

Nesne Yönelimli Analiz ve Tasarım

Java'da dosyanın adı ne ise sınıfın da adı o olmalıdır.

**IntelliJ'de Alt+Insert ile parametresiz, 1 veya daha fazla parametreli kurucular vb. veya hızlıca getter-setter'lar oluşturulabilir. Çok önemli, güzel bir özellik!

Nesne print edildiğinde nesnenin sınıfındaki "public String toString()" şeklinde tanımlanmış fonksiyonu çalıştırılır. Bu yöntem nesneyle ilgili bilgi vermek amacı ile kullanılır. Bu metot @Override edilmeli çünkü Java gizliden Object sınıfından kalıtım alıyor, onu ezmek için.

Eğer bir sınıfın kalıtım almışsa ve kalıtım alan sınıfın toString metodu bulunmuyorsa ve bu nesne yazdırılmak istenirse öncelikle kalıtım alınan sınıfın toString metodu var mı ona bakar, yoksa Object'inkini yazar.

Sınıf içerisinde static ve dinamik(nesne) üyeleri bulunur. Sınıf içerisindeki static olarak tanımlanmış üyeleri nesne oluşturulmadan "SinifAdi.NesneAdi" şeklinde kullanabiliriz.

Örn javanın kendisinde bulunan Math sınıfının pow fonksiyonu. Math sınıfından obje oluşturulmadan Math.pow() şeklinde kullanılabilir.

Atanmayan değişkenin default değeri 0'dır.(int, float, double)

`SinifAdi nesneAdi=new SinifAdi(...);` şeklinde nesne oluşturulur.

`new SinifAdi(..)` şeklinde direkt olarak referanssız nesne oluşturarak print edilirse `toString` fonksiyonu çalışır. Yani daha sonra kullanmayıcağız yalnızca `toString` metodundan yararlanmak istediğimiz zamanlar olursa bu yöntemi kullanabiliriz.

System.in → klavyeyi, konsolu temsil eder

Klavyeden veri girişi alabilmek için import java.util.Scanner; şeklinde Scanner'ı import etmek gereklidir ve sonrasında Scanner'dan nesne oluşturmak gereklidir.

`Scanner input=new Scanner(System.in);` → şeklinde nesnesi oluşturulur.

sonrasında örn:

```
System.out.print ("Yarıçapı giriniz ");
int yarıCap = input.nextInt(); → yarıCap'a alınan değer atanır.
```

next-VeriTipiAdı- şeklinde çalışır. (Örn: nextFloat, nextDouble gibi)

```
input.next() → boşluğa kadar okuma yapar(boşluk dahil) (Yani string)
```

```
input.nextLine() → komple satır(enter'a kadar) okuma yapar (Yani string)
```

```
import java.text.DecimalFormat; → ile sayılar biçimlendirilebilir.
```

Öncelikle nesne üretilir,

```
DecimalFormat fmt = new DecimalFormat ("0.#####"); → fmt adında üretilmiş ve sayının virgülüden sonra 4 basamağını al demış.
```

Örn 28.274333882308138 sayısını fmt.format(28.274333882308138) şeklinde print edersek 28,2743 çıktısı alırız.

Encapsulation (Kapsulleme)

Nesne yönelimli programlamada herhangi bir nesnenin yani sınıfın fonksiyonlarını, verilerini ve değişkenlerini diğer nesnelerden saklayarak ve bunlara erişimini sınırlılaştırarak yanlış kullanımlardan koruyan bir konsepttir. Yani değişkenleri private olarak tanımlayıp get set kullanıdır.

Java paketleme sistemi bulunur, aynı paketi başta belirttiğimiz birbirleri ile haberleşebiliyor olur.

Örneğin x,y,z class'ı varsa ve her birinde başta

```
package cc.ders2.Kalitim;
```

dersek Kalitim paketinin içerisinde bulunan main fonksiyonunu çalıştırır. (Main tek olmalı.)

Soyut Sınıf (Abstract)

Nesne yönelimli programlamada nesnesi yaratılamayan sınıflara denir. Nesne yaratılamamasının nedeni, sınıfın kullanıcı arayüzünde yer alan bir veya daha çok sayıdaki iletinin gerçekleştirilmemesidir.

Yani abstract sınıfın `SınıfAdı nesneAdı=new SınıfAdı()` şeklinde nesne türetilemez.

Normalde sınıf tanımlamasında `public class SınıfAdı` şeklinde tanımlama yapılır, abstract oluşturmak için `abstract public class SınıfAdı` denmesi yeterlidir.

Aynı şekilde soyut fonksiyonlar da olabilir. Örneğin A temel sınıfında bulunan X fonksiyonu B, C,...,Z kalıtım alan sınıfların her biri için farklı şekilde davranışacaksa A temel sınıfındaki

X'i soyut fonksiyon yaparak tanımlayabiliriz.

`public abstract int X();` şeklinde temel sınıfta gövdesiz olarak tanımlanır, kalıtım alan sınıflarda her birinde yapacağı işe göre içi doldurulur.

***Soyut fonksiyon bulunacak sınıfın da muhakkak soyut olarak tanımlanmış olması gereklidir.

Örn;

Bu şekilde bir GeometrikSekil soyut sınıfı tanımladığımızı varsayırsak, Alan ve Çevre geometrik şekillerde hesaplama olarak farklı olacağı için gövdelerini yazmayız ve kalıtım alacak sınıfların bunu doldurmasını bekleriz. Bu sınıfın kalıtım alacak soyut olmayan her sınıf bu gövdesiz fonksiyonları @Override ederek doldurmak zorundadır.

```
abstract public class GeometrikSekil {  
    private String renk;  
    private Date olusturulmaTarihi;  
    public GeometrikSekil(String renk) {  
        olusturulmaTarihi=new Date();  
        this.renk=renk;  
    }  
    @Override  
    public String toString() {  
        String cikti="Tarih: "+olusturulmaTarihi;  
        cikti+="\\n Renk: "+renk;  
        cikti+="\\nAlan: "+Alan();  
        cikti+="\\nCevre: "+Cevre();  
        return cikti;  
    }  
    public abstract double Alan();  
    public abstract double Cevre();  
}  
  
public class Daire extends GeometrikSekil{  
    private double yariCap;  
    public Daire(String renk, double yariCap) {  
        super(renk);  
        this.yariCap=yariCap;  
    }  
    @Override  
    public double Alan() {  
        return Math.PI*Math.pow(yariCap, 2);  
    }  
    @Override  
    public double Cevre() {  
        return Math.PI*2*yariCap;  
    }  
}  
  
public class Kare extends GeometrikSekil {  
    private int kenar;  
    public Kare(String renk, int kenar) {  
        super(renk);  
        this.kenar=kenar;  
    }  
    @Override  
    public double Alan() {  
        return Math.pow(kenar, 2);  
    }  
}
```

```

@Override
public double Cevre() {
    return 4*kenar;
}

public class Program {
    public static void main(String[] args) {
        Daire daire=new Daire("Mavi", 12);
        System.out.println(daire);
        System.out.println();
        Kare kare=new Kare("Yeşil", 54);
        System.out.println(kare);
    }
}

```

Kalıtım

Kalıtım yapılacakken temel sınıf abstract yapılır ki temel sınıfın nesne oluşturulamasın, kalıtım alan sınıflardan oluşturulabilisin. yani temel sınıf A ise abstract public class A şeklinde tanımlama yapılır.

Kalıtım alan sınıfta ise extends kullanılır (Temel sınıf A, alan B ise) → `public class B extends A` şeklinde kalıtım alınır.

Kalıtım alan sınıfta temel sınıfın kurucusu `super(...)` şeklinde çalıştırılır. Örn: (A temel, B kalıtım alan sınıf ise)

```

abstract public class A {
    private double konumX;
    private double konumY;
    private String renk;

    public Sekil(double konumX, double konumY, String renk) {
        this.konumX = konumX;
        this.konumY = konumY;
        this.renk = renk;
    }
}

```

şeklinde temel sınıf tanımlandıysa,

```

public class Daire extends B {
    private float yaricap;

    public Daire(double konumX, double konumY, String renk, float yaricap) {
        super(konumX, konumY, renk);
        this.yaricap = yaricap;
    }
}

```

ve B kalıtım alan sınıfı ek olarak yarıcap bilgisi mevcut ise kurucu fonksiyonunda temel sınıfındaki bilgilerle beraber yarıcap alınır ve `super(konumX, konumY, renk);` şeklinde öncelikle temel sınıfın kurucusu çalıştırılır ve ardından yarıcap'a atama yapılır.

Komut satırında kalıtım çalıştırmak için,

```
javac -d . KalitimUygulamasi.java X.java Y.java Z.java  
java cc.ders2.Kalitim.KalitimUygulamasi
```

şeklinde çalıştırılır. KalitimUygulamasi main içeren dosyadır, X temel, Y ve Z ise kalıtım alan sınıflarıdır.

Pluginlerden Paket üzerinde sağ tık → Diagrams şeklinde paketin diyagramını da oluşturup, görüntüleyebiliriz.

Çok Biçimlilik

Bir türün bir başka tür gibi davranışabilme ve bu tür gibi kullanılabilme özelliği. Overloading de buna bir örnek olabilir.

Kalıtım alan sınıfı `@Override` ile tanımlanır. Yani bu etiket konulduğunda çok biçimlilik yapılmış olur ve temel sınıfındaki X fonksiyonunu (örneğin fonk adı X ise) geçersiz kılarak X fonksiyonu çağrııldığında temel sınıfındaki X'i çalıştırırmak yerine kalıtım alan sınıfındaki X fonksiyonu çalıştırılır.

Örn:

```
public class A {  
    public double X(){  
        System.out.println("Alan hesaplanıyor...");  
        return 0;  
    }  
}  
public class B extends A {  
    @Override  
    public double X(){  
        System.out.println("Alan hesaplanıyor...");  
        return 0;  
    }  
}
```

tanımlamaları yapıldıktan sonra,

`B deneme=new B();` dendikten sonra `deneme.X()` çağrııldığından çok biçimlilikten dolayı normalde A temel sınıfındaki X çağrılacakken, B kalıtım alan sınıfındaki X çağrırlar.

Aynı zamanda normalde bir dizi oluşturduğumuzda tüm elemanlarının tek veri tipinde olması gereklidir. Örneğin intse tümü int olmalı. Fakat çok biçimlilik sayesinde normalde

temel sınıf üzerinden oluşturulmuş örneğin temel sınıf A ise A [] dizi=new A[3] şeklinde 3 elemanlı bir dizi oluşturursak kalıtım alan sınıflarından oluşturulan nesneleri normalde bu dizide A sınıfından nesne olmadığı için tutamayacakken, çok biçimlilik sayesinde tutabiliriz. Örn: Temel sınıf A, kalıtım alan B, C

```
A [] dizi=new A[3];
```

```
B b1=new B(..);
```

```
B b2=new B(..);
```

```
C c1=new C(..);
```

normalde b1, b2, c1 nesnelerini dizi dizisinde tutamayacakken çok biçimlilik sayesinde tutabiliriz.

```
dizi[0]=b1; dizi[1]=b2; dizi[2]=c1;
```

Aynı zamanda çok biçimlilik sayesinde herhangi bir fonksiyona temel sınıf nesnesi parametresi alarak parametre olarak gelecek kalıtım alan nesneler de ortak olarak kullanılabilir. Örn:

```
public static void yazdir(A x){  
    System.out.println(x);  
    System.out.println("Method="+x.metot());  
}
```

x instanceof A → örneğin if ile kullanıldığında x nesnesi A sınıfından üretilen bir nesne mi demektir.

Arayüz (Interface)

Nesne yönelimli programlama dillerinde arayüz, değişik sınıflardan nesnelerin kategorize edilmesini sağlayan bir soyut tür çeşididir.

Bir çeşit soyutlama türüdür. Abstract class da soyutlamadır, arayüz de soyutlamadan bir çeşididir.

Örneğin bir programda bir veritabanına bağlanmamız gerekiyor ve bunu örneğin MySQL'e göre değil de herhangi bir veritabanına bağlayabilecek şekilde organize etmemiz gerekiyor. Bunu da interfaceler sağlayabilir.

Interfacede ve interface'e bağlanacak sınıflarda aynı imzaya sahip fonksiyonlar bulunmak zorundadır. Aynı zamanda bağlanan sınıfların fonksiyonlarının @Override olması gereklidir.

Örn:

```
interface VeritabaniSurucu {  
    public void baglan();  
    public void baglantiSonlandir();  
}
```

şeklinde interface anahtar kelimesiyle interface oluşturulur.

```
public class PostgreSQLSurucu implements VeritabaniSurucu {  
    @Override  
    public void baglan() {  
        System.out.println("PostgreSQL veritabanına bağlanıyor...");  
    }  
    @Override  
    public void baglantiSonlandir() {  
        System.out.println("PostgreSQL veritabanı bağlantısı sonlandırılıyor...");  
    }  
}  
public class MySQLSurucu implements VeritabaniSurucu {  
    @Override  
    public void baglan() {  
        System.out.println("MySQL veritabanına bağlanıyor...");  
    }  
    @Override  
    public void baglantiSonlandir() {  
        System.out.println("MySQL veritabanı bağlantısı sonlandırılıyor...");  
    }  
}
```

şeklinde ise soyutlanacak sınıflarda implements interfaceAdi şeklinde soyutlaması yapılır ve muhakkak fonksiyonların override edilmesi gereklidir.

Daha sonra main içerisinde

```
VeritabaniSurucu veritabaniSurucu= new kullanilacakVeritabaniClassName();
```

şeklinde nesne oluşturulup veritabanı işlemleri yapmayı sağlayan sınıfa parametre olarak bunuonderiziriz.

```
VeritabaniIslemleri veritabaniIslemleri = new VeritabaniIslemleri(veritabaniSurucu);
```

```
public class VeritabaniIslemleri {  
    private VeritabaniSurucu veritabani;  
  
    public VeritabaniIslemleri(VeritabaniSurucu veritabani) {  
        this.veritabani = veritabani;  
    }  
    public void baglan(){  
        veritabani.baglan();  
    }  
.. diğer fonksiyonlar-ortak olduğu için veritabani. diyip interfacedeki fonksiyonlar yazılır.  
}
```

şeklinde olsun.

Mainde nesneyi oluşturup buna gönderdikten sonra

```
veritabaniIslemleri.baglan();
```

 dersek oluşturduğumuz veritabanının baglan fonksiyonu çalışır.

Interfaceler doğrudan soyuttur yani abstracttir. Normal şartlarda interface içerisinde hiçbir fonksiyonun gövdesi bulunamaz, yalnızca static ve default ise bulunabilir. Interface içerisinde static olarak bir tanım yapıp bunu bu interface'i implement eden tüm sınıflarda ortak bir örneğin tanım haline getirebiliriz. Örn:

```
static String tanim(){  
    System.out.println("Tanım yapıyorum.");}
```

veya default olarak bir fonksiyon tanımlayıp implemente eden sınıflarda bu fonksiyonun gövdesini kodlamasalar da default olarak interface'de bulunan fonksiyonun çalışmasını da sağlayabilirim. Eğer implemente eden sınıf default olan fonksiyonu kodlarsa bu durumda implemente eden sınıftaki fonksiyon çalışır. Örn.

```
default int x(){  
    System.out.println("Interfaceden calisiyorum");}
```

***Interface ile abstract'tan farkı interfacelerde tanımlanan her şeyin public olmasıdır. Abstract'ta private ve protected bulunabilir. Ayrıca tüm değerleri interface'in finaldir, yani bir değer tanımlandıysa atama muhakkak yapılmalıdır. Yani bunun da anlamı sabitlerdir. (C ve C++'daki const) Örn: int sayı; dersek interface içerisinde, hata alırız. Sabit bir sayı değeri atamak gereklidir, örneğin int sayı=5 dersek, implemente eden tüm sınıflarda sayı sabit olarak 5 olacaktır. Ayrıca interfacelerde kurucu metot tanımlanamazken, abstract sınıflarda kurucu metot tanımlanabilir.

İstemci kodu içerisinde değişiklik yapılmaması istenen durumdur. Tek değişikliği mainde yapmak istenen durumdur. Yani yarın özür gün başka bir veritabanı eklenliğinde class'ını interface'e göre hazırlayıp ardından interface üzerinden main içerisinde bu yeni veritabanının objesini üretmek yeterli olacak. İstemci kod burada VeritabanıIslemleri sınıfıdır. Main sınıfı ise VeritabaniUygulaması'dır.

UML'in bir çok gösterim şekli vardır. 14 taneden yaklaşık 10 taneyi ders kapsamında göreceğiz.

Etkinlik şemasında dörtgenler karar verilen yer olarak ifade edilir.

Yazılım Geliştirme Yaşam Döngüsü

1. Analiz

Geliştirilecek yazılımın gereksinimleri belirlenir.

2. Tasarım

Kısıtlar ihlal edilmeden yazılım gereksinimlerinin nasıl karşılanması gerektiğini belirlenir. UML'in kullanıldığı yerdir.

3. Gerçekleme

Önceki aşamalarda geliştirilen modeller kodlanarak yazılım uygulamasına dönüştürülür.

4. Test

Geliştirilen uygulama; işlevsellik, başarıım ve güvenlikle ilgili gereksinimler dikkate alınarak test edilir.

Bu aşamada birim testi (unit testing), tümleştirme testi (integration testing), sistem testi (system testing), kabul testi (acceptance testing) yapılır.

5. Bakım&Konuşlandırma

Sürüm Yöneticisi (Release Managers) yazılımı kullanıma hazırlar.

Uygulama kullanılmaya başladıkten sonra meydana gelen değişiklik gereksinimleri bu aşamada (maintenance) ele alınır.

UML Diyagramı

ATM İçin Yazılım Geliştirme Gereksinimlerin Belirlenmesi Diyagramı

ATM aşağıdaki donanım bileşenlerinden oluşur:

Tuş takımı; ile veri girişi yapılır.

Ekran; mesajların görüntülenmesinden sorumludur.

Para bölmesi; para alma ve para verme işleminden sorumludur.

Kart bölmesi; banka kartını alır ve doğrular.

Kullanıcılar banka kartı ile para çekme, para yatırma, bakiye görüntüleme işlemlerini yapabilmelidirler.

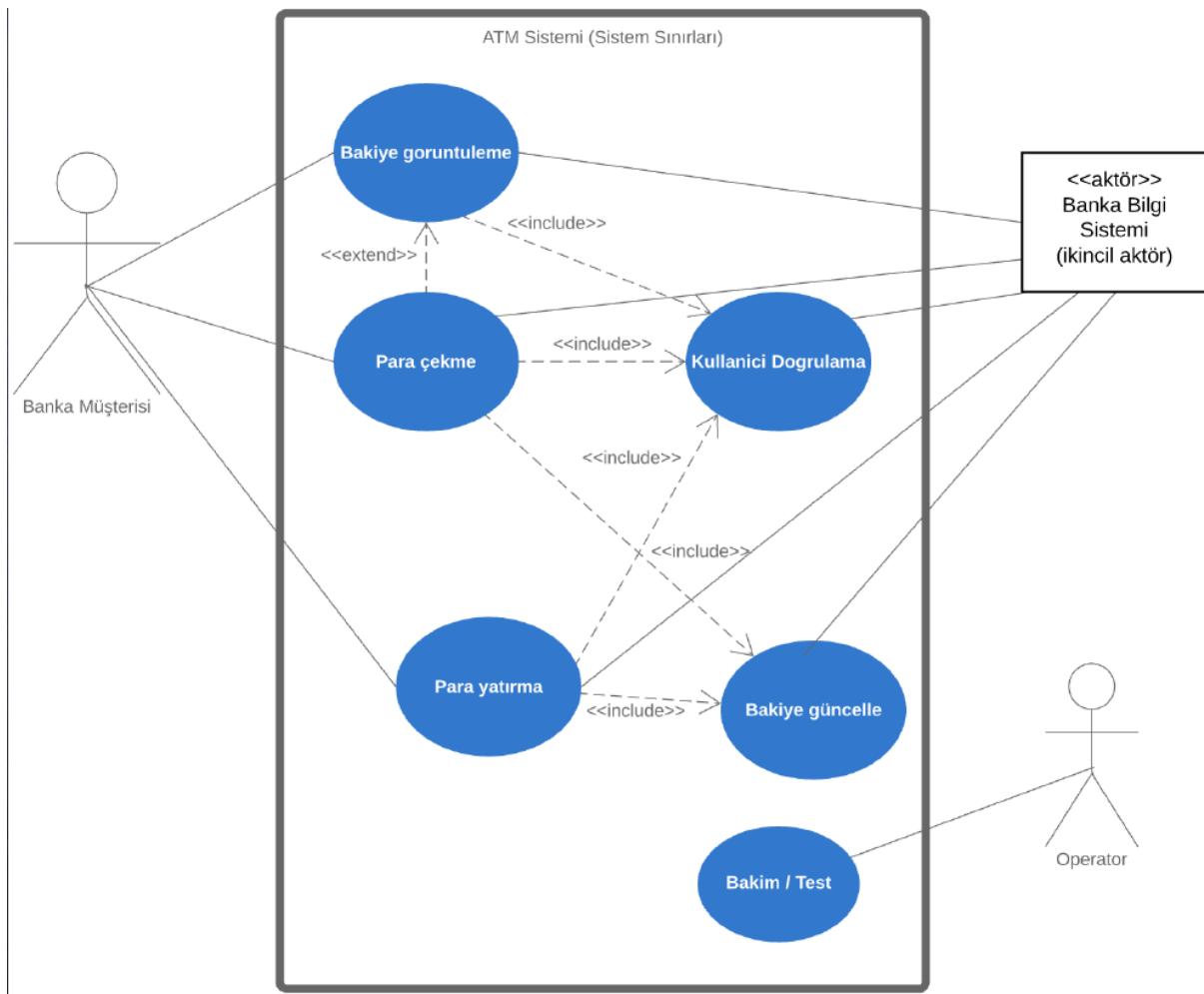
Banka görevlileri ATM sisteminin bakımını (para doldurma, sıkışan/yutulan kartları alma v.b.) yapmalıdır.

Tüm işlemlerde güvenlik gereksinimleri göz önünde bulundurulmalıdır (yetki kontrolü).

Sistem, Banka Bilgi Sistemiyle (kredi kartı servis sağlayıcıları ile) çevrimiçi çalışmalıdır.

ATM İçin Kullanım Durumu (Use Case) Diyagramı

Kullanım durumları "sistem aktör açısından ne yapar" ile ilgilenir. Nasıl yapar ile ilgilenmez.



Communication: Aktör-sistem etkileşimi

Include: Birden fazla kullanım durumu için gerekli işlevler (code reuse)

Extends: istisna olarak ya da ara sıra gerçekleşen kullanım durumlarını ifade eder. Soyutlama adına (karmaşıklığı azaltmak) ayrı olarak verilebilir.

ATM İçin Para Çekme Kullanım Durumu (Use Case) Olay Akışı (Ana senaryo-başarılı)

1. ATM sistemi Ekrana, müşteriden “kartını kart bölmeye takmasını” isteyen mesaj yazdırır.
2. Kart sahibi kartını kart bölmeye takar.
3. ATM sistemi kart doğrulamasını yapar.
4. Ekrana şifre girilmesini isteyen mesaj gönderilir.
5. Tuş takımını kullanarak girilen şifreyi alır.
6. Kullanıcı doğrulama ve yetki kontrolü için banka bilgi sistemine istek gönderilir.
7. Banka sistemi erişim isteğini kabul eder ve gerekli bilgileri gönderir.

