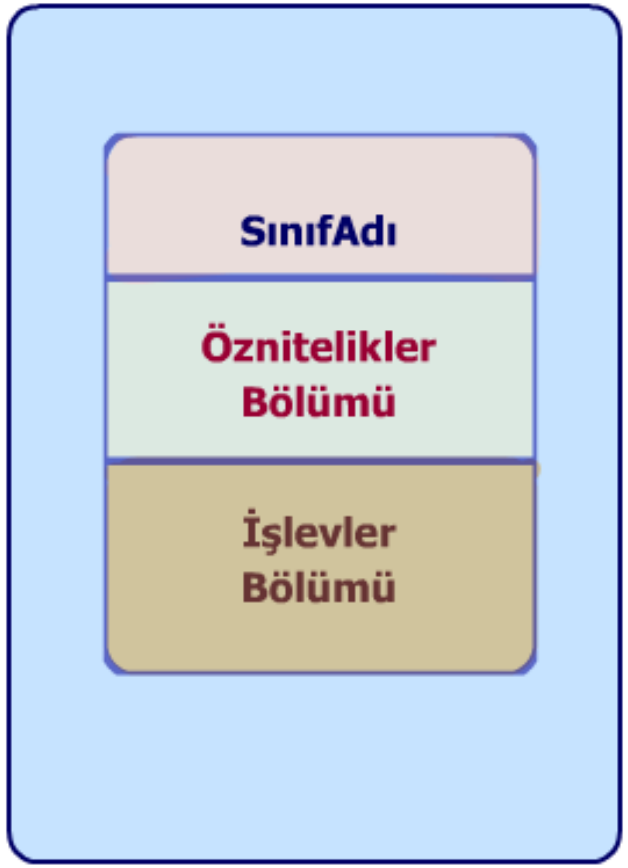
- Çözümleyiciler, tasarımcılar, programcılar, testçiler, kalite sorumluları, müşteriler/kullanıcılar, teknik yazarlar yazılım geliştirme işinde rol alan şahıslardır. Bunlardan herbiri sistemin değişik yönleriyle farklı bakış açılarıyla ve farklı detayda ilgilenirken farklı UML diyagramlarından faydalanırlar. Özetleyecek olursak Projeye dahil olan herkesin faydalanacağı bir veya daha fazla diyagram vardır



- Nesneler arasında ilişki kurmak için UML bir takım grafiksel elemanlara sahiptir. Bu elemanlar kullanılarak diyagramlar oluşturulur.

- Kavramsal (conceptual) bir model olan Sınıf Diyagramları, hem gereksinimlerin müşteriye özetlenmesinde hem de tasarımda kullanılmaktadır. Sınıf Diyagramları, bazı detayları saklayarak müşteri-çözümleyici iletişimini desteklerken, detaylı haliyle de tasarımcıya ve programcıya yardımcı olmaktadır. Sınıf Diyagramlarının tasarımı ilgilendiren kısmı oldukça kapsamlıdır.
- Nesne tabanlı sistemlerin doğru tasarlanması tamamen tasarımcının tecrübe ve bilgisi ile ilgilidir, bu sebeple tasarlanacak Sınıf Diyagramlarının, **türetme** (**inheritance**), **soyut sınıflar**, **sınıf hiyerarşileri**, **tasarım örnekleri** (**design pattern**) gibi detaylarının düşünülmesi gerekmektedir. Bu noktada UML'nin "doğru tasarım nasıl yapılır?" sorusuna cevap vermediğini sadece bir modelleme dili olduğunu belirtmek gerekir. Sistemin performansı veya gelecekte yüzleşeceğimiz güncelleme ve bakım sorunları, yazılan kodun başka projelerde veya ek modüllerde yeniden kullanılabilir olması (code reuse) gibi detaylar tasarımcı açısından çok önemli olmalıdır.

- Sınıf Diyagramları UML 'in en sık kullanılan diyagram türüdür.
- Sınıflar nesne tabanlı programlama mantığından yola çıkarak tasarlanmıştır.
- Sınıf diyagramları bir sistem içerisindeki nesne tiplerini ve birbirleri ile olan ilişkileri tanımlamak için kullanılırlar.
- UML'de sistem yapısını anlatmak için Sınıf Diyagramlarını kullanırız. Sınıf Diyagramları, aralarında bizim belirlediğimiz ilişkileri içeren sınıflar topluluğudur.
- UML' de bir sınıf yanda görüldüğü gibi 3 bölmeden oluşan dikdörtgen ile ifade edilir. Bunlara ek olarak notlar (notes) ve Kısıtlar (Constraints) tanımlanabilir.
- Dikdörtgenin en üstünde sınıfın adı vardır.
- Sınıfın sahip olduğu **Öznitelikler** SınıfAdı'nın hemen altına ikinci bölmeye yazılırken, son bölmeye de sınıfın sahip olduğu **İşlevler** yazılır. Genel bir kullanım şekli olarak sınıflara isim verilirken her kelimenin ilk harfi büyük yazılır.



Öznitelik

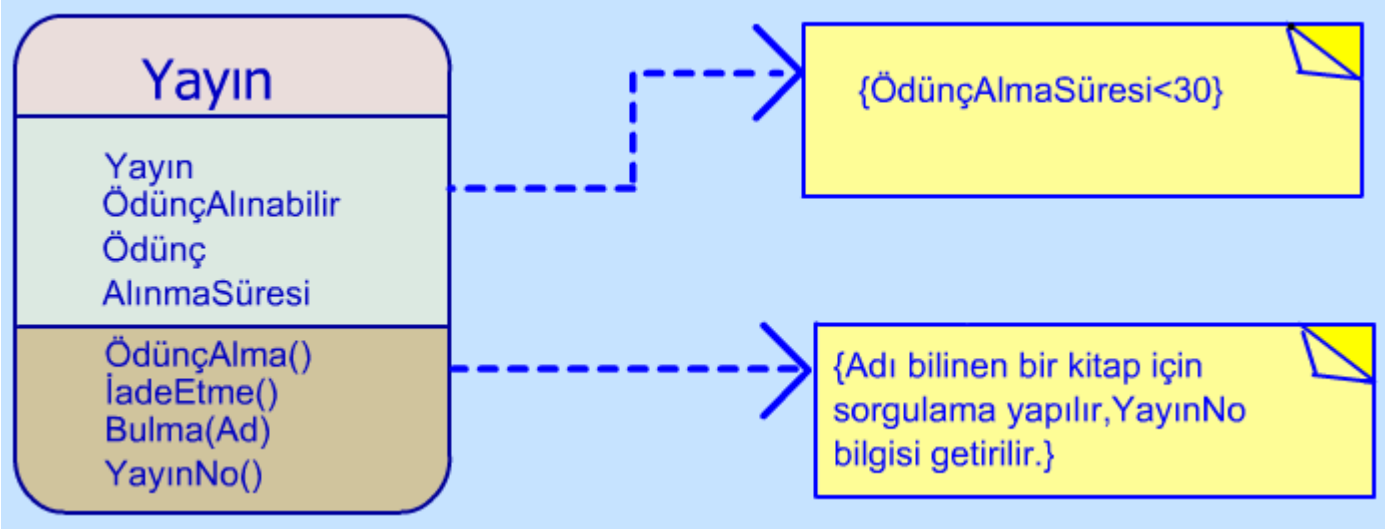
- Sınıfların kendilerini karakterize eden özellikleri vardır. Mesela yayın nesnesinin Ad, ÖdünçAlınabilir ve ÖdünçAlınmaSüresi bilgileri **öznitellik (attribute)** bildirir.
- Bir özniteliğin değerini sonradan değiştirebileceğimiz gibi, yandaki örnekte *Adı*özniteliğinde olduğu gibi varsayılan bir değer de atanabilir. Atama işlemi öznitelik adından sonra "=" işareti yazılıp daha sonra atanacak değer sayısal bir değer ise doğrudan, sayısal bir değer değil ise tırnak içinde yazılarak ilk değer ataması yapılır.
- Atanan bu değerler number (sayı), string (katar), boolean (lojik 0-1), float (ondalık sayı), double (gerçel sayı) veya kullanıcı tanımlı bir veri tipi olabilir.
- Yayın ve Kitap sınıflarının özniteliklerinin UML notasyonunda nasıl ifade edildiği görülmektedir.

İşlev

- Sınıfların bir diğer önemli elemanı da İşlevler (Operations)'dir. İşlevler bir sınıfta iş yapabilen elemanlardır. Bu iş başka bir sınıfa yönelik olabileceği gibi, kendi içindeki bir iş de olabilir. Sınıf Diyagramları'nda işlevler şekildeki gibi, özelliklerin hemen altında gösterilirler.
- İşlevler öznitelliklerden farklı olarak, doğru olarak çalışabilmeleri için dışarıdan birtakım bilgilere ihtiyaç duyabilir, veya birtakım bilgileri dışarıya verebilir, ya da bunların hiçbirini yapmazlar.
- Yandaki Sınıf Diyagramı'nda İşlev1'in çalışması için dışarıdan bir bilgiye ihtiyaç yoktur; ve bu işlev dışarıya da herhangi bir bilgi vermez. İşlev2, işini yapabilmek için birtakım bilgilere ihtiyaç duyar. Örneğin, bir toplama işlemi yapıyorsa toplanacak elemanları ister. İşlev3 ise çalıştıktan sonra işinin sonucunu dış dünyaya verir.

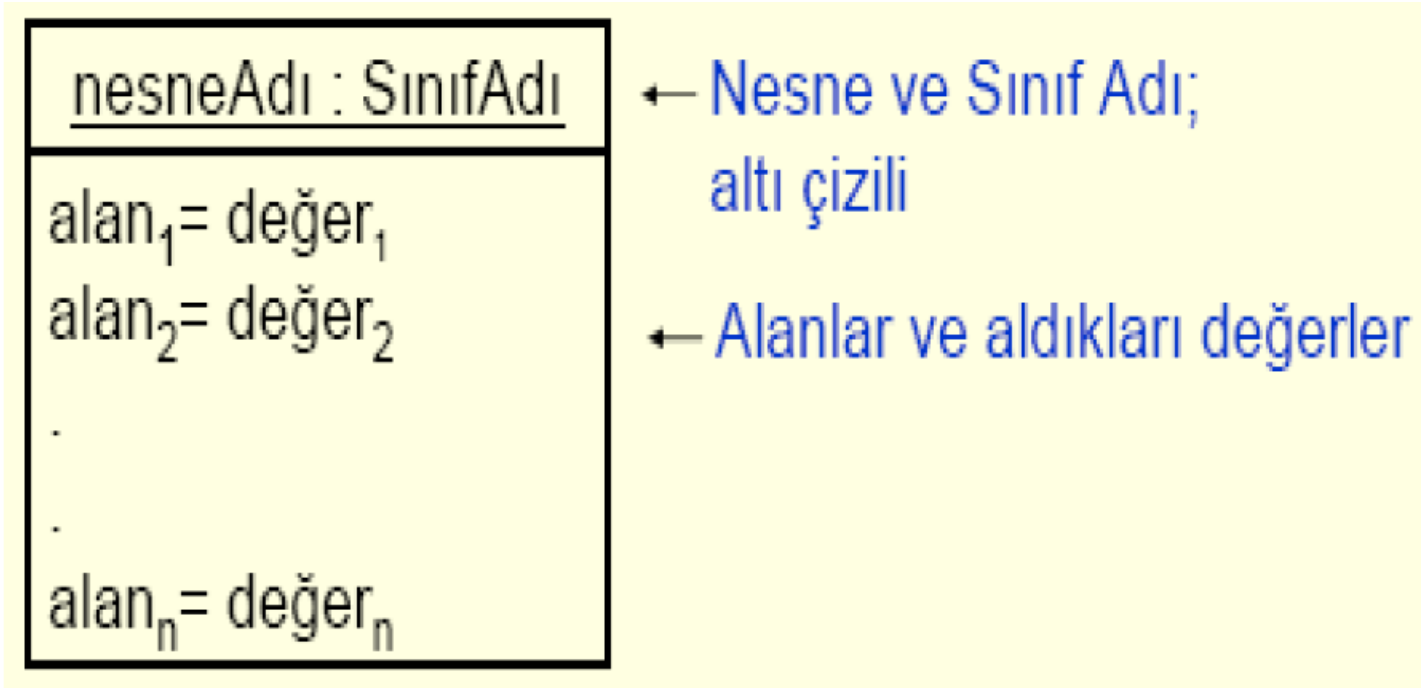
Kısıtlar ve Notlar

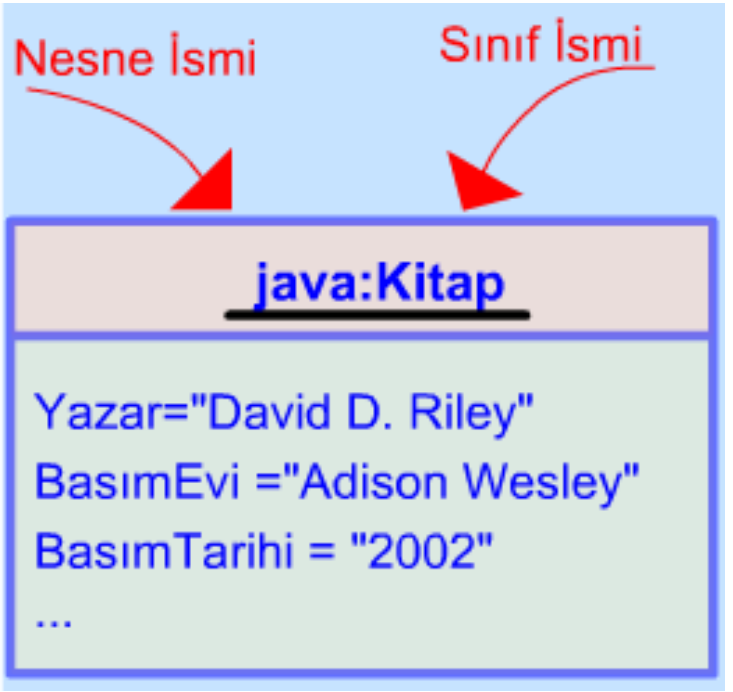
- Bir Sınıf Diyagramında kullanılabilecek temel yapılar Öznitelik ve İşlev olmasına rağmen, kısıtlar (constraints) ve notlar (notes) dediğimiz elemanları da bunlara ekleyebiliriz. **Notlar** genellikle işlevler ve özellikler hakkında bilgi veren opsiyonel kutucuklardır. **Kısıtlar** ise sınıfa ilişkin birtakım koşulların belirtildiği ve parentez içinde yazılan bilgilerdir.

