Tasarım Modelinin (*Design Model*) Oluşturulması

Bu aşamada, nesneye dayalı yönteme göre problemin mantıksal çözümü oluşturulur.

Tasarım modelinde yazılım sınıfları ve aralarındaki işbirliği (etkileşim) belirlenir.

Bu model iki tip UML diyagramı ile ifade edilir:

- Nesneler arası etkileşimi gösteren **etkileşim diyagramları** (***interaction diagrams***)
- Yazılım sınıflarını ve aralarındaki bağlantıları gösteren **sınıf diyagramları**

Etkileşim diyagramlarının çizilmesi, nesnelerin davranışlarının belirlenmiş olduğu anlamına gelir.
Bunu yapabilmek için tasarımda nesnelere gerekli **sorumlulukları**n atanması (*assignment of **responsibilities***) gerekir.

Nesnel tasarımı (*object design*) gerçekleştirebilmek için iki ana konuda bilgiye gerek duyulur:

- Sorumluluk atama prensipleri
- Tasarım kalıpları (*design patterns*)

Sınıfların Sorumlulukları

Nesnel Tasarımın (*Object Design*) genel ifadesi:
İsteklerin belirlenmesi ve uygulama uzayı modelinin oluşturulmasından sonra,
- yazılım sınıflarının belirlenmesi,
- yazılım sınıflarına metotların eklenmesi (sorumlulukların atanması) ve
- sorumlulukları yerine getirmek üzere nesneler arası mesajların belirlenmesidir.

Nesnel tasarımın temeli nesnelere **sorumluluklar**ın atanmasıdır.

Nesnelerin sorumlulukları 2 gruba ayrılır:
- Bilinmesi gerekenler
  - Kendi özel verileri
  - İlgili diğer nesneler
  - Üzerinde hesap yapabileceği, hesapla elde edebileceği bilgiler
- Yapılması gerekenler
  - Hesap yapma, nesne yaratma/yok etme
  - Başka nesneleri harekete geçirme
  - Başka nesnelerin hareketlerini denetleme

Sorumlulukları yerine getirmek için metotlar oluşturulur.

Bir sorumluluğu yerine getirmek için bir metot başka nesnelerdeki metotlarla iş birliği yapabilir.

Bir sistemdeki sorumluluklar o sistem için yazılmış olan senaryolardan (*use-case*) ve sözleşmelerden (*contract*) elde edilir.

Tasarım Kalıpları (*Design Patterns*)

Yazılım sınıflarının belirlenmesinde ve onlara uygun sorumlulukların atanmasında tasarım kalıplarından yararlanılacaktır.

Tasarım kalıplarının varlığı ilk olarak bir mimar olan Christopher Alexander[1,2] tarafından ortaya konulmuştur.

Şu soruları sormuştur:

Kalite, kişiye göre değişmeyen ve ölçülebilen objektif bir kavram mıdır?

İyi (kaliteli) tasarımlarda var olan ve kötü tasarımlarda olmayan nedir?

Benzer problemleri çözmek için oluşturulan ve beğenilen (kaliteli) mimari yapılarda ortak özellikler (benzerlikler) olduğunu belirlemiştir. Bu benzerliklere kalıplar (*patterns*) adını vermiştir.

Her kalıp gerçek dünyada defalarca karşılaşılan bir problemi ve o problemin çözümünde izlenmesi gereken temel yolu tarif etmektedir.

Türkçesi; *Aklın yolu birdir.*

Bir problemle karşılaşan tasarımcı eğer daha önce benzer problemle karşılaşan tasarımcının uyguladığı başarılı çözümü biliyorsa (kalıp) her şeyi yeniden keşfetmek yerine aynı çözümü tekrar uygulayabilir.

[1] Alexander, C., Ishikawa, S., Silverstein, M., *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[2] Alexander,C., *The Timeless Way of Building*, Oxford University Press, 1979.

Mimarlıkta olduğu gibi yazılım geliştirmede de benzer problemlerle defalarca karşılaşılmaktadır.

Yazılımcılar deneyimleri sonucunda bir çok problemin çözümünde uygulanabilecek prensipler ve deneyimler (kalıplar) oluşturmuşlardır.

Bu konudaki ilk önemli yayın dört yazar tarafından hazırlanan bir kitap olmuştur:
Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns*, Reading MA, Addison-Wesley, 1995. (Ekim 1994'te yayımlandı)

Bu yazarlara dörtlü çete (*gang of four*) adı takılmış ve ortaya koydukları kalıplar GoF kalıpları olarak adlandırılmıştır. Dersin ilerleyen bölümlerinde GoF kalıpları ele alınacaktır.

Bu bölümde anlaşılması daha kolay olan ve C.Larman tarafından oluşturulan **GRASP** kalıpları ele alınacaktır.

GRASP: Genel Sorumluluk Atama Kalıpları

GRASP (*General Responsibility Assignment Software Patterns*) nesnelere sorumluluk atamada yol gösteren temel kalıpların genel adıdır.
Kalıplar 3 bölümden oluşur: İsim, Problem, Çözüm.

Bu üç temel bölümün dışında kalıplarda açıklayıcı ve yardımcı ek bölümlerde bulunabilir.

1. Bilginin Uzmanı (*Information Expert or Expert*)

Problem: Sınıflara sorumlulukları atamanın temel prensibi nedir?
Çözüm: Bir sorumluluğu bilginin uzmanına, yani onu yerine getirecek veriye (bilgiye) sahip olan sınıfa atayın.

Örnek:

POS sisteminde satışın toplam bedelinin bilinmesine gerek vardır.

Sorumlulukları atamaya başlamadan önce sorumlulukların açıkça tanımlanması gerekir.

Dersin "Tasarım" bölümünde gösterileceği gibi sorumluluklar, sözleşme ve senaryolardan elde edilir.

Örnek: *Satışın toplam bedelinin belirlenmesinden kim sorumludur?*

Uzman kalıbına göre bu bilgiye sahip olan sınıf aranacak.

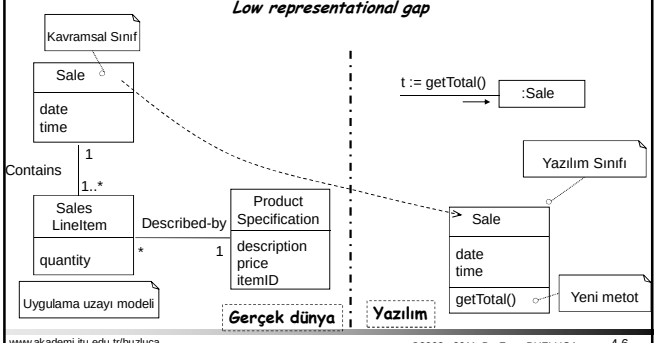
Arama önce yazılım sınıfları arasında yapılır.

Eğer henüz böyle bir yazılım sınıfı oluşturulmamışsa uygulama modelindeki kavramsal sınıflar incelenir. Bunlardan uygun olan tasarım modeline yazılım sınıfı olarak getirilir.

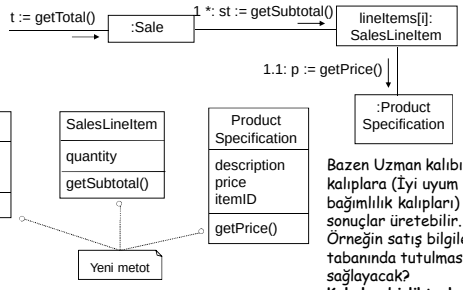
Örneğimizde tasarıma henüz yeni başlandığını varsayarsak elimizde yazılım sınıfı olmadığından uygulama uzayındaki kavramsal sınıflar incelenecektir.

Burada Sale kavramsal sınıfı bu sorumluluğu almak için uygun görünmektedir. Bu kavramsal sınıftan aynı isimde bir yazılım sınıfı oluşturulabilir.

