2. Iterasyon

İlk iterasyonda örnek POS sistemindeki satış senaryoları grubunun doğal akışı ele alınmıştı. İkinci iterasyonda ise senaryolardaki alternatif akışlar gerçeklenmeye başlanır.

Bazı büyük senaryo grupları (*use case*) bir kaç iterasyon sürebilir. Daha küçükler bir iterasyonda bitebilir. Hatırlatma: Bir iterasyon yaklaşık 4 hafta sürer.

Yeni iterasyonda uygulama domeninin analizi kısa sürebilir, çünkü eklenecek kavramsal sınıfların ve ilişkilerin sayısı azdır.

Eski analiz çizimlerinin kullanılması yerine tersten (kodlamadan ya da tasarım diyagramlarından) gidilerek (*reverse engineering*) problem domeninin diyagramı çıkartılır.

Çünkü ilk analizden sonraki aşamalarda bir çok kararlar verildi, bazı sınıflar elendi ya da yenileri eklendi. Bu nedenle geçerli olan, son varılmış olan tasarım modelidir (ya da kodlamadır).

Dersin bundan sonraki kısmında daha karmaşık problemler ele alınarak bu problemlerin çözümüyle birlikte yeni kalıplar ve prensipler açıklanacaktır.

Bu bölümde ele alınan problemler:

Yansı 7.3'teki senaryoya ilişkin etkileşim diyagramında da görüldüğü gibi tasarlanmakta olan POS sistemi bazı hizmetler için dış birimlerle (aktörlerle) işbirliği yapmak durumundadır.

Bu hizmetler, vergi hesabı, kredi kartı onaylama, muhasebe hesapları olabilir.

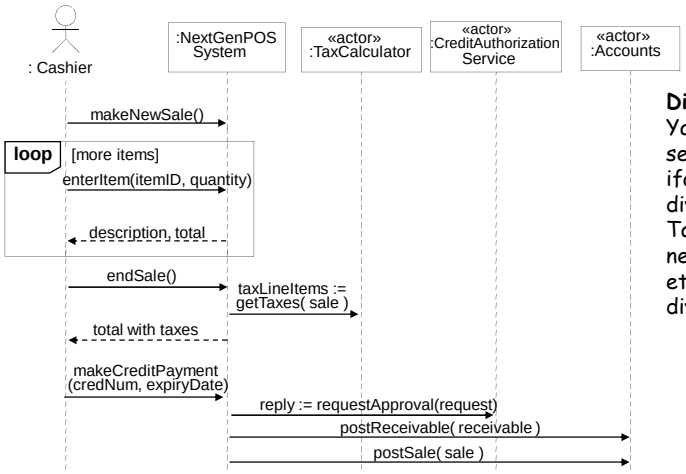
Özellikle bizim denetimimiz dışında olan birimlerin değişme olasılığı yüksektir ve bu değişimler bizim tasarlamakta olduğumuz POS sistemini etkileyebilir.

Bu nedenle sistemimizi dış birimlerdeki değişimlerden en az etkilenecek şekilde tasarlamaya çalışacağız.

Diğer bir problem ise aynı hizmet için belli koşullara göre farklı birimlerden hizmet alınmasıdır. Örneğin bazı kredi kartları için bir merkezden, bazıları için de farklı bir merkezden onay alınabilir.

Tasarlanan sistemin bu tür isteklere yanıt verebilecek yapıda olması istenir.

Alternatif akışlar ve dış birimlerle (aktörlerle) ilişkiler:



Dikkat:
Yandaki diyagram, senaryoları (*use-case*) ifade eden bir ardışıl diyagramdır.
Tasarım aşamasındaki nesneler arasındaki etkileşimi gösteren diyagram değildir.

Diğer GRASP Kalıpları (Sorumlulukların Atanmasının Devamı)

Daha önceki derslerde ilk beş GRASP kalıbı ele alınmıştı. Şimdi diğer dört kalıp incelenecek.

6. Çok Şekillilik (*Polymorphism*)

Problem: Tiplere bağlı alternatifler nasıl ele alınmalı? Sisteme kolay monte edilebilen (*pluggable*) yazılım parçaları nasıl gerçeklenir?

Çözüm: Birbirleriyle ilgili alternatif davranışlar, tiplere (sınıflara) bağlı olarak değişiklik gösteriyorsa bu davranışları çok şekilli (*polimorphic*) metotlar kullanarak gerçekleyiniz.
Koşullu deyimler (if-else, case) kullanarak tipleri test edip ona göre ilgili metodu çağırmak böyle durumlar için iyi bir yöntem değildir.

Kolay monte edilebilirlilik (pluggable):

Nesneye dayalı yazılımları, kullanıcı ve sunucu (*client–server*) mantığı ile ele almak mümkündür.

Bir yazılım parçasının (sınıfın) sisteme kolay monte edilebilmesi demek bir sunucu sınıfın sistemden çıkarılması, onun yerine sisteme başka bir sınıfın yerleştirilmesi ve kullanıcıların bundan etkilenmemesidir.

Çok şekillilik (*polymorphism*) sayesinde, bir sınıf bir nesnenin tipini (hangi sınıftan yaratıldığını) bilmeden ona mesaj gönderebilmektedir.

Mesajı alan nesnenin metodu canlanır ve kendi tipine bağlı olarak bir işlevi gerçekleştirir.

Mesaj gönderilme işlemi bir adres (işaretçi veya referans) ile yapılmaktadır. Bu adres aynı taban sınıftan (soydan) türeyen farklı sınıflardan yaratılan nesnelere işaret edebilir.

Böylece aynı satırda, programın çalışması sırasında (*run-time*) farklı tipte nesnelere mesaj göndermek mümkün olur.

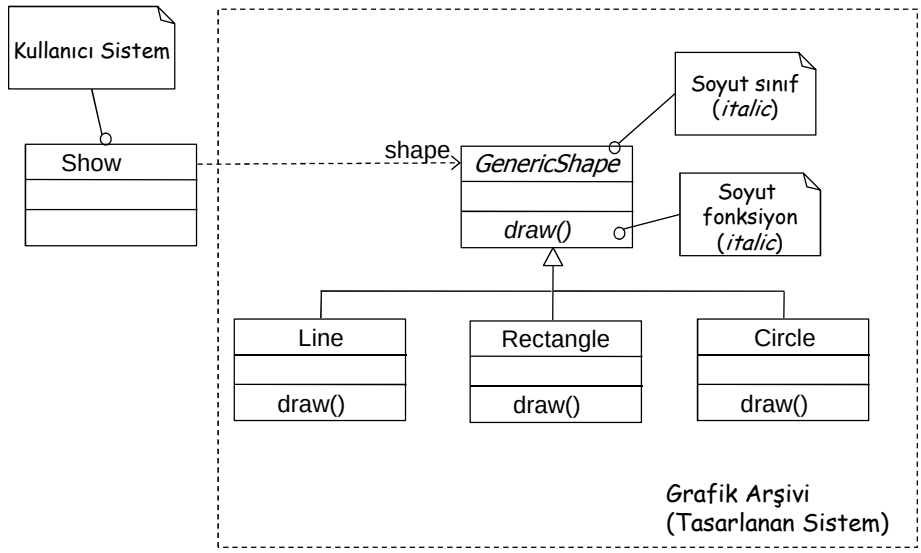
Örnek:

Değişik şekiller içeren bir grafik arşiv programı hazırlanmıştır. Tüm şekiller GenericShape adlı soyut bir sınıftan türetilmiştir. Her sınıfta o şekli ekrana çizen çok şekilli (*polimorphic*) bir metot (draw) vardır.

Örnek programda çok şekilliliğin işlevini anlayabilmek için show fonksiyonunu incelemek gerekir.

Görüldüğü gibi show fonksiyonu şekillerin tipinden bağımsız olarak onlara draw mesajını göndermektedir. Böylece grafik arşivindeki ekleme ve çıkarmalardan show sistemi etkilenmez.

Örnek Tasarımın UML Diyagramı



Örnek Kodlama (C++):

```
class GenericShape{                                    // Abstract base class
  protected:
     int x, y;                                         // Genreral purpose coordinates
  public:
    GenericShape(int x_in, int y_in){ x = x_in; y = y_in; }  // Constructor
    virtual void draw() const =0;                      // pure virtual function
};

class Line:public GenericShape{          // Line class
 protected:
  int x2, y2;                            // End coordinates of line
 public:
  Line(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),  x2(x2_in),y2(y2_in)
  { }
  void draw()const;                      // virtual draw function
};
void Line::draw()const
{
  cout << "Type: Line" << endl;
  cout << "Coordinates of end points: " << "X1=" << x <<  " ,Y1=" << y <<
                                          " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```



```
class Rectangle:public GenericShape{        // Rectangle class
 protected:
   int x2,y2;                                // coordinates of 2nd corner point
 public:
   Rectangle(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),
                                          x2(x2_in),y2(y2_in)
   { }
   void draw()const;                      // virtual draw
};


void Rectangle::draw()const
{
   cout << "Type: Rectangle" << endl;
   cout << "Coordinates of corner points: " << "X1=" << x << " ,Y1=" << y <<
                                          " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```

```
class Circle:public GenericShape{              // Circle class
 protected:
    int radius;
 public:
    Circle(int x_cen,int y_cen,int r):GenericShape(x_cen,y_cen), radius(r)
    { }
    void draw() const;                         // virtual draw
};

void Circle::draw()const
{
    cout << "Type: Circle" << endl;
    cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
    cout << "Radius: " << radius << endl;
}
```


Grafik arşivi kullanan program (show):

```
void show(const GenericShape &shape)  //Değişik şekillerin adresini alabilir
{
   shape.draw();                       // Hangi draw fonksiyonunun çağırılacağı
}                                      // derleme aşamasında belli değildir.


// Sınama amaçlı ana program
int main()
{
   Line line1(1, 1, 100, 250);
   Circle circle1(100, 100, 20);
   Rectangle rectangle1(30, 50, 250, 140);
   Circle circle2(300, 170, 50);
   show(circle1);              // show parametre olarak değişik şekil nesnelerini alabilir
   show(line1);
   show(circle2);
   show(rectangle1);
   return 0;
}
```

Bu programa yeni şekil sınıfları eklemek (monte etmek) kolaydır.

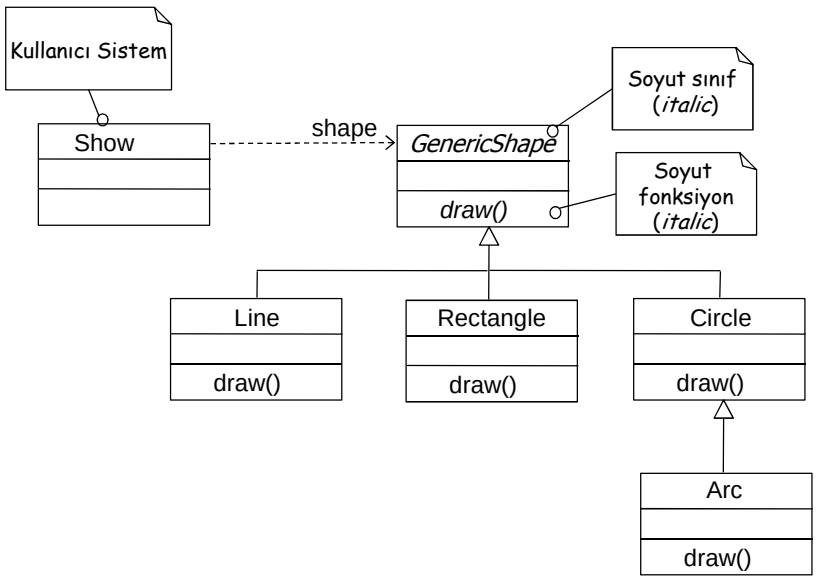
show fonksiyonunda şekiller ekrana çıkarılırken tip testi yapılmaz. Programa yeni bir şekil eklenmesi show fonksiyonunu etkilemeyecektir.

Örneğin grafik arşivine örnekte gösterildiği gibi yay (Arc) sınıfı eklendiğine show sisteminde hiç bir değişiklik olmaz.

```
class Arc:public Circle{                    // Arc class
 protected:
    int sa, ea;                             // Start and end angles
 public:
    Arc(int x_cen,int y_cen,int r, int a1, int a2):Circle(x_cen,y_cen,r),
                                                sa(a1),ea(a2)
     {}
    void draw() const;                      // virtual draw
};

void Arc::draw()const
{
   cout << "Type: Arc" << endl;
   cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
   cout << "Radius: " << radius << endl;
   cout << "Start and end angles: " << "SA=" << sa <<  " ,EA=" << ea << endl;
}
```


Sisteme yeni bir şeklin eklenmesi:

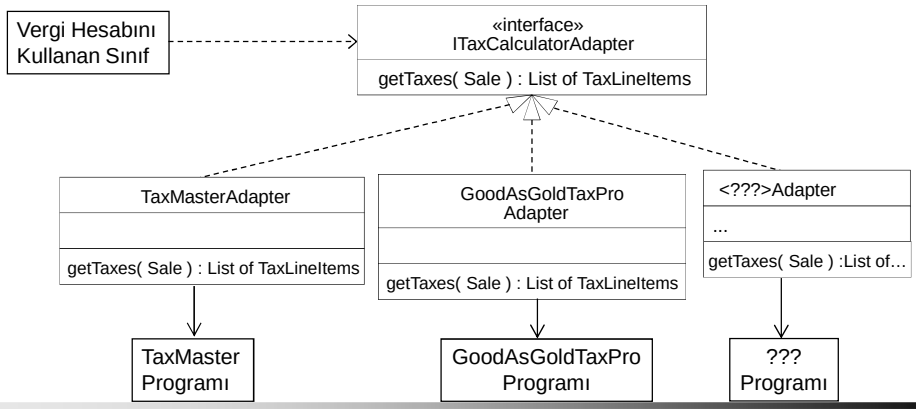


Örnek POS sisteminde çok şekilliliğin kullanılması:

Sistemde vergi hesabının üçüncü parti programlar tarafından yapıldığı varsayılıyor.

Tasarlanan POS sisteminin var olan farklı vergi hesaplama programları ile ya da gelecekte alınacak yeni bir hesaplama programıyla çalışabilmesi isteniyor.

Bu problemi çözmek için POS yazılımı ile vergi hesaplama programları arasında ilişki sağlayan arayüz sınıfları (*adapter*) oluşturulur. Her adaptör sınıfının içinde çok şekilli getTaxes metodu bulunur.



7. Yapay Sınıf (*Pure Fabrication*)

Nesneye dayalı programlamanın temelini, yazılımı oluşturan unsurların gerçek dünyadaki varlıklara benzer şekilde tasarlanması oluşturmaktadır.

Bu nedenle gerçek dünyadaki varlıklar ile onları yazılımda temsil eden unsurlar arasında büyük benzerlik bulunur.

Ancak bazı durumlarda, sorumlulukların gerçek dünyadaki varlıklara atanması uyum ve bağımlılık açısından sorunlara neden olabilir.

Böyle durumlarda tasarım aşamasında gerçek dünyada olmayan yapay sınıflar yaratılabilir.

Problem: Uzman kalıbının önerdiği çözüm, iyi uyum ve az bağımlılık kavramları ile çelişiyorsa sorumlulukları hangi sınıfa atamak gerekir?

Çözüm: Eğer sınıflar arası bağımlılıkları azaltıyorsa ve yazılımın tekrar kullanırlılığını arttırıyorsa, birbirleriyle uyumlu sorumlulukları gerçek dünyada var olmayan yapay bir sınıfa atayabilirsiniz.

Örnek: POS sisteminde satış bilgilerinin bir veri tabanına kayıt edilmesi gerekir.

Uzman kalıbına göre bu sorumluluk Sale sınıfına atanabilir. Ancak bu sorumluluklar Sale sınıfının uyumluluğunu bozar.

Bu problemi çözmek için nesneleri veri tabanına kayıt etmekle görevli yapay bir sınıf yaratılabilir (PersistentStorage).

Yaratılan yapay sınıf (örnekte PersistentStorage) daha genel yapıdadır, başka nesneleri kayıt etmek için de kullanılabilir.

Ayrıca başka projelerde tekrar kullanılabilme (*reusability*) olasılığı da yüksektir.

Yazılım sınıflarının tasarımında iki temel yöntem uygulanır:

1. Gerçek dünyadaki kavramlardan elde etmek (*representational decomposition*).
2. İşlevlerden elde etmek (*behavioral decomposition*).

Nesneye dayalı programlarda yazılımın kalitesini arttırmak için öncelikle birinci yöntem tercih edilir.

Ancak uyum ve bağımlılık problemleri oluştuğunda bunları çözmek için işlevleri temel alarak benzer işlevleri bir sınıf içinde toplamak gerekebilir.

Yapay sınıf (*pure fabrication*) kalıbı ikinci yöntemi temel almaktadır.

Burada dikkat edilmesi gereken nokta yapay sınıfların gereğinden fazla kullanılmamasıdır.

Özellikle işleve dayalı program yazmaya alışmış olan programcılar, sistemleri gerçek dünyadaki varlıklara ayırmak yerine işlevlere ayırmak eğilimindedirler.

Eğer yapay sınıflar çok fazla kullanılırsa yazılım gerçek dünyadan uzaklaşır, anlaşılırlılık azalır, sınıflar arası bağımlılık artar.

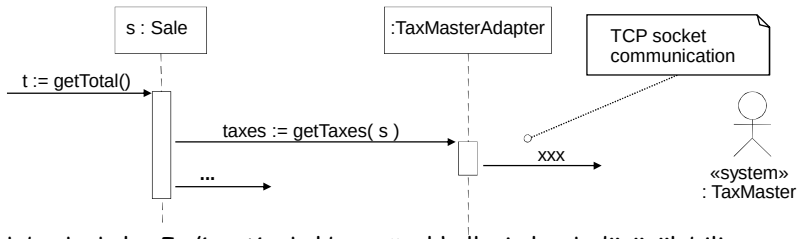
8. Dolaylılık ya da Arabirim (*Indirection*)

Problem: Özellikle kolay değişebilen iki birim arasındaki bağımlılığı azaltmak için sorumluluklar nasıl atanmalıdır?

Çözüm: İki birim arasındaki bağımlılığı azaltmak için sorumlulukları bir ara nesneye atayın.

Vergi hesaplama programları ile POS sistemi arasına konan adaptör nesneleri de dolaylılık kalıbına uygundurlar. Bu nesneler çok şekillilik özelliği ile birlikte POS sisteminin dış değişimlerden etkilenmesini önlerler.

Satış bilgilerini veri tabanına kayıt etmek için oluşturulan yapay sınıf da (PersistentStorage) Sale sınıfı ile veritabanı arasında bir arabirimdir.



Bir çok başka kalıp *Indirection* kalıbının özel halleri olarak düşünülebilir.

Örneğin: Adapter, Facade gibi GoF kalıpları.

9. Değişimlerden Korunma (*Protected Variation*)

Hatırlatma: Yazılımların maliyetlerini arttıran unsurların başında, yazılımın bir yerindeki değişimin (nesne, kullanılan başka bir program olabilir) programın diğer kısımlarını etkilemesi gelmektedir.

Problem: Nesneler, sistemler ve alt sistemler, bu birimlerdeki değişim ve kararsızlıklar birbirlerini en az etkileyecek şekilde nasıl tasarlanmalıdır?

Çözüm: Sistemde değişimlere açık, kararsız noktaları belirleyin ve bu değişim noktalarının etrafında kararlı bir arayüz oluşturacak şekilde sorumlulukları atayın.

Örnek POS sisteminde, vergi hesaplamada değişik programların kullanabilmesi ve bu programların veri aktarımı için farklı yöntemler kullanabilmesi sistemin karasız ve değişime açık noktalarından biridir.

POS sistemi ile vergi programları arasına konan adaptör nesneleri, dolaylılık ve çok şekillilik özellikleri sayesinde sistemi dış değişimlerden korumaktadırlar.

Değişimlerden korunma, nesneye dayalı programlamada kullanılan bir çok prensibin ve kalıbın temelini oluşturmaktadır.

Nitelik ve davranışların bir bütün oluşturması (*encapsulation*), veri gizleme (*data hiding*), çok şekillilik (*polymorphism*), dolaylılık bu prensibin sonucu oluşmuş kavramlardır.

Değişimlerden korunma prensibinin bir türevi de "**Yabancılarla Konuşma**" (*Don't Talk to Strangers*) diğer adıyla Demeter Kanunu (*Law of Demeter* - LoD) prensibidir. (Karl Lieberherr, http://www.ccs.neu.edu/home/lieber/)

Bu prensip bir nesnenin mesaj gönderebileceği nesnelere sınırlamalar getirmektedir. Buna göre bir nesnenin metodu içinde mesaj gönderilebilecek nesneler şunlardır:

- Kendisi (this)
- Metodun parametresi olan nesne
- Nesnenin niteliği (üyesi) olan nesne
- Nesnenin üyesi olan bir grubun (liste, vektör, vb.) elemanı olan nesne
- Metodun içinde yaratılan nesne

Bu tip nesneler tanıdık (*familiar*) olarak nitelendirilir. Dolaylı olarak (tanıdık bir nesne üzerinden) erişilebilen nesneler ise yabancı (*stranger*) nesnelerdir.

Bir metot içinden yabancı nesnelere mesaj gönderilmesi yabancı nesnelere bağımlılık yaratır. Üstelik bu bağımlılık UML diyagramlarında görülmez ve fark edilmesi oldukça zordur. Bu nedenle tercih edilmez.

Yabancı nesnelere mesaj gönderilmesine ilişkin bir örnek aşağıda gösterilmiştir:

```
public class Register
{
   private Sale sale;
   public void zayifBirMetot()
   {
      Money total = sale.getTotal();            // Sale tanıdık bir nesnedir, çünkü üye
      Money amount = sale.getPayment().getTenderedAmount(); // Yabancı nesneye mesaj
      :
   }
   :
}
```

Yukarıda sale.getPayment() mesajı ile yabancı bir nesnenin (Payment) adresi alınıyor ve yabancı nesneye getTenderedAmount() mesajı gönderiliyor.

Bir çağrı ne kadar çok ara adımdan oluşuyorsa o kadar tehlikelidir:

```
obj1.m1().m2().¨¨.mn();
```

Bu bilgiyi almak Register sınıfının içinde gerekli ise Sale sınıfına uygun bir metot koyarak (sorumluluk atayarak) yabancı nesneler arasındaki ilişki ortadan kaldırılabilir:

```
Money amount = sale. getTenderedAmountOfPayment();
```