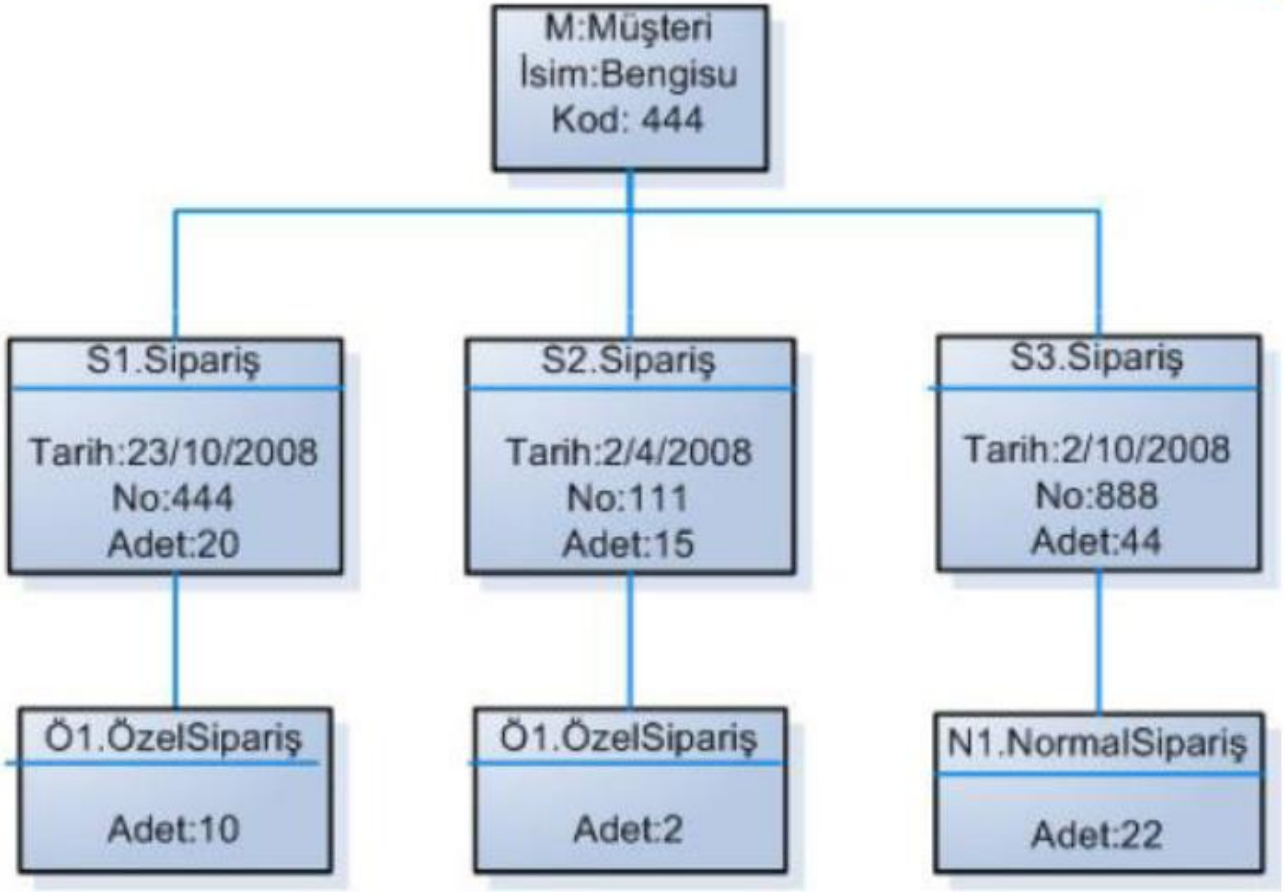


ERİŞİM

- **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.


nesneAdı : SınıfAdı
alan1= değer1
alan2= değer2
.
.
alann= değern
Nesne ve Sınıf Adı; altı çizili
Alanlar ve aldıkları değerler


Nesne İsmi
Sınıf İsmi
java:Kitap
Yazar="David D. Riley"
BasımEvi ="Adison Wesley"
BasımTarihi = "2002"
...

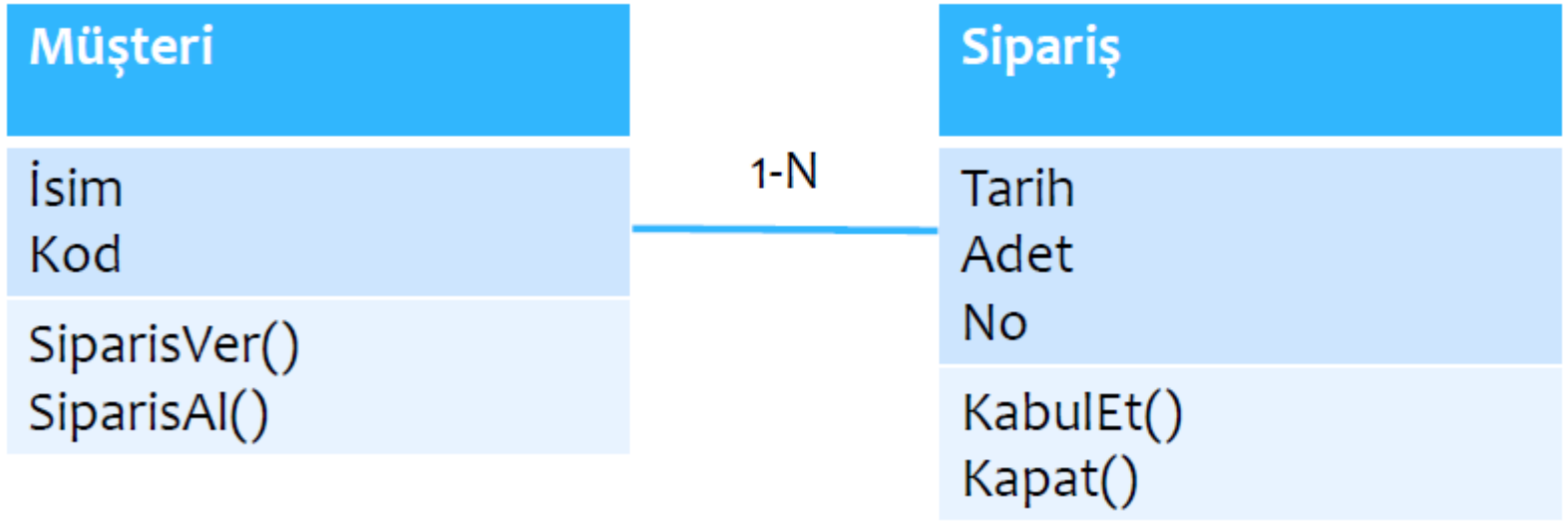

M:Müşteri
İsim:Bengisu
Kod: 444
S1.Sipariş
Tarih:23/10/2008
No:444
Adet:20
S2.Sipariş
Tarih:2/4/2008
No:111
Adet:15
S3.Sipariş
Tarih:2/10/2008
No:888
Adet:44
Ö1.ÖzelSipariş
Adet:10
Ö1.ÖzelSipariş
Adet:2
N1.NormalSipariş
Adet:22

Sınıflar Arası İlişki (Assocation)

- UML'de sınıflar arasındaki ilişki, iki sınıf arasına düz bir çizgi çekilerek ve bu çizginin üzerine ilişkinin türü yazılarak gösterilir. Örneğin, Üye ve Yayın sınıfları olsun. Üye ile Kitap sınıfı arasında "Ödünç alma" ilişkisi vardır. Bu, Sınıf Diyagramı'nda aşağıdaki gibi gösterilir, ve sağ tarafında yazdığı şekilde ifade edilir.

Sınıflar Arası İlişki (Assocation)

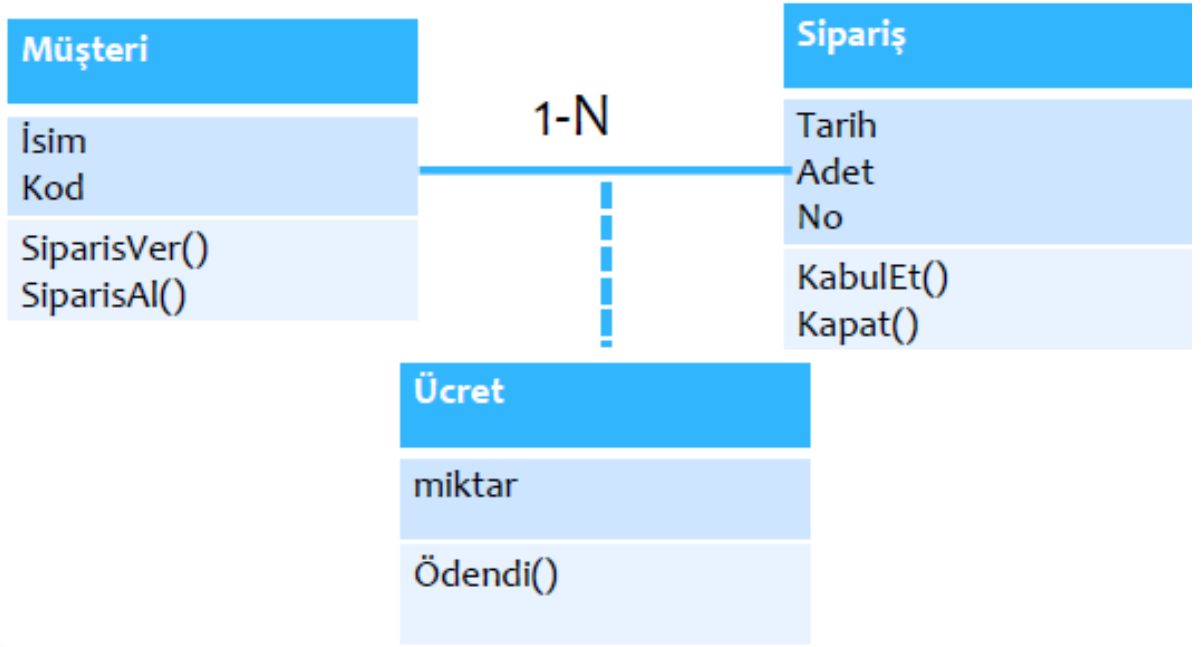
- UML'de sınıflar arasındaki ilişki, iki sınıf arasına düz bir çizgi çekilerek ve bu çizginin üzerine ilişkinin türü yazılarak gösterilir.



İlişki Sınıfları

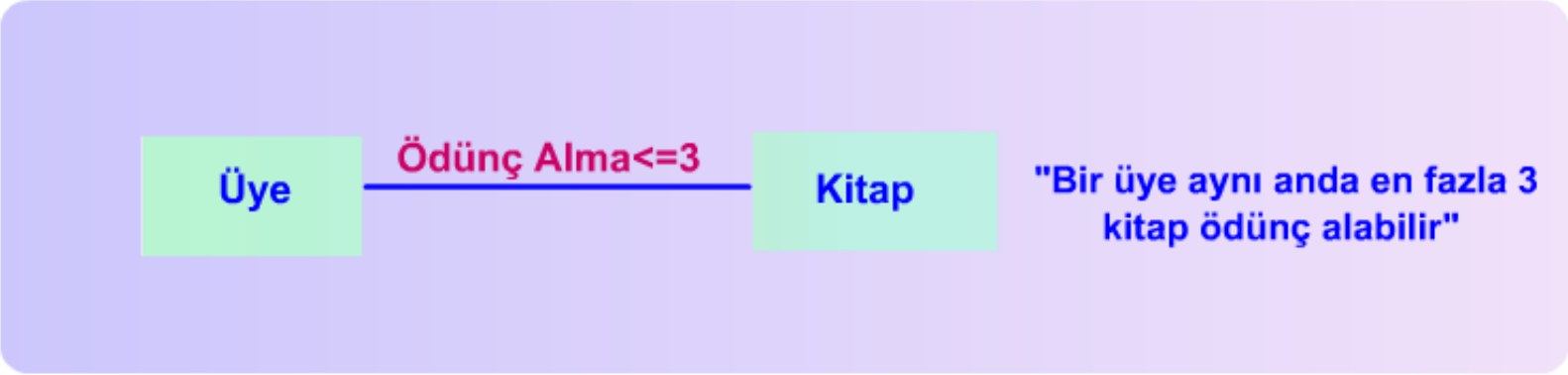
- Bazı durumlarda sınıflar arasındaki ilişki, bir çizgiyle belirtebilecekşekilde basit olmayabilir.
- Bu durumda ilişki sınıfları kullanılır.
- İlişki sınıfları bildigimiz sınıflarla aynıdır.
- Sınıflar arasındaki ilişki eğer bir sınıf türüyle belirleniyorsa UML ile gösterilmesi gerekir.

