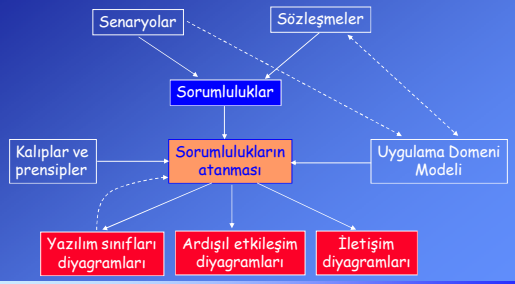
## Senaryoların Gerçeklenmesi (*Use-Case Realization*)

Bu bölümde; senaryoların birbirleriyle etkileşimde olan (işbirliği yapan) yazılım sınıfları ve nesneler şeklinde nasıl tasarlanacağı ele alınacaktır.

Bu aşamada senaryolardan ve uygulama domeninin modelinden yola çıkılır.

Karmaşık işlemlerin yer aldığı sistemlerde sözleşmelerden de yararlanılır.

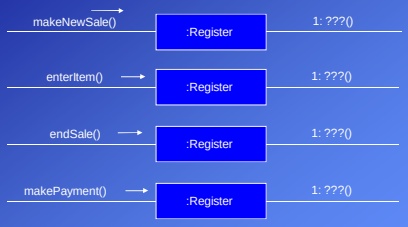







**İletişim diyagramları ve ardışıl diyagramların kullanımı:**

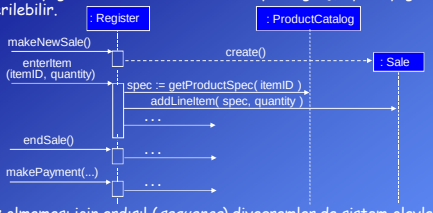
İletişim (*communication*) diyagramları kullanılması durumunda her sistem olayı (giriş) için ayrı bir diyagram çizmek gerekir.



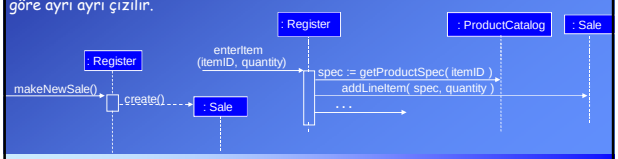





Ardışıl (*sequence*) diyagramlarda ise tüm sistem olayları (giriş) aynı diyagram üzerinde gösterilebilir.




Şekillerin karışık olmaması için ardışıl (*sequence*) diyagramlar da sistem olaylarına göre ayrı ayrı çizilir.








## Tasarım Örnekleri

**Satışın Başlatılması:**
Bu örnekte makeNewSale işlemine ilişkin sözleşmeden (*contract*) yararlanılarak sorumluluklar belirlenecek ve bu sorumluluklar GRASP kalıplarının yardımıyla uygun sınıflara atanacaktır.

**Contract CO1: makeNewSale**

**Operation:** makeNewSale()
**Cross References:** Use Cases: Process Sale
**Preconditions:** none

**Postconditions:**
- A Sale instance s was created (instance creation).
- s was associated with the Register (Association formed).
- Attributes of s were initialized.

Sözleşmenin son koşulları incelenerek sorumluklar belirlenir:
- Sale sınıfından s nesnesinin yaratılması,
- s'nin uygun Register nesnesi ile bağlanması,
- s'nin başlangıç koşullarının sağlanması.

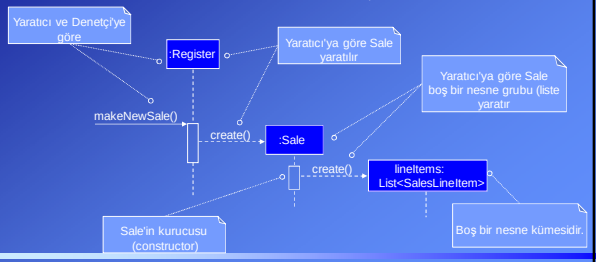
Bu sorumlulukların atanacağı sınıflar daha önce elde edilmiş olan yazılım sınıfları arasında aranır. Eğer uygun bir yazılım sınıf yoksa analiz modelindeki kavramsal sınıflar taranır.

Bizim örneğimizde tasarıma yeni başladığımız varsayılırsa elimizde hiç yazılım sınıfı olmayacaktır. Bu durumda analiz modelindeki kavramsal sınıflar bize kaynak olacaktır.