Low representational gap



Sale sınıfı tek başına toplam bedeli belirleyemez.
Satış kalemlerinin bedellerine gerek vardır. Bunun için her satış kaleminin bedeli SalesLineItem sınıfından alınacaktır.
SalesLineItem bir kalem bedelini belirleyebilmek için miktar (adet) bilgisine sahiptir. Bir ürünün birim fiyatını ProductSpecification sınıfından alacaktır.



Bazen Uzman kalıbı, diğer kalıplara (İyi uyum ve az bağımlılık kalıpları) aykırı sonuçlar üretebilir.
Örneğin satış bilgilerinin veri tabanında tutulmasını kim sağlayacak?
Kalıplar birlikte kullanılmalı.

2. Yaratıcı (*Creator*)

Bir nesnenin yaratılma sorumluluğunun kime (hangi sınıfa) verileceği konusunda yaratıcı kalıbı yol gösterir.

Problem: Bir sınıftan nesne yaratma sorumluluğu kime aittir?

Çözüm: Aşağıdaki koşullardan biri geçerli ise B sınıfına A sınıfından nesne yaratma sorumluluğunu atayın.

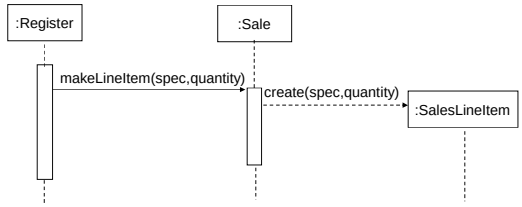
- B, A nesnelerini içeriyorsa.
- B, A nesnelerinin kaydını tutuyorsa.
- B, A nesnelerini yoğun olarak (*closely*) kullanıyorsa.
- A nesnelerinin yaratılması aşamasında kullanılacak olan başlangıç verilerine B sahipse (B, A'nın yaratılması açısından bilgi uzmanıdır.)
- Bu koşulları sağlayan birden fazla sınıf varsa öncelik yaratılacak olan nesneyi içeren sınıfa verilmelidir.

Örnek:

Satış kalemlerini (SalesLineItem) kim yaratacak?

Yansı 4.21'deki uygulama modeli incelendiğinde bir satışın bir çok satış kalemi içerdiği görülür.

Buna göre satış kalemlerini (SalesLineItem) yaratmak satışın (Sale) sorumluluğu olacaktır.



3. Az Bağımlılık (*Low Coupling*)

Bir sınıfın bağımlılığı; kendi işleri için başka sınıfları ne kadar kullandığı, başka sınıflar hakkında ne kadar bilgi içerdiği ile ilgilidir.

Fazla bağımlılık tercih edilmez, çünkü

- Bir sınıftaki değişim diğer sınıfları da etkiler.
- Sınıfları bir birlerinden ayrı olarak anlamak zordur.
- Sınıfları tekrar kullanmak zordur.

Nesneye dayalı programlarda SınıfX'in SınıfY'ye bağımlılığı aşağıdaki durumlarda ortaya çıkar:

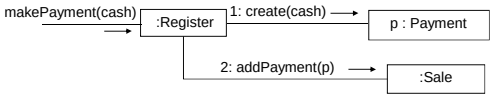
- SınıfX'in içinde SınıfY cinsinden bir üye (referans ya da nesne) vardır.
- SınıfX'in nesneleri SınıfY'nin nesnelerinin metotlarını çağırıyor.
- SınıfX'in bir metodu parametre olarak SınıfY tipinden veriler (referans ya da nesne) alıyor/döndürüyor.
- SınıfX'in bir metodu SınıfY tipinden bir yerel değişkene sahip.
- SınıfX, doğrudan ya da dolaylı olarak SınıfY'den türetilmiştir (altsınıfıdır).

Kalıp:

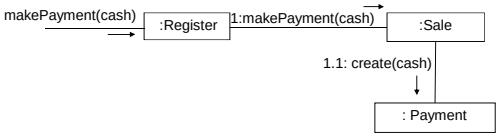
Problem: Diğer sınıfların değişimlerinden etkilenmeme, tekrar kullanılabilirlik nasıl sağlanır?
Çözüm: Sorumlulukları, sınıflar arası bağımlılık az olacak şekilde atayın.

