BILGISAYAR MUHENDISLIĞI TASARIM

Dr.Öğr.Üyesi Betül AY

NESNE YÖNELİMLİ ANALİZ VE TASARIM

Nesne Yönelimli Düşünme

- Nesneye yönelik düşünme, eldeki sorunları veya kavramları incelemeyi, onları parçalara ayırmayı ve bunları nesneler olarak düşünmeyi içerir.
- Örneğin, Twitter'da bir tweet veya çevrimiçi bir alışveriş web sitesinde bir ürün nesneler olarak kabul edilebilir.
- Kodunuzdaki şeyleri temsil etmek için nesneleri kullanarak kodunuz düzenli (**organized**), esnek (**flexible**) ve yeniden kullanılabilir (**reusable**) olur.
    1. *Nesneler, ilgili ayrıntıları ve belirli fonksiyonları farklı, kolayca bulunabilen yerlere yerleştirerek kodu düzenli tutarlar.*
    2. *Nesneler kodu esnek tutar, böylece kodun kalanını etkilemeden ayrıntılar nesne içinde modüler bir şekilde kolayca değiştirilebilir.*
    3. *Nesneler, oluşturulması gereken kod miktarını azalttıklarından ve programları basit tuttuklarından kodun yeniden kullanılmasına izin verir.*

Nesne Yönelimli Düşünme

- Nesneler, cansız nesneler olsalar bile, yazılım üretiminde farkındadırlar (**self-aware**).
- Örneğin, bir cep telefonu özelliklerini “bilir”.
- Benzer şekilde, nesne yönelimli modellemede, sandalye gibi bir nesne boyutlarını ve yerini bilecektir.
- Nesne yönelimli düşünmede canlı ya da cansız her şey nesne kabul edilir.
- Nesne cansız olsa bile davranışının ve özelliklerinin farkındadır.

Yazılım Sürecinde Tasarım

- Yazılım geliştirildiğinde, genellikle bir süreçten (**process**) geçer.
- Basit bir ifadeyle, bir process bir problem alır ve yazılımı içeren bir çözüm yaratır.
- Bir process yinelemelidir (**iterative**).
- Bu iterasyonlar tespit edilen problemlere dayalı bir grup gereksinimden oluşmaktadır ve bu gereksinimleri kavramsal tasarım modelleri (**conceptual design mock-ups**) ve teknik tasarım diyagramları (**technical design diagrams**) oluşturmak için kullanmaktadır.
- Bu process her bir ihtiyaç grubu için tekrarlanır ve sonuçta proje için eksiksiz bir çözüm oluşturulur.
- Bu process atlandığında, özellikle iş derhal kodlama ile başladığında ve gereksinimlerin ve tasarımın anlaşılmaması durumunda birçok proje başarısız olur ☹

Yazılım Sürecinde Tasarım

Standish Grubu (https://www.standishgroup.com/) tarafından yapılan bir ankette, ankete katılanların% 13'ü eksik gereksinimlerin projelerini bozduğunu belirtti! Doğrudan uygulama çalışmasına dalmak, projenin başarısız olmasının önde gelen bir nedenidir.

Yazılım Sürecinde Gereksinim ve Tasarım

Gereksinim (Requirements)

- Gereksinimler, müşteri (client) veya kullanıcı isteğine (user request) bağlı olarak bir üründe uygulanması gereken şartlar veya yeteneklerdir.
- Bunlar bir projenin başlangıç noktasıdır, "müşterinizin ne istediğini anlamalısınız".
- Gereksinimleri ortaya çıkarmak için, müşterinin vizyonundan daha fazlasını sormak ve istemek önemlidir:
  - *Müşteri vizyonunun araştırılması*
  - *Neyi eksik söylediğinin ortaya çıkarılması*
  - *Müşterinin göz önünde bulundurmadığı sorunlar hakkında sorular sorulması…*
- Kodlamaya başlamadan önce ne ürettiğinizi ve müşterinizin bir üründen ne istediğini tam olarak anlamanız gereklidir.

Yazılım Sürecinde Gereksinim ve Tasarım

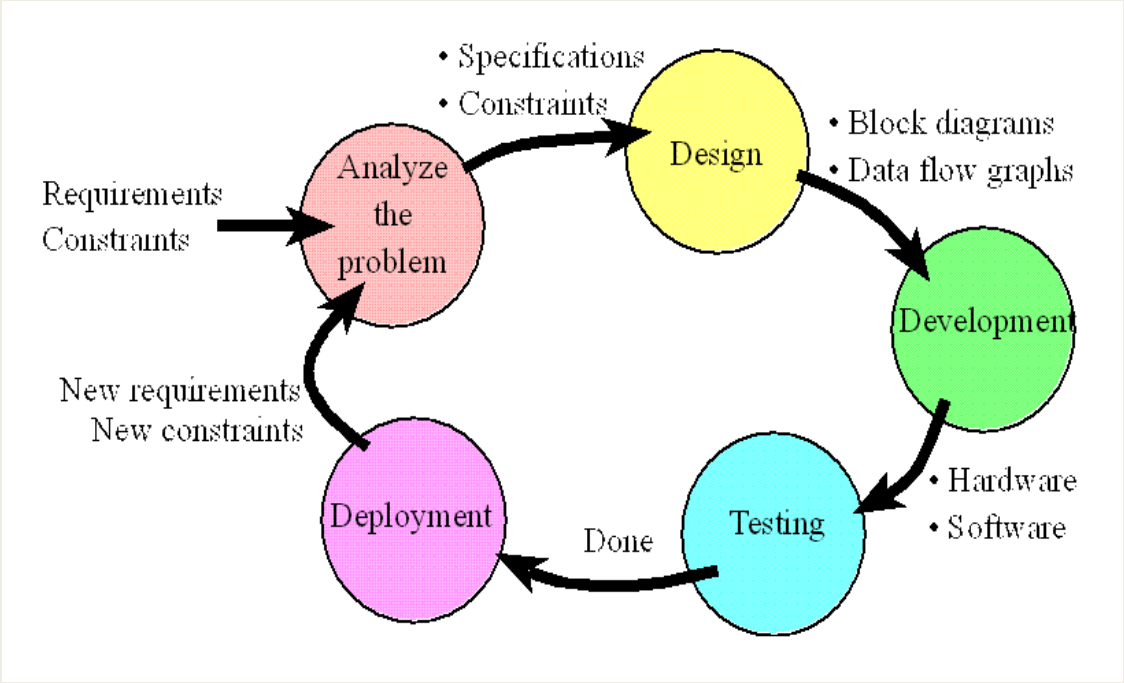
Tasarım (Design)

- Gereksinimler oluşturulduktan sonra bir sonraki adım kavramsal ve teknik tasarımdır.
- Kavramsal tasarım bir ürünün nasıl çalışacağını göstermek ve tartışmak için basit bir yol sağlayarak müşteriler ve kullanıcılar ile tasarım kararlarını netleştirmeye yardımcı olur.
- Teknik tasarımlar, çözümün teknik ayrıntılarını tanımlamak için kavramsal tasarımlar ve gereksinimler üzerine kuruludur.
- Kavramsal tasarımda, geliştirilmekte olan yazılımın ana bileşenleri ve bağlantıları ile bunların ilişkili sorumlulukları ana hatlarıyla belirtilmiştir.
- Teknik tasarım bu bilgiyi bir sonraki aşamaya getirir - bu sorumlulukların nasıl yerine getirildiğini açıklamayı amaçlar.

Yazılım Sürecinde Gereksinim ve Tasarım

- Sistem geliştirme sürecinin bir life cycle olarak ele alın.
- Gereksinimleri alın ve problemleri analiz edin.
- Çözümü tanımlamak için bir algoritma oluşturun.
- Algoritmayı sözde kod(pseudo code) ya da akış diyagramı ile gösterin.
- Oluşturduğunuz algoritmayı koda çevirin.
- Kodunuzu test edin.

(Test—>Kod Yaz—>Test—>Kod Yaz…)



http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C7_DesignDevelopment.htm

Kalite Nitelikleri için Tasarım

- Yazılım geliştirilirken istenilen gereksinimlerin nasıl elde edileceği üzerine ayrıntılara yoğunlaşmak önemlidir. Bu nedenle bu derste yazılım oluşturmada gerekliliklerin ve tasarımın önemine dikkat çekilecektir.
- Tasarım konusunda uzlaşma gerektiren bazı kısıtlamalar (restrictions) vardır.
- İstenilen işlevselliğe dayalı yazılım gereksinimlerinin yanı sıra, bu işlevselliğin ne kadar iyi çalışması gerektiğini tanımlayan kalite özellikleri de vardır.
- Örneğin, bir evin ön kapısını tasarlamayı düşünelim. Güvenlik, önemli olabilecek kalite niteliğidir, ancak kapıya çok fazla kilit eklerseniz, kolayca açılması zor olabilir ve kullanımı zor olabilir. İyi bir tasarım güvenliği kolaylık ve performansla dengelemelidir.

Bağlam ve Sonuçlar (Context and Consequences)

- **Bağlam**, tasarımdaki kalitenin dengesine karar verirken önemli bilgiler sağlar.

Örneğin, halkın erişebileceği kişisel bilgileri depolayan yazılımlar, yalnızca şirket çalışanları tarafından kullanılan yazılımlardan farklı güvenlik gereksinimlerine sahip olacaktır.

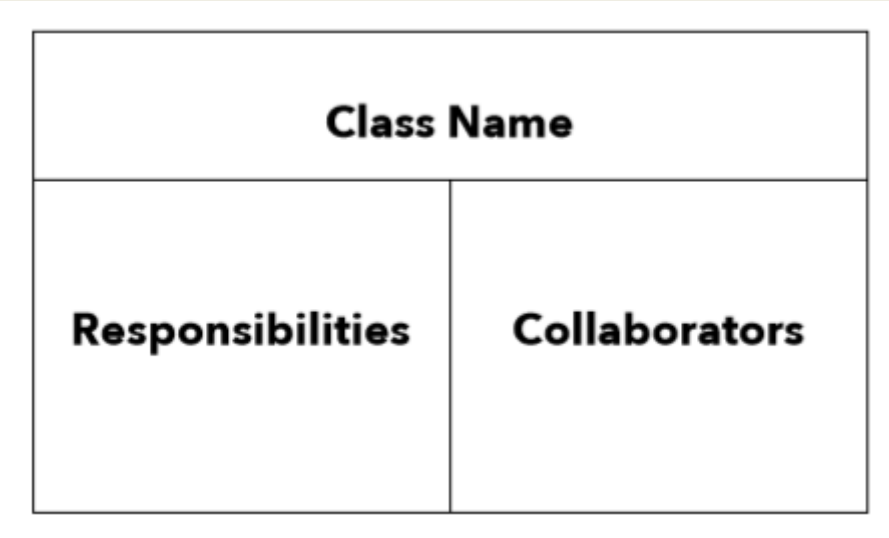
- Yazılım tasarımı **sonuçları** da dikkate almalıdır. Bazen, yazılım tasarımında yapılan seçimlerin istenmeyen sonuçları olabilir.

Örneğin, az miktarda veri için iyi işleyen bir fikir, büyük miktarda veri için pratik olmayabilir.

- Nitelikler arasındaki denge (balance), tasarım sırasında anlaşılmalı ve dikkate alınmalıdır. Hangi niteliklerin gerekli olduğunu öncelik sırasına koymak ve anlamak önemlidir.
- Ürünü geliştirmek için kalitelerin maliyet, zaman ve insan gücü gibi kaynaklarla dengelenmesi gerekir.

Sınıf Sorumluluk İşbirliği (Class Responsibility Collaborator)

Kavramsal tasarımı oluştururken bileşenleri (components), sorumlulukları (responsibilities) ve bağlantıları (connections) yüksek düzeyde temsil etmeye yardımcı olacak önemli bir teknik vardır. Bu teknik, Class, Responsibility, Collaborator (CRC) kartlarının kullanımıdır. CRC kartları üç bölümden oluşur, sınıf adı, sınıfın sorumlulukları ve ortak çalışanlar.



ÖRNEK CRC

- CRC kartlarının birlikte nasıl çalıştığını daha iyi anlamak için, farklı kurslara kayıtlı öğrencileri modellememiz gereken basit bir öğrenci kayıt problemini düşünelim.
- Öğrenci CRC Kartını, öğrencinin adı, öğrenci kimliği ve öğrencinin bir derse kaydolmasını veya dersi bırakmasını sağlayan sorumluluklar gibi özellikleri kolayca tanımlayabiliriz.
- Bu örnekte, işbirlikçi sınıf “ders” sınıfı olacaktır.
- Ders CRC Kartı, ders kodu, ders adı gibi soumlulukları ve “eğitmen” gibi işbirlikçi sınıfından oluşacaktır.

Öğrenci ve Ders CRC Kartı

Öğrenci	
Öğrenci İsmi Öğrenci No Bölümü Ders Kayıt Ders Çıkarma	Ders Sınıfı

Ders	
Ders İsmi Ders Kodu Ders İçeriği Ders Eğitmeni	Eğitmen Sınıfı

CRC Kartlarının Avantajları

- Karmaşık bir sistemi kolayca anlamamızı sağlar. Temelde, işbirlikçi sınıflar arasındaki etkileşimleri birer birer yavaş yavaş inşa ederek daha karmaşık tasarımlar oluşturmaya izin verir.
- CRC kartları basit bir tekniktir ve çok az eğitim almış herkes tarafından kolayca kullanılabilir ve pahalı bilgi işlem kaynakları gerektirmez (bir tahta veya bir kağıt ve bir kalem yeterli olacaktır).
- CRC temel olarak, bir takımdaki farklı kişilerin birlikte çalışarak tasarımı geliştirmelerini ve takımdaki herkesin katkıda bulunmalarını sağlayan bir beyin fırtınası aracıdır.
- CRC, Extreme Programming gibi diğer resmi nesne yönelimli tasarım metodolojileriyle kullanılabilir ve Unified Modeling Language (UML) gibi modelleme dilleriyle birlikte kullanılabilir.

Nesne Yönelimli Modelleme

Tasarımda Modellerin Oluşturulması

- Bir yazılım geliştirme projesi üzerinde çalışırken, sorunu çözmek için kod oluşturma aşamasına geçmemek önemlidir.
- Tasarım adımı, gereksinimlerinizi anlamak ve ürünü oluşturmak arasındadır. Yinelemeli olarak hem problem alanı hem de çözüm alanı ile ilgilenir.
- Tasarım, yazılım geliştirmede çok önemli bir adımdır ve bu süreci kolaylaştırmak için zaman içinde geliştirilen birçok yaklaşım vardır.

Örneğin, belirli tasarım problemleri için bazı tasarım stratejileri ve programlama dilleri oluşturulmuştur.

- Tasarım sürecini kolaylaştırmak için bir yaklaşım nesne yönelimli yaklaşımdır.
- Bu, problemdeki kavramların ve çözüm uzaylarının nesneler olarak tanımlanmasına izin verir - nesneler hem kullanıcılar hem de geliştiriciler tarafından anlaşılabilecek bir kavramdır, çünkü nesne yönelimli düşünme birçok alana uygulanır.

Tasarımda Modellerin Oluşturulması

- Nesne yönelimli tasarım iki adımdan-kısımdan oluşur:

1. **Kavramsal tasarım**, problemdeki temel nesneleri tanımlamak için nesne yönelimli analiz kullanır ve sorunu yönetilebilir parçalara ayırır.
2. **Teknik tasarım**, nitelik ve davranışları dahil nesnelerin ayrıntılarını daha da hassaslaştırmak için nesne yönelimli tasarım kullanır, bu nedenle geliştiricilerin çalışan yazılım olarak kullanması için yeterince açıktır.

- Yazılım tasarımı sırasında amaç, yazılımın tüm nesnelerinin "modellerini" oluşturmak ve geliştirmektir.
- Yazılım modelleri, nesneler için tasarım sürecini anlamanıza ve düzenlemenize yardımcı olur.
- Modeller ayrıca yazılımınız için tasarım dokümantasyonu olarak da kullanılır. Aslında, modeller genellikle Java gibi nesne yönelimli bir dil için iskelet bir kaynak kod gibidir.

Tasarımda Modellerin Oluşturulması

- Yazılım modelleri genellikle, UML olarak adlandırılan görsel bir gösterimde ifade edilir.
- Nesneye yönelik modelleme, farklı yazılım konularına odaklanmak için kullanılabilecek farklı model veya UML diyagramlarına sahiptir.
- Örneğin, hangi nesnelerin ne yaptığını ve nasıl ilişki kurduğunu tanımlamak için yapısal bir model kullanılabilir.
- Artık modellerin tasarımda oynadığı roller ve modeller ile kodlama dilleri arasındaki ilişkiyi anladığımıza göre, şimdi programlama dillerinin tarihini inceleyeceğiz.

Programlama Dillerinin Gelişimi

- Dil, düşünceleri ve fikirleri birbirlerine iletmek için bir sistemi tanımlamada kullandığımız bir kelimedir.
- Yazma, okuma, konuşma, resim çizme ve jest yapma tamamen dilin bir parçasıdır!
- “Hayatta kalmak” ve insanlar tarafından kullanılmak için diller sürekli gelişiyor olmalıdır.
- Programlama dilleri de tıpkı geleneksel diller gibi zaman içinde gelişmiştir.
- Programlama paradigmalarının tarihini bilmek önemlidir.
- Bir yazılım geliştiricisi olarak, daha eski dilleri ve tasarım paradigmalarını kullanan sistemlerle karşılaşabilirsiniz ☺
- **Ödev: Cobol, Fortran, Algol 68, Pascal, C ve Nesne Yönelimli Programlama (Java, C++, c#) dillerinin hangi ihtiyaçtan doğduğu ve bir önceki programlama dilinin hangi problemini çözdüğü üzerine bir araştırma yapınız.**

Tasarım Prensipleri

Soyutlama (Abstraction)

- Soyutlama, bir kavramı, önemsiz ayrıntıları göz ardı eden ve bir kavram kapsamında kavram için gerekli olan temel unsurları vurgulayan basitleştirilmiş bir tanımlamaya bölmektedir.
- Bir program sınıflar, fonksiyonlar ve metotlar gibi öğeler içermektedir.
- Nesne yönelimli modellemede, soyutlama doğrudan bir sınıf kavramına aittir.
- Bağlam ya da belirli bir bakış açısı bir soyutlama oluştururken çok önemlidir. Bunun nedeni, bağlamın bir kavramın temel özelliklerini değiştirebileceğidir.
- bir oyun uygulamasında, bir kişinin temel özellikleri bir oyuncu bağlamında olacaktır. Öte yandan koşu egzersizinde bir kişinin temel özellikleri bir sporcu bağlamında olacaktır.
- Yazılım geliştirme bağlamına en uygun soyutlamayı seçmek tasarımcıya bağlıdır ve bir soyutlama oluşturmadan önce bağlamın anlaşılması gerekir.

Soyutlama (Abstraction)

- Bir soyutlamanın temel özellikleri (characteristics) iki yolla anlaşılabilir:
  1. *Temel özellikler (attributes)*
  2. *Temel davranışlar (behaviours) veya sorumluluklar (responsibilities).*
- Temel özellikler, zaman içinde kaybolmayan özelliklerdir. Değerleri değişebilmesine rağmen, özniteliklerin kendileri değişmez.