- Müşteri ile Sipariş sınıfı arasında ilişki vardır. Fakat müşteri satın alırken Ücret ödemek zorundadır
- Bu ilişkiyi göstermek için Ücret sınıfı ilişki ile kesikli çizgi ile birleştirilir.



Kısıtlar

Bazı durumlarda belirtilen ilişkinin bir kurala uyması gerekebilir. Bu durumda ilişki çizgisinin yanına kısıtlar (constraints) yazılır. Örneğimizde bir üye aynı anda en fazla 3 kitap ödünç alabilir olsun.



İlişki Tipleri

İlişkiler her zaman bire-bir olmak zorunda değildirler. Eğer bir sınıf, n tane başka bir sınıf ile ilişkiliyse biz buna bire-çok ilişiki deriz. Örneğin bir dersi 30 öğrenci alıyorsa Öğretmen ile Öğrenci sınıfları arasında 1-30 bir ilişki vardır. Çizelgede bunu gösterirken Öğretmen sınıfına 1, Öğrenci sınıfına ise 30 yazarız. Gösterimi aşağıdaki gibidir:

İlişki Tipleri

Temel ilişki tipleri aşağıdaki gibi listelenebilir:

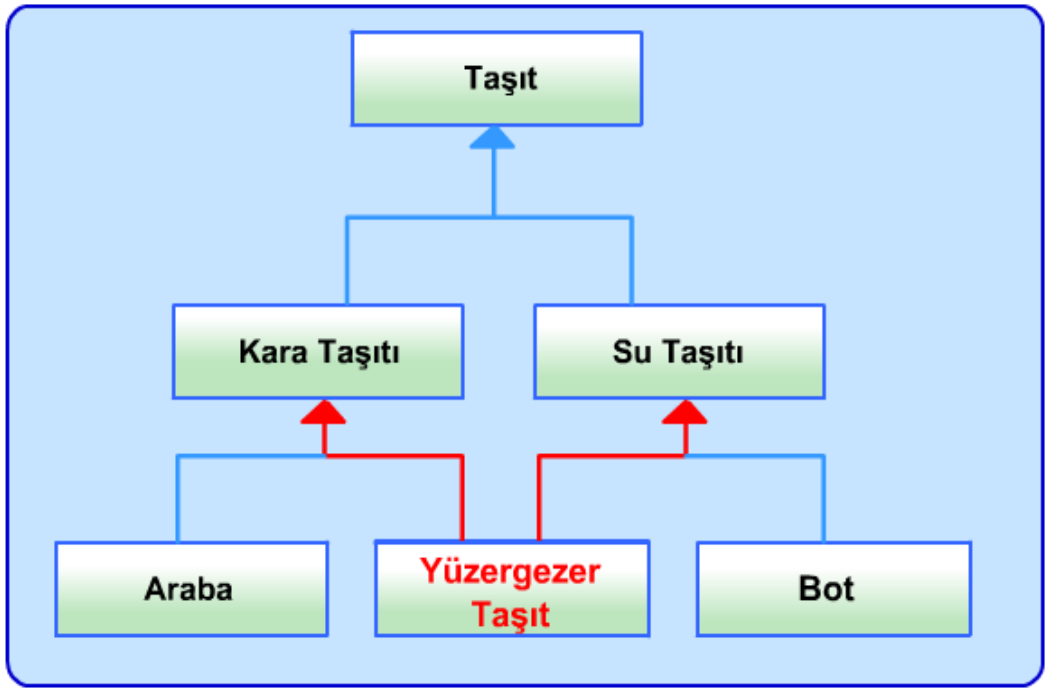
1	Bire-bir
2	Bire-çok
3	Bire-bir veya daha fazla
4	Bire-sıfır veya bir
5	Bire-sınırlı aralık (Örneğin: bire-[0,20] aralığı)
6	Bire-n (UML'de birden çok ifadesini kullanmak için * simgesi kullanılır)
7	Bire-beş ya da Bire-sekiz

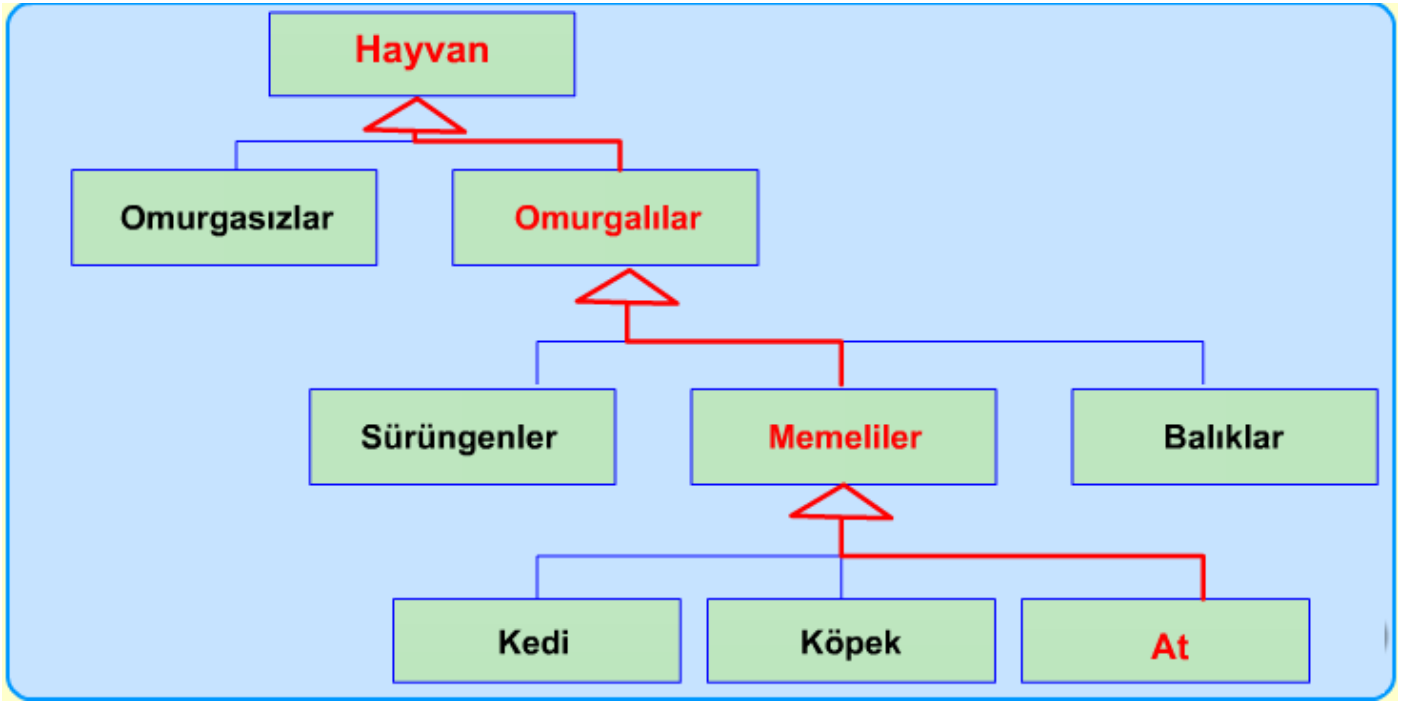
Nesneler arasında ortak özellikler varsa, bunu her sınıfta belirtmek yerine ortak özellikleri bir sınıfta toplayıp, diğer sınıfları da ortak sınıftan türeterek ve yeni özellikler ekleyerek organizasyonu daha etkin hale getirmeye, nesne yönelimli programlamada ***kalıtım (inheritance)*** denir.

Kalıtıma;

- Nesnenin Görevini arttırarak, yeni üye fonksiyonları ilave etmek
- Bir sınıfın üye fonksiyonlarından birinin görevlerinin farklı biçimde yerine getirecek şekilde değiştirmek

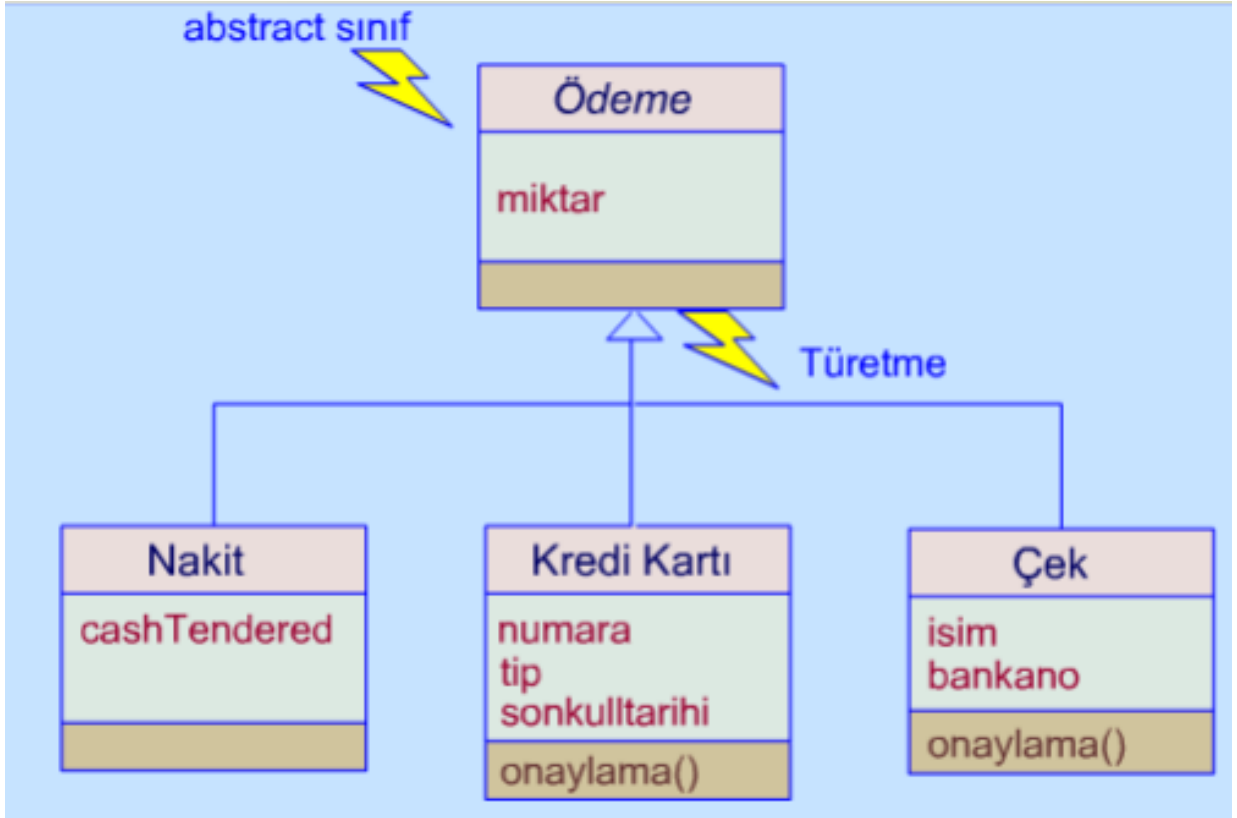
Türetme yapabilmek için öncelikle, tanımlanmış en az bir sınıfın mevcut olması gerekir. Türetme işleminde kullanılan bu mevcut sınıfa taban sınıf (base class), türeme sonucunda ortaya çıkacak yeni sınıfa ise türemiş sınıf (derivated class) denir.


Taşıt
Kara Taşıtı
Su Taşıtı
Araba
Yüzergezer Taşıt
Bot

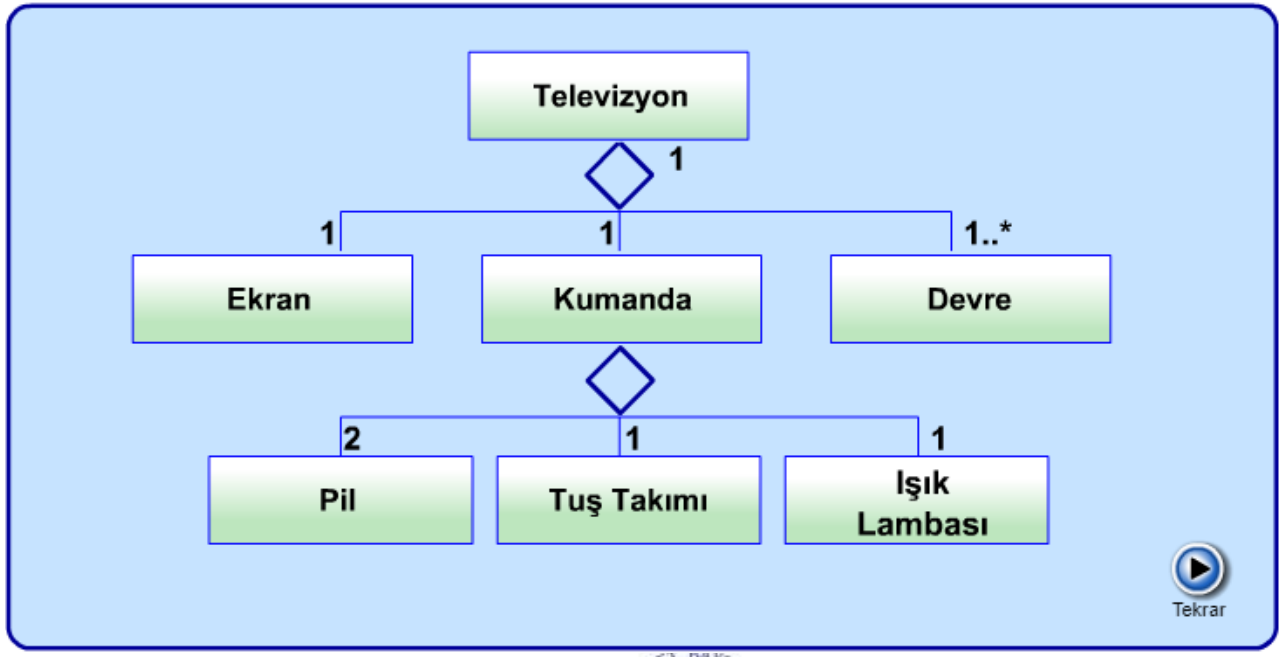

Hayvan
Omurgasızlar
Omurgalılar
Sürüngenler
Memeliler
Balıklar
Kedi
Köpek
At

Soyut (Abstrct) Sınıflar

- Eğer ortak özelliklerin toplandığı sınıftan gerçek nesneler türetilmesini engellemek istiyorsak **Soyut Sınıf (Abstract Class)** oluştururuz. UML'de bir sınıfın Soyut (Abstract) olması için sınıf ismini italik yazarız.
- Aşağıdaki örnekte Ödeme sınıfını Soyut (Abstract) Sınıfa örnek verebiliriz. Ödeme nakit, çekle ya da kredi kartıyla yapılabilir. Üçü için de birer sınıf yaratılır, özellik ve işlevleri anlatılır. Ama üçünün de ortak özelliği olan ne kadar ödeme yapılacağı bilgisi için Ödeme soyut sınıfı yaratılır. Diğer üç sınıf bu sınıftan türetilir.