Örnek:

POS sisteminde bir ödeme nesnesinin yaratılıp satış nesnesi ile ilişkilendirilmesi gerekiyor.
Gerçek dünyada ödeme POS terminaline yapıldığından gerekli bilgilere sahip olan odur. Yaratıcı kalıbına göre Register nesnesi Payment nesnesini yaratacaktır.



Bu durumda Register sınıfının Payment sınıfı hakkında bilgiye sahip olması gerekir.
Aynı sorumluluk aşağıdaki şekilde Sale sınıfına atanırsa bağımlılık azaltılmış olur.



4. Denetçi (*Controller*)

Denetçi kalıbı model ile görüntünün ayrılığı (***Model-View Separation Principle***) prensibine dayanır.

"Model-View Separation Principle":

- İş katmanı nesnelerini (*business layer objects*) doğrudan kullanıcı ara yüzü katmanı (UI) nesnelerine doğrudan bağlamayın.
- Uygulama işlerini (örneğin vergi hesabı) ara yüz (UI) nesnelerinin içine koymayın.

Ara yüz (UI) nesneleri sadece giriş/çıkış birimleri ile ilgili işleri yapmalı. Örneğin fare ile bir seçim yapıldığında (olay olduğunda) bu olay (gerekli bilgilerle birlikte) iş katmanına iletilmeli.

"Model-View Separation" prensibinin yararları:

- Uygulama modeli ile ara yüz ayrı ayrı oluşturulup geliştirilebilir.
- Ara yüzde oluşan değişikliklerin iş katmanını etkilemesi en aza indirgenir..
- Tek bir iş modeli için birden fazla görüntü yaratılabilir.
- İş katmanındaki işler ara yüz olmadan da yürütülebilir.
- İş katmanı kolaylıkla başka bir projeye (yazılıma) taşınabilir.

Denetçi kalıbının açıklaması:

Sistem olayları (*system event*), dış aktörler tarafından üretilen ve sistem işlemleri (*system operations*) ile ilişkili olaylardır.

Örneğin POS sisteminde kasa görevlisi "End Sale" butonuna bastığında bir sistem olayı yaratmış olur.

Problem: Sistem olaylarını arayüz katmanından alıp ilgili nesnelere yönlendirmekle kim sorumludur?

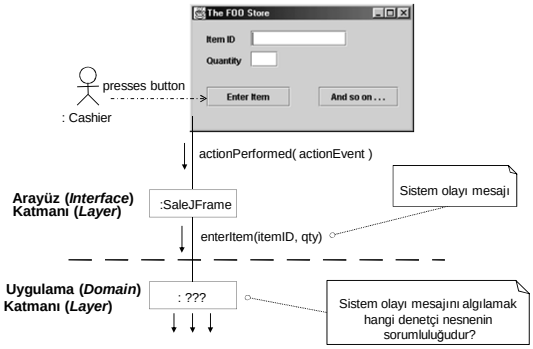
Çözüm: Sistem olaylarını algılama ve değerlendirme sorumluluğunu alacak sınıfı aşağıdaki iki seçenekten birini kullanarak oluşturun:

- Tüm sistemi, cihazı veya alt sistemi temsil eden bir sınıf (*facade controller*). **Görüntü denetçi**
- Bir kullanım senaryosunu temsil eden sınıf (*use-case or session controller*). **Senaryo** veya **oturum denetçisi**

Genellikle; <UseCaseName>Handler, <UseCaseName>Coordinator, <UseCaseName>Session şeklinde isimlendirilirler.

Not: "Window", "Applet", "Frame" gibi sınıflar bu gruba girmezler. Bunlar sadece olaylarla ilgili mesajları denetçi nesneye aktarırlar.