Örnek sistemimizde bütün girişler terminale yapıldığından Register sınıfı denetçi (*controller*) olarak atanmaya uygundur.
Analiz modeli incelendiğinde satış ile ilgili bilgiler de terminalden geçtiği ve satışı terminalin yoğun olarak kullandığı görülür. Yaratıcı kalıbına göre Sale nesnelerini Register sınıfının yaratması uygun olacaktır.
Bu işlem sonucunda zaten Sale nesnesi Register nesnesine bağlanmış olacaktır.
Analiz modeline göre Sale nesnesi satış kalemleri içermektedir. Bu nedenle başlangıç işlemi olarak satış kalemlerini içerebilen boş bir liste yaratılır.








**Örnek:** enterItem

**Contract CO2: enterItem**

**Operation:** enterItem(itemID: ItemID, quantity: integer)
**Cross References:** Use Cases: Process Sale
**Preconditions:** There is a sale underway

**Postconditions:**
- A SalesLineItem instance sli was created (instance creation).
- sli was associated with the current Sale (association formed).
- sli.quantity became quantity (attribute modification)
- sli was associated with a ProductSpec. based on itemID match (association formed)

- Denetçi nesne: Tüm sistem için görüntü denetçi olarak Register kullanılır.
- SalesLineItem yaratma: Uygulama domeni modeli incelendiğinde Sale sınıfının SalesLineItem sınıflarını içerdiği görülür. Sale, SalesLineItem yaratır.
- SalesLineItem'da yer alan quantity, Register tarafından Sale'e gönderilmeli. SalesLineItem yaratılırken (*constructor*) bu değer parametre olarak kullanılır.
  Register'dan Sale'e makeLineItem mesajı gitmeli. Bu mesaj gerekli parametreleri içermeli.
- SalesLineItem nesnesi uygun ProductSpecification ile ilişkilendirilmeli.
  Belli bir itemID'ye bağlı olarak ProductSpecification'ı bulmak kimin sorumluluğudur? ProductCatalog, ProductSpecification'ları içerir. getSpecification metoduna sahip olmalı.

Burada ortaya bir problem çıkmaktadır:

Müşterinin satın aldığı ürünün kodu (bar kod numarası) terminal üzerinden sisteme girilmektedir. Bu itemID' ye bakarak o ürünün hangisi olduğunu (ProductSpecification) bulmak kimin sorumluluğudur?

Ürünlerle ilgili bilgilerin (tanım, fiyat) yer aldığı nesneler (ProductSpecification) bir katalog nesnesinde (ProductCatalog) tutulmaktadır.

Register ve ProductCatalog nesnelerinin sistemin başlangıcında birlikte yaratıldıkları varsayılabilir. Bu durumda kataloğa getProductSpecification mesajını gönderip o ürüne ait ProductSpecification nesnesini almak sorumluluğu Register nesnesinde olabilir.

Önce Register nesnesine enterItem(id, qty) mesajıyla satın alınan ürünün kodu ve miktarı girilmektedir.

Register nesnesi ProductCatalog nesnesine getProductSpecification(id) mesajıyla ilgili ürünün tanımlayıcı nesnesini sormaktadır.

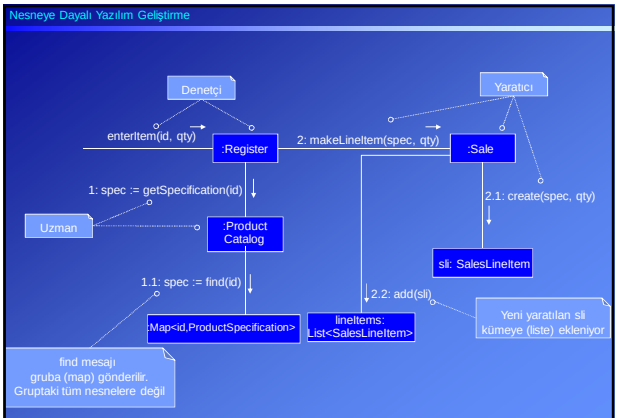
ProductCatalog nesnesi sahip olduğu nesne grubuna (Map) find(id) mesajını göndererek ilgili ürünün tanımlayıcı nesnesini arar ve bulunan tanımlayıcıyı Register nesnesine iletir. Böylece kod numarası sisteme girilmiş olan ürünün ne olduğu belirlenmiş olur.