8. ATM sistemi ekrana çekilecek tutarları listeler.
9. Kart sahibi çekilecek tutarı girer.
10. ATM sistemi kartı çıkartır.
11. Kart sahibi kartı alır.
12. ATM sistemi parayı ve makbuzu verir.
13. Kart sahibi parayı ve makbuzu alır.

Not: Kırmızı renkli işlem basamakları Kullanıcı Doğrulama (erişim denetimi) kullanım durumunu ifade eder. Yani 8'den sonrası artık kullanıcının denetimi, authentication'ı geçtiği anlamına gelir.

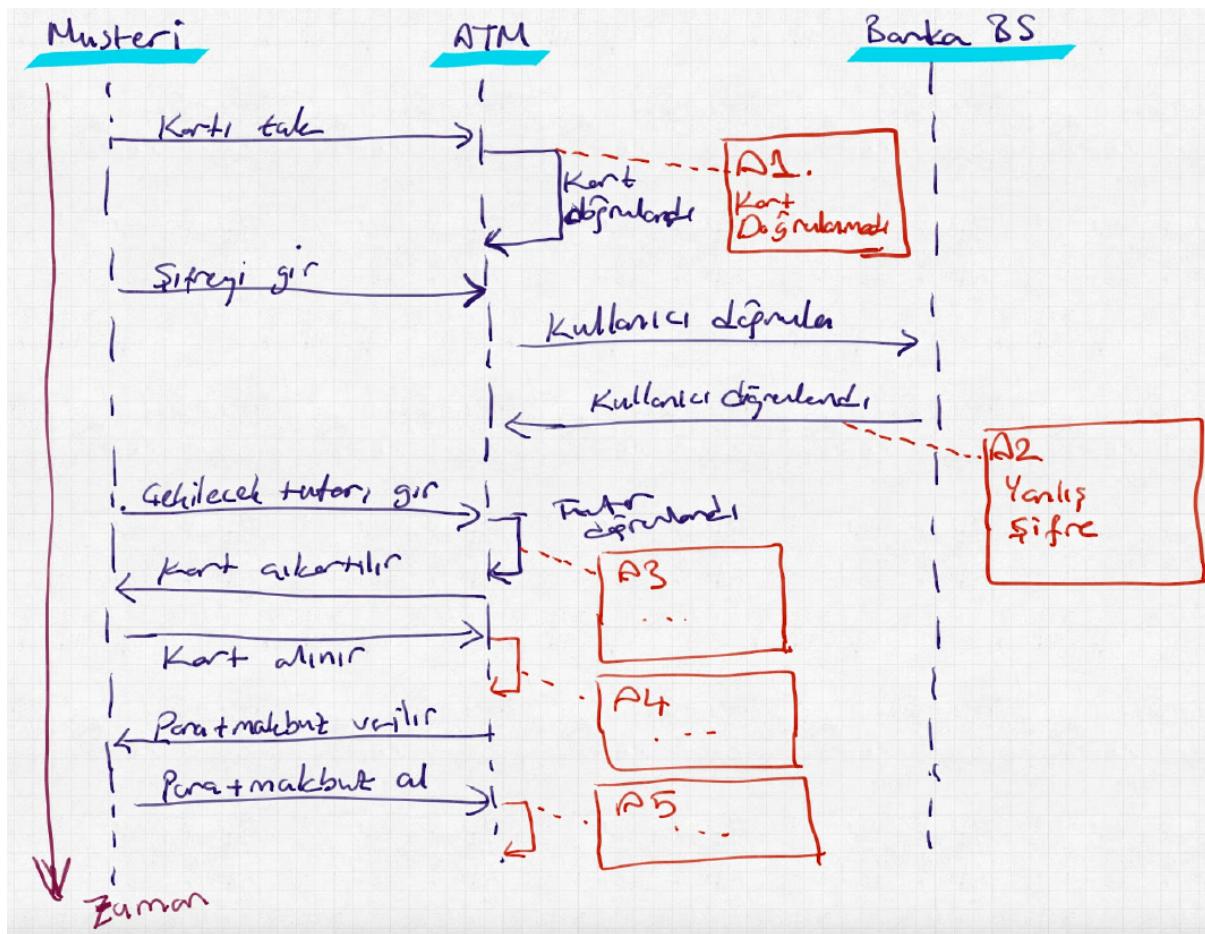
ATM İçin Para Çekme Kullanım Durumu (Use Case) Olay Akışı (Alternatif akışlar)

Parantez içindeki sayılar ana senaryodaki adımlara ifade eder. Kırmızılar ise bu durumda olacak yeni aksiyonları ifade eder.

- A1. Kart doğrulanamadı (3)
 4. Kart bölmesi kartı çıkartır.
 5. İşlem sonlandırılır.
- A2. Yanlış şifre (6)
 7. 3 kez yanlış girilmiş ise kart yutulur.
 8. İşlem sonlandırılır.
- A3. Yeterli para yok (9)
 10. ATM de yeterli bakiye yoksa yeni tutar girmesi istenir.
 11. Hesapta yeterli bakiye yoksa yeni tutar girmesi istenir.
- A4. Banka kartı alınmadı(10)
 11. 30 sn. boyunca alınmadı ise kart yutulur
 12. Gerekli bilgilendirme yapılır
 13. İşlem sonlandırılır
- A5. Para alınmadı (12)
 13. 30 sn. boyunca alınmadı ise para yutulur
 14. Gönderilen ve yutulan para karşılaştırılır.
 15. Gerekli ise hesap güncellenmesi yapılır
 16. Gerekli bilgilendirme yapılır
 17. İşlem sonlandırılır

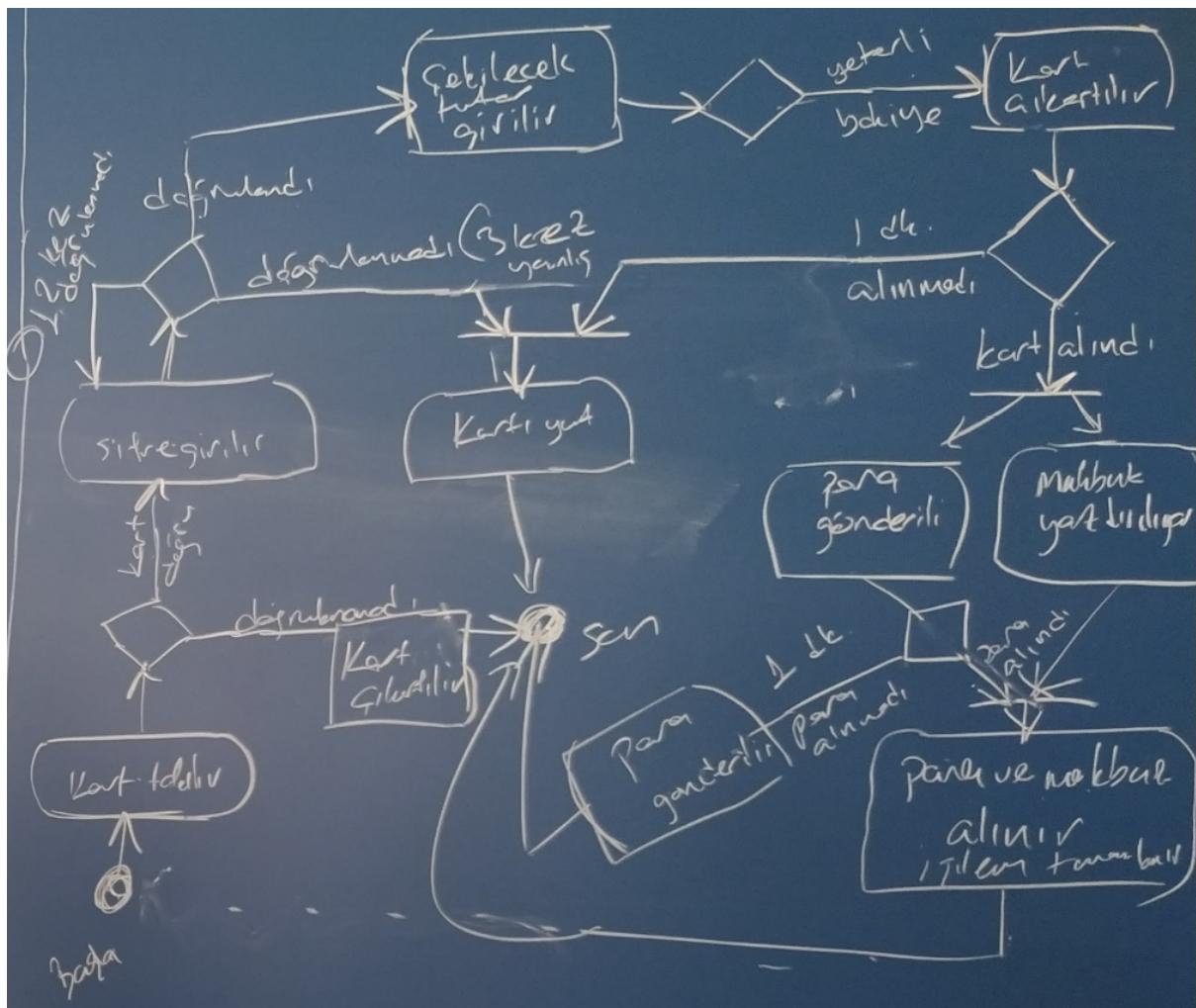
ATM İçin Para Çekme Sıralama Şeması(Sequence Diagram)

Nesneler arası işbirliğinin zamana göre anlatımı/anlaşılması gereğinde oluşturulabilir.

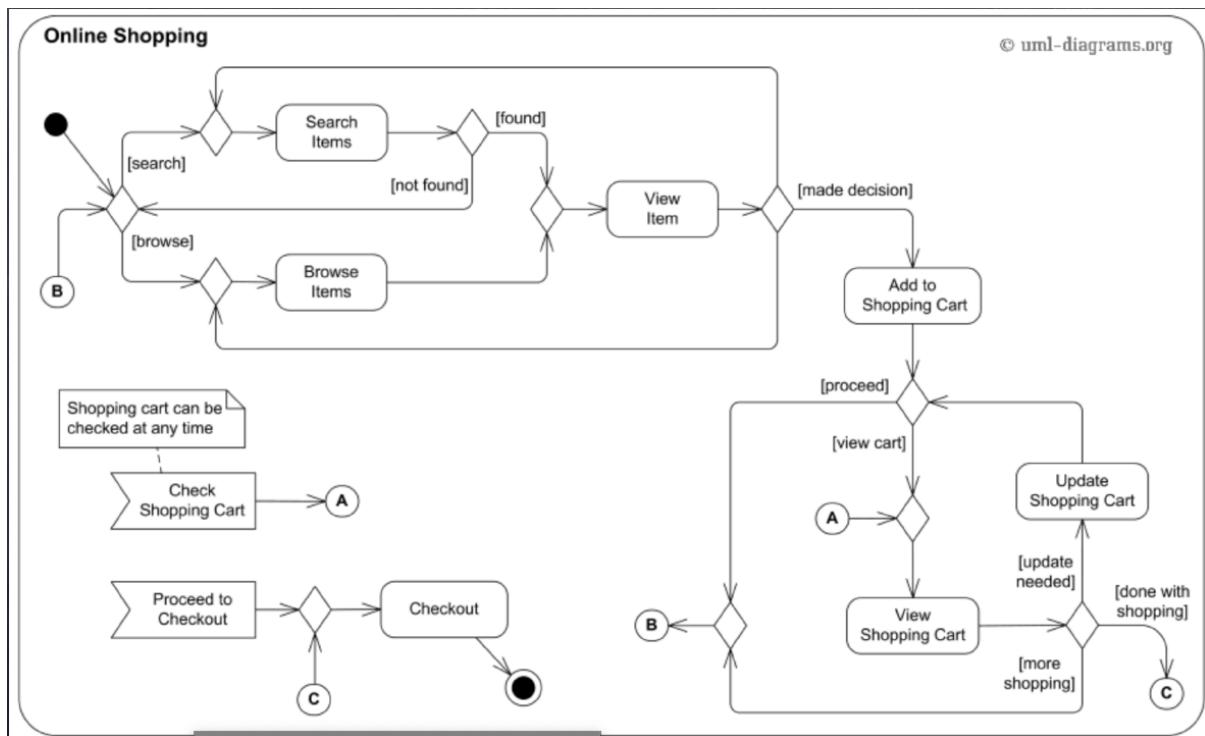


ATM İçin Para Çekme Etkinlik Şeması (Activitiy Diagram)

Ana akışla birlikte alternatif akışlarında gösterir. Akış şemalarına benzediği için anlaşılması daha kolay gelebilir.



Başka bir örnek;



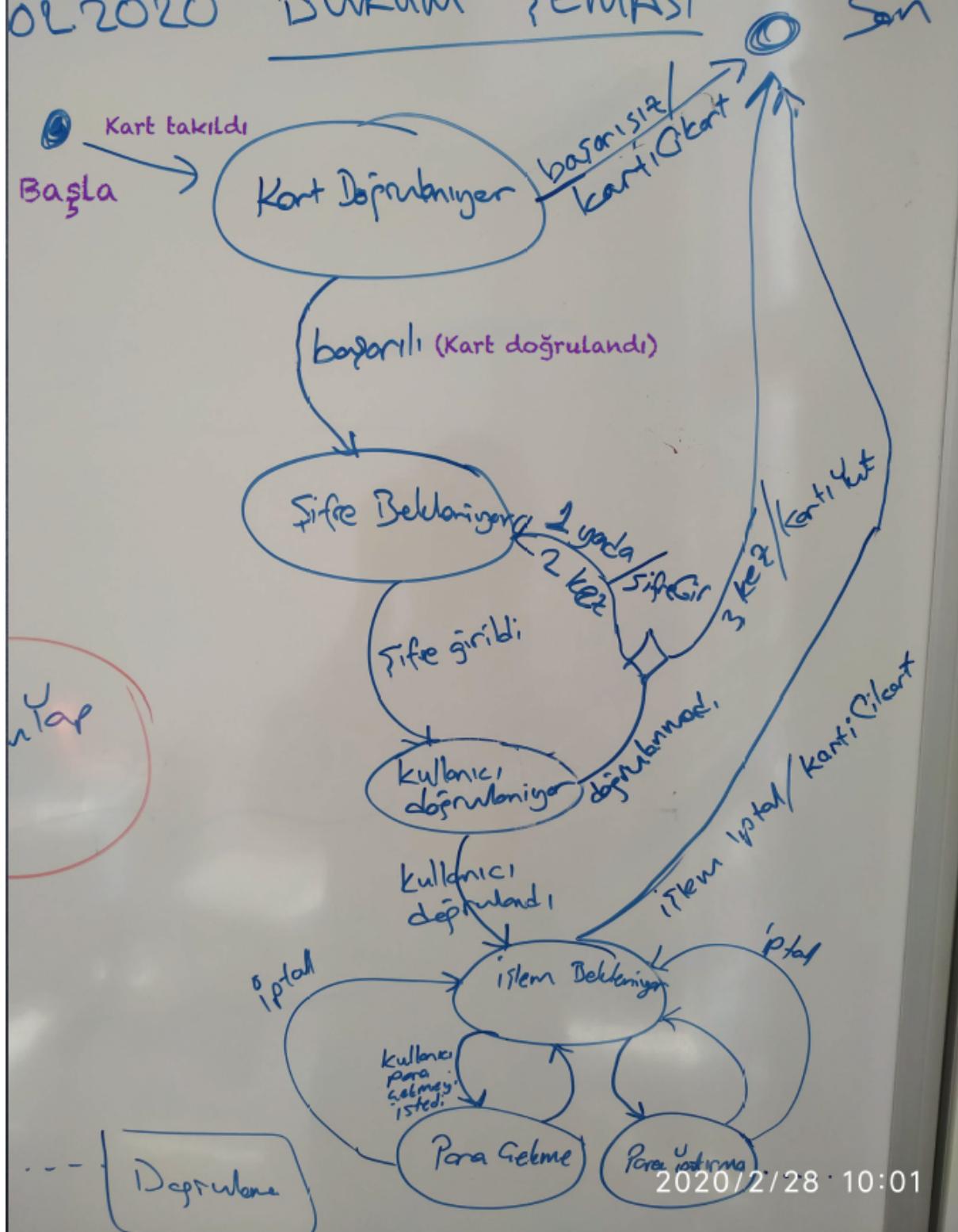
ATM İçin Durum Makinası (State Machine) Diyagramı

Sistemin içinde bulunduğu durumları ve durumlar arası geçişleri modellemek için kullanılır.

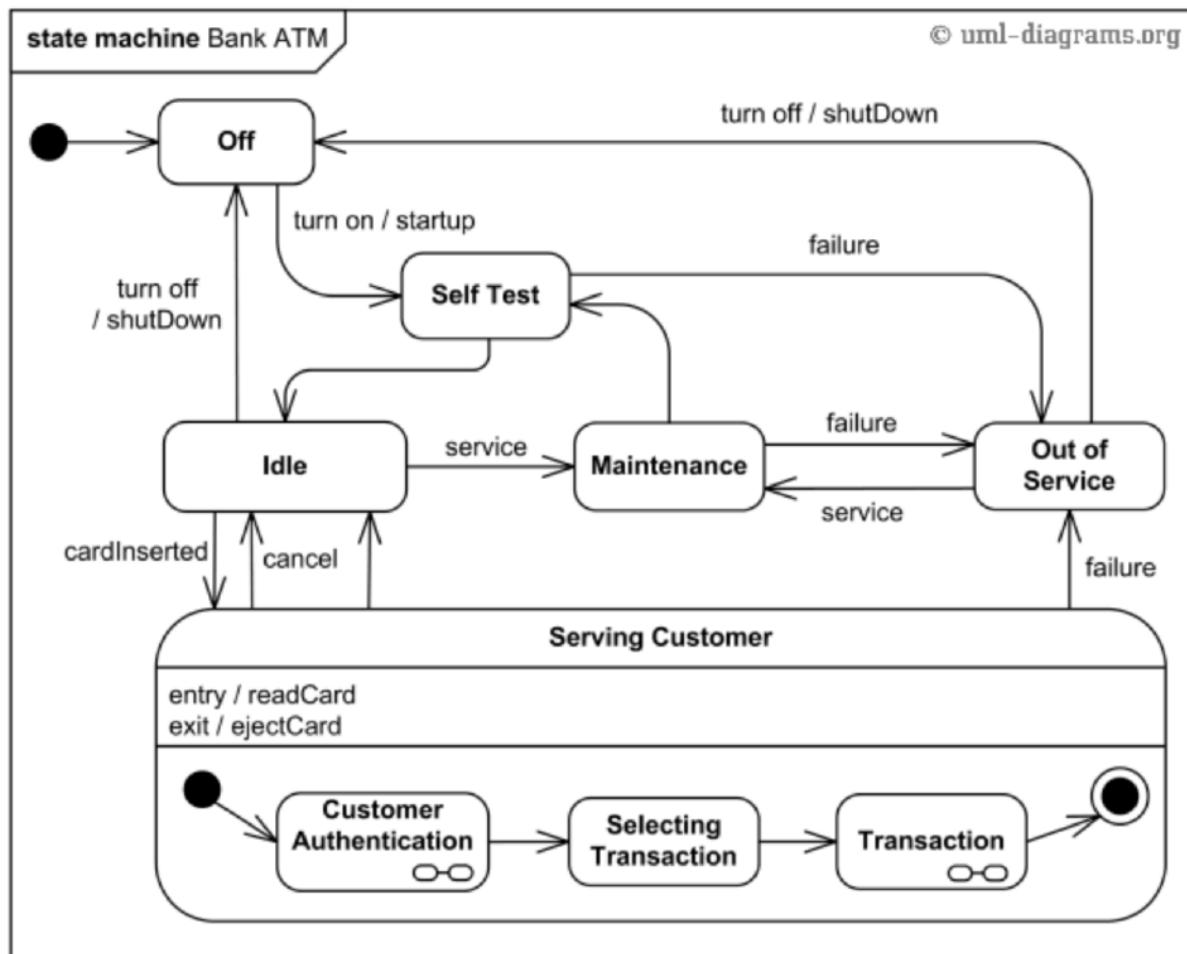
Haberleşme protokollerı, kullanıcı arayüzleri vb. olay güdümlü sistemlerin anlaşılabilirliğini sağlar.

Durum makinesi diyagramında ok üzerinde işlem yazmıyorsa bu varsayılan durumdur. Yani diğer yönlere gitmezse varsayılan olarak o yöne gider.

OL2020 DURUM SEMASI



Başka bir örnek, benzer iş;



Nesnelerin Belirlenmesi

Gerçek hayatı çevremizde gördüğümüz her şey nesnedir. Nesnelerin durum ve davranışları bulunur.

Örn:

İnsan için durumlar; adı, yaşı, boyu v.s. iken davranışlar; öğrenmek, anlamak, uyumak, konuşmak, koşmak v.s.'dir.

Bisiklet için durumlar; rengi, o anki vitesi, hızı, tekerlek sayısı, vites sayısı v.s. iken davranışlar; fren yapması, hızlanması, yavaşlaması, vites değiştirmesi v.s.'dir.

Yazılım nesneleri de durum ve davranışlara sahiptir.

Nesnelerin durumları nitelik(özellik) olarak da adlandırılır ve üye değişkenler ile ifade edilir.

D davranışları ise üye fonksiyonlar/methodlar/yöntemler kullanılarak gerçekleştirilir.

Nesneler belirlenirken, analiz aşamasında oluşturulan gereksinim listesi (kullanım durumları) üzerinde metinsel analiz yapılır (Abbott's tekniği).

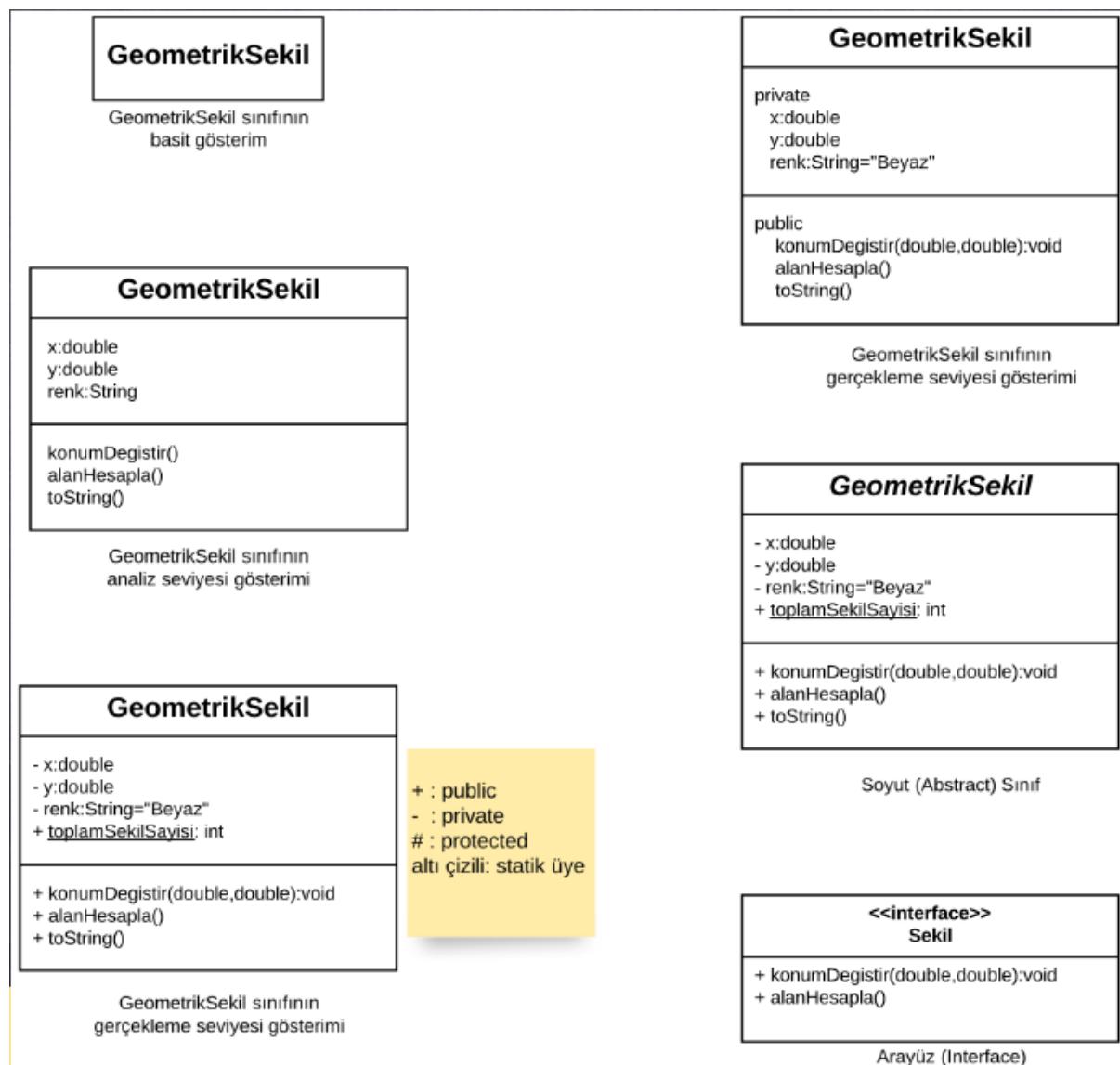
Gereksinim listesindeki **isim ve isim tamlamaları sınıf adaylarıdır.**

Özellikleri ya da **davranışları** bulunan isim ya da isim tamlamaları sınıf olabilir.