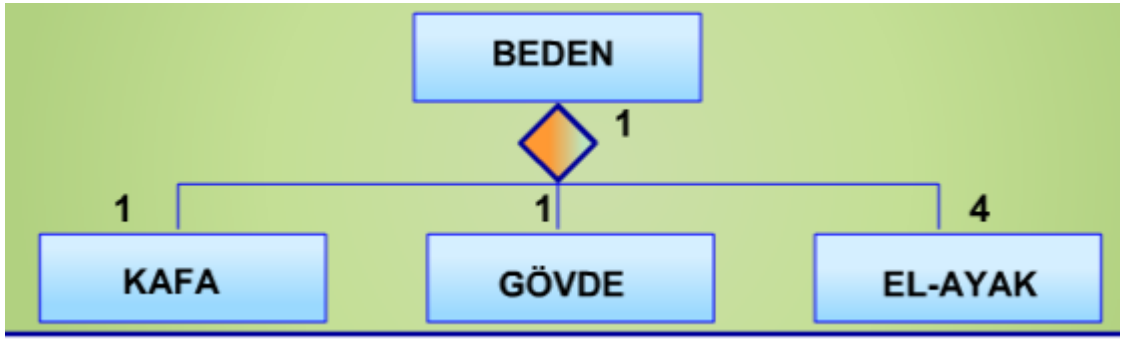


- Bazı sınıflar birden fazla parçadan oluşur.Bu tür özel ilişkiye "Aggregation" denir.Mesela ,bir TV 'yi ele alalım.Bir televizyon çeşitli parçalardan oluşmuştur.Ekran,Uzaktan Kumanda,Devreler vs.. Bütün bu parçaları birer sınıf ile temsil edersek TV bir bütün olarak oluşturulduğunda parçalarını istediğimiz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



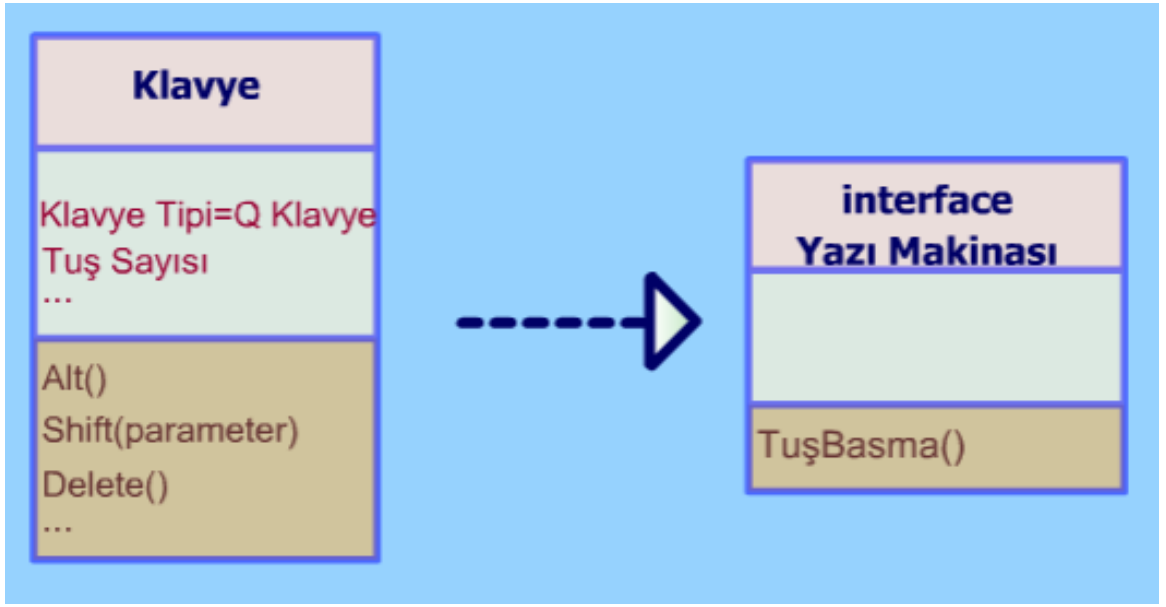
Birleşik İlişkilendirme

- Sadece "Beden" nesnesini oluşturup, sonradan bedene el, ayak, baş takmak çok mantıklı olmazdı. Bu tür ilişkilerin gösterilmesine ise **Birleşik İlişkilendirme** (Composite Assocation) denir. Bu ilişki türü 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 durumda takılacak parçalar **kısıt** (constraint) ile belirtilir.
- Gerçek bir Beden nesnesi oluştuğunda mutlaka ve mutlaka 1 Kafa, 1 Gövde ve 4 El_Ayak nesnesi yaratılacaktır. Görüldüğü gibi sıkı bir parça-bütün ilişkisi mevcuttur.

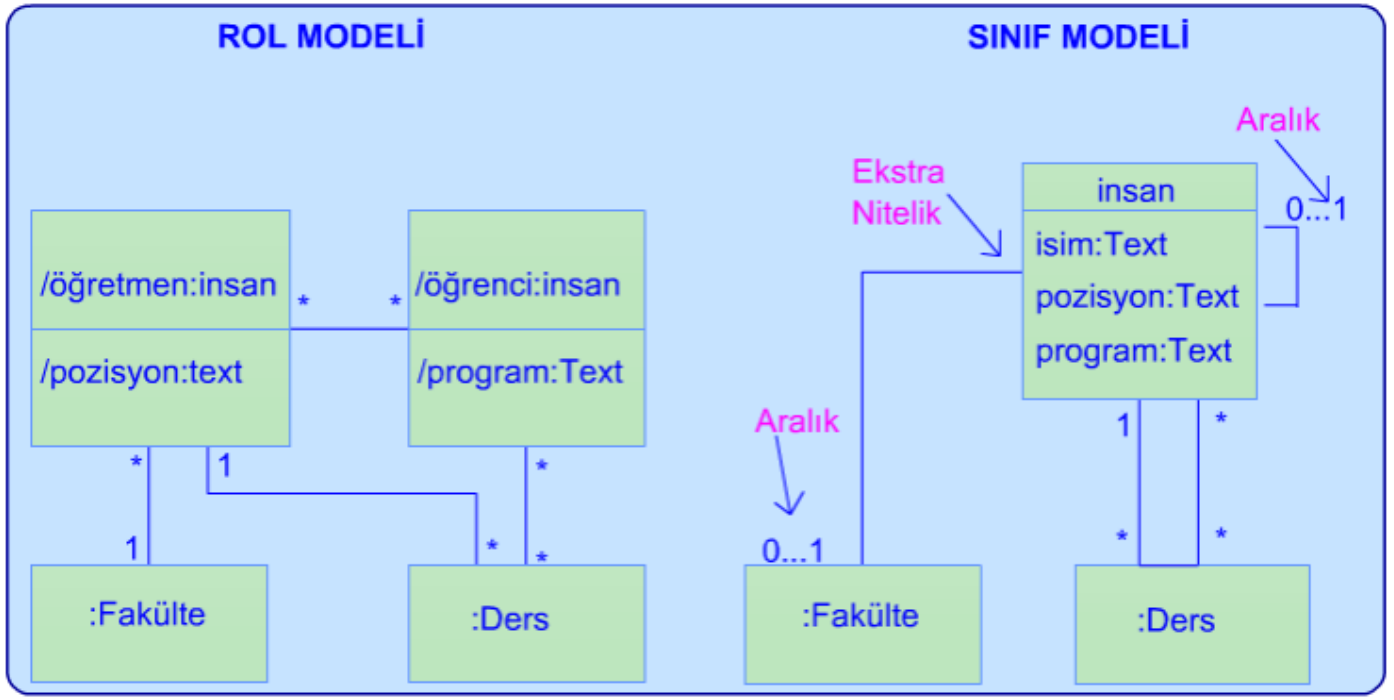


Bazı durumlarda bir sınıf sadece belirli işlemleri yapmak için kullanılır. Herhangi bir sınıfla ilişkisi olmayan ve standart bazı işlemleri yerine getiren sınıfa benzer yapılara arayüz(interface) denir.

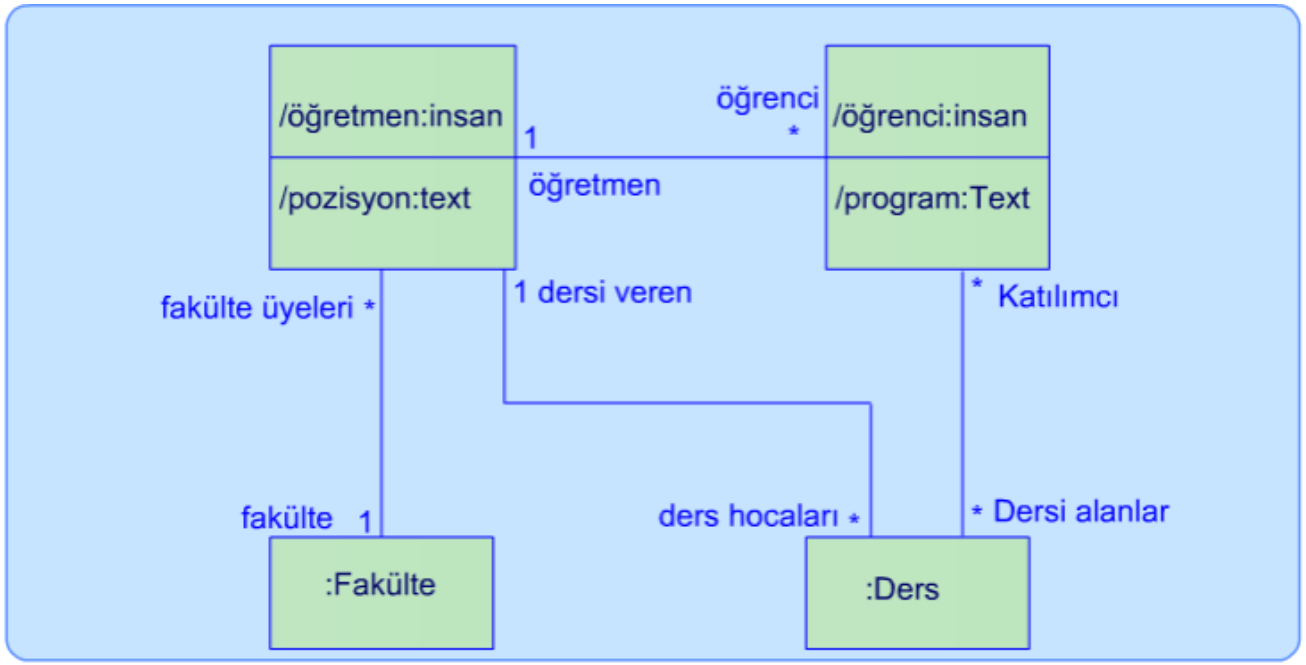
- Arayüzlerin özellikleri yoktur. Yalnızca bir takım işleri yerine getirmek için başka sınıflar tarafından kullanılırlar.
- Mesela, bir "TuşaBasma" arayüzü yaparak ister onu "KUMANDA" sınıfında istersek de aşağıdaki şekilde görüldüğü gibi "KLAVYE" sınıfında kullanabiliriz.



- Olgu tek bir kimliği olan ve hareketi, durumu belli olan bir varlıktır. Sınıflandırıcı ise olgunun tanımıdır. Sınıflandırıcı Rolü ise olgunun kullanımını tanımlamaktadır.
- Sınıflar bütün tanımlamayı verirken, Roller sadece bir kullanışı verir. Aşağıdaki şekil Rol modeli ile Sınıf modeline örnek vermektedir.