Kataloğu tarama sorumluluğu Sale nesnesine de verilebilirdi ancak bu durumda Sale nesnesi ProductCatalog nesnesine bağlanmış olurdu.

Ürünün tanımlayıcı bilgisini alan Register nesnesi bu bilgiyi ve miktarı Sale nesnesi makeLineItem(spec, qty) mesajıyla iletir. :Sale nesnesi bu bilgileri içeren bir sl:SalesLineItem nesnesi yaratır ve bu nesneyi kendi satış kalemlerini tuttuğu listeye ekler.



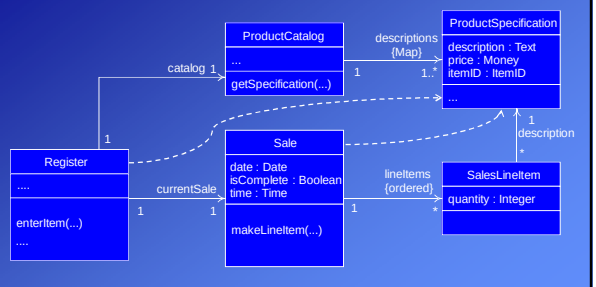









Etkileşim diyagramları ile birlikte yazılım sınıfları diyagramları da oluşturulur.
Bu diyagramlar ile ilgili daha fazla bilgi ileriki bölümlerde verilecektir.





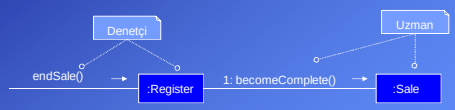



**Örnek:** endSale

**Contract CO3: endSale**

| | |
|---|---|
| **Operation:** | endSale() |
| **Cross References:** | Use Cases: Process Sale |
| **PreConditions:** | There is a sale underway |
| **PostConditions:** | - Sale.isComplete became true (attribute modification) |

•Denetçi nesne: Tüm sistem için görüntü denetçi olarak Register kullanılıyor.
• isComplete değişkenini true yapmak kimin sorumluluğudur?
Uzman kalıbına göre Sale'in sorumluluğudur.








**Satışın toplam bedelinin hesaplanması:**
Önceki tasarım örneklerinde sorumluluklar sözleşmelerden elde edilmişti. Bu örnekte ise sorumluluk senaryolardan elde edilecektir.

**Main Success Scenario (or Basic Flow):**
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
*Cashier repeats steps 3-4 until indicates done.*
5. System presents **total** with taxes calculated.

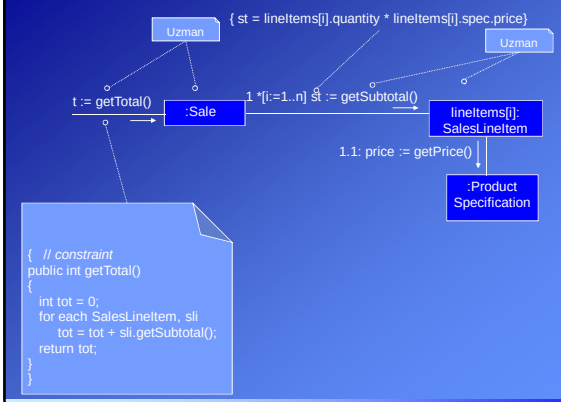
Senaryoya göre satışın toplam bedelinin hesaplanması gerekiyor.
Uzman kalıbına göre gerekli bilgilere Sale ulaşabilir.
Toplam bedel = tüm satış kalemlerinin toplamı
Satış kalemi bedeli = ürün miktarı * ürün birim fiyatı

Gerekli bilgiler ve uzmanları:

| Bilgi | Uzman |
|---|---|
| ProductSpecification.Price | ProductSpecification |
| SalesLineItem.quantity | SalesLineItem |
| Bir satıştaki tüm kalemler | Sale |



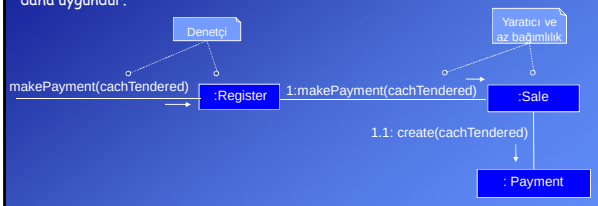




**Ödeme nesnesinin yaratılması:**

GRASP kalıplarından Uzman ve Yaratıcı'ya göre Ödeme (Payment) nesnesini yaratmak için iki aday vardır: Register ve Sale.