Gereksinim listesindeki **eylemler(füller)** nesnelerin **üye yöntemlerini** ifade edebilir.

Sınıflar belirlendikten sonra tasarım ayrıntıları içermeyen **Analiz Sınıf Şeması (kavramsal model)** oluşturulur. Bu şemada çok fazla detay yer almaması gereklidir. Amaca göre değişir. Sınıf diyagramı gösterim şekilleri;

(Soyut sınıflar italik)



Gerçekleme seviyesi gösterimi koda en yakın olandır ve buna bakarak koda dönüşüm yapılır. Örn;

```

package cc.ders6;

public class GeometrikSekil {

    private double x;
    private double y;
    private String renk;
    public static int toplamSekilSayisi;

    public void konumDegistir(double x, double y){
        .....
    }

    .....
}

```

Nesne adayları (Craig Larman)

Fiziksel ya da elle tutulabilir nesneler (ürün, insan, bilgisayar ...)

Yerler (okul , bina, yerleşme, sınıf, oda ...)

İşlemler (transactions) (para çekme..., kayıt yaptırmaya...)

Roller (yönetici, öğrenci, personel, kayıtlı kullanıcı v.b.)

Harici sistemler (Veritabanları, banka bilgi sistemi, web servisleri...)

Kuruluşlar (okul, işyeri, firma...)

Olaylar (ActionListener, ActionEvent, KeyListener, KeyEvent...)

Loglar ...

Veri Toplulukları

İçerisinde çok sayıda element tutulan yapılardır. Bu çoklu elementleri yapıda tutmak ve sonrasında bunlar üzerinde toplu işlemler yapabilmek adına burada elementleri tutarız. Örneğin diziler bunun 1 örneğidir. Fakat bu konuda dizilerden çok soyut veri tiplerini yani list gibi daha çok kullanacağız. Yani dinamik diziler. Örneğin Java'da List interface'inden gelen ArrayList, Vector, LinkedList bunlara örnektir. Daha çok ArrayList kullanılacak.

Örneğin Kitap adında bir class var ve sıradan bir dizi içerisinde adı ve fiyatını tutan değişkenlere sahip. Biz bu sınıftan bir dizi oluşturabilir ve bu Kitap'tan oluşturulacak nesneleri bu dizide saklayabiliriz.

```

Kitap[] kitaplarStatikDizi=new Kitap[2];
kitaplarStatikDizi[0]=new Kitap("NYP",110.00);

```

```
kitaplarStatikDizi[1]=new Kitap("Yapay Zeka",125.00);
```

Örneğin ArrayList kullanımı;

```
List<Kitap> kitaplarAL = new ArrayList<Kitap>();
kitaplarAL.add(new Kitap("AL Veritabanı Yönetim Sistemleri",100.00));
kitaplarAL.add(new Kitap("AL Nesne Yönelimli Programlama",125.00));
kitaplarAL.add(2,new Kitap("AL Bilgisayar Ağları",150.00));
```

Görüleceği üzere List interface'inden ArrayList türetiliyor ve Kitap tür olarak içerisinde atılıyor, nesnesin adı da kitaplarAL olarak belirlenmiş. Sonrasında içerisinde bulunan add fonksiyonu ile bu dinamik listeye elemanlar ekleyebiliyoruz. Son eklenen Bilgisayar Ağları kitabıının kodunda başta bulunan 2, kitaplarAL ArrayList'inin kaçinci indeksine eklemek istedigimizdir.

Normal diziden farklı olarak ArrayList'de elemanlara erişirken;

```
for (int i=0; i<kitaplarAL.size();i++)
    System.out.println(kitaplarAL.get(i).getAdi());

//foreach
for(Kitap kitap:kitaplarAL)
    System.out.println(kitap.getAdi());

System.out.println("***** Lambda Expressions *****");
kitaplarAL.forEach(kitap -> System.out.println(kitap));
```

Döngü ile erişirken .get(indeks) ile erişim yapılıyor.

Ek olarak görüleceği üzere lambda expressionslar ile de kolayca erişim sağlanabilir.

Örneğin LinkedList kullanımı;

Kod yazımı açısından vs. ArrayList ile hiçbir fark yoktur. Tek fark, şu şekilde tanımlanır;

```
List<Kitap> kitaplarLL = new LinkedList<Kitap>();
```

Örneğin Vector kullanımı;

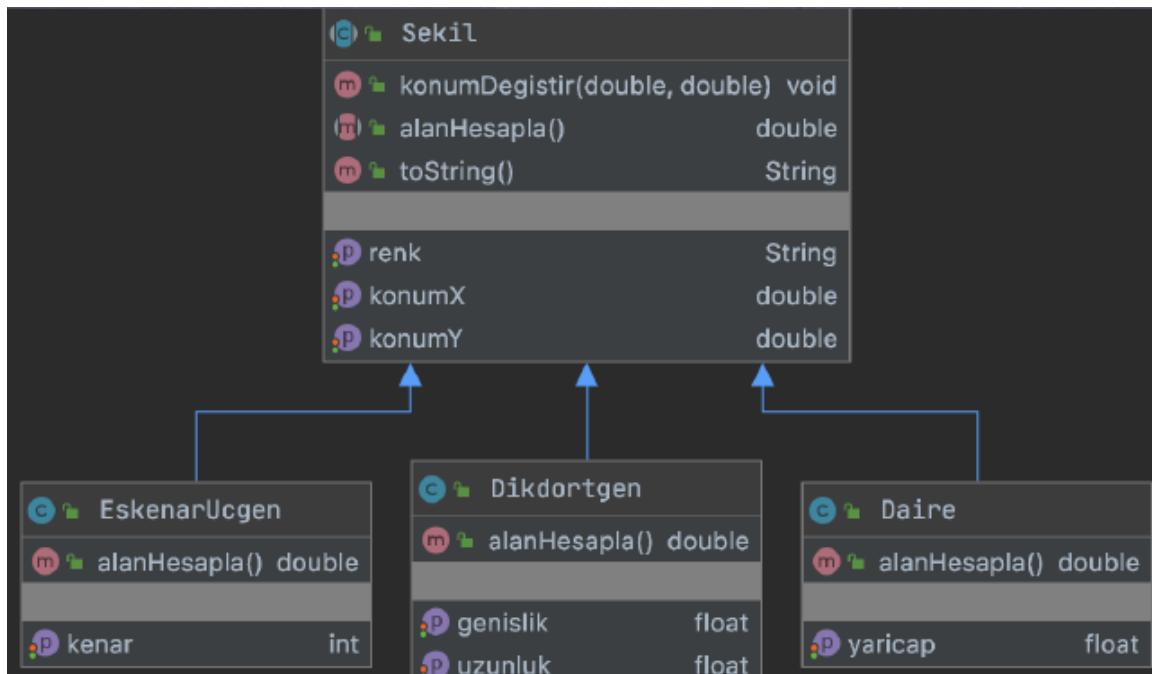
Kod yazımı açısından vs. ArrayList ile hiçbir fark yoktur. Tek fark, şu şekilde tanımlanır;

```
List<Kitap> kitaplarV = new Vector<Kitap>();
```

Elementler arası bağıntılar

Kalıtım Bağıntısı

Sınıf diyagramında kalıtımın gösterim şekli;



İçin boş üçgenle kalıtım aldığı(base) sınıfı gösterir.

Türetilmiş(derived) sınıfların doğruluğunu sorgulamak için == yerine kalıtım için "is a" ve "is kind of" ifadeleri kullanılır. Örn;

Daire is a Sekil

Daire is kind of Sekil

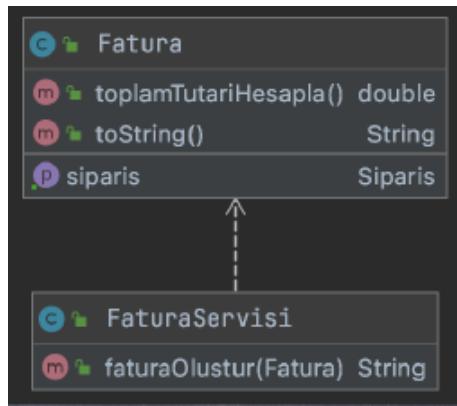
Dependency Bağıntısı

Buna Dependency Injection denir.

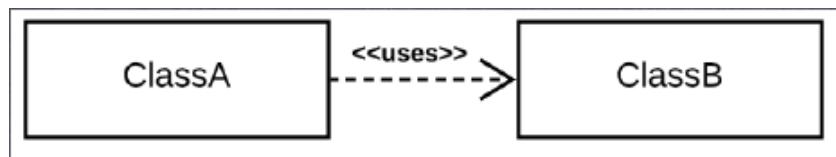
Bu bağıntı oluşturacağınız bir sınıf içerisinde başka bir sınıfın nesnesini kullanacaksanız new anahtar sözcüğüyle oluşturmamanız gerektiğini söyleyen bir yaklaşımdır. Gereken nesnenin ya constructor'dan ya da Setter metoduyla parametre olarak alınması gerektiğini vurgulamaktadır.

Bir elementin (istemci- ClassA) diğer elementi (tedarikçi-ClassB) kullanımını ifade eder.

Örneğin Fatura Servisi sınıfının diğer elementi olan Fatura kullanması dependency bağıntısına örnektir. Yani fatura servisi fatura oluşturmak için Fatura'nın nesnesine ihtiyaç duyar. Fatura Servisi istemci sınıf-kod, Fatura ise tedarikçi koddur.



Sınıf diyagramında dependency gösterim şekli;



Dependency doğruluğunu sorgulamak için == yerine kalıtım için "uses" ifadesi kullanılır.
Örn;

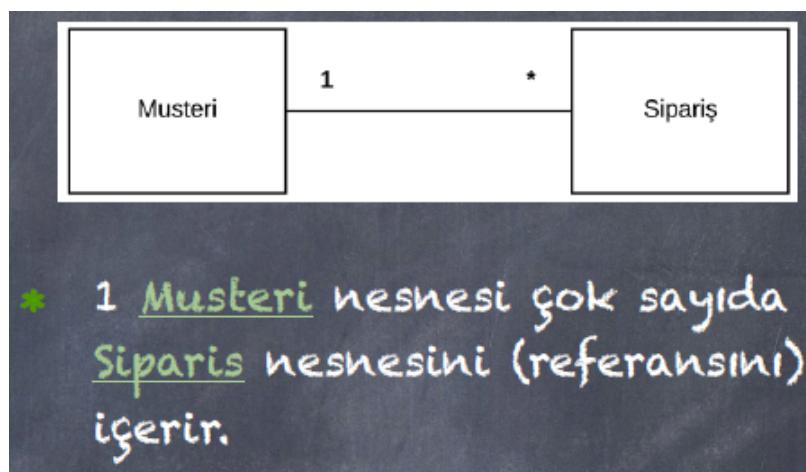
ClassA uses **ClassB**

Association Bağıntısı

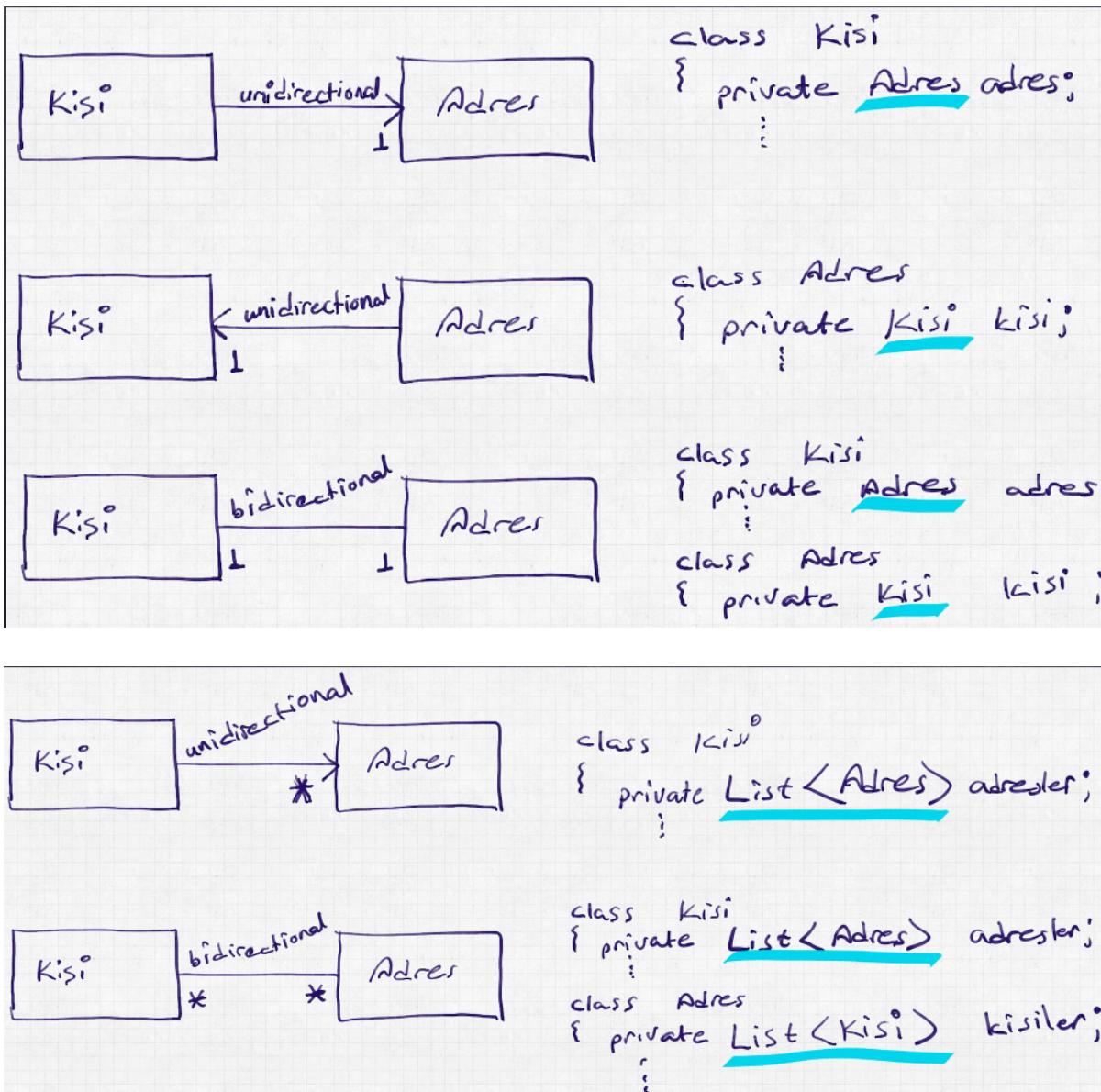
Bir nesne diğer nesneyi üye (özellik) olarak içerir.

Yani üye fonksiyonda kullanıyorsa Dependency, üye özellik olarak kullanıyorsa Association oluyor.

Sınıf diyagramında association gösterim şekli;



"Bidirectional" (iki yönlü) ve "unidirectional" (tek yönlü) olabilir. Bidirectional olduğunda ok işaretini kullanılmaz. Nesneler 1 adete sahip olursa örneğin `private Adres adres;`, 1(gösterim 1)'i, 1 nesneden fazla nesneye sahip olacaksa yani veri topluluklarını kullanacaksa çok (gösterim *) olur.



Göründüğü üzere 1. için **Kisi** sınıfı **Adres** sınıfından üye nesneye sahip olduğu fakat **Adres** sınıfı **Kisi** sınıfından üye nesneye sahip olmadığından unidirectional fakat 3. örnekte 2'si de sahip olduğu için bidirectional.

Association doğruluğunu sorgulamak için == yerine kalıtım için "has a" ifadesi kullanılır.
Örn;

Musteri has a Siparis

"Aggregation" ve "Composition" bağıntıları "Association" bağıntısının özel halleridir.

Composition Bağıntısı

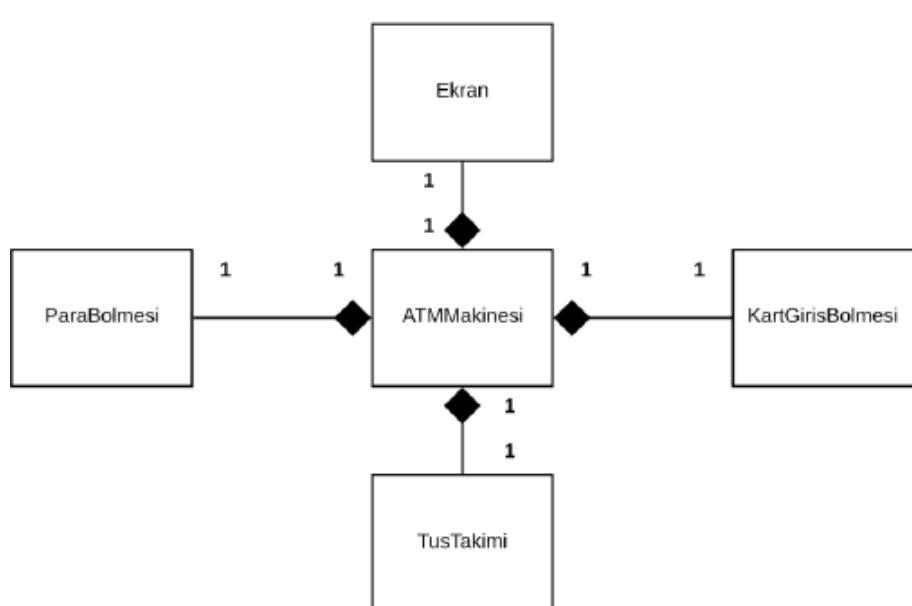
Bütün parça bağıntısı (whole/part)

Parça nesneler bütün içerisinde yer alırlar. Parçaları bütün oluşturur ve yok eder.

Her parça en çok 1 bütün içerisinde bulunur. Bir bütünde çok sayıda parça yer alabilir.

Aralarındaki bağıntının çok güçlü yani parçaların bütüne göre var olup yokmasına bağlıysa composition bağıntısı olur.

Sınıf diyagramında composition gösterim şekli;



```
public class ATMMakinesi
{
    private Ekran ekran;
    private TusTakimi tusTakimi;
    private ParaBolmesi paraBolmesi;
    private KartGirisBolmesi kartBolmesi;

    public ATM() {
        ekran=new Ekran();
        tusTakimi=new TusTakimi();
        paraBolmesi=new ParaBolmesi();
        kartBolmesi=new KartGirisBolmesi();
    }
}
```

İçi dolu elmas şeklinde ifade edilir.

**Nesne genelde bu örnekteki gibi kurucu içerisinde oluşturulur.

Composition doğruluğunu sorgulamak için == yerine kalıtım için "is part of" ifadesi kullanılır. Örn;

Ecran is part of ATMMakinesi

Aggregation Bağıntısı

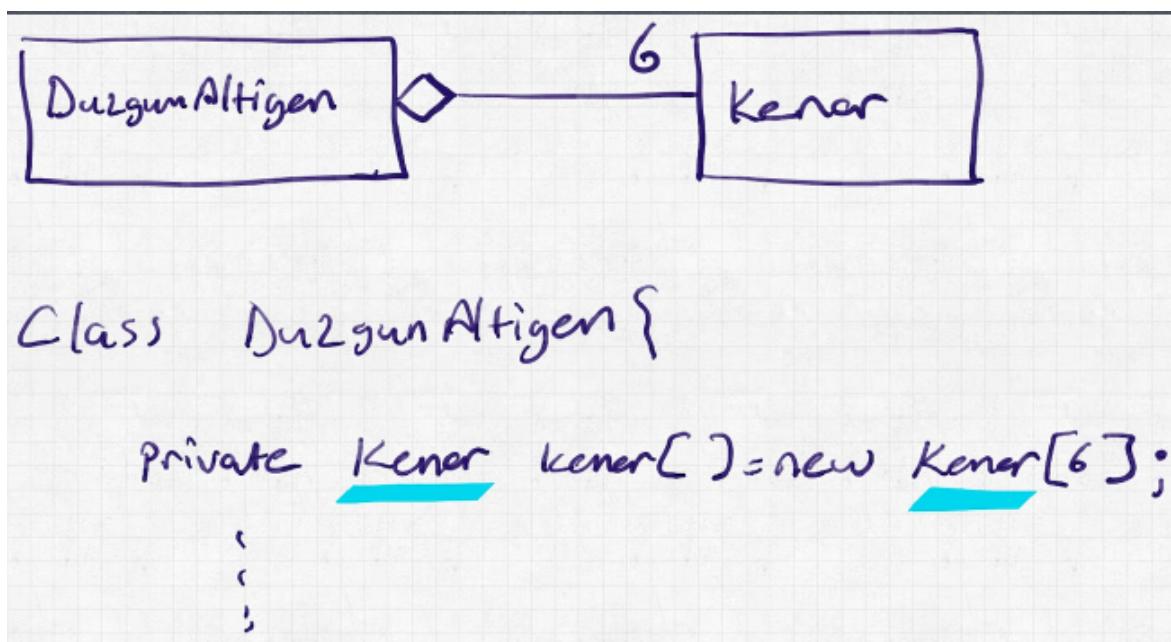
Bütün parça bağıntısı (whole/part)

Parça nesneler bütün içerisinde yer alırlar. Parçalar başka bütün içerisinde de kullanılabilir.

Bütün yok edildiğinde parçanın yok edilmesi gerekmeyebilir. (başka bütün içerisinde kullanılıyor olabilir)

Aralarındaki bağıntının çok güçlü olmadığı yani parçaların bütüne göre var olup yokmasına bağlı değilse aggregation bağıntısı olur.

Sınıf diyagramında aggregation gösterim şekli;



İçin boş elmas elmas şeklinde ifade edilir.