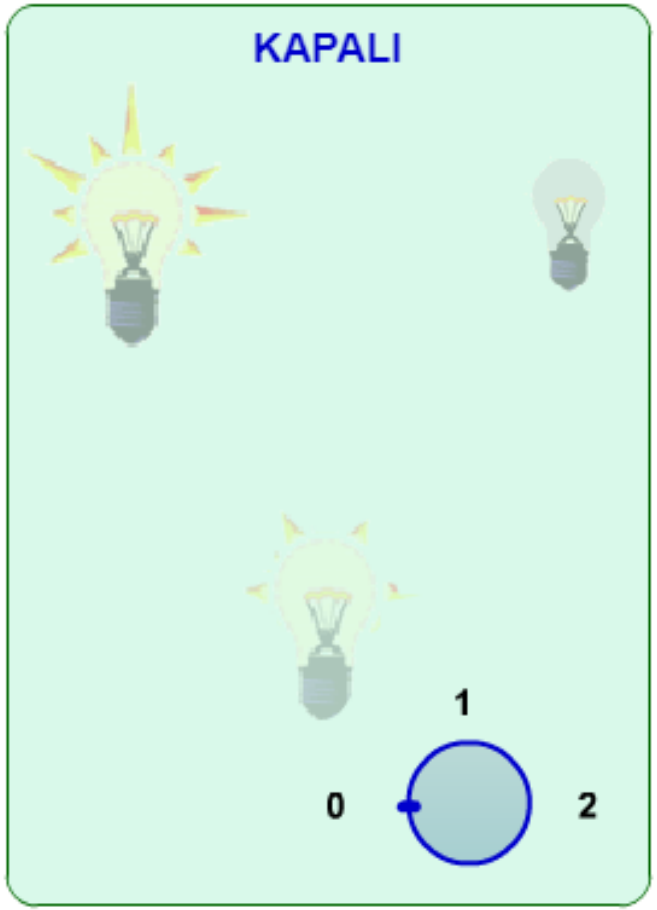
- Sınıf modelinde "İnsan" sınıfı isim, pozisyon ve program özelliklerine sahiptir, aynı zamanda Fakülte ve Ders sınıflarıyla da ilişkilidir. Rol modelinde ise "İnsan" sınıfı öğretmen ve öğrenci rollerine sahiptir. UML'de Rol adlarını belirtirken önce "/" karakteri sonra rol adı yazılır, ":" karakterinden sonra da sınıf adı yazılır.
- Yukarıdaki okul örneğini rol modelini göz önünde bulundurarak daha detaylı çizelim.



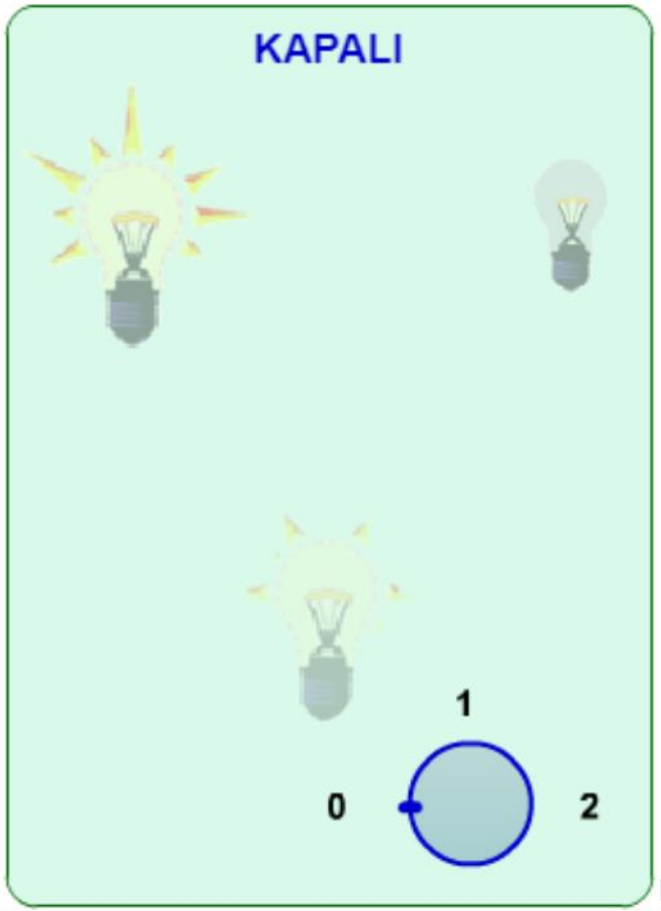
DURUM (STATE) DİYAGRAMLARI

- State diyagramları genel olarak sistemlerin davranışlarını gösterir. Her diyagram tek bir sınıfın nesnelerini ve sistem içerisinde nesnelerin durumlarını (state), geçişlerini (transition), olaylarını (Events) içermektedir.
- Mesela bir lambayı düşünelim, lambanın yaşam süresince 3 farklı durumda olabileceğini söyleyebiliriz:

1. durum: Lambanın kapalı olması,
2. durum: Lambanın açık olması,
3. durum: Lambanın düşük enerji harcayarak yanması



- Bu 3 durum arasında geçişler mümkündür. Bu geçişleri sağlayan ise çeşitli olaylardır. Örneğin lambayı kapalı durumdan açık duruma getirmek için lambanın anahtarını çevirmek gerekir. Aynı şekilde lambayı düşük enerjili halde yanan durumundan açık durumuna getirmek için lamba anahtarını yarım çevirmek gerekir.
- Lamba nesnesinin durum değiştirirken yaptığı işlere Aksiyon (action), lambanın durum değiştirmesini sağlayan mekanizmaya ise olay (event) denilmektedir.
- Sınıf Diyagramları ile nesnenin statik modellemesi yapılırken, Durum Diyagramları ile bir nesnenin dinamik modellemesi yapılmaktadır. Yani Dinamik modellemede nesnenin ömrü boyunca gireceği durumlar belirlenmektedir.

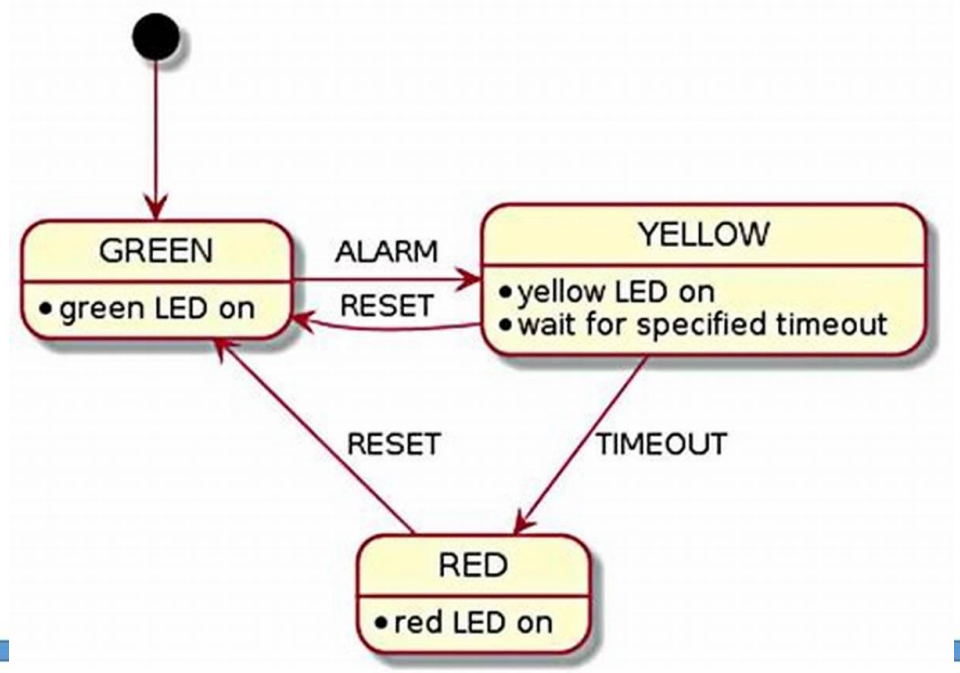


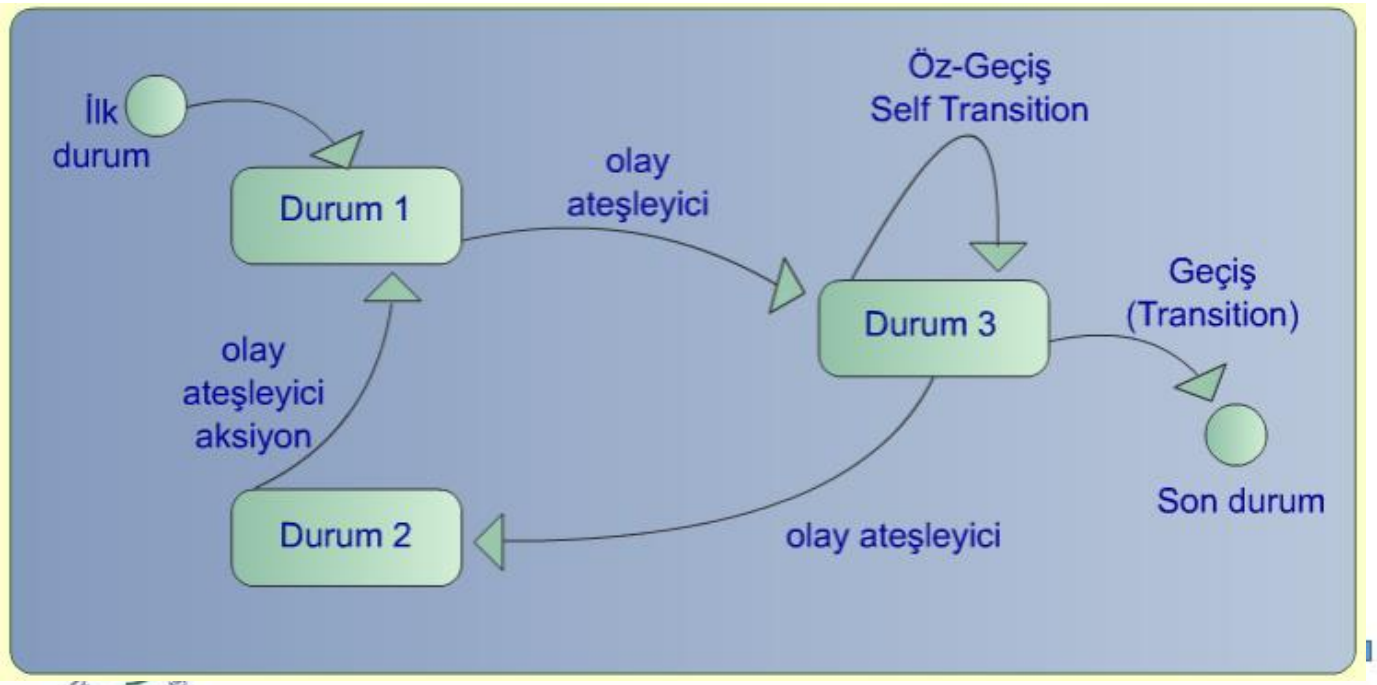
- Durum diyagramları bir sistemin davranışlarını modeller ve bir olay gerçekleştiğinde olası tüm durumları tanımlar. Her bir diyagram bir sınıfın tek bir nesnesini ele alır ve sistem içerisindeki farklı durumlarını irdeler. Bir durum diyagramı aşağıdaki elemanlardan oluşur:

Durum Makinası (State Machine)
Durum (State)
Olay (Event)
Aksiyon (Action)
Geçiş (Transition)
Öz-Geçiş (Self Transition)
İlk Durum (Initial State)
Son Durum (Final State)

State Machine (Durum Makinesi)

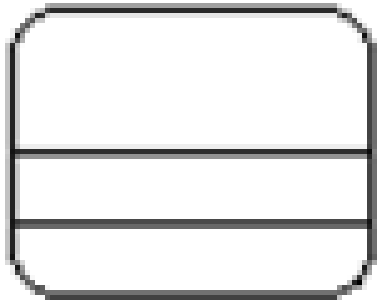
- "State Manchine", bir nesnenin bütün durumlarını bir şema halinde gösteren yapıdır



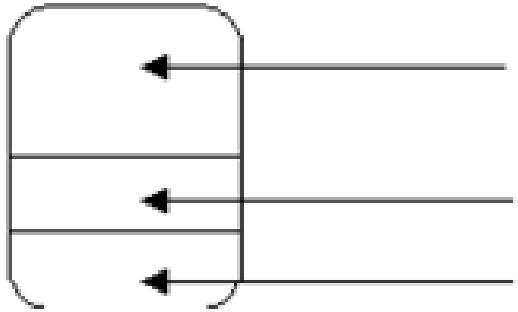


State (Durum)

- Nesnenin ya da sistemin x anındaki durumunu ifade etmek için kullanılır. Köşeleri yuvarlatılmış dikdörtgenler ile gösterilir.



State gösterimi



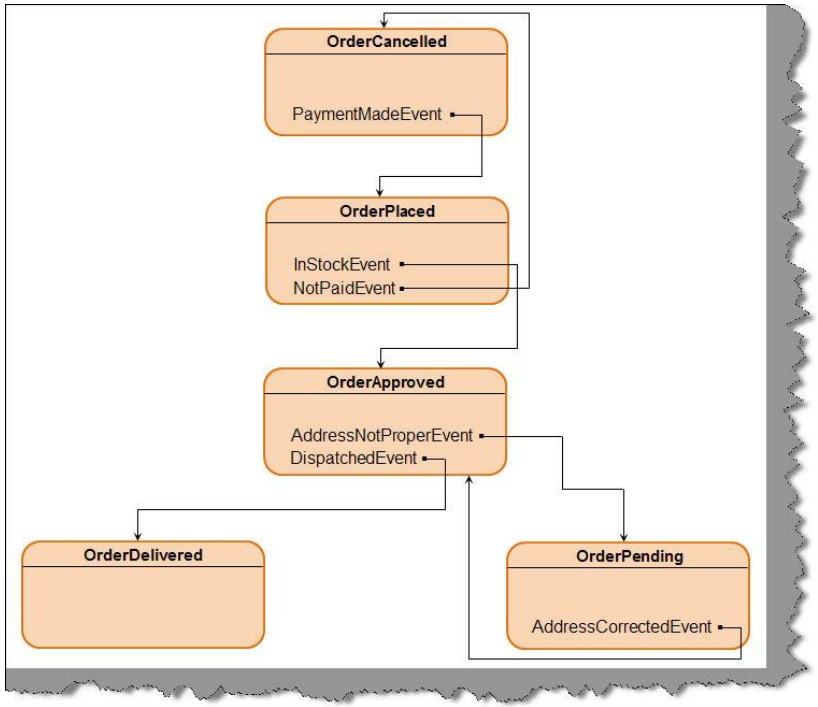
State elemanının yapısal gösterimi

Event (Olay)

- Nesnenin "state" ler arasındaki geçişini sağlayacak yordama event(olay) denir.