Az bağımlılık (*Low coupling*) kalıbına göre Ödeme'nin Satış tarafından yaratılması daha uygundur.




Birden fazla seçenek olduğunda uyum ve bağımlılık unsurlarını dikkate almak yazılımın daha sağlam olmasını ve tekrar kullanılabilmesini sağlayacaktır.





**Başlangıç İşlemleri:**

Sistemlerin ilk çalışmaya başladıklarında olması gerekenler de ayrı bir senaryo grubu (*use-case*) olarak yazılabilir.

Başlangıç aşamasında yapılacak işleri tasarım en son aşamasında belirlemek uygundur.

Nesneye dayalı yöntemde bir başlangıç nesnesi (*initial domain object*) belirlenir.

Bu nesne program çalışmaya başladığında yaratılacak ilk nesnedir. Başlangıç nesnesi, doğrudan içerdiği **diğer nesneleri yaratma** ve aralarındaki **bağlantıyı (görünürlüğü) sağlama** sorumluluğunu üstlenecektir.

Başlangıç nesnesinin yaratılma yeri programlama diline göre değişebilir:

```
public class Main                          // Java
{
public static main( String[] args)
{
  // Store is initial domain object
  Store store = new Store();
  Register register = store.getRegister();        // register Store'da yaratılıyor
  ProcessSaleJFrame frame = new ProcessSaleJFrame(register);  // Frame ile register
  .....                                                       // bağlandı
 }
}
```





**Para üstünün hesabı:**

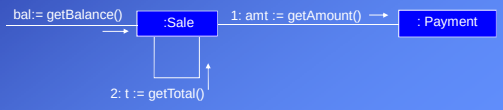
Process Sale senaryo grubuna göre para üstünün hesaplanması ve gösterilmesi gerekiyor.

Biz arayüz katmanı ile ilgilenmediğimizden para üstünün ekranda nasıl gösterildiğinin değil, nasıl hesaplandığı üzerinde duracağız.

Para üstünü hesaplamak için toplam satış bedeline ve müşterinin ödediği miktar bilgisine gerek vardır. Uzman kalıbına göre para üstünü hesaplayacak bilgilere Satış (Sale) ve Ödeme (Payment) sınıfları sahip olabilir.

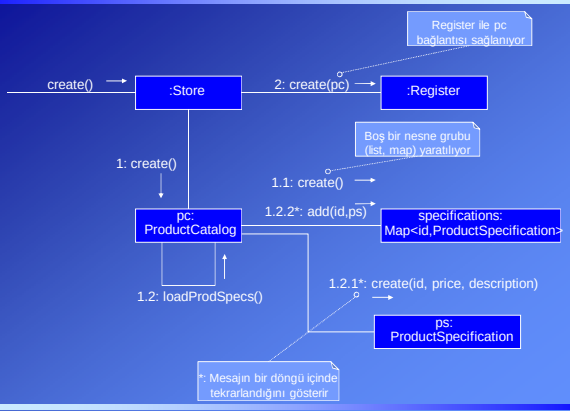
Eğer bu sorumluluk Payment sınıfına verilirse toplam bedeli öğrenmek için Sale sınıfına başvurmasını gerektirir. Bu da önceden olmayan yeni bir bağımlılık yaratır.

Sorumluluk Sale sınıfına verilirse o da Payment sınıfına gerek duyar. Sale sınıfı Payment nesnelerinin yaratıcısı olarak zaten bu sınıf hakkında bilgi sahibi olmak zorundadır. Bu durumda yeni bir bağımlılık yaratılmış olmaz.












**Satış kayıtlarının tutulması:**

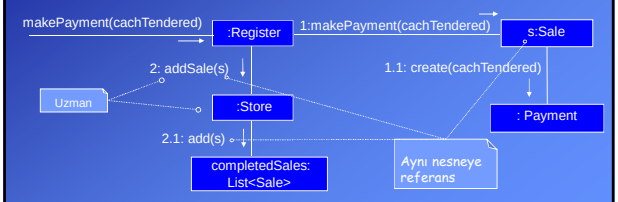
Sonlandırılan satışları bilmek ve kayıtlarını tutmak kimin sorumluluğundadır.

Uygulama modeli incelendiğinde satışlarla ilgili bilgilere sahip olan Dükkan (Store) sınıfıdır.

Eğer Store sınıfı başka sorumluluklar ile fazla yüklenmediyse bu sorumluluğu alabilir.

Eğer Store çok yüklü ise satışların kayıtlarını tutmak için SatışDefteri (SalesLedger) adlı yeni bir sınıf düşünülebilir. Hatta bu sınıf geri dönülerek uygulama modeline de eklenebilir.

Biz örneğimizde sorumluluğu Store sınıfına verdiğimizi varsayalım:





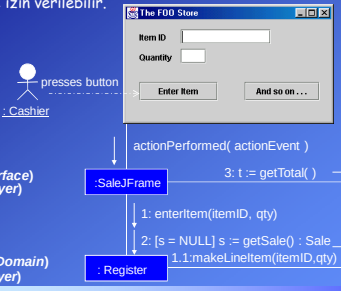



**Arayüz ile uygulama arasındaki bağlantının sağlanması:**

Başlangıç aşamasında arayüz nesnelerine denetçi nesnelerin referansları gönderilir.

Denetçi kullanılması katmanların ayrılmasını sağlar. Ancak tüm mesajların denetçi üzerinde geçmesi yavaşlamaya neden olabilir

Bu nedenle bazı durumlarda arayüz nesnelerinin iş katmanındaki nesnelere doğrudan erişmesine izin verilebilir.




Yandaki örnekte arayüz nesnesi (SaleJFrame) Sale nesnelerine bağımlı olmuştur.

Sale sınıfının kararlı (çok değişmeyen) bir sınıf olduğu varsayılırsa bu bağımlılık problem yaratmayacaktır.

## Görünürlük (*Visibility*)

B nesnesinin A nesnesine görünür olması, A'nın B'ye erişebilmesi anlamına gelir.

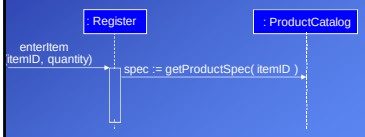
Görünürlük Türleri:

- Nitelik Görünürlüğü (*Attribute visibility*): B, A'nın bir niteliğidir (üyesidir).
- Parametre Görünürlüğü (Parameter visibility): B, A'nın bir metodunun parametresidir.
- Yerel Görünürlük (*Local visibility*): B, A'nın bir metodunda yerel değişkendir.
- Global Görünürlük (*Global visibility*): B, A'nın global uzayındadır.

A nesnesinin B nesnesine mesaj gönderebilmesi için B A'ya görünür olmalıdır.

**Nitelik Görünürlüğü:** Bir nesne diğerinin üyesi olduğundan, görünürlük nesnenin yaşam süresince devam eder.

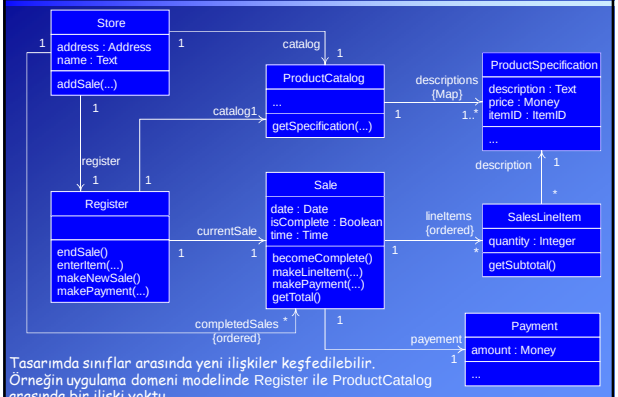
```
class Register
{
  .....
   private ProductCatalog catalog;
  .....
} // atama Kurucu metodta
```




```
public void enterItem(itemId,qty)
{
  .....
   spec= catalog.getProductSpec(itemID);
  .....
}
```





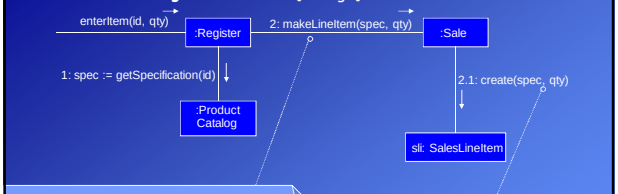



Tasarımda sınıflar arasında yeni ilişkiler keşfedilebilir. Örneğin uygulama domeni modelinde Register ile ProductCatalog arasında bir ilişki yoktu.





**Parametre Görünürlüğü:** Bir metodun içinde geçerlidir.




```
{
makeLineItem(ProductSpecification spec, int qty)
{
  ...
  sli = new SalesLineItem(spec,qty);
  ...
}
}
```

```
{  // SalesLineItem kurucu metodu
SalesLineItem(ProductSpecification spec, int qty)
{
  ...
  description = spec;  // parametreden niteliğe
  ...
}
}
```





**Bağımlılık İlişkisi (*Dependincy Relationship*):**

UML'de bağımlılık en genel haliyle, bir birimin (sınıf, senaryo) başka bir birim ile ilgili bilgilere sahip olması gerekliliğini ifade eder.

Tasarım aşamasında ise bağımlılık, yazılım sınıfları arasındaki nitelik görünürlüğü dışındaki görünürlükleri ifade eder.

Örneğin; yansı 5.8'de Register nesnesi ProductCatalog nesnesine getSpecification mesajını gönderdiğinde ProductSpecification tipinden bir yanıt alır. Bu yanıt enterItem metodunda bir yerel değişken olarak saklanır. Bu nedenle Register nesnesinin ProductSpecification nesnesine bir bağımlılığı vardır (yerel görünürlük).

Ayrıca Sale nesnesi makeLineItem mesajının içinde ProductSpecification tipinden bir parametre alır. Bu nedenle Sale ProductSpecification'a bağımlıdır (parametre görünürlüğü).

UML diyagramlarında bağımlılık kesik çizgili bir ok ile ifade edilir.





## Tasarım (Yazılım) Sınıfı Diyagramları (*Desgin Class Diagrams-DCD*)

Tasarım aşamasında etkileşim diyagramları oluşturulurken buna paralel olarak yazılım sınıflarını ifade eden UML sınıf diyagramları da çizilir.

Bu diyagramlarda; yazılım sınıflarının nitelikleri, niteliklerin tipleri, metotların parametreleri ve üyelerin erişim hakları büyük ölçüde belirtilir.

Ayrıca yazılım sınıfları arasındaki ilişkiler ve bağımlılıklar yönlü olarak gösterilir.

**Yazılım sınıfları arasındaki bağlantılar (*Navigability*):**

İki yazılım sınıfı arasında doğrudan bir bağlantı olması çoğunlukla bir nitelik görünürlüğünün sonucudur.

B nesnesi A nesnesinin üyesidir ve A, B'ye mesaj göndermektedir.

Sınıf diyagramlarında bunu ifade etmek için A sınıfından B sınıfına bir ok çizilir.

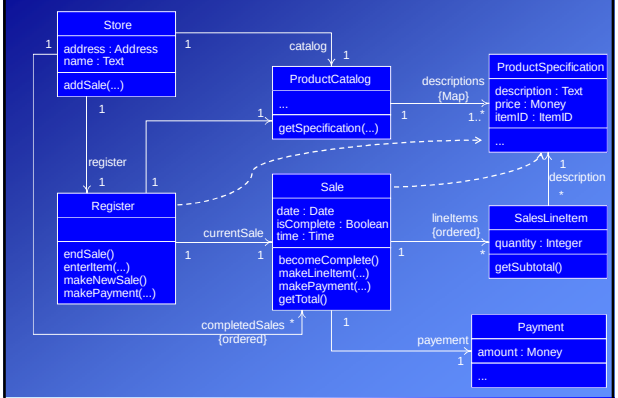
Uygulama domenindeki bağlantılar oluşturulurken iki kavramsal sınıf arasında gerçek dünyada bir ilişki olup olmadığı araştırılmıştı.

Tasarım aşamasındaki sınıf diyagramlarında ise gerçekten iki yazılım sınıfı arasında bir görünürlük ilişkisi olup olmadığı araştırılır.

Yazılım sınıfları arasındaki bağlantılar (*navigability*) iletişim diyagramlarının incelenmesiyle bulunabilir. Örneğin 5.5, 5.8, 5.12'deki diyagramlar incelenebilir.







Üyelerin veri tipleri ve erişim hakları belli olduğu ölçüde sınıf diyagramlarında gösterilir.