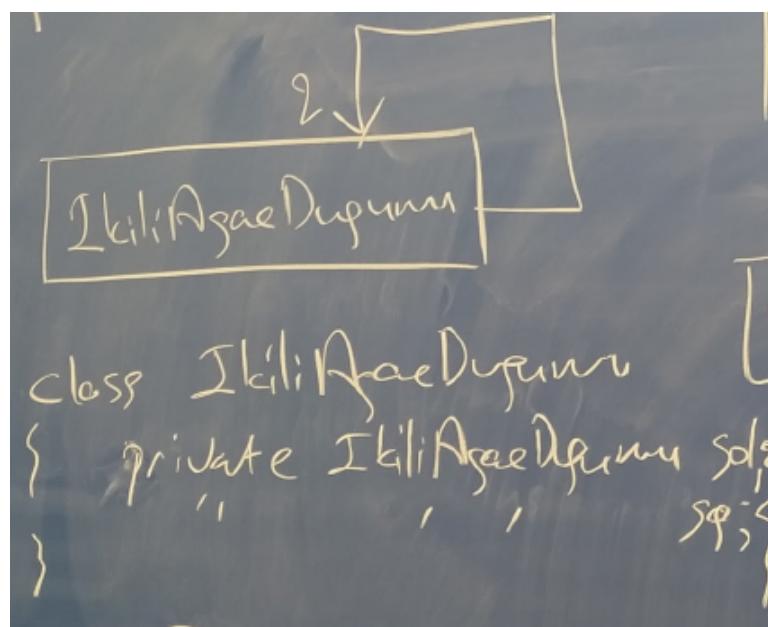
Aggregation doğruluğunu sorgulamak için == yerine kalıtım için "consists of" ifadesi kullanılır. Örn;

DuzgunAltigen consists of Kenar

Reflexive Bağıntısı

Nesnenin kendisini içерdiği durumdur. Yani recursive yapılar. Örneğin ağaç yapılarında düğüm 2 çocuğa sahip fakat çocukların her biri de yine düğümdür.

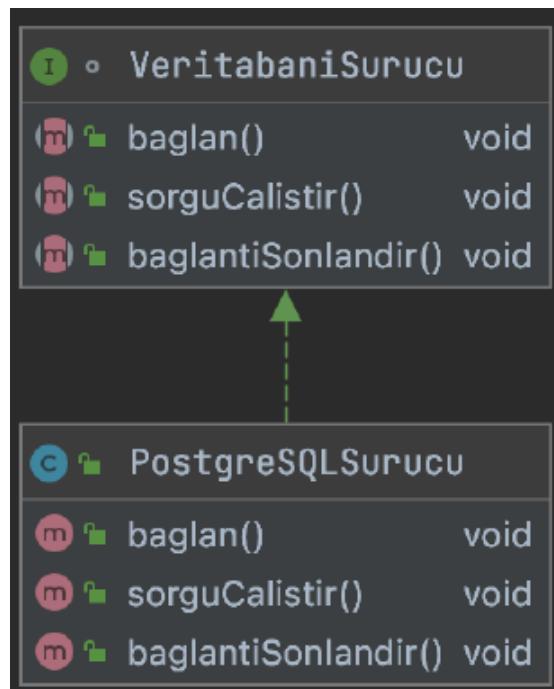
Sınıf diyagramında reflexive gösterim şekli;



Buradaki oktaki "2" ifadesi kendi içerisinde 2 üyesinin bulunduğu manasına gelir.

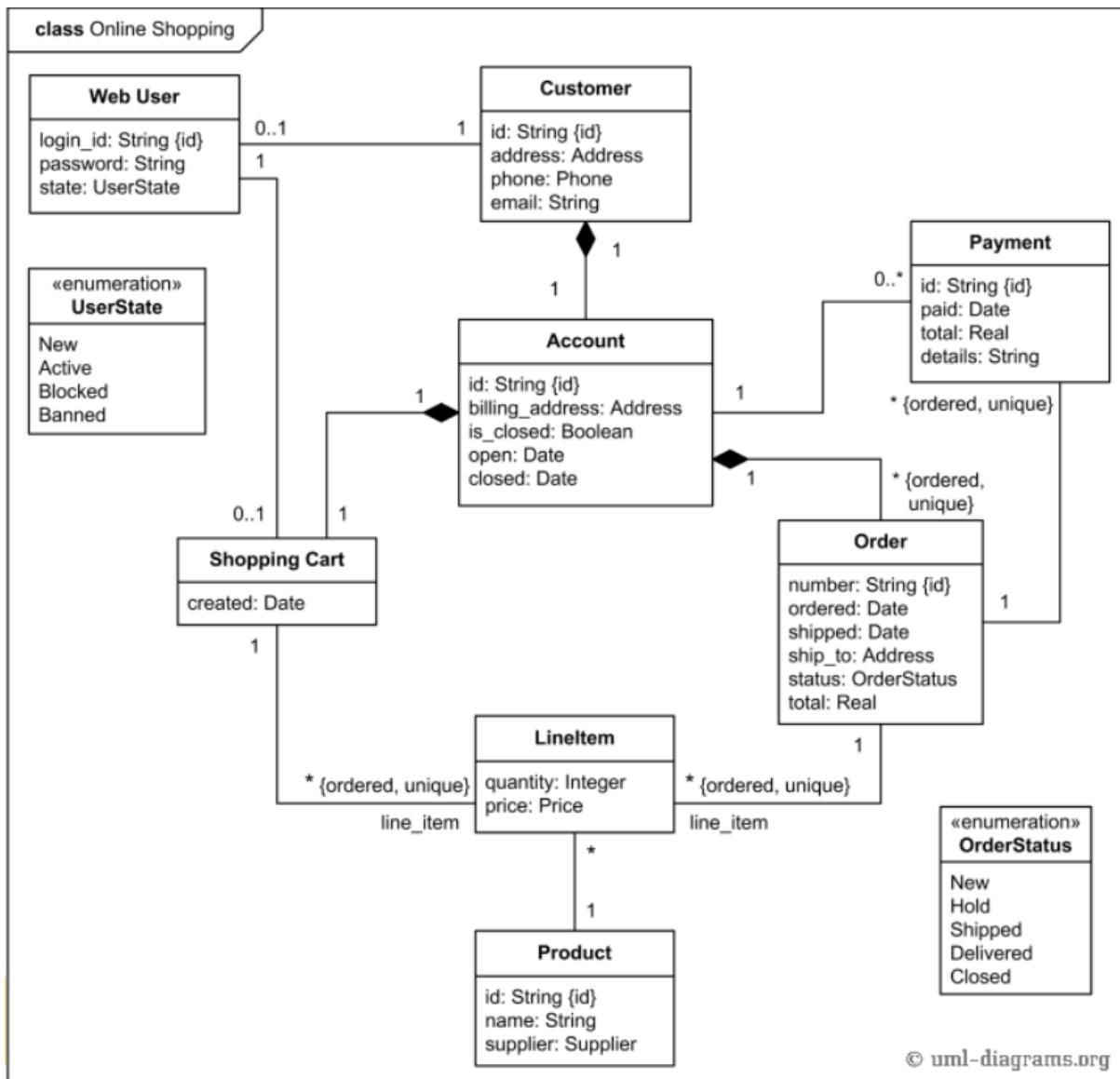
Interface Bağıntısı

Sınıf diyagramında interface gösterim şekli;



Kalitimdan farklı olarak kesikli çizgi ile ifade edilir.

****Sınıf Şemasına bir örnek;



Enumeration'lar sınıf benzeri yapıda olan fakat fonksiyon içermeyen, yalnızca durum, üye içeren yapılardır. Sınıf şemalarında yalnızca sınıflar birbirine bağlanırlar. Bundan dolayı enumlar bağlı değillerdir.

0..1 ifadesi hiç olmayabilir veya 1 anlamına gelir.

Class Responsibility Collaboration(CRC) Kartları

Nesnelerin sorumluluklarını ve bu sorumlulukları yerine getirmek için hangi nesnelerle işbirliği içerisinde olduğunu gösterir.

Sistem içerisindeki tüm nesneler için CRC kartı oluşturulabilir.

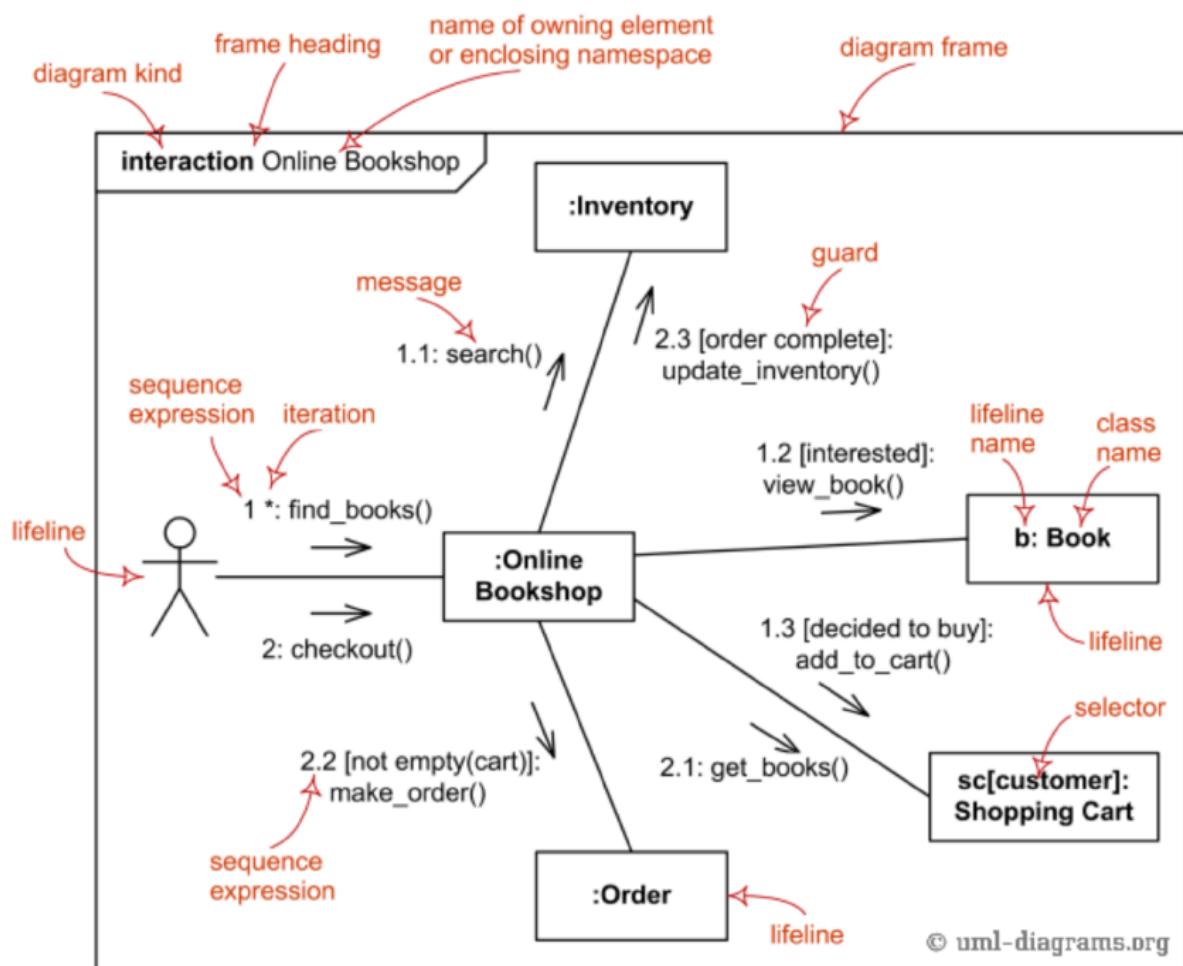
Nesne geliştirecek olanların, hangi servislere ihtiyaç olduğunu anlamasını kolaylaştırır. Sağlanacak servisler için hangi nesnelere ve ilgili servislerine ihtiyaç duyulduğu anlaşılır.

Örn ATM uygulamasında ATM'nin CRC kartı.

ATM	
Sorunluweise	İşbirliği Sunumları
baska	kendisi
KartDagindan Kullanicidaginda	Kart Dolmesi BankaSS, Tuz Takimi (Veri Dizi) Elanın (nesnelerin) Görsütle
İşlenen	Linear Perakende, Parakotirm Belirli Görsütlere

Haberleşme Şeması

Örn kitap satışı;



Sistem Tasarımı

Analiz aşaması uygulama alanına (application domain) odaklanır. Uygulama alanı gerçekleştirilecek yazılımın çalışacağı ortamı ifade eder.

Tasarım aşaması ise Çözüm alanına (solution domain) odaklanır.

Tasarım aşaması, sistem tasarımları ve nesne tasarımları olarak iki bölüme ayrılabilir.

Sistem tasarımı kapsamında sistem alt bileşenlerine ayrılır. Sistemle ilgili tasarım hedefleri belirlenir ve kısıtlar ihlal edilmeden bu hedeflere ulaşmak için gerekli mimari oluşturulur.

Analiz aşaması:

gereksinimler, kısıtlar belirlenir

kullanım durumları tanımlanır

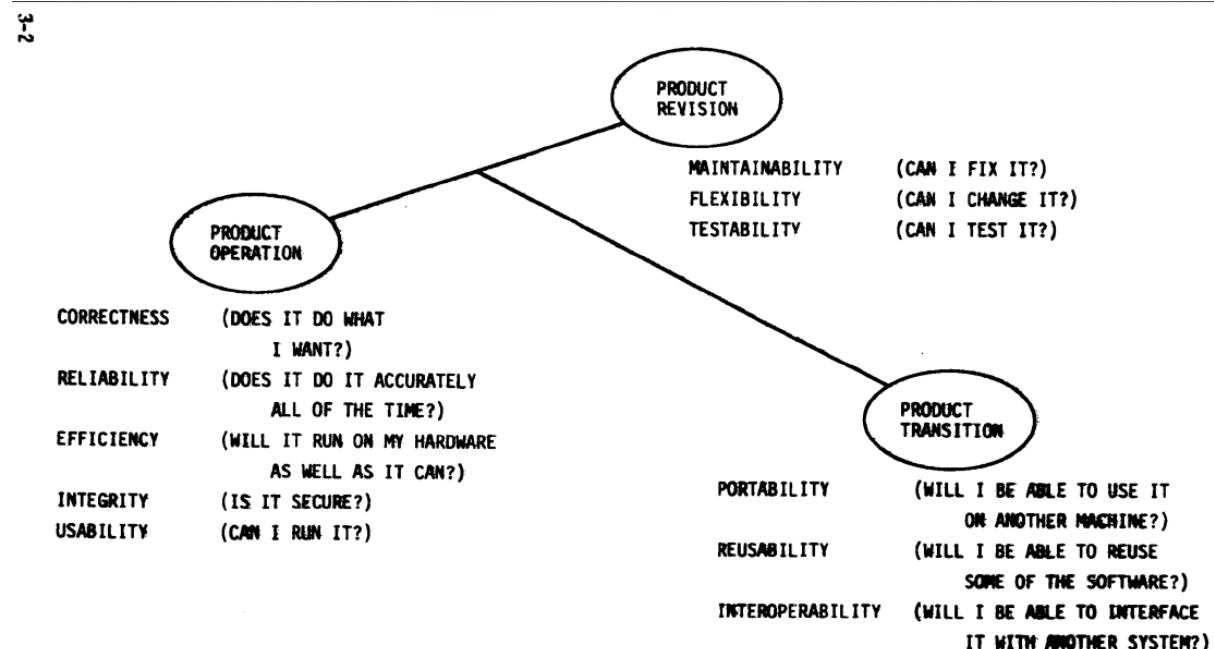
analiz sınıf şeması/nesne modeli, dinamik modeller (sıralama şeması, etkinlik şeması vb.) oluşturulur

Sistem Tasarım Aşaması:

tasarım hedefleri (kalite ölçütleri) belirlenir
 sistem alt bileşenlerine ayırtılırlar
 tasarım hedeflerine ulaşmayı sağlayacak yazılım mimarisi (mimari stiller), veri modelleri, erişim denetimi yöntemleri vb. belirlenir.

1. Tasarım Hedeflerinin Belirlenmesi / Yazılımların Kalitesi (McCall Kalite Üçgeni)

Bir yazılımin kalitesine etki eden faktörler üç sınıfa ayrılır.



Kalite Ölçütlerinin Birbirlerine Olumsuz Etkileri

Integrity ↔ Efficiency

Güvenlik denetimi ek kontrol ve fazladan yer anlamına gelir
 bankacılık uygulaması

Portability ↔ Efficiency

tüm platformları desteklemesi için eklenen denetimler fazladan kaynak kullanımı ve gecikmealamına gelir

Reusability (Flexibility, Maintainability) ↔ Efficiency

program daha küçük parçalar halinde yazılırsa hız azalır.
 gömülü sistemler, gerçek zamanlı sistemler...

2. Sistemin alt bileşenlerine ayırtılıması

Karmaşıklığı azaltmak için yazılım sistemi **alt bileşenlerine** (alt sistem) ayrırlar.

Alt Sistemler, birbirleriyle yakından ilişkili; sınıflar, ilişkiler ve kısıtlardan oluşur.

Alt sistemler sayesinde;

problem çözümü kolaylaşır,
 çok sayıda kişinin aynı projede aynı anda çalışabilmesi sağlanır,
 sistemin anlaşılması, anlatılması kolaylaşır,
 bakım kolaylaşır
 modülerlik, kod tekrar kullanımı artar...

Analiz aşamasındaki alt bileşenler

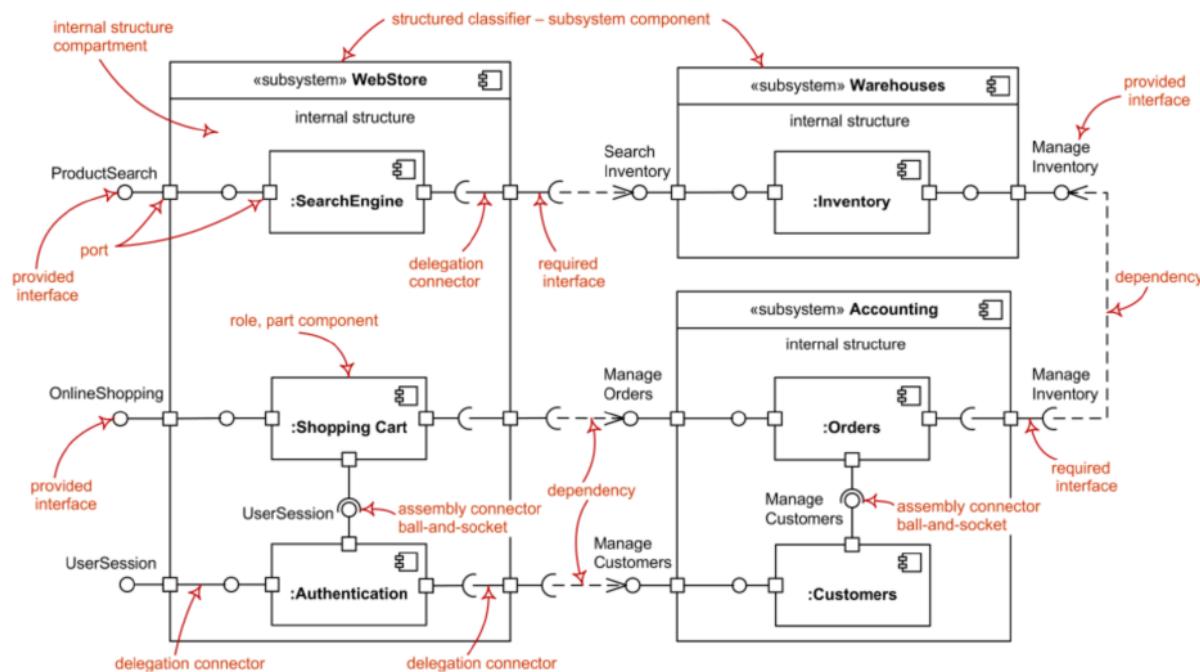
sınıf

paket (UML paket şeması)

Tasarım aşamasındaki alt bileşenler için “UML Component” şeması kullanılabilir.

UML Component Diagramı

Yapısal gösterim şekillerindendir. “Component” (Bileşen), sistemin diğer bölmeleriyle haberleşmeyi sağlayan arayzlere (API-Application Programming Interface) sahip alt sistemler ya da sistemlerdir. Ortak amaca hizmet eden sınıfların bir araya getirilmesiyle oluşur.



Şekilde bağıntılı üç alt sistem (WebStore(Web mağazası), Warehouse(depo, ambar), Accounting(Muhasebe)) bulunmaktadır. Alt sistemler içerisinde başka alt sistemlerde bulunmaktadır.

“SearchEngine” sağladığı “ProductSearch” arayüzüyle ürünlerin aranmasını sağlar. Bu işlem için “Inventory” bileşeninin sağladığı “SearchInventory” arayüzü kullanır.

“Shopping Cart” bileşeni çevrimiçi alış-veriş arayüzü sağlar. Alış-veriş için kullanıcının oturum açması gereklidir. Siparişlerin yönetimi için “Orders” bileşeninin “Manage Orders” arayüzü kullanır.

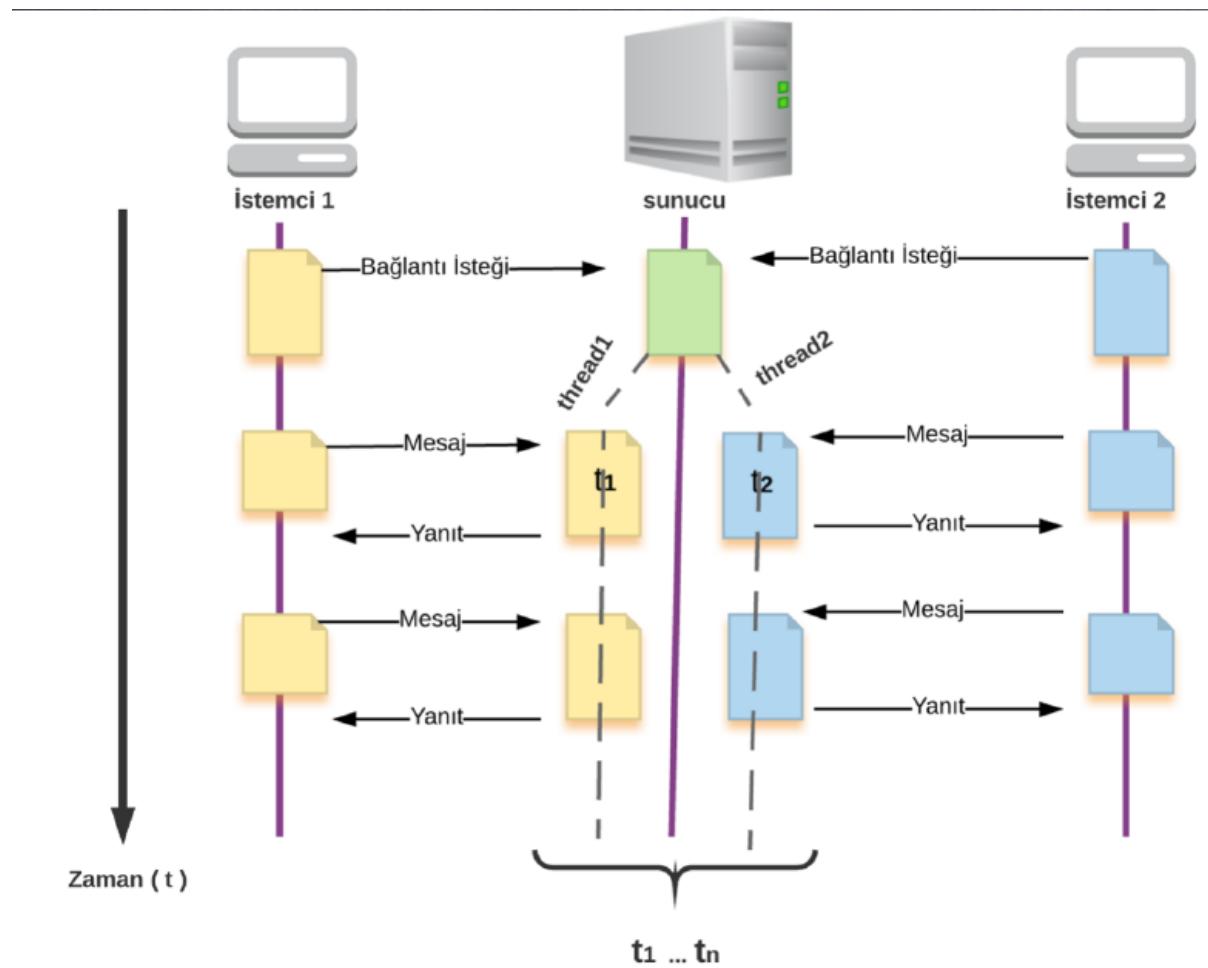
İyi bir tasarım için;



3. Yazılım mimarileri (mimari stiller)

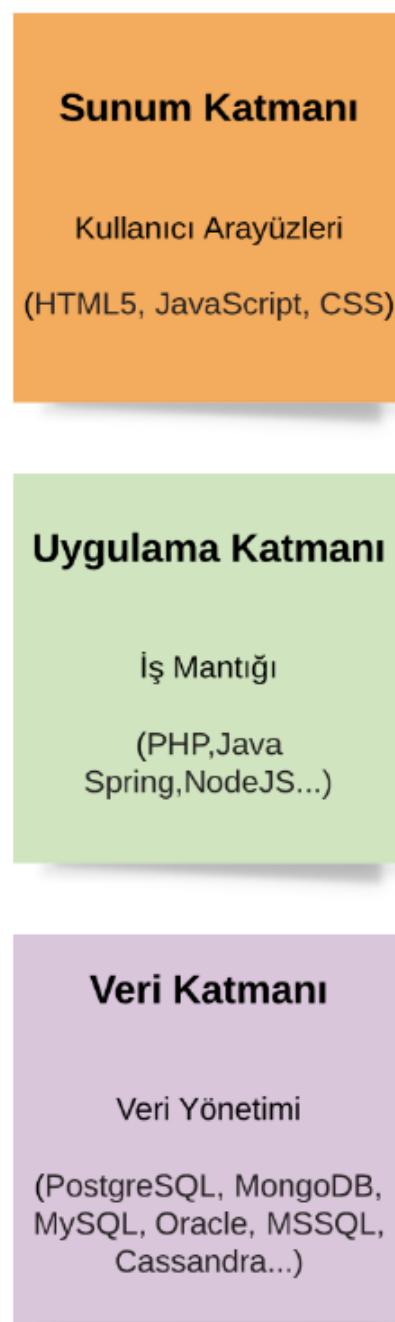
1. İstemci/Sunucu (Client/Server)

Sunucu istek alarak birden fazla client'a multithreading özelliği ile thread döndürür. Yani istemci ister, sunucu yanıt döndürür.



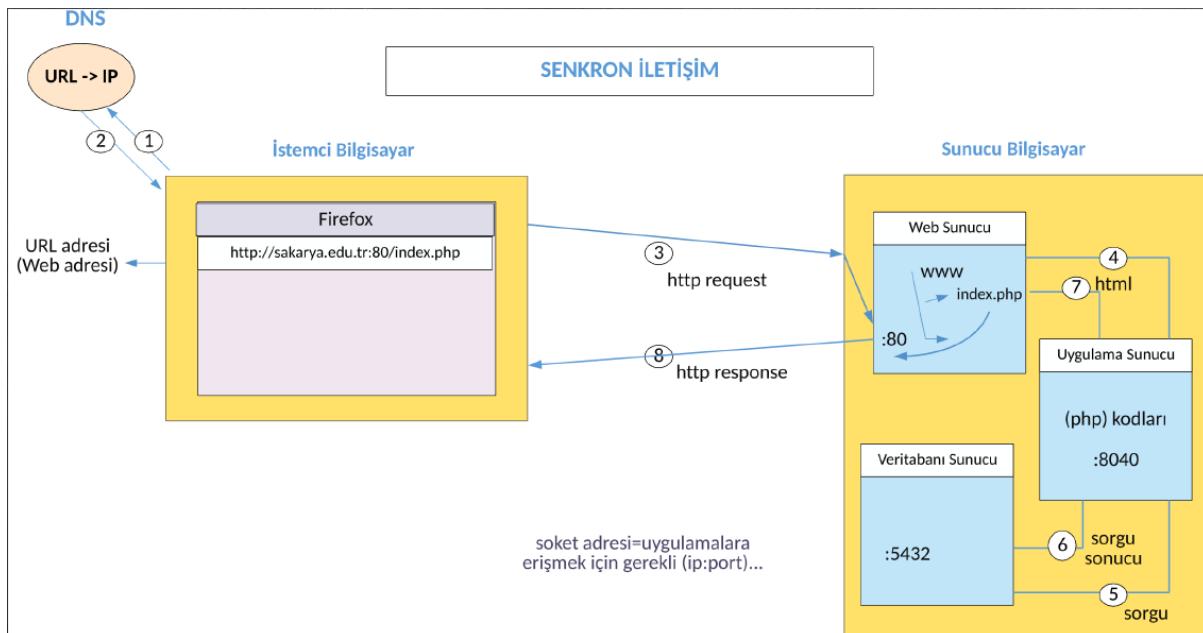
2. Üç Katmanlı Web Mimarisi (Three-tier Web Architecture)

Sunum, iş mantığı ve veri yönetimi fonksiyonlarının fiziksel olarak ayrıldığı İstemci/Sunucu mimarisidir.



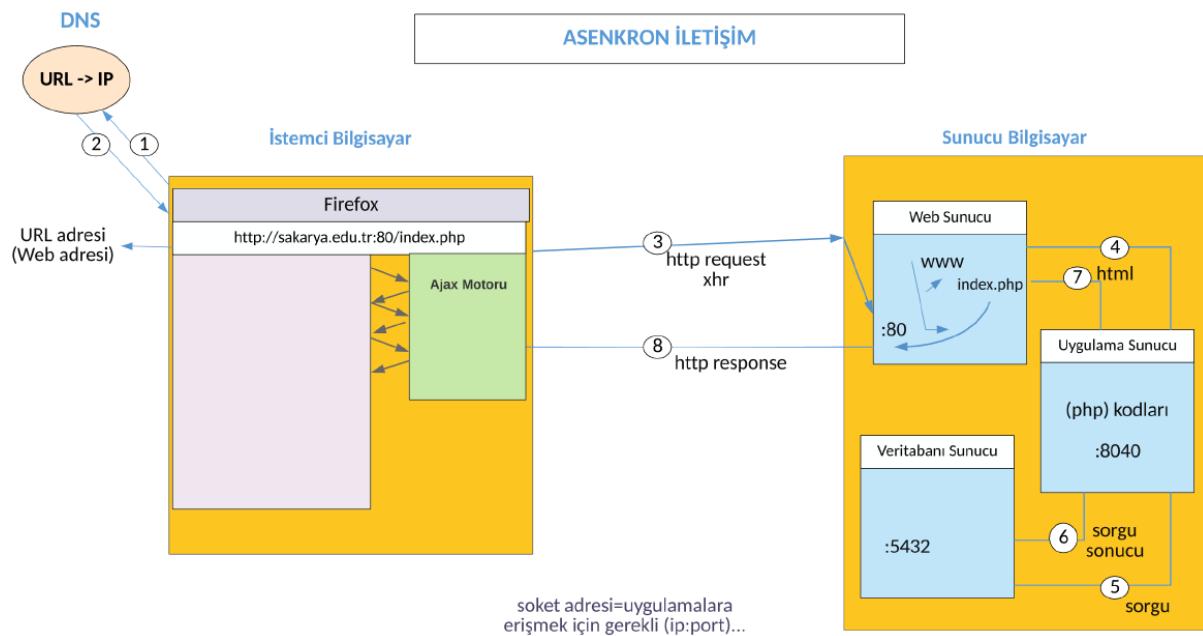
Sunum katmanı denilen Front-End, Uygulama Katmanı Denilen Back-End kısmı.

Senkron İletişim



Asenkron İletişim

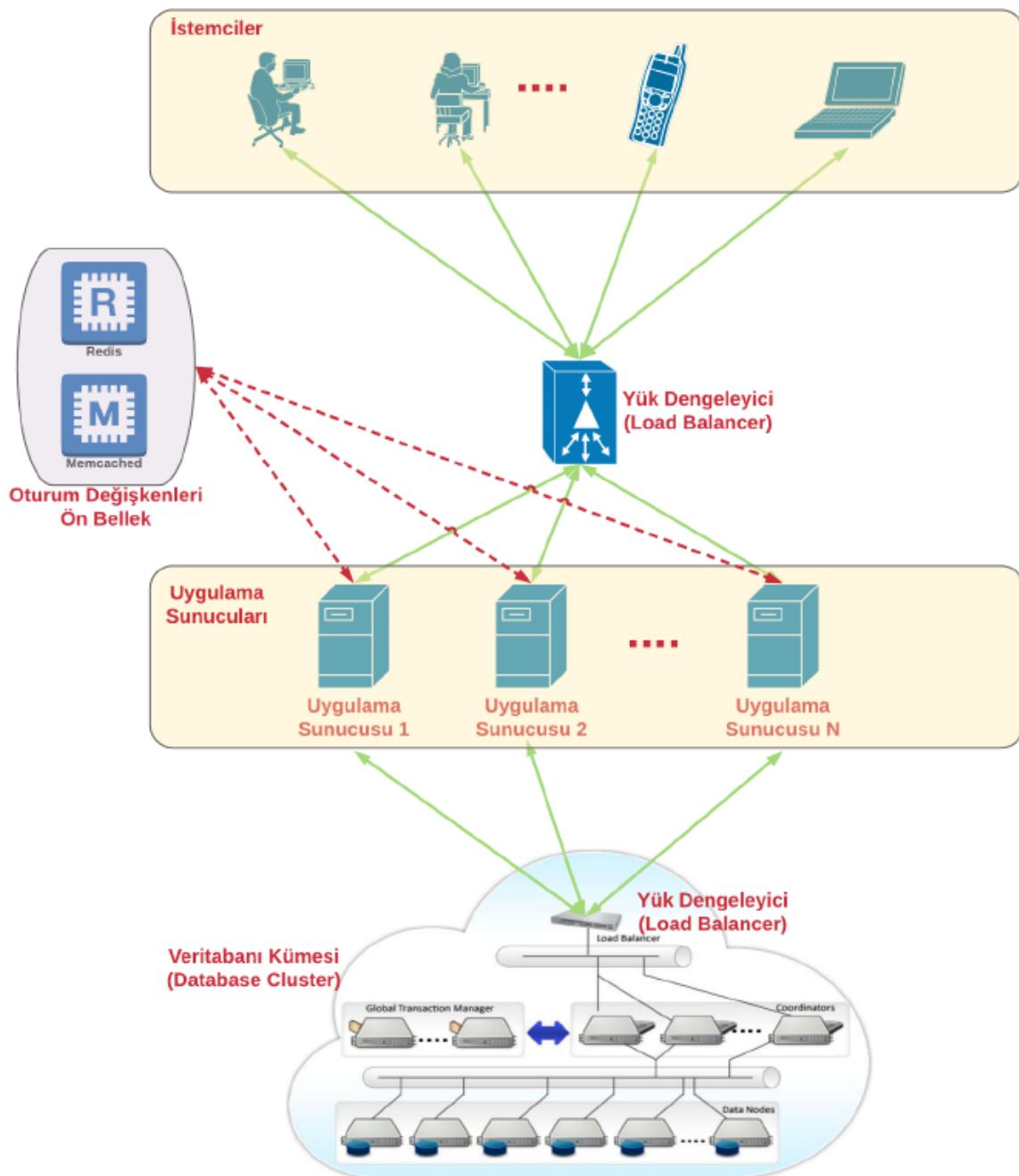
Senkronandan farklı web arayüzü istemci bilgisayarda oluşur. İstemci ile sunucu arasında yalnızca veri transferi olur. React, Angular gibi, Single Page.



3. İstemci/Sunucu(Ölçeklenebilir Web Uygulama Mimarisi)

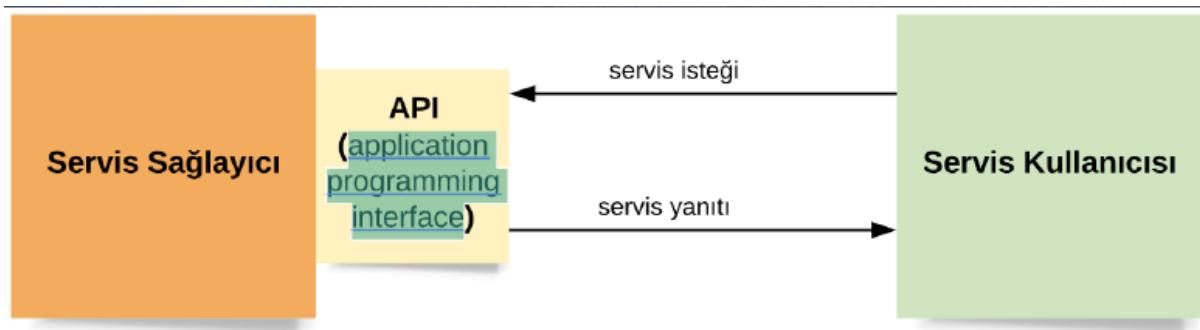
İstemci sayısı arttıkça sunucu yetersiz kalacağından sunucu sayısı arttırılır. Bu yapıda ise uygun sunucuya isteği aktaran bir yapı gereklidir. Burada bu işi Load Balancer yapar.

Yaptığı iş istemciden istek geldiğinde gelen isteği uygun, boşta olan sunucuya aktarmaktır.



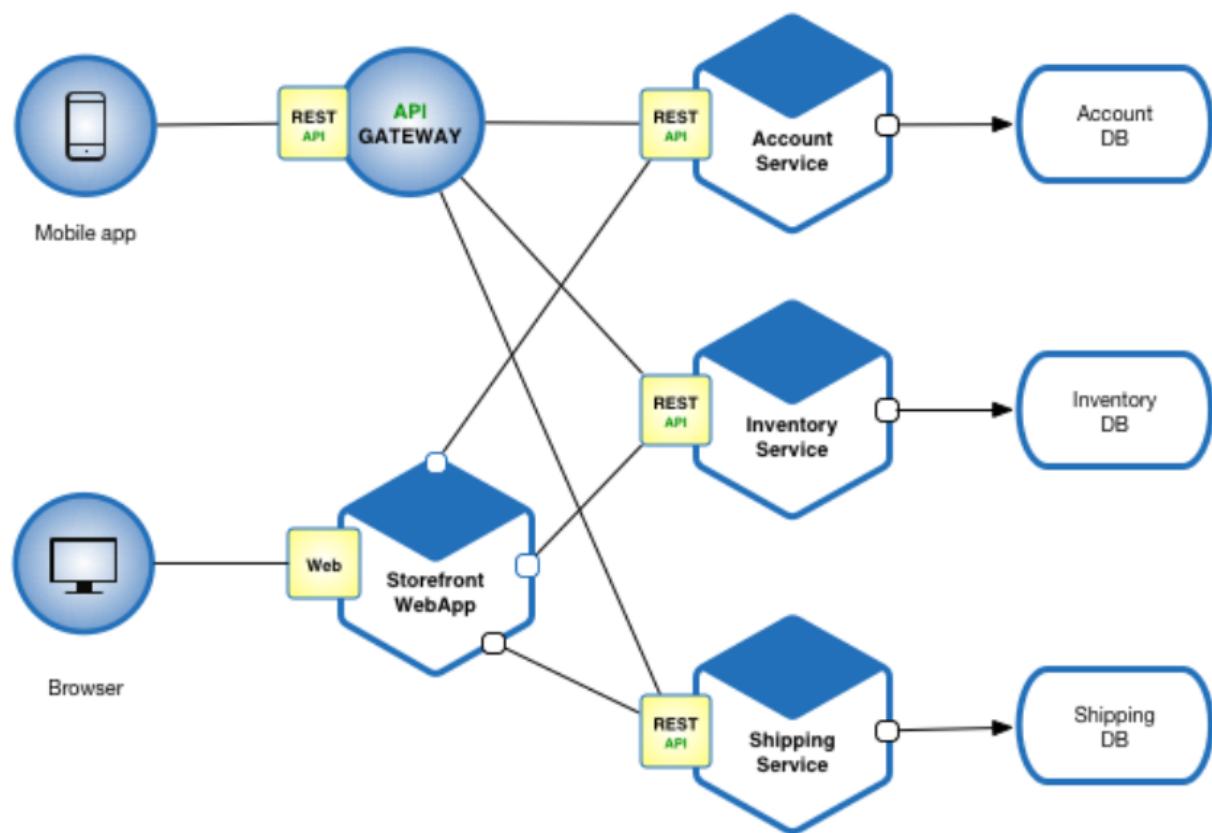
4. Servis Yönetimli Yazılım Mimarisi (Service-Oriented Architecture (SOA))

Web uygulamaları kullanıcılarla hizmet vermek için üretilir. Servis uygulamaları yani web servisi uygulamaları ise sistemlerin birlikte çalışmasını sağlamak için kullanılır. Servisler birbirlerinden bağımsızdır. RESTful sık kullanılır.



5. Mikro Servis Mimarisi

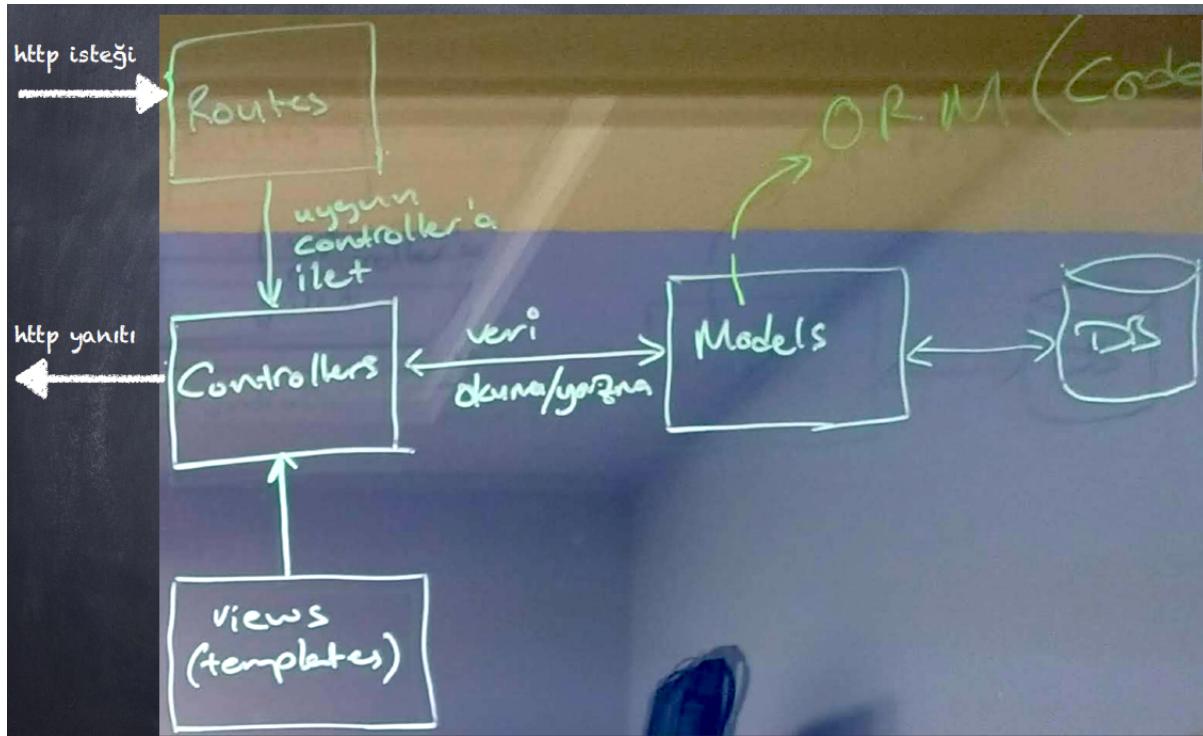
Yaygın olarak kullanılan, servis yönelimli mimarinin modern bir türevidir. Modüllerini servisler haline getirme üzerine kuruludur. Yani normalde Java'da bir program yaparken örneğin ATM, bunda para çekmeyi bir modül yani aynı paket, aynı istemci-sunucuda çalışacak vaziyetteyken biz bunu bir servis haline getiriyoruz. Bu sayede farklı modüller farklı dillerle geliştirilebiliyor.



6. MVC

Sorumlulukları ayırtırır. Veritabanı ile ilgili işlemler Model kısmında yapılır. Görünüm, tasarım işlemleri View kısmında yapılır. Controller ise istekleri alır, model kısmında yaptığı

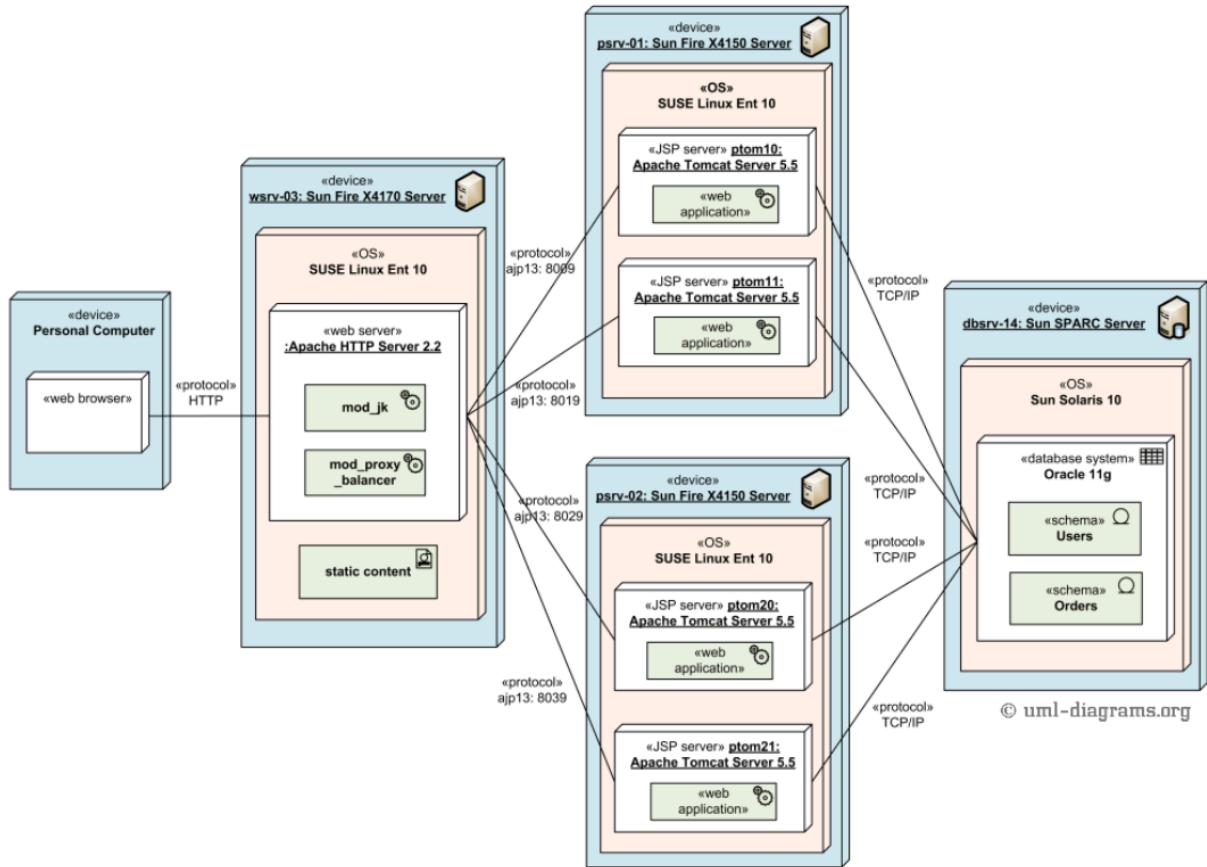
İşlemleri view ile birleştirerek geri döndürür.



4. Sistemin Konuşlandırılması/ UML "Deployment" Diagramı

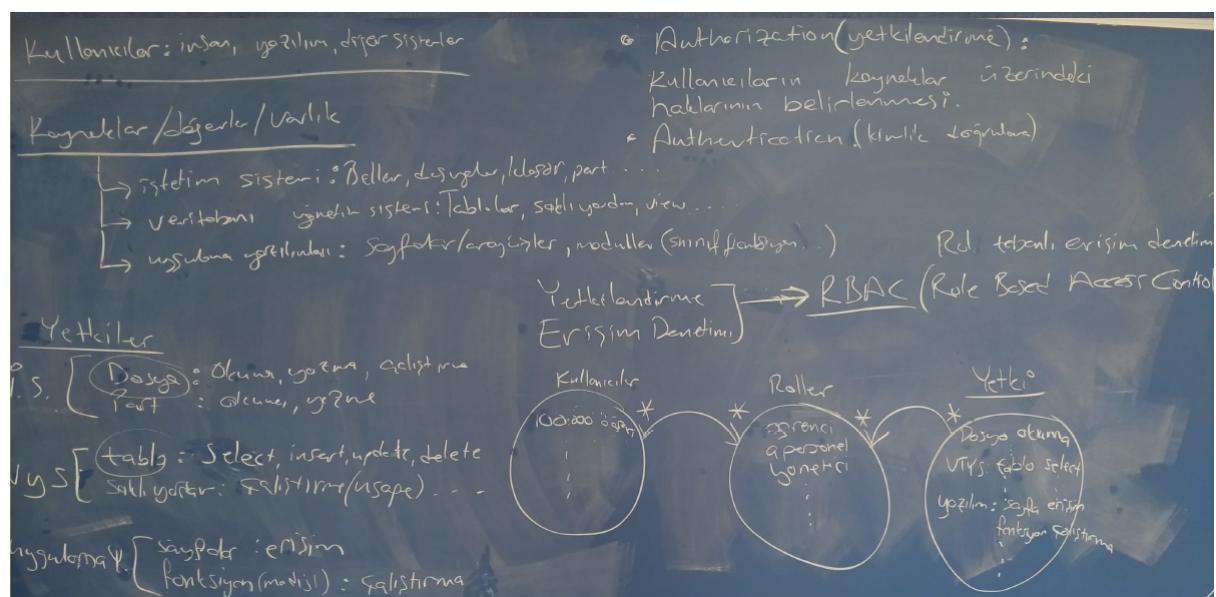
Yazılım ürünlerinin/bileşenlerinin (artifact/component) düğümlere (cihaz, işletim sistemi, sanal makine vb.) konuşlandırılmasıyla ilgili bilgi verir.

Yazılımların konuşlandırdığı fiziksel topolojiyi gösterir.



5. Erişim Denetimi (Rol Tabanlı Erişim Denetimi)

Yetki (Permission): Kaynaklar (Sayfa, modül, veritabanı) üzerindeki sahip olunan haklar. Yetkiler rollerle ilişkilendirilir, kullanıcılar rollere üye yapılır. Böylece yetkilerin yönetimi daha kolay yapılır.



Bu işlemde etkinlik matrisi de oluşturulur. Örn;

Case Study: Online Shopping Application				
ACTIVITY MATRIX				
		External Entities & Roles		
Assests	Action	Guest	Registered User	Admin
Customer Data	Create	Always	Own Data	Always
	Read	Never	Own Data	Always
	Update	Never	Own Data	Always
	Delete	Never	Own Data	Always
Login Page	Access	Always	Always	Always
Customer Profile Page	Access	Never	Always	Never
Login Function	Execute	Always	Always	Always
Add Shopping Chart Function	Execute	Never	Always	Never
Checkout Fuction	Execute	Never	Always	Never

Action'lar yetkilerdir.

6. Erişim Denetimi (Web Application (Spring Security))

Güvenlik için erişim denetimi merkezi olarak yapılmalıdır. Örn;

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry
            .antMatchers("/resources/**", "/").permitAll()
            .antMatchers("/dashboard").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and() HttpSecurity
        .formLogin() FormLoginConfigurer<HttpSecurity>
            .defaultSuccessUrl("/dashboard")
            .and() HttpSecurity
        .logout() LogoutConfigurer<HttpSecurity>
            .permitAll();
}

@GetMapping("/registeredUser")
@PreAuthorize("hasAuthority('RegisteredUser')")
public String moderatorAccess() {
    return "RegisteredUser Board.";
}

@GetMapping("/administrator")
@PreAuthorize("hasAuthority('Administrator')")
public String adminAccess() { return "Admin Board."; }

```

Örneğin burada antMatchers ile resources'a "/" ile herkesin erişebileceği, dashboard'a ise yalnızca ADMIN rolü olanların erişebileceği bildirilmiştir.

Erişim denetimi hem uygulama hem veritabanı için de yapılmalıdır. Yani işlem gereken ne ise onda yapılmalı.

Veritabanı açısından ise örn;

Yetkilendirme İşlemleri

PUBLIC: Tüm roller / kullanıcılar.

kullaniciAdi: Tek bir kullanıcı.

ALL: Tüm yetkiler.

- rol1 isimli role customers tablosu üzerinde seçim yapma yetkisi ver.

```
GRANT SELECT ON "customers" TO "rol1";
```

- Tüm rollere customers tablosu üzerinde kayıt ekleme yetkisi ver.

```
GRANT INSERT ON "customers" TO PUBLIC;
```

- rol1 isimli kullanıcıya customers tablosu üzerinde tüm yetkileri ver.

```
GRANT ALL ON "customers" TO "rol1";
```

- rol1 isimli rolün customers tablosu üzerindeki güncelleme yetkisini geri al.

```
REVOKE UPDATE ON "customers" FROM "rol1";
```

- rol1 isimli rolün customers tablosu üzerindeki tüm yetkilerini geri al.

```
REVOKE ALL ON "customers" FROM "rol1";
```

- rol1 kullanıcısının Sema1 içerisindeki nesnelere ait tüm yetkileri geri alınır.

```
REVOKE ALL ON SCHEMA "Sema1" FROM "rol1";
```

7. Nesnelerin Depolanması

Bazı nesnelerin (entity object) kalıcı olarak saklanması gereklidir.

Kalıcı depolama için klasik dosyalar ya da veritabanı yönetim sistemleri (ilişkisel VYS, nosql vb.) kullanılabilir.

İlişkisel VYS kullanılması durumunda saklanacak nesneler için tablo oluşturulur.

Tablodaki her satır bir nesneye, tablo ise sınıfı karşılık gelir.

Sınıfın adı ile veritabanındaki tablo adı, sınıfındaki her üye değişkenin adı ile veritabanında her üye değişkenin yani sütunun adı aynı olmalıdır. Tabloda her sütun üye değişken, her satır nesnedir.

Bu veritabanını soyutlamak gereklidir ve bu işlem Interface ile yapılabilir. Örneğin ATM örneğindeki soyutlama yani Postgre'yi nesne olarak gönderince Postgre ile çalışır. Buna **Repository Pattern** denir.

Nesne Tasarımı

Tasarım İlkeleri: S.O.L.I.D

Tasarım ile gerçeklemenin birbirinden ayrılması sağlanır. Yani bu ilkelerin mevcut olduğu bir programda değişiklik gerektiren bir durumda kaynak kod üzerinde çok çok az işlem yapılacaktır.

Modüllerin bağımlılığı iyi yönetilirse;
tasarım daha kararlı olur, gerçeklemedeki değişikliklerden (çok) etkilenmez
“cohesion” artar “coupling” azalır
sonradan değişiklik kolaylaşır
kod tekrar kullanımı/modülerlik artar
yeni özellik kazandırma kolaylaşır
anlaşılabilirlik artar
 karmaşıklık azalır (gerekli bağımlılıklar ortadan kalkar)
birim test yazımı kolaylaşır

SRP	The Single Responsibility Principle	<i>A class should have one, and only one, reason to change.</i>
OCP	The Open Closed Principle	<i>You should be able to extend a classes behavior, without modifying it.</i>
LSP	The Liskov Substitution Principle	<i>Derived classes must be substitutable for their base classes.</i>
ISP	The Interface Segregation Principle	<i>Make fine grained interfaces that are client specific.</i>
DIP	The Dependency Inversion Principle	<i>Depend on abstractions, not on concretions.</i>

The Single Responsibility (SRP)

Bir sınıfın(modülün) değişmesi için tek bir neden olmalıdır. Her sınıfın tek bir sorumluluğu olmalıdır.

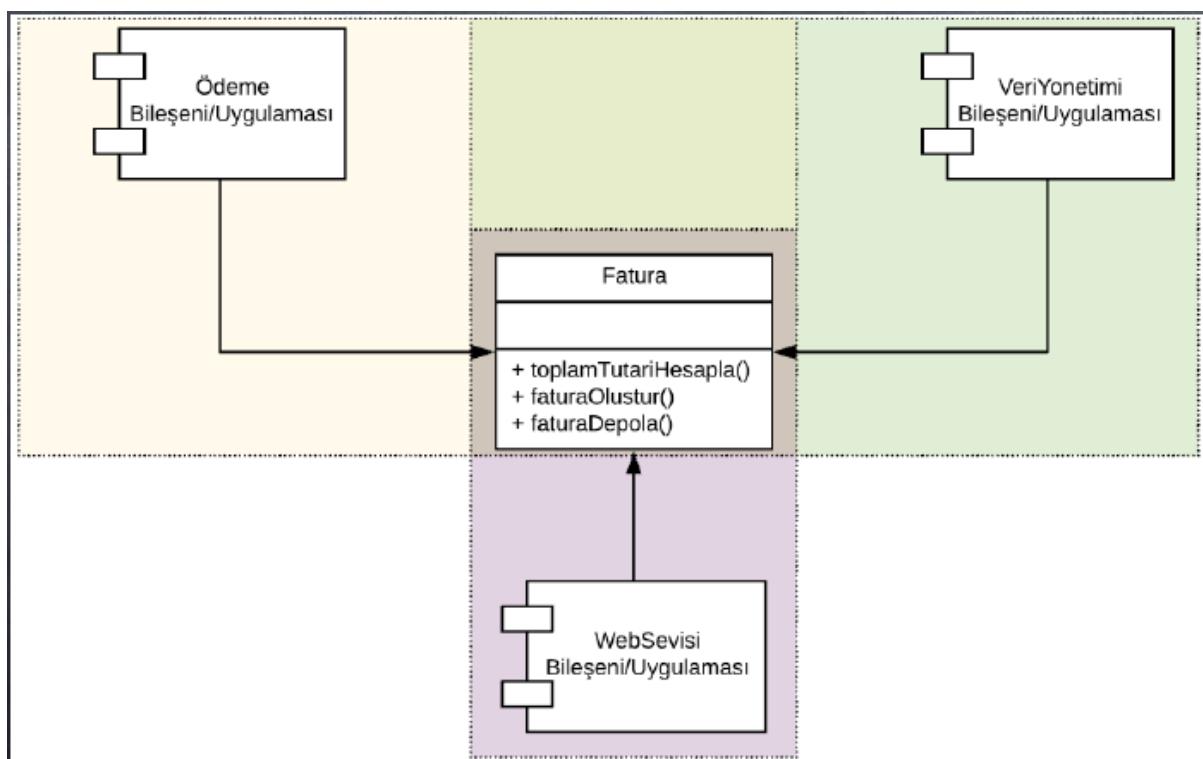
Bir sınıf için; genel yönetim(business logic), veri yönetimi (saklama, erişme v.s.), sunum (gui, json, xml v.s.), iletim (gönder ,al v.s.), nesne oluşturma (factories), main vb. işlemlerinin her biri farklı sorumluluk olarak ele alınmalıdır.

Bir modülün tek bir sorumluluğu olmalı ve bu sorumluluğu mükemmel olarak yerine getirmelidir.

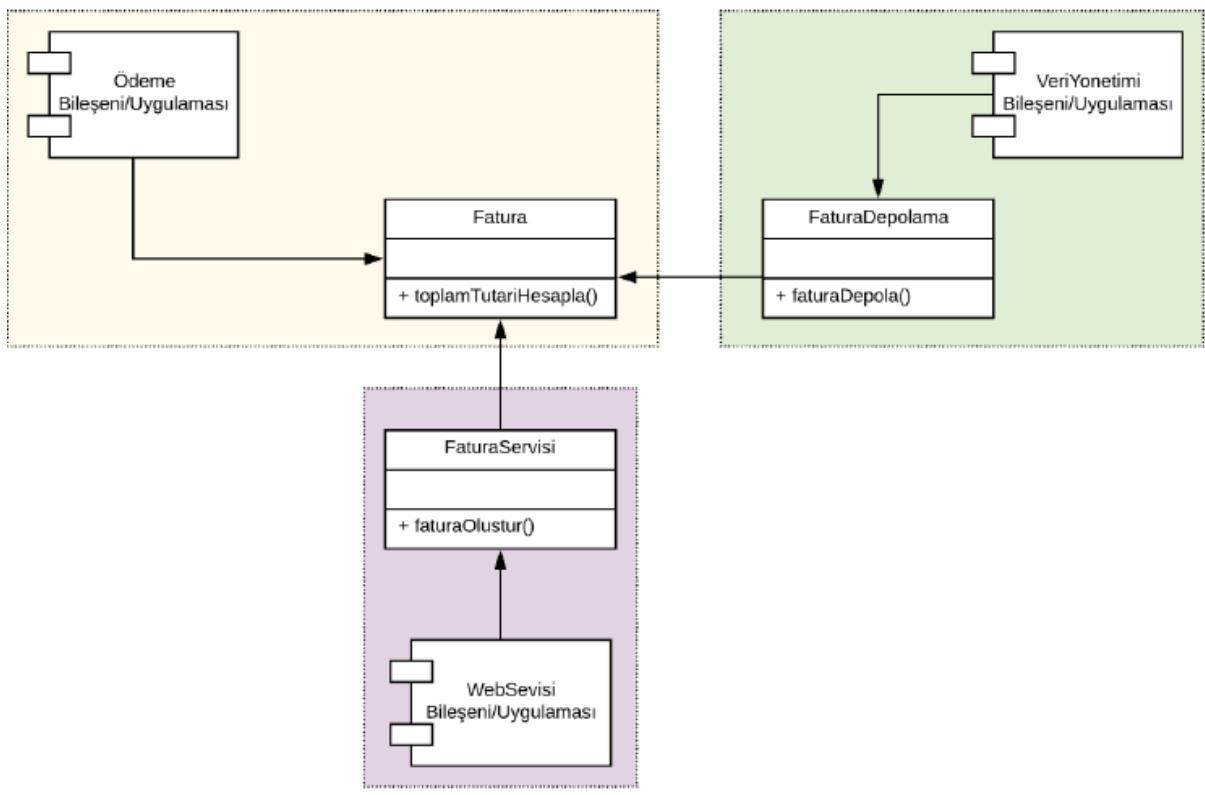
Bir module'ye birden fazla sorumluluk yüklenirse;
değişiklik yapmak zorlaşır ve bakım aşamasının maliyeti yükselir
değişikliğin yapıldığı module doğrudan bağlı modüllerde de değişiklik gerekebilir
doğrudan bağlı olmayan modüllerin tekrar derlenmesi, paketlenmesi, gönderilmesi v.s.
gerekebilir
sonunda her değişiklik çok karmaşık hale gelir ve yan etkiler ortaya çıkar (side effects,
regresyon hataları)

anlaşılması ve yönetilmesi zorlaşır
birim testi yazmak zorlaşır

Örneğin bu örnek için Fatura sınıfı 3 farklı iş正在做着和正在被做的
ve bu istenen bir şey değil.



Biz bunu sorumluluklarının her biri ayrı modüller tarafından yerine getirecek şekilde düzenliyoruz ve bu sayede örneğin fatura servisindeki bir değişiklik sadece WebServisi modülünü etkileyecektir.



The Open Closed (OCP)

Bu prensip, değişiklik yapmadan genişleyebilen modüller geliştirmeyi amaçlar.

Mevcut kodu değiştirmeden yeni kodlar ekleyerek, yeni özellikler eklemeyi sağlar.

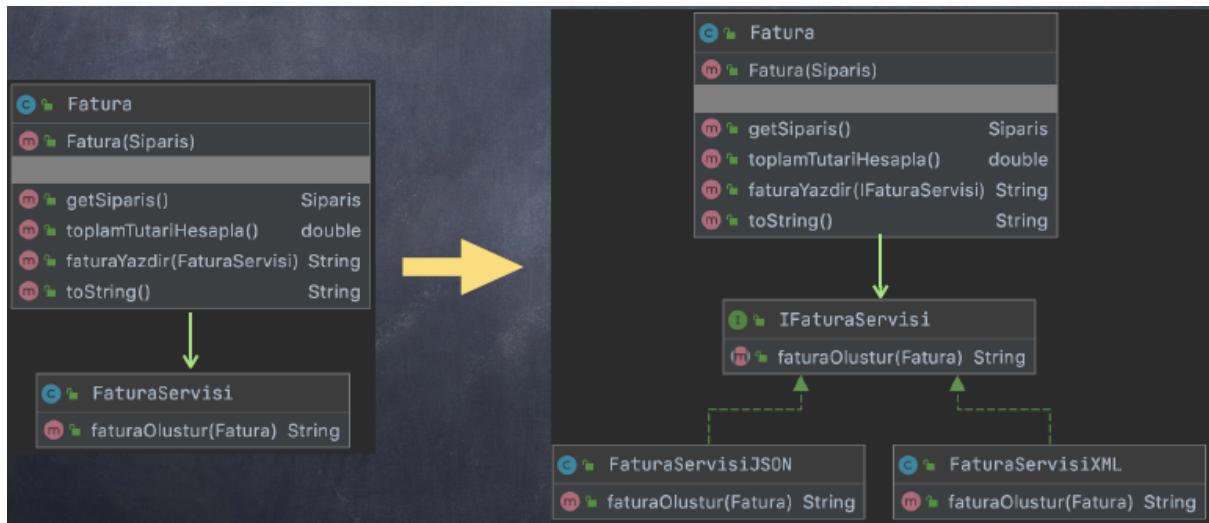
“Open for extension”: geliştirilen modüller yeni gereksinimler ortaya çıktığında bunları desteklemek üzere genişleyebilmeli.

“closed for modification”: geliştirilen modüller hata giderme dışında değiştirilmemeli.

Mevcut kodda değişiklik yapmadan yeni özellik ekleyebilmenin yolu, bağlı olunan modüllerin gerçeklemeleri yerine soyutlamalarının kullanılmasıdır. (dependency inversion)

Bu yapı genişlemeye açık, değişikliğe kapalıdır. Adı da buradan gelmektedir.

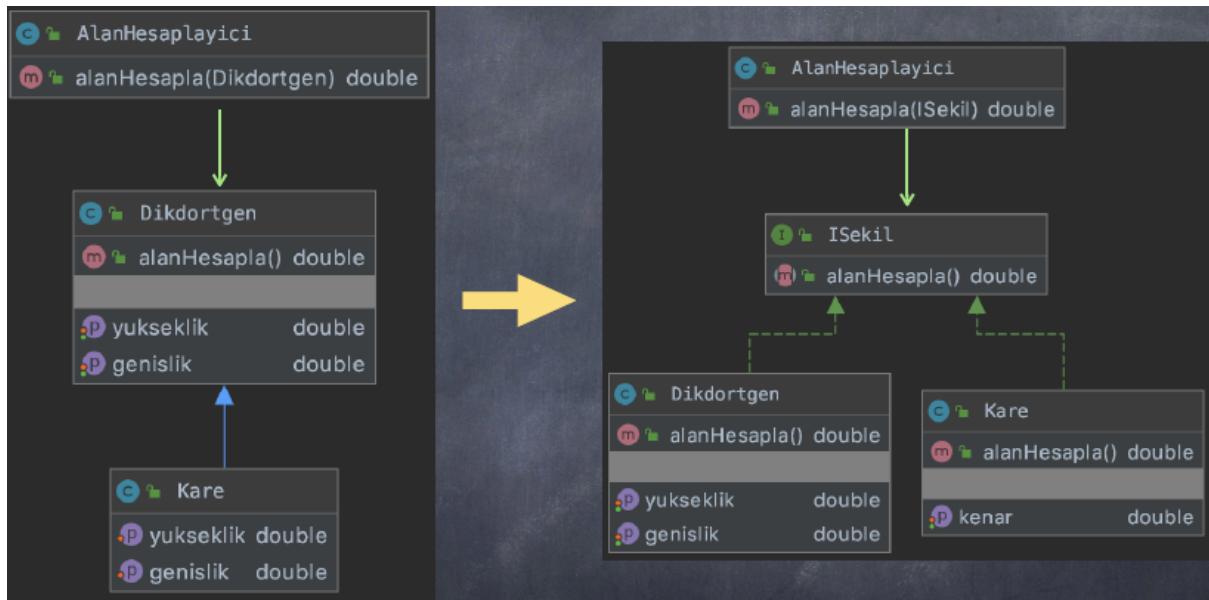
Soldaki örnekte, Fatura modülü FaturaServisi modülünün gerçeklemesine bağlı. Gerçekleme içerisinde yapılacak değişiklik istemci modülü(Fatura) de etkileyebilir. Sağdaki örnekte, OCP ilkesi uygulanmıştır. Fatura modülü FaturaServisi modülünün soyutuna (IFaturaServisi) bağlı. Gerçeklemelerdeki değişiklikler (yeni özellik gereksinimleri gibi) istemci modülü etkilemeyecektir.



The Liskov Substitution Principle (LSP)

Temel sınıf nesneleriyle çocukların nesneleri, istemci modül içerisinde sorun çıkarmadan yer değiştirebilmelidir ve kalıtım hiyerarşisinin tutarlı olarak belirlenmesini sağlar. Open/Closed ilkesine uymayan hiyerarşilerin önüne geçmemizi sağlar.

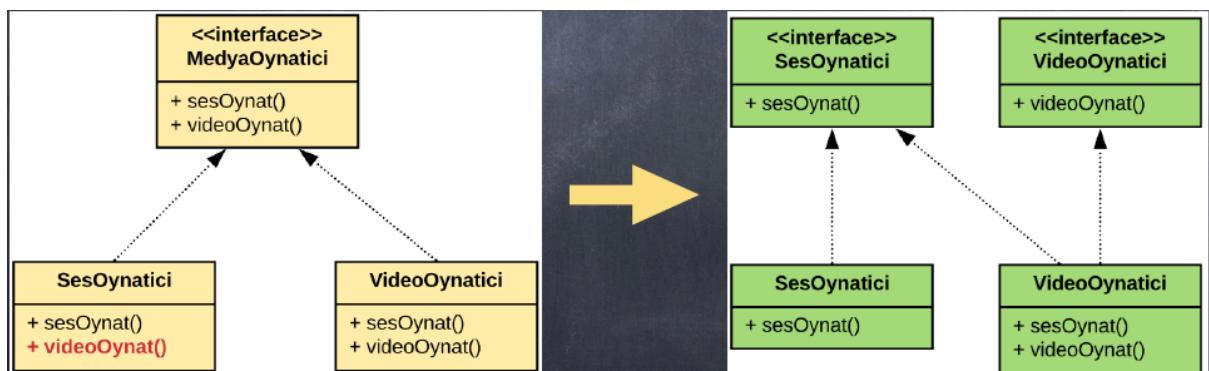
Bu örnekte AlanHesaplayıcı sınıfının alanHesapla metodunda Dikdortgen nesnesini kullanıyor fakat bunu kullanırken Kare Dikdörtgen'den kalıtım aldığı için parametre olarak Kare de verilse bunu da Dikdortgen sınıfından kalıtım aldığı için alanHesapla metodunda kullanabiliyoruz. Fakat burada şöyle bir tutarsızlık ortaya çıkıyor, dikdörtgende genişlik ve yükseklik aynı değildir, karede aynı olduğu için alanHesaplayıcı burada problem çıkarıyor hesaplamlarda. Yani aslında alanHesapla dikdörtgenin hesabı için doğru tasarlanmış ve bu da farklı hesaplamlarda soruna yol açıyor. Bunu sağlayabilmek için de sağ taraftaki şekele getiriyoruz. Yani interface ile yine soyutlama kullanıyoruz ve bu interface üzerinden Dikdortgen ve Kare'yi turetiyoruz, gerçekliyoruz. Dolayısıyla alanHesapla'yı her nesne için doğru kodlayabiliyoruz. Sonuç olarak artık Kare, Dikdortgenden kalıtım almıyor, her 2'si de ISekil interface'inden arayüz kalıtımı yapıyor.



The Interface Segregation Principle (ISP)

Bu prensip, modüllerin kullanmayacakları arayzlere bağlı kalmaması gerektiğini ifade eder. Kullanılmayacak arayzlere bağlı kalmak, bunlarda yapılacak (bağlı diğer modüller tarafından) değişikliklerden etkilenme (side effects) anlamına gelmektedir. SRP ilkesinin ihlal edilmesidir. Böyle bir durum ortaya çıktığında arayüzlerin ayrılması gereklidir.

Bu örnek için solda SesOynatıcı modülü, öyle bir fonksiyonu olmadığı halde, videoOynat() yöntemini içermek zorunda bırakılıyor-ISP nin ihlali durumu olmuş oluyor ve buna çözüm olarak sağdaki şekilde arayüz, 2 farklı arayüz olarak düzenlenerek VideoOynatıcı modülü her 2 arayüzden kalıtım alarak, gerçekleştirmesi durum çözülmüş oluyor.



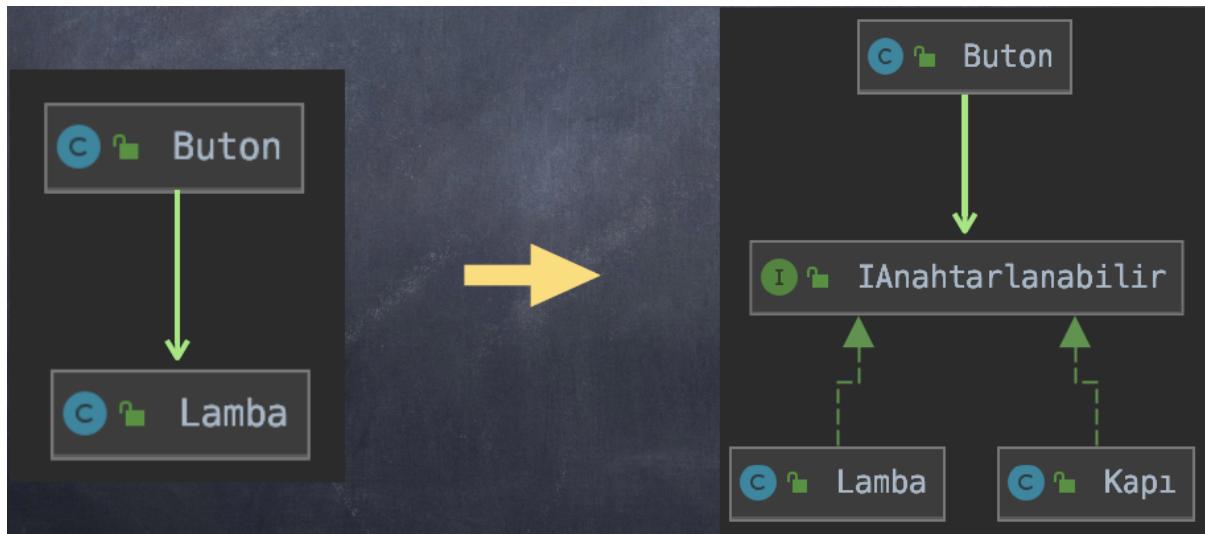
The Dependency Inversion Principle (DIP)

Nesneler arasındaki bir bağlantıda, yüksek seviyeli modül ile düşük seviyeli modül (her ikisi birden) soyutlamaya bağlı olmalı. Her ikisi birbirine doğrudan değil arayüz üzerinden bağlanmalı. Böylece modülde yapılacak değişiklik diğer modüller etkilememiş olur. Değişiklik yapmak kolaylaşır. Yeni özellik eklemek kolaylaşır (OCP) ve modüllerin tekrar kullanım oranı artar.

OCP ve LSP uygulandığında, aynı zamanda DIP uygulanmış olur.

Soyutlama olayı, veritabanı vs. yani bir modül başka bir modülü kullanacağı zaman o modülün gerçeklemesine değil soyutuna bağlı olmalıdır, onu kullanmalıdır. Buradaki soyut kavramı sağlayacak olan da interface veya soyut sınıfıtır.

Bu örnekte sol tarafta buton yalnızca lamba için kullanabilecekken bunu Anahtarlanabilir arayüzüyle soyutlayarak butonun aldığı gerçeklemeye göre yani Lamba veya Kapıya göre çalışabilmesini sağlamış oluyoruz.



Tasarım Desenleri

Yazılımlar geliştirilirken karşılaşılan genel tasarım problemlerini tanımlayarak, bu problemin en uygun nasıl çözülebileceğini ((high coherence, low coupling) kod tekrar kullanımını artırmak ve değişikliği kolaylaştırmak için) ve çözümün ortaya çıkaracağı sonuçları anlatır.

Programlama dillerinden bağımsızdır.

Yazılım geliştiricilerin tasarımlarla ilgili tartışma yapmasını kolaylaştırır.

Tasarım desenleri dört bölümden oluşur:

desenin adı

problemin tanımı: desenin ne zaman/nerede kullanılabileceği

çözüm: problemin nasıl çözüleceği (kullanılması gereken bileşenler, aralarındaki bağıntı vb.)

sonuç: deseni uygulamanın sonuçları?

Creational(Nesne oluşturma) Desenleri: Nesnelerin uygun bir şekilde oluşturulmasını sağlayacak mekanizmalar içerir.

Structural(Yapisal) Desenleri: Nesnelerin sistemler içerisinde uygun olarak yerleştirilmesini sağlayacak desenlerdir.

Behavioral(Davranışsal) Desenleri: Nesnelerin birbirleriyle uygun olarak etkileşiminini düzenleyen mekanizmalar.

Creational (Nesne oluşturma)	Structural (Yapisal)	Behavioral (Davranışsal)
Singleton Pattern	Adapter Pattern	Template Method Pattern
Factory Pattern	Composite Pattern	Mediator Pattern
Abstract Factory Pattern	Proxy Pattern	Chain of Responsibility Pattern
Builder Pattern	Flyweight Pattern	Observer Pattern
Prototype Pattern	Facade Pattern	Strategy Pattern
	Bridge Pattern	Command Pattern
	Decorator Pattern	State Pattern
		Visitor Pattern
		Interpreter Pattern
		Iterator Pattern
		Memento Pattern

Singleton

Amaç bir sınıfın yalnızca tek nesnesi olmasını ve bu nesneye global olarak erişilmesini sağlamaktır.

Nesne yoksa oluşturulur ve döndürülür, varsa olan döndürülür.

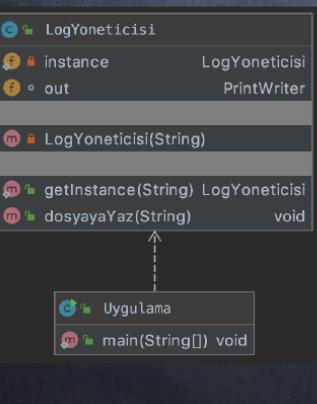
Nesnenin new komutu ile dışarıdan oluşturulabilmesini engellemek için yapıçı yöntem “private” yapılır.

Yapıcı gibi çalışan “static” bir yöntem tanımlanır. Nesne oluşturma görevi bu yöntemindir. Oluşturulan nesne “static” bir üyede saklanır.

Örneğin bir loglama sistemimiz var ve biz bu nesneden program içerisinde yalnızca 1 tane üretilmiş olmasını garanti altına almak istiyorsak singleton desenini kullanırız. Yani bir nesneden 2. kez üretilmesini engellemiş oluruz ve o sınıfla işimiz olduğumuzda

yalnızca tek nesne ile muhatap oluruz. Bu tek nesne oluşturulmasını da singleton tasarım desenine sahip olan (bu örnek için LogYonetici) sınıfında nesne oluşturulmasını sağlıyoruz.

Nesneye erişmek istediğimizde ise sınıfın tanımlı getInstance yöntemi ile erişim sağlıyoruz. (Uygulama main içerisinde görüldüğü üzere)



```
public class LogYonetici {

    private static LogYonetici instance;
    PrintWriter out;

    private LogYonetici(String logDosyasi){
        try {
            out = new PrintWriter(new FileWriter(logDosyasi, true), true);
        } catch (IOException e) {e.printStackTrace();}
    }

    public static synchronized LogYonetici getInstance(String logDosyasi){
        if(instance==null)
            instance = new LogYonetici(logDosyasi);
        return instance;
    }

    public void dosyayaYaz(String mesaj) {
        out.println(LocalDateTime.now() + ":" + mesaj);
    }
}

public class Uygulama {
    public static void main(String [] args){
        LogYonetici.getInstance("Log.txt").dosyayaYaz("[WARNING]:uyari mesaj 1");
    }
}
```

Burada getInstance metodundaki synchronized ifadesi multithread yani paralel programlama mevcutsa başka bir thread de nesne oluşturabileceği ve nesneden 2. kez oluşabileceği için bunun önüne geçmek içindir.

Observer (Gözlemci)

Amacı çok sayıda nesneye, gözlemediğleri nesnede meydana gelen olayı bildirmektir.

Kullanım örnekleri:

mağazaya ürün geldiğinde ilgili, favorilere ekleyen müşterilere bildirim gönderilmesi, ürün indirime girdiğinde bildirim gönderilmesi

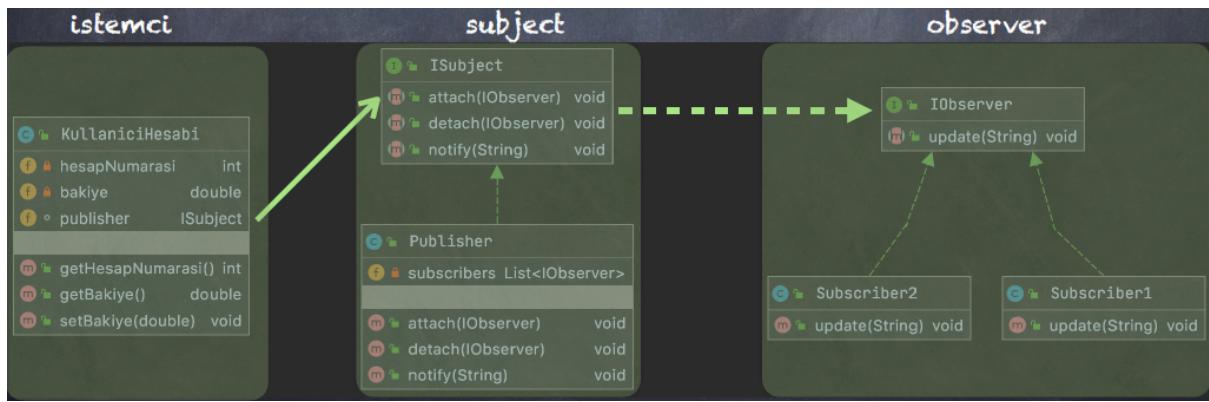
bakiye değiştiğinde müşteri, loglama sistemi ve gerçek zamanlı görüntüleme/anormal durum tespit sistemine bilgi gönderilmesi vb.

Böyle bir etkileşim Publish(Yayın)-Subscribe(abone) olarak da adlandırılır.

Subject (publisher- yayıcı): gözlemcilerin kaydedilmesi/çıkartılması, istemcideki değişikliklerin gözlemcilere bildirilmesi işlemlerinden sorumludur.

Observer (subscriber-abone): istemcideki olayların yansıtıldığı/gönderildiği nesne.

İstemci (kullanıcıHesabı): olayın üretildiği nesne.



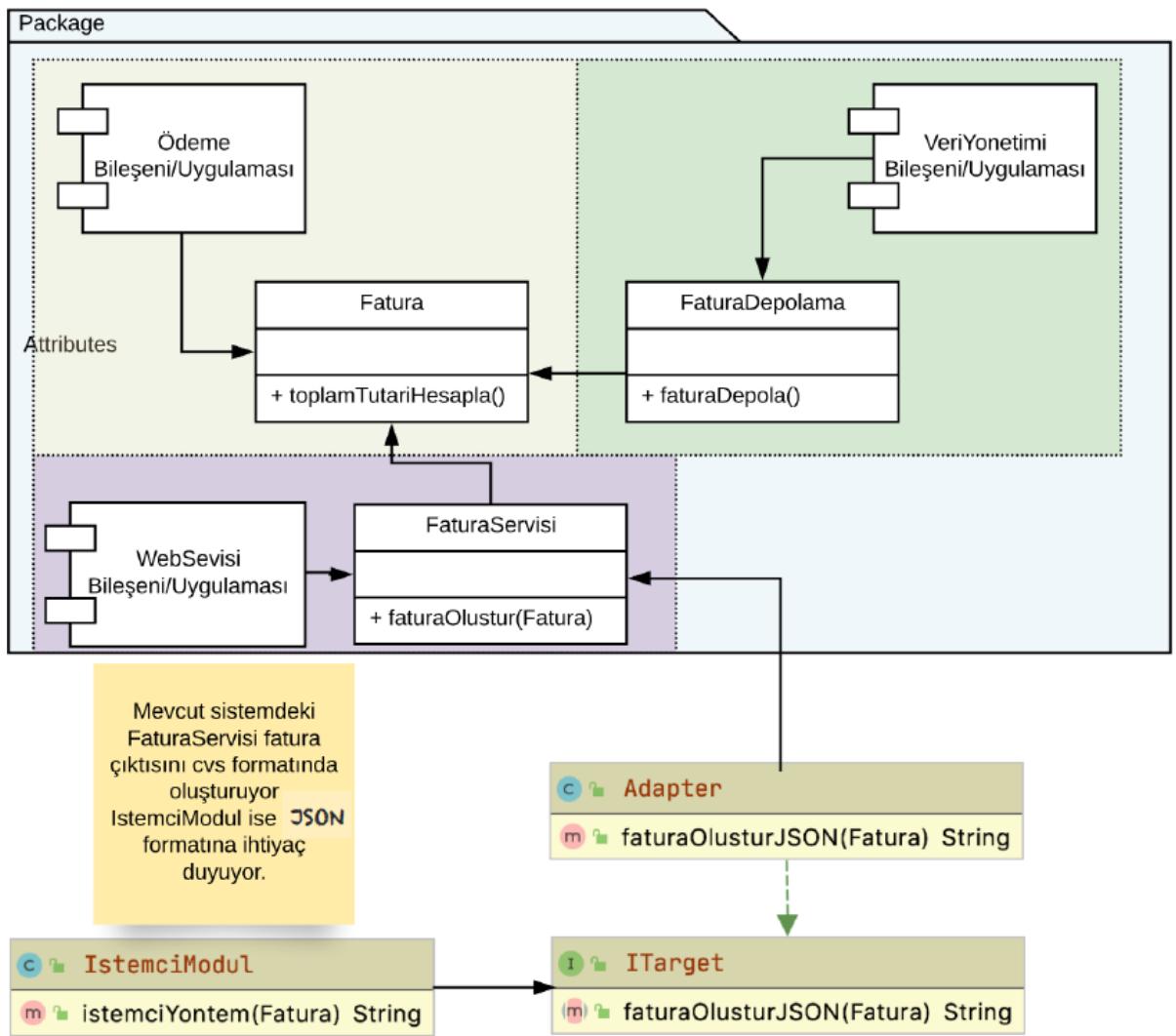
Adapter

Bir sınıfın arayüzüni istemci tarafından beklenen başka bir arayüze dönüştürmek için kullanılır.

Mevcut sınıfların (kaynak kodunu değiştirmeden) diğer sınıflarla çalışabilmesini sağlar. Bu desen sayesinde, daha önceden geliştirilmiş modüller, yeni modüller ile birlikte (bu modüllere uygun arayzlere sahip olmadığı durumlarda) kullanabiliriz.

Bu deseni kullandığımız prizlerde 220volt fakat örneğin telefonumuz 5volt ile şarj ediliyorsa prize taktığımız adaptörün bu 220volt'u 5volt'a çevirmesi olarak benzetebiliriz.

Bu örnek için örneğin faturayı CSV formatında alıyor ve geriye JSON formatında döndürüyor.



Facade

Karmaşık yapıdaki bir sınıf topluluğu (kütüphane, alt bileşen, eskiden yazılmış kodlar vb.) için basitleştirilmiş arayüz sağlar. (Örnekteki SiparisOlusturFacade, Fatura ve Siparis eski sınıfları için basitleştirilmiş bir arayüz olmuş ve Uygulama main içerisinde bunun üzerinden tek metod çağrıarak işi çözüyoruz.)

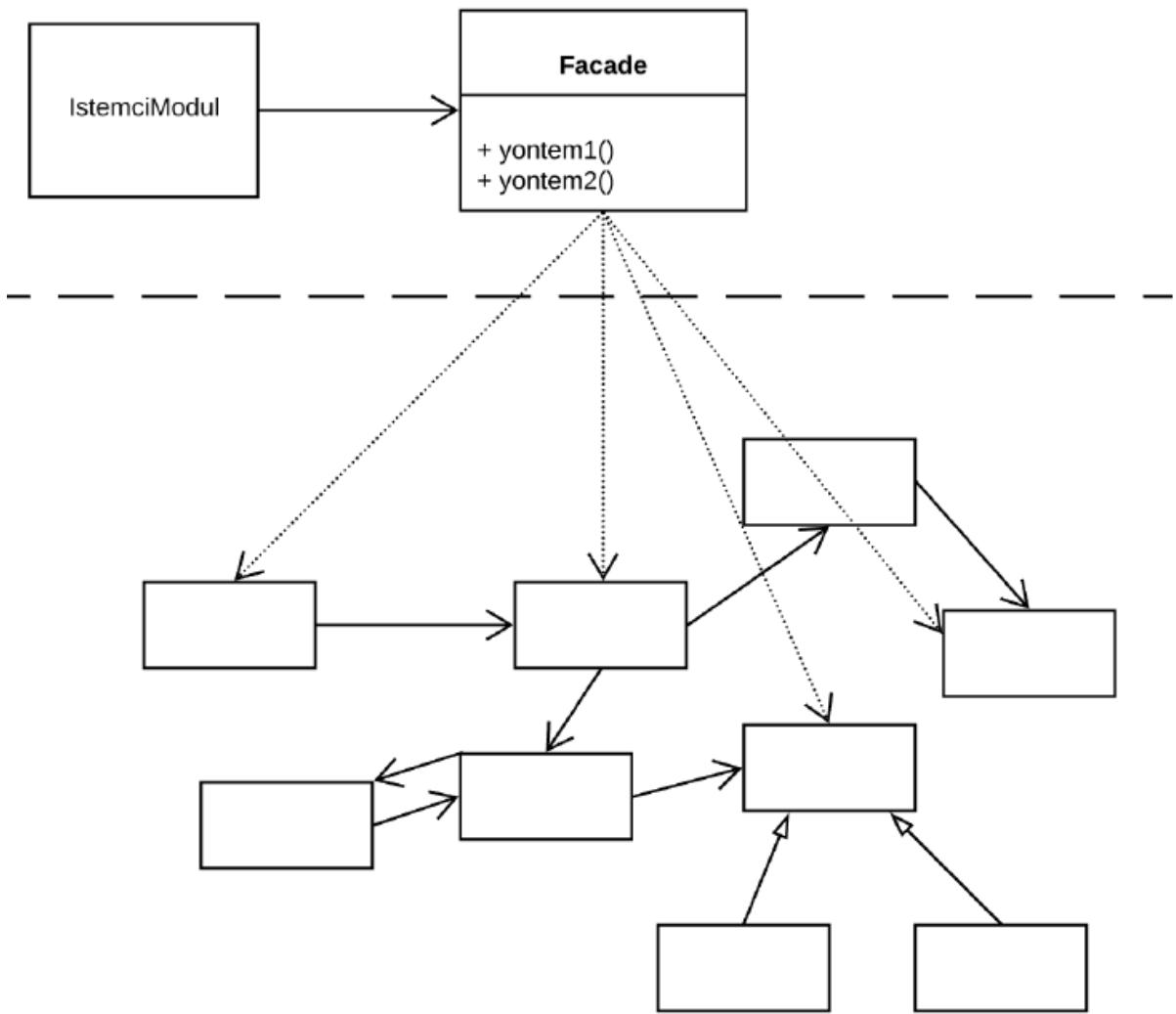
İstemci kodun karmaşık alt sistemle etkileşimini kolaylaştırır (loosly coupling).

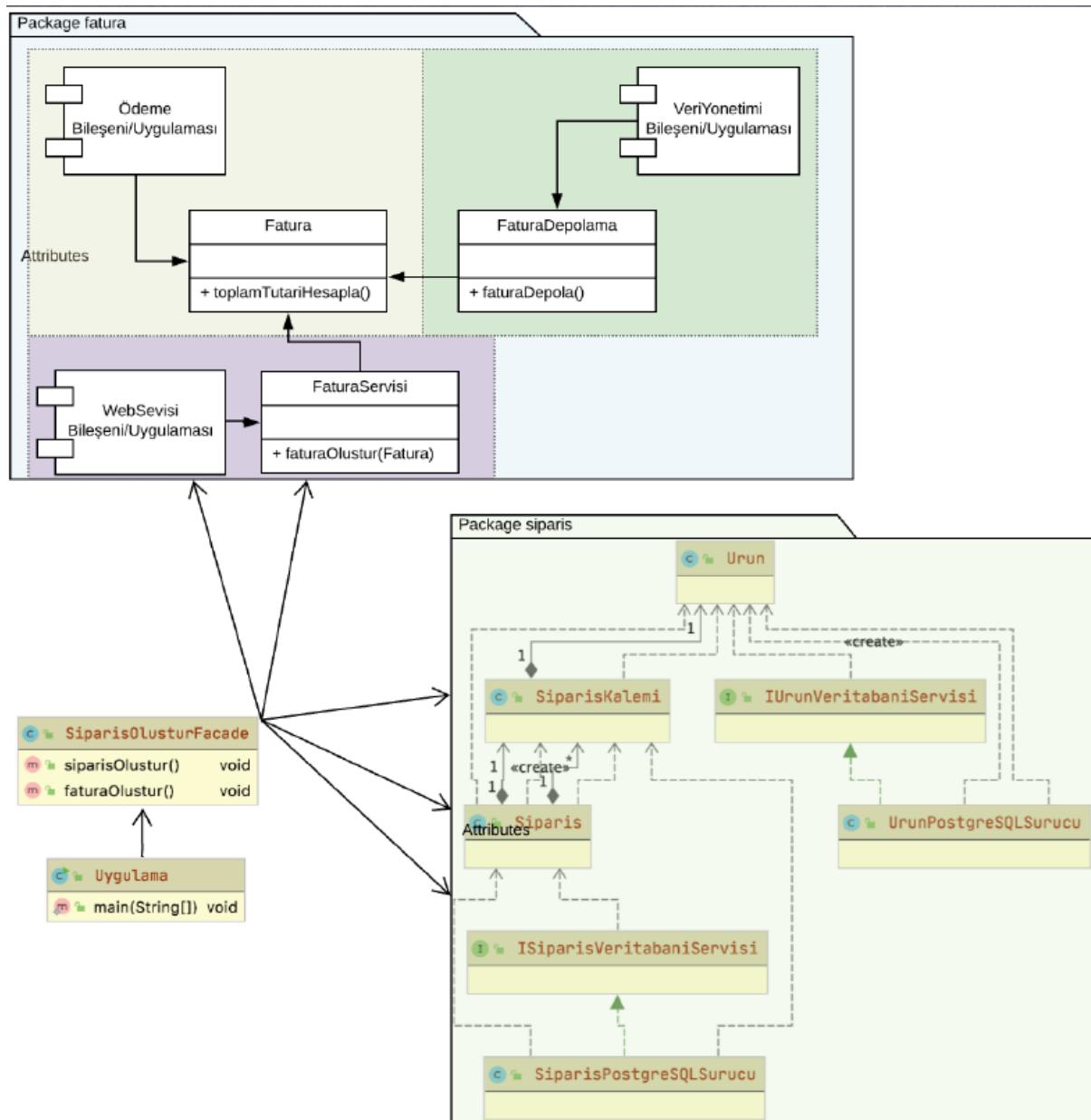
İyileştirilme imkanı olmayan eski modüllerle etkileşim imkanı sağlar.

Anlaşılabilirliği artırır.

Yazılım içerisinde katmanlar oluşturulmasına imkan verir.

Adaptörde arayüzü dönüştürürken, burada ise eski karmaşık modüller soyutluyoruz.





Factory Method

Bir sınıfın nesne oluşturmak gerekiğinde, bu sorumluluğu istemcikoddan ayırmak (kapsülleme/SRP) için kullanılır.

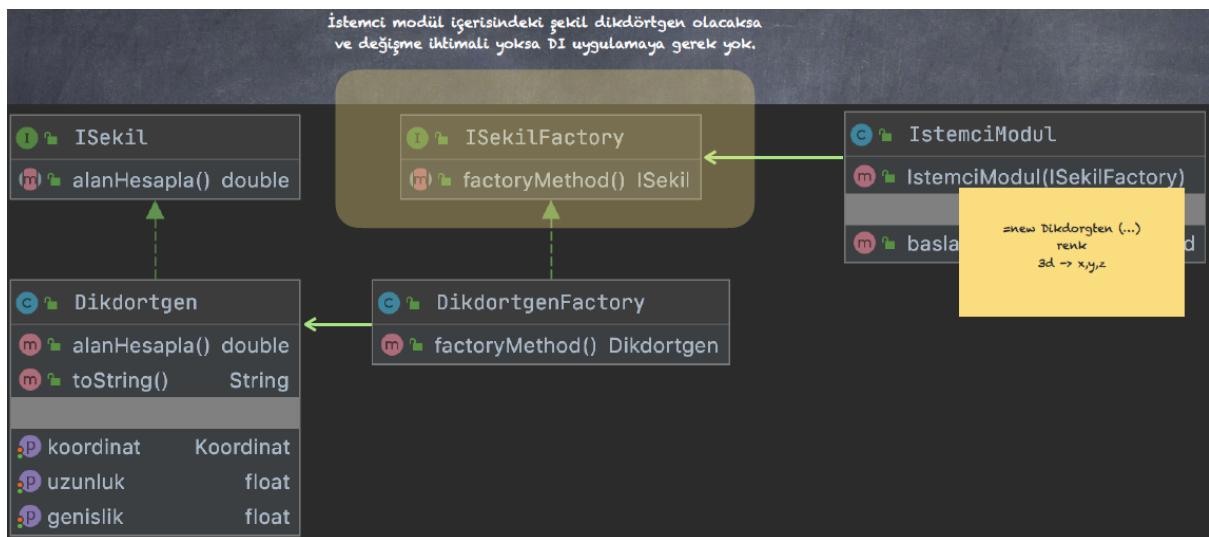
Özellikle çok sayıda parametre göndermek gerekiğinde (kalıtım söz konusu ise) nesne oluşturma işi karmaşıklaşır (kod tekrarı, değişikliklerden istemci kodun etkilenmesi, kodların kötü görünmesi...).

Bu yapıda ise nesne oluşturmak gerekiğinde “factory method” çağrılarılarak nesne oluşturulur. Sınıfın bulunduğu yol/paket, istisna yönetimi v.s. uğraşmaya gerek yok. Nesne oluşturmayla ilgili herhangi bir değişiklikten (yolların değişimi, parametre değişimleri v.s.), istemci kod etkilenmez.

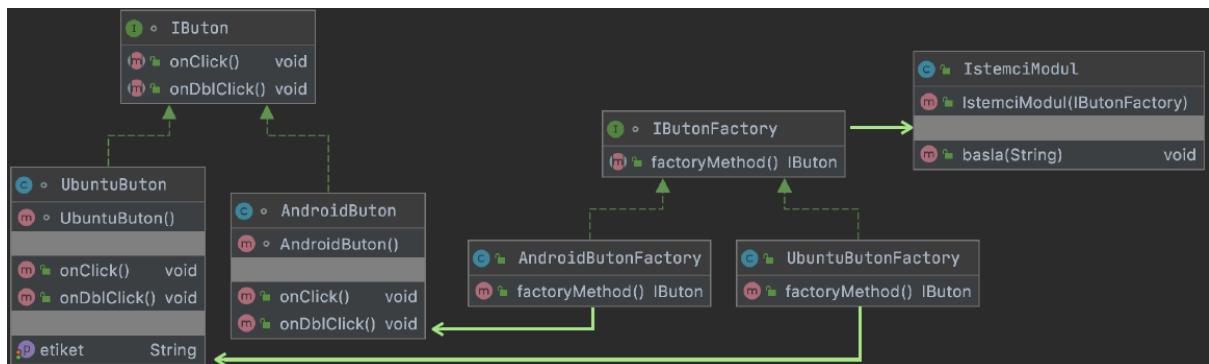
Nesne oluşturulması da bir iştir ve biz bu dikdörtgen nesnesi oluşturma işini factorymethod'a yaptıracağız. Yani başta IstemciModülde new Dikdortgen deyip dikdörtgen nesnesi oluşturacak olsak ve sonrasında dikdörtgene renk bilgisi de verilecek olursa istemci kodda da değişiklik yapılması gerekecektir ve aralarında bağımlılık olacaktır. Bunu bu şekilde engelliyoruz.

Örnekte Open Closed ilkesine bağlı olarak bağımlılığı azaltmak için daha sonradan daire nesnesi oluşturmak için DaireFactory sınıfı vs. de oluşturulabilir diye araya ISekilFactory eklenmiş.

Oluşan nesne standart nesne olur yani factoryMethod'u çağırırken herhangi bir parametre girmediğimizden dolayı her nesne oluşturmak için factoryMethod çağrıdığımızda factoryMethod içerisinde parametre olarak girilen değerlerle bir nesne oluşur.



Benzer başka bir örnek;



Strategy

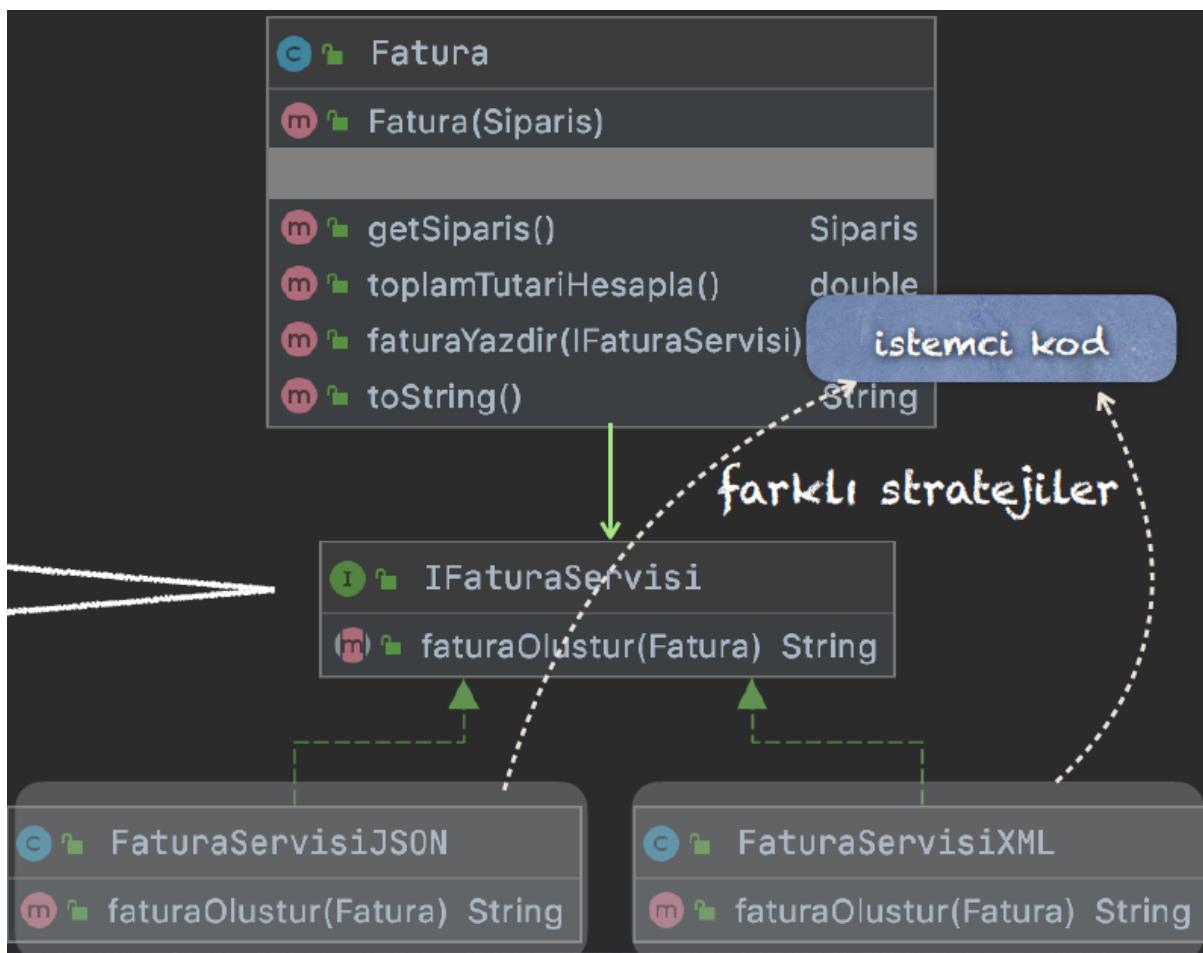
Aynı istemci kodun, farklı algoritmaları/stratejileri desteklemesini sağlar.

Örneğin; geliştirdiğiniz uygulama “bubble sort” algoritmasını kullanıyorken, istemci kodu

değiştirmeden, bu algoritma yerine “quick sort” algoritmasını çalıştırmasını sağlamak isterseniz, bu deseni kullanabilirsiniz.

Alternatif algoritmalarдан uygun olanının, çalışma zamanında seçilmesi gereken durumlarda kullanılabilir.

Farklı algoritmalar/stratejiler soyut bir modülden (arayüz) türetilir/gerçeklenir. İstemci kod içerisinde, bu algoritmalar yerine soyut modül kullanılır (program to interface...)



Yani bu örnek için main içerisinde `IFaturaServisi faturaServisi=new FaturaServisiXML();` dersek faturayı XML formatında, `IFaturaServisi faturaServisi=new FaturaServisiJSON();` dersek JSON formatında oluşturur.

Iterator

Veri toplulukları (collections) içerisinde elementler (nesneler) bulunur ve bu elementler çeşitli veri yapıları (dizi, bağlı liste, ağaç, graf vb.) kullanılarak bir arada tutulur.

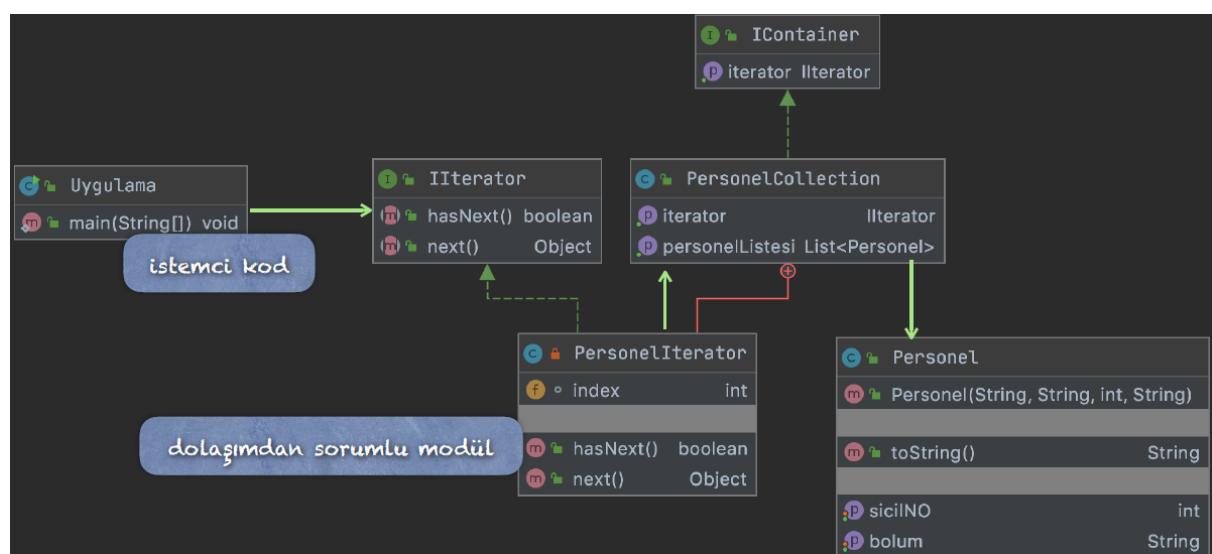
Veri toplulukları içerisindeki her bir elemente erişilmesi gereklidir. Bu işlem dolaşım (traversal) denir.

Veri toplulukları için kullanılan veri yapıları basit olduğunda (dizi, liste vb.) dolaşım için kullanılacak algoritma basit bir şekilde gerçekleştirilebilir.

Kullanılan veri yapıları karmaşıklaşıkça dolaşım algoritmaları zorlaşıbilir ve zaman zaman farklı dolaşım algoritmalarına (ağaç veri yapısı için preorder, postorder, inorder gibi) ihtiyaç duyulabilir.

Iterator deseni, istemci modülün, veri topluluğu içerisinde kullanılacak dolaşım algoritmalarından etkilenmesini önlemek için, bu işlemi (sorumluluk) başka bir nesnenin (iterator) yapmasını ister (SRP gereği).

Bu dolaşım işini biz istersek istemci kodda bir değişiklik yapmadan bu işi yapan PersonelCollection sınıfının inner sınıfı olan PersonelIterator sınıfındaki metodlarda değişiklik yaparak örneğin level-order'dan pre-order vs. değiştirebiliriz. Elemanları gezme işini bu PersonelIterator sınıfı üstleniyor.



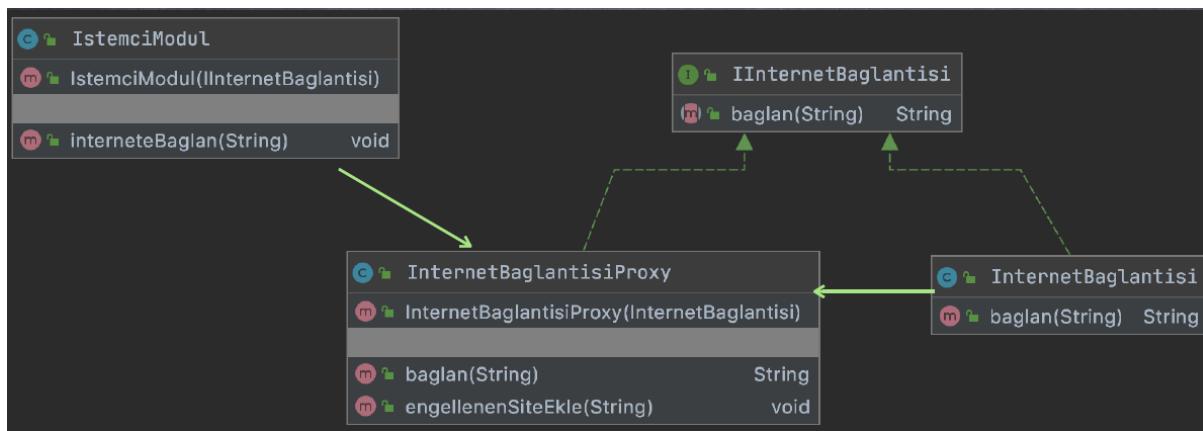
Proxy

Sınıfın fonksiyonlarını (arayüzü) değiştirmeden davranışını değiştirmek istediğimizde kullanabiliriz.

Sınıfı değiştirmek, onu kullanan diğer sınıfların etkilenmesine neden olabilir.

Örneğin bu diyagram için aynı internet bağlantısı için eğer engellenen bir siteye erişim sağlamak istiyorsak Proxy sayesinde farklı davranışarak erişmemizi engelleyebiliyor.

Not: InternetBaglantisi'ndan InternetBaglantisiProxy'ye giden ok tam tersi olacak.



Builder

Karmaşık nesnelerin (îçerisinde çok sayıda üye değişken ve üye nesne olan) oluşturulması için kullanılır.

Karmaşık bir nesnenin yapımını, temsilinden (sunumundan) ayırır. Böylece, aynı yapım süreci farklı temsiller oluşturabilir.

Nesnelerin farklı temsillerinin (sunumlarının) her biri için ayrı ayrı yapıcı tanımlamak yerine, nesne oluşturma işini adım adım gerçekleştiren “builder” deseni kullanılabilir.

Böylece nesne oluşturma işi nesnenin kendisinden ayrılmış olur (SRP).

Nesne oluşturma işlemi istemci koddan ayrılmış olur (SRP, loosely coupling)

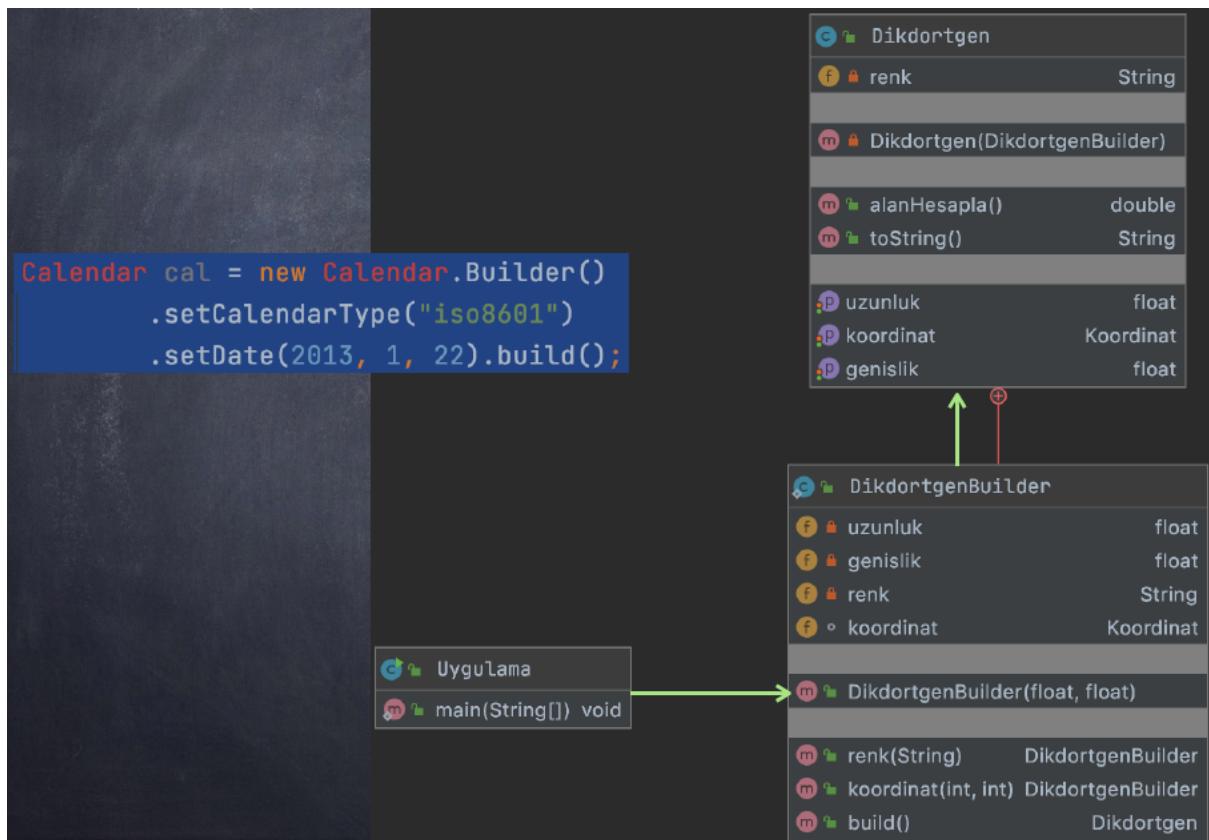
Yani örnek verecek olursak bir sınıf içerisinde 10 tane üye değişken varsa ve bu üye değişkenlerin atanması için 1'den fazla durumu değerlendirek 1'den fazla farklı sayıda parametre alacak kurucular oluşturma gibi durumlarda bu desen kullanılır. Yani builder'ın görevi nesneleri oluşturmak olacak, bu deseni kullandığımız sınıfı herhangi bir kurucu olmayacak, builder bu işi halledecek.

Builder sınıfı nesnenin tüm üye değişkenlerini/nesnelerini içermelidir.

Nesne oluşturmak gereki̇inde, builder sınıfının (**static olmalı**) nesnenin ilgili özelliklerine ilk değer

ataması yapan yöntemleri sırasıyla çağrırlar.

Java'da yalnızca inner yani sınıf içerisindeki sınıflar static olarak tanımlanabilir.



Çağrılan son yöntem (build) Dikdörtgen nesnesini oluşturur. Nesne oluşturmak için, nesnenin varsayılan yapıcısına, nesnenin ilgili üyelerinin değerlerini içeren DikdortgenBuilder sınıfı gönderilir.

Bu örnek için oluşturulan builder yapısı dikdörtgenin uzunluk ve genişlik değişkenlerinin kurucuya zorunlu olarak gönderilmesi fakat renk ve koordinat bilgilerinin opsyonel olacağı şekilde tasarlanmıştır. Bu durum da opsyonel olan değişkenler için metotlar oluşturularak sağlanıyor ve bu metotları builder yapıcısı sonrası ":" larla birleştirip sonunda ise .build() dediğimizde nesne oluşmuş luyor. Bu