

# NESNEYE DAYALI YAZILIM MÜHENDİSLİĞİ

Binnur Kurt



*binnur.kurt@ieee.org*

İSTANBUL TEKNİK ÜNİVERSİTESİ  
*Bilgisayar Mühendisliği Bölümü*



Version 1.0.6

## AMAÇ

- ▶ Yazılım Mühendisliğinin Hedefi : Kaliteli Yazılım
- ▶ Nesneye Dayalı (ND) Yaklaşım ile Nasıl Ulaşılır?
- ▶ Yazılım Yaşam Çevrimi:
  - İsteklerin Çözümlemesi, Sistem Çözümleme, Tasarım, Kodlama, Tümlleştirme, Sınama, Bakım
  - ND Çözümleme, ND Tasarım, ND Sınama
  - Standartlar: RUP ve UML 
- ▶ Bilgisayar Destekli Yazılım Mühendisliği – CASE
  - Rational Software – Rational Rose Enterprise Edition 

*Tell me and I forget. Teach me and I remember.  
Involve me and I learn.*

—Benjamin Franklin

## Kitaplar

1. “Managing Software Requirements: A Use Case Approach,” Dean Leffingwell, Don Widrig Addison Wesley, 2003, ISBN: 0-321-12247-X
2. “UML Distilled - A Brief Guide to the Standard Object Modeling Language,” Martin Fowler, Kendall Scott, Addison Wesley, 1999, ISBN: 0-201-65783-X
3. “UML Reference Manual,” James Rumbaugh, Ivar Jacobson, Grady Booch, Addison Wesley, 1999, ISBN: 0-201-30998-X
4. “Visual Modeling with Rational Rose 2000 and UML,” Terry Quatrani, Addison Wesley, 1999, ISBN: 0-201-69961-3
5. “Writing Effective Use Cases,” A.Cockburn, Addison Wesley
6. “Rational Unified Process Made Easy: A Practitioner's Guide to the RUP,” Per Kroll, Philippe Kruchten, Addison Wesley

## İÇERİK

1. Bütünleştirilmiş Modelleme Dili:  
UML—Unified Modeling Language
2. Bütünleştirilmiş Süreç Modeli:  
RUP—Rational Unified Process
3. Rational Rose Enterprise’ a Genel Bir Bakış
4. Uygulama: Royal Service Station
5. Tasarım Kalıpları

# 1

## UML’E GİRİŞ

## UML NEDİR?

- ▶ UML yazılım sisteminin önemli bileşenlerini tanımlamayı, tasarlamayı ve dokümantasyonunu sağlayan grafiksel bir modelleme dilidir
- ▶ Yazılım geliştirme sürecindeki tüm katılımcıların (kullanıcı, iş çözümleyici, sistem çözümleyici, tasarımcı, programcı,...) gözüyle modellenmesine olanak sağlar,
- ▶ UML gösterimi nesneye dayalı yazılım mühendisliğine dayanır.

## Ortak Bir Dil

$$\int_0^{\infty} \frac{1}{x^2} dx$$

- ▶ Tüm mühendisler  $\int$  simgesinin anlamını bilir
- ▶ Simge basit olsa da arkasındaki anlam karmaşık ve derindir!
- ▶  $\infty$ : Ters sekiz?

$$\overset{c}{\text{—}||\text{—}}$$

- ▶ Kapasite?

## Yöntem Savaşı

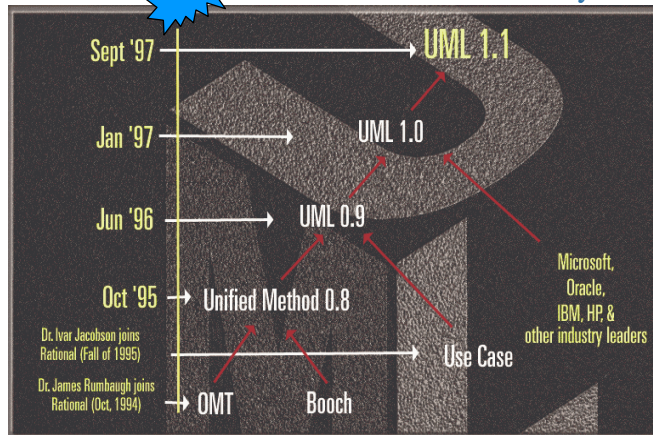
- ▶ Ekim 1995: Herkese açık ilk taslak (Sürüm 0.8)
- ▶ Temmuz 1997: Sürüm 1 OMG'ye standart olarak sunuldu
- ▶ Kasım 1997: UML OMG tarafından standart olarak kabul edildi.
- ▶ Güncel sürüm: UML 2



Three Amigos:  
Booch, Rumbaugh, Jacobson

## UML

Kasım '97 UML OMG tarafından onaylandı



## UML'in Kazanımları

- Yazılım sistemi herhangi bir kod yazmadan önce profesyonelce tasarlanır ve dokümantasyonu yapılır
- Yeniden kullanılabilir kod parçaları kolaylıkla ayırt edilir ve en yüksek verimle kodlanır
- Daha düşük geliştirme maliyeti
- Tasarımdaki mantıksal boşluklar tasarım çizimlerinde kolaylıkla saptanabilir

## UML'in Kazanımları—2

- Daha az sürpriz – yazılımlar beklendiğimiz şekilde davranırlar
- Overall design will dictate the way software is developed – tüm tasarım kararları kod yazmadan verilir
- UML “resmin tamamını” görmemizi sağlar
- More memory and processor efficient code is developed
- Sistemde değişiklik yapmayı kolaylaştırır

## UML'in Kazanımları—3

- Less 're/learning' of the system takes place
- Diagrams will quickly get any other developer up to speed
- Programcılar arasında daha etkin bir iletişime olanak sağlar

## UML'in Geliştirme Sürecindeki Yeri

- ▶ Three Amigos UML'i geliştirirken, dilin belirli bir süreç modeline bağlı olmamasına özen gösterdiler.
- ▶ Farklı süreç modelleri: RUP, Shlaer-Mellor, CRC ve Extreme Programming kullanılabilir.
  - RUP : Three Amigos tarafından geliştirildi.
  - Derste RUP'u inceleyeceğiz
- ▶ Bu nedenle UML farklı yazılım projelerine cevap verebilecek genelliğe sahiptir:  
E-Ticaret Uygulaması × Askeri Uygulamalar  
Dokümantasyon, Sınama, Performans
- ▶ UML nasıl yazılım geliştirileceğini söylemez!

## Hatırlatma

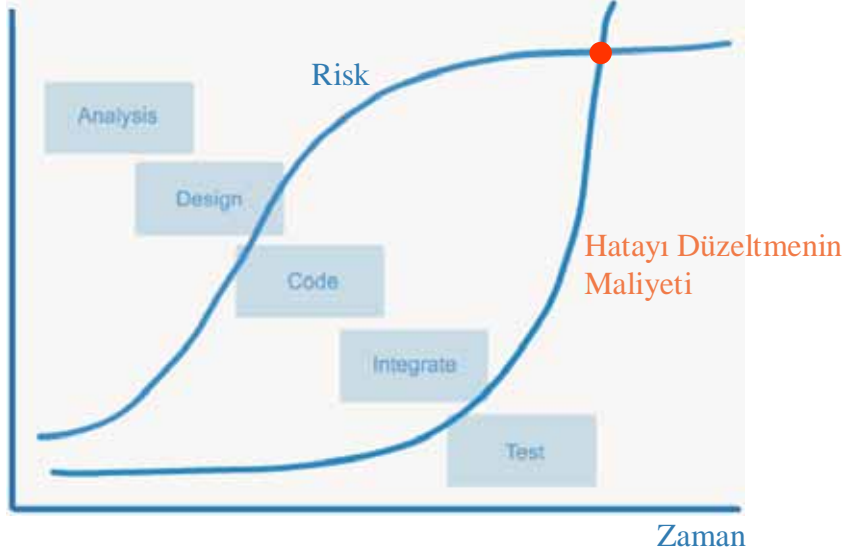
- ▶ “Süreç Yönetimi” konusuna kısa bir geri dönüş yapalım:
  - Şelale Modeli
  - V-Modeli
  - Spiral Model
  - Artımsal ve Yinelemeli Modeller

## Şelale Modeli





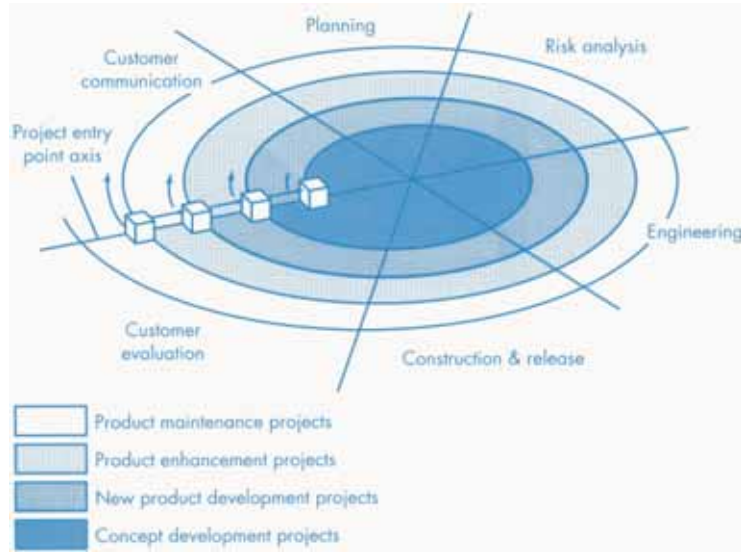
## Şelale Modelin Yitirimleri



Nesneye Dayalı Yazılım Mühendisliği

17

## Spiral Model



Nesneye Dayalı Yazılım Mühendisliği

18

## Spiral Modelin Kazanımları

- ▶ Takım yazılım yaşam çevriminin tüm aşamalarına katılır,
- ▶ Kısa sürede ve düzgün aralıklarla geri besleme alınır,
- ▶ Riskli bileşenler önceden kestirilebilir ve gerçekleştirilebilir,
- ▶ İşin ölçeği ve karmaşıklığı önceden keşfedilebilir,
- ▶ Çalışan bir sürümün varlığı takımın moralini yüksek tutar,
- ▶ Projenin durumu daha kesin olarak değerlendirilebilir.

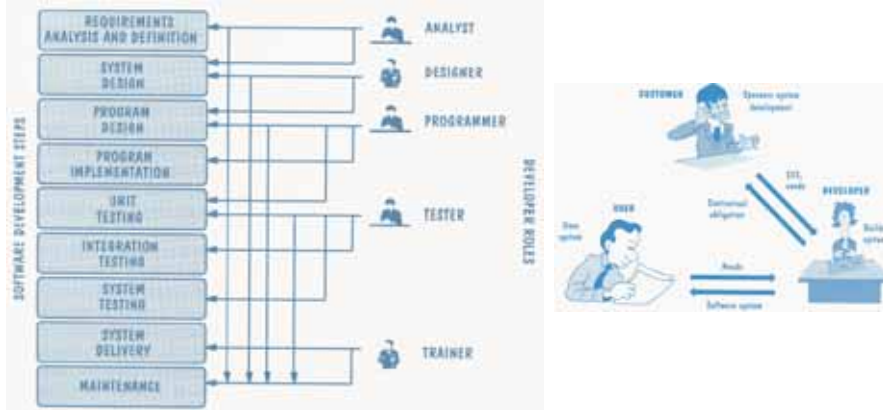
## Rational Unified Process

- ▶ RUP yinelemeli, artımsal, mimari merkezli, risk güdümlü, kullanım senaryolarına dayalı bir yazılım geliştirme süreci modelidir.
- ▶ RUP iyi tanımlanmış ve yapılandırılmış bir yazılım sürecidir: **Kimin Neden** sorumlu olduğu, işlerin **Nasıl** ve **Ne Zaman** yapılacağı açıkça tanımlanır.

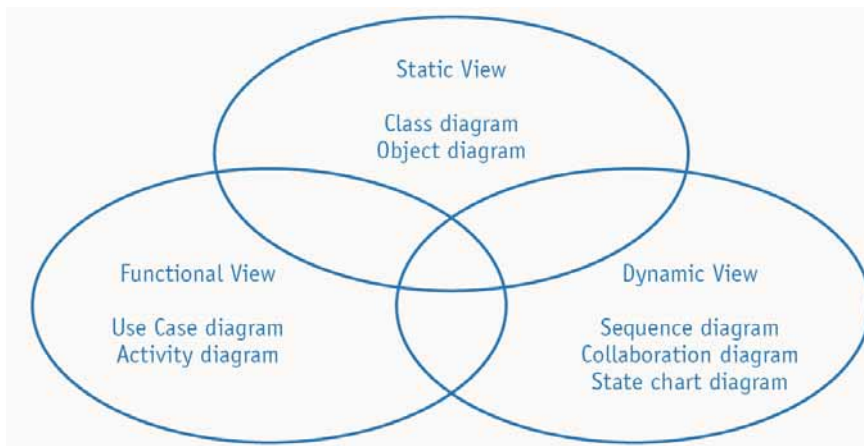


## UML'e Genel Bir Bakış

- UML'de çok sayıda farklı şema var: Kullanım Senaryosu Şeması, Sınıf Şeması, İş Birliği Şeması, Ardışıl Şema,...
- Amaç isteme ve projeye farklı bakış açılarından bakmaktır.

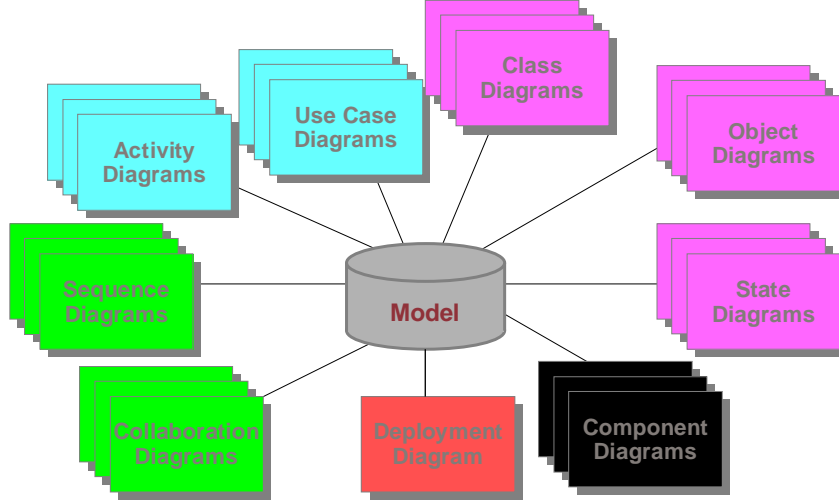


## UML'e Genel Bir Bakış



## UML'e Genel Bir Bakış

Unified Modeling Language 1

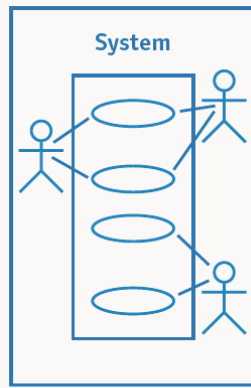


Nesneye Dayalı Yazılım Mühendisliği

23

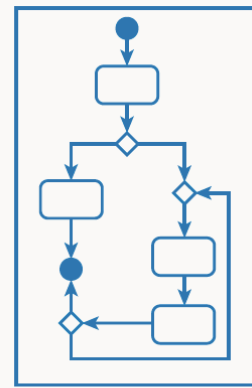
## İşlevsel Görünüm

Unified Modeling Language 1



Name  
Assumptions  
Pre-conditions  
Dialog  
Post-conditions  
Exceptions  
Future Enhancements  
Open Issues

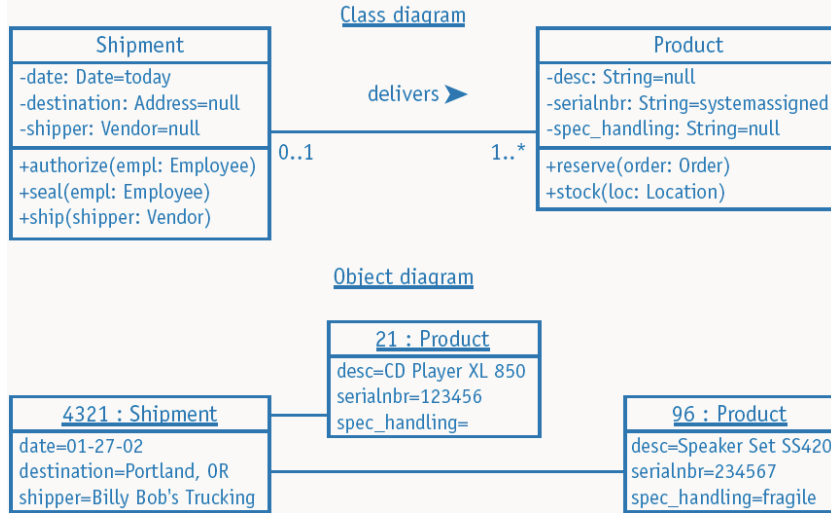
Use Case Narrative



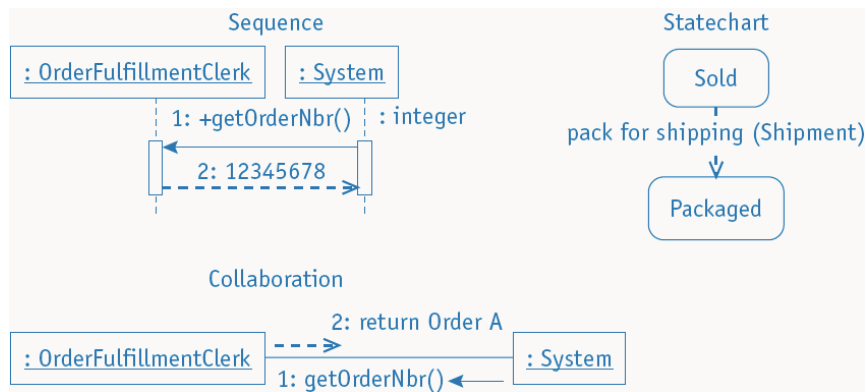
Nesneye Dayalı Yazılım Mühendisliği

24

## Durağan Görünüm

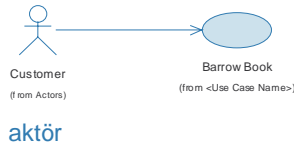


## Devingen Görünüm

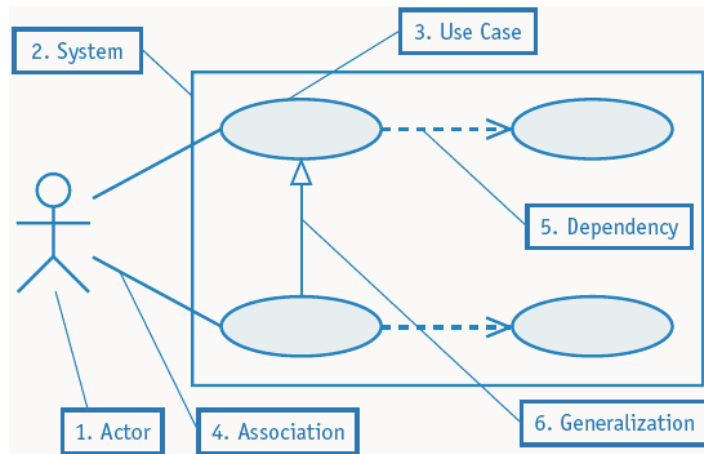


## Kullanım Senaryosu Şeması

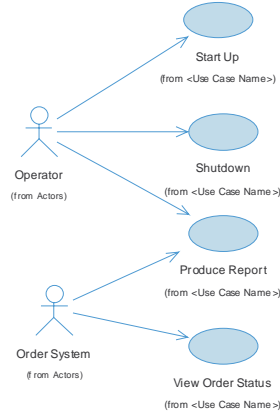
- Kullanım senaryosu şeması, tasarlanacak sisteme kullanıcı gözüyle bakıldığındaki davranışını tanımlar.
- Şemanın anlaşılması oldukça kolaydır.
- [Bu nedenle] Hem geliştirme ekibinin hem de müşterinin ortak olarak çalışabileceği bir şemadır.
- Analizde yardımcı olur, tasarımda isteklerin anlaşılmasında yardımcı olur.



## Kullanım Senaryosu Şemasının Bileşenleri



## Kullanım Senaryosu Şeması



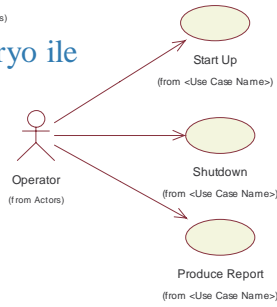
- Bir aktör birden fazla kullanım senaryosunda yer alabilir
- Aynı senaryoda birden fazla aktör olabilir.

## Aktör

- Aktör eylemi başlatan nesnedir.
- Aktör nesnesi mutlaka bir kişi olmak zorunda değil.
- Soyut bir nesne olabilir: zaman, tarih, ...
- Aktör sistemin dışından bir nesne olabilir
- Kullanılan Simge:



- Her aktör en az bir senaryo ile ilişkilendirilmelidir:

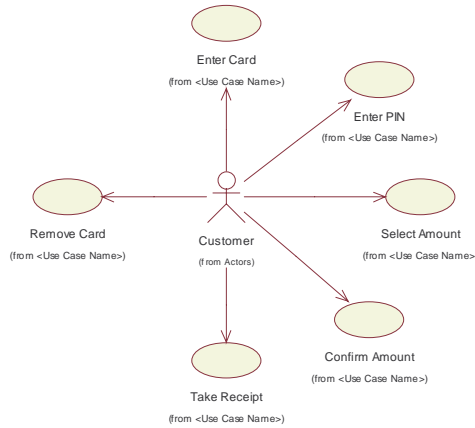


## Kullanım Senaryolarının Sağladığı Kazanımlar

- ▶ Sistemin erimini, sınırlarını belirler.
- ▶ Böylelikle geliştirilecek sistemin boyutunu ve karmaşıklığını kafamızda daha rahat canlandırabiliriz.
- ▶ Kullanım senaryoları isteklerin çözümlenmesine çok benzemektedir: daha nettir ve tamdır.
- ▶ Basit oluşu müşteri ile geliştirme ekibi arasında iletişime olanak tanır.
- ▶ Geliştirme aşaması için temel oluşturur.
- ▶ Sistem testi için temel oluşturur.
- ▶ Kullanıcı klavuzu hazırlamaya yardımcı olur.

## Çözünürlük Ne Olmalı?

- ▶ Kullanım senaryosunun kullanımına ilişkin bir örnek:  
ATM cihazından para çekmek



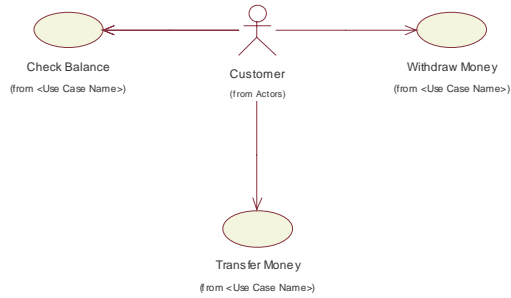


## Çözüm

- Kullanım senaryosu, aktör için bir amacı yerine getirmelidir.



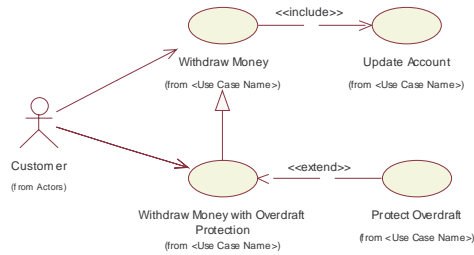
Amaç: para çekmek



## Kullanım Senaryoları Arası İlişkiler

- ▶ Kullanım senaryoları arasında üç tür ilişki bulunabilir
- ▶ İçerme «include»  
Bir senaryo grubu içinde kullanılan başka bir senaryo grubudur
- ▶ Genişletme «extend»  
Senaryo grupları doğal akışa göre verilirler. Bu akıştan olan sapmalar genişletme ilişkisi ile ana senaryodan olan sapma gösterilir.
- ▶ Genelleştirme  
Sınıflar arasındaki türeme ilişkisine benzer. Genel bir senaryo grubundan özel bir senaryo grubu türetilir.

## Örnek Kullanım Senaryosu



## Kullanım Senaryosu Anlatımı

1. Müşteri kartını ATM cihazına tanıtır. Sistem karttaki bilgileri okur ve doğrular.
- 2.Sistem PIN kodunu sorar. Müşteri PIN kodunu girer. Sistemi PIN kodunu doğrular.
- 3.Sistem hangi tür işlem yapmak istediğini sorar. Müşteri “Para Çek“i seçer
- 4.Sistem çekilecek miktarı sorar. Müşteri miktarı girer.
- 5.Sistem hesap türünü sorar. Müşteri hesap türünü girer.
- 6.Sistem ATM ağını kullanarak kimlik, PIN kodu ve çekilen miktarı doğrular.
- 7.Sistem makbuz istenip istenmediğini sorar. Bu işlem cihazda kağıt varsa yürütülür.
- 8.Sistem müşteriden kartı yuvasından almasını ister. Müşteri kartını alır. (Bu istek müşterinin kartı cihazda unutmadığından emin olmak için güvenlik amacıyla yapılır.)
- 9.Sistem istenilen miktar banknotu verir .
- 10.Eğer müşteri istemişse sistem kağıt makbuzu verir. Senaryo sona erer.

## Kullanım Senaryosu Anlatımı

- ▶ Standart bir format yok.
- ▶ Her firma kendine uygun bir format belirleyebilir.

- ✓ Senaryo: **Senaryo adı**
- ✓ Özet tanıtım: **Senaryonun kısa bir tanımı**
- ✓ Ön koşullar: **Senaryonun başlaması için sağlanması gereken koşullar**
- ✓ Sonuç koşulları: **Senaryonun sonunda neler olduğu tanımlanır**
- ✓ Ana Akış: **Sistem için olağan senaryo durumunda gerçekleşen etkileşimlerin bir listesi verilir.**
- ✓ Alternatif Akış: **Olası alternatif etkileşimlerin tanımlanması**
- ✓ Sıradışı Akış: **Beklenmeyen yada öngörülmeyen olayların gerçekleştiği senaryolar tanımlanır**

## Kullanım Senaryolarının Yazılması

- ▶ Aktörlerin Belirlenmesi:
  - Sistemin temel işlevlerini kim kullanacak?
  - Sistemin bakımını ve işletimini kim yapacak?
  - Sistem hangi cihazları kullanacak?
  - Diğer hangi sistemlerle etkileşimde bulunacak?
  - Sistemin çıkışlarını kimleri ilgilendirir?

## Sistem Davranışının Belirlenmesi

- ▶ Aktörlerden yararlanarak sistem davranışının belirlenmesi
  - Aktörlerin temel işlevi nedir?
  - Aktör sistem bilgilerine erişmeli mi?
  - Durum değişiklikleri aktöre bildirilecek mi?
  - Aktör hangi işlevlere ihtiyaç duyar?
- ▶ Bazı davranışlar aktörlerden yola çıkarak belirlenemeyebilir.  
Bu durumda aşağıdaki soruları da sormak uygun olur:
  - Sistemin gerek duyduğu giriş ve çıkış nedir?
  - Sistem dış olaylardan etkilenir?
  - Şu andaki sistemin eksiklikleri ve problemleri nelerdir?
  - Periyodik olarak gerçekleştirilen işlemler var mı?

## Kullanım Senaryolarının Saptanması

- ▶ Olası sistem kullanıcıları ile görüşme yapmak
- ▶ Joint Requirements Planning Workshop (JRP)

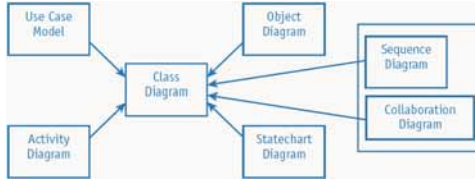
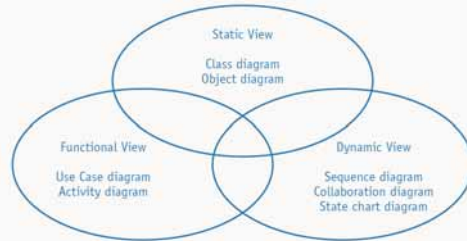


- Beyinfırtınası: olası tüm aktörler saptanır
- Beyinfırtınası: olası tüm senaryolar saptanır
- Her senaryo için Kullanım Senaryo Anlatımı kağıda aktarılır ve doğrulanır



- Model CASE Tool kullanılarak bilgisayarda oluşturulur

## Sınıf Şeması

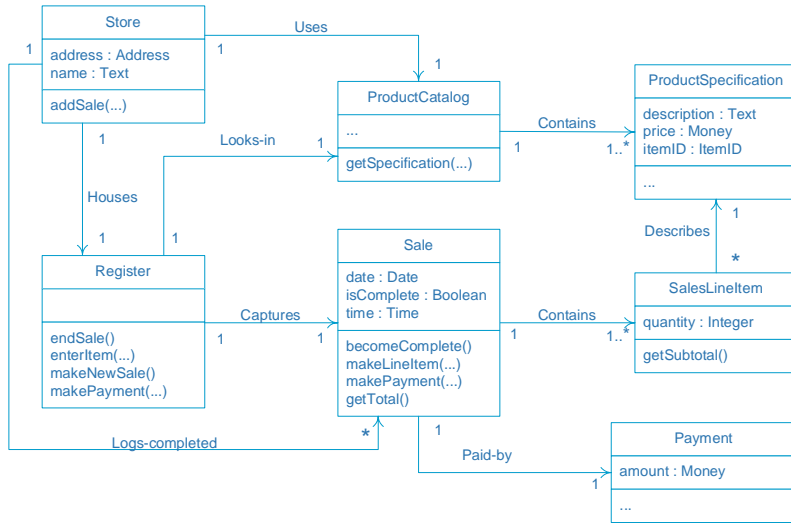


- ▶ Amaç çözülmek istenen probleme ilişkin dünyanın doğru, özlu, anlaşılır ve sınanabilir bir modelini oluşturmak.
- ▶ Modelde gerçek dünyayı oluşturan kavramsal sınıflar ve nesnelere yer alır.
- ▶ Model UML ile görsel biçime getirilir.

## Sınıf Şemasının Bileşenleri

- Nitelikler : Sınıfların nitelikleri
- İşlemler
- Stereotypes
- Özellikler: Sınıf tanımlamalarının durumunu ve bakımını izlemek için bir yöntem
- Bağlantı: Sınıflar arasındaki ilişkiler
- Kalıtım

## Örnek Sınıf Şeması



## Kavramsal Sınıfların Belirlenmesi

- En çok kullanılan iki temel yöntem:
  1. Kavramsal sınıfların kategori listesinden yararlanmak
  2. Kullanım senaryolarındaki isimlerden yararlanmak
- Örnek Kategoriler
  - Fiziksel ve somut nesnelere
  - Yer
  - İşlem
  - Hizmet
  - Olay Roller
  - Başka nesnelere içeren kaplar (container)

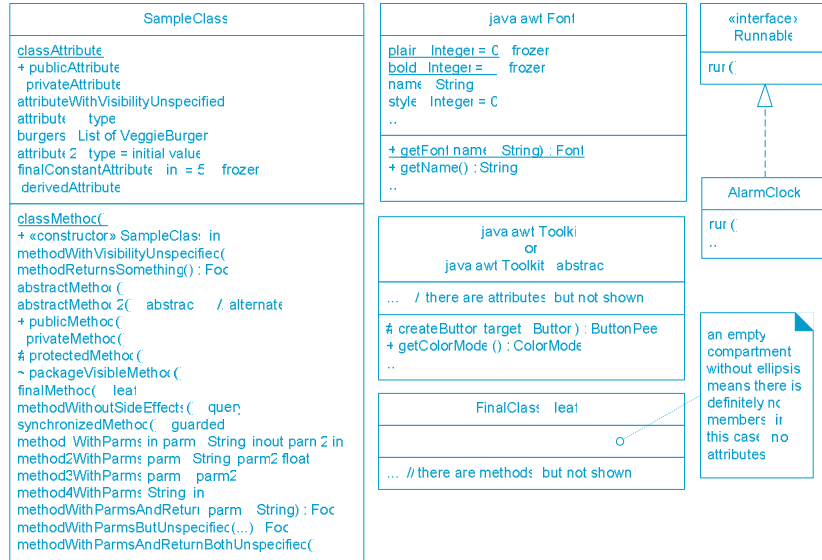
## Örnek: Kullanım Senaryolarından Yararlanmak

1. **Müşteri** kartını **ATM cihazına** tanıtır. Sistem **kart**taki bilgileri okur ve doğrular.
2. Sistem **PIN kodunu** sorar. Müşteri PIN kodunu girer. Sistemi PIN kodunu doğrular.
3. Sistem hangi tür **işlem** yapmak istediğini sorar. Müşteri "Para Çek"i seçer
4. Sistem çekilecek **miktarı** sorar. Müşteri miktarı girer.
5. Sistem **hesap türünü** sorar. Müşteri hesap türünü girer.
6. Sistem ATM ağını kullanarak kimlik, PIN kodu ve çekilen miktarı doğrular.
7. Sistem **makbuz** istenip istenmediğini sorar. Bu işlem cihazda **kağıt** varsa yürütülür.
8. Sistem müşteriden kartı yuvasından almasını ister. Müşteri kartını alır. (Bu istek müşterinin kartı cihazda unutmadağından emin olmak için güvenlik amacıyla yapılır.)
9. Sistem istenilen miktar banknotu verir .
10. Eğer müşteri istemişse sistem kağıt makbuzu verir. Senaryo sona erer.

## Gereksiz Sınıfların Elenmesi

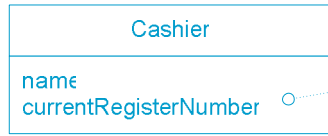
- Artık Sınıflar (Redundant Classes): Aynı unsuru ifade eden iki sınıftan daha tanımlayıcı olan alınır. Kişi—Müşteri: müşteri
- İlgisiz Sınıflar (Irrelevant Classes): Problemin çözümü ile ilgili olmayan yada çözümlemenin o iterasyonunda gerekli olmayan sınıflar silinir.
- Belirsiz Sınıflar (Vague Classes): Sınırları iyi çizilmemiş, fazla genel tanımlı olan sınıflar silinir.
- Nitelikler (Attributes): Nitelikler de isimler ile ifade edildiğinden sınıflar ile karıştırılabilir. Kendi başına varlıkları anlamlı olmayan sadece başka sınıfların niteliklerini oluşturan unsurlar olası sınıflar listesinden silinir.
- İşlemler: Sadece başka nesnelere üzerinde uygulanan işlemler sınıf olamaz. Kendi nitelikleri olan ve başka olaylardan etkilenen işlemler sınıftır.
- Roller: Sınıflar arasındaki ilişkiyi ifade eden roller sınıf olamaz

## Kavramsal Sınıfların Nitelikleri





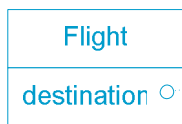
## Betimleme Sınıflarına İhtiyaç Duyulması



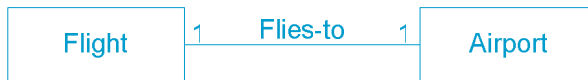
a "simple" attribute but being used as a foreign key to relate to another object



## Örnek-2



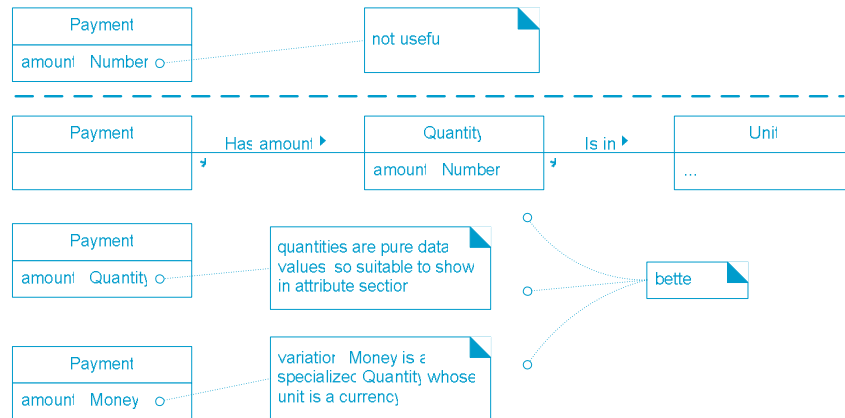
destination is a complex concept



## Örnek-3

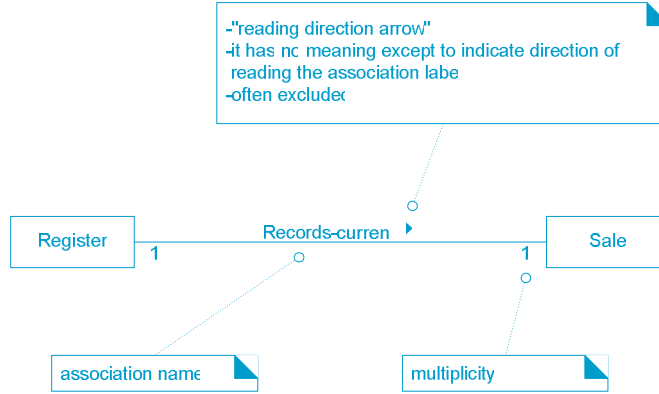


## Örnek-4



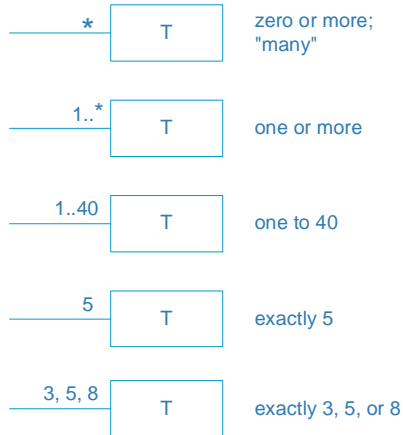
## Kavramsal Sınıflar Arasındaki Bağlantıların Belirlenmesi

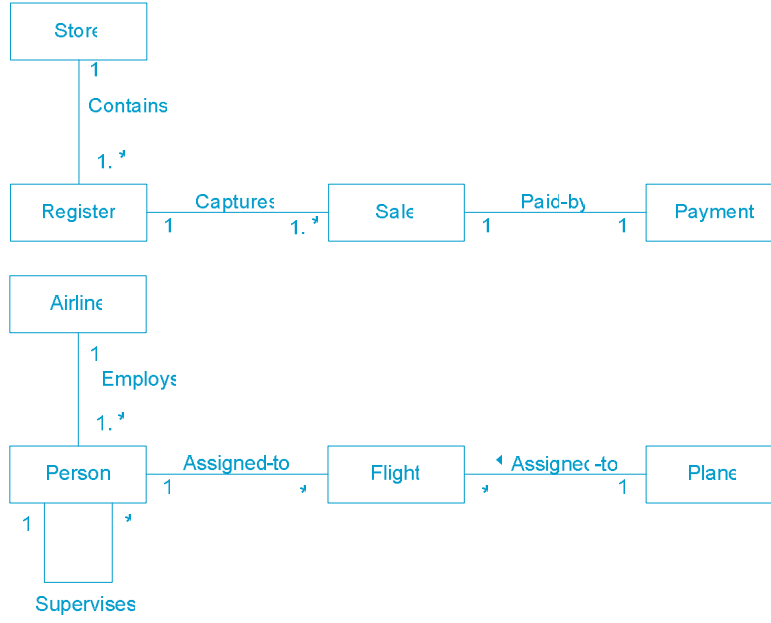
- Uygulama domeninin modeli oluşturulurken ilk aşamada kavramsal sınıflar bulunur.
- İkinci aşamada ise bu sınıflar arasındaki bağlantılar belirlenir.



## Çoğullama Sayısı

- Çoğullama sayısı o sınıftan bir nesnenin kaç tane nesne ile geçerli olarak ilişkilendirilebileceğini gösterir.



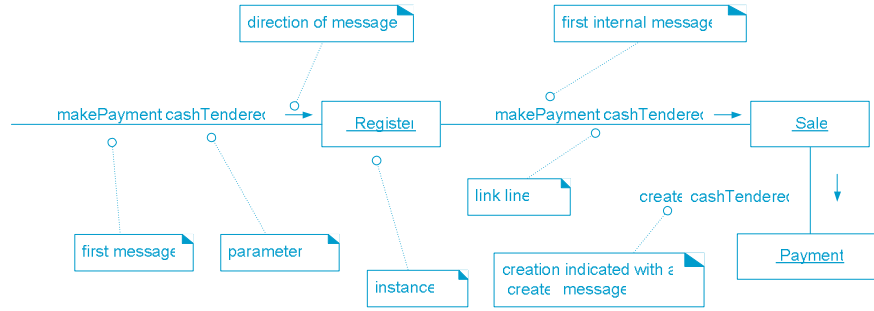


## Tasarım Modelinin Oluşturulması

- Bu aşamada, nesneye dayalı yöntemle göre problemin mantıksal çözümü oluşturulur.
- Tasarım modelinde yazılım sınıfları ve aralarındaki işbirliği (etkileşim) belirlenir.
- Bu modelin en önemli kısmını nesnelere arası etkileşimi gösteren etkileşim şemalarının (interaction diagram) çizilmesi oluşturur.
- Etkileşim şemaları ile birlikte yazılım sınıflarını gösteren sınıf şemaları da çizilir.

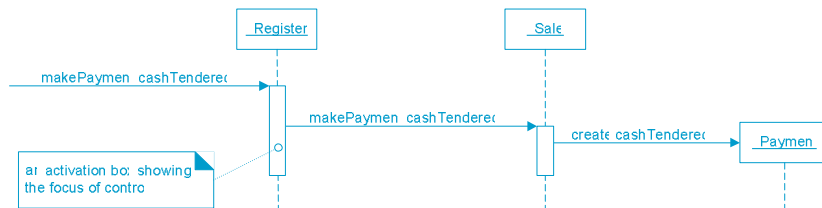
## Etkileşim Şemaları

- UML’de iki tür etkileşim şeması vardır:  
1. İşbirliği Şeması

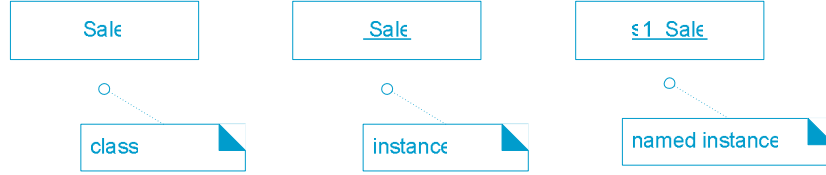


## Etkileşim Şemaları

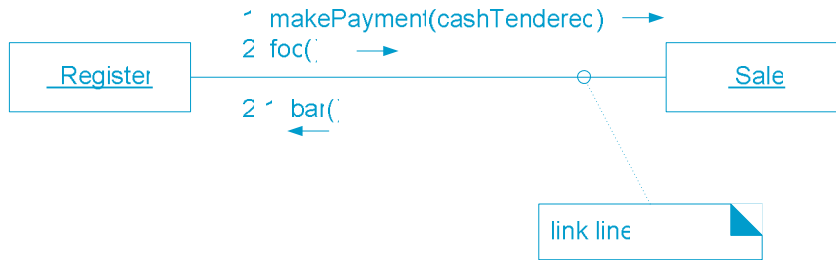
- 2. Ardışık Şema



## Sınıf ve Nesne Gösterimi

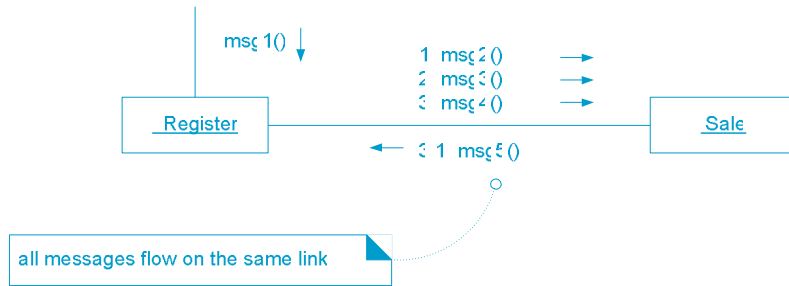


## İşbirliği Şeması Notasyonu



## İşbirliği Şeması Notasyonu

Unified Modeling Language 1

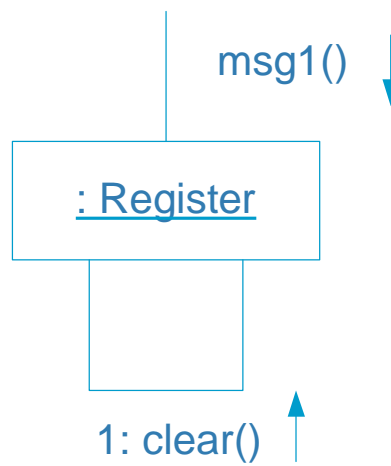


Nesneye Dayalı Yazılım Mühendisliği

61

## İşbirliği Şeması Notasyonu

Unified Modeling Language 1



Nesneye Dayalı Yazılım Mühendisliği

62

## Nesne Yaratmak

Unified Modeling Language 1

create message with optional initializing parameters. This will normally be interpreted as a constructor call

if an unobvious creation message name is used the message may be stereotyped for clarity

Nesneye Dayalı Yazılım Mühendisliği

63

## Mesajları Numaralamak—1

Unified Modeling Language 1

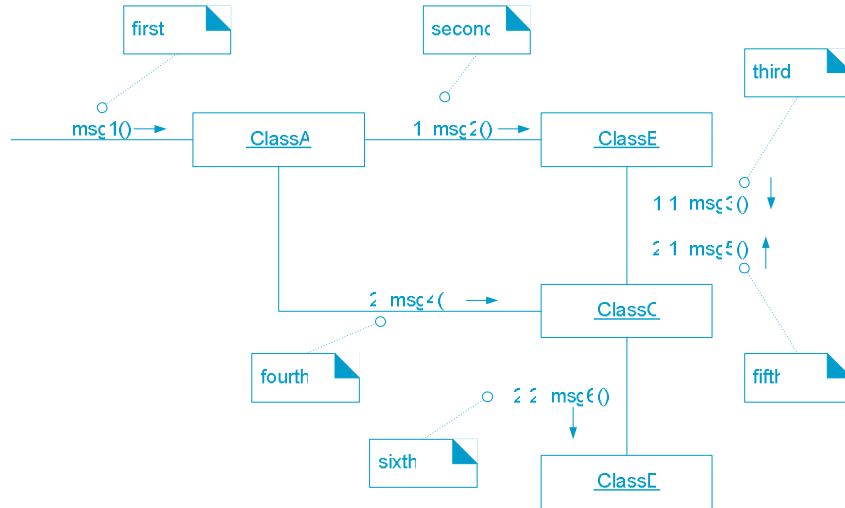
not numbered

Nesneye Dayalı Yazılım Mühendisliği

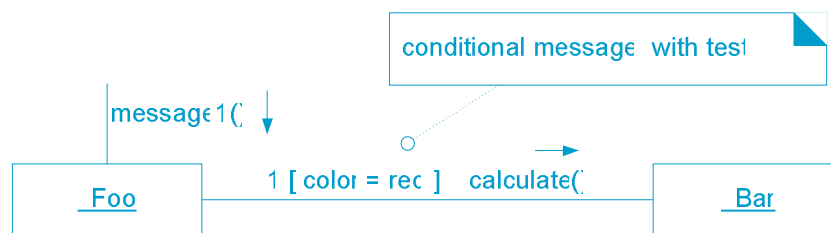
64



## Mesajları Numaralamak—2

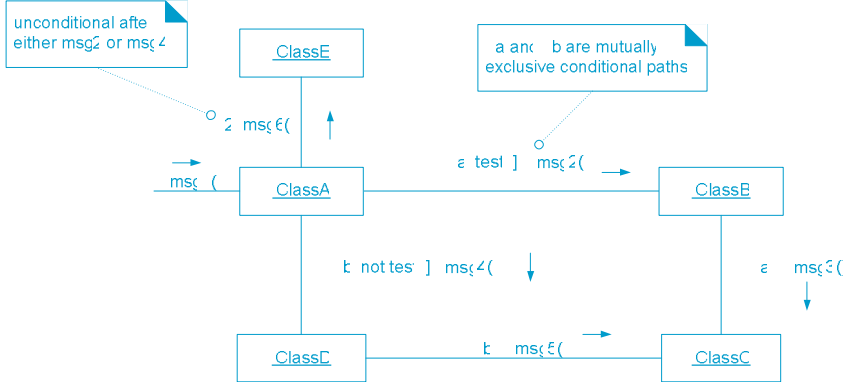


## Koşullu Mesajlar



## Karşılıklı Dışlamalı Mesajlar

Unified Modeling Language 1

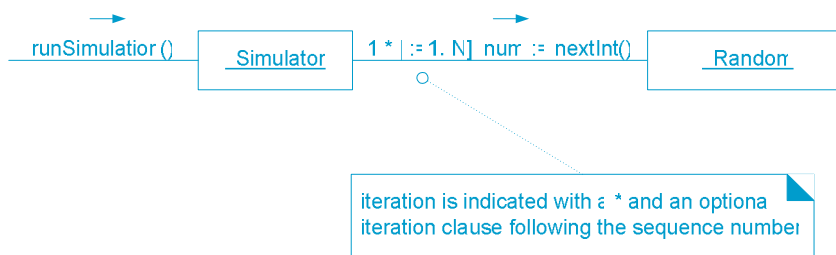


Nesneye Dayalı Yazılım Mühendisliği

67

## Döngü

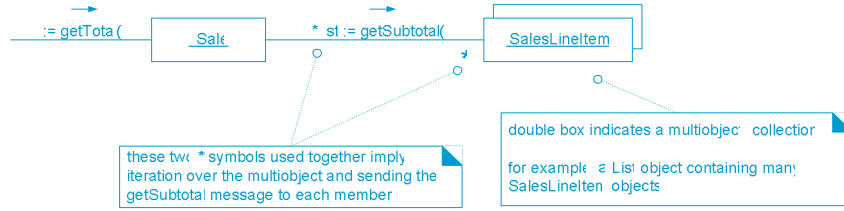
Unified Modeling Language 1



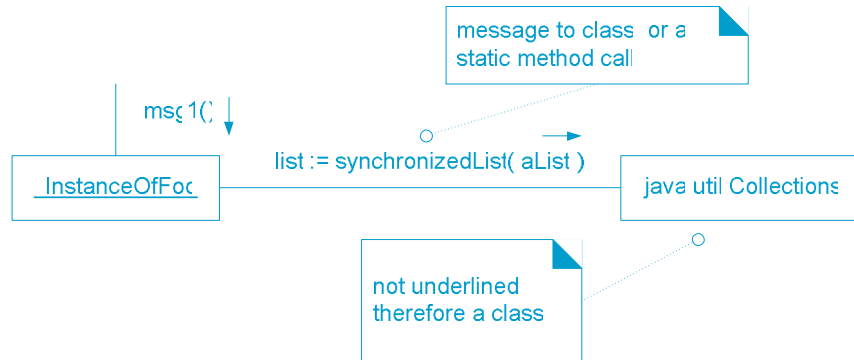
Nesneye Dayalı Yazılım Mühendisliği

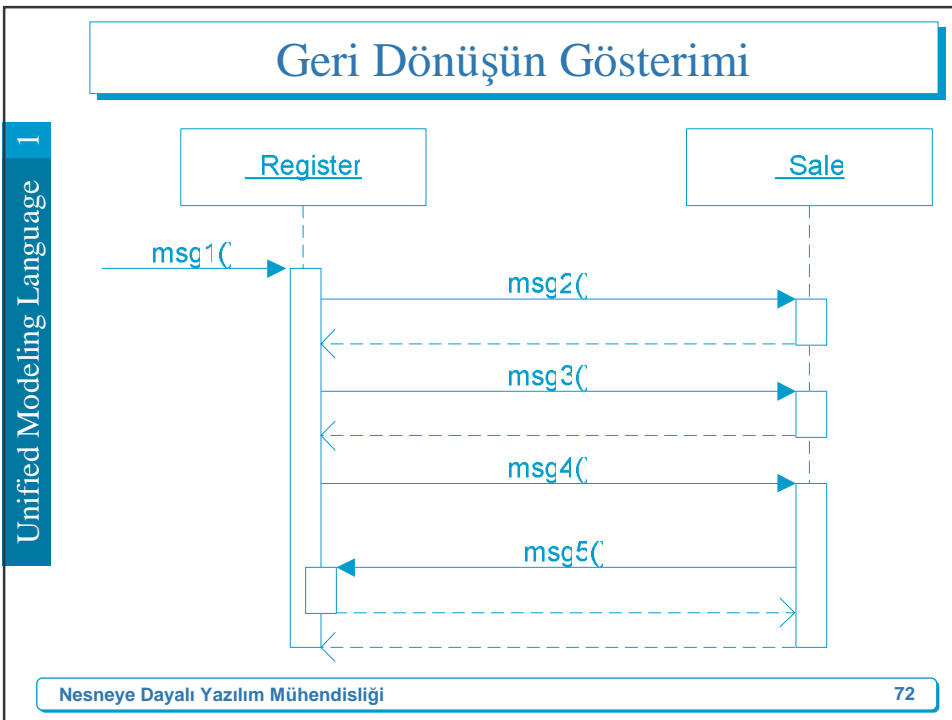
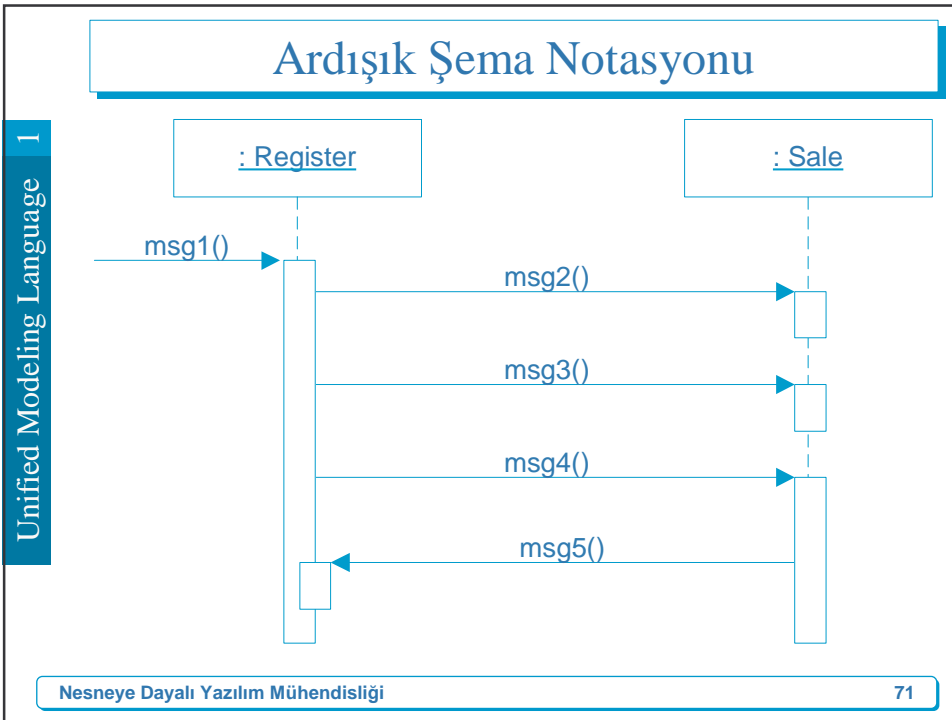
68

## Nesneler Üzerinde Döngü Kurmak

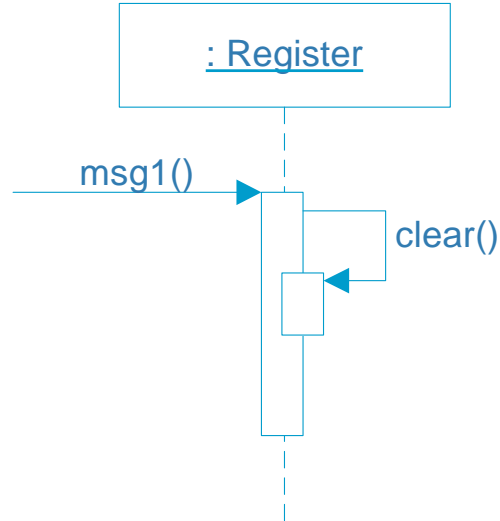


## Sınıfa Mesaj Göndermek

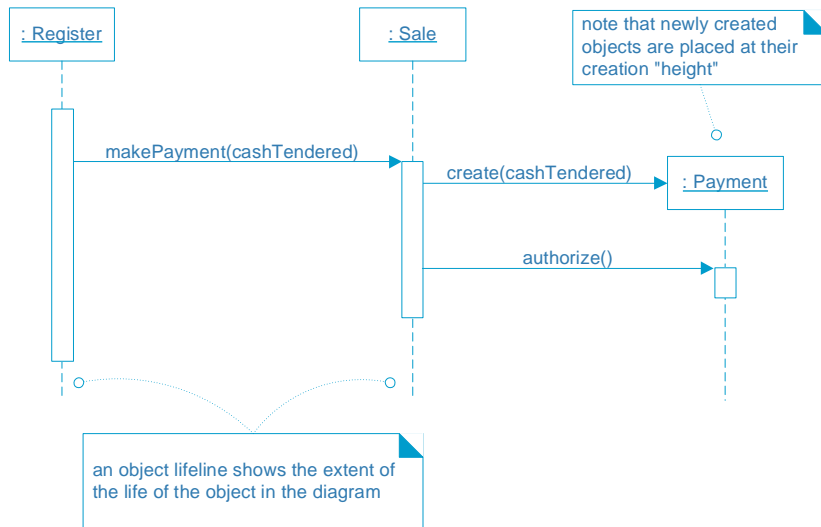




## Kendisine (this) Mesaj Gönderme

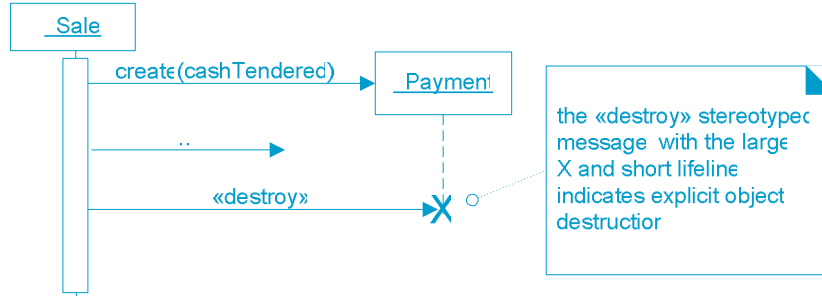


## Nesne Yaratmak



## Nesnenin Yok edilmesi

Unified Modeling Language 1

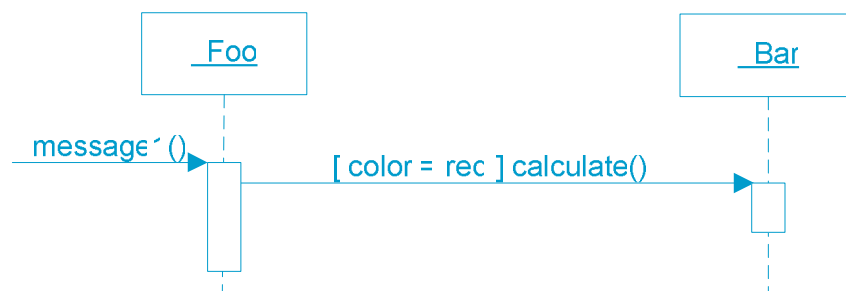


Nesneye Dayalı Yazılım Mühendisliği

75

## Koşullu Mesajlar

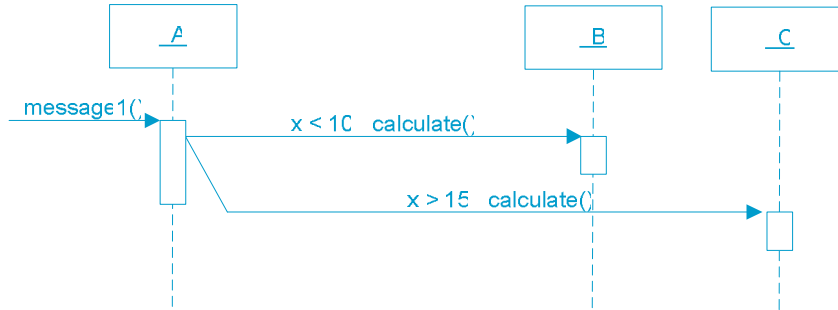
Unified Modeling Language 1



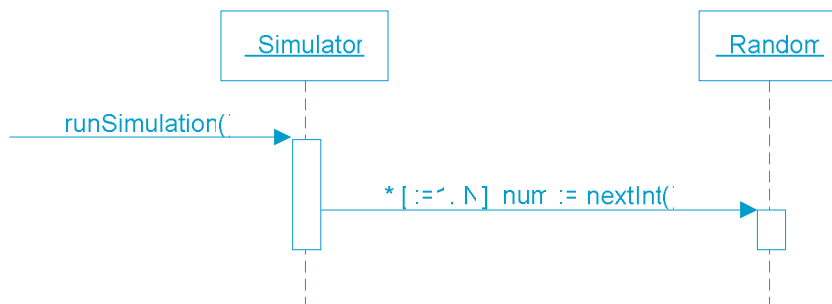
Nesneye Dayalı Yazılım Mühendisliği

76

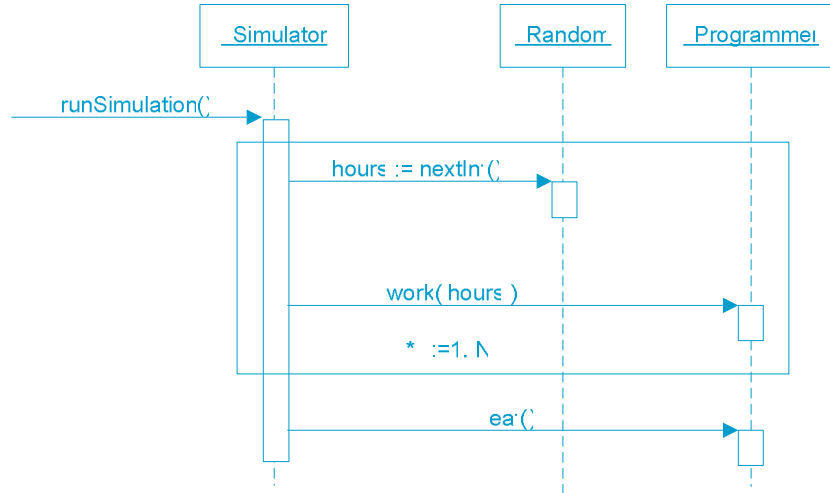
## Karşılıklı Dışlamalı Mesajlar



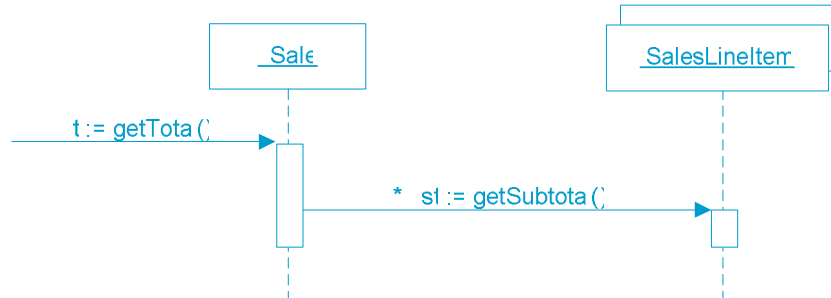
## Döngü



## Bir Dizi Mesajdan Oluşan Döngü

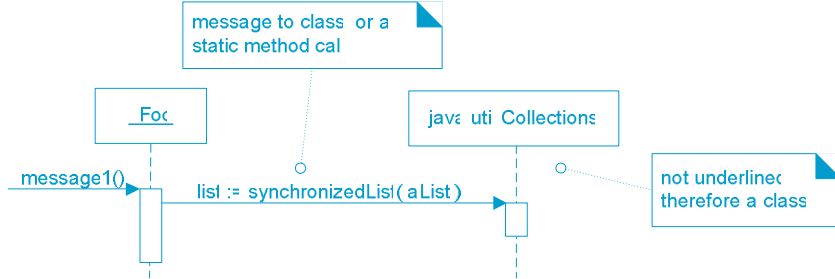


## Nesne Gruplarına Mesajlar





## Sınıf Metodunu Çağırarak



## Sorumlulukların Atanması yolu ile Nesnelerin Tasarımı

- Unified Modeling Language 1
- Nesne Tasarımının Genel İfadesi: İsteklerin çözülmesi, uygulama alanının modelinin kurulmasından sonra, yazılım sınıflarına metodların eklenmesi ve istekleri yerine getirmek üzere nesnelere mesajların belirlenmesidir.
  - Nesnel tasarımın temeli nesnelere sorumlulukların atanmasına dayanır:
    - Bilinmesi Gerekenler:
      - Kendi Özel Verileri
      - İlgili Diğer Nesnelere
      - Hesap yaparak elde edebileceği bilgiler
    - Yapılması Gerekenler:
      - Hesap yapma, nesne yaratma/yoketme
      - Başka nesnelere hareket geçirme
      - Başka nesnelere hareketini denetleme

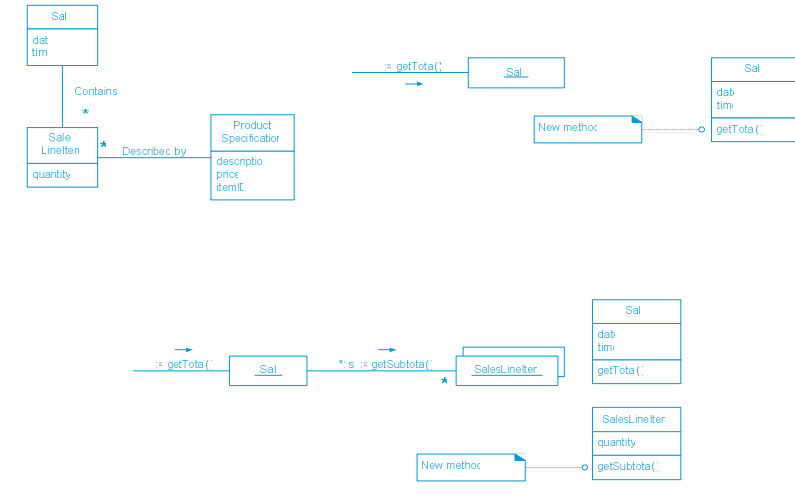
## Kalıplar

- Yazılımcılar deneyimleri sonucunda bir çok problemin çözümünde uygulanabilecek prensipler ve deyimler yaratmışlardır.
- Bu deyim önce internet’te tartışma gruplarında ortaya atıldı
- “Design Patterns, Elements Of Reusable Object Oriented Software” kitabıyla ünlendi. Yazarları: Gamma, Helm, Johnson, Vlissides—Gang of Four
- Bu prensipler belirli yapısal kurallara göre yazılarak yazılım geliştiren kişilere yol göstermek üzere oluşturulmuştur:
  - GRASP:
    - Expert
    - Creator
    - High Cohesion
    - Low Coupling
    - Controller

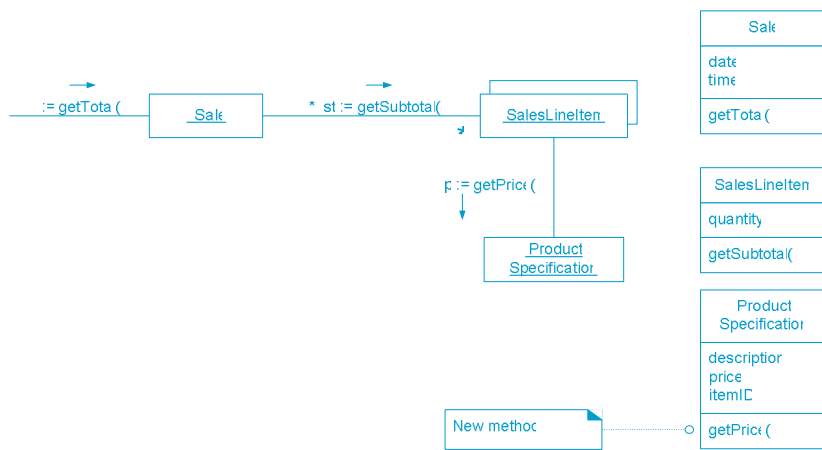
## Expert

- Çözüm: Bir sorumluluğu bilginin uzmanına, onu yerine getirecek veriye sahip olan sınıfa atayın.
- Problem: Nesnelere sorumluluklarını atamanın temel prensibi nedir?

# Örnek



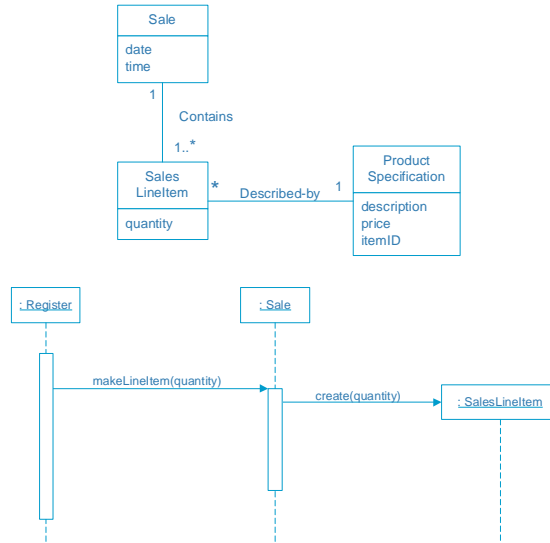
# Örnek



## Creator

- Çözüm: Aşağıdaki koşullardan biri geçerli ise B sınıfına A sınıftan nesne yaratma sorumluluğu atayın:
  - B, A nesnelərini içeriyorsa
  - B, A nesnelərini kaydı tutuyorsa
  - B, A nesnelərini kullanıyorsa
  - A nesnelərini yaratılması aşamasında kullanılacak olan başlangıç verilerine B sahipse
- Problem: Bir sınıftan nesne yaratma sorumluluğu kime ait?

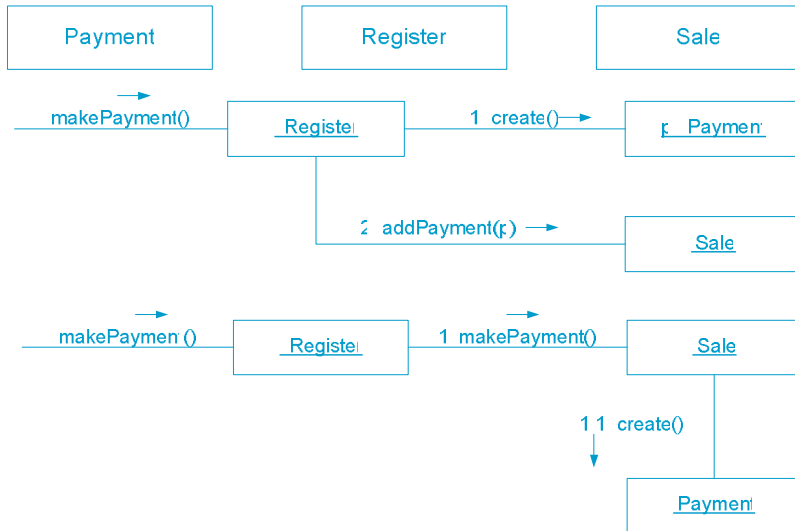
## Örnek



## Az Bağımlılık—Low Coupling

- Çözüm: Sorumlulukları sınıflar arası bağımlılığı az olacak şekilde atayın.
- Problem: Diğer sınıfların değişikliklerinden etkilenmeme, tekrar kullanılabilirlik nasıl sağlanır?

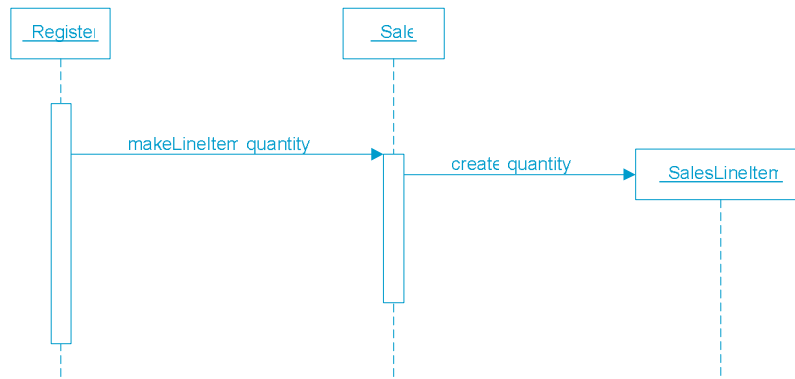
## Örnek



## İyi Uyum—High Cohesion

- Çözüm: Sorumlulukları sınıf içinde iyi bir uyum olacak şekilde atayın.
- Problem: Karmaşıklık nasıl idare edilebilir?
- Eğer bir sınıf birbiri ile ilgili olmayan işler yapıyorsa veya çok fazla iş yapıyorsa sınıfta uyum kötüdür:
  - Anlaşılabilirlik azalır
  - Bakım zorlaşır
  - Tekrar kullanılabilirlik güçleşir
  - Değişikliklerden çok fazla etkilenir

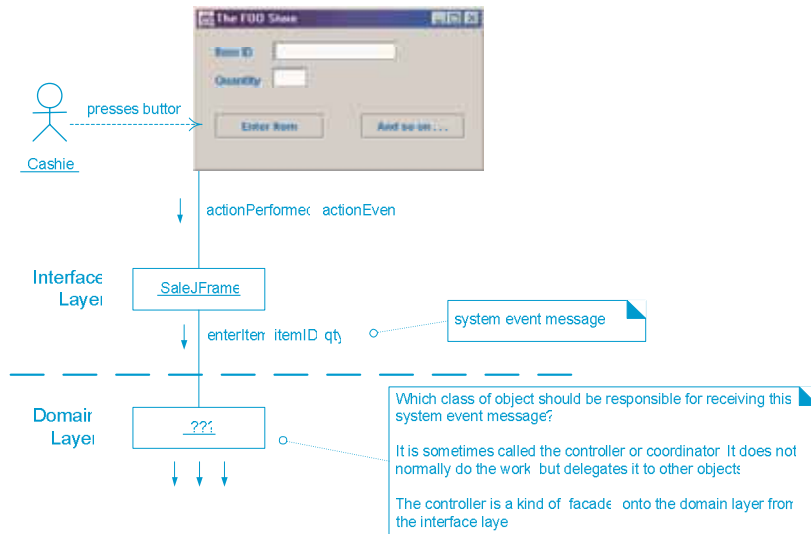
## Örnek



## Denetçi—Controller

- Çözüm: Sistem olaylarını algılama ve değerlendirme sorumluluğunu alacak sınıfı aşağıdaki iki seçenektan birini kullanarak oluşturun:
  - Tüm sistemi, cihazı veya alt sistemi temsil eden bir sınıf
  - Bir kullanım senaryosunu temsil eden bir sınıf
- Problem: Sistem olayları ile ilgili işleri yapmakla kim sorumludur?
- Sistem olayları dış aktörler tarafından üretilen olaylardır.

## Örnek



## 2

## RATIONAL UNIFIED PROCESS (BÜTÜNLEŞTİRİLMİŞ SÜREÇ)



## RUP NEDİR?

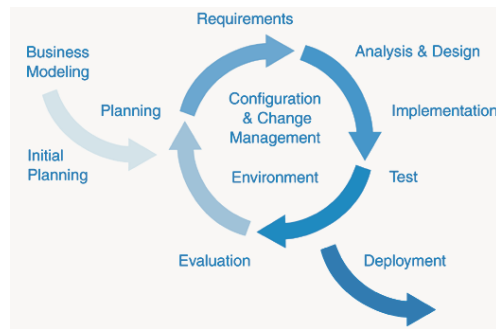
- ▶ RUP yinelemeli, artımsal, mimari merkezli, risk güdümlü, kullanım senaryolarına dayalı bir yazılım geliştirme süreci modelidir.
- ▶ RUP iyi tanımlanmış ve yapılandırılmış bir yazılım sürecidir: **Kimin Neden** sorumlu olduğu, işlerin **Nasıl** ve **Ne Zaman** yapılacağı açıkça tanımlanır.