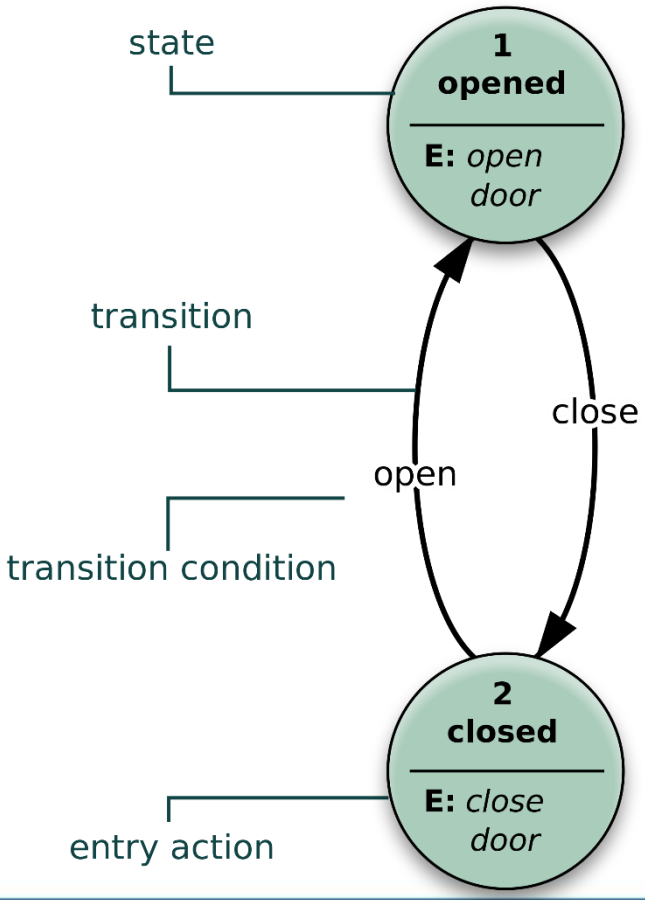
Action (Aksiyon)

- Nesnenin bir durumdan diğer bir duruma geçtiğinde yaptığı işlere "action" denilmektedir. Action, çalıştırılabilir herhangi bir durum olabilir.



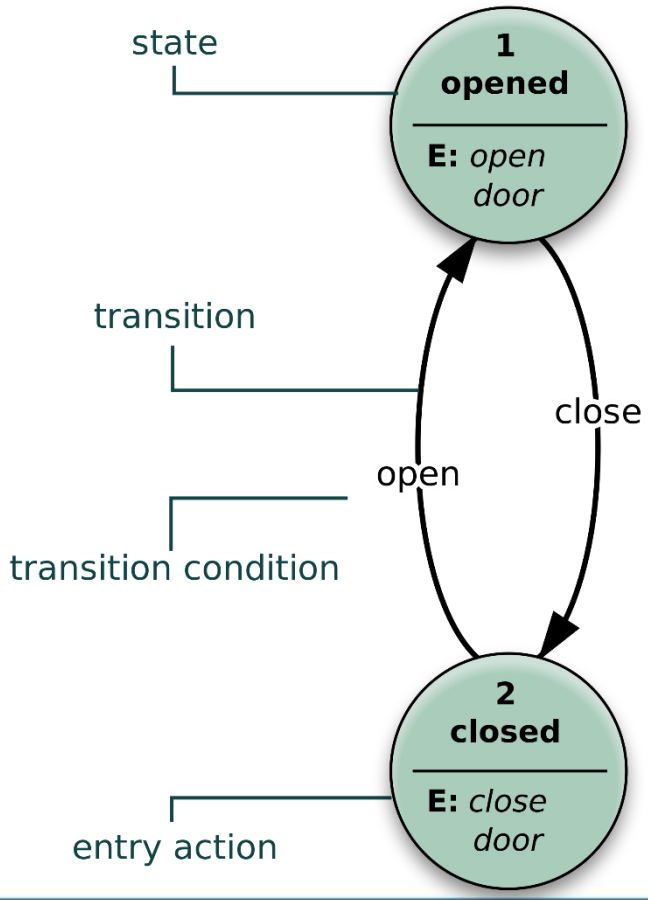
Transition (Geçiş)

- Nesnenin bir durumdan diğer bir duruma geçişini ifade eder. Ok sembolu ile gösterilir.
- Bir nesnenin her durumu arasında bir ilişki olmayabilir. Örneğin lamba açıkken lamba kapatılamıyorsa bu iki durum arasında bir geçiş yoktur denir.
- Bir transiton(geçiş)'da 4 yapı vardır. Bu yapılardan ikisi doğal olarak "hedef(target)" ve "kaynak(source)" durumudur.
- Geçişler kaynak durumdan hedef duruma doğru yapılır. Üçüncü yapı Kaynak durumdan hedef duruma geçişi sağlayan "event trigger(olay ateşleyicisi)" dır. Son yapı ise nesnenin durum geçişi sonrasında ne şekilde davranacağını belirleyen "action" dır.



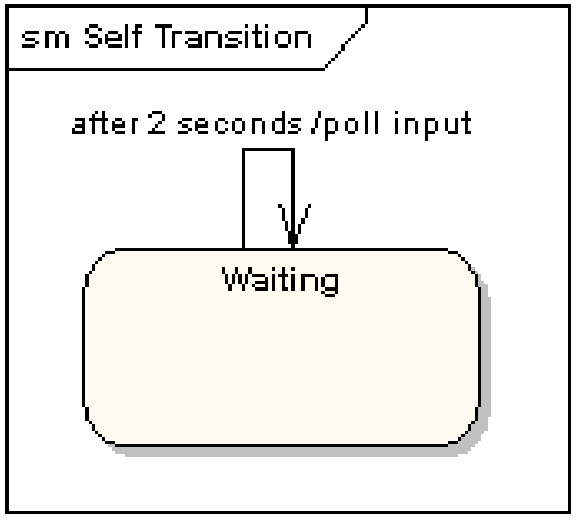
Transition (Geçiş)

- Bir eylemin (Action) bitimiyle beraber o transition a "triggerless" transition denir.
- Eğer bir olay, bazı olay veya eylem tamamlanmasından sonra meydana gelirse, eylem veya olay “guard condition (Nöbet Durumu)” olarak adlandırılır.
- Çoğu zaman iki durum arasında geçiş sağlayabilmek için, bir event (olay) gerçekleştirilmiş olmasının ötesinde, tanımlı bir koşulun sağlanmış olması da gerekir.
- Guard condition (Nöbet Durumu) tamamlandıktan sonra geçiş(Transition) oluşur. Bu nöbet durum (Guard Condition) / olay(Event) / eylem (Action) köşeli parantez ile gösterilir.
- Diyagramlarda koşullar baklava sembolü ile gösterilirler.



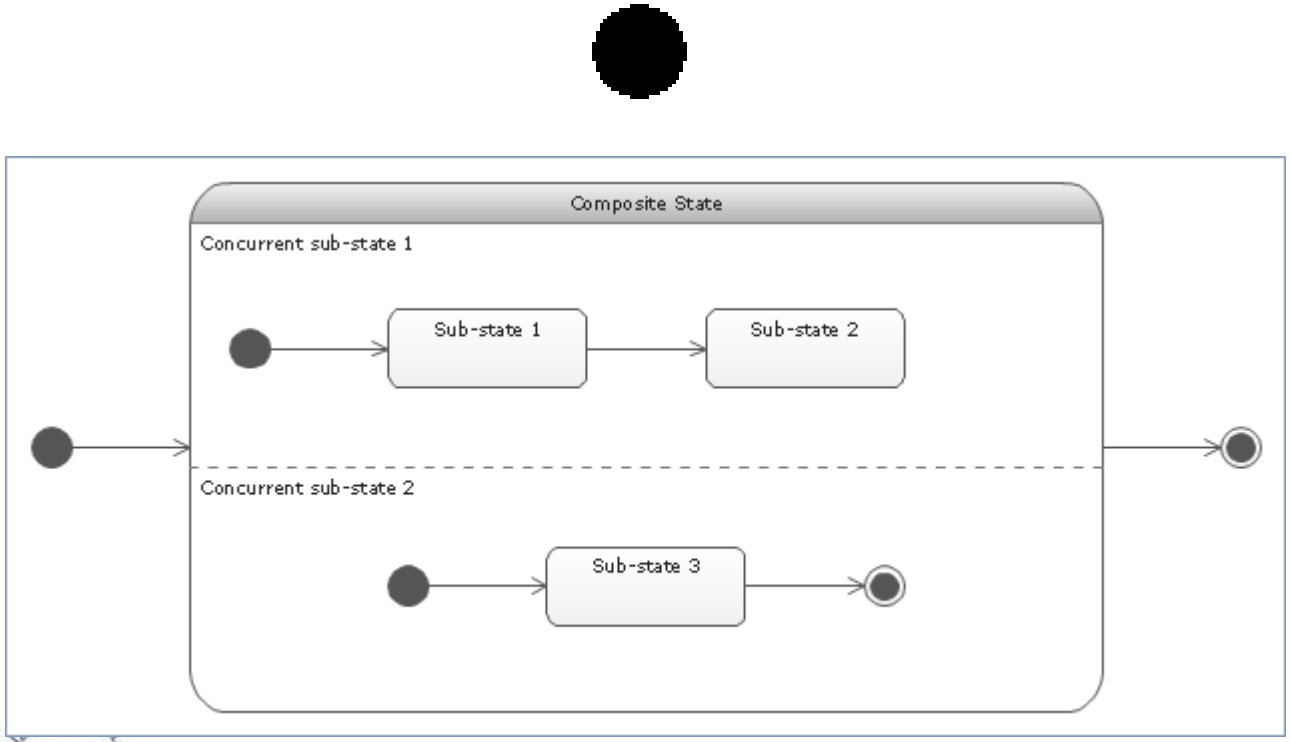
Self Transition (Öz Geçiş)

- Bazen bir nesnenin durum geçişleri farklı iki durum arasında olmayabilir.
- Mesela lamba açık durumda olduğu halde lambayı tekrar açmaya çalışmak "self transition" kavramına bir örnektir.
- Kısaca hedef durum ve kaynak durumun aynı olduğu durumlar "Self Transition" olarak adlandırılır.
- Daha net söylersek; geçişin kaynak ve hedef durumu aynı olur. Bu durum özgeçiş olarak adlandırılır.



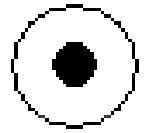
Initial State (İlk Durum))

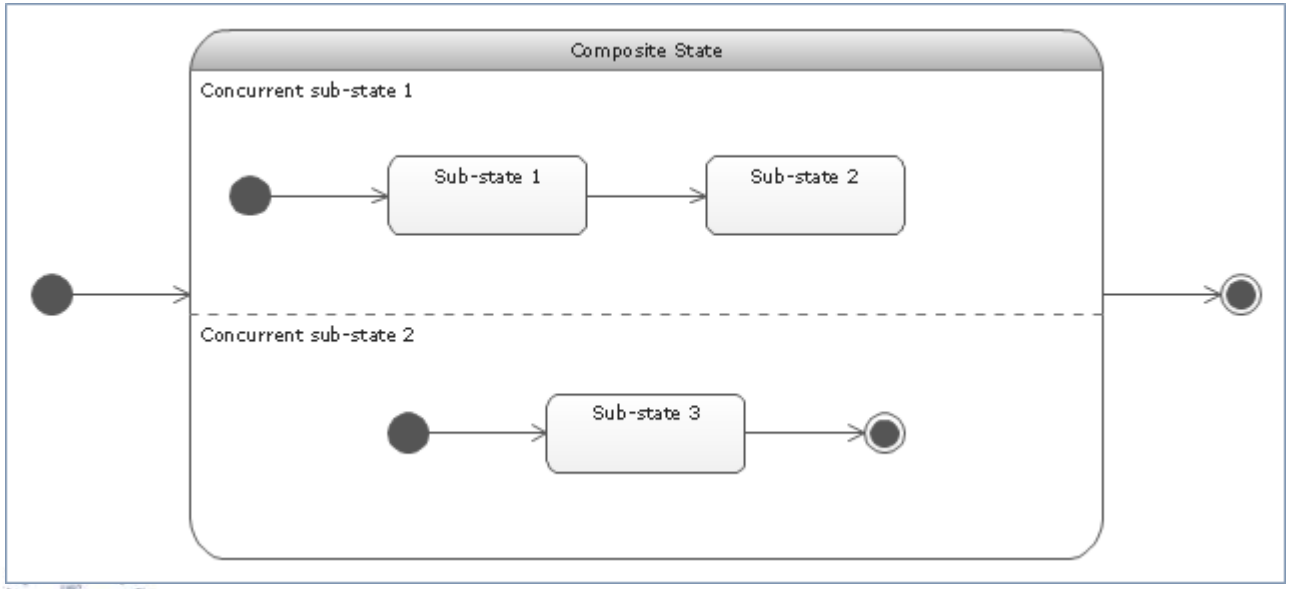
- Yaşam döngüsünün ilk eylemi ya da başlama noktasını ifade eden elemandır.
- İçi dolu yuvarlak ile gösterilir.
- Sözde durum (pseudo state) olarak da adlandırılır. Sözde durum denilmesinin sebebi değişkeni veya herhangi bir eyleminin olmayışıdır.



Final State (Son Durum)

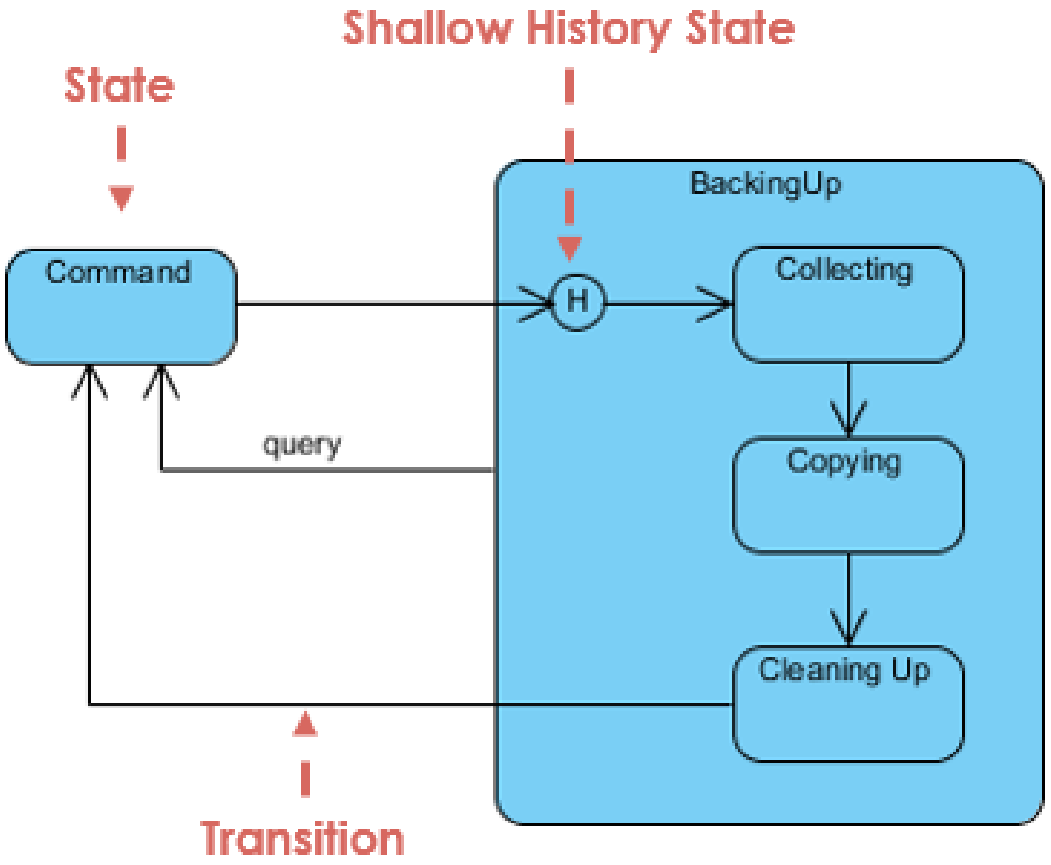
- Sistemin yaşam döngüsü içerisindeki son durumunu ifade eder.
- Bir nesne birden fazla "Final State" durumuna sahip olabilir..





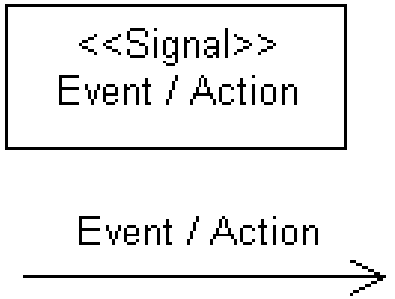
History States (Geçmiş Durumlar)

- Bir akışta, bir nesne, transa ya da bekleme durumuna geçebilir. Belirli bir olayın gerçekleşme süresinde bekleme durumuna girdiği zaman son aktif durumuna geri dönmek istenebilir.
- Daire içerisinde H harfiyle gösterilir.



Signal (Sinyal)

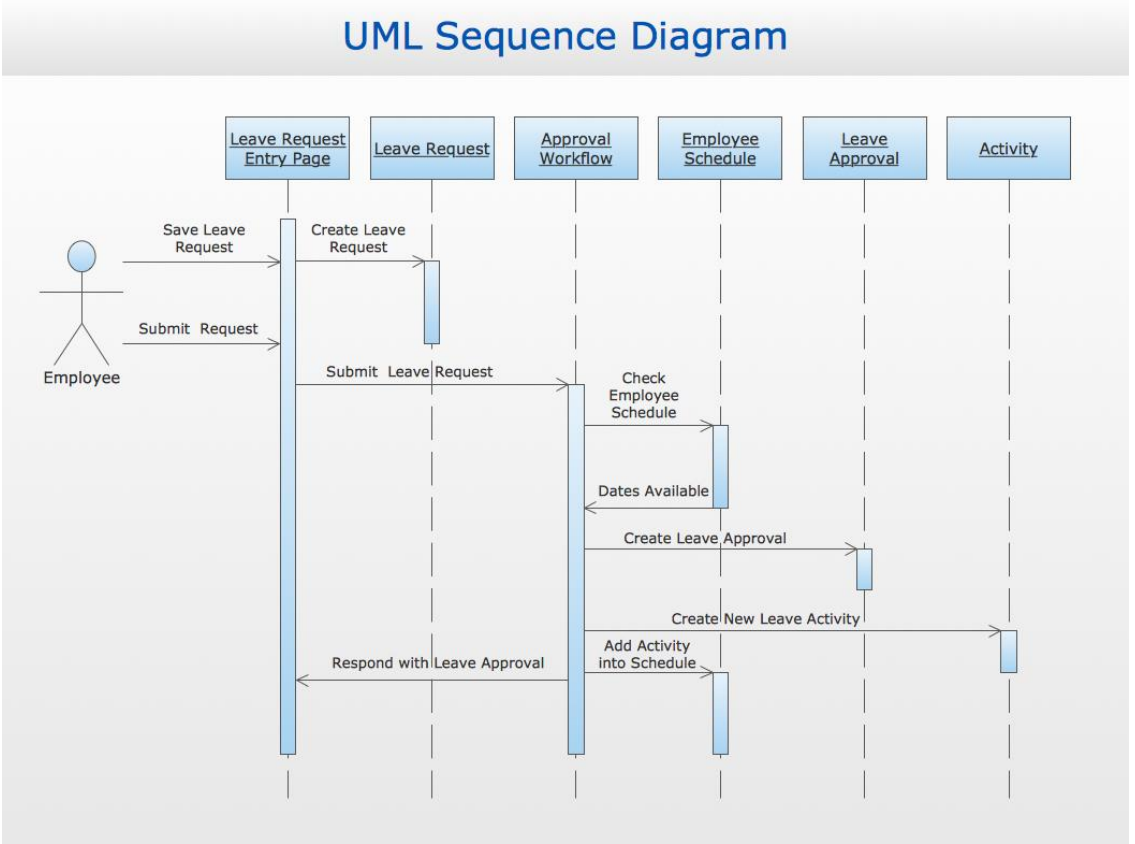
- Duruma bir mesaj yada tetikleyici gönderimi olduğunda geçiş oluşur ve mesaj event (olay) ile gönderildiğinde sinyal olarak adlandırılır. << Sinyal>> şeklinde gösterilir.



ARDIŞIL DİYAGRAMLAR

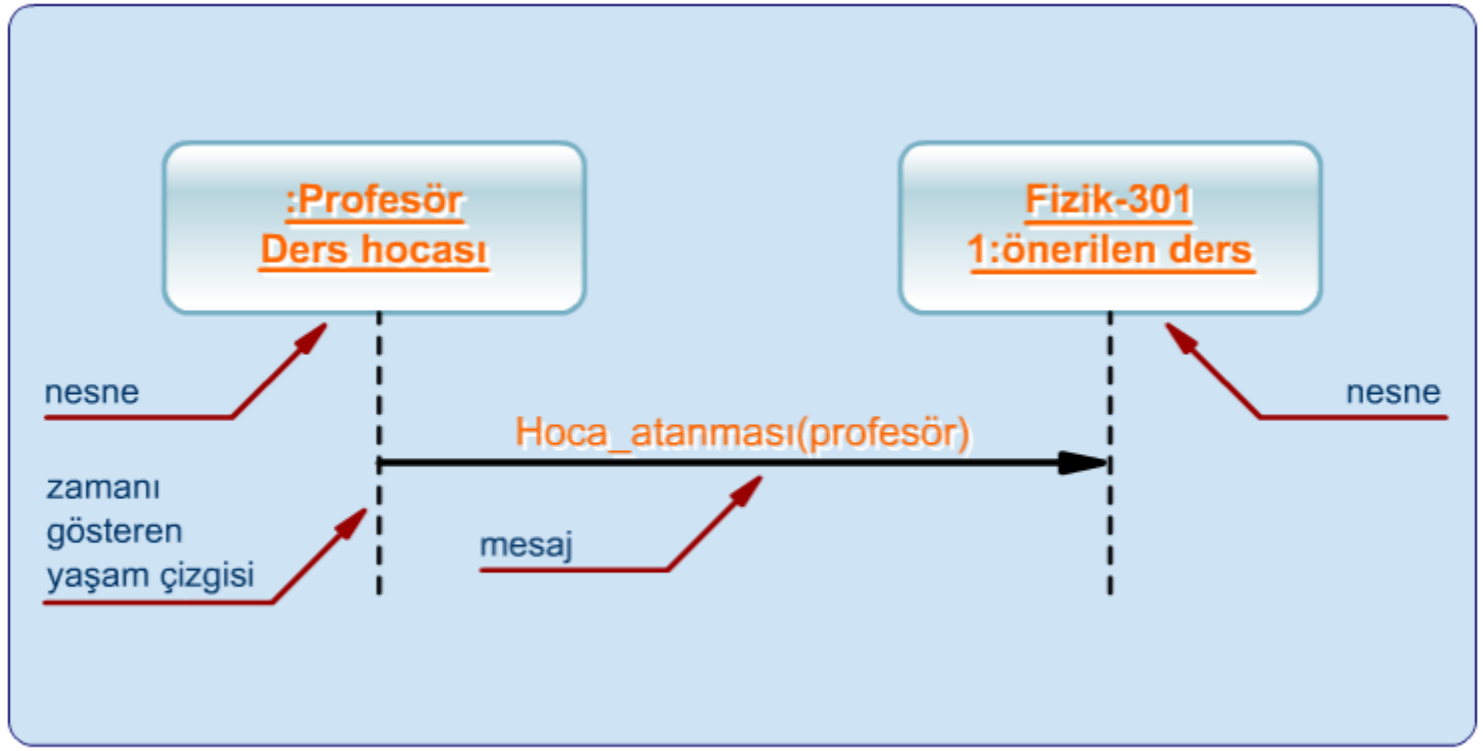
- Ardışıl Diyagramlar, zaman içerisinde nesnelerin birbirleriyle nasıl bir iletişim içinde olduklarını gösterir. Böylece genişletilen görüş alanına, önemli bir boyut da dahil edilir: Zaman. Buradaki anahtar fikir, nesneler arasındaki etkileşimin belirli bir ardışıllık içinde gerçekleşmesi ve bu ardışıllığın başlangıçtan sona doğru giderken belirli bir süre tutmasıdır.

- Ardışıl (Sequence) diyagramlar, sistemdeki nesneler ya da bileşenler arasındaki mesaj akışının olaylarını, hareketlerinin ardaşık şekilde modellenmesinde kullanılan diyagramlar sequence diyagramlardır.



- Ardışıl Diyagramlar nesneler, mesajlar ve zaman olmak üzere üç parçadan oluşurlar. ***Nesneler***, içinde ismi altı çizili olarak yazılmış dikdörtgenler ile, ***mesajlar*** kesintisiz düz ok işareti ile, ***zaman*** ise aşağı doğru dikey kesikli çizgi ile gösterilir.

- Sequence diyagramlarının akışı soldan sağa doğru olmalıdır. Sequence diyagramları oluşturmadan önce senaryo (use case) oluşturulmalıdır.

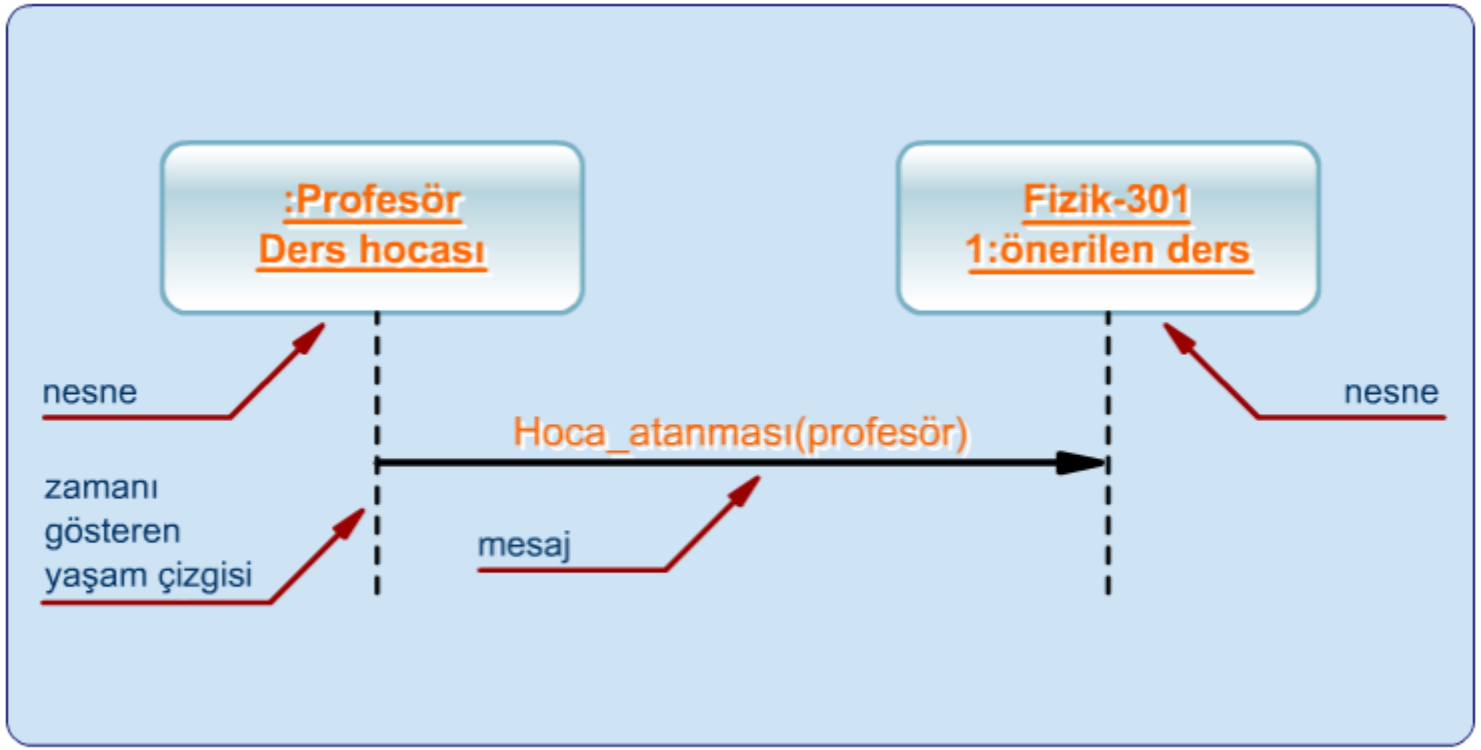


Şekil 1. Basit Ardışıl Diyagram örneği (Derse profesör atanması)

- Bir sequence diyagramı nesnelerden, mesajlardan ve zaman çizelgesinden oluşmaktadır.

Sequence diyagrami iki boyutludur:

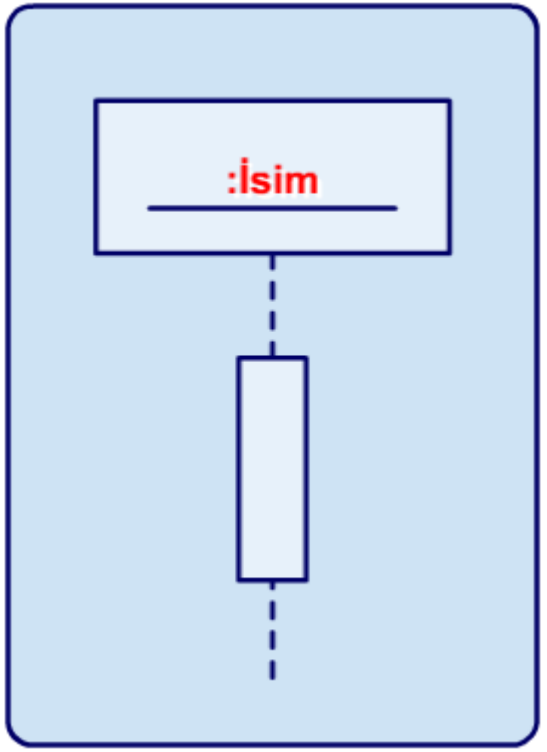
1. **Dikey boyut**: Mesajların/olayların sırasını oluşma zamanı sıralarına göre gösterir.
2. **Yatay boyut:** Mesajın gönderildiği nesne örneklerini (object instances) gösterir.



Şekil 1. Basit Ardışıl Diyagram örneği (Derse profesör atanması)

Nesneler (Objects)

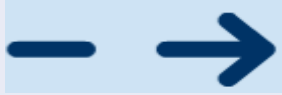
- Nesneler diyagramın en üstünde soldan sağa doğru dizilirler.
- Nesneler, diyagramı basitleştirmek için herhangi bir sırada yerleştirilebilir. Her bir nesne yaşam çizgisi (lifeline) denilen kesikli çizgiler ile aşağı doğru yayılır.
- Yaşam çizgileri boyunca yer alan dar dikdörtgenlere *aktivasyon (activation)* denilir. Aktivasyonlar, nesnenin taşıdığı işin gerçekleşmesini temsil eder. Dikdörtgenin genişliği aktivasyonun devam süresini belirler. Bu yandaki şekildeki gibi gösterilir.



Mesajlar (Messages)

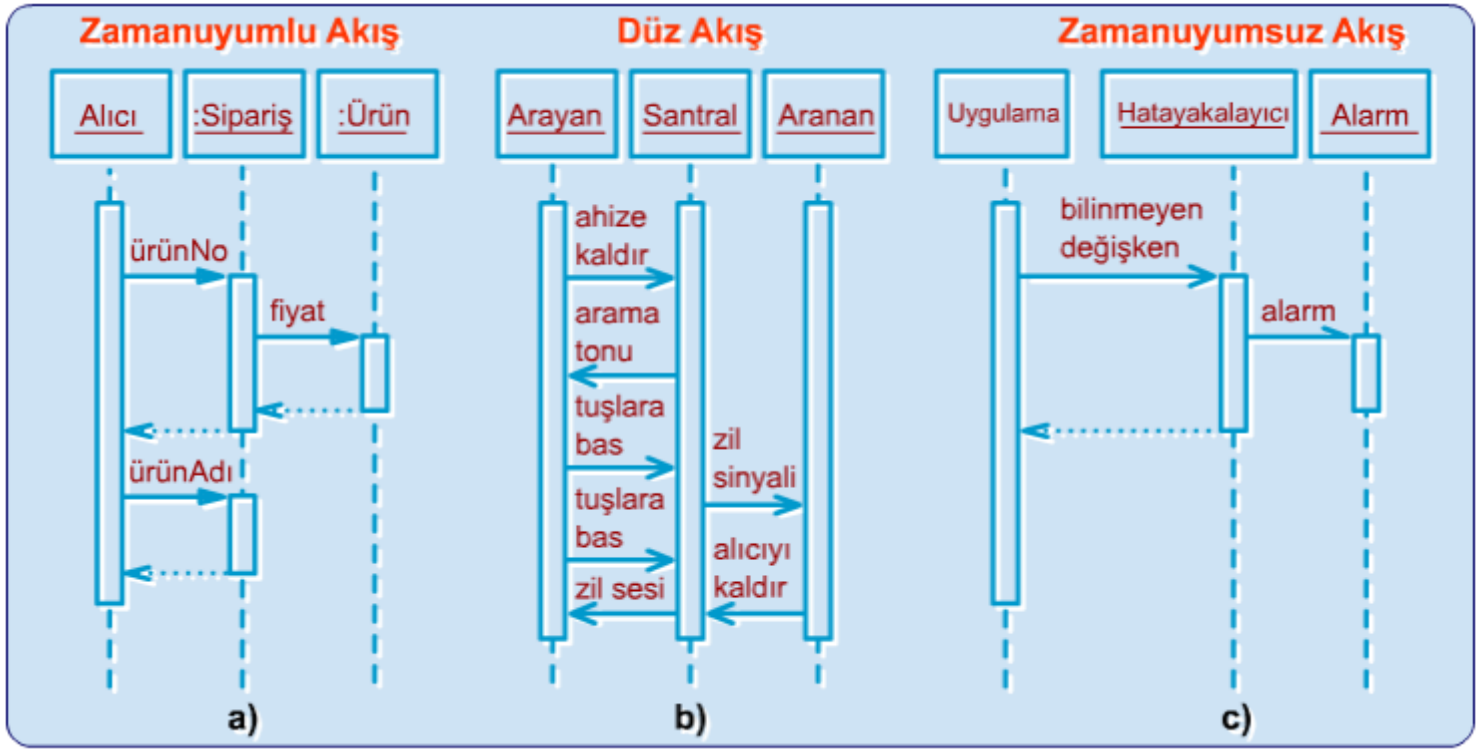
- Ardışıl diyagramda farklı nesnelerin birbirleri ile etkileşimi mesajlar ile gösterilir.
- Mesaj gösterimi mesajların tipine göre değişir. Sequence diyagramlarda mesajlar, basit(simple) mesajlar, nesne oluşurken ya da bellekten silinirken kullanılacak özel mesajlar ve mesaj yanıtları olarak değişik şekillerde gösterilir.

Mesajlar (Messages)

Mesaj Tipleri ve Gösterimleri	
Basit(Simple) Mesaj Tipi: Basit mesajlar nesneler arasındaki akış kontrolünün iletimini göstermek için kullanılır. Nesnelerin methodlarını doğrudan çağıramazlar. Sık kullanılan mesaj tipi değildir.	
Senkron (Syncronous)/ Çağrı yapan(Call) Mesaj Tipi: Zaman uyumlu mesaj. Nesne mesajı alıcı nesneye gönderir ve onun işlemini bitirmesini bekler, bu durumda senkron mesaj tipi kullanılır. Nesne tabanlı programlamada çağırılan birçok method senkron çalıştığından en çok kullanılan mesaj tipidir.	
Asenkron Mesaj Tipi : Senkron mesajların tersine, asenkron mesajlar nesneye mesaj gönderdikten sonra cevap beklemeden işleme devam etmesinin gösteriminde kullanılır. Genellikle komut zincirlerinde kullanılır.	
Dönüş(Return) Mesaj Tipi: Senkron mesajlarda alıcı nesnenin işleminin bitimini, gönderen nesneye bildirmesinde kullanılır.	

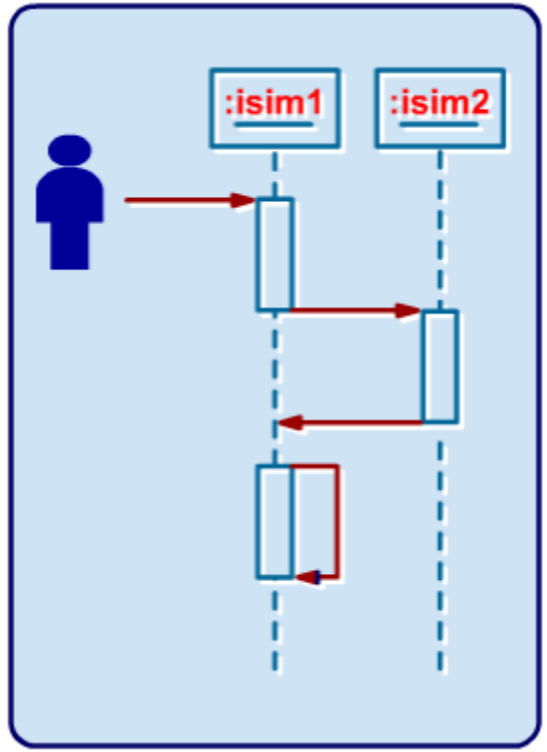
Mesajlar (Messages)

- **(a)** diyagramı zaman uyumlu akışa örnektir: Nesneler arasında gönderilen mesajın geri gönderdiği değer beklenir.
- **b)** diyagramında düz akış örneklenmiştir: Oklar kontrolü bir nesneden diğerine geçirir.
- **(c)** diyagramı ise zaman uyumsuz akışa örnektir: Hata yakalıyıcı nesnesi alarm nesnesine mesaj iletir ama cevabı beklemeden uygulama nesnesine hata değerini gönderir.



Zaman (Time)

- Ardışıl Diyagramlar zamanı dikey doğrultuda gösterir. Zaman en tepede başlar ve aşağı doğru yayılır. Tepeye daha yakın olan mesaj aşağıya daha yakın olan mesajdan zaman olarak daha önce gerçekleşir.
- Ardışıl Diyagramlar'da, aktör figürü tipik ardışıllığı başlatıyor olarak görünse de, gerçekte aktör figürü Ardışıl Diyagram sembolleri arasında yer almaz.



JAVADOC

Java Dökümantasyonu (Javadoc)

- Java dili üç tür yorumu destekler

Sr.No.	Comment & Description
1	**/* text */** The compiler ignores everything from /* to */.
2	**//text** The compiler ignores everything from // to the end of the line.
3	**/** documentation */** This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation.

Java Dökümantasyonu (Javadoc)

- Belge yorumlarımız, resmi *Java Platform API Spesifikasyonunu* tanımlar.

(https://www.oracle.com/tr/technical-resources/articles/java/javadoc-tool.html)

- Java dili programların belgelenmesi için Javadoc isimli bir araç sunmaktadır.
- Javadoc, JDK ile birlikte gelen bir araçtır
- Bu araç Java programlarının paketlerinin otomatik olarak HTML formatında belgelenmesini sağlamaktadır.
- Belgeleme sonucu programların uygulama programlama arayüzleri (API) HTML sayfaları olarak tek tek üretilir.
- Bu sayfalara bir internet tarayıcısı ile bakılarak programların arayüzleri, metotları, özellikleri, programlar arası ilişkiler tarayıcı üzerinden gezilerek öğrenilebilir.

Java Dökümantasyonu (Javadoc)

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{@code}	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{@code text}
{@docRoot}	Represents the relative path to the generated document's root directory from any generated page.	{@docRoot}
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
@exception	Adds a **Throws** subheading to the generated documentation, with the classname and description text.	@exception class-name description
{@inheritDoc}	Inherits a comment from the **nearest** inheritable class or implementable interface.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class.	{@link package.class#member label}
{@linkplain}	Identical to {@link}, except the link's label is displayed in plain text than code font.	{@linkplain package.class#member label}

Java Dökümantasyonu (Javadoc)

Tag	Description	Syntax
@param	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@serial	Used in the doc comment for a default serializable field.	@serial field-description | include | exclude
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods.	@serialData data-description
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description
{@value}	When {@value} is used in the doc comment of a static field, it displays the value of that constant.	{@value package.class#field}
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text