Örneğin, bir aslan kavramının bir yaş niteliği olabilir. Bu değer değişebilir, ancak aslanın her zaman bir yaş özelliği vardır.

- Temel özelliklere ek olarak, bir soyutlama da bir kavramın temel davranışlarını açıklar.

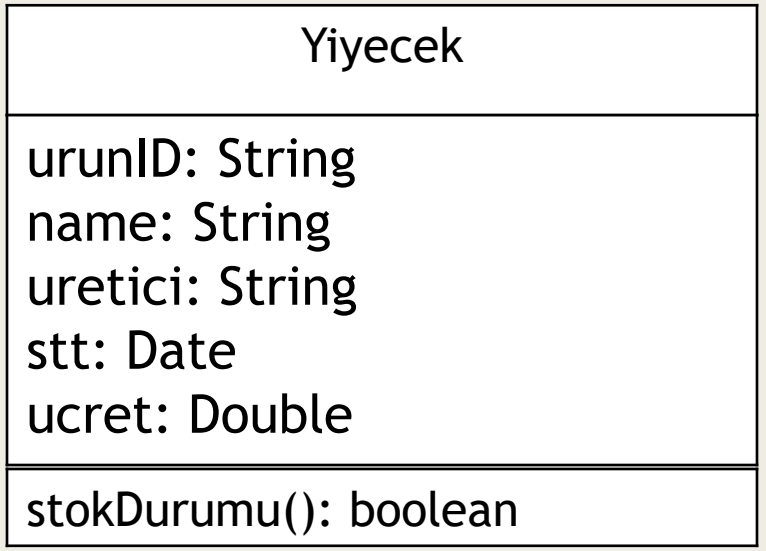
Bir aslanın avlanma, yemek yeme ve uyku gibi davranışları olabilir. Bunlar aynı zamanda aslan soyutlamasının yaşam amacı için yaptığı sorumluluklardır.

Örnek Soyutlama Tasarımı

- Soyutlama, UML sınıf diyagramları kullanılarak tasarım düzeyinde uygulanabilir. Tasarım, sonunda koda dönüştürülür.

Yiyecek	
Ürün ID Ürün ismi Üretici ismi Son tüketim tarihi Ürün Fiyatı Stok Durumu	

(CRC kartı)



(Sınıf Diagramı-UML)

Örnek Soyutlama Tasarımı

- Sınıf diagramında her kavram ya da sınıf bir kutu içinde gösterilir. Kutu içerisinde üç bölüm olacaktır:

Sınıf İsmi
Özellikler (attributes)
Davranışlar (operations)

- Sınıf ismi, Java sınıfınızın ismi olacaktır.
- Özellikler kullanacağınız değişkenlerdir. Değişken ismi ve tipi için standart bir template vardır:

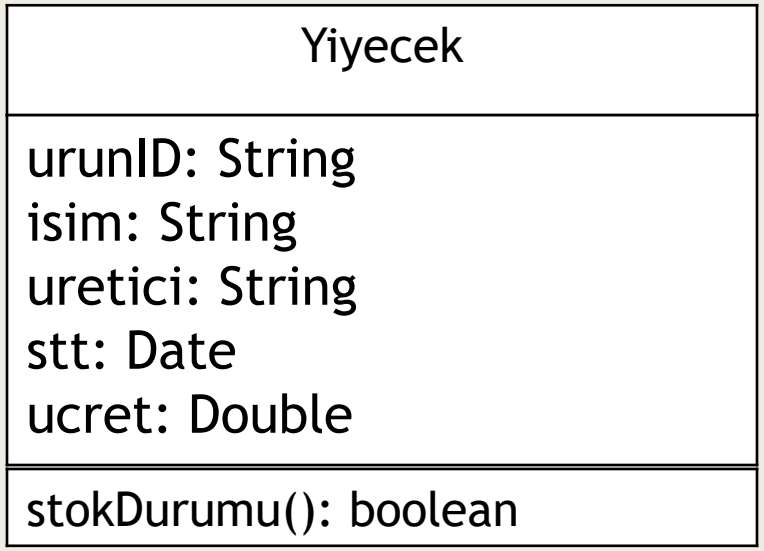
<değişken ismi>:<değişken türü>

- Davranışlar Java metotlarınızdır. Bu bölüm soyutlamanın davranışlarını tanımlar. Standart kullanım template şu şekildedir:

<metot ismi>(parametreler) : < return türü>

Örnek Soyutlama Tasarımı

```
public class Yiyecek
{
public String urunID;
public String isim;
public String uretici;
public Date stt;
public double ucret;
public boolean stokDurumu()
 {
  }}
```



İyileştirme-kapsülleme (Encapsulation)

- İkinci tasarım prensibi iyileştirilmedir.
- Bu prensibin arkasında üç önemli fikir vardır:
  1. *Paketleme (bundle) özelliği: Değerleri (ya da verilerin) ve davranışları (ya da fonksiyonların) kendi kendine yetebilen bir nesneye dönüştürebilen (self-contained object) paketleme yeteneğidir.*
  2. *Açığa çıkarma (expose) özelliği: Bir nesnenin belirli bir verisini ya da fonksiyonunu genellikle bir arayüz (interface) ile diğer nesnelerden erişilebilecek şekilde "açığa çıkarma" yeteneğidir.*
  3. *Kısıtlama (restrict) özelliği: Sadece nesne içindeki belirli fonksiyonlara ve verilere kısıtlı erişme yeteneğidir.*
- Bir nesne için bir sınıf tanımladığında bundling oluşur.
- Soyutlama ilkesi, belirli bir bağlamda bir kavramla ilgili hangi niteliklerin ve davranışların belirleneceğine yardımcı olur.
- Kapsülleme ilkesi bunu bir adım daha ileri götürür ve bu özelliklerin aynı sınıfta bir araya getirilmesini (paketlenmesini-bundled) sağlar.

İyileştirme-kapsülleme (Encapsulation)

- Bir nesnenin verileri yalnızca o nesneyle alakalı şeyleri içermelidir.

Örneğin, bir aslan nesnesi hangi yiyeceği avladığını “bilir” ancak farklı bir kıtada hangi hayvanların yaşadığını bilmez, çünkü bu ilgili olduğu veri değildir.

- Bu nedenle, bir sınıf sadece onunla ilgili hangi özelliklerin veya verilerin olduğunu bilir.
- Bir sınıf ayrıca metotlar aracılığıyla davranışları tanımlar. Metotlar, gerçek davranışlara erişmek için nesnedeki nitelik değerlerini veya verileri değiştirir.
- Bazı “metotlar” diğer sınıftaki nesnelere maruz bırakılabilir veya erişilebilir hale getirilebilir. Bu durum, sınıfı kullanmak için diğer nesnelere bir arayüz (interface) sağlar.

Örnek Kapsülleme Tasarımı

- Kapsülleme tasarım prensibi üç fikir içerir:
  1. *Verileri işleyen (değiştiren) veriler ve fonksiyonlar bir self-contained içerisine paketlenir (bundle).*
  2. *Nesnenin verileri ve fonksiyonları diğer nesneler tarafından erişilebilir hale getirilebilir (expose).*
  3. *Nesnenin verileri ve fonksiyonları yalnızca nesnenin içinde sınırlandırılır (restrict).*
- UML sınıf diagramları bir self-contained nesne içinde verileri ve fonksiyonları paketlemektedir. Bununla birlikte erişim ve kısıtlama, + ve – işaretlerle gösterilebilir.

Ogrenci
-notOrtalamasi: float -lisansProgrami: String
+getNotOrtalamasi(): float +getLisansProgrami(): String +setNotOrtalamasi(float) +setLisansProgrami(String)

Örnek Kapsülleme Tasarımı

```
public class Ogrenci{
private float notOrtalamasi;
private String lisansProgrami;
}
class test{
   public static void main (String[] args) {
      Ogrenci ahmet=new Ogrenci();
      ahmet.notOrtalamasi=3.2;
      ahmet.lisansProgrami="Yazilim Muhendisligi";
      System.out.println("Ahmet'in not ortalamasi:"+ahmet.notOrtalamasi);
   }
}
```

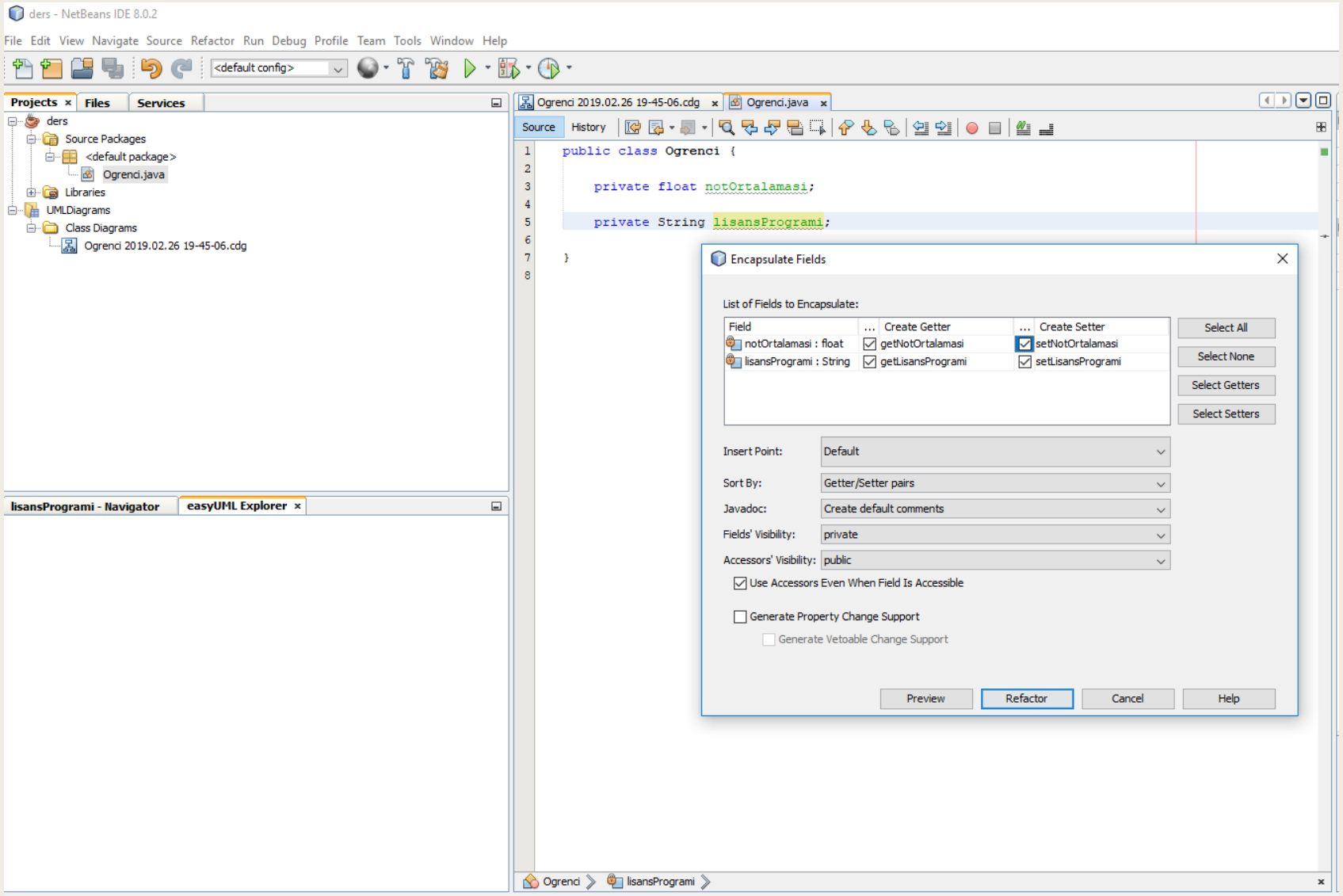
Örnek Kapsülleme Tasarımı

```
public class Ogrenci{
private float notOrtalamasi;
private String lisansProgrami;
public void setNotOrtalamasi(float notOrtalamasi){
this.notOrtalamasi= notOrtalamasi;
}
public float getNotOrtalamasi(){
return notOrtalamasi;
}
public void setLisansProgrami(String lisansProgrami){
this.lisansProgrami= lisansProgrami;
}
public String getLisansProgrami(){
return lisansProgrami;
}
}
```

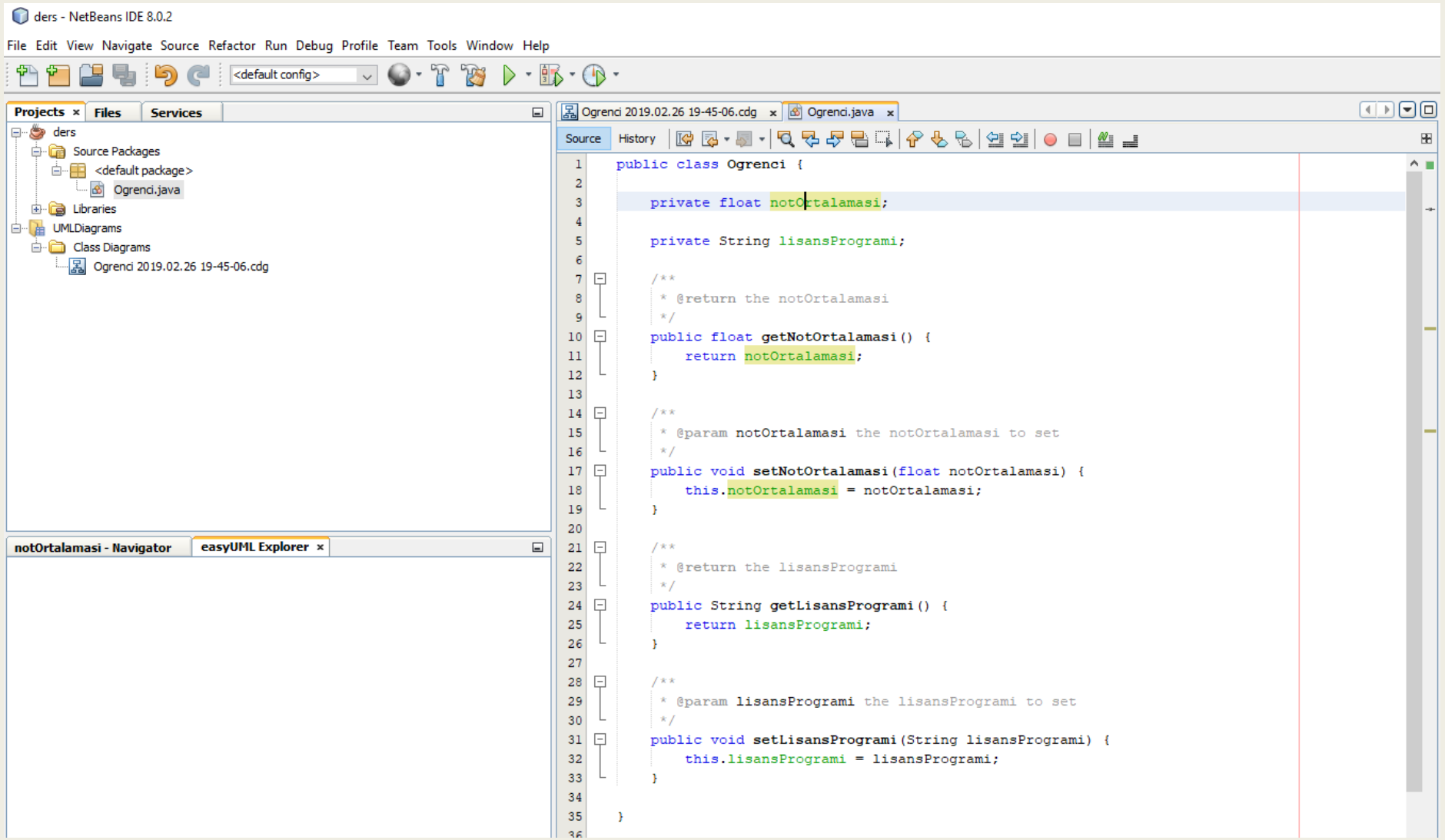
Örnek Kapsülleme Tasarımı

```
class test{
   public static void main (String[] args) {
      Ogrenci ahmet=new Ogrenci();
      ahmet.notOrtalamasi=3.2;
      ahmet.setNotOrtalamasi(3.2);
      System.out.println("Ahmet'in not ortalamasi:"+ahmet.notOrtalamasi);
      System.out.println("Ahmet'in not ortalamasi:"+ahmet.getNotOrtalamasi);
      ahmet.lisansProgrami="Yazilim Muhendisligi";
      ahmet.setLisansProgrami("Yazilim Muhendisligi");
      System.out.println("Ahmet'in bolumu:"+ahmet.lisansProgrami);
      System.out.println("Ahmet'in bolumu:"+ahmet.getLisansProgrami);
   }
}
```

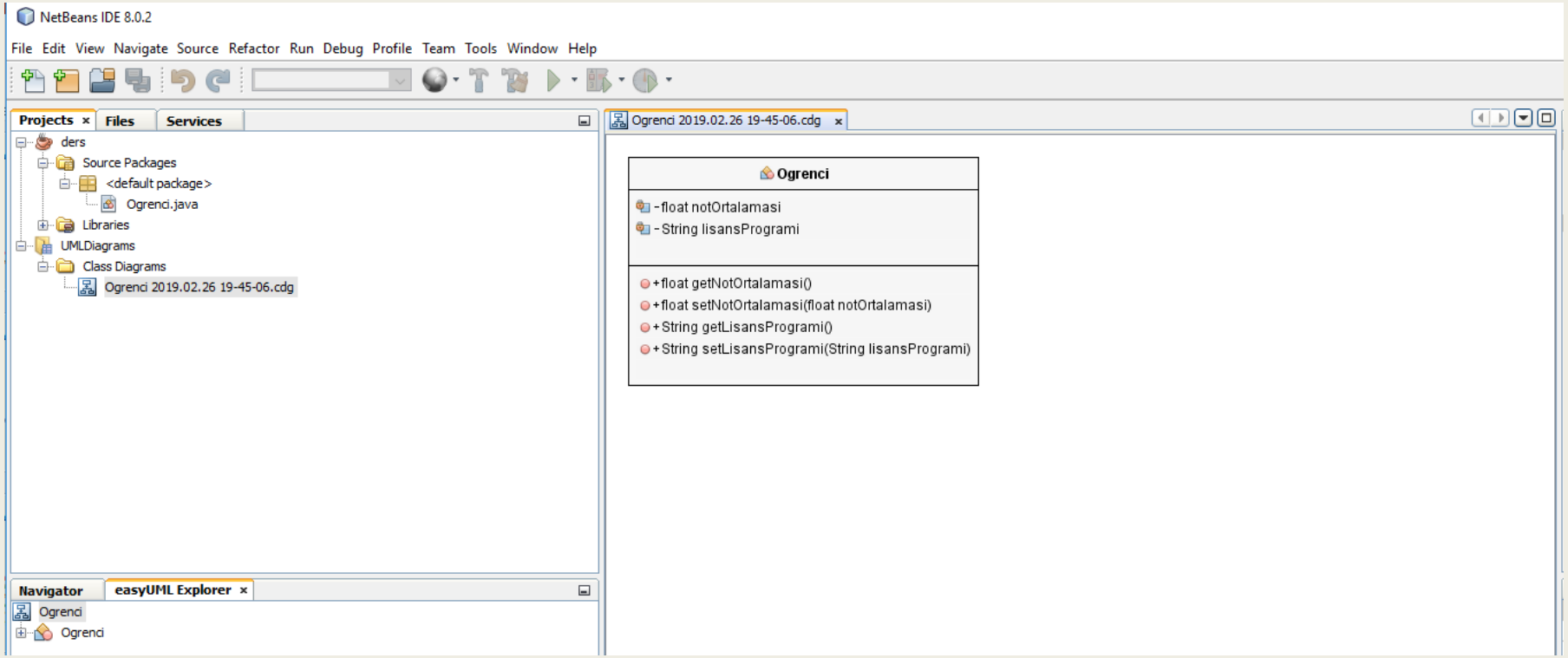
Örnek Kapsülleme Tasarımı



Örnek Kapsülleme Tasarımı



Örnek Kapsülleme Tasarımı-UML Diagram



Ayrıştırma (Decomposition)

- Ayrıştırma tasarım ilkesi bir bütün alır ve farklı bölümlere ayırır.
- Aynı zamanda tersini yapar ve farklı fonksiyonlara sahip ayrı parçalar alır ve bir bütün oluşturmak için bunları birleştirir.
- Ayrışmada, bütün ile parçalar arasındaki etkileşimi tanımlayan üç tür ilişki vardır:
  1. *Birliktelik (Association)*
  2. *Münasebet (Aggregation)*
  3. *Oluşum (Composition)*
- Oluşum (composition) elde etmek için verilerimiz veya nesnelerimiz arasında kurabileceğimiz ilişki türleri birliktelik (association) veya münasebet (aggregation) olabilir.
- Dersin bitmesi ve ders nesnesinin (object) yok olması durumunda bu dersi alan öğrencilerin varlığı devam eder.
- Ev nesnesinde odalar nesnesi varken, evin yok olması durumda oda nesneleri de yok olur.