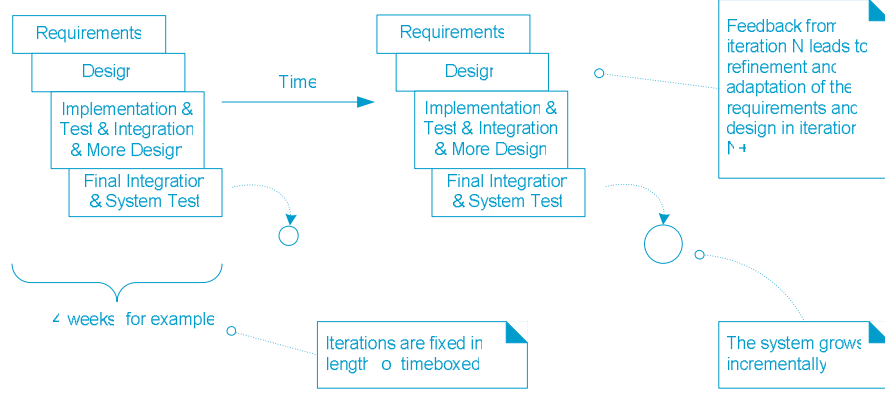
## Temel RUP Yaklaşımı

- ▶ “Attack major risks early and continuously...or they will attack you”
- ▶ Yürütülebilir yazılıma odaklanmak,
- ▶ Projede değişikliklere başta izin vermek.
- ▶ Riskleri erken gidermek.
- ▶ Sistemi bileşenlerle tasarlamak,
- ▶ Tek bir takım olarak çalışmak,
- ▶ Kalite odaklı çalışmak ✗ “Önce ürünü çıkar, kaliteyi sonra artırırısın?”

## Yinelemeli Geliştirme Yaklaşımı



- ▶ Her bir çevrim bir önceki çevrimin çıktısını girdi olarak kabul eder.



## Kaç iterasyon? Her iterasyonun Süresi?

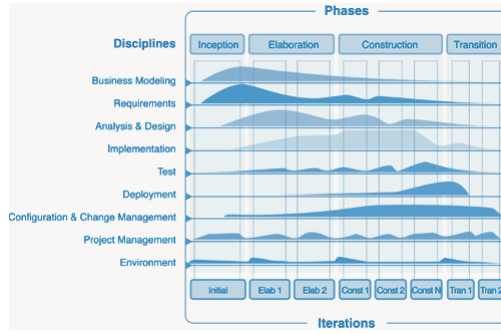
- ▶ 2 hafta-2 ay,
- ▶ Daha karmaşık projeler daha uzun iterasyon süresi demek değildir,
- ▶ Daha karmaşık proje daha fazla iterasyon demektir,
- ▶ Takım üyelerinin deneyim düzeyi,
- ▶ Paralel geliştirme takımlarının varlığı,
- ▶ Başlangıç iterasyonları daha uzun tutulabilir (Her iterasyonda deneyim artar!)

## Yinelemeli Yaklaşımın Kazanımları

- ▶ Değişen isteklere uyum,
- ▶ Erken geri besleme,
- ▶ Büyük sistemlerde çözümlene kolaylığı,
- ▶ Risklerin erken sezilmesi ve giderilmesi,
- ▶ Yeniden kullanılabilirliği kolaylaştırır,
- ▶ Erken ürün elde etme: şelale modelinde “big-bang”
- ▶ Hataları birkaç iterasyonda saptama ve düzeltme
- ▶ Her iterasyonda deneyim kazanma

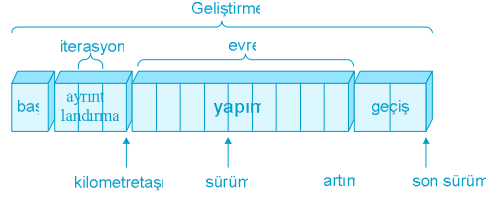


## RUP'un İki Boyutu



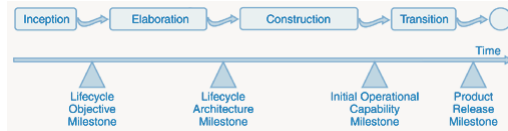
- ▶ Devinimli boyut: yatay eksen—çevrimler, evreler, iterasyonlar, ve kilometre taşı
- ▶ Durağan boyut: dikey eksen—roller, faaliyetler

## Evreler



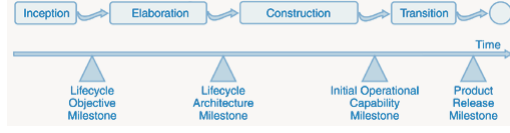
- ▶ Başlangıç: yapılabirlilik, tamam/devam kararı
- ▶ Ayrıntılılandırma: teknik olarak karmaşık ve riskli görevlerin (tasarım, kodlama, test gibi) iteratif olarak oluşturulması
- ▶ Yapım: Gerçekleminin tamamlanması (ara ve alfa sürümlerin üretilmesi)
- ▶ Geçiş: Müşteri ihtiyaçlarını karşılayan ve beta testi yapılmış ürününün teslimi

## Başlangıç—Inception



- ▶ Amaç:
  - Projenin konusunu anlamak,
  - Ticari kullanım senaryosu oluşturmak,
  - Devam etmek için takımı ortak etmek
- ▶ Kilometre taşı:
  - **L**ife**C**ycle **O**bjective Milestone (LCO)

## Ayrıntılandırma—Elaboration



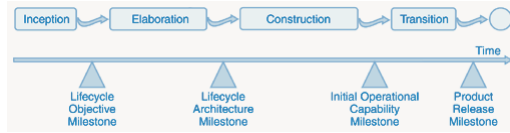
► Amaç:

- Önemli teknik risklerin azaltılması,
- Temel tasarım mimarisini oluşturmak,
- Sistemi gerçekleştirmek için gerekli olanların anlaşılması

► Kilometre taşı:

- **L**ife**C**ycle **A**rchitecture Milestone (LCA)

## Yapım—Construction



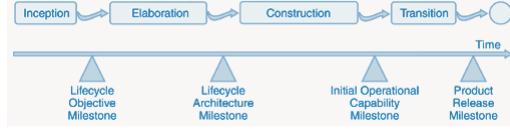
► Amaç:

- İlk kullanıma hazır sürümün çıkarılması

► Kilometre taşı:

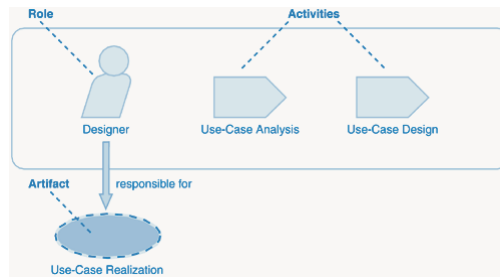
- **I**nitial **O**perational **C**apability Milestone (IOC)

## Geçiş—Transition



- ▶ Amaç:
  - Ürünün son sürümünü çıkarmak ve müşteriye teslim etmek.
- ▶ Kilometre taşı:
  - **P**roduct **R**elease Milestone (PR)

## RUP'un Duragan Bileşenleri



- ▶ Duragan yapı, süreç bileşenleri—faaliyetler, disiplinler, ürünler ve roller arasındaki mantıksal bağın nasıl olacağını belirler.
- ▶ Süreç, kimin? Neyi? Nasıl? ve Ne Zaman? yapacağını tanımlar.

## Dört Temel Modelleme Elemanı

- ▶ Roles. The who
- ▶ Activities. The how
- ▶ Artifacts. The what
- ▶ Workflows. The when

## Rol

- ▶ Rol kişinin/grubun proje boyunca giydiği şapkaya benzer,
- ▶ Kişi proje boyunca farklı şapkalar giyebilir,
- ▶ Rol bir kişinin ilgili işi nasıl yapması gerektiğini ve o rol için kişilerin sahip olması gereken özellikleri ve sorumlulukları tanımlar,
- ▶ Bir kişi bir yada daha fazla rol alabilir, birkaç kişi aynı rolü oynayabilir.

## Eylem—Activity

- ▶ Belirli bir role ait eylem o rolü üstlenen kişinin yapması gereken birim işi tanımlar.
- ▶ Her eylem eylemin açık bir amacı vardır. Genellikle bu bazı çıktıların (model, plan gibi) güncellenmesi yada yaratılması cinsinden ifade edilir.
- ▶ Her eylem bir role atanmıştır.
- ▶ Eylem genellikle birkaç saat/gün alır ve birkaç çıktıyı etkiler.
- ▶ Eylem planlamada kullanılabilir büyüklükte olmalı
- ▶ Eylem birkaç kez tekrar edilebilir—özellikle bir iterasyon diğerine geçerken yeniden gözden geçirilebilir—aynı rolde, ama aynı kişi olmak zorunda değil .

## Adımlar

- Eylemler üç temel sınıfa ayrılabilen adımlara bölünmüştür:
- ▶ Düşünme (Thinking): rolü yerine getirecek kişi görevin doğasını anlar, giriş çıktılarını toplar ve inceler ve sonuç üretir,
  - ▶ Yerine Getirme (Performing): Rol bir çıktı üretir yada günceller,
  - ▶ Gözden Geçirme (Reviewing): Rol sonuçları belirli bir kritere göre gözden geçirir.



## Çıktı—Artifact

- ▶ Çıktı bir süreç tarafından üretilen, değiştirilen yada kullanılan bir bilgi parçasıdır.
- ▶ Çıktılar projenin en somut elemanlarıdır: sonuç ürünü ortaya çıkarılırken projenin ürettikleri yada kullandıkları.
- ▶ Çıktı bir eylemi gerçekleştirmek için roller tarafından girdi olarak kullanılır ve diğer eylemlerin sonucu yada ürünüdür.

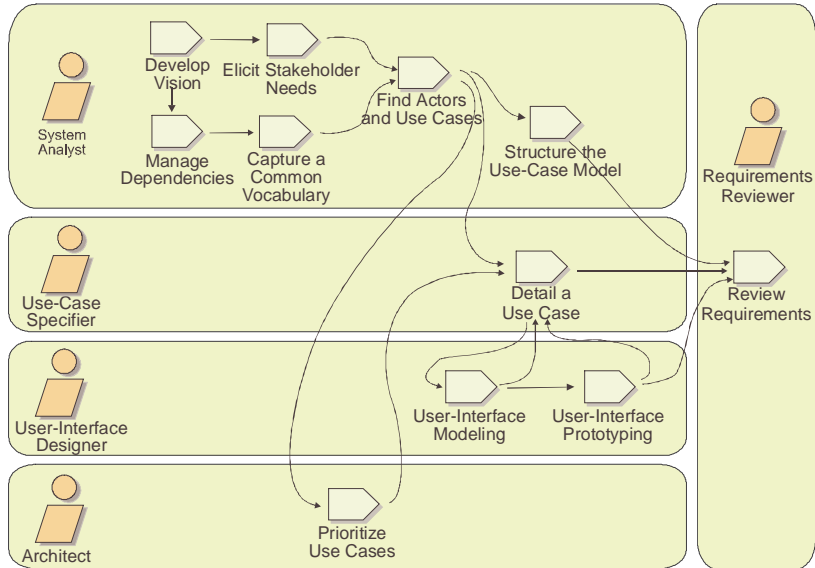
## Çıktının Biçimleri

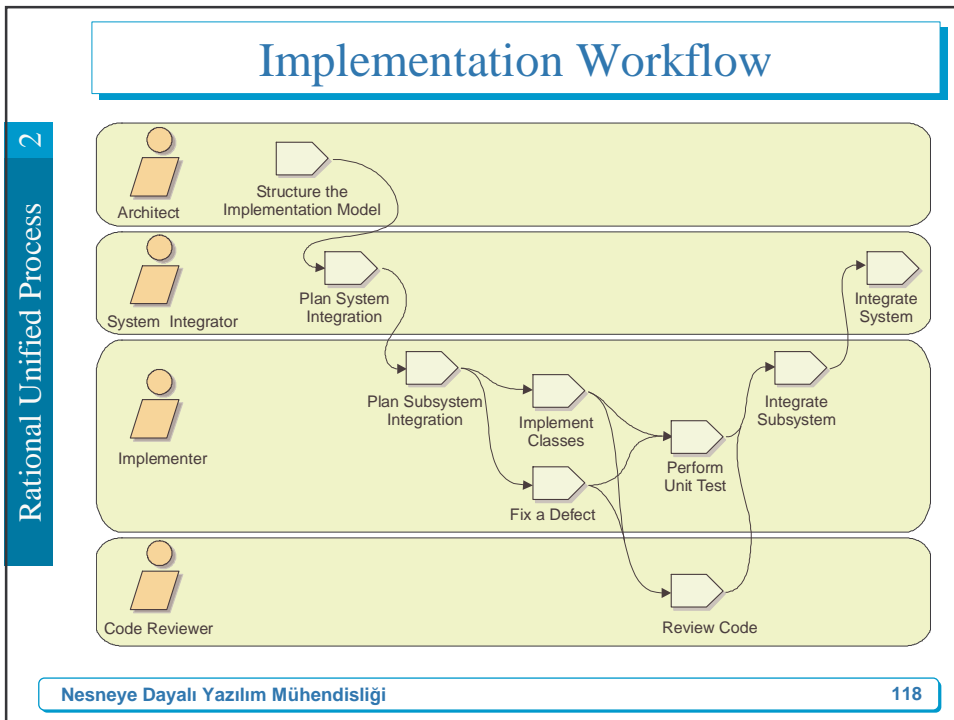
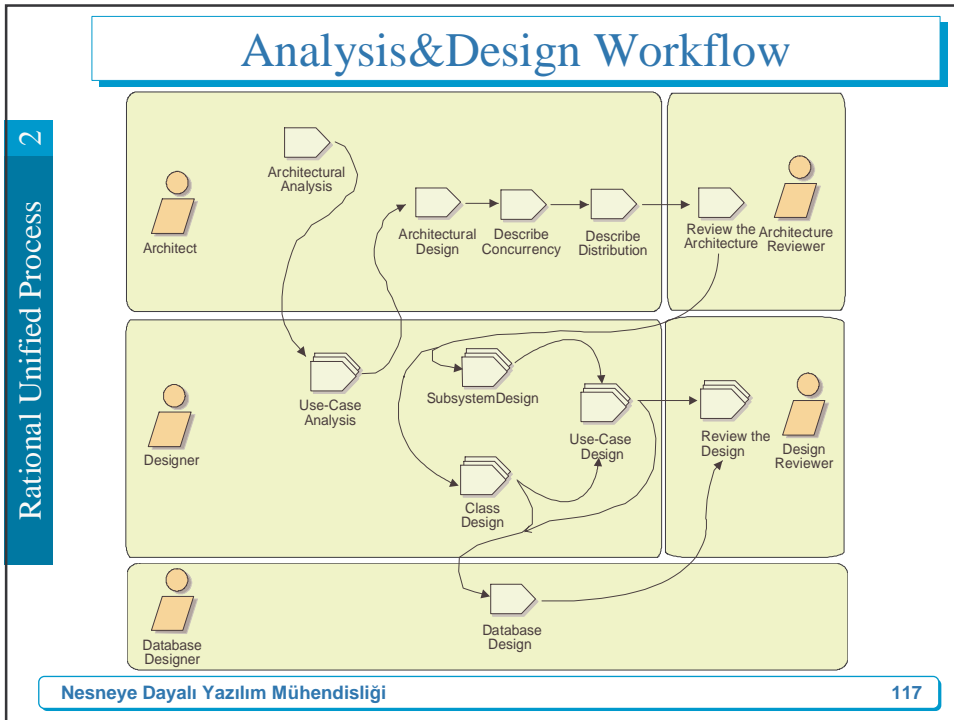
- Çıktılar çeşitli şekil yada biçimlerde olabilir:
- ▶ Model: Use-Case Model, Design Model
  - ▶ Model Bileşeni: Sınıf, Use-Case (UC)
  - ▶ Doküman
  - ▶ Kaynak Kod
  - ▶ Yürütülebilir Kod

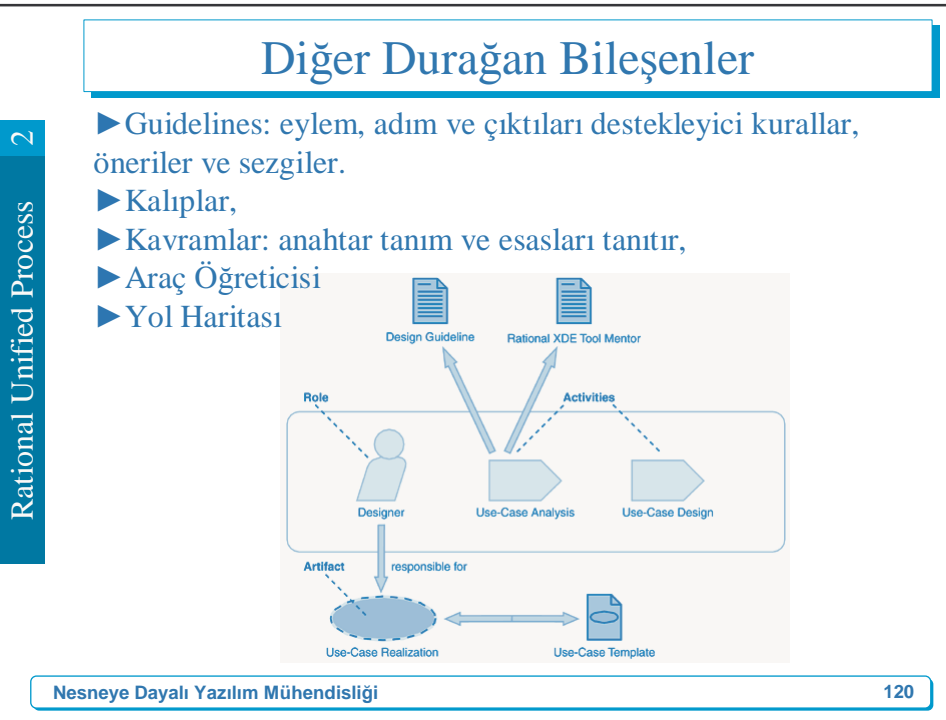
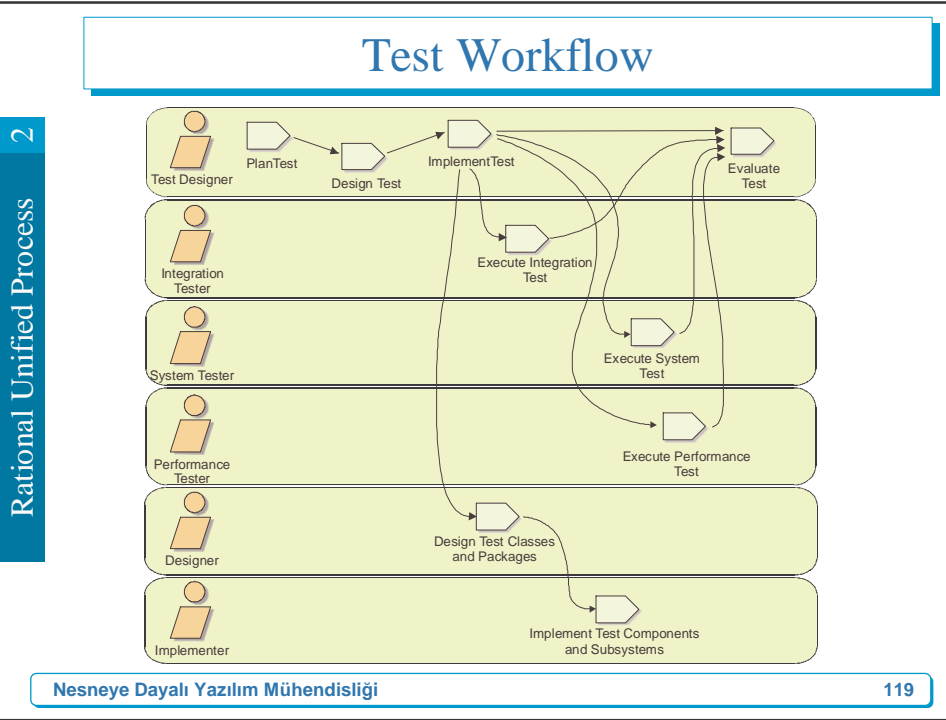
## İş Akışı—Workflow

- ▶ Roller, Eylemler ve Çıktılar tam olarak bir süreç oluşturmazlar.
- ▶ Eylemleri anlamlı bir sıraya sokmak ve roller arasındaki etkileşimi tanımlamak için bir mekanizmaya ihtiyaç vardır.
- ▶ İş akışı çeşitli şekil ve biçimlerde olabilir. Bunlardan en yaygın olarak kullanılan ikisi:
  - Disiplin: Yüksek-seviye iş akışı
  - İşakış Detayı: disiplin içinde tanımlanmış işakışları

## Requirement Workflow







## Disiplin—Discipline

- ▶ Business modeling
- ▶ Requirements management
- ▶ Analysis and design
- ▶ Implementation
- ▶ Deployment
- ▶ Test
- ▶ Project management
- ▶ Change management
- ▶ Environment

## Summary

- The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of a software-intensive system
- A software development process defines **Who** is doing **What**, **When** and **How** in building a software product
- The Rational Unified Process has four phases: **Inception**, **Elaboration**, **Construction and Transition**
- Each phase ends at a major milestone and contains one or more iterations
- An **iteration** is a distinct sequence of activities with an established plan and evaluation criteria, resulting in an executable release

# 3

## RATIONAL ROSE ENTERPRISE

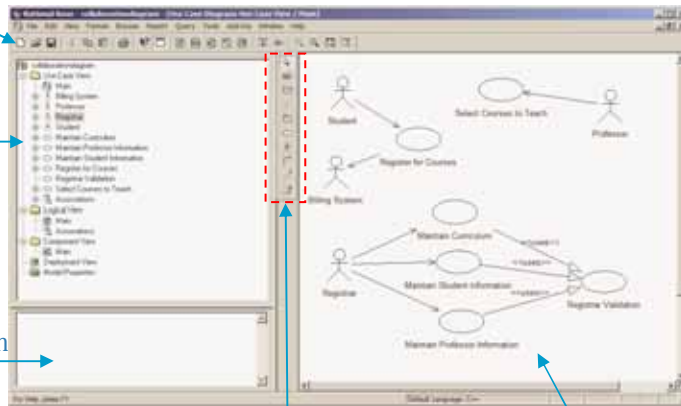
### Rational Rose Workspace

Rational Rose Enterprise 3

Main Toolbar

Project tree view

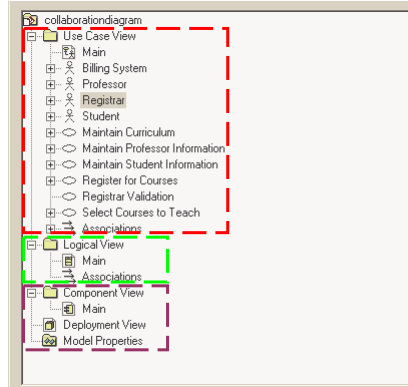
Documentation



Dynamic toolbar

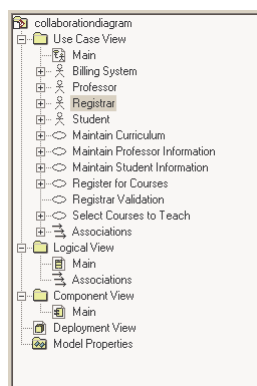
Diagram

## Project Tree



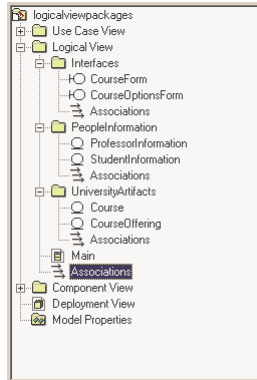
- Use case view
  - Used for analysis
  - Contains UC diag., actors.
- Logical view
  - Used for design
  - Contains packages, classes, assoc.
  - Class, sequence and similar diag.
- Component & Deployment view
  - Used for components design and final deployment diagrams

## Project Tree Usage



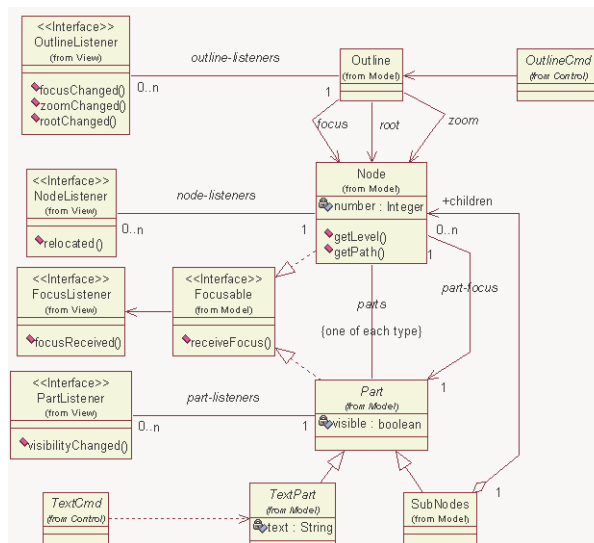
- Double-click on an item to open its specification.
- Right-click on an item to add a diagram or sub-items.
- Most items can be dragged and dropped into a diagram to create an instance.
- If an item is modified, the change is reflected in the entire project.

## Logical View



- Contains the project's packages, classes, interfaces and associations.
- Classes and interfaces contains their fields and methods.
- Classes are drag-and-drop-able into diagrams to create instances.
- If a class is modified in one place, the change is reflected to the whole project.

## Class Diagram





## Class Diagram Toolbar

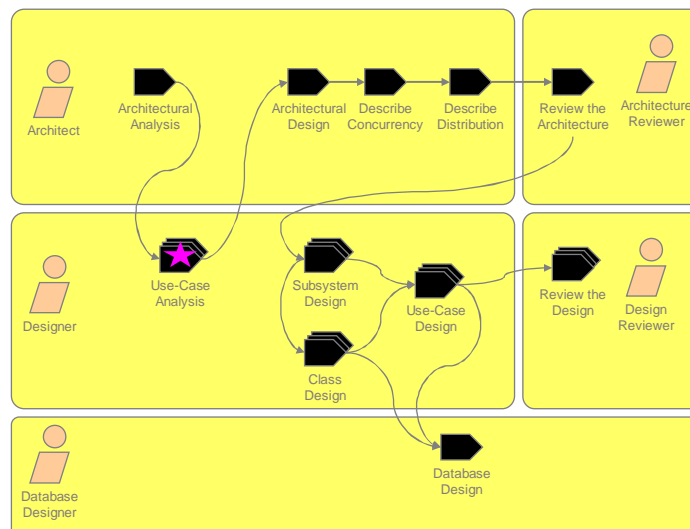


1. Select items
2. Add text
3. Add note
4. Bind note to item
5. Create new class
6. Create new interface
7. Draw new association
8. Association class
9. Create new package
10. Draw dependency
11. Class inheritance
12. Interface implementation

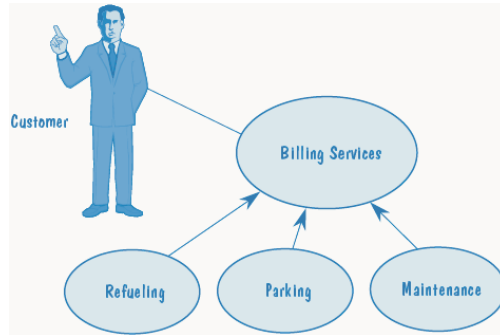
# 4

## Practice Session

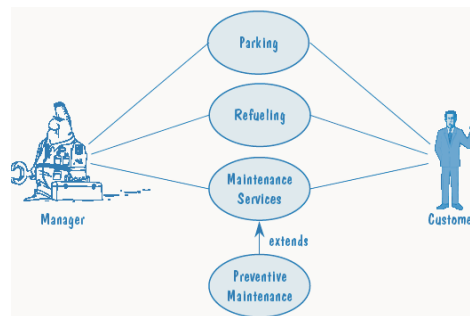
The Royal Service Station provides three types of services to its customers: refueling, vehicle maintenance, and parking. That is, a customer can add fuel to the tank in his or her vehicle (car, motorcycle or truck), can have the vehicle repaired, or can park the vehicle in the station parking lot. A customer has the option to be billed automatically at the time of purchase (of fuel, maintenance, or parking) or to be sent a monthly paper bill. In either case, customer can pay using cash, credit card, or personal check. Royal Service Station fuel is sold according to price per gallon, depending on whether the fuel is diesel, regular, or premium. Service is priced according to the cost of parts and labor. Parking is sold according to daily, weekly, and monthly rates. The price for fuel, maintenance services, parts and, parking may vary; only Manny the station manager can enter or change prices. At his discretion Manny may designate a discount on purchases for a particular customer; this discount may vary from one customer to another. A 5% local sales tax applies to all purchases.



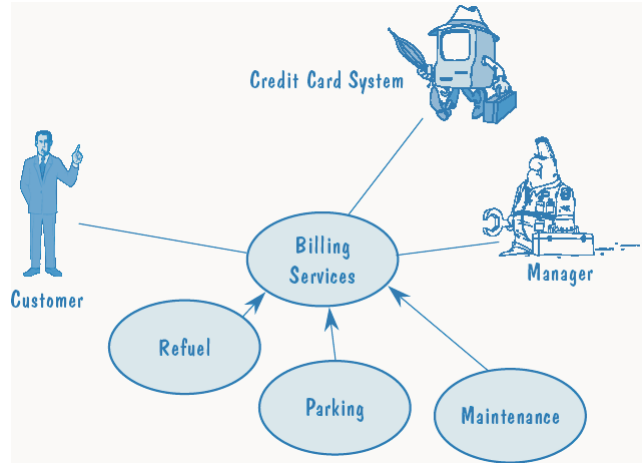
# Requirements



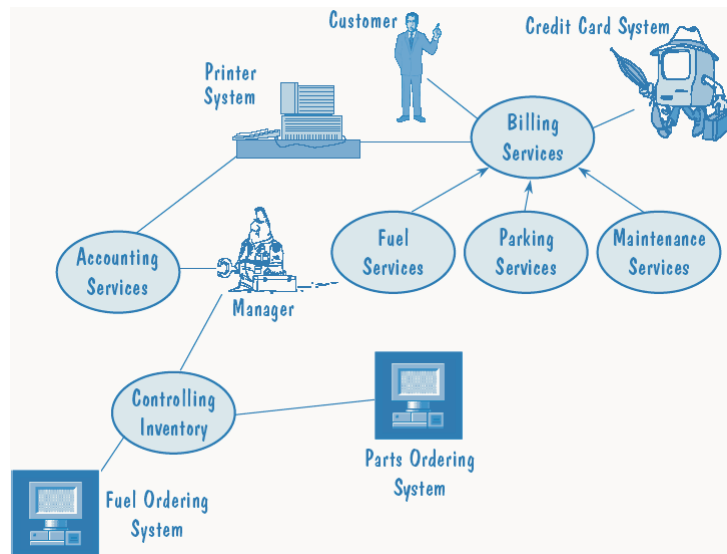
# First Extension



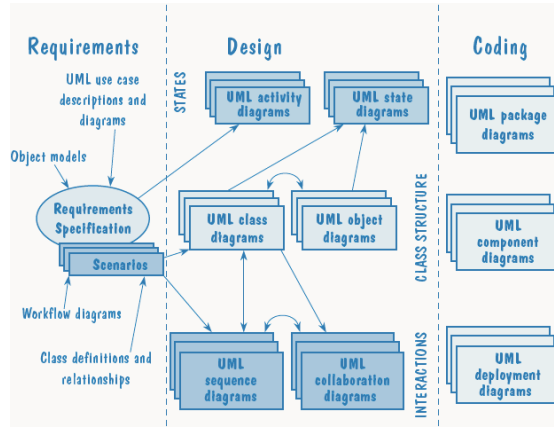
## Second Extension



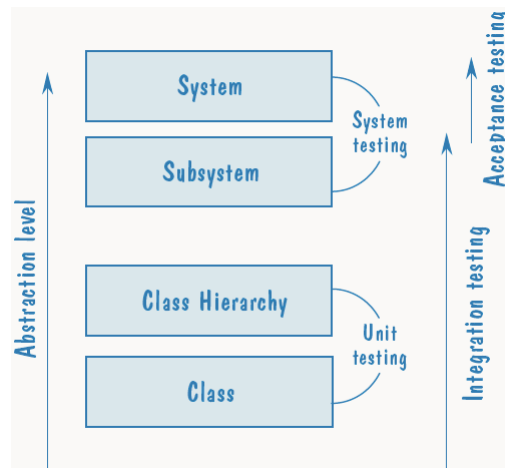
## Third Extension



# How UML Supports the Development



# Relationship of Testing Types to OO



## System Design

4

Royal Service Station

- ▶ The design starts with extracting nouns
- ▶ Aim is to find classes

A **customer** has the option to be billed automatically at the time of **purchase** (of **fuel**, **maintenance**, or **parking**) or to be sent a monthly **paper bill**. In either case, customer can pay using **cash**, **credit card**, or **personal check**. Royal Service Station fuel is sold according to **price** per gallon, depending on whether the fuel is diesel, regular, or premium. Service is priced according to the cost of parts and labor. Parking is sold according to daily, weekly, and monthly rates. The price for fuel, maintenance services, parts and, parking may vary; only Manny the **station manager** can enter or change prices. At his discretion Manny may designate a **discount** on purchases for a particular customer; this discount may vary from one customer to another. A 5% local sales **tax** applies to all purchases.

4

Royal Service Station

- Personal check
- Paper bill
- Credit card
- Customer
- Station manager
- Purchase
- Fuel
- Service
- Discount
- Tax
- Parking
- Maintenance
- Cash
- Prices

## First Grouping of Attributes and Classes

1<sup>st</sup> Step

4  
Royal Service Station

### Attributes

- Personal check
- Credit card
- Discount
- Tax
- Cash
- Price

### Classes

- Paper bill
- Customer
- Station manager
- Purchase
- Fuel
- Service
- Parking
- Maintenance

## Scenario Script

4  
Royal Service Station

- The system applies only to regular repeat customers. A regular repeat customer means a customer identified by **name**, **address** and **birthday** who uses the station's services at least once per month for at least six months.
- The system will send **periodic messages** to customers, reminding them when their vehicles are due for maintenance. Normally, maintenance is needed every six months.

## First Grouping of Attributes and Classes

2<sup>nd</sup> Step

4  
Royal Service Station

### Attributes

- Personal check
- Credit card
- Discount
- Tax
- Cash
- Price
- Birthday
- Name
- Address

### Classes

- Paper bill
- Customer
- Station manager
- Purchase
- Fuel
- Service
- Parking
- Maintenance
- Periodic messages

## Scenario Script

4  
Royal Service Station

- The system must handle the data requirements for interfacing with other systems. A credit card system is used to process credit card transactions for products and services. The credit card system uses the card number, name, expiration date, and amount of the purchase. After receiving this information, the credit card system confirms that the transaction is approved or denied. The **parts ordering system** receives the part code and number of parts needed. It returns the date of parts delivery. The **fuel ordering system** requires a fuel order description consisting of fuel type, number of gallons, station name, and station identification code. It returns the date when the fuel will be delivered.
- The system will track credit history and send **warning letters** to customers whose payments are overdue.



## First Grouping of Attributes and Classes

3<sup>rd</sup> Step

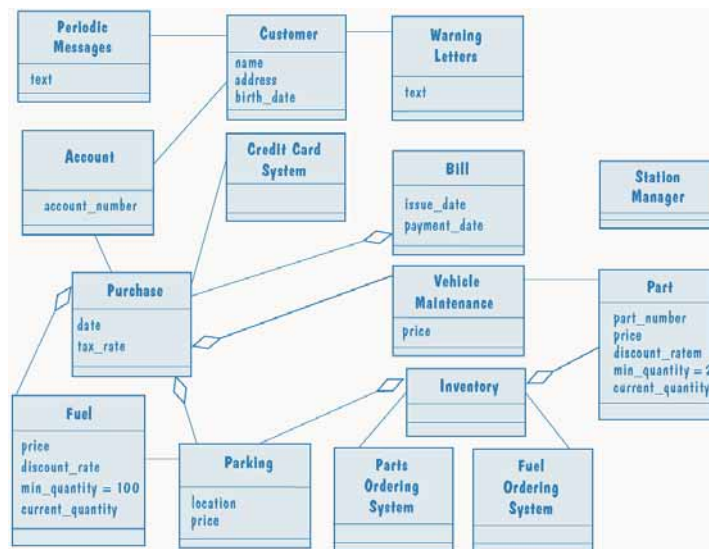
### Attributes

- Personal check
- Credit card
- Discount
- Tax
- Cash
- Price
- Birthday
- Name
- Address

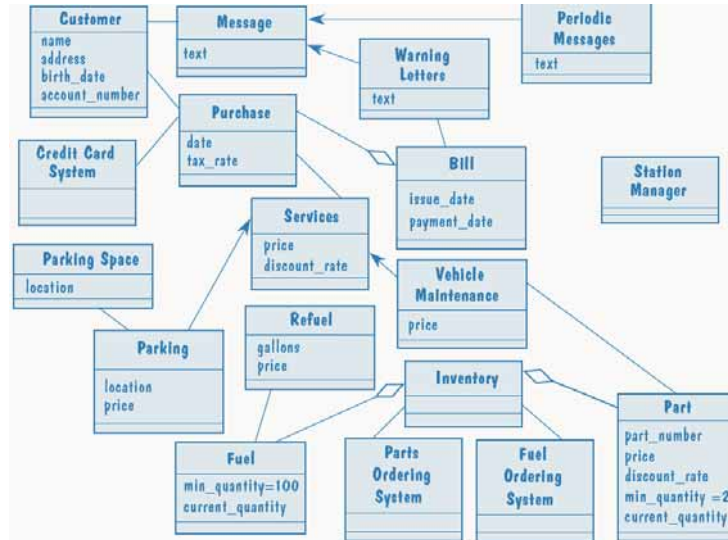
### Classes

- Paper bill
- Customer
- Station manager
- Purchase
- Fuel
- Service
- Parking
- Maintenance
- Periodic messages
- Inventory
- Credit Card System
- Part-Ordering System
- Fuel-Ordering System

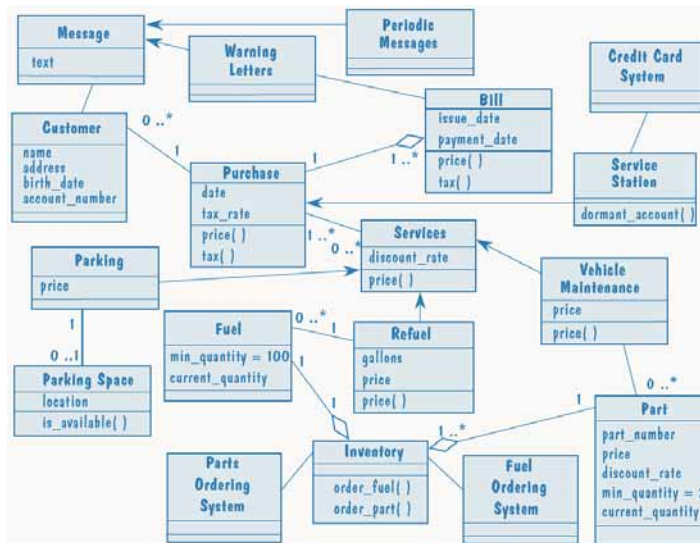
## First Iteration



## Second Iteration

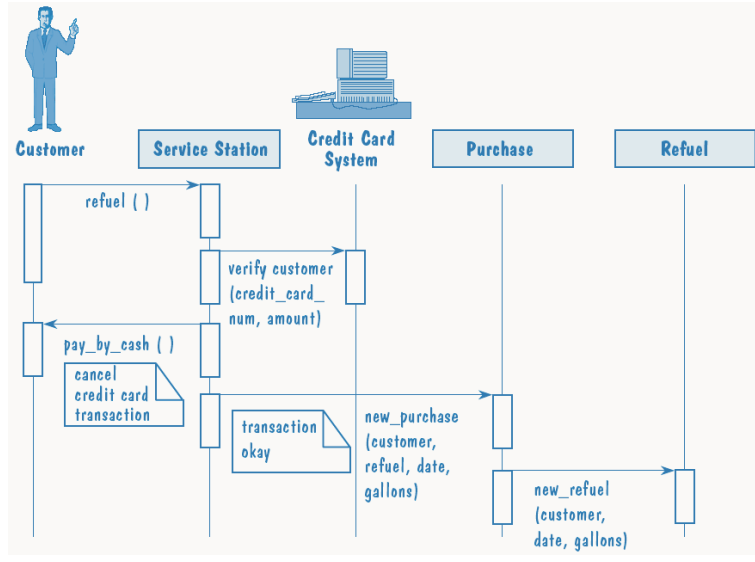


## Third Iteration



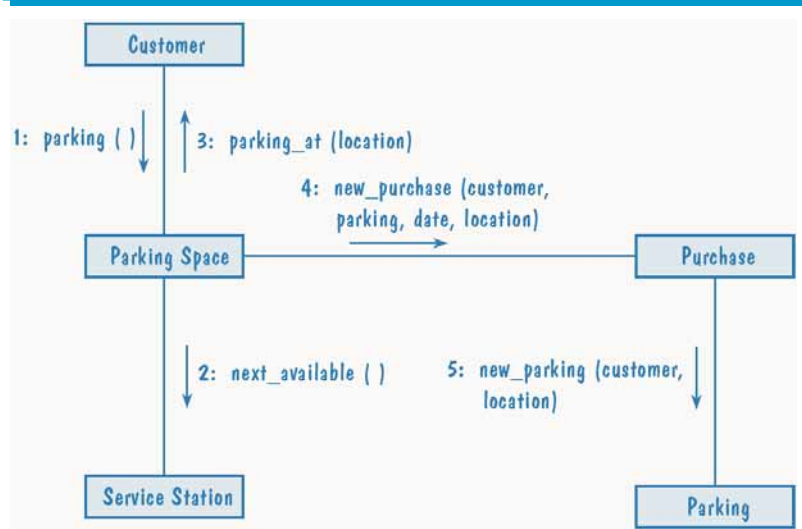
## Sequence Diagram for the *refuel* use case

4  
Royal Service Station

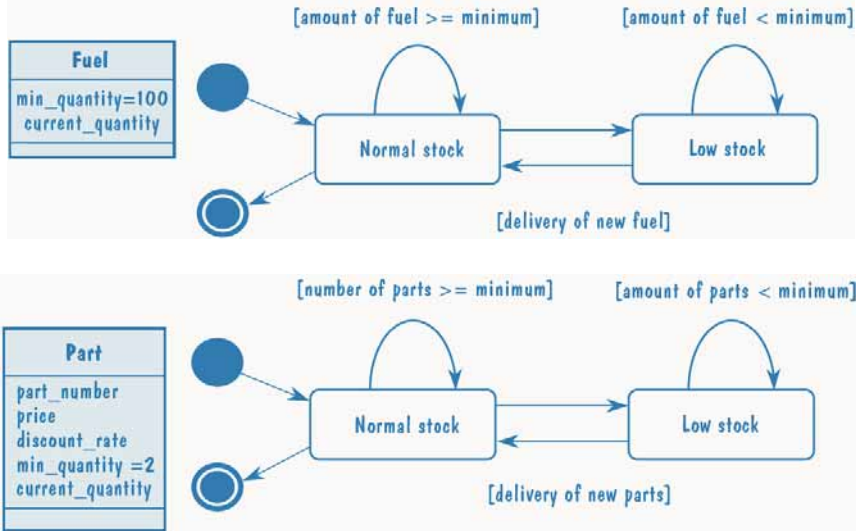


## Collaboration Diagram for the *parking* use case

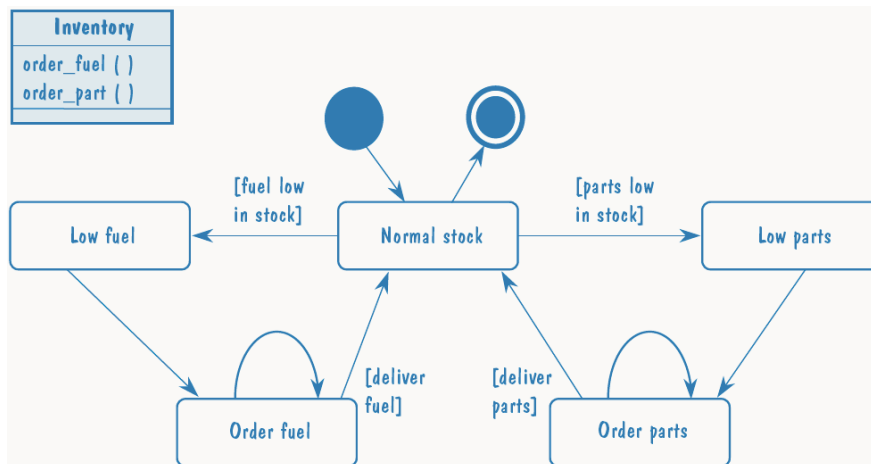
4  
Royal Service Station



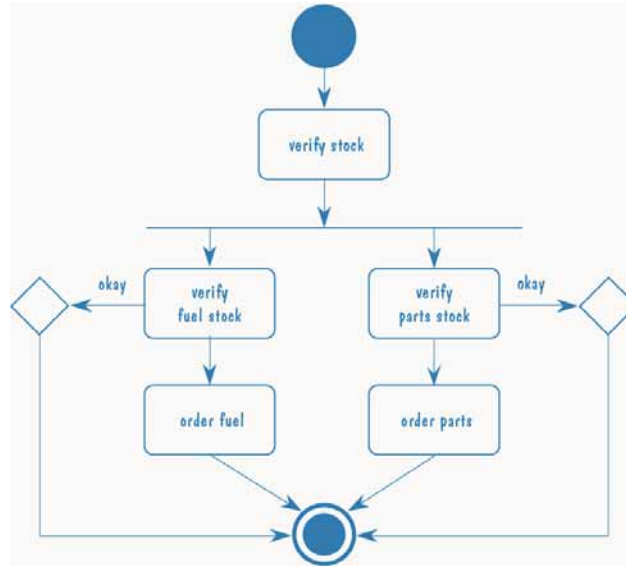
## State Diagram for *fuel* and *parts* classes



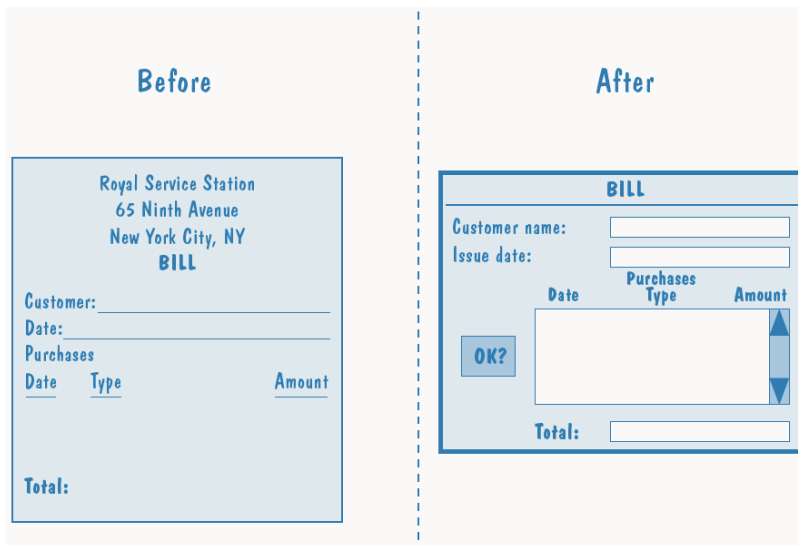
## State Diagram for *inventory* class



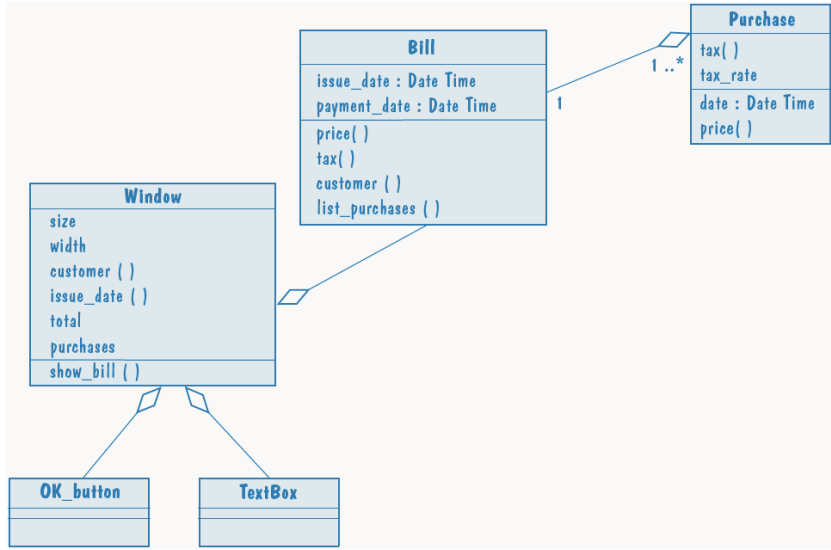
## Activity Diagram for *inventory* class



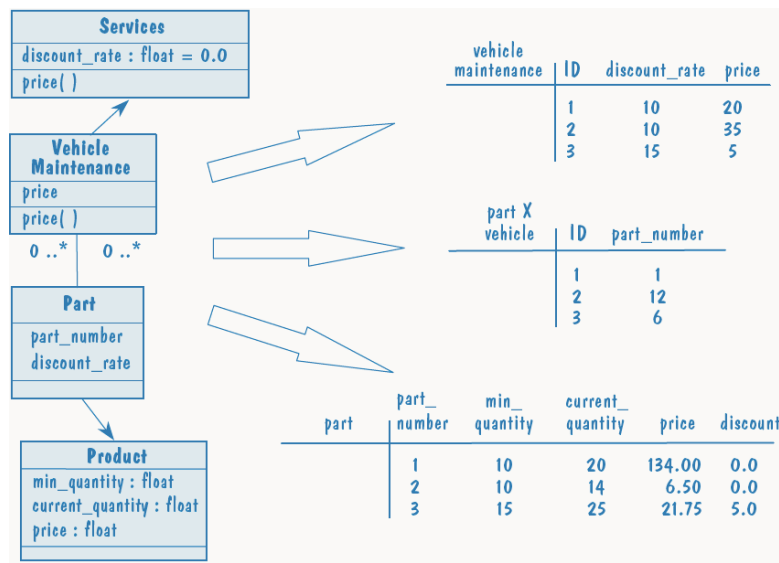
## User Interface Design



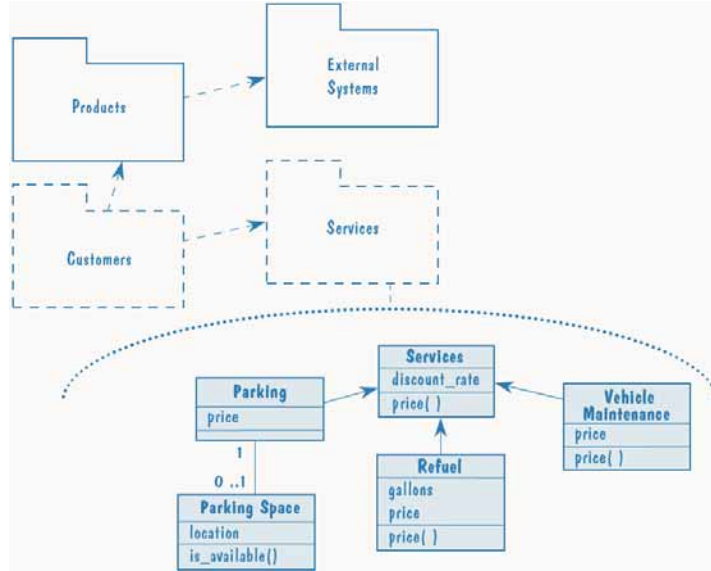
## Class Design for User Interface



## Data Management Design



## Package Diagram



# 5

## Design Patterns

## What is a pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

**Christopher Alexander**

“The timeless way of building” & “A pattern language”

## Architectural Patterns

- Original Concept conceived of patterns by **Christopher Alexander** in the 1970s – first arose in the book *The Timeless Way of Building*
- He defined a hierarchical collection of architectural design patterns for use in designing future buildings



## Patterns in Software Engineering

- Patterns were then adapted by the software world.
- Jan O. Borchers points out however that Software Design Patterns are considered as a useful language for communication *among software developers* and as a vehicle for introducing less experienced developers into the field – different to the architectural case

## Motivation

- Developing software is hard
- Developing reusable software is even harder
- Proven solutions include patterns and frameworks

## Familiar Patterns

- Learning to develop good software is similar to learning to play good chess!

### To become a Chess Master:

1. First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
2. Then learn principles
  - e.g., relative value of certain pieces, strategic value of centre squares, power of a threat, etc.

## Familiar Patterns

- However, to become a master of chess, one must study the games of other masters
  - These games contain patterns that must be understood, memorized, and applied repeatedly
  - There are hundreds of these patterns

## Programming Patterns

To become a Software Design Master:

1. First learn the rules
  - e.g., the algorithms, data structures and languages of software
2. Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.

## Programming Patterns

- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
  - There are hundreds of these patterns

## What are Software Patterns?

- Patterns are the recurring solutions to the problems of design
- Patterns support reuse of software architecture and design
- The learning objective of this course is that ALL of the class will be able to describe and implement solutions to software engineering problems to fellow problem solvers in terms of *object-oriented design patterns*

## More Explicitly

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - “Patterns == problem/solution pairs in a context”
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

## Benefits of Patterns

- Learning about existing and searching for domain-specific patterns would:
  - **Improve Communication**
    - Among Designers on a team, among designers on different teams, between a designer and herself
  - **Improve documentation**
    - The use of pattern names in documentation carries a lot of information and saves space
  - **Reuse without creation and maintenance of code libraries**
    - Not tied to any specific programming language
  - **Improve future designs as collective design experience is applied to new projects**
    - Sometimes called Knowledge Management

## Domain Specific Patterns

- Application domains represent the context aspect of design patterns:
  - **Communications**
  - **Distributed Computing**
  - **Data Structures**
  - **Building Object-Oriented Frameworks**

## When To Use Patterns

1. Solutions to problems that recur with variations
  - No need for reuse if the problem only arises in one context
2. Solutions that require several steps
  - Not all problems need all steps
  - Patterns can be overkill if solution is simple linear set of instructions
3. Solutions where the solver is more interested in the existence of the
  - Solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really wants to understand

## What makes it a Pattern ?

- A pattern must:
  - solve a problem,
    - it must be useful!
  - have a context,
    - It must describe where the solution can be used.
  - recur,
    - It must be relevant in other situations.
- Teach,
  - It must provide sufficient understanding to tailor the solution.
- have a name.
  - It must be referred to consistently.

## Benefits of Design Pattern

- Design patterns enable large-scale reuse of software architectures.
  - They also help document systems to enhance understanding.
- Patterns explicitly capture expert knowledge and design tradeoffs, and make this expertise more widely available.
- Patterns help improve developer communication.
  - Pattern names form a vocabulary
- Patterns help ease the transition to object-oriented technology.

## Drawbacks of Design Pattern

- Patterns do not lead to direct code reuse.
  - For reuse of code:  
 $O-O \text{ Frameworks} == \text{Design Patterns} + \text{Code}$
- Patterns are deceptively simple.
- Teams may suffer from pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.
- Integrating patterns into a software development process is a human-intensive activity.

## Misconceptions of Patterns

- A Pattern is a solution to a problem in a context
  - Missing Recurrence, Teaching and a Name
- Patterns are just jargon, rules, programming tricks, data structures, etc..
- Seen one, seen them all
- Patterns need tool or methodological support to be effective

## Misconceptions of Patterns

- Patterns guarantee reusable software, higher productivity, etc...
- Patterns 'generate' whole software architectures
- Patterns are for (object-oriented) design or implementation
  - Design Patterns are not the only type of software pattern!
- There's no evidence that patterns help anybody



## Different Types of Patterns

- O-O Design Patterns
  - Design Patterns: Elements of Reusable O-O Software
- Distributed/Concurrent Programming Patterns
- User Interface Design Patterns
- Architectural Patterns
- Software Process Patterns

## Summary

- Mature engineering disciplines have handbooks that describe successful solutions to known problems
  - automobile designers don't design cars using the laws of physics, they adapt adequate solutions from the handbook known to work well enough
  - The extra few percent of performance available by starting from scratch typically isn't worth the cost
- Patterns can form the basis for the handbook of software engineering
  - If software is to become an engineering discipline, successful practices must be systematically documented and widely disseminated

## A Pattern has 4 essential elements

**Pattern name**

**Problem**

**Solution**

"the pattern provides an abstract description of a design problem and a general arrangement of elements solves it"

**Consequences**

## OOP

- In a pure objected-oriented language, everything is an object!
  - Objects consist of state and behavior and must be explicitly created (and destroyed in C++)
  - Objects use services provided by other objects
- Design patterns deal with issues relating to the behavior of objects, the lifetime of objects, the interface of objects, structural relationships between objects, etc.
- The GoF book categorizes design patterns into **structural**, **behavioral** and **creational** patterns.

## Design Pattern Space - GoF

- Creational patterns
  - Deal with initializing and configuring classes and objects
- Structural patterns
  - Deal with decoupling interface and implementation of classes and objects
- Behavioral patterns
  - Deal with dynamic interactions among societies of classes and objects

## Creational Patterns - GoF

- Factory Method
  - Method in a derived class creates associates
- Abstract Factory
  - Factory for building related objects
- Builder
  - Factory for building complex objects incrementally
- Prototype
  - Factory for cloning new instances from a prototype
- Singleton
  - Factory for a singular (sole) instance

## Structural Patterns - GoF

- Adapter
  - Translator adapts a server interface for a client
- Bridge
  - Abstraction for binding one of many implementations
- Composite
  - Structure for building recursive aggregations
- Decorator
  - Decorator extends an object transparently

## Structural Patterns (cont'd)

- Facade
  - Facade simplifies the interface for a subsystem
- Flyweight
  - Many fine-grained objects shared efficiently
- Proxy
  - One object approximates another

## Behavioral Patterns - GoF

- Chain of Responsibility
  - Request delegated to the responsible service provider
- Command
  - Request as first-class object
- Interpreter
  - Language interpreter for a small grammar
- Iterator
  - Aggregate elements are accessed sequentially

## Behavioral Patterns (cont'd)

- Mediator
  - Mediator coordinates interactions between its associates
- Memento
  - Snapshot captures and restores object states privately
- Observer
  - Dependents update automatically when a subject changes
- State
  - Object whose behavior depends on its state

## Behavioral Patterns (cont'd)

- Strategy
  - Abstraction for selecting one of many algorithms
- Template Method
  - Algorithm with some steps supplied by a derived class
- Visitor
  - Operations applied to elements of an heterogeneous object structure

## Classifying Patterns

		Purpose		
		Creational	Strucural	Behavioural
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

## Before there were Patterns

- First came C++ Idioms
  - James Coplien
- Example Idiom:  
“Resource acquisition is initialisation”

```
void use_file(const char *fn)
{
    FILE *f = fopen(fn, "w");
    // use f
    fclose(f);
}
```

## C++ Idioms

- What if something goes wrong while using “f”? Will “f” get closed?
- How can we guarantee that the call to fclose(f) will happen?
- What about exceptions?

## “Resource acquisition in initialisation” Idiom

- A much better choice is to move all initialisation action into a constructor, and all release actions into a destructor.
- This technique is called "resource acquisition in initialisation."
- Simplified usage protocol:
  - No need for user of object to call operations to acquire and release resources for it
  - Users can start using the object right after it has been created

## C++ Code for the Idiom

```
#include <stdio.h>
class FilePtr {
    FILE *p;
public: FilePtr( const char *n, const char *a ) { p = fopen(n,a); }
    FilePtr( FILE *pp) { p = pp; }
    ~FilePtr() { fclose(p); }
    operator FILE*() { return p; }
};

void use_file(const char *fn) {
    FilePtr f(fn, "w");
    // use f
}
```

- Note that no matter what happens, the destructor ~FilePtr() gets called for f.



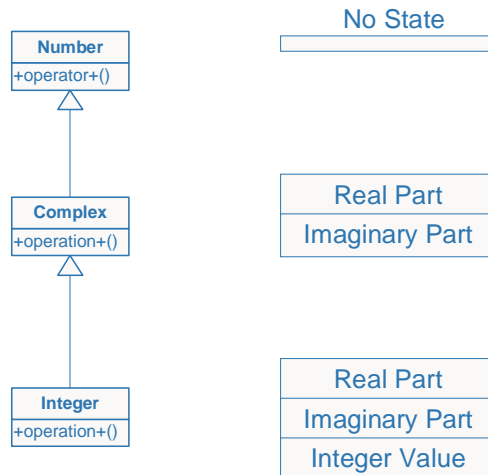
## Another Example of the Idiom

- Memory acquisition in C++
  - C++ doesn't do garbage collection
  - For every 'new' operation you write you have to write a corresponding 'delete'

```
void use_buffer(size_t x) {
    char* buffer = new char[x];
    // use buffer
    delete[] buffer;
}
```

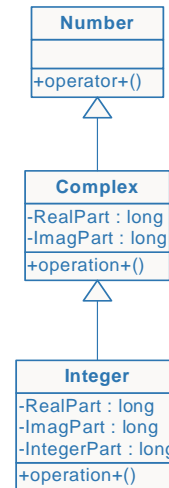
```
class Buffer{ //Uses Idiom "Resource acquisition is initialisation"
    char *p;
public: Buffer( size_t x ) { p = new char[x]; }
    ~ Buffer() { delete[] p;}
};
```

## O-O Inheritance Problem



## O-O Inheritance Problem

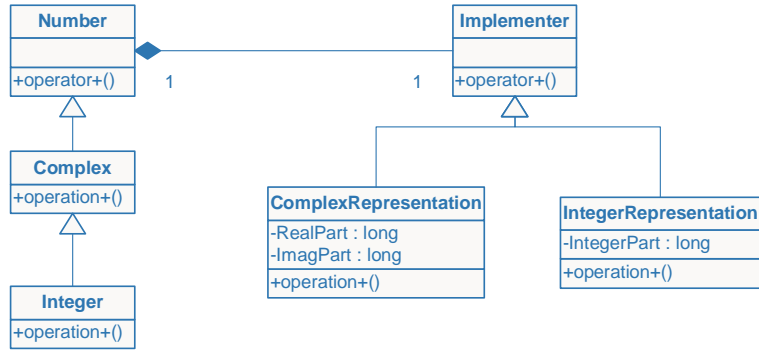
- Conceptually Integers are a specialisation of Complex numbers
- Problem with *implementation inheritance* here:
  - Integer objects inherit the attributes RealPart + ImagPart that they don't require
  - Makes Integer objects bigger and more complex than they should be



## O-O Inheritance Problem

- What is the solution here?
  - Need to decouple abstractions (numbers) from their implementation (classes)
- Solution:
  - A Design Pattern!
  - Which one?
    - Bridge Pattern (in C++)

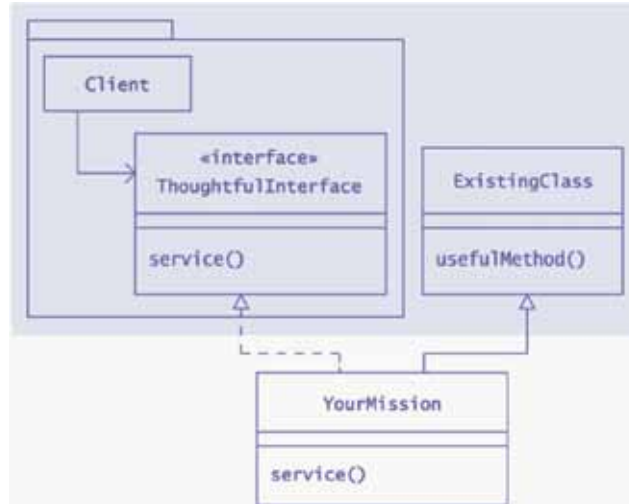
## O-O Inheritance Problem



## Adapter

- ▶ When you need to implement an expected interface, you may find that an existing class performs the services a client needs but with different method names. You can use the existing class to meet the client's needs by applying the ADAPTER pattern.
- ▶ The intent of ADAPTER is to provide the interface a client expects, using the services of a class with a different interface.

## Adapter



## Facade

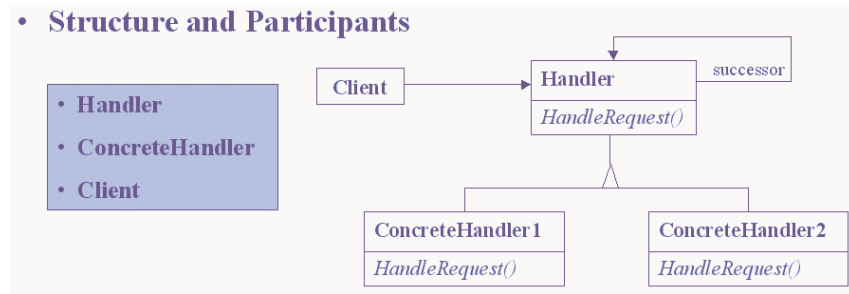
► A toolkit or subsystem developer often creates packages of well-designed classes without providing any applications that tie the classes together.

► The reusability of toolkits comes with a problem: The diverse applicability of classes in an OO subsystem may offer an oppressive variety of options. A developer who wants to use the toolkit may not know where to begin. This is especially a problem when a developer wants to apply a normal, no-frills, *vanilla* usage of the classes in a package. The FACADE pattern addresses this need. A facade is a class with a level of functionality that lies between a toolkit and a complete application, offering a vanilla usage of the classes in a package or a subsystem. The intent of the FACADE pattern is to provide an interface that makes a subsystem easy to use.

## Chain of Responsibility

**Intent** - Chain the receiving objects and pass the request along the chain until an object handles it.

• **Structure and Participants**



## Chain of Responsibility

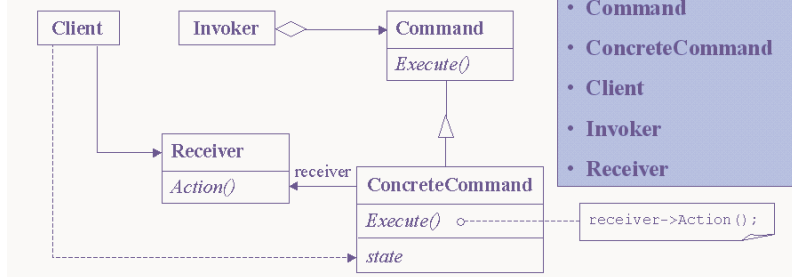
• **Applicability - use when:**

- more than one object may handle a request, and the handler isn't known a priori.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

## Command

**Intent** - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo operations.

• **Structure and Participants**



- Command
- ConcreteCommand
- Client
- Invoker
- Receiver

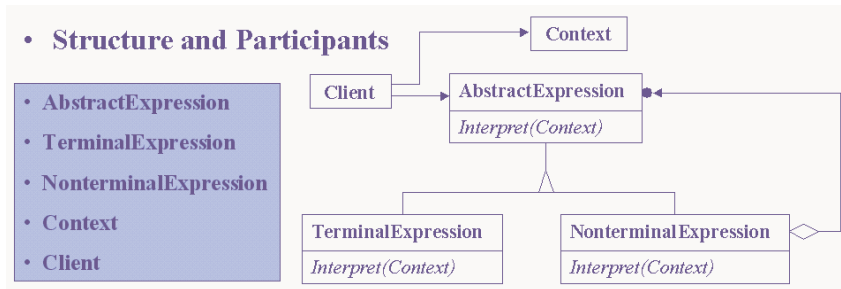
## Command

► **Applicability - use when you want to:**

- parameterize objects by an action to perform.
- specify, queue, and execute requests at different times.
- support undo.
- support logging changes so that they can be reapplied in case of a system crash.
- structure a system around high-level operations built on primitives operations.

## Interpreter

**Intent** - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



## Interpreter

► **Applicability - use when you want to:**

- Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees.

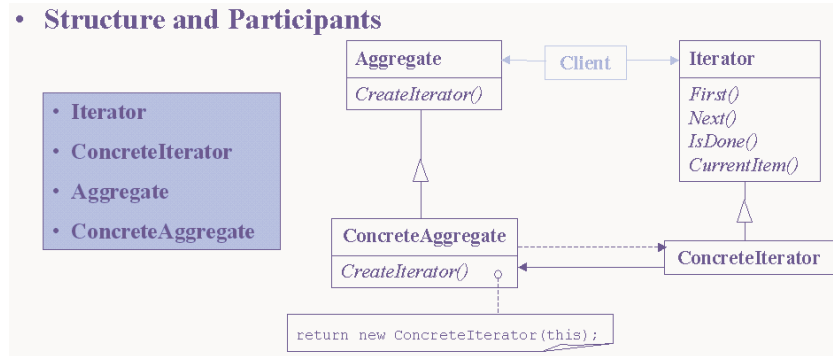
The Interpreter works best when:

- the grammar is simple.
- efficiency is not a critical concern.

## Iterator

**Intent** - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

• **Structure and Participants**



## Iterator

► **Applicability - use when you want to:**

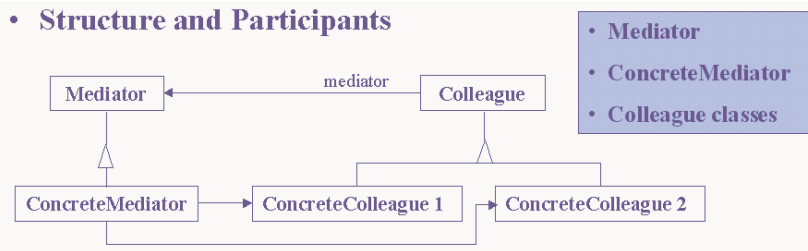
- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).



## Mediator

**Intent** - Define an object that encapsulates how a set of objects interact. Promote loose coupling and let vary the interaction between objects independently.

• **Structure and Participants**



## Mediator

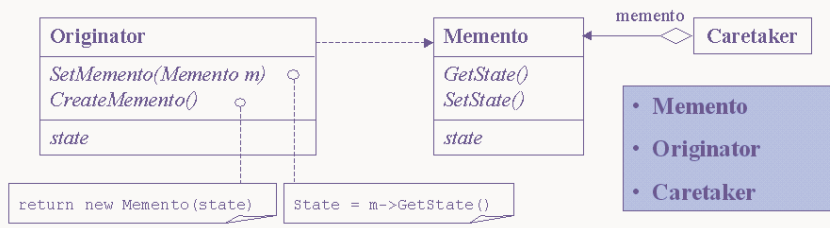
► **Applicability** - use when you want to:

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should be customizable without a lot of subclassing.

## Memento

**Intent** - Without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later.

• **Structure and Participants**



## Memento

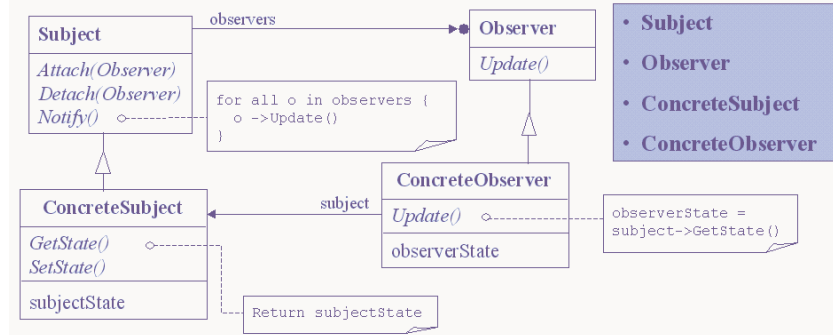
► **Applicability - use when you want to:**

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

## Observer

**Intent** - Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

• **Structure and Participants**



- Subject
- Observer
- ConcreteSubject
- ConcreteObserver

## Observer

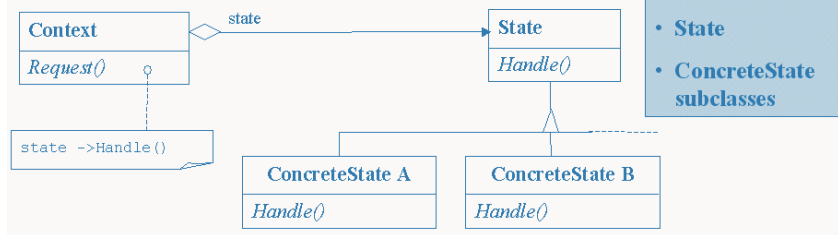
► **Applicability** - use when you want to:

- an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- a change to one object requires changing others, and you don't know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who these objects are.

## State

**Intent** - Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

### • Structure and Participants



## State

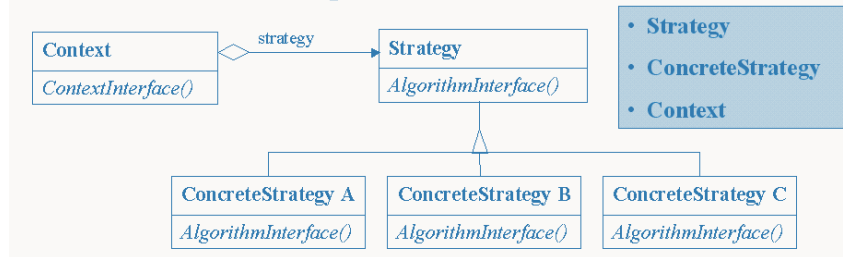
### ► Applicability - use when you want to:

- an object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- operations have large, multipart conditional statements that depend on the object's state.

## Strategy

**Intent** - Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

• **Structure and Participants**



## Strategy

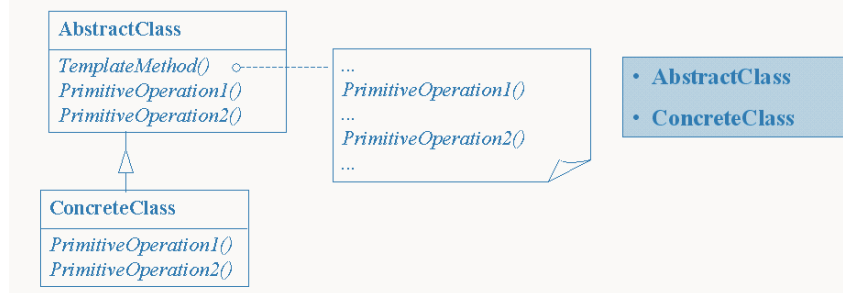
► **Applicability - use when you want to:**

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviours.
- you need different variants of an algorithm.
- an algorithm uses data that clients shouldn't know about. Use strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviours, and these appear as multiple conditional statements in its operations.

## Template Method

**Intent** - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of algorithm without changing the algorithm's structure.

• **Structure and Participants**



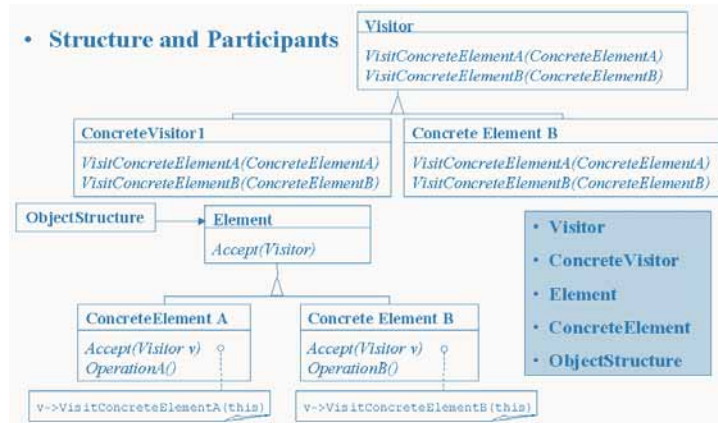
## Template Method

► **Applicability - use when you want to:**

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary.
- when common behaviour among subclasses should be factored and localized in a common class to avoid code duplication.
- to control subclasses extensions.

## Visitor

**Intent** - Lets you define a new operation to be performed on the elements of an object structure, without changing the classes of the elements on which it operates.



## Visitor

► **Applicability** - use when you want to:

- an object structure contains many classes with different interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" classes with these operations.
- the class defining object structure rarely change, but you often want to define new operations over the structure.