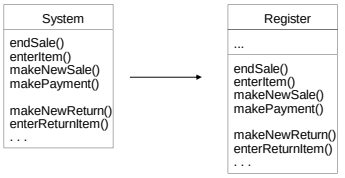
Denetçi nesneleri sistem olaylarını algıladıktan ve bazı kontrol işlerini yaptıktan sonra çoğunlukla bu olayları, ilgili işlemleri yapacak olan nesnelere yönlendirirler.



Denetçi arayüzden gelen mesajları alacak, gerekli kontrolleri yapacak (örneğin sıraları doru mu?) ve o mesajı asıl işi yapacak olan ilgili nesneye gönderecek.

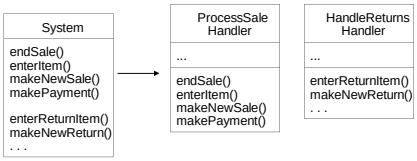
Görüntü denetçi
(*Facade Controller*)

Tüm sistem olaylarını algılamak tek bir denetçinin sorumluluğunda olur.



Oturum denetçisi
(*Session Controller*)

Her oturum (senaryo) için ayrı bir denetçi görevlendirilir.



Denetçi tipinin belirlenmesi:

- Eğer bir sistemdeki olaylar fazla değilse tüm sistem için tek bir denetçi seçilmesi uygun olur (*facade controller*).
- Eğer olay sayısı denetçinin uyumluluğunu bozacak kadar fazla ise senaryolara ya da cihazlara farklı denetçiler atanması uygun olur.
- Bir senaryo içinde sistem olaylarının üzerinde çeşitli kontrol işlemleri yapılacaksa (örneğin sıraları önemli ise) senaryo denetçisi atamak daha uygundur.
- Denetçiler (tipi nasıl olursa olsun) sistem olayları ile ilgili işleri çoğunlukla kendileri yapmazlar, bu olayları uygun mesajlar ile ilgili nesnelere aktarırlar.

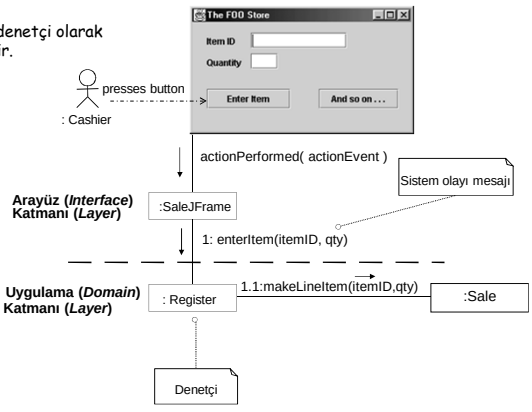
Yararları:

- Arayüz ile uygulama katmanları ayrılmış olur. Bu programın arayüzden bağımsız olmasını ve tekrar kullanılabilmesini sağlar.

Denetçi nesnenin görevini arayüzdeki nesneler de üstlenebilir, ama bu tekrar kullanılabilirliliği ve esnekliği ortadan kaldırdığı için tercih edilmez.

- Bir senaryo kapsamında gerçekleştirilecek sistem işlemlerinin sırası denetim altında tutulur. Örneğin "makePayment" mesajının "endSale" mesajından önce gelmesi önlenir.

Örnek:
Register denetçi olarak seçilmiştir.



5. İyi Uyum (*High Cohesion*)

Eğer bir sınıf birbiri ile ilgili olmayan işler yapıyorsa veya çok fazla iş yapıyorsa o sınıfta uyum kötüdür ve bu durum aşağıdaki sorunlara neden olur:

- Sınıfın anlaşılması zordur.
- Sınıfın bakımını yapmak zordur.
- Sınıfı tekrar kullanmak zordur.
- Değişikliklerden çok etkilenir.

Kalıp:

Problem: Karmaşıklık nasıl idare edilebilir?

Çözüm: Sorumlulukları sınıf içinde iyi bir uyum olacak şekilde atayın.

Bir sınıfın uyumu (işlevsel uyum); ona atanan sorumlulukların birbirleri ile ilgili olmasına ve aynı konuda yoğunlaşmaları ile ilgilidir.

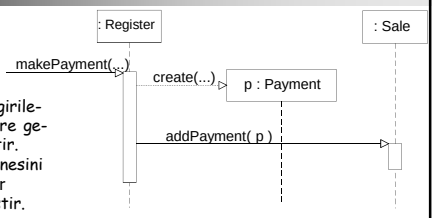
Bazı durumlarda uyum göz ardı edilerek büyük sınıflar yaratılabilir.

Örneğin, veri tabanı işlemlerini tek bir programcının sorumluluğuna vermek için tüm veri tabanı işlemlerinin toplandığı bir sınıf tasarlanabilir.

Diğer bir örnek de dağıtılmış sistemlerdeki nesne kullanımıdır. Uzaktan bağlantıları sık sık yapmamak için bu tür sınıflar mümkün olduğu kadar çok hizmet verecek şekilde tasarlanabilir.

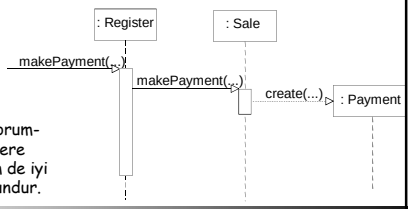
İyi uyum konusunda da ödeme işlemi ve ödeme nesnesinin (Payment) yaratılması kullanılabilir.

Ödeme terminale (Register) girileceğinden yaratıcı kalıbına göre gerekli bilgilere Register sahiptir. Üstteki şekilde Payment nesnesini yaratma sorumluluğu Register sınıfından nesnelere verilmiştir.



Ancak terminal üzerinde yapılan çok işlem varsa bunlarla ilgili tüm sorumlulukların Register sınıfından nesnelere verilmesi bu sınıfın uyumunu bozacaktır.

Bu nedenle Payment yaratma sorumluluğunun Sale sınıfından nesnelere verilmesi hem az bağımlılık hem de iyi uyum kalıbı açısından daha uygundur.



Bu aşamaya kadar karşılaştığımız Tasarım Prensipleri:

- **Low Representational Gap (between real-world and software)**
  Nesneye dayalı programlamanın (temel prensibidir).
  Yazılım sınıflarını oluştururken gerçek dünyadaki kavramlardan esinleniriz.
  Yazılım sınıfları kavramsal sınıflar ile aynı isimlere sahip olurlar.
  Yazılım sınıfları gerçek dünyadaki kavramsal sınıflar ile aynı (benzer) işleri yaparlar.
- **Separation of concerns:** "Concern" bir yazılımın içinde yer alan katmanlar, konular ve işlevlerdir.
  Örneğin UI, veri modeli, iş katmanı.
  Satışın toplam bedelinin hesaplanması, kredi kartı işlemleri, vs.
  Değişik konularla ilgili sorumlulukları aynı sınıfa atamayın.
  Örneğin Sale sınıfı UI, veritabanı ya da envanter işleri yapmamalı..
- **Model-View separation:**
  Bu prensip, "separation of concerns" prensibinin özel bir durumudur.
  Do not connect non-UI objects (business layer objects) directly to UI objects.
  Do not put application logic (such as a tax calculation) in the UI object methods.

Bu aşamaya kadar karşılaştığımız Tasarım Prensipleri : (devamı)

- **Controller** (GRASP)**:** İki katman arasına denetçi nesne koyun.
- **Creator** (GRASP) **:** "x nesnesini kim yaratacak?".
- **Information expert** (GRASP) **:** Bir sorumluluğu, onu yerine getirebilecek bilgiye sahip olan sınıfa atayın.
- **Low Coupling** (GRASP) **:** Sınıflar arası bağımlılık az olsun.
- **High cohesion** (GRASP) **:** Uyumlu bir sınıf az sayıda ve ilgili işler yapan metotlara sahiptir.
- **Modular Design:** Modüler bir sistem uyumlu ve az bağımlı modüllerden oluşur.