Association

- Birliktelik, iki nesne arasında, bir süre birbirleriyle etkileşime girebilecek gevşek bir ilişki olduğunu gösterir.
- Birbirlerine bağımlı değillerdir - eğer bir nesne imha edilirse, diğeri var olmaya devam edebilir.
- Birliktelik ilişkisine örnek bir müşteri ve otel olabilir.
- Bir kişi bir otelle etkileşime girebilir, ancak kendi işletmesi olmayabilir.
- Bir otel birçok insanla etkileşime girebilir.

Association

```
public class Student {
      public void play( Sport sport ){
            execute.play( sport );
      }
      …
}
```

- Öğrenci oynaması için bir spor nesnesi seçer ancak öğrenci spor nesnesine sahip değildir. Sadece oynamak için onunla etkileşime girer.
- Herhangi bir sayıda spor öğrenci tarafından oynanabilir ve herhangi bir sayıda öğrenci spor yapabilir.

Aggregation

- Münasebet, parçalar tüme ait olsa da, bağımsız olarak da var olabilirler.
- Örneğin yolcu uçağı ve kabin ekibi…
- Uçak kabin ekibi olmadan hizmet veremez, ancak kabin ekibi ayrılırsa uçak nesnesi var olmaya devam eder. Aynı şekilde kabin ekibi de uçağa binmediyse varlığını sürdürür.

Aggregation

```
public class Airliner {
      private ArrayList<CrewMember> crew;

      public Airliner() {
            crew = new ArrayList<CrewMember>();
      }

      public void add( CrewMember crewMember ) {
            …
      }
}
```

Uçak sınıfında, kabin ekibi üyelerinin bir listesi vardır. Kabin ekibi üyelerinin listesi boş bırakılmıştır ve public bir metot yeni kabin ekibi üyelerinin eklenmesine izin vermektedir. Bir uçağın bir ekibi var. Bu, bir uçağın sıfır veya daha fazla ekip üyesine sahip olabileceği anlamına gelir.

Composition

- Bir bütün, parçaları olmadan var olamaz ve bütün, eğer yok edilirse, o zaman, parçalar da yok edilir.
- Örneğin ev ve oda ilişkisi…
- Bir ev birden fazla odadan oluşur ancak ev yok olursa odalar da yok olur.

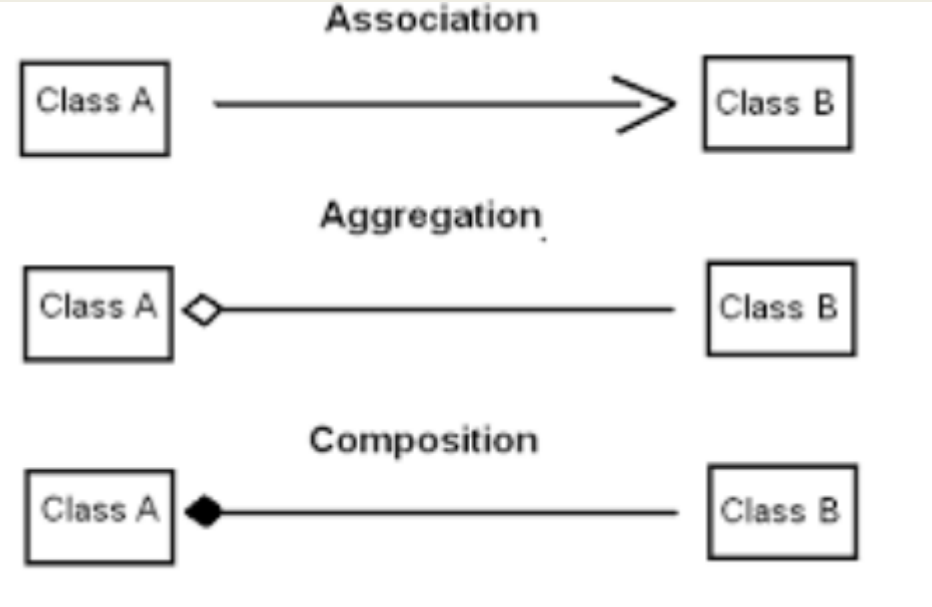
- Ev nesnesinin yanındaki doldurulmuş elmas, evin ilişkideki “bütün (whole)” olduğu anlamına gelir.

Composition

```
public class House {
	private Room room;

	public House(){
		room = new Room();
	}
}
```

- Oda nesnesinin başka bir yerde tanımlanması gerekmez, iki parça biri olmadan diğeri olmayacak şekilde sıkıca bağlanır

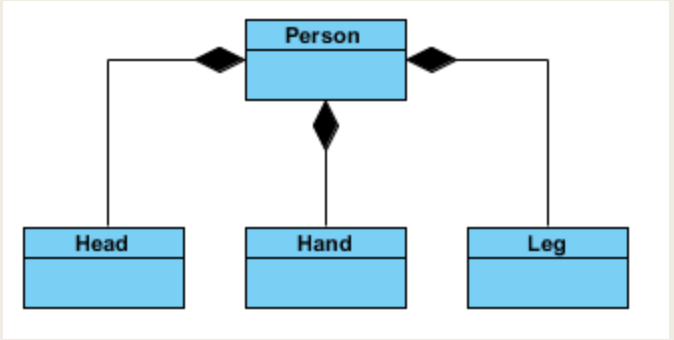
Ayrıştırma (Decomposition)



- Association: İki nesne arasında her zaman bir bağlantı vardır.
- Aggregation: İki nesne arasında bir parça-bütün (whole-part) ilişki vardır. Ancak biri diğerisiz yaşayabilir.
- Composition: İki nesne arasında bir parça-bütün ilişkisi vardır. Ancak biri diğerisiz yaşayamaz.

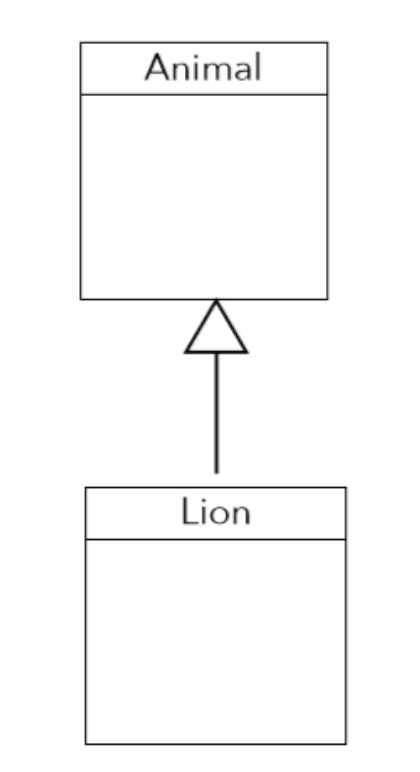
Ayrıştırma (Decomposition)

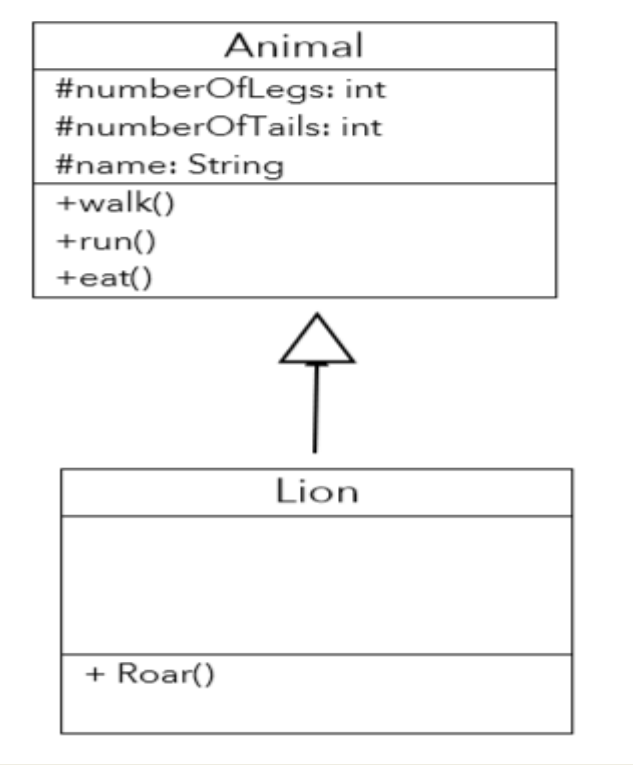
- Association: İki sınıf arasındaki ilişkiyi gösterir. Bu ilişki tek yönlü ya da çift yönlü olabilir:
  - *Müşteri sipariş verir.*
  - *A, B ile evlidir*
  - *B, A ile evlidir.*
- Aggregation: Bir "bütün (whole)" sınıf, daha küçük "parça (part)" sınıflarla ilişkilidir. Ya da bir küçük "parça" sınıf, daha büyük bir "bütün" sınıfın parçasıdır:
  - *Derneğin üyeleri vardır.*
- Composition: "bütün", "parçaların" oluşmasından veya yok olmasından sorumludur.
  - *Bir üniversitenin bölümleri vardır.*
  - *Bir kişinin başı, eli ve bacakları vardır.*



Genelleme (Generalization)

- Genelleştirme tasarım ilkesi, iki veya daha fazla sınıf arasında tekrarlanan, ortak veya paylaşılan özellikleri alır ve bunları başka bir sınıfa dönüştürür.
- Böylece kod tekrar kullanılabilir ve özellikler alt sınıflar tarafından miras alınabilir.





Genelleme (Generalization)

- Javada erişim seçenekleri

  Public: Her yerden erişilebilir.

  Private: Sadece sınıf içinden erişilebilir.

  Protected: Sınıf içinde ve türetilen sınıflar içinden erişilebilir.

- Protected özelliği kapsullenmiş (encapsulated) sınıflardan, türetilen (subclass) sınıflardan ve aynı paket (package) içindeki sınıflardan erişilebilir.

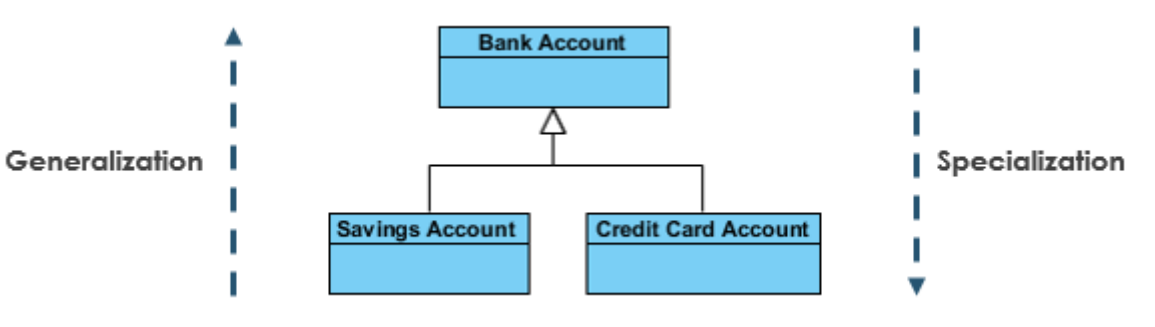
```
public abstract class Animal {
      protected int numberOfLegs;
      protected int numberOfTails;
      protected String name;

      public Animal( String petName, int legs, int
tails ) {
            this.name = petName;
            this.numberOfLegs = legs;
            this.numberOfTails = tails;
      }

      public void walk() { … }
      public void run() { … }
      public void eat() { … }
}
```

```
public class Lion extends Animal {
      public Lion( String name, int legs, int tails
) {
            super( name, legs, tails );
      }

      public void roar() { … }
}
```

Kaynak: Software Design and Architecture Specialization, by University of Alberta

Genelleme (Generalization) ve Özelleşme(Specialization)

- **Genelleme**, benzer nesne sınıflarını tek, daha genel bir sınıfta birleştirme mekanizmasıdır. Genelleme, bir dizi varlık arasındaki ortaklıkları tanımlar. Ortaklık, niteliklerden, davranışlardan veya her ikisinden de olabilir.
- Özelleşme, genelleme işleminin tersi bir süreçtir, mevcut bir sınıftan yeni alt sınıflar oluşturmak demektir.



Kaynak: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

Kalıtım (Inheritance) Çeşitleri

- Javada yalnızca tek bir uygulama kalıtımına (implementation inheritance) izin verilir.
- Örneğin, Animal sınıfı (bir önceki örnekteki) çoklu alt sınıflara süper sınıf olabilir: Lion sınıfı, Wolf sınıfı veya Deer sınıfı. Bu sınıfların her biri özel davranışlara veya özelliklere sahip olabilir, bu yüzden bir Lion nesnesi nasıl kükreyeceğini bilir, ancak bir Wolf nesnesini nasıl kullanacağını bilemeyebilir.
- Alt sınıflar ayrıca başka bir sınıfa üst sınıf olabilir. Kalıtım, istediğiniz kadar sınıf arasında dolaşabilir.
- Kalıtım, ilgili sınıfların tek bir üst sınıfa genelleştirilmesine izin verir ve yine de alt sınıfların aynı nitelik ve davranış kümesini korumasını sağlar.
- Bu, koddaki fazlalığı kaldırır ve değişikliklerin uygulanmasını kolaylaştırır.

Kaynak: Software Design and Architecture Specialization, by University of Alberta

Arayüz Kalıtımı (Interface Inheritance)

- C ++ gibi diğer diller çoklu kalıtımı destekler.
- Java ise genellemenin bir diğer şekli olan arayüz kalıtımını sunarak tek uygulama kalıtımı sınırlamasını çözer.
- Javada uygulama kalıtımını ifade etmek için “extends” kelimesini kullanırız.
- Bir alt sınıf, bir üst sınıfı “extends” eder. Yani alt sınıf hem üst sınıf gibi hem de kendi sınıfı gibi davranır.
- Alt sınıf, üst sınıfın “uygulama ayrıntılarını” devralır.
- Bir arayüz yalnızca metodu belirtir (yapılandırıcısı, metot gövdesi ve nitelikleri olmadan).

```
public interface IAnimal {
      public void move();
      public void speak();
      public void eat();
}
```

Kaynak: Software Design and Architecture Specialization, by University of Alberta

Arayüz Kalıtımı (Interface Inheritance)

- Bir arayüzü kullanmak için “implements” kelimesini kullanırız.

```
public class Lion implements IAnimal {
     /* Attributes of a lion can go here */

     public void move() { … }
     public void speak() { … }
     public void eat() { … }
}
```

```
public class Lion implements IAnimal {
     public void speak() {
          System.out.println( "Roar!" );
     }
}

public class Wolf implements IAnimal {
     public void speak() {
          System.out.println( "Howl!" );
     }
}
```

Kaynak: Software Design and Architecture Specialization, by University of Alberta

Nesne Yönelimli Belgesel

- Hayvanlar alemi belgeseli çekerek şimdiye kadar öğrendiğimiz tüm nesne yönelimli araçlarını kullanmaya çalışalım.
- Belgeselimizde etoburlar ve otoburlar henüz biz insanların zarar vermediği vahşi bir ormanda yaşıyorlar. Tamamen doğal bir ortamda birbirleri ile etkileşim içinde olup su içme, avlanma, seslenme vb. aktivitelerini yerine getiriyorlar.
- Belgeselimizi çekmeye hayvanları ikiye ayırmakla başlıyoruz: **Etobur** ve **Otobur**.
- Etobur hayvanlar, avlayarak yiyeceğini buluyorlar. Otoburların ise işleri çok kolay otlama ve su içme.
- Fakat otoburlar, etoburlara dikkat etmesi gerekiyor. Çünkü etoburlar en çok otoburlar avlıyorlar. Tabi güçleri yetiyorsa etoburları da avladıkları oluyor.
- Etobur ve otobur diye iki nesne normal nesne oluşturmak doğru mu? Yoksa etobur ve otobur olmak üzere iki arayüz (interface) mi kullanmalıyız?

Nesne Yönelimli Belgesel

- Çekeceğimiz belgeselin senaryo başlıkları:

1. Etobur ve Otobur diye iki arayüzümüz olacak.
2. Kedigil, Ayigil ve Kusgiller diye soyut sınıflarımız olacak ve diğer somut sınıflar bu sınıflardan türeyecekler.
3. Caddy, Kaplan nesneleri Kedigillerden türeyecek. Ve dolayısıyla Etobur olacaklar ve avlanma yetenekleri olacak.
4. BozAyi nesnesi Ayigil soyut sınıfından türeyecek. Ayigil soyut nesnesi de iki ayrı arayüzü uyguladığı için (interface, implements) iki farklı yeteneğe sahip olacak. Yani hem otçul hem etçil olacak.
5. Serce nesnemiz de Kusgiller soyut sınıfından tureyecek ve Kusgiller sınıfı da Otoburu uyguladığı için (implements) Otobur olacak ve Etobur hayvanlar tarafından avlanabilecek.

Ödev: Bu senaryoya göre UML diagramını çiziniz. Class'lar içinde avlan(), otla(), icSu(), seslenme() gibi metotları tasarımınıza uygun yerleştiriniz.

Not ☺ 21 Ekim Pazaresi günü çizdiğiniz UML tasarımı uygulama dersinde kontrol edilecektir.

Tasarım Karmaşıklığını Değerlendirme

- Programlama yaparken modülleri basit tutmak önemlidir.
- Bir sistem çeşitli modüllerin bir kombinasyonudur.
- Sistem kötü bir tasarıma sahipse, modüller yalnızca diğer belirli modüllere bağlanabilir ve başka hiçbir şey yapamaz.
- İyi bir tasarım, herhangi bir modülün çok fazla sorun yaşamadan birbirine bağlanmasını sağlar.
- Başka bir deyişle, iyi bir tasarımda, modüller birbiriyle uyumludur ve bu nedenle kolayca bağlanabilir ve yeniden kullanılabilir.
- Tasarım karmaşıklığını değerlendirmek için iki ölçüt kullanılır: **Coupling (bağlama)** ve **cohesion (yapışma).**

Kaynak: Software Design and Architecture Specialization, by University of Alberta

Coupling (Bağlama)

- Bağlama, bir modül ile diğer modüller arasındaki karmaşıklığa odaklanır. Bağlama iki uç nokta arasında dengelenebilir: sıkı bağlama ve gevşek bağlama.
- Bir modül diğer modüllere çok fazla güveniyorsa, diğerlerine "sıkıca bağlanır". **Bu kötü bir tasarım**.
- Bununla birlikte, bir modül iyi tanımlanmış arayüzler üzerinden diğer modüllere kolayca bağlanıyorsa, diğerlerine “gevşek bir şekilde bağlanır”. **Bu iyi bir tasarım.**
- Bir modülün bağlaşımını değerlendirmek için göz önünde bulundurulacak ölçütler şunlardır: Derece (degree), kolaylık (easy) ve esneklik (flexible).
  - ***Derece**, modül ve diğerleri arasındaki bağlantı sayısıdır. Örneğin, bir modül diğer modüllere sadece birkaç parametre veya sınırlı arayüzler ile bağlı olursa derecesi küçük olacaktır ve gevşek bir şekilde bağlanmış olacaktır.*
  - ***Kolaylık**, modül ve diğerleri arasındaki bağlantıların ne kadar açık olduğu ile ilgilidir. Bağlantıların yapılması için diğer modüllerin uygulamalarını anlamaya gerek kalmadan bağlantıların yapılması kolay olmalıdır.*
  - ***Esneklik**, diğer modüllerin bu modül için ne kadar değiştirilebilir olduğunu gösterir. Gelecekteki bağlama için diğer modüller kolayca değiştirilebilir olmalıdır.*

Coupling (Bağlama)

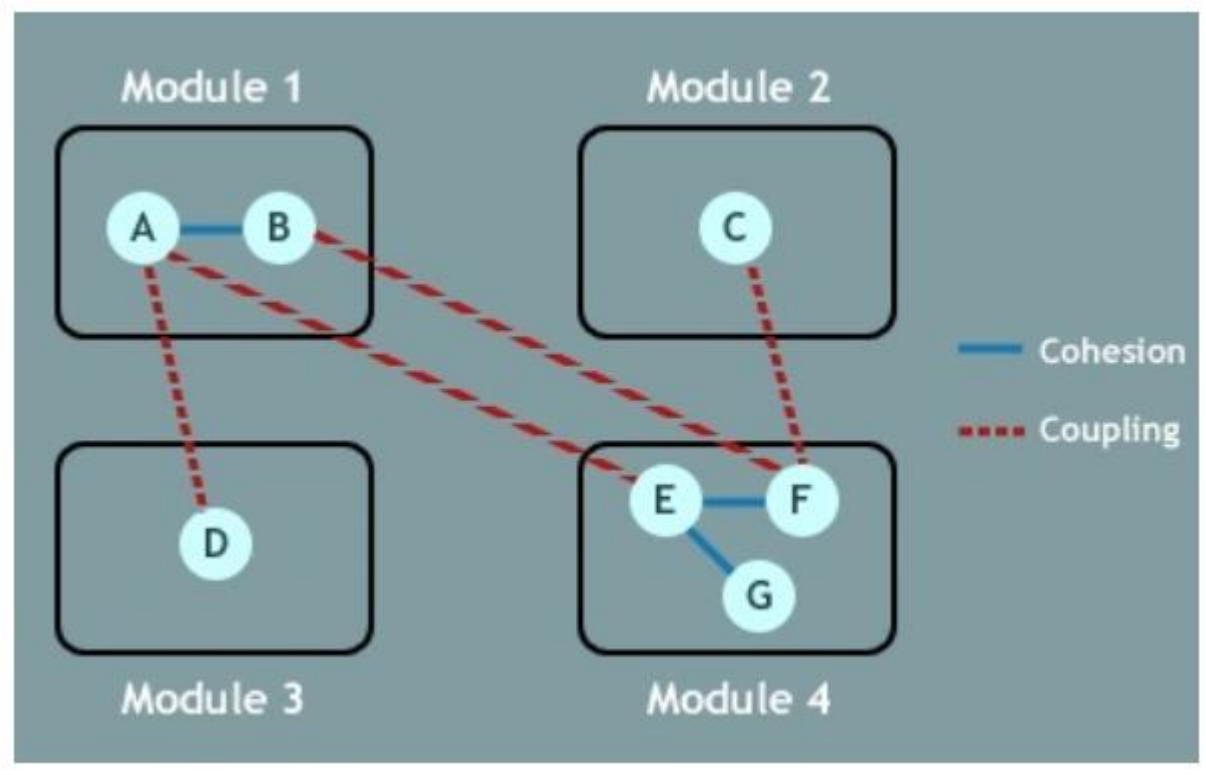
1. *Bir modül diğer modüllere çok sayıda parametre veya arayüz aracılığıyla bağlanırsa,*
2. *Bir modüle karşılık gelen modülleri bulmak zorsa,*
3. *Bir modül sadece belirli diğer modüllere bağlanabilir ve değiştirilemez ise*

Sistemin sıkıca bağlandığını ve kötü bir tasarıma sahip olduğunu düşünebiliriz.

Cohesion (Yapışma)

- Yapışma, bir modül içindeki karmaşıklığa odaklanır ve bir modülün sorumluluklarının netliğini gösterir. Bağlama gibi, yapışma da iki uç nokta arasında çalışabilir: yüksek yapışma (high cohesion) ve düşük yapışma (low cohesion).
- Bir görevi yerine getiren ve başka hiçbir şey yapmayan veya açık bir amacı olan bir modülün high cohesion'ı vardır.
- Öte yandan, eğer bir modül birden fazla amacı kapsıyorsa, bir yöntemi anlamak için bir kapsüllemenin kırılması gerekiyorsa veya modülün net olmayan bir amacı varsa, low cohesion'ı vardır.
- İyi bir tasarım high cohesion olmalıdır.
- Bir modülün birden fazla sorumluluğu varsa, modülü bölmek iyi bir fikirdir.

Cohesion and Coupling



Kaynak: https://freefeast.info/difference-between/difference-between-cohesion-and-coupling-cohesion-vs-coupling/

Bilginin Gizlenmesi

- İyi tasarlanmış bir sistem iyi düzenlenmiş ve çeşitli tasarım ilkelerinin yardımı ile elde edilmiştir.
- Bir yazılım sisteminin her bir bileşeninin başka her şeyi bilmesi gerekmez.
- Modüller sadece işlerini yapmak için ihtiyaç duydukları bilgiye erişebilmelidir.
- Bilgilerin modüllerle sınırlandırılması, böylece doğru şekilde kullanılması ve her şeyin “gizlenmesi” için sadece asgari miktarda bilgiye ihtiyaç duyulması yeterlidir.
- Bilgi gizleme genellikle hassas verilerle ilişkilendirilir; veriler ne kadar hassas olursa, erişimi o kadar sınırlı olur.
- Yazılım tasarımında, bilgi gizleme ayrıca algoritmalar veya veri gösterimleri gibi değişken detaylarını gizlemek için de kullanılır.
- Öte yandan, varsayımlar (assumptions) gizli değildir ve tipik olarak API'ler ve arayüzlerde ifade edilir.

Bilginin Gizlenmesi

- Bilgi gizleme, geliştiricilerin, üzerinde çalıştıkları modülün uygulama ayrıntılarını bilmelerine gerek kalmadan, modüller üzerinde ayrı ayrı çalışmalarını sağlar.
- **A good rule of thumb:** Uygulama detayları gibi değişebilecek şeyler gizlenmeli ve değişmeyen şeyler, varsayımlar gibi, ara yüzlerle ortaya çıkarılmalıdır.
- Java'da 4 seviye ile bilgi gizleme sağlanabilir:
  - Public • Protected • Default • Private

Kaynaklar:

Design Patterns: Elements of Reusable Object-Oriented Software, by Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Introduction to Design Patterns in C++ with Q4, by Alan Ezust, Paul Ezust.

Software Design and Architecture Specialization, by University of Alberta, Coursera Online.

TASARIM KALIPLARI

Tasarım Kalıpları Nedir?

- Programlama dillerinden bağımsız olarak, şimdiye kadar en iyi tasarımları bir arada toplayan ve gruplayan bir platformdur.
- Karşılaşılan sorunlara bütün ve standart tasarımlar sunarak ortak bir tasarım dili oluştururlar.
- Kısacası nesne yötelimli programlamaya yeni başlayan programcılıar için ortak dil oluştururlar.
- Uygulanan tasarımı tek tek tüm detaylarıyla anlatmaktansa, tasarımın adını söylemek yeterli olacaktır.
- Her tasarım bir şablon oluşturuyor. Tasarımı bilen bir programcı hangi şablonun kullanılması gerektiğini bilecektir…

Tasarım Kalıpları Nedir?

- Yazılım dünyasında “tasarım kalıbı” ifadesini ilk kullanan kişi bir yazılımcı değil Christopher Alexander isimli bir mimar oldu.
- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over , without ever doing it the same way twice”
- “Her bir kalıp öncelikle ortamımızda tekrar tekrar oluşan bir problemi tanımlar ve sonra da bu problem için çekirdek bir çözümü tarif eder öyle ki siz bu çözümü iki kere aynı şekilde yapmadan, milyonlarca kez kullanabilirsiniz”.
- Aynı tasarım kalıbını defalarca uygulasanız bile hiçbir iki uygulamanız birbirinin aynısı olmayacaktır.

Tasarım Kalıpları

- ***Finding responsibilities***
- ***Highly Cohesive Objects***
- ***Lowly Coupled Objects***

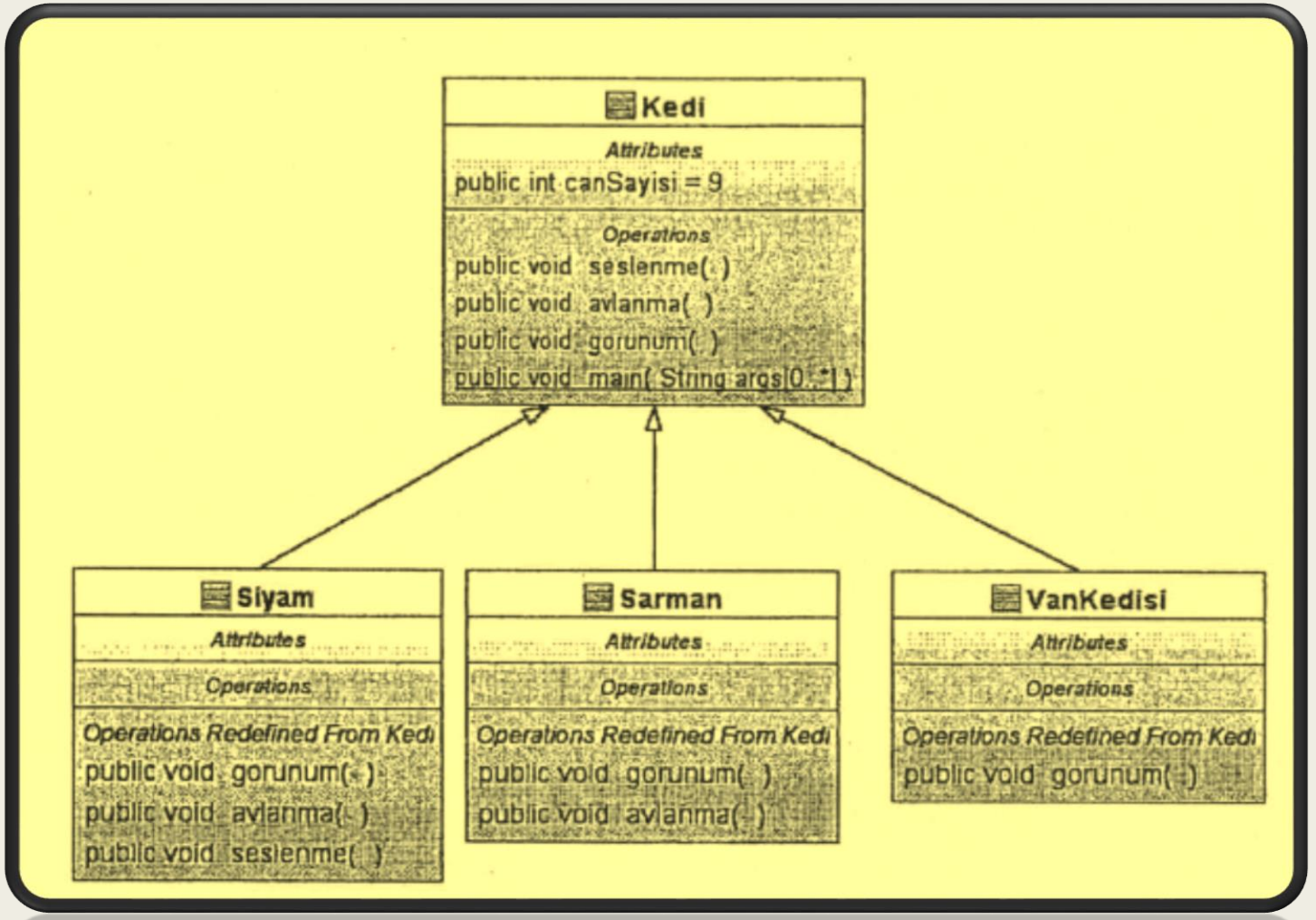
- Yanı tasarım kalıpları yüksek cohesion (yapışma) ve düşük coupling (bağlama) yapıları kurgulamamıza yardımcı olur.
- 1994 yılında Gang of Four (4'lü çete) olarak adlandırılan Eric Gamma, Richard Helm, Ralph Johnson ve John Vlissides ünlü "Tasarım Kalıpları (Design Patterns) kitabını yazmışlardır.
- Kitaplarında 3 farklı kategoride 23 tane tasarım kalıbına yer vermişlerdir:
  1. *Yaratımsal (Creational) - Nesnelerin yaratılması (5 tasarım kalıbı)*
  2. *Yapısal (Structural) – Nesneler arasındaki yapısal ilişkiler (7 tasarım kalıbı)*
  3. *Davranışsal (Behavioral) – Nesnelerin çalışma zamanı davranışlarının değiştirilmesi (11 tasarım kalıbı)*

Strateji Tasarım Kalıbı (Strategy Design Pattern)

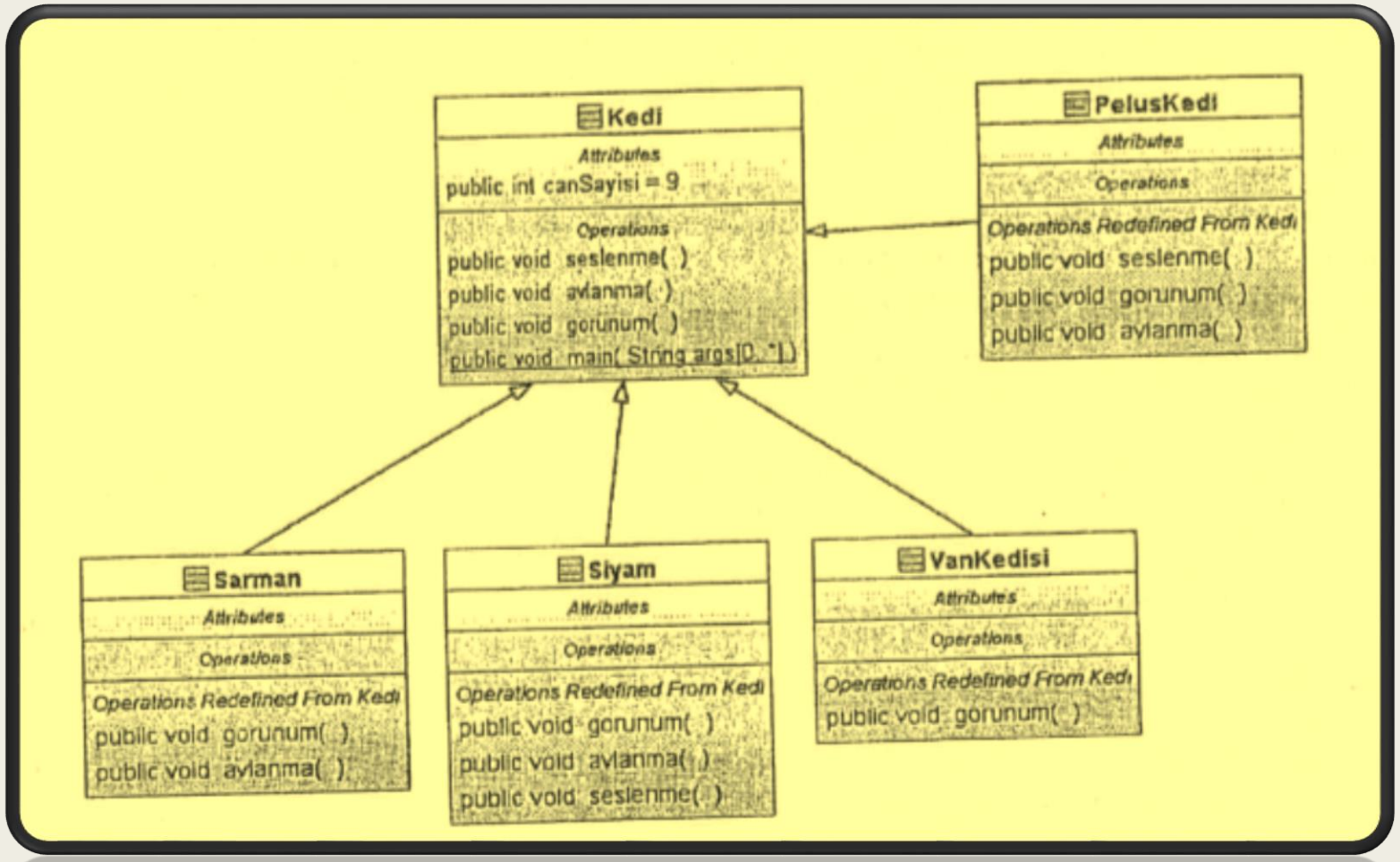
Kedi Similasyonu

```
public class Kedi {
    public int canSayisi = 9;
    public void seslenme(){
        System.out.println("miyav");
    }
    public void avlanma(){
        System.out.println("Fare Avla");
    }
    public void gorunum(){
        System.out.println("Ben kediyim");
    }
    public static void main(String[] args) {
        Kedi sarman = new Kedi();
        sarman.avlanma();
        sarman.seslenme();
        sarman.gorunum();
```

Kedi Similasyonu



Kedi Similasyonu



Kedi Similasyonu

- Tasarımcılar peluş kedinin de sisteme hızlıca dahil edilmesini istedi ancak “peluş kedi”, “kedi” nesnesinden türetildiğinde avlanma ve seslenme metotlarının ezilmesi (override) gerekti.
- Bu bir yama çözüm, her bir nesne için bu şekilde yama çözümler sunmak programın bakım ve güncelleme maliyetini arttıracaktır.
- Tasarım kalıplarındaki genel kural :

  “Değişkenlik gösterecek tüm özellik ve davranışları bir araya getir ve ayrı kalıplarda tut!”
- Seslenebilme ve avlanabilme diye iki arayüz tanımlamak bu tasarımda mantıklı olacaktır.
- **Seslenebilme** arayüzünü 3 ayrı sınıf implements edecek: Miyav, Roarr ve Seslenemez. **Avlanabilme** arayüzünü ise 2 sınıf uygulayacak: Avlanabilir ve Avlanamaz.
- Böylelikle bu tasarıma yeni bir davranış eklemek kolay olacak, tek yapmamız gereken bu davranış sınıfının arayüz uygulamasını (implements) sağlamak.

Arayüze Programlama

- **Uygulama Programlama:** Somut bir sınıf değişkeninden yine somut bir sınıf türetilmesi.

```
Siyam sekerkiz = new Siyam();
sekerkiz.seslen();
```

- **Arayüze Programlama:** Üst sınıf veya super sınıfa programlama olarak da bilinmektedir. Değişken tipimiz super sınıf veya arayüz olmalı ve new komutu ile de ilgili sınıfı üretmeliyiz.

```
Kedi sekerkiz = new Siyam();
sekerkiz.seslen();
```

Kedi Similasyonu

```
public class Kedi {
    public int canSayisi = 9;
    public void seslenme(){
        System.out.println("miyav");
    }
    public void avlanma(){
        System.out.println("Fare Avla");
    }
    public void gorunum(){
        System.out.println("Ben kediyim");
    }
    public static void main(String[] args) {
        Kedi sarman = new Kedi();
        sarman.avlanma();
        sarman.seslenme();
        sarman.gorunum();
```

```
public class Kedi {
    public int canSayisi = 9;
    Seslenebilme seslenmeYetenegi;
    Avlanabilme avlanmaYetenegi;

    public void seslenmeIslemi(){
        seslenmeYetenegi.seslenme();
    }
    public void avlanmaIslemi(){
        avlanmaYetenegi.avlanma();
    }
    public void gorunum(){
        System.out.println("Ben kediyim");
    }
    public static void main(String[] args) {
        Kedi sarman = new Kedi();
        sarman.gorunum();
```

Kedi Similasyonu

- Sarman isimli bir kedi nesnesi tanımladığımızda yapılandırıcısına ilgili davranışları ekliyoruz.

```
public class Sarman extends Kedi {
    public void gorunum(){
        System.out.println("Ben Sarman kedisiyim...");
    }
    public Sarman(){
        seslenmeYetenegi = new Miyav();
        avlanmaYetenegi = new Avlanabilir();
    }
}
```

- Sarman kedisi kedi sınıfından ilgili metot ve özellikleri alıyordu. Şimdi ise sarman kendisinin yapılandırıcısında yeteneklerin yaratılması sağlandı.
- Bu doğru bir tasarım mıdır?

Kedi Similasyonu

- Yeteneklerin yapılandırıcıya konulması temel tasarım hatalarından birisidir.
- Sarman her yaratıldığında bu yeteneklerle yaratılacak ve değiştirilmesi gereken durumlar olacaktır.
- Örneğin, Sarman doğduğunda henüz bir bebek kedi olacak ve avlanamayacak, seslenmesi de “miyavv” değil “mivvvv” gibi olacak☺
- Bu tasarım hatası uygulamaya programlama hatasıdır. Ancak istediğimiz ve yapılması gereken **“arayüze programlamadır”**.

Kedi Similasyonu

- **Avlanabilme** arayüzümüz avlanabilme yeteneklerini bir arada tutuyor ve istenildiğinde kullandırıyor:

```
public interface Avlanabilme {
    public void avlanma();
}
```

- Arayüz iki ayrı davranışı barındırıyor. İki ayrı sınıf bu arayüzü uyguluyor.

Avlanabilen davranış:

```
public class Avlanabilir implements Avlanabilme{
    public void avlanma(){
        System.out.println("Avlanabilirim. ");
    }
}
```

Kedi Similasyonu

Avlanamayan davranış:

```
public class Avlanamaz implements Avlanabilme {
    public void avlanma(){
        System.out.println("Avlanamaz");
    }
}
```

- Diğer arayüzümüz ise **Seslenebilme**. Bu arayüzü uygulayan 3 davranış var şu an için: Miyav, roar ve seslenemez.

```
public interface Seslenebilme {
    public void seslenme();
}
```

Kedi Similasyonu

Miyav davranışı: Miyavlayabilen kediler bu davranışı kullanır.

```
public class Miyav implements Seslenebilme {
    public void seslenme(){
        System.out.println("Miyavv");
    }
}
```

Roar davranışı: Daha büyük kediler daha gür ve korkunç sesler çıkarabiliyor.

```
public class Roarr implements Seslenebilme {
    public void seslenme(){
        System.out.println("Roarrr");
    }
}
```

Kedi Similasyonu

Seslenemez davranışı: Seslenme yeteneği olmayan kediler bu davranışı kullanabilir

```
public class Seslenemez {
    public void seslenme(){
        System.out.println("seslenemez");
    }
}
```

Şimdi Sarman kedisini yeniden yaratalım:

```
public class KediSimulasyon {
    public static void main(String[] args) {
        Kedi nankor = new Sarman();
        nankor.seslenmeIslemi();
        nankor.avlanmaIslemi();
    }

}
```

Kedi Similasyonu

- OOP (nesne yönelimli programlama) temel araçlarından birisi de “setter” lerdir.
- Bu metot davranışı ile sınıf içinde saklı olan metot ve değişkenlere ulaşabilir ve bunları değiştirebiliriz.
- Her iki davranış için de Kedi sınıfı içine yeni setter metotları tasarlıyoruz:

```
public void setSeslenmeYetenegi(Seslenebilme ses){
    seslenmeYetenegi = ses;
}
public void setAvlanmaYetenegi(Avlanabilme avla){
    avlanmaYetenegi = avla;
}
```

Bu iki metot aracılığı ile dışarıdan ve run-time esnasında davranış girilebilir ve içerideki davranış değişkenlere atayabiliriz.

Kedi Similasyonu

- Kedi super sınıfına yeni iki metodu tanımladıktan sonra, bu sefer VanKedisi sınıfımız üzerinden similasyonumuzu çalıştıralım.

```
public class VanKedisi extends Kedi {
    public VanKedisi(){
        seslenmeYetenegi = new Seslenemez();
        avlanmaYetenegi = new Avlanamaz();
    }
    public void  gorunum(){
      System.out.println("ben Van Kedisiyim.");
    }
}
```

Kedi Similasyonu

- Similasyonda yavru VanKedisi yaratıp çalıştıralım:

```
public class KediSimulasyon {
    public static void main(String[] args) {
        Kedi minnos = new VanKedisi();
        minnos.seslenmeIslemi();
        minnos.avlanmaIslemi();
    }
}
```

- Peki kedi büyüdüğünde ne olacak? Tasarıma dönüp davranış eklememiz imkansız. Bu değişikliği rum-time'da yapmamız gerekiyor.
- setAvlanmaYetenegi ve setSeslenmeYetenegi metotları sayesinde bu değişikliği run-time da kolaylıkla yapabiliriz.

Kedi Similasyonu

```
public class KediSimulasyon {
    public static void main(String[] args) {
        Kedi minnos = new VanKedisi();
        minnos.seslenmeIslemi();
        minnos.avlanmaIslemi();
        minnos.setAvlanmaYetenegi(new Avlanabilir());
        minnos.setSeslenmeYetenegi(new Miyav());
        minnos.seslenmeIslemi();
        minnos.avlanmaIslemi();

    }
}
```

Kedi Similasyonu

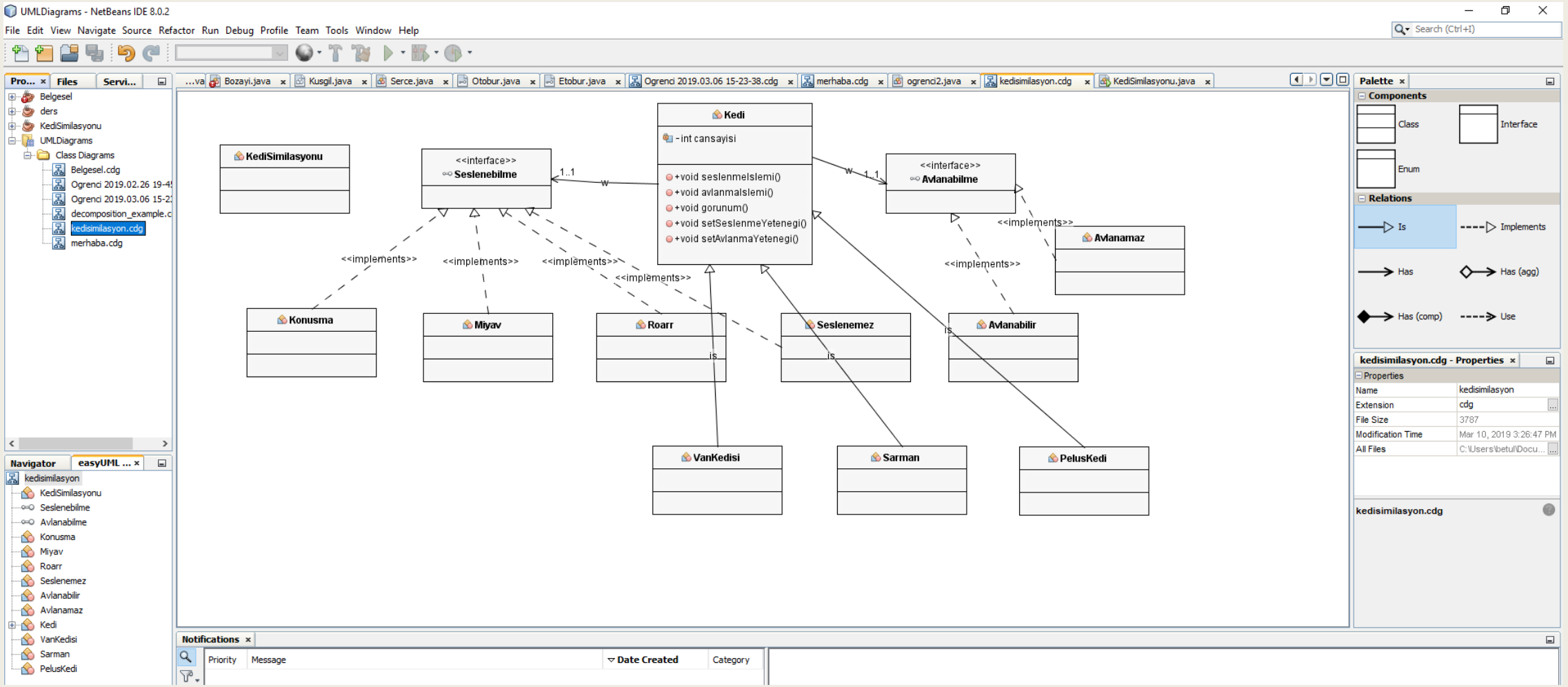
- Tasarımımıza çizmeli kediyi dahil edelim.

```
public class Konusma implements Seslenebilme{
    public void seslenme() {
        System.out.println("Konuşabiliyorum.. Vır vır, Dır
dır..");
    }

}
```

Kedi Similasyonu

```
public class KediSimulasyon {
    public static void main(String[] args) {
        Kedi minnos = new VanKedisi();
        minnos.seslenmeIslemi();
        minnos.avlanmaIslemi();
        minnos.setAvlanmaYetenegi(new Avlanabilir());
        minnos.setSeslenmeYetenegi(new Miyav());
        minnos.seslenmeIslemi();
        minnos.avlanmaIslemi();

        minnos.setSeslenmeYetenegi(new Konusma());
        minnos.seslenmeIslemi();


    }
}
```

Kedi Similasyonu



Strateji Tasarım Kalıpları

- Değişkenlik gösteren algoritmaları ve davranışları bir araya getirip (encapsulation) algoritmaları geliştirebilir hale getirmeli. Böylece arayüzün arkasına toplanan ve ileride geliştirilmeye açık davranışların kontrolsüz değiştirilmesinin önüne geçilir.
- Şimdiye kadar öğrendiğimiz tasarım prensipleri:
  1. *Değişkenlik gösterecek davranışların bir araya getir,*
  2. *Kalıtım yerine kompozisyon kullan,*
  3. *Uygulamaya program yazmak yerine (programming to implementation), arayüze yazılım yap (programming to interface).*

Tasarım Kalıplarının (Design Patterns) Temel Kuralları

1. Uygulamaya (implements) değil de arayüze (interface) program yazılmalı
2. Nesneler arasında esnek bağ oluşturulmalı, nesneler kesinlikle birbirlerine somut bağ oluşturmamalı
3. Değişken ve gelişmeye açık olan bölümler tespit edilmeli ve bir çatı altında tutulmalı
4. Kalıtım (inheritance) yerine kompozisyon (composition) kullanılmalı
5. Tasarım geliştirmeye açık olmalı fakat değiştirilmeye kapalı olmalı
6. Somut sınıflar kullanılmamalı. Somut sınıflar yerine daha çok soyut sınıflar (abstract veya interface) kullanılmalı.

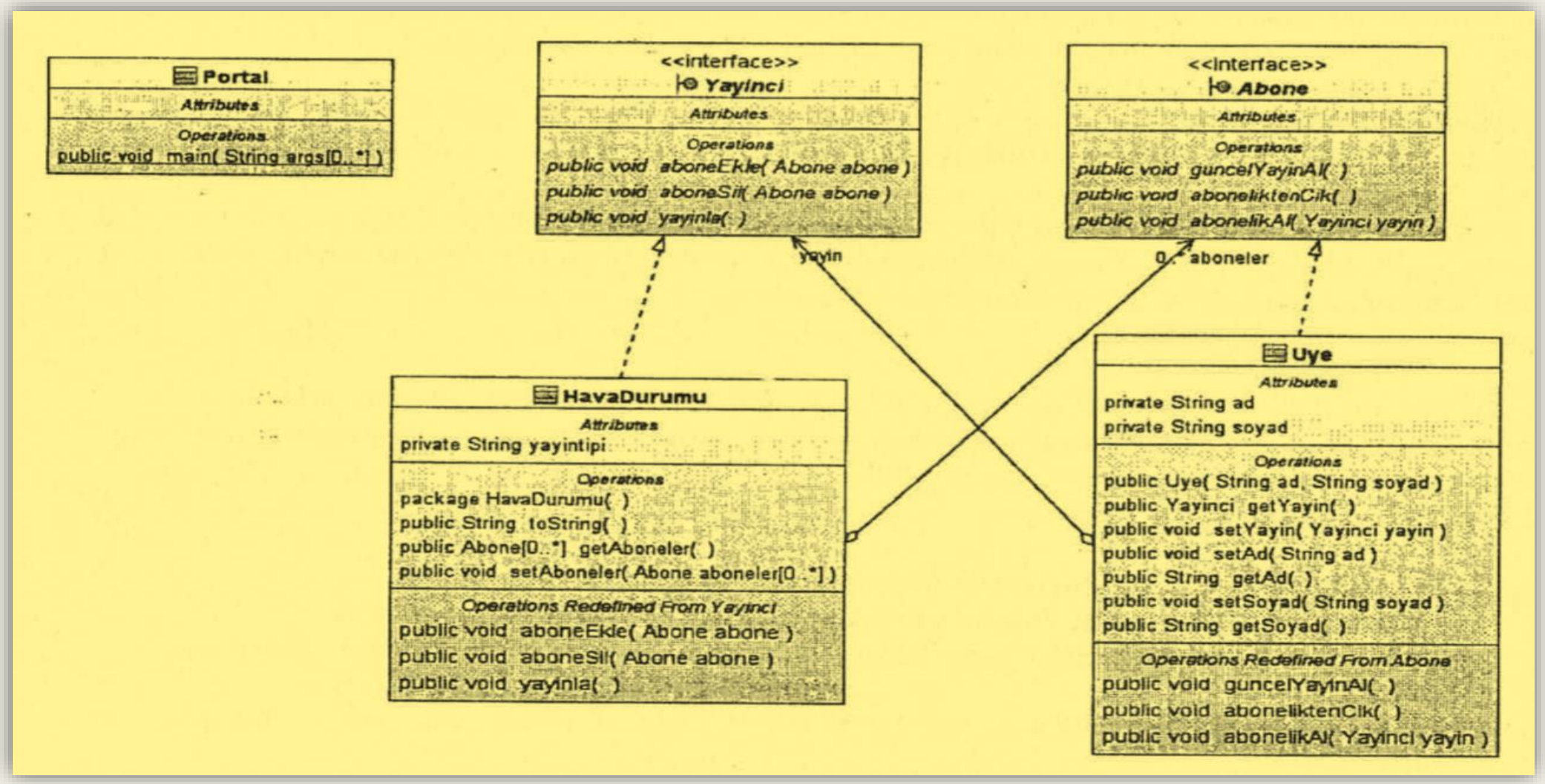
Gözlemci Tasarım Kalıbı (Observer Design Pattern)

- İçerik portal olan bir firma, müşterilerine özel servisler sunmak istiyor.
  - *Finans ile ilgilenecek olan müşterilere finans bilgileri,*
  - *Politika ile ilgilenecek olan müşterilere bilgileri,*
  - *Hava durumu ile ilgilenecek olan müşterilere hava durumu bilgileri sağlanacaktır.*
  - *İsteyen müşteri istediği servise abone olabilecek ve bu servislerden faydalanabilecek. İstediği zaman ise servisten çıkabilecek.*
  - *Servisi alabilecek müşteri sayısında bir sınırlama olmayacak.*

Gözlemci Tasarım Kalıbı (Observer Design Pattern)

- Gözlemci tasarım kalıbında 1-n ilişkisi vardır.
  - *1: yayını yapanlar*
  - *n: bu yayına abone olanlar*
- Portalımızın hava durumu yayını servisini tasarlamak istiyoruz.

Örnek Yazılım Tasarımı



Gözlemci Tasarım Kalıbı (Observer Design Pattern)

- **Yayıncı:** Tüm yayın işlerinin çatısını oluşturuyor. Arayüz olarak yaratılmalı. İçinde abonelik ve çıkarma işlemi ve yayınlanacak bilgilerin metot duyurusu bulunmaktadır. Bu arayüzü uygulayacak her bilgi sınıfı ilgili metotlarda tanımlanmalıdır.
- **Abone:** Tüm abonelik işlemlerini tasarlar. Abone olma ve abonelikten çıkma, güncel yayınları alma gibi özellikler barındırır.
- **HavaDurumu:** Hava durumu nesnesi Yayıncı nesnesini uygulayan bir nesnedir. Abonelere (N tane) otomatik olarak duyurulması gereken bilgiyi yaratır ve yayar.
- **Uye:** Abonelik işlemi bu nesne içinde tanımlanır. Abone arayüzünü uygular.

Gözlemci tasarım kalıbında, aboneler, yayınlanan bilgileri otomatik olarak alırlar. Her abone tek tek bilginin alınması için çaba sarf etmez.

Gözlemci Tasarım Kalıbı (Observer Design Pattern)

- Yayıncı arayüzü:

```
public interface Yayinci {

    public void aboneEkle(Abone abone);
    public void aboneSil(Abone abone);
    public void yayinla();


}
```

- Abone arayüzü:

```
public interface Abone {
    public void guncelYayinAl();
    public void aboneliktenCik();
    public void abonelikAl(Yayinci yayin);

}
```

Gözlemci Tasarım Kalıbı

```
import java.util.ArrayList;

public  class HavaDurumu implements Yayinci{
    private ArrayList<Abone> aboneler = new
ArrayList<Abone>();
    private String yayintipi;
    HavaDurumu(){
        yayintipi = "Hava Durumu";
    }
    public String toString(){
        return yayintipi;
    }
    public void aboneEkle(Abone abone) {
        getAboneler().add(abone);
    }
    public void aboneSil(Abone abone){
        getAboneler().remove(abone);
    }
    public void yayinla(){
        for (int i=0;i<getAboneler().size();i++){
            getAboneler().get(i).guncelYayinAl();
        }
    }

    public ArrayList<Abone> getAboneler(){
        return aboneler;
    }
    public void setAboneler(ArrayList <Abone> aboneler){
        this.aboneler = aboneler;
    }

}
```

Gözlemci Tasarım Kalıbı

```
public class Uye implements Abone {
    private Yayinci yayin;
    private String ad;
    private String soyad;
    public Uye(String ad,String soyad){
        setAd(ad);
        setSoyad(soyad);
    }
    public void guncelYayinAl() {
        System.out.printf("%s %s %s adlı yayını
aldı..\n",getAd(),getSoyad(),yayin.toString());
    }

    public Yayinci getYayin(){
        return yayin;
    }
    public void setYayin(Yayinci yayin){
        this.yayin = yayin;
    }
```

```
    public void aboneliktenCik() {

        getYayin().aboneSil(this);
        System.out.printf("%s %s %s abonelikten
çıktı.\n",getAd(),getSoyad(),yayin.toString());

    }
    public void abonelikAl(Yayinci yayin) {
        setYayin(yayin);
        yayin.aboneEkle(this);
        System.out.printf("%s %s %s abone
oldu.\n",getAd(),getSoyad(),yayin.toString());

    }

    public void setAd(String ad){
        this.ad = ad;
    }
    public String getAd(){
        return ad;
    }
    public void setSoyad(String soyad){
        this.soyad = soyad;
    }
    public String getSoyad(){
        return soyad;
    }

}
```

Gözlemci Tasarım Kalıbı

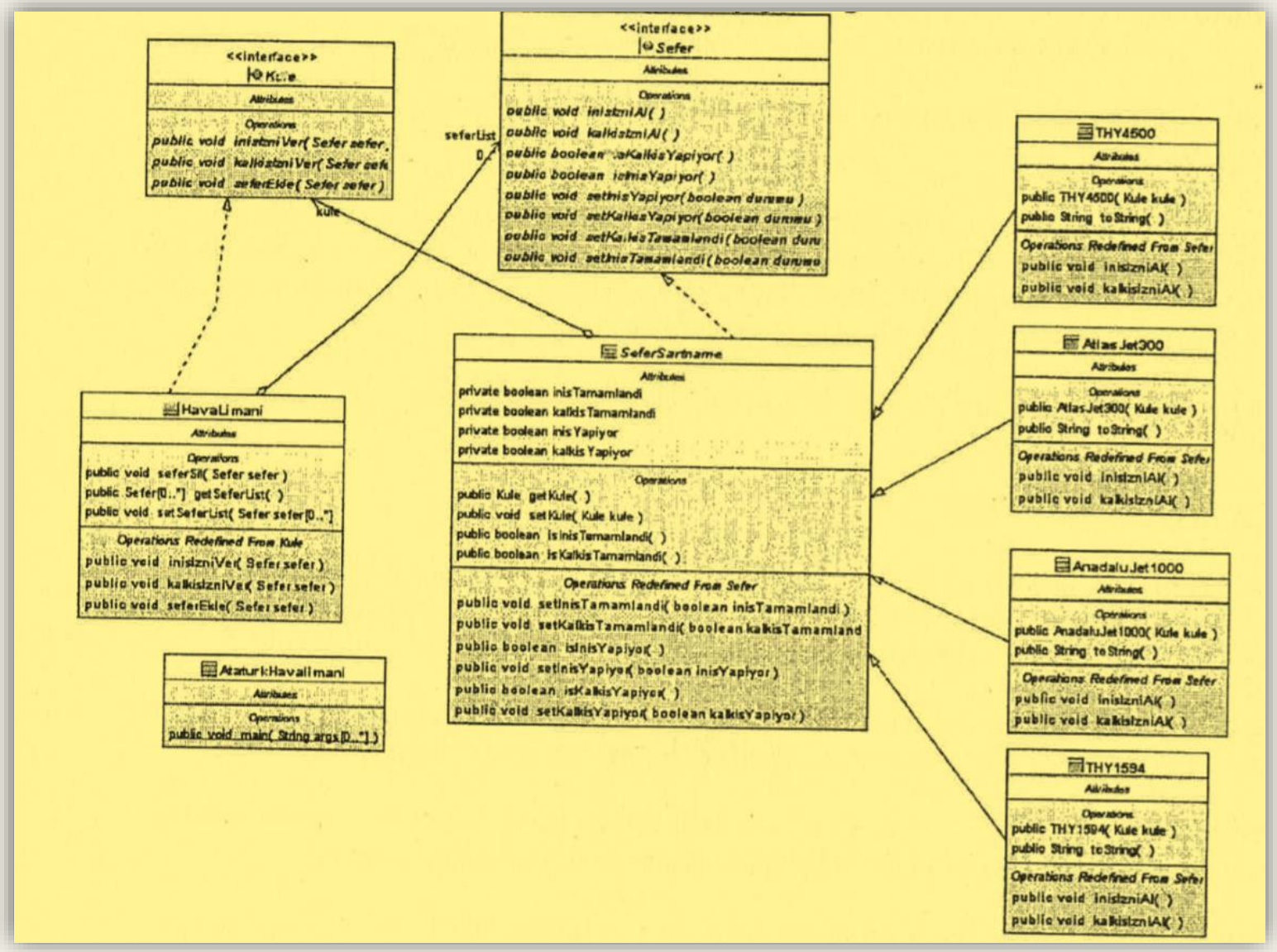
```
public class Portal {
    public static void main(String[] args) {
        Yayinci havadurumu = new HavaDurumu();

        Abone uye01 = new Uye("Sarya","Demir");
        Abone uye02 = new Uye("Arjin","Demir");
        Abone uye03 = new Uye("Mercan","Dede");

        uye01.abonelikAl(havadurumu);

        System.out.println("Havadurumu Yayını 01");
        havadurumu.yayinla();
        uye02.abonelikAl(havadurumu);
        uye03.abonelikAl(havadurumu);

        System.out.println("Havadurumu Yayını 02");
        havadurumu.yayinla();

        uye02.aboneliktenCik();
        System.out.println("Havadurumu Yayını 03");
        havadurumu.yayinla();
    }

}
```

```
Sarya Demir Hava Durumu abone oldu.
Havadurumu Yayını 01
Sarya Demir Hava Durumu adlı yayını aldı..
Arjin Demir Hava Durumu abone oldu.
Mercan Dede Hava Durumu abone oldu.
Havadurumu Yayını 02
Sarya Demir Hava Durumu adlı yayını aldı..
Arjin Demir Hava Durumu adlı yayını aldı..
Mercan Dede Hava Durumu adlı yayını aldı..
Arjin Demir Hava Durumu abonelikten çıktı.
Havadurumu Yayını 03
Sarya Demir Hava Durumu adlı yayını aldı..
Mercan Dede Hava Durumu adlı yayını aldı..
```

Medyatör Tasarım Kalıbı (Mediator Design Pattern)

- Nesne yönelimli programlamanın en önemli konularından biri ise yaratılan nesnelerin birbirleri ile olan iletişimidir.
- Nesne yönelimli programlamada ise, nesnelerin birbirleri ile iletişimininin koordinasyonunu yönetebilen tasarım kalıbı Medyatör Tasarım Kalıbıdır.
- Medyatör özellikle Gözlemci tasarım kalıbı ile birlikte kullanıldığında başarılı sonuçlar elde edilmektedir.
- Gözlemci tasarım kalıbında 1-n ilişkisi vardır. Gözlemci nesneler n tane olarak düşünüldüğünde, bu n nesnenin birbirleri ile iletişim problem yaşayabilme ihtimaline karşılık Medyatör tasarım kalıbı bu sorunu çözebilmektedir.

Medyatör Tasarım Kalıbı



Dekoratör Tasarım Kalıbı (Decorator Design Pattern)

- Kalıtım ile nesneleri birbirinden türeterek, nesnelere yeni özellikler ve davranışlar eklemeyi temel nesne yönelim bilgisi ile yapabiliyoruz.
- Ancak temel kalıtım kurallarını aşağıdaki örneğe uyguladığımızda problemlerle karşı karşıya kalabiliyoruz:

  Bilgisayar satışı yaptığımız bir bilgisayar toptancımız var. Bilgisayar ile ilgili her türlü donanım ve aksesuarı satabiliyoruz. İnternet e-ticaret sitemiz üzerinden bilgisayar ve aksesuarlarını satmak istiyoruz. Yalnız e-ticaret sitemizin en önemli özelliklerinden birisi, siteyen giren kişinin kendine özel bir konfigürasyon kurabilmesi ve bunu satın alabilmesidir.
- Bir bilgisayarı toplamak için sınırsız sayıda seçenek vardır. Ekran kartı, monitor, klavye, ses kartı, ana kart, işlemci…

Dekoratör Tasarım Kalıbı

- E-ticaret sitemizde her seçenek için ayrı bir menu ve uygulama yaptığımızı düşünelim. Sınırsız sayıda menu yaratmamız gerekmektedir:
  - *Bilgisayar*
  - *Monitörlü bilgisayar*
  - *DVD romlu bilgisayar*
  - *Monitörlü, DVD romlu bilgisayar*
  - *Ses kartlı bilgisayar*
  - *Ses kartlı, monitörlü bilgisayar*
  - *Harddiskli bilgisayar*
  - *Harddiskli dvd romlu bilgisayar*
  - …..
- Standart kalıtım yolu ile bilgisayar sınıfından bu seçenekleri türeterek elde etmeye çalışsaydık, bu tasarımın yazılımı bile aylar sürebilirdi.

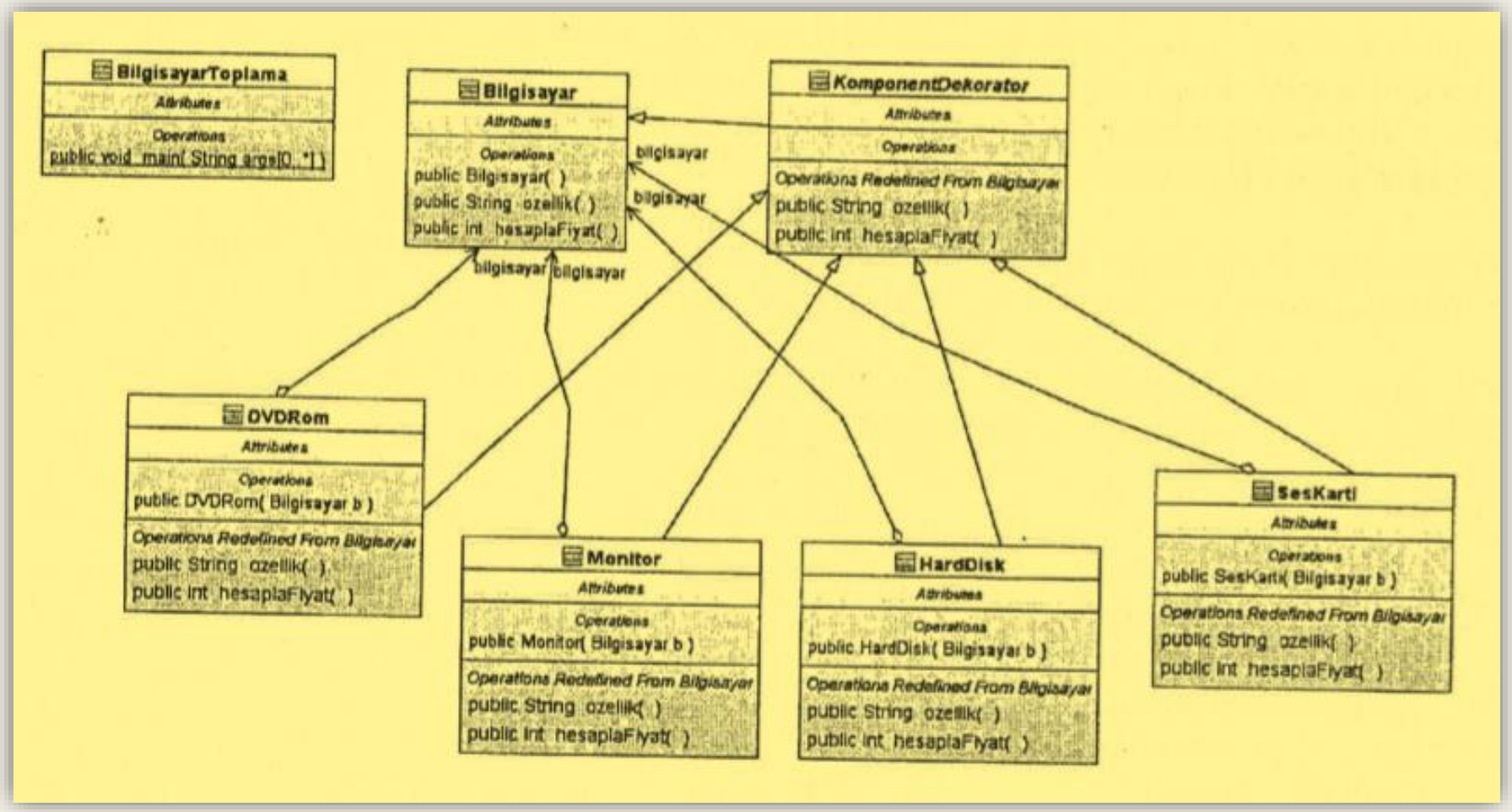
Dekoratör Tasarım Kalıbı

- E-ticaret sitemize decorator tasarım kalıbı ile her türlü konfigürasyonu çok kolay bir şekilde yapma olanağı veriyoruz.
- Kalıtım yönteminin tersine, konfigürasyonu belirleme işini derleme esnasında değil de çalıştırma esnasında yapıyoruz.

Tasarım Kuralı: Uygulamaya program yazmıyoruz, arayüze uygulama yapıyoruz. Böylece çalışma (run-time) esnasında istediğimiz değişikliği rahatlıkla yapabiliyoruz.

- Sistemimizde DVDRom, Monitör, HardDisk ve SesKarti komponentleri var.
- Temel bir bilgisayar sınıfımızı çerçeveleyerek (wrap) yeni özellikler kazandırmaya çalışıyoruz.
- Her kompenent bilgisayar özelliğini değiştirecek ilgili özelliklerin bilgisayar sınıfını çerçevelemesini (wrap: dekorasyona ekleme) KompenentDekorator soyut sınıfımız üstleniyor.
- Tüm kompenentler ise KompenentDekorator sınıfımızdan türüyorlar ve bilgisayar nesnesine sahipler

Dekoratör Tasarım Kalıbı



Dekoratör Tasarım Kalıbı

- Tasarımımızı incelemeye Bilgisayar nesnemizden başlayalım. Çünkü yapmak istediğimiz şey dekorasyonu bu nesnenin üzerine kurmak. Tüm özellikleri bu nesnenin etrafina çevirmek (wrap).

```
public class Bilgisayar {
    public Bilgisayar(){

    }
    public String ozellik(){
        return("Bilgisayar : ");
    }
    public int hesaplaFiyat(){
        return 400;
    }
```

- Ozellik() motodumuz bilgisayaramınızın özelliklerini tutmak için kullanılacak.
- hesaplaFiyat() ise her eklenen konfigürasyon için fiyat hesaplaması yapacak.

Dekoratör Tasarım Kalıbı

- Dokorasyon (Dekoratör) sınıfımızı ise soyut sınıf olarak tanımlıyoruz. Soyut sınıftan türeyen her iki metodun gövdesini tanımlayacaktır.

```
public abstract class KomponentDekorator extends Bilgisayar
{
    public abstract String ozellik();
    public abstract int hesaplaFiyat();
}
```

- İlk dekorasyonumuz DVDRom dekorasyonu:

```
public class DVDRom extends KomponentDekorator{
    Bilgisayar bilgisayar;
    public DVDRom(Bilgisayar b){
       bilgisayar = b;
    }
    public String ozellik(){
        return bilgisayar.ozellik()+" - DVD Rom ";
    }
    public int hesaplaFiyat() {
        return bilgisayar.hesaplaFiyat()+80;
    }
}
```

Dekoratör Tasarım Kalıbı

- Harddisk:

```
public class HardDisk extends KomponentDekorator{
    Bilgisayar bilgisayar;
    public HardDisk(Bilgisayar b){
        bilgisayar = b;
    }
    public String ozellik(){
        return bilgisayar.ozellik()+" - Harddisk ";
    }

    public int hesaplaFiyat() {
        return bilgisayar.hesaplaFiyat()+120;
    }
}
```

- Monitor:

```
public class Monitor extends KomponentDekorator{
    Bilgisayar bilgisayar;
    public Monitor(Bilgisayar b){
        bilgisayar = b;
    }
    public String ozellik(){
        return bilgisayar.ozellik()+" - Monitör ";
    }
    public int hesaplaFiyat() {
        return bilgisayar.hesaplaFiyat()+170;
    }
}
```

Dekoratör Tasarım Kalıbı

- Ses Kartı:

```
public class SesKarti extends KomponentDekorator{
    Bilgisayar bilgisayar;
    public SesKarti(Bilgisayar b){
        bilgisayar = b;
    }
    public String ozellik(){
        return bilgisayar.ozellik()+" - Ses Kartı ";
    }
    public int hesaplaFiyat() {
        return bilgisayar.hesaplaFiyat()+65;
    }
```

- Vivapc bilgisayar firmasının 3 farklı bilgisayar modeli olsun. Firma olarak 3 farklı tipte bilgisayar toplama seçeneği sunulsun.
- Konfigürasyonlar şu şekilde olacaktır:

Vivapc_basic: HardDisk

Vivapc_orta: HardDisk, Monitor

Vivapc_ileri: HardDisk, Monitor, DVDRom, SesKarti

Dekoratör Tasarım Kalıbı

```
public class BilgisayarToplama {
    public static void main(String[] args) {
        Bilgisayar vivapc_basic = new Bilgisayar();
        Bilgisayar vivapc_orta = new Bilgisayar();
        Bilgisayar vivapc_ileri = new Bilgisayar();

        vivapc_basic = new HardDisk(vivapc_basic);
        System.out.printf(" vivapc_basic : Konfigürasyon Fiyatı :%d,
Özellikler (%s)\n",vivapc_basic.hesaplaFiyat(),vivapc_basic.ozellik());

        vivapc_orta  = new HardDisk(vivapc_orta);
        vivapc_orta = new Monitor(vivapc_orta);
        System.out.printf(" vivapc_orta: Konfigürasyon Fiyatı :%d,
Özellikler (%s)\n",vivapc_orta.hesaplaFiyat(),vivapc_orta.ozellik());

        vivapc_ileri  = new HardDisk(vivapc_ileri);
        vivapc_ileri = new Monitor(vivapc_ileri);
        vivapc_ileri = new DVDRom(vivapc_ileri);
        vivapc_ileri = new SesKarti(vivapc_ileri);
        System.out.printf(" vivapc_ileri: Konfigürasyon Fiyatı :%d,
Özellikler (%s)\n",vivapc_ileri.hesaplaFiyat(),vivapc_ileri.ozellik());

    }

}
```

Dekoratör Tasarım Kalıbı

Program Çıktısı:

```
vivapc_basic : Konfigürasyon Fiyatı :520, Özellikler (Bilgisayar :  -
Harddisk )
 vivapc_orta: Konfigürasyon Fiyatı :690, Özellikler (Bilgisayar :  -
Harddisk  - Monitör )
 vivapc_ileri: Konfigürasyon Fiyatı :835, Özellikler (Bilgisayar :  -
Harddisk  - Monitör  - DVD Rom  - Ses Kartı )
```

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- Singleton tasarım kalıbı, bir nesnenin sadece ve sadece bir kez yaratılmasını sağlar.
- Birden fazla ihtiyaç duyulması durumunda, daha önceden yaratılmış olan nesnenin kullanılmasını sağlar.
- Bir uygulamamız olsun ve uygulamamızda veritabanı erişimi de kullandığımızı düşünelim. Veritabanına bağlantıyı sağlayan bir sınıfımız olsun.
- Normalde aynı oturumda, veritabanına bağlanılmışsa bir daha bağlanılmasına gerek yoktur. Bir kez daha bağlanılması hem yavaşlığa hem de gereksiz kaynak kullanımına neden olacaktır.
- Eğer veritabanı bağlantısını sağlayan nesnenin sadece bir kez yaratılmasını garanti alına alamazsak, uygulama döngüsü içinde hata ile bile olsa birden fazla yaratılma olasılığı olabilir.
- Bir sınıfın veya nesnenin sadece bir kez yaratılmasını garanti altına alan tasarım Singleton Tasarım Kalıbıdır.

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- Nesnemizin kontrolsüz yaratılmasının önüne geçmenin ilk yolu, nesnemizin yapılandırıcısını public yapmamaktır. Bunun yerine private yaparak dış dünyanın buna erişimini engellemeliyiz.

```
public class VeriTabaniBaglanti {
    private VeriTabaniBaglanti(){
        System.out.println("Veritabanı bağlantısı sağlandı");
    }
}
```

- Aşağıdaki sınıf derleme hatası verecektir:

```
public class Test {
    public static void main(String[] args) {
        VeriTabaniBaglanti vb = new VeriTabaniBaglanti();

    }
}
```

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- Her private değişkende olduğu gibi, private yapılandırıcıya erişmek için, public bir metoda (getter) ihtiyacımız var.
- VeriTabaniBaglanti sınıfımıza aşağıdaki metodu ekliyoruz:

```
public static VeriTabaniBaglanti getBaglantiNesne(){
    return new VeriTabaniBaglanti();
}
```

- Artık VeriTabaniBaglanti nesnemizin yapılandırıcısını çağırabiliyoruz, artık static metodumuzu direct kullanabiliyoruz:

```
public class Test {
    public static void main(String[] args) {
        VeriTabaniBaglanti.getBaglantiNesne();

    }
}
```

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- Yaratılan VeriTabaniBaglanti nesnesini bir değişkene atarak bu değişkenin kontrolü ile kontrollerimiz yapabiliriz:

```
public class VeriTabaniBaglanti {
    private static VeriTabaniBaglanti birkezYaratma;
    private VeriTabaniBaglanti(){
        System.out.println("Veritabanı bağlantısı sağlandı");
    }
    public static VeriTabaniBaglanti getBaglantiNesne(){
        if(birkezYaratma == null) {
            birkezYaratma = new VeriTabaniBaglanti();
        }
        return birkezYaratma;

    }
}
```

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

```
public static synchronized  VeriTabaniBaglanti getBaglantiNesne(){
      if(birkezYaratma == null) {
          birkezYaratma = new VeriTabaniBaglanti();
      }
      return birkezYaratma;
  }
```

- Synchronized işleminin performans açısından ciddi maliyeti vardır. Normal bir işlemle kıyaslığında en az 100 kat daha fazla işlem yapıyor.
- Bu durumda ilgili blok her çağrıldığında senkronizasyon işlemi yapacak ve bu da uygulamayı çok yoracaktır.

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- VeriTabaniBaglanti sınıfımız:

```
public class VeriTabaniBaglanti {
    private static volatile VeriTabaniBaglanti birkezYaratma;
    private VeriTabaniBaglanti(){
        System.out.println("Veritabanı bağlantısı sağlandı");
    }

    public static VeriTabaniBaglanti getBaglantiNesne(){
        if(birkezYaratma == null) {
            synchronized(VeriTabaniBaglanti.class){
                if(birkezYaratma == null) {
                    birkezYaratma = new VeriTabaniBaglanti();
                }
            }
        } else {
            System.out.println("Bağlantı zaten var. Bunu
kullanabilirsin.");
        }
        return birkezYaratma;
    }
}
```

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- **Volatile**
  - *Sadece değişkenlere uygulanır*
  - *Çoklu prosesli işlemlerde kullanılmak için tercih edilir*
  - *Volatile tanımlaması ile değişkenin sekronize edilmiş çoklu işleme adandığını ve derleyicinin bu değişkeni optimize etmesi için uğraşmaması gerektiğini belirtiyoruz.*

Yalnızlık-Tekil Tasarım Kalıbı (Singleton Design Pattern)

- Test sınıfımız:

```
public class Test {
    public static void main(String[] args) {
        VeriTabaniBaglanti.getBaglantiNesne();
        VeriTabaniBaglanti.getBaglantiNesne();
        VeriTabaniBaglanti.getBaglantiNesne();
        VeriTabaniBaglanti.getBaglantiNesne();

    }
}
```

- Uygulama Çıktımız:

```
Veritabanı bağlantısı sağlandı
Bağlantı zaten var. Bunu kullanabilirsin.
Bağlantı zaten var. Bunu kullanabilirsin.
Bağlantı zaten var. Bunu kullanabilirsin.
```

4’Lü Çete (Gang of Four) Tasarım Kalıpları

- **Creational Kalıplar (Yaratımsal)**

  Yazılım sistemindeki nesnelerin yaratılışı hakkında yol gösterirler.

  - *Singleton*

- **Structural Kalıplar (Yapısal)**

  Yazılım sistemindeki nesnelerin birbirleriyle olan ilişkilerini gösteren kalıptır.

  - *Decorator*

- **Behavioral Kalıplar (Davranışsal)**

  Nesne davranışlarını takip eden kalıptır.

  - *Observer*
  - *Mediator*
  - *Strategy*

4’Lü Çete (Gang of Four) Tasarım Kalıpları

- **Creational Kalıplar (Yaratımsal):** Nesnelerin oluşturulması ve yönetilmesi ile ilgili kalıplardır. (5 Tasarım Kalıbı)
  - Singleton: Uygulamanın yaşam süresince bir nesnenin bir kez oluşturulmasını sağlar.
  - Abstract Factory: Birbirleri ile ilişkili sınıfların oluşturulmasını düzenler.
  - Builder: Biden fazla parçadan oluşan nesnelerin üretilmesinden sorumludur.
  - Factory Method: Aynı arayüzü kullanan nesnelerin üretiminden sorumludur.
  - Prototype: Var olan nesnelerin kopyasının üretiminden sorumludur.

4’Lü Çete (Gang of Four) Tasarım Kalıpları

- **Structural Kalıplar (Yapısal):** Nesnelerin birbirleri ile olan ilişkilerini düzenleyen kalıplardır. (7 Tasarım Kalıbı)
  - Adapter: Uygulamada ki bir yapıya dışarıdaki bir yapıyı uygulamayı düzenler.
  - Bridge: Nesnelerin modelleme ve uygulanmasını ayrı sınıf hiyerarşilerinde tanımlanmasını düzenler.
  - Composite: Ağaç yapısında ki nesne kalıplarının hiyerarşik olarak iç içe kullanılmasını düzenler.
  - Decorator: Bir yapıya dinamik olarak yeni metotlar eklenmesini düzenler.
  - Façade: Alt sistemlerin direkt olarak kullanılması yerine alt sistemdeki nesneleri kullanan başka bir nesne üzerinden kullanılmasını sağlar.
  - Flyweight: Sık kullanılan nesnelerin bellek yönetimini kontrol etmek için kullanılan bir tasarım desenidir.
  - Proxy: Oluşturulması karmaşık veya oluşturulması zaman alan işlemlerin kontrolünü sağlar.

4’Lü Çete (Gang of Four) Tasarım Kalıpları

- **Behavioral Kalıplar (Davranışsal):** Birden fazla sınıfın bir işi yerine getirirken nasıl davranacağını belirleyen kalıplardır. (11 Tasarım Kalıbı)
  - Chain of responsibility: Bir isteğin belli sınıflar içinde gezdirilerek ilgili sınıfın işlem yapmasını yönetir.
  - Command: İşlemlerin nesne haline getirilip başka bir nesne(invoker) üzerinden tetiklendiği bir tasarım desenidir.
  - Interpreter: İşlemlerin nesne haline getirilip başka bir nesne(invoker) üzerinden tetiklendiği bir tasarım desenidir.
  - Iterator: Nesne koleksyonlarının elemanlarını belirlenen kurallara göre elde edilmesini düzenler.
  - Mediator: Çalışmaları birbirleri ile aynı arayüzden türeyen nesnelerin durumlarına bağlı olan nesnelerin davranışlarını düzenler.

4’Lü Çete (Gang of Four) Tasarım Kalıpları

- Memento: Bir nesnenin tamamının veya bazı özelliklerinin tutularak sonradan tekrar elde edilmesini sağlar.
- Observer: Bir nesnede meydana gelen değişikliklerde içinde bulundurduğu listede bulunan nesnelere haber gönderen tasarım desenidir.
- State: Nesnelerin farklı durumlarda farklı çalışmalarını sağlar.
- Strategy: Bir işlemin birden fazla şekilde gerçekleştirile bilineceği durumları düzenler.
- Template method:Bir algoritmanın adımlarının abstract sınıfta tanımlanarak farklı adımların concrete sınıflarında overwrite edilip çalıştırılmasını düzenler.
- Visitor: Uygulamada ki sınıflara yeni metotlar eklenmesini düzenler.

Kaynaklar:
Design Patterns: Elements of Reusable Object-Oriented Software, by Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides.
Mehmet Demir, Object Oriented Lecture Notes.
http://harunozer.com/makale/tasarim_desenleri__design_patterns.htm

MİMARİ TASARIMI

Genel Bakış

- Yazılım mimarisi, tüm bir yazılım sisteminin temel tasarımıdır.
- Sistemde hangi öğelerin bulunduğunu, her bir öğenin hangi işlevi olduğunu ve her bir öğenin birbiriyle nasıl ilişkili olduğunu tanımlar.
- Tüm sistemin büyük bir resmidir (sistemin nasıl çalıştığını gösterir).
- Bir yazılım sistemi tasarlamak için, bir yazılım mimarı birçok faktörü göz önünde bulundurmalıdır:
  - *Sistemin amacı,*
  - *Sistemin kullanıcıları,*
  - *Kullanıcılar için en önemli olan nitelikler (qualities),*
  - *Sistemin çalışacağı yer.*

Genel Bakış

- Yazılım mimarisi, özellikle büyük sistemler için önemlidir.
- Eğer genel sistemin başlangıcından itibaren net bir tasarımı varsa, geliştiricilerin izlemesi gereken sağlam bir temel vardır.
- Her geliştirici daha sonra neyin uygulanması gerektiğini ve istenen ihtiyaçları etkin bir şekilde karşılamak için işlerin nasıl ilişkili olduğunu bilir.
- Bu durum çatışmaları, çoğaltmayı ve gereksiz işleri yapmayı önler.
- Yazılım mimarisi, sistemin korunmasını, yeniden kullanılmasını ve uyarlanmasını kolaylaştırır.
- Yazılım mimarisini geliştirmek için mimarların sistemin paydaşlarını (stakeholders) dikkate alması gerekir.
- Paydaşlar, eldeki yazılıma ilgi duyan kişilerdir. Ya sistemi kullanıyorlar ya da bir şekilde faydalanıyorlar.

Genel Bakış

Paydaş (Stakeholder)	Tanımı
Yazılım Geliştiricileri (Software Developers)	Yazılım mimarisi, geliştiricilerin yapılması gerekenler konusunda güçlü bir yönlendirme ve organizasyon sağlayarak yazılım oluşturmasına ve geliştirmesine yardımcı olur.
Proje Müdürleri (Project Managers)	Yazılım mimarisi, proje yöneticilerine olası riskleri tanımlamalarına ve projeyi başarılı bir şekilde yönetmelerine yardımcı olmak için yararlı bilgiler sağlar. Yazılım mimarisi, proje yöneticilerinin görev bağımlılıklarını ve değişimin etkilerini anlamalarına ve iş görevlerini koordine etmelerine yardımcı olur.
Müşteriler (Clients)	Müşteriler, fonlar gibi, sistem hakkında önemli kararlar alırlar. Yazılım mimarisi, müşterilerle iletişim için bir temel oluşturur, böylece ne için para ödediklerini ve ihtiyaçlarının karşılandığını anlarlar.
Son Kullanıcılar (End users)	Kullanıcılar, yazılımın gerçekte nasıl tasarlandığını önemsemez, ancak onlar için “iyi çalıştığını” önemser.

Kruchten’s 4+1 View Model

- Bir yazılım sisteminin tüm davranışını ve gelişimini yakalamak için çoklu bakış açıları (multiple perspectives) gereklidir. Bunun için dikkate alınması gereken birkaç husus vardır:
  1. *Yazılımın işlevselliği, bu önemli hususlardan birisidir. İşlevsellik, bir sistemin müşterinin istediği amacı yerine getirmek için ne yaptığını içerir. Bu işlevselliğe odaklanmak logical view (mantıksal görünüm) olarak adlandırılan bir perspektif ile ilişkilendirilir.*
  2. *Diğer bir husus, sistemin verimliliği veya alt işlemlerin (subprocesses) etkileşimi gibi özellikleri kullanarak yazılımın ne kadar iyi çalıştığıdır. Bu özellikler sistemin performansını ve ölçeklenebilirliğini etkiler. Mantıksal görünümdeki nesnelerin uyguladığı süreçlere odaklanmak, process view (süreç görünümü) olarak adlandırılan bir perspektife yol açar.*
  3. *Yazılım, development view (geliştirme görünümünü) de içerebilir. Bu bakış açısı, yazılımın hiyerarşik yapısı gibi uygulama hususlarına odaklanır. Sistemin programlama dilleri bu yapıyı ağır bir şekilde etkileyecek ve bu nedenle geliştirme üzerine kısıtlamalar getirecektir.*

Kruchten’s 4+1 View Model

4. *Yazılımın bir başka perspektifi physical view (fiziksel görünümden) görülebilir. Yazılım, etkileşime giren ve konuşlandırılması (deploy) gereken fiziksel bileşenlere sahip olacaktır. Bu farklı öğeler ve bunların konuşlandırılması arasındaki etkileşim, sistemin çalışma şeklini etkiler.*

- Bu dört görünüm, istemcinin tanımladığı şekilde yazılımın amacını veya istenen yeteneklerini paylaşır. Mantıksal, süreç, geliştirme ve fiziksel görünümler Philippe Kruchten’ın 4+1 Görünüm Modelini oluşturur.
- Bu model, yazılım mimarisinde ele alınması gereken temel hususları veya önemli bakış açılarını anlamanın bir yoludur.

Logical View (Mantıksal Görünüm)

- Bir sistemin işlevsel (fonksiyonel) gereksinimlerine odaklanan mantıksal görünüm, genellikle sistemin nesnelerini içerir.
- Bu nesnelerden, mantıksal görünümü göstermek için bir UML sınıf diyagramı oluşturulabilir.
- Sınıf diyagramı ile tüm sınıfları, niteliklerini ve davranışlarını tanımlayarak, temel soyutlamaları ve terminolojiyi anlamak kolaylaşır.
- Sınıf diyagramları, veritabanı diyagramlarını belirlemek için de kullanışlıdır.
- Sınıf diyagramı, sınıfların nasıl etkileşimde bulunduğunu ve verilerin bir veritabanında birbirleriyle nasıl ilişkili olması gerektiğini görmeyi kolaylaştırır.

!!! Bir sistemin mantıksal görünümüyle ilgili en etkili UML diyagramlarından bazıları sınıf (class) diyagramı ve durum (state) diyagramıdır.
Hem sınıf diyagramı hem de durum diyagramı bir sistemin sınıflarına ve nesnelerine odaklanır.

Process View (Süreç Görünümü)

- Süreç görünümü, işlevsel olmayan (fonksiyonel olmayan) gereksinimlerin elde edilmesine odaklanmaktadır.
- Bunlar, performans ve kullanılabilirlik gibi kalite özelliklerini içeren sistem için istenen nitelikleri belirleyen gereksinimlerdir.
- İşlem görünümü ayrıca, mantıksal görünümdeki nesnelere karşılık gelen işlemleri de sunar.

!!! Bir sistemin işlem görünümüyle ilgili en etkili UML diyagramlarından bazıları, aktivite (activity)diyagramı ve dizilim (sequence) diyagramıdır.
Aktivite diyagramı, bir sistem için işlemleri veya aktiviteleri gösterebilir.
Dizilim diyagramı, nesnelerin birbirleriyle nasıl etkileşime girdiğini, bu yöntemlerin nasıl uygulandığını ve hangi sırada olduğunu gösterir.

Development View (Geliştirme Görünümü)

- Temel olarak, geliştirme görünümü hiyerarşik yazılım yapısını ve proje yönetimini kapsar.
- Ayrıca programlama dili, kütüphaneler ve araç setleri gibi unsurları da göz önünde bulundurur.
- Yazılım geliştirmenin ayrıntıları ve bunu desteklemek için neyin dahil olduğu ile ilgilenmektedir.
- Ayrıca, zamanlama, bütçeler ve iş atamaları gibi yönetim ayrıntılarını da kapsar.

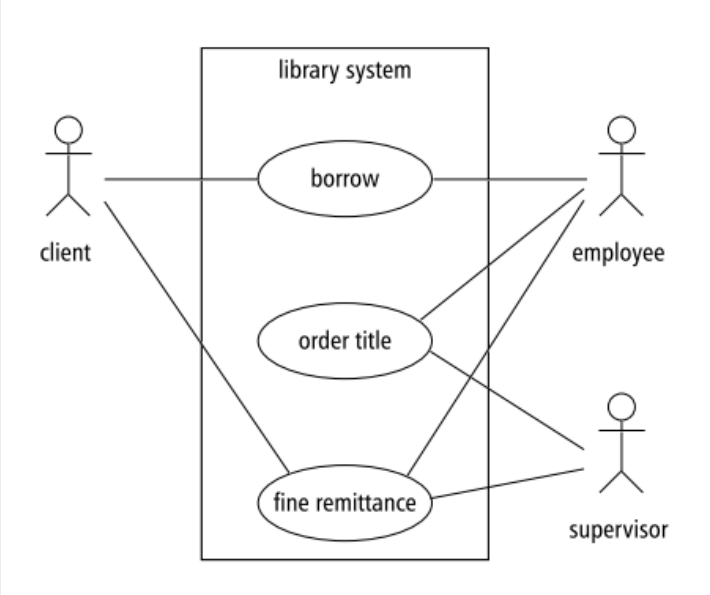
Physical View (Fiziksel Görünüm)

- Fiziksel görünüm, mantıksal, süreç ve geliştirme görünümlerindeki öğelerin sistemi çalıştırmak için farklı düğümlere veya donanımlara nasıl eşlenmesi gerektiğini ele alır.

!!! Bir sistemin fiziksel görünümü ile ilgili en etkili UML diyagramlarından biri dağılım (deployment) diyagramıdır. Bir sistemin parçalarının donanım veya yürütme (execution) ortamlarına nasıl dağıtıldığını ifade etmektedir.

Use Case

- Senaryo, belirli bir görev örneğinin nasıl yürütüldüğünü anlatan bir hikayedir. kullanıcı tarafından başlatılan çeşitli olaylar dizisidir
- Use case diyagramı, bir dizi use case hakkında genel bilgi sağlar. Her kullanım durumu, kullanım durumunun adıyla bir elips olarak gösterilir.



UML Use Case Diagram

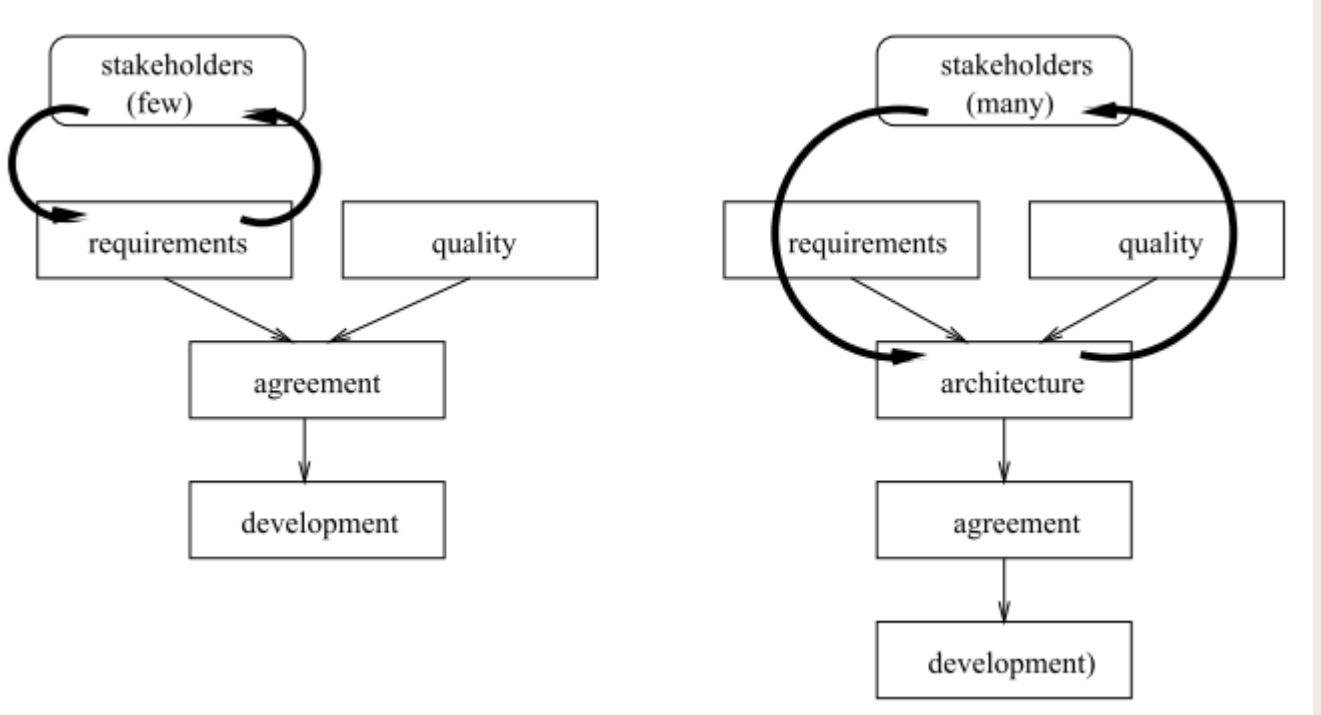
Senaryolar

- Senaryolar bir sistemin kullanım durumlarıyla (use case) veya kullanıcı görevleriyle (user task) uyumludur ve diğer dört görünümün birlikte nasıl çalıştığını gösterir.
- Her senaryoda, nesneler ve işlemler arasındaki etkileşimin sırasını tanımlayan bir komut dosyası (script) vardır.
- Bu script:
  - *Mantıksal görünümde tanımlanan anahtar nesneleri,*
  - *Süreç görünümünde açıklanan süreçleri,*
  - *Geliştirme görünümünde tanımlanan hiyerarşiyi,*
  - *Fiziksel görünümde belirtilen farklı düğümleri içerir.*
- Görünümlerin hiçbiri birbirinden tamamen bağımsız değildir.
- 4 + 1 görünüm modeli, bir yazılım sisteminin yapısını anlamak için birçok duruma uyacak şekilde kalıplanabilir.
- Karmaşık bir problemi birçok farklı perspektif ile görebilmek, yazılımınızın daha çok yönlü olmasına yardımcı olur.

Yazılım Mimarisinin Amacı

- Yazılım mimarisi, yazılım sistemlerinin büyük ölçekli yapısı ile ilgilenir.
- Bu büyük ölçekli yapı, erken, temel tasarım kararlarını yansıtır.
- Bu karar süreci, bir yandan fonksiyonel ve kalite gereksinimlerinin müzakere edilmesini ve dengelenmesini ve diğer yandan olası çözümleri içerir.
- Yazılım mimarisi kesinlikle gereksinim mühendisliğini takip eden bir aşama değildir, ikisi iç içe geçmiş durumdadır.
- Geleneksel tasarım içe dönüktür: bir takım gereksinimler verildiğinde, bu gereksinimleri karşılayan bir sistemin nasıl oluşturulacağına odaklanır.
- Yazılım mimarisi, bir yandan fonksiyonel ve kalite gereksinimlerinin tartışılmasını ve dengelenmesini ve diğer yandan olası çözümleri içerir.

Yazılım Mimarisinin Amacı



Geleneksel Modeller

Yazılım Mimarisi içeren Süreç Modelleri

Mimari Tasarımı

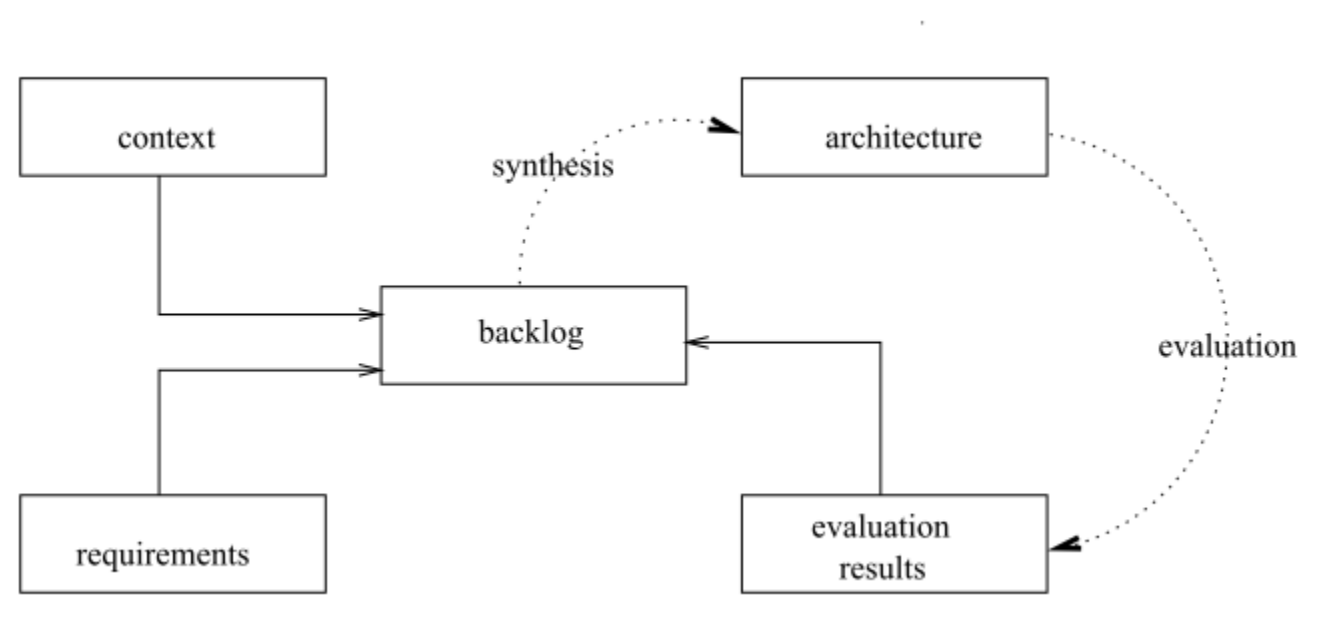
- Tasarım bir problem çözme faaliyetidir ve bu nedenle deneme yanılma meselesidir.
- Tasarım sırasında, sistem, her biri bir bütün olarak sistemden daha düşük bir karmaşıklığa sahip olan parçalara ayrılırken, parçalar birlikte kullanıcının problemini çözer.
- Tasarım süreci yaratıcı bir süreçtir ve tasarımcıların kalitesi ve uzmanlığı başarısı için kritik bir belirleyicidir. Ancak, yıllar boyunca, yazılım tasarlamada bize hizmet edebilecek bir takım fikirler ve kılavuzlar ortaya çıkmıştır.
- Bunun sonucunda mimari tasarım yöntemleri geliştirilmiştir. Buna iyi bir örnek Attribute Driven Design (ADD)'dır.
- ADD sürecinin girdisi gereksinimlerdir.

Mimari Tasarımı

- ADD, yukarıdan aşağıya ayrışma (decomposition) işlemi olarak tanımlanmaktadır.
- Her iterasyonda, daha fazla ayrıştırma için bir veya birkaç bileşen seçilir.
- İlk iterasyonda yalnızca bir bileşen vardır: Sistem.
- Kalite özelliği senaryolarından, mevcut adımda ele alınacak önemli bir kalite özelliği seçilmiştir.
- Örneğin, kütüphane sistemimizde, sistemin ilk üç katmana ayrıştırılmasına karar vermiş olabiliriz: bir sunum katmanı, bir iş mantığı katmanı ve bir veri katmanı.
- Bir sonraki ADD adımında, sunum katmanını ayrıştırmaya karar verebilir ve bu ayrışmayı başlatan kalite niteliği olarak kullanılabilirliği seçebiliriz

Mimari Tasarımı

- Daha sonra kalite niteliğini karşılayan bir kalıp (pattern) seçilir. Örneğin, veri öğelerinin doğru girilip girilmediğini doğrulamak için bir veri doğrulama modeli (data validation pattern, Folmer ve diğerleri, 2003) uygulanabilir.
- Son olarak, bir sonraki iterasyona hazırlanmak için kalite özniteliği senaryoları belirlenir.



Mimari Tasarımının Genel İş Akışı

Mimari Tasarımı

- Backlog'ın güncellenmesi, sonuçlarının değerlendirilmesi, backlog içinde kullanılacak öğelerin belirlenmesi mimar tarafından yapılır ve diğer paydaşlarla iletişim kurulmasını gerektirmez.
- Deneyimli bir mimar, bazı yöntemlerin tasarım yinelemesini nasıl gerçekleştireceğini anlatmaktansa, belirli bir sorunun nasıl ele alınacağını bilir.
- Mimari tasarım genellikle deneyimli tasarımcılar tarafından yapıldığından, verilen ve ihtiyaç duyulan rehberlik miktarı daha azdır.
- Tasarım sürecinin sonucunu belgelemek için teknikleri kullanır: kararlar, gerekçeleri ve ortaya çıkan tasarım.

Yazılım Mimarisinin Kalite Ölçütleri

- Kalite özellikleri, bir sistemin tasarımını, çalışma zamanı performansını ve kullanılabilirliğini ölçmek için kullanılan sistemin ölçülebilir özellikleridir. Bu kalite özellikleri şunlardır:
  1. Sürdürülebilirlik (maintainability): Sisteminizin kolaylıkla değişiklik yapma yeteneğine sahip olması. Sistemler yaşam döngüsü boyunca değişime uğrayacaklar, bu nedenle bir sistem bu değişiklikleri kolaylıkla kaldırabilmelidir.
  2. Tekrar Kullanılabilirlik (reusability): Sisteminizin fonksiyonlarının veya bölümlerinin bir başka sistemde ne ölçüde kullanılabileceğidir. Yeniden kullanılabilirlik, daha önce yapılmış bir şeyi yeniden uygulama maliyetini azaltmaya yardımcı olur.
  3. Esneklik (Flexibility): Bir sistemin gereksinim değişikliğine ne kadar iyi uyum sağlayabileceği. Oldukça esnek bir sistem gelecekteki ihtiyaç değişikliklerine zamanında ve uygun maliyetli bir şekilde adapte olabilir

Yazılım Mimarisinin Kalite Ölçütleri

4. Değiştirilebilirlik (Modifiability): Bir sistemin değişikliklerle başa çıkma, yeni birleşme veya mevcut fonksiyonelliği kaldırma yeteneği. Bu elde edilecek en pahalı tasarım kalitesidir, bu nedenle uygulama değişikliklerinin maliyeti bu özellik ile dengelenmelidir.
5. Test edilebilirlik (Testability): Testler hızlı, kolay bir şekilde yapılabildiğinden ve bir kullanıcı arayüzü gerektirmediğinden sistemler test edilmelidir. Bu, arızaların tespit edilmesine yardımcı olacaktır, böylece sistem serbest bırakılmadan önce düzeltilebilirler.

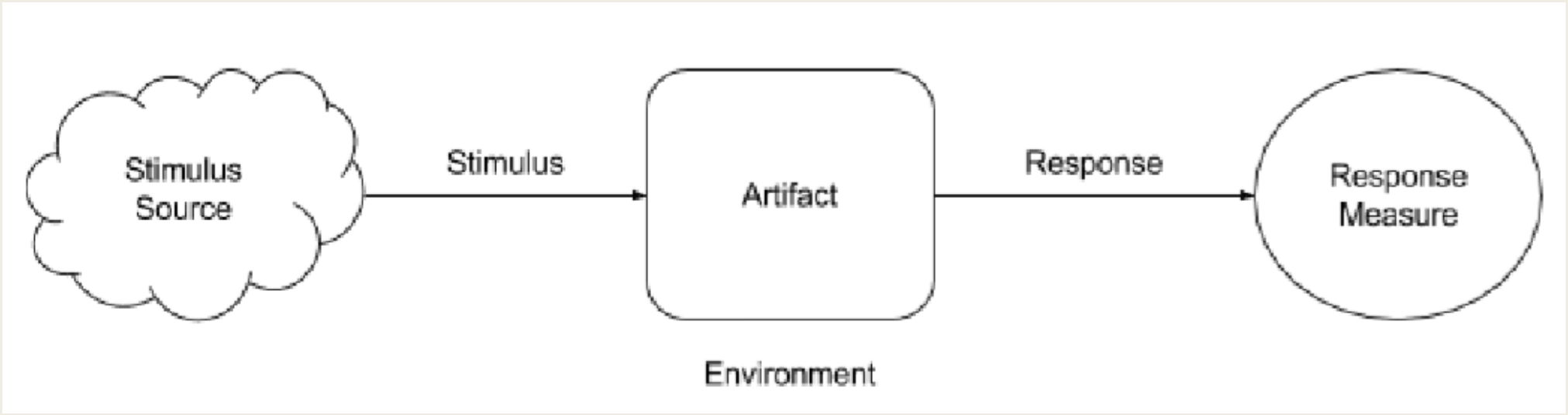
Yazılım Mimarisinin Kalite Ölçütleri

- Kullanıcı tarafından dikkate alınacak olan kalite özellikleri şunlardır:
  1. Erişilebilirlik (availability): Sistemin belirli bir süre boyunca çalıştığı süre. Bir sistemin kullanılabilirliği çalışma süresiyle ölçülür, böylece sistemin sistem hataları, yüksek yükler veya güncellemeler gibi sorunlardan ne kadar iyi kurtulduğu tespit edilebilir. Bir sistem bu sorunların aksama sürelerine neden olmasını önleyebilmelidir.
  2. Birlikte çalışabilirlik (Interoperability): Sisteminizin iletişimleri anlama ve verileri harici sistemlerle paylaşma yeteneği. Bu, sistemin arayüzleri anlama ve bu harici sistemler ile belirli koşullar altında bilgi alışverişinde bulunmak için kullanma yeteneğini de içerir. Bu, iletişim protokollerini, veri formatlarını ve sistemin kiminle bilgi alışverişinde bulunabileceğini içerir.
  3. Güvenlik (security): Sistemin hassas verileri yetkisiz ve yetkisiz kullanımlara karşı koruma yeteneği.

Bir Mimariyi Analiz Etme ve Değerlendirme

- Bir sistemi tasarlamaya ek olarak, tüm paydaşların endişelerini veya gereksinimlerini ele alıp almadığını belirlemek için tasarımın nasıl değerlendirileceğini bilmek önemlidir.
- Yazılım mimarisini analiz etmek ve değerlendirmek, yazılımın soyut doğası nedeniyle zor olabilir.
- Ancak, bir sistemin davranışlarını, kalite özelliklerini ve çeşitli özelliklerini metodik olarak analiz etmek ve değerlendirmek önemlidir.
- Kalite niteliklerini ölçmek için, bir sistemin özellik için belirlenen gereklilikleri yerine getirip getiremediğini belirlemek için kalite özellik senaryoları kullanılabilir. İki tür senaryo vardır:
  - *Herhangi bir sistemi karakterize etmek için kullanılan **genel bir senaryo***
  - *Belirli bir sistemi karakterize etmek için kullanılan **somut bir senaryo***

Bir Mimariyi Analiz Etme ve Değerlendirme



Bir Mimariyi Analiz Etme ve Değerlendirme