GoF Tasarım Kalıpları

GoF (*Gang of Four*) kalıpları 1994'te yayımlanan dört yazarlı bir kitap ile duyurulmuştur:

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Reading MA, Addison-Wesley.

Kitapta 23 kalıp yer almaktadır. Bunlardan 15 tanesi daha yoğun kullanılmaktadır.

GoF Kalıpları 3 gruba ayrılır:

Creational Patterns:	**Structural Patterns:**	**Behavioral Patterns:**
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Bu dersin kapsamında çok kullanılan GoF kalıplarından bazıları tanıtılacaktır.

1. Adaptör (*Adapter*) (GoF)

Bu kalıp, istenen işi yapan hazır sistemlere (sınıfa) sahip olduğumuzda, ancak bu hazır sınıfın arayüzünün bizim beklediğimizden farklı olduğu durumlarda kullanılır.

Temel işlevi bir sınıfın arayüzünü başka bir şekle dönüştürmektir.

Adaptör kalıbı daha karmaşık bir problemin çözümünde de kullanılır:

Tasarlanmakta olan sistemin aynı iş için birden fazla farklı sistem ile (sınıflar) ile ilişki kurması gerekebilir.

İstenen işi yapan hazır sınıfların arayüzleri bizim beklediğimizden ve birbirlerinin-kinden farklı olabilir.

Örneğin vergi hesabı için birbirinden farklı üçüncü parti yazılımlar kullanılabilir.

Bu durumda adaptör kalıbı farklı arayüzleri, aynı ortak şekle dönüştürmek için kullanılır.

Aynı durum kredi kartı asıllama programları veya muhasebe programları için de geçerlidir.

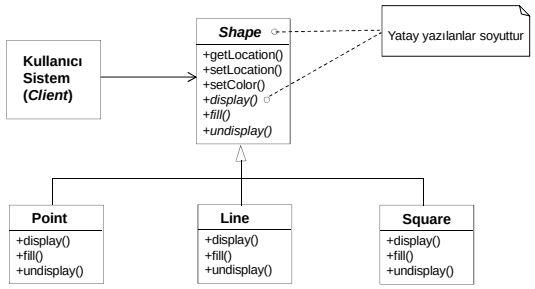
Problem: İstenen işi yapan fakat farklı arayüzleri olan benzer birimler için tek bir kararlı arayüz nasıl yaratılır?

Çözüm: Birimin orijinal arayüzünü bir adaptör nesnesi kullanarak başka bir arayüze dönüştürün.

POS sistemi ile ilgili örneğe geçmeden önce daha basit bir örnek verilecektir.

Örnek:

Nokta, çizgi, kare şekillerini içeren bir destek sistemi tasarlamak gerekiyor.
Asıl sistemin (kullanıcı sistem) şekillerin tipinden bağımsız olması isteniyor.
Bu nedenle tüm şekilleri Shape adında soyut bir sınıftan türetiyoruz.

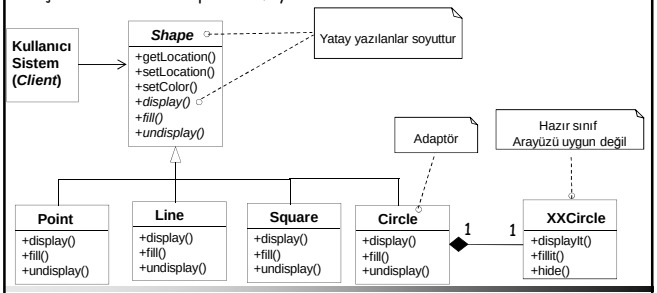


Bir süre sonra istekler değişir! Çember şeklinin de sisteme eklenmesi istenir.

Bu durumda Circle, *Shape* sınıfından türetilebilir, ancak çembere özgü davranışların yeniden yazılması gerekir.

Elimizde çember ile ilgili işleri yapan hazır bir sınıf bulunabilir ancak bu hazır sınıfın arayüzü bizim daha önce oluşturduğumuz *Shape* arayüzünden farklı olacaktır.

Bu durumda hazır XXCircle sınıfını içeren ve arayüzünü bize gerekli olan şekle dönüştüren bir Circle adaptör sınıfı yazılır.



Java'da kodlama:

```
class Circle extends Shape  {
    ...
    private XXCircle pxc;              // hazır sınıftan nesnelere referans içeriyor.
    public Circle(....) {              // constructor, parametreleri olabilir.
        pxc = new XXCircle(....);      // hazır sınıftan nesne yaratılıyor, parametre alabilir.
    }
    void public display() {            // arayüzü uyumlu hale getiren metot
        pxc.displayIt();               // hazır sınıfın metodu çağırılıyor.
    }
}
```

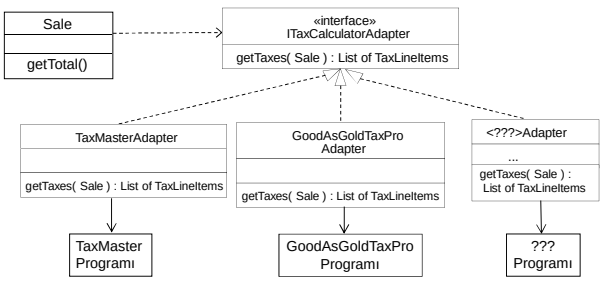
C++'da kodlama:

```
class Circle : public Shape  {
   private:
      XXCircle  *pxc;                  // hazır sınıftan nesnelere işaretçi içeriyor.
...                                    // diğer üyeler
};
Circle::Circle(....) {                 // constructor, parametreleri olabilir.
      pxc = new XXCircle(....);        // hazır sınıftan nesne yaratılıyor, parametre alabilir.
}
void Circle::display() {               // arayüzü uyumlu hale getiren metot
      pxc->displayIt();                // hazır sınıfın metodu çağırılıyor.
}
Circle::~Circle() {                    // destructor
      delete pxc;                      // içerilen nesne bellekten siliniyor.
}
```


Örnek:

POS sisteminde vergi hesabı aynı anda için birbirinden farklı üçüncü parti yazılımlar kullanılabilir.

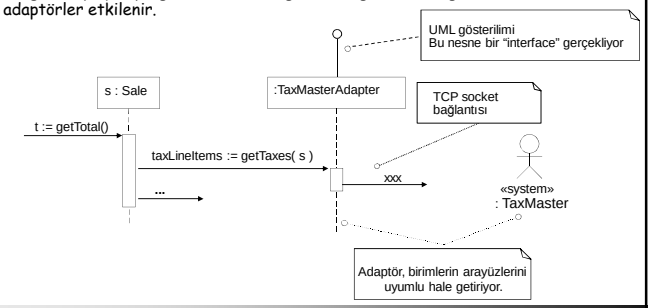
Arayüzleri farklı olan ve farklı şekillerde çalışan vergi hesabı programları ile kendi sistemimiz arasına adaptörler koyarak ve bu adaptörlerin hepsini ortak bir üst sınıftan türeterek farklı programların arayüzlerini aynı şekle getiriyoruz.



Böylece vergi hesabını yaptıracak olan Sale sınıfı hangi programı kullanırsa kullansın her zaman getTaxes mesajını gönderiyor.

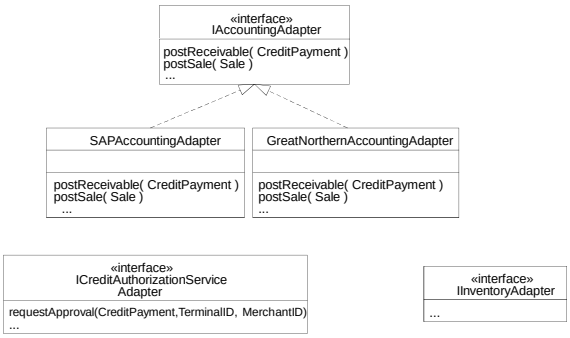
Bu mesajı alan ilgili adaptör aşağıda gösterildiği gibi kendi bağlı olduğu programa uygun mesajları göndererek vergi hesabını yaptırır.

Vergi hesaplayan programlarda bir değişim olduğunda bu değişimlerden sadece adaptörler etkilenir.

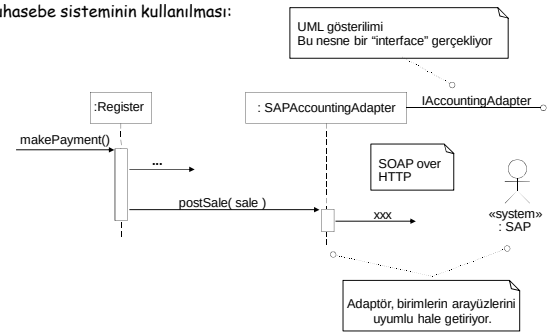


Aynı durum kredi kartı asıllama programları ve muhasebe programları için de geçerlidir.

Bu dış birimler ile onları kullanacak olan sınıf arasına adaptörler yerleştirilir.

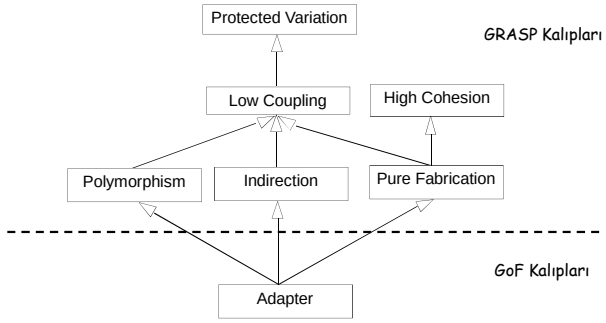


Muhasebe sisteminin kullanılması:



Karmaşık ve özel kalıplar daha temel GRASP kalıpları (prensipler) ile ifade edilebilirler.

Örneğin adaptör kalıbında değişimlerden etkilenmeyi azaltacak şekilde dolaylılık ve çok şekilliliğe sahip yapay sınıf kullanılmıştır.

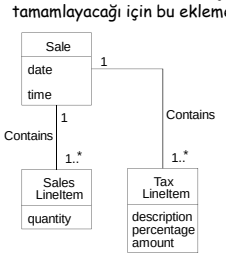


Tasarım Aşamasında Çözümleme Aşamasına İlişkin Yeni Kavramların Keşfedilmesi

Yazılım sisteminin tasarımı yapılırken, çözümleme aşamasındaki modelde yer almamış olan yeni kavramlar ortaya çıkabilir.

Örneğin adaptörlerin tasarımında oluşturulan getTaxes metodu, vergi kalemleri listesi (TaxLineItems) geri döndürür. Bu kavram (vergi kalemleri) çözümleme aşamasında modelde yer almamış olabilir.

Tasarım aşamasında keşfedilen kavramlar, eğer gerek görülürse çözümleme modeline eklenebilir. Bir kaç iterasyondan sonra çözümleme (analiz) modeli işlevini tamamlayacağı için bu ekleme gereksiz de olabilir.



Hatırlatma: Analiz modeli,
- problem uzayındaki kavramları daha iyi anlayabilmek,
- yazılım sınıflarını oluştururken esinlenmek için kullanılır.

Eğer çözümleme modeli sonraki aşamalarda tekrar bir kaynak olarak kullanılacaksa yeni keşfedilen kavramların modele eklenmesi uygun olabilir.

Daha çok tercih edilen yöntem ise, tasarım modelinden geri gidilerek (*reverse-engineering*) sonraki aşamalarda kullanılabilecek çözümleme modelinin oluşturulmasıdır.

2. Fabrika (*Factory*), Somut Fabrika (*Concrete Factory*)

Adaptörlerin kullanılması, bu nesneleri kimin yaratacağı problemine neden olur.

Diğer bir problem ise belli bir durumda olası adaptörlerden hangisinin yaratılıp kullanılacağıdır.

Adaptörleri yaratma işi, bunu kullanacak olan yazılım sınıflarından birine (uygulama sınıfı) atanırsa bu sınıfın uyumu bozulur ve değişken adaptörlere bağımlı hale gelir.

Uygulama sınıfı hangi durumda hangi adaptörü kullandığını bilmemelidir.

Önemli tasarım prensiplerinden biri; bir yazılım sistemindeki değişik işler yapan kısımların (katmanların) birbirlerinden bağımsız olarak ele alınmasıdır (***seperation of concerns***). Buna göre de adaptörlerin uygulama nesnelerinin dışındaki bir nesne tarafından yaratılması uygun olacaktır.

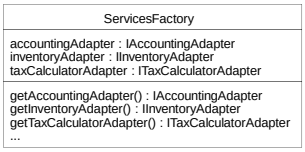
Bu bölümde GoF kalıplarından soyut fabrikanın (***Absract Factory***) daha basit bir şekli olan fabrika (*Concrete Factory*) kalıbı ele alınmıştır.

Problem: Nesnelerin yaratılmasında karmaşık kararlar vermek gerekiyorsa ve uyumluluğu arttırmak için yaratma işlemlerinin diğer işlemlerden ayrılması isteniyorsa nesne yaratılma sorumluluğu nasıl atanmalıdır?

Çözüm: Nesnelerin yaratılma sorumluluklarını yapay bir sınıf olan fabrika sınıfına atayın.

Fabrikalar sadece adaptörler için hazırlanmazlar. İlerideki konularda başka nesnelerin yaratılması sorumluluğunun da fabrika nesnelerine verildiği görülecektir.

Örneğin POS sisteminde adaptör nesnelerini yaratmak için ServicesFactory adında bir yapay sınıf oluşturulabilir.



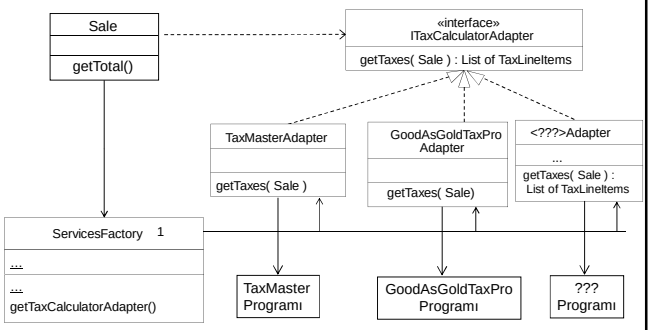
Vergi hesabı programını kullanmak isteyen nesne gerektiğinde ServicesFactory nesnesinin getTaxCalculatorAdapter metodu çağırılacaktır.

Bu metot o andaki kriterlere göre hangi tipte bir adaptörün kullanılacağına karar verecek, eğer yoksa uygun bir adaptör nesnesi yaratacak ve bu nesnenin adresini geri döndürecektir.

Böylece vergi hesabı yapılmak istendiğinde bu adaptör ile ilişki kurulacaktır (getTaxes metodu çağrılacaktır).

ServicesFactory sınıfının get metotları adaptörlerin üstsınıflarına ilişkin adresler geri döndürmektedir. Çünkü üst sınıfın adresini taşıyabilen bir işaretçi o sınıftan türeyen tüm alt sınıflara da işaret edebilir.

Adaptörler ve Fabrika:



Soyut Fabrika (*Abstract Factory*) (GoF)

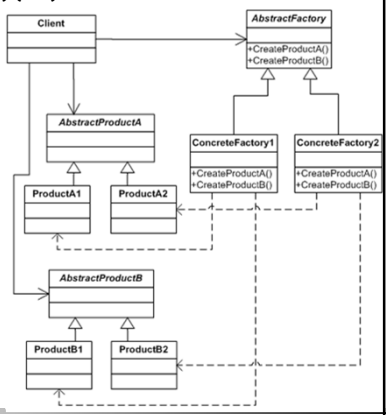
Bazı durumlarda birbirleriyle ilişkili birden fazla nesne yaratmak gerekir.

Örneğin yandaki şekilde gösterildiği gibi belli koşullarda ProductA1 sınıfından nesne ile ProductB1 sınıfından nesne birlikte kullanılacaktır.

Benzer şekilde de ProductA2 sınıfından nesne ile ProductB2 sınıfından nesne birlikte kullanılacaktır.

Bu durumda birbirleriyle ilişkili nesneleri (aile) yaratan farklı fabrika sınıfları oluşturulur.

Bu fabrika sınıfları ortak bir soyut fabrikadan türetilir.



Fabrika Metodu (*Factory Method*) (GoF)

Yaratma işi ayrı fabrika sınıfları yerine metotlara atanır.

Yaratıcı (fabrika) metotlar, yaratılan nesneleri kullanacak olan sınıfların üyeleri olabilirler.

Burada sınıflar, farklı nesneler içinden kendileri ile ilgili olanları yaratıp kullanırlar.

Örnek:

Sistemde farklı tipte dosyalar (grafik, metin vs.)

ve farklı tipteki dosyaların kullanacağı farklı tipte dokümanlar olsun.

Grafik dosyası grafik dokümanları, metin dosyası metin dokümanları yaratıp kullansın.

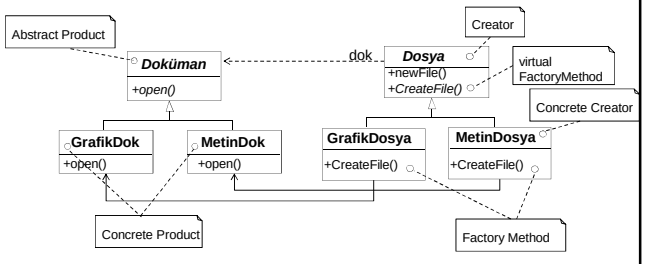
İleride sisteme yeni dosya tipleri ve yeni doküman tipleri gelebilir.

Çözüm:

- Tüm dosyalar soyut bir taban sınıftan türetilir.
- Hangi dokümanın yaratılacağına taban sınıftan türetilen dosya sınıfı karar verir.
- Yeni bir dosya için alt sınıf türetildiğinde hangi tipte dokümanın yaratılacağı bu alt sınıftaki fabrika metodunda belirlenir.

Fabrika metodunun yarattığı nesneleri uygun biçimde yok eden metotlar da (***disposal method***) yazılabilir.

Örnek Tasarım:




```
Dosya::newFile():
    dok = CreateFile();
    dok->open();
```

```
GrafikDosya:: CreateFile():
    return (new GrafikDok());
```

```
MetinDosya:: CreateFile():
    return (new MetinDok());
```


3.Tekil Nesne (*Singleton*) (GoF)

Fabrika nesnesi yeni bir problem oluşturur: Fabrika nesnesini kim yaratacak ve bu nesneye nasıl erişilecek?

İstenenler:

- Fabrika sınıfından sadece tek bir nesne yaratılmalı,
- Programın gerekli tüm yerlerinden bu nesneye erişilebilmeli.

Problem: Bir sınıftan sadece tek bir nesne yaratılması (tekil nesne) isteniyor. Bu nesne diğer nesnelere global tek bir erişim noktası sağlayacak.

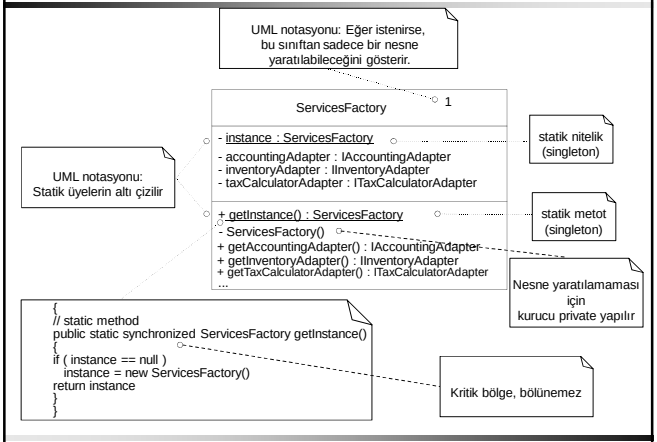
Çözüm: Sınıfın içine tekil nesneyi yaratıp adresini döndüren statik bir metot yerleştirin.

Hatırlatma: Statik metotlar, üyesi oldukları sınıftan henüz bir nesne yaratılmadan önce de çağırılabilirler.

Örnek sistemde, ServicesFactory sınıfına, bu tipten nesnelere işaret edebilen statik bir nitelik üyesi, ve nesne yaratan statik bir metot eklenir.

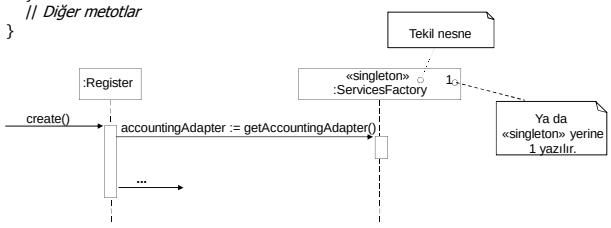
Statik metot (getInstance) çağırıldığında eğer daha önceden bir fabrika nesnesi yaratılmamışsa nesne yaratır ve adresini geri gönderir.

Eğer daha önceden nesne yaratılmışsa yenisi yaratılmaz var olan nesnenin adresi gönderilir.



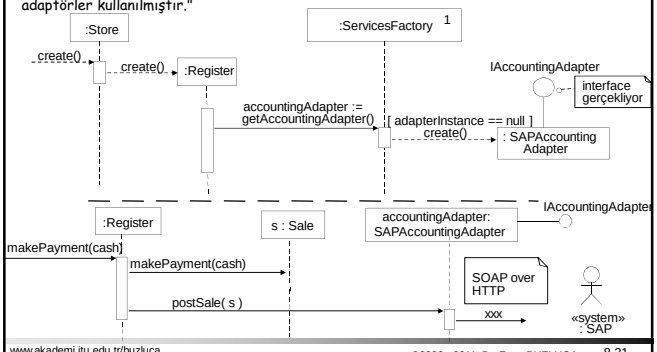
Programın herhangi bir yerinde fabrika nesnesine ulaşılıp bir adaptör nesnenin adresi öğrenilebilir.

```
public class Register
{
   public Register( ProductCatalog catalog )   // constructor
   {
     ...
     accountingAdapter= ServicesFactory.getInstance().getAcountingAdapter();
     ...
   }
   // Diğer metotlar
}
```



Değişken Arayüzlü Dış Hizmetlere Genel Bir Bakış

Değişik arayüzlere sahip dış hizmetlere (vergi hesabı, muhasebe, kredi kartı asıllama) erişimde ortaya çıkan problemlerin çözümü için çeşitli kalıplardan yararlanılmıştır. Kalıpların kullanımı yazılımcılar arasında ortak bir dil oluşturur: " Dış hizmetlerin değişken arayüzlerinin oluşturduğu sorunları çözmek için tekil fabrika nesneleri tarafından yaratılan adaptörler kullanılmıştır."



4.Strateji (*Strategy*) (GoF)

Bir sınıfın belli bir konudaki davranışının, o sınıftan yaratılan bir nesnenin yaşamı süresinde defalarca değişmesi istenebilir.

POS sisteminde bu probleme örnek olarak değişik koşullarda farklı indirim politikalarının uygulanması gösterilebilir.

Örneğin belli günlerde %10, belli günlerde %15 indirim yapılabilir. Ödeme miktarı belli bir değeri geçtiğinde sabit bir indirim uygulanabilir. Hatırlı müşterilere belli bir indirim yüzdesi uygulanabilir.

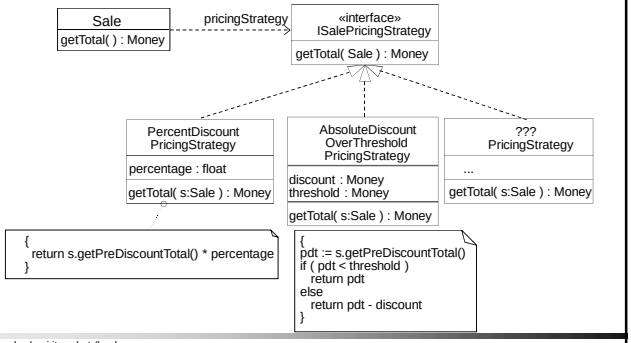
Tüm bu işler Sale sınıfının toplam bedeli hesaplama getTotal() sorumluluğunun (davranışının) değişik halleri gibi görünmektedir.

Ancak bu değişken algoritmaları Sale sınıfının içine koymak uyum ve bağımlılık problemleri yaratacaktır.

Problem: Birbirleriyle ilgili olan fakat farklılık gösteren algoritmalar (davranışlar) nasıl tasarlanmalıdır? Bu algoritmalardaki değişimin sistemi etkilemesi nasıl önlenmelidir?

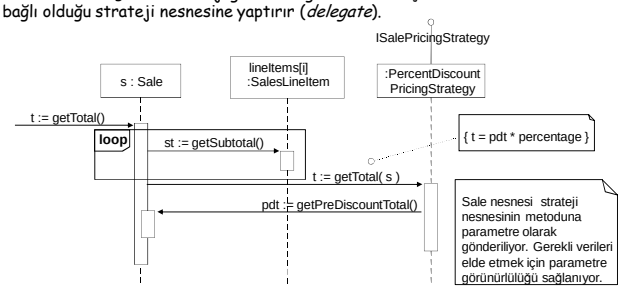
Çözüm: Her algoritmayı ayrı bir sınıf içinde gerçekleyin. Bu sınıfların arayüzleri aynı olmalıdır.

Bu kalıba göre ortak bir üst sınıftan türeyen indirim sınıfları (SalePricingStrategy) tasarlanacaktır. Bu sınıfların hepsinde çok şekilli bir getTotal metodu bulunacaktır. Bu metot parametre olarak Satış nesnesini (Sale) alacak ve gerekli indirimleri uyguladıktan sonra müşterinin ödemesi gereken toplamı belirleyecektir.



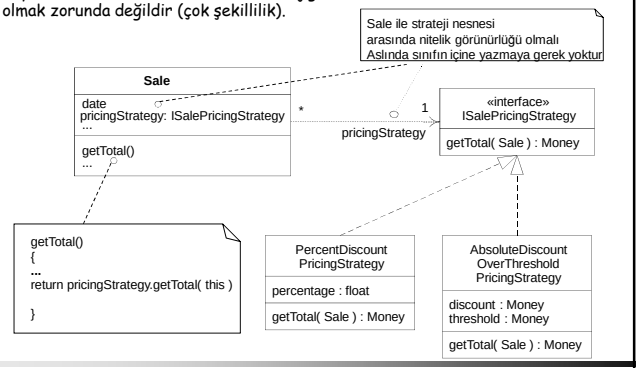
Strateji nesnesi bir bağlam nesnesi (*context object*) ile ilişkilendirilir.
Bu örnekte bağlam nesnesi Satış (Sale) nesnesidir.

Sale nesnesine getTotal mesajı gönderildiğinde Sale bu işlemin bir kısmını o anda bağlı olduğu strateji nesnesine yaptırır (*delegate*).

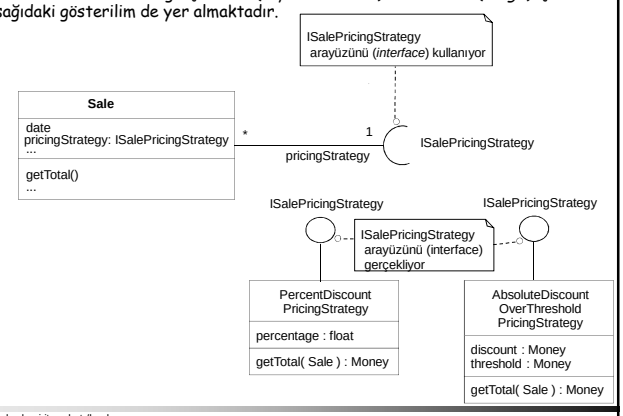


Bağlam nesnesi (Sale) tek tek farklı indirim nesneleri ile ilişkilendirilmez. Onun yerine indirim nesnelerinin türetildiği soyut sınıf (ya da *interface*) ile ilişkilendirilir.

Böylece farklı indirim nesnelerine mesaj gönderilebilir ve Sale bundan haberdar olmak zorunda değildir (çok şekillilik).



UML 2.0'da "interface" gerçekleme (*implementation*) ve kullanma (*usage*) için aşağıdaki gösterilim de yer almaktadır.

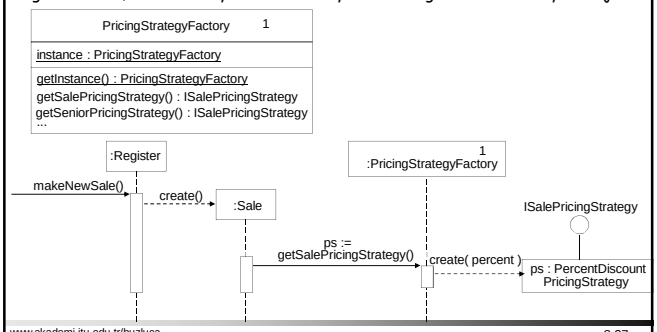


Creating strategies with a Factory:

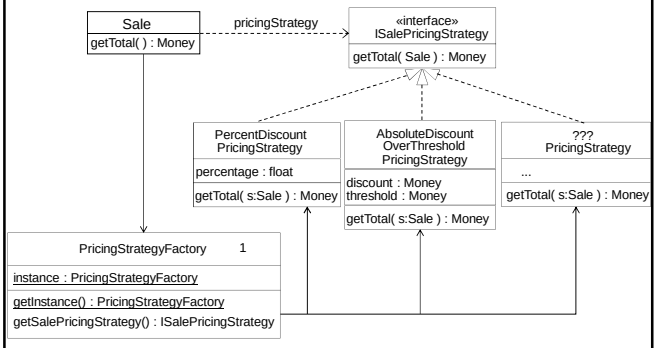
The Factory pattern can be applied to create the necessary strategy object.

A PricingStrategyFactory can be responsible for creating strategies .

The new factory is different than the ServicesFactory. This supports the goal of High Cohesion, each factory is focused only on creating a related family of objects.



Fabrika ile birlikte sistemin sınıf diyagramı:



5.Bileşik Nesne (*Composite*) (GoF)

Bazı durumlarda bir nesne belli bir iş için tek bir nesne (atomik bir nesne) ile ilişkilendirildiği gibi aynı iş için bir nesne grubu (liste, vektör) ile de ilişkilendirilebilir.

Esnekliği sağlamak için ilgili nesnenin, o iş için tek bir nesne ile mi yoksa bir nesne grubuyla mı ilişkili olduğundan habersiz (bağımsız) olması istenir.

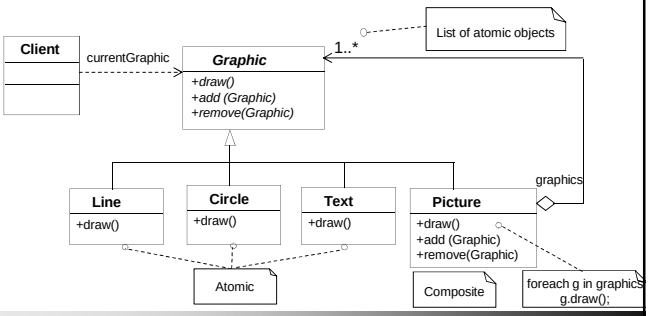
Kalıp:

Problem: Nesne grupları veya bileşik nesneler, tek bir nesne (atomik) gibi çok şekilli olarak nasıl tasarlanır?

Çözüm: Bileşik ve atomik nesneleri aynı soyut sınıftan (ara yüzden) türetin. Bileşik nesnelerin içine atomik nesneleri içeren bir liste yerleştirin.

Örnek: (GoF kitabından)

Bu örnekte kullanıcı sınıf (Client) hem atomik şekil nesneleriyle (Line, Circle, Text) ilişkilendirilebilmekte (hizmet alabilmekte) hem de bu atomik nesneleri içeren bir bileşik nesne (Picture) ilişkilendirilebilmektedir.



Örnek:

Örnek POS sisteminde buna benzer bir problem indirim stratejilerinde ortaya çıkmaktadır. Bazen tek bir indirim stratejisi kullanıldığı gibi aynı anda birden fazla indirim stratejisi de geçerli olabilir. Hatta bu stratejiler birbirleri ile çatışabilir.

Örneğin;

- Pazartesi günü belli bir değerin üzerinde alışveriş yapanlara sabit bir indirim uygulanır.
- Hatırlı müşterilere (kart sahibi) %15 indirim yapılır.
- Belli bir üründen (markadan) alındığında toplam satış bedelinden %5 indirim yapılır.

Bu durumda hatırlı bir müşteri Pazartesi günü alışveriş yaparsa ve o günün özel markasından satın alırsa nasıl bir indirim stratejisi uygulanacak?

Problemin özellikleri:

- Sale sınıfı nesnelerinin bazen tek bir indirim stratejisi kullanırken bazen de aynı anda birden fazla strateji kullanması gerekecektir. "*Composite*" kalıbı bu probleme ilişkin çözüm üretir.
- İndirim stratejisi satışla ilgili farklı bilgilere bağlıdır: Zaman (Pazartesi), toplam bedel, müşteri tipi, belli bir satış kalemi.
- Stratejiler birbirleriyle çatışmaktadır.

Çözüm:

Örnekte strateji sınıfları, ortak bir üst sınıftan (ISalePricingStrategy) türetildiği gibi aynı üst sınıftan bir bileşik sınıf da (CompositePricingStrategy) türetilecektir.

CompositePricingStrategy sınıfı, herhangi bir strateji sınıfının nesnelerini içerebilen pricingStrategies adlı bir listeye sahiptir. Belli bir anda geçerli olan stratejiler bu listeye eklenecektir.

Aynı anda birden fazla strateji geçerli olduğunda nasıl davranılacağı da ayrı bir stratejidir.

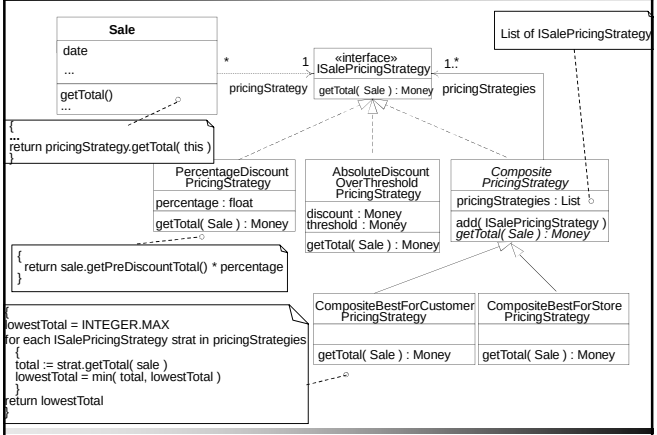
Örneğin müşteri için en iyi indirim seçilerek en düşük fiyat uygulanabilir. Çok gerçekçi olmamakla birlikte dükkan için en iyisi seçilerek indirimler arasında en düşük olan da uygulanabilir.

Aynı anda geçerli olan stratejilerin listesinin nasıl taranacağı da birer strateji sınıfı olarak tasarlanır. Bu sınıflar CompositePricingStrategy sınıfından türetilirler.

Bunlardan CompositeBestForCustomerPricingStrategy adlı sınıf listedeki tüm indirimleri uygulayarak olabilecek en düşük toplam değeri geri döndürür.

Sale, toplamı belirlerken bazen tek bir nesne ile ilişkilendirilebilir (örneğin PercentageDiscountPricingStrategy) bazen de bileşik bir nesne ile ilişkilendirilebilir (CompositeBestForCustomerPricingStrategy).

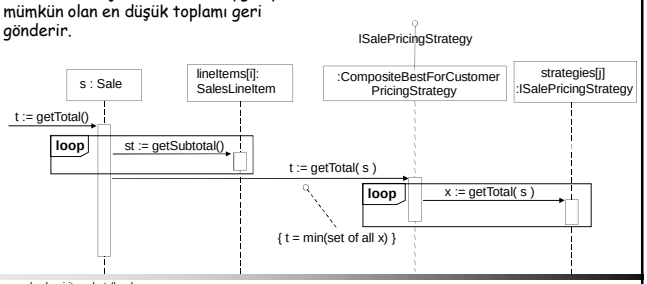
Bağlam nesnesi bu ilişkilendirilmeden habersizdir, çünkü tüm sınıflar ortak bir üst sınıf olan ISalePricingStrategy sınıfından türetilmiştir.



Bu örnekte Sale nesnesi, CompositeBestForCustomerPricingStrategy adlı sınıftan yaratılan nesne ile ilişkilendirilmiştir.

Sale nesnesine getTotal mesajı gönderildiğinde Sale önce tüm satılan kalemlerin fiyatlarını toplar sonra da indirim yapılmak üzere bağlantılı olduğu strateji nesnesine getTotal mesajını gönderir.

Bileşik strateji nesnesi listesindeki tüm indirim stratejilerini tek tek uygulayarak mümkün olan en düşük toplamı geri gönderir.




```
// superclass so all subclasses can inherit a List of strategies
public abstract class CompositePricingStrategy implements ISalePricingStrategy
{
  protected List pricingStrategies = new ArrayList();

  public add( ISalePricingStrategy s )
  {
    pricingStrategies.add( s );
  }
  public abstract Money getTotal( Sale sale );
} // end of class

// a Composite Strategy that returns the lowest total of its inner SalePricingStrategies
public class CompositeBestForCustomerPricingStrategy extends CompositePricingStrategy
{
   public Money getTotal( Sale sale )
   {
     Money lowestTotal = new Money( Integer.MAX_VALUE );
     // iterate over all the inner strategies
     for( Iterator i = pricingStrategies.iterator(); i.hasNext(); )
     {
        ISalePricingStrategy strategy = (ISalePricingStrategy)i.next();
        Money total = strategy.getTotal( sale );
        lowestTotal = total.min( lowestTotal );
      }
      return lowestTotal;
   }
} // end of class
```


Çoklu Stratejilerin Yaratılması

Satış başladığında (yaratıldığında) ilgili fabrika sınıfından (PricingStrategyFactory) bir strateji nesnesi istenir. Fabrika kendi algoritmasına ve koşullara göre belli bir bileşik stratejinin kullanılmasına karar verebilir.

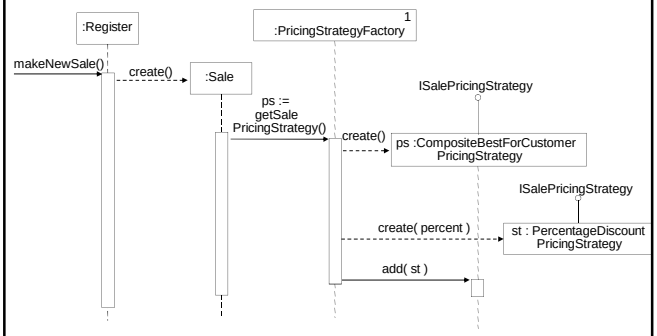
Verilen örnekte Fabrika nesnesi CompositeBestForCustomerPricingStrategy adlı sınıftan bir strateji kullanılmasına karar vermiş ve yarattığı strateji nesnesinin adresini (ps) Sale nesnesine geri göndermiştir.

Satış işlemleri ilerledikçe gerekli olan indirim nesneleri yaratılarak bu listeye eklenebilir.

Satış başladığında dükkanın belli bir sabit indirim stratejisi varsa ilk olarak listeye bu strateji ile ilgili indirim sınıfı eklenebilir.

Örnekte satış başladığında o gün için geçerli olan bir yüzdelik indirimin geçerli olduğu varsayılmış ve PercentageDiscountPricingStrategy nesnesi yaratılarak bileşik nesnenin listesine eklenmiştir.

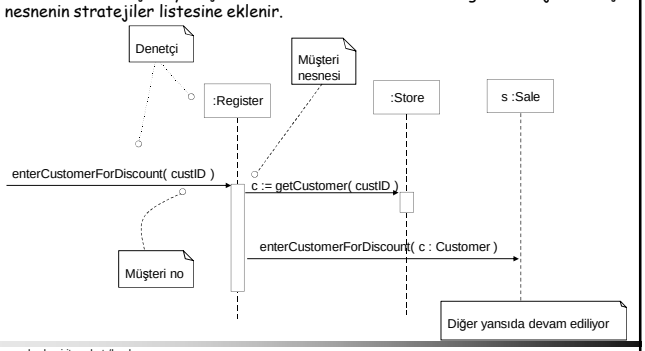
Örnek: Çoklu Stratejilerin Yaratılması ve Kullanılması

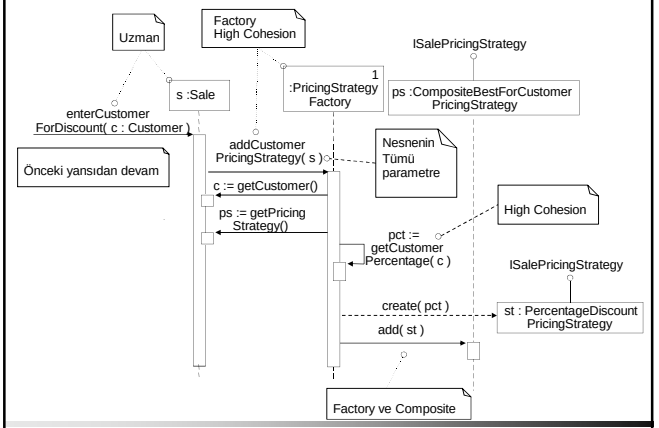


Müşteriye özel stratejinin eklenmesi:

Eğer müşterilere göre özel indirimler yapılıyorsa müşteri bilgilerinin sisteme girilmesini sağlayan bir işlem de tasarlanmalıdır.

Kartı okutulan müşteriye ilişkin özel bir indirim varsa onunla ilgili strateji de bileşik nesnenin stratejiler listesine eklenir.





Tasarımın Değerlendirilmesi

Yansı 8.38 ve 8.39'da gösterilen tasarımı GRASP kalıpları ve tasarım prensipleri açısından inceleyelim:

• Register nesnesi PricingStrategyFactory nesnesine doğrudan mesaj göndermiyor. Yeni strateji yaratılması işi Sale nesnesi üzerinden yapılıyor. Sale zaten PricingStrategyFactory nesnesine bağlı olacağından Register ile PricingStrategyFactory arasında yeni bir bağlantıya gerek yoktur (*Low coupling*).

Ayrıca Sale yeni strateji konusundaki bilgiye sahip olan uzmandır (*Expert*).

• Register müşteri numarası (customerID) ile müşteri (Customer) nesnesinin yaratılmasını Store nesnesinden istemektedir. Store nesnesi bu konun uzmanıdır. Ayrıca bu durum gerçek dünyaya yakınlık (*low representational gap*) açısından da uygundur. Müşteri bilgilerini Register yerine Sale nesnesi Store'dan isteseydi bağımlılık artardı.

•Müşteri numarasının (customerID) müşteri (Customer) nesnesine dönüştürülmesi nesneye dayalı tasarımda sık izlenen bir yoldur.

Tasarımın ilk aşamalarında sadece numara yeterli gibi görünse de ileri aşamalarda müşterinin tüm bilgilerini içeren nesne gerekli olabilir. Bu nedenle tasarımın daha başında, sisteme genellikle bir kullanıcı arayüzünden girişi yapılan kimlik numaraları nesnelere dönüştürülür ve nesneler arasında bu nesnelerin referansları aktarılır.

Bu yöntem bir kalıpta yer almamaktadır.

• addCustomerPricingStrategy(s:Sale) mesajında Sale tipinden s nesnesi fabrika nesnesine gönderiliyor. Daha sonra fabrika kendisine gerekli olan Customer ve PricingStrategy bilgilerini soruyor.

s:Sale nesnesinin tamamını parametre olarak göndermek yerine sadece o anda gerekli olan parçaları da aktarmak mümkün olabilirdi. Ancak nesneye dayalı tasarımda bir nesnenin alt parçalarını aktarmak yerine tamamının referansının aktarılması tercih edilir.

Bu durum tasarımda esneklik sağlar. Tasarımın ileriki aşamalarında yeni parçaların da aktarılması gerektiği ortaya çıksa bile tasarımda fazla değişiklik olmaz.

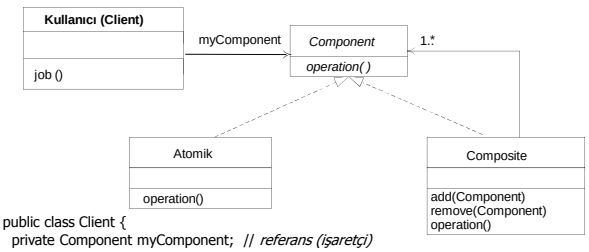
Aktarımlar referanslar (adresler) üzerinden yapıldığından bu durum fazla bilgi akışına neden olmaz. Eğer parametre olarak gönderilen nesnenin değiştirilmesi istenmiyorsa C++'da sabit referans parametreler kullanılmalıdır.

Bileşik Nesne (*Composite*) kalıbı sadece stratejiler ile birlikte kullanılan bir kalıp değildir.

Eğer bir sistemin (veya bir nesnenin), tek nesneleri (atomik) ve çoğul nesneleri (grupları, örneğin liste) benzer şekilde görmesi ve kullanabilmesi isteniyorsa izlenmesi gereken yol "Composite" kalıbında açıklanmıştır.

Derste verilen örnekte atomik ve çoğul nesnelere örnek olarak strateji nesneleri kullanılmıştır.

Bileşik Nesne (*Composite*) kalıbının genel yapısı:




```
public class Client {
  private Component myComponent;  // referans (işaretçi)

  public job()
  {
     .....
     myComponent.operation();  // atomik de olabilir composit de
  }

}
```

6. Ön Yüz (*Facade*) (GoF)

Yazılım geliştirmede karşılaşılan durumlardan biri de belli işler için önceden var olan eski bir sistemin kullanılmasıdır.

Örneğin bazı muhasebe işlemleri için eski bir programın bazı kısımları kullanılıyor olabilir. Büyük bir olasılıkla bu eski programın yerine ileride yenisi alınacaktır.

Bazen de henüz tasarımı yapılmamış ya da tasarımı değişebilecek karmaşık bir alt sistemin bazı hizmetlerinin kullanılması istenebilir.

Böyle durumlarda tasarımı yapılmakta olan yeni sistem ile değişme olasılığı yüksek olan alt sistem arasında bağlantıyı sağlamak için bir ön yüz (cephe) nesnesi hazırlanır. Alt sistemdeki hizmetlere bu cephe üzerinden erişilir.

Karmaşık alt sistemin tüm hizmetlerini kullanmak gerekmiyorsa ön yüzde sadece gerekli olanlar gösterilebilir.

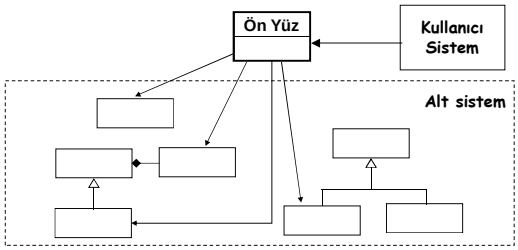
Ayrıca cephenin ardındaki değişiklikler asıl sistemi etkilemez.

Problem: Değişik arayüzlere ve yapılara sahip hizmetleri içeren karmaşık bir alt sistem var. Alt sistem ileride değişebilir ve alt sistemin sadece bazı olanaklarını kullanmak istiyorum. Bu alt sistem ile bağlantı nasıl sağlanmalı?

Çözüm: Alt sistem ile bağlantıyı sağlayan bir ön yüz (*facade*) nesnesi tanımlayın. Ön yüz nesnesi alt sistemdeki tüm hizmetler için tek bir ortak erişim noktası oluşturacaktır.

Ön yüz nesnesinin içine kullanıcı sistemin gerek duyduğu işleri yapan metotlar yerleştirilecektir. Kullanıcı sistem bu metotları çağırdığında metotların içinde alt sistemin gerekli yerlerine erişilecektir.

Böylece alt sistem değiştiğinde sadece ön yüz nesnesi değişecek, kullanıcı sistem bundan etkilenmeyecektir.



Örnek:

Bu kalıbı örnek POS sistemi üzerinde açıklamak için değişken işletme kuralları (*pluggable business rules*) ele alınacaktır.

Problem:

NextGen POS sistemini satın alan şirketler senaryoların önceden belli olan bazı noktalarında sistemin farklı davranışlar göstermesini (farklı kurallara uymalarını) isteyebilirler.

Bu kurallar bazı işlemlerin engellenmesini ya da geçersiz kılınmasını gerektirebilir.

Örneğin:

- Satış başladığında ödemenin hediye çeki ile yapılacağı belirlendiğinde müşterinin sadece bir ürün almasına izin verilmek istenebilir. Bu durumda birinciden sonraki enterItem işlemleri engellenecektir.
- Hediye çeki ile ödeme yapıldığında para üstünün nakit ödenmesi engellenmeli.

Yazılım mimarı, konuların ayrılığı (*seperation of concerns*) prensibine uygun olarak kuralları denetleyen ayrı bir alt sistem (POSRuleEngine) oluşturmak isteyecektir.

Asıl sistem tasarlanırken kurallarla ilgili alt sistemin tasarımı henüz yapılmamış olabilir, ileride kuralların değişmesi, yenilerinin eklenmesi olasılığı olabilir. Verdiği tüm hizmetlere gerek duyulmayan karmaşık bir alt sistem kullanılmak istenebilir.

Bu nedenle kuralların yer aldığı alt sisteme bir ön yüz üzerinden erişilmesi yararlı olacaktır.

Örnekte POSRuleEngineFacade adında "*singleton*" özelliğine sahip bir ön yüz tasarlanacaktır.

Program çalışırken belli bir işlemin geçerli olup olmadığı bu ön yüzdeki fonksiyonlar çağırılarak sınanacaktır.

Ön yüzdeki fonksiyonlar kurallar alt sistemindeki gerekli fonksiyonları çağırarak işlemin geçerliliğine karar vereceklerdir.

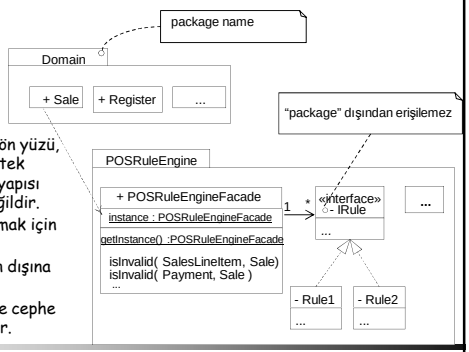
Kural sisteminin değişmesi tasarladığımız sistemi (Sale sınıfını) etkilemeyecektir.

```
public class Sale
{
  public void makeLineItem (ProductSpecification spec, int quantity)
  {
     SalesLineItem sli = new SalesLineItem (spec, quantity);
     // call to the Facade
     if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this) )
        return;              // not accepted
     lineItems.add( sli );   // OK, accepted
  }
// ...
} // end of class
```

Facade Singleton ile erişiliyor

Konuların ayrılığı (*seperation of concerns*) prensibine uygun olarak ilgili nesneler gruplanarak paketler (***package***) oluşturulabilir.

Java'da "package", C++'da ise "namespace" ve "library" kavramları bu gruplamalara karşı düşmektedir. UML'de "package" notasyonu aşağıda gösterilmiştir.



Yandaki örnekte, POSRuleEngineFacade ön yüzü, "kurallar alt sistemi"ne tek giriştir. Alt sistemin iç yapısı dışarıdan erişilebilir değildir.

Yazılım kalitesini arttırmak için iki şey yapılmıştır:

1. Kurallar asıl sistemin dışına çıkartılmıştır.
2. Kurallar alt sistemine cephe üzerinden erişilmiştir.

Adaptör ile Ön Yüz Arasındaki Farklılık

İki kalıp birbirine benzeyen problemlerde kullanılmaktadırlar. Her ikisinde de ilişki kurmamız gereken hazır bir sistem (sınıflar) vardır.

Bu benzerliğe karşın aralarında belirgin farklar vardır:

- Ön yüz kalıbında uyum sağlamaya çalışılan bir ara yüz yoktur.
  Adaptör kalıbında ise tasarlanmakta olan sistemin ara yüzü bellidir ve ilişki kurulacak olan diğer sistemin ara yüzünü ile tasarlanmakta olan bu sistemimin ara yüzü uyumlu hale getirilmesi amaçlanmaktadır.
- Ön yüzde çok şekillilik (*polymorphism*) yoktur.
  Adaptör çoğunlukla birden fazla sistemle ilişki kurulmasını sağladığından çok şekillik ile birlikte kullanılır. Eğer ara yüzü uyumlu hale getirilecek tek bir sistem varsa adaptör kalıbı da çok şekillik olmadan kullanılır.

Ön yüzde asıl amaç karmaşık bir sisteme daha basit bir ara yüzden erişilmesini sağlamaktır. Ayrıca alt sistemin bizi ilgilendirmeyen kısımlarından da yalıtılmış oluruz.

Özet:

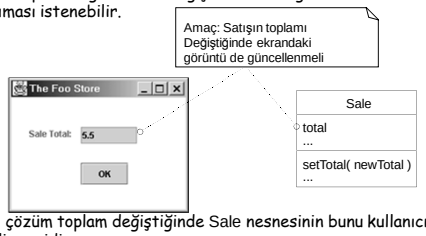
Ön Yüz karmaşık bir alt sistemin arayüzünü basitleştirir, adaptör ise alt sistemin arayüzünü başka bir arayüze dönüştürür.

7. Gözlemci (*Observer*) / Yayıncı-Abone (*Publish-Subscribe*) (GoF)

Bir nesnedeki değer değişimi başka nesneler tarafından bilinmek istenebilir.

Değer değişimi ile ilgilenen nesnelerin sayısı programın çalışması sırasında değişebilir.

Örneğin satışın toplam değerindeki değişimin anında grafik kullanıcı arayüzüne (ekrana) yansıması istenebilir.



Akla gelen bir çözüm toplam değiştiğinde Sale nesnesinin bunu kullanıcı arayüzündeki nesnelere bildirmesidir.

Bu çözümün sorunları vardır:
Bilgi yayımlayan nesne bilgiyi almak isteyenlere bağımlı olur.
Ayrıca yukarıdaki örnekte, uygulama nesneleri (Sale) arayüz (ekran) nesnelerine bağlanmış olurlar.

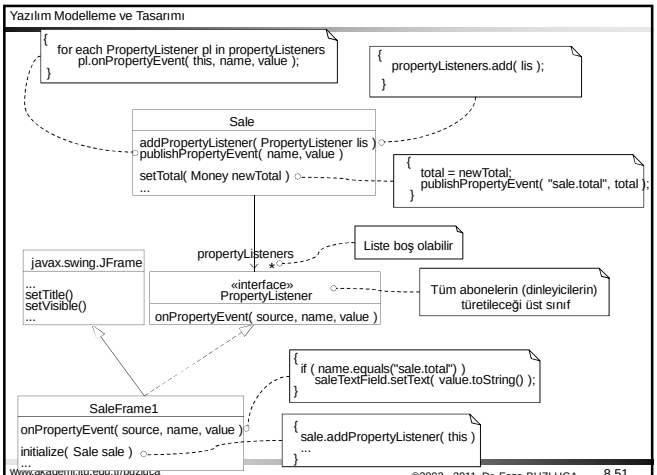
Problem: Değişik tipte abone (*subscriber*) nesneleri, yayıncı nesnelerindeki durum değişikliklerini sezmek ve bu değişimlere kendilerine göre bir tepki göstermek isterler.

Yayıncı nesnenin, abone nesnelerine fazla bağımlı olması istenmez. (*Low coupling*)

Çözüm: Abone nesnelerini ortak bir üstsınıftan (*interface*) türetin. Yayıncı nesne isteyen aboneleri kendi listesine kayıt eder. Yayıncı kendi durumunda bir değişiklik olduğunda (bir olay meydana geldiğinde) listesindeki abonelere bu durumu bildirir.

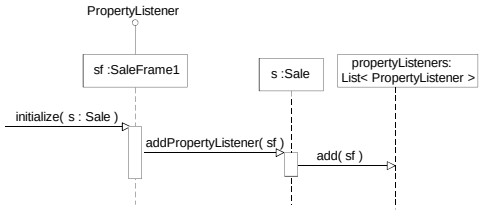
NextGen POS Sistemi için örnek bir çözüm:

1. PropertyListener adında bir üst sınıf (Java: *interface*, C++: *Abstract class*) yaratılır. Bu sınıftan türeyen tüm alt sınıflarda onPropertyEvent adında bir işlem olacaktır.
2. SaleFrame1 adlı ekran-pencere nesnesi üst sınıftan türetilecek ve onPropertyEvent işlemi bu alt sınıfta gerçeklenecek.
3. SaleFrame1 nesnesi yaratılırken bilgi alacağı Sale nesnesinin adresi ona aktarılır.
4. SaleFrame1 nesnesi, addPropertyListener mesajı ile kendini yayıncı Sale nesnesinin listesine kayıt ettirir (abone olur).
5. Durumunda bir değişiklik olduğunda Sale nesnesi kendi PropertyListeners listesindeki tüm nesnelere (abonelere - dinleyicilere) bu değişikliği bildirir.
6. Sale nesnesi, hepsi de aynı PropertyListener üst sınıfından türemiş oldukları için abone nesneleri arasındaki farkı bilmez. Bağımlılık sadece üst sınıfadır.

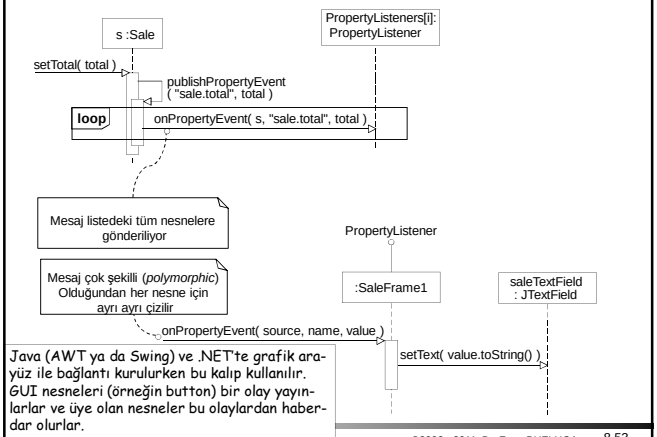


Önceki sınıf diyagramında metotların gövdeleri açıklama kutularına yazılmıştır. Eğer bu yeteri kadar açıklayıcı olmazsa ardışıl diyagramlar da çizilir.

SaleFrame1 gözlemci nesnedir. Yayıncı Sale nesnesine abone olma isteğinde bulunur ve Sale nesnesi tarafından aboneler listesine kayıt edilir.



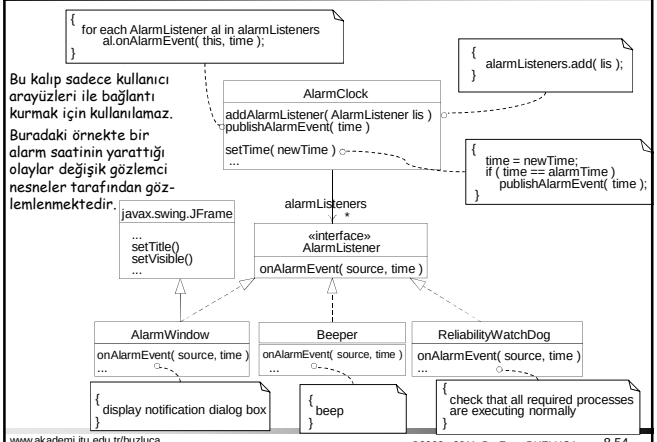
Uygulama nesnesi ile arayüz nesneleri arasındaki bağımlılık en az düzeyde tutulmuştur. Yayıncı nesnenin abone listesi boş da olabilir.



Java (AWT ya da Swing) ve .NET'te grafik arayüz ile bağlantı kurulurken bu kalıp kullanılır. GUI nesneleri (örneğin button) bir olay yayınlarlar ve üye olan nesneler bu olaylardan haberdar olurlar.

Bu kalıp sadece kullanıcı arayüzleri ile bağlantı kurmak için kullanılamaz.

Buradaki örnekte bir alarm saatinin yarattığı olaylar değişik gözlemci nesneler tarafından gözlemlenmektedir.



8. Köprü (*Bridge*) (GoF)

GoF'un tanımına göre köprü kalıbının tarifi:

Soyutlama (*abstaraction*) ile gerçeklemeyi (*implementation*) birbirinden ayrı tutun, böylece ikisini bağımsız olarak değiştirebilirsiniz.

Burada gerçeklemeden kastedilen, bir sınıfın gerçeklenmesi (işlevlerini yerine getirmesi) için kullanılan diğer sınıflardır (*delegation*).

Köprü (*bridge*) kalıbı diğer kalıplardan daha karmaşıktır ve ilk bakışta tarifin yardımıyla anlaşılması zordur, ancak buna karşın yaygın kullanım alanı vardır.

Kalıp aşağıdaki örneğin yardımıyla açıklanacaktır.

Örnek problem:

Dikdörtgenler çizen bir program yazılması istenmektedir.
Ancak dikdörtgenlerin bazıları hazır bir çizim programı kullanılarak (drawing program – DP1), bir kısmı da başka bir hazır çizim programı kullanılarak (drawing program – DP2) kullanılarak, çizilecektir.

Dikdörtgenler iki noktası ile tanımlanırlar:

Örnek problem (devam):

Hazır çizim programlarının içeriği:

	DP1	DP2
Çizgi çizimi:	`draw_a_line( x1, y1, x2, y2)`	`drawline( x1, x2, y1, y2)`
Çember çizimi:	`draw_a_circle( x, y, r)`	`drawcircle( x, y, r)`

Programın müşterisi; dikdörtgenleri kullanan yapının (liste, program parçası), dikdörtgenlerin hangi programla çizildiğinin farkında olmamasını istiyor.

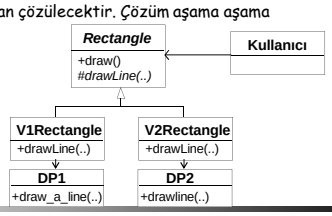
Zaten dikdörtgenler hangi programla çizileceklerini kendileri bilecekler.

Tasarım (Çözüm önerileri):

Problem önce köprü kalıbı kullanılmadan çözülecektir. Çözüm aşama aşama geliştirilecektir.

İlk çözümde aynı soyut sınıftan türeyen iki farklı dikdörtgen sınıfı tasarlanacaktır.

Farklı dikdörtgen sınıfları çizim için farklı sınıfları (DP1 ya da DP2) kullanmaktadırlar.



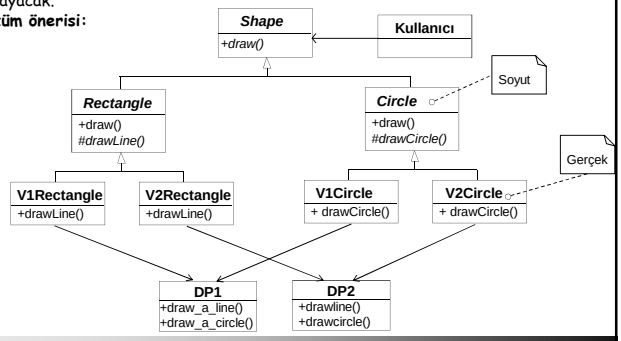
Çözüme ilişkin Java kodu:

```
abstract class Rectangle {
 private double _x1,_y1,_x2,_y2;
 public void draw () {              // Dikdörtgen kendi çiziminden sorumlu
    drawLine(_x1,_y1,_x2,_y1);
    drawLine(_x2,_y1,_x2,_y2);
    drawLine(_x2,_y2,_x1,_y2);
    drawLine(_x1,_y2,_x1,_y1);
 }
 abstract protected void drawLine ( double x1, double y1,
double x2, double y2);
}

class V1Rectangle extends Rectangle {
 drawLine( double x1, double y1, double x2, double y2) {
    DP1.draw_a_line( x1,y1,x2,y2);
 }
}

class V2Rectangle extends Rectangle {
 drawLine( double x1, double y1, double x2, double y2) {
   // arguments are different in DP2 and must be rearranged
   DP2.drawline( x1,x2,y1,y2);
 }
}
```


Örnek problem değişiyor:

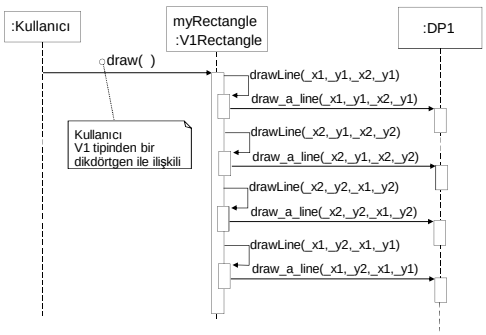
Yeni gereksinimler ortaya çıkıyor.
Çizim programında **çember** desteğinin de olması isteniyor.
Çizim programını kullanacak olan sistem, hangi şekli çizdirdiğini bilmek zorunda olmayacak.

Çözüm önerisi:

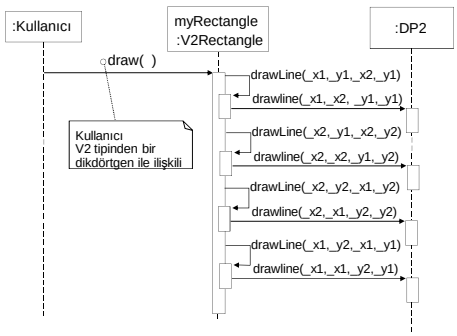


Sistemin çalışması:

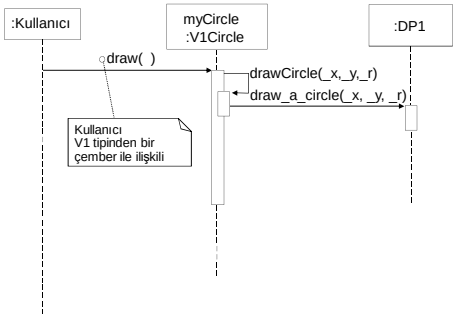
Eğer kullanıcı o anda V1 tipinden bir dikdörtgenle ilişkilendirildiyse bu dikdörtgenin çizdirilmesi aşağıdaki etkileşim diyagramında gösterildiği gibi olur:



Eğer kullanıcı o anda V2 tipinden bir dikdörtgenle ilişkilendirildiyse bu dikdörtgenin çizdirilmesi aşağıdaki etkileşim diyagramında gösterildiği gibi olur:



Eğer kullanıcı o anda V1 tipinden bir çember ile ilişkilendirildiyse çizim aşağıdaki etkileşim diyagramında gösterildiği gibi olur:




```
abstract class Shape {
  abstract public void draw ();
}
abstract class Rectangle extends Shape {
 public void draw () {
  drawLine(corner1x,corner1y,corner2x,corner2y);
  drawLine(corner1x,corner1y,corner2x,corner2y);
  drawLine(corner1x,corner1y,corner2x,corner2y);
  drawLine(corner1x,corner1y,corner2x,corner2y);
 }
 abstract protected void
       drawLine( double x1, double y1,
                 double x2, double y2);
 private double corner1x, corner1y, corner2x,
corner2y;
}

class V1Rectangle extends Rectangle {
 void drawLine ( double x1, double y1,
                 double x2, double y2)
{
   DP1.draw_a_line( x1,y1,x2,y2);
  }
}
class V2Rectangle extends Rectangle {
 void drawLine (double x1, double x2,
                double y1, double y2)
{
   DP2.drawline( x1,x2,y1,y2);
}
```

```
abstract class Circle extends
Shape {
  public void draw () {
    drawCircle( cornerX, cornerY,
radius);
  }
  abstract protected void
  drawCircle (double x, double y,
double r);
   private double cornerX, cornerY, radius;
}
class V1Circle extends Circle {
 void drawCircle(x,y,r) {
   DP1.draw_a_circle( x,y,r);
 }
}
class V2Circle extends Circle {
 void drawCircle(x,y,r) {
   DP2.drawcircle( x,y,r);
 }
}
```


Çözümün değerlendirilmesi:

Önceki çözüm nesneye dayalı olmakla beraber bazı sorunlar içerdiğinden iyi bir çözüm değildir.

Sorun:

Çizgi ve çember çizimleri için DP1 ve DP2 dışında üçüncü bir çizim paketi DP3 kullanılmaya karar verilse her şekil için bir sürüm (V3) daha yaratmak gerekecek ve farklı şekil sınıfı sayısı altıya çıkacak.

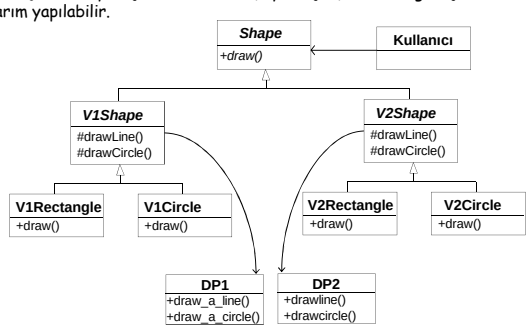
Çemberden sonra çizim sistemine yeni bir şekil eklenmek istense üç şekil sınıfı daha eklemek gerekecek (V1xx, V2xx, V3xx).

Bu tasarımdaki sorun, soyutlama (kavramsal şekiller) ile onların gerçeklenmesinin (çizim programlarının) birbirlerine çok sıkı bağlı olmasıdır. Bu nedenle yeni bir kavram ya da yeni bir gerçekleme biçimi eklenmek istediğinde sınıf sayısı çok artmaktadır.

Kalıtımın (*inheritance*) gereğinden fazla kullanılması tasarımın kalitesini düşürmektedir.

Başka Bir Çözüm Önerisi:

Önceki çözümde yanlış kalıtım zinciri (hiyerarşisi) kullanıldığı düşünülerek aşağıdaki tasarım yapılabilir.



Ancak bu yeni bakış açısı da sınıf sayısının artışı ile ilgili problemi çözmez.
Soyutlama ile gerçekleme hala sıkı biçimde bağlıdır.

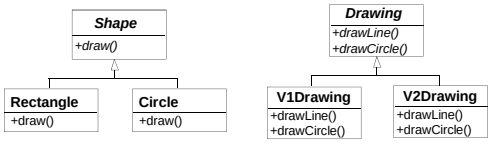
Uygun Çözüm:

Köprü (*Bridge*) kalıbını bilmeden iki önemli tasarım prensibini kullanarak da uygun bir çözüm geliştirmek mümkün olabilirdi. Bu prensipler:

- "*Find what varies and encapsulate it*".
- "*Favor composition over inheritance*".

Bizim örneğimizde değişik tipte şekiller ve değişik tipte çizim programları var. Buna göre paketlenmesi gereken ortak kavramlar şekiller ve çizim programlarıdır.

Burada **Shape** sınıfı değişik şekillerin ortak özelliklerini toplamaktadır. Şekiller kendilerini ekrana nasıl çizeceklerini bilmek sorumluluğuna sahiptirler (+draw()). **Drawing** nesneleri ise çizgi ve çember çizimini (drawLine(), drawCircle()) bilmektedirler.



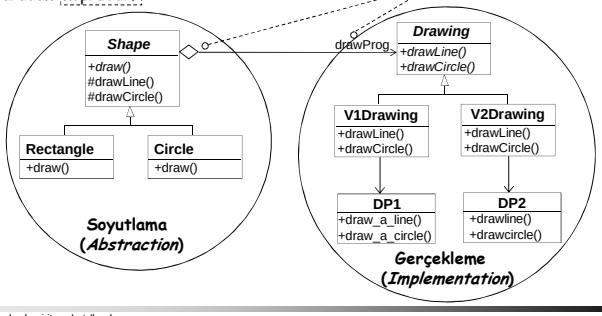
İki grup arasındaki ilişki belirlenirken iki olasılık vardır:

Ya şekiller çizim programlarını kullanacaktır ya da çizim programları şekilleri kullanacaktır.

İlk olasılık nesneye dayalı prensipler açısından daha uygundur.

Şekiller kullanılacak olan gerçekleme yöntemine işaret eden bir işaretçi içerirler.

Bu aradaki köprüdür.



Çözümün C++ Kodu:

```
//DP1 ve DP2 daha önceden hazırlanmış olan çizim arşivleri
class DP1 {   // Birinci grup çizim programları
public:
     void static draw_a_line (double x1, double y1, double x2, double y2);
     void static draw_a_circle (double x, double y, double r);
};
class DP2 {   // İkinci grup çizim programları
public:
    void static drawline (double x1, double x2, double y1, double y2);
    void static drawcircle (double x, double y, double r);
};

// Varolan arşivleri kullanan çizim programları (implementation)
class Drawing {          // Bütün çizim programlarının türeyeceği soyut sınıf
public:
   virtual void drawLine (double, double, double, double)=0;
   virtual void drawCircle (double, double, double)=0;
};
```



```
class V1Drawing : public Drawing {   // Birinci grubu (DP1) kullanan sınıf
public:
  void drawLine (double x1, double y1, double x2, double y2);
  void drawCircle( double x, double y, double r);
};

void V1Drawing::drawLine ( double x1, double y1,        double x2, double y2) {
      DP1::draw_a_line(x1,y1,x2,y2);  // DP1'i kullanıyor
}
void V1Drawing::drawCircle (double x, double y, double r) {
     DP1::draw_a_circle (x,y,r);
}

class V2Drawing : public Drawing {      // İkinci grubu (DP2) kullanan sınıf
public:
     void drawLine (double x1, double y1, double x2, double y2);
     void drawCircle(double x, double y, double r);
};
void V2Drawing::drawLine (double x1, double y1,        double x2, double y2) {
     DP2::drawline(x1,x2,y1,y2); // DP2'yi kullanıyor
}
void V2Drawing::drawCircle (double x, double y, double r) {
    DP2::drawcircle(x, y, r);
}
```



```
// Çizim programlarını kullanacak olan şekiller (Abstraction)
class Shape {                  // Bütün şekillerin türeyeceği soyut sınıf
public:
    Shape (Drawing *);         // Kurucu: Kullanacağı çizim programını alacak
    virtual void draw()=0;
protected:
    void drawLine( double, double, double , double);
    void drawCircle( double, double, double);
private:
   Drawing *drawProg;                 // Bağlı olduğu çizim programı (köprü)
};
Shape::Shape (Drawing *dp) {   // Kurucu: Kullanacağı çizim programına bağlanıyor
     drawProg= dp;
}
void Shape::drawLine( double x1, double y1, double x2, double y2){
    drawProg->drawLine(x1,y1,x2,y2);
}
void Shape::drawCircle(double x, double y, double r){
    drawProg->drawCircle(x,y,r);
}
```



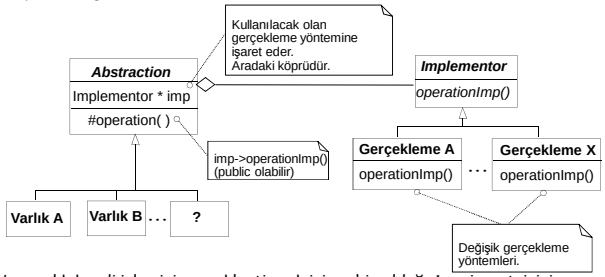
```
//Gerçek şekil sınıfları
class Rectangle : public Shape{
public:
    Rectangle (Drawing *, double, double, double, double);
    void draw();
private:
    double _x1,_y1,_x2,_y2;
};

Rectangle::Rectangle (Drawing *dp, double x1, double y1,
                                   double x2, double y2) : Shape(dp) {
   _x1= x1; _y1= y1;
   _x2= x2; _y2= y2;
}

void Rectangle::draw () {   // Hangi programa bağlandıysa onu kullanıyor
    drawLine(_x1,_y1,_x2,_y1);
    drawLine(_x2,_y1,_x2,_y2);
    drawLine(_x2,_y2,_x1,_y2);
    drawLine(_x1,_y2,_x1,_y1);
}
```



```
class Circle : public Shape{
public:
   Circle (Drawing *, double, double, double);
   void draw();
private:
   double _x, _y, _r;
};
Circle::Circle (Drawing *dp,double x, double y, double r) : Shape(dp) {
   _x= x;  _y= y; _r= r;
}
void Circle::draw () {
   drawCircle( _x, _y, _r);
}
int main () {                          // Test amaçlı ana program
   Shape *s1, *s2;
   Drawing *dp1, *dp2;
   dp1= new V1Drawing;
   s1=new Rectangle(dp1,1,1,2,2); // Şekil dp1 programına bağlanıyor
   dp2= new V2Drawing;
   s2= new Circle(dp2,2,2,4);      // Şekil dp2 programına bağlanıyor
   s1->draw();                     // Kullanıcı hangi şekle mesaj gönderdiğini bilmiyor.
   s2->draw();                     // Ayrıca şeklin hangi çizim programını kullandığını da bilmiyor.
   delete s1; delete s2;
   delete dp1; delete dp2;
   return 0;
}
```

Gerçek bir yazılımda bu adresler bir fabrikadan alınabilir.

Köprü (*Bridge*) Kalıbının Genel Şeması:



Her varlık kendi işlemini gerçekleştirmek için sahip olduğu **imp** işaretçisinin gösterdiği gerçekleme nesnesi ile mesajlaşır. Bu işaretçi aradaki bağlantıyı sağlayan köprüdür.
Varlıklar gerek duyduklarında bir fabrika sınıfından ilgili gerçekleme nesnesinin adresini isterler.
Bu yapıda varlıkların ve gerçekleme yöntemleri birbirlerinden bağımsız olarak tasarlanmışlardır.

9. Dekoratör (*Decorator*) (GoF)

Bir sınıfın belli bir sorumluluğu (davranışı) çeşitli koşullara göre değişik biçimlere giriyorsa her davranış biçimi bir **strateji** olarak gerçeklenip bağlam (*context*) sınıfının dışında ayrı sınıflar olarak tasarlanıyordu (bkz. Strateji kalıbı).

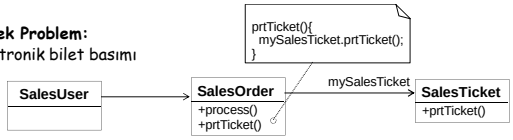
Ancak bir nesnenin davranışı bir dizi işlemden oluşuyorsa ve bu işlemlerin sırası ve sayısı belli koşullara göre değişiyorsa strateji kalıbı gerekli çözümü sağlayamaz.

Örneğin bir nesnenin D() davranışı (sorumluluğu) bazen d1,d2,d3 işlemlerinden, bazen d3,d1; bazen d4,d1,d3 işlemlerinden oluşabilir.

Bu problem için dekoratör kalıbı bir çözüm önermektedir.

Kalıbın temel işlevi bir nesneye dinamik olarak yeni davranışlar (sorumluluklar) eklenmesini (ya da çıkarılmasını) sağlamaktır.

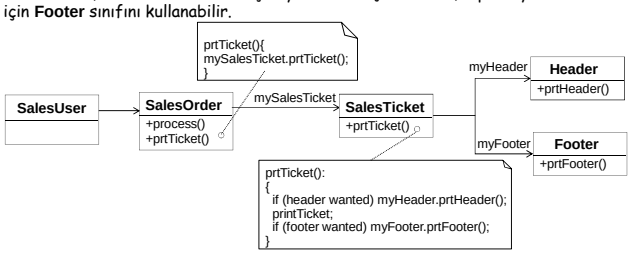
Örnek Problem:
Elektronik bilet basımı



SalesUser , **SalesOrder** sınıfını kullanan bir sınıftır. **SalesOrder** sınıfı biletleri basmak için **SalesTicket** sınıfından yararlanır (*delegation*).

Problemin devamı:

Bazı biletlerin başına (*header*) bazılarının da sonuna (*footer*) özel bilgilerin basılması istenebilir.
Bu durumda, **SalesTicket** sınıfı başlık yazdırmak için **Header**, dip not yazdırmak için **Footer** sınıfını kullanabilir.



Ancak birden fazla farklı başlık ve dip not varsa ve değişik zamanlarda bunların değişik kombinasyonlarının yazdırılması isteniyorsa bu tasarım esnek bir çözüm olmaz; prtTicket() metodunun zaman içinde değiştirilmesi gerekir.

Dekoratör kalıbı ile çözüm:

Dekoratör kalıbına göre tüm alt işlemler (başlık, dip not) ayrı bir dekoratör sınıf olarak tasarlanırlar.

Programın çalışması sırasında, dekoratör sınıflarından yaratılan nesneler istenen sırada bir listeye yerleştirileceklerdir.

Aynı arayüze sahip olmaları ve ortak bir listede yer alabilmeleri için hepsi ortak bir soyut Dekoratör sınıfından türetilir.

Ana işlemi (bilet basımı) oluşturan sınıf bir somut bileşen (*concrete component*) sınıfı olarak tasarlanır.

Bu sınıf da diğerleri ile aynı listeye gireceğinden tüm sınıflar ortak bir soyut bileşen (*component*) sınıfından türetilir.

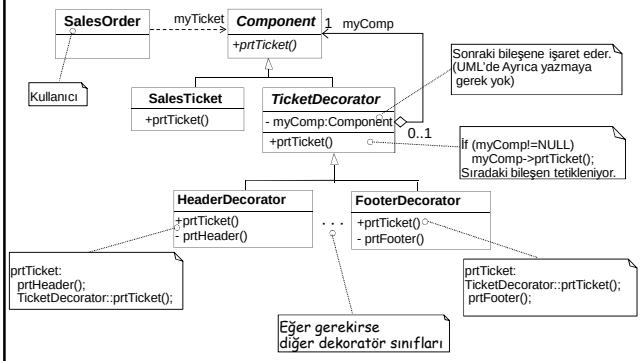
Kullanıcı sınıf (SalesOrder), bileşenlere işaret edebilen bir işaretçiye sahiptir.

Alt işlemler hangi sırada yapılacaksa ona göre bir liste oluşturulur ve listedeki ilk elemanın adresi kullanıcı sınıftan yaratılan nesneye verilir.

Kullanıcı nesne ilk dekoratörü (alt işlemi) canlandırır. Her dekoratör listede kendisinden sonra gelenin metodunu canlandırır.

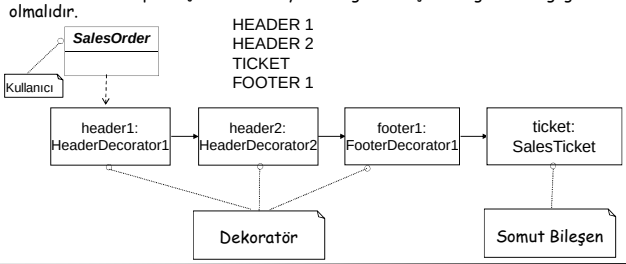
Listenin uygun şekilde oluşturulması ve ilk elemanın adresinin kullanıcı nesneye sunulması bir dekoratör fabrikasının sorumluluğu olacaktır.

Örnek problemin dekoratör kalıbı ile çözüm:



Programın çalışması sırasında alt işlemlerin hangi sırada yapılması isteniyorsa bileşen (*component*) nesneleri o sırada birbirlerine bağlanarak ilk nesnesinin adresi kullanıcı nesneye verilecektir.

Bileşenler birbirlerine bağlanırken baştaki bileşenler somut bir bileşen (*concrete component*) ya da dekoratör, ortadakiler dekoratör, en sondaki bileşen ise somut bir bileşen (*concrete component*) olmalıdır.
Örnek problemde; aşağıda gösterildiği gibi önce iki başlık, ardından bilet ardından da bir dip not çıkması isteniyorsa bağlantıda şekilde gösterildiği gibi olmalıdır.



Örnek Program:

Bu programda Header1, Header2 (benzer şekilde Footer1ve Footer2) sınıflarının birbirlerinden farklı işlemler yaptıkları varsayılarak ayrı dekoratörler olarak tasarlanmışlardır.
Aslında bu probleme özgü olarak sadece bilete basacakları yazı farklı olacaksa dekoratörün içine yazı ile ilgili bir değişken koyup tek bir Header ve tek bir Footer dekoratörü tasarlamak yeterli olurdu.

Örnek C++ Kodu:

```
class Component {                          // Sanal (Abstract) bileşen sınıfı
  public:
     virtual void prtTicket()=0;
};

class SalesTicket : public Component{      // Somut (Concrete) bileşen sınıfı
  public:
    void prtTicket(){                      // Ana işlem
        cout << "TICKET" << endl;
    }
};
```

```
class Decorator : public Component {     // Dekoratörlerin türetiminin başlangıç noktası
  public:
    Decorator( Component *myC){          // Kurucu (Constructor)
      myComp = myC;                      // Sonraki bileşenin adresini alıyor
    }
    void prtTicket(){                    // Sonraki bileşenin metodunu çağırıyor
        if (myComp!=0)
            myComp-> prtTicket();
    }
    private:
      Component *myComp;
};

class Header1 : public Decorator {       // Header1 dekoratörü
  public:
    Header1(Component *);
    void prtTicket();
};
Header1::Header1(Component *myC):Decorator(myC){}
void Header1::prtTicket(){
    cout << "HEADER 1" << endl;          // Header1'e özgü işlem
    Decorator::prtTicket();              // Üst sınıftan gelen metod çağırılıyor.
}
```



```
class Header2 : public Decorator {       // Header2 dekoratörü
  public:
    Header2(Component *);
    void prtTicket();
};
Header2::Header2(Component *myC):Decorator(myC){}
void Header2::prtTicket(){
        cout << "HEADER 2" << endl;
        Decorator::prtTicket();
}

class Footer1 : public Decorator {       // Footer1 dekoratörü
  public:
    Footer1(Component *);
    void prtTicket();
};
Footer1::Footer1(Component *myC):Decorator(myC){}
void Footer1::prtTicket(){
        Decorator::prtTicket();
        cout << "FOOTER 1" << endl;

}
```

Footer2 benzer şekilde yazılır.


```
class SalesOrder {                       // Sınama amaçlı kullanıcı sınıf
   Component *myTicket;                  // Kullanacağı bilet sınıfına (bileşenine) işaret edecek
public:
   SalesOrder(Component *mT):myTicket(mT){}
   void prtTicket(){
       myTicket->prtTicket();
   }
};

int main()  // Test amaçlı ana program
{
   SalesOrder sale(new Header1(new Header2(new Footer1(new SalesTicket()))));
   sale.prtTicket();
   return 0;
}
```

Gerçek bir yazılımda bu adres fabrika sınıfından istenir

Dekoratörlerin listesi oluşturuluyor. Bu işlem başka bir sınıf (örneğin bir fabrika sınıfı) tarafından yapılabilir. Kullanıcı sınıf ilgili işleme (burada bilet basımı) gerek duyduğunda fabrikadan istekte bulunur.

Dekoratör Kalıbının Genel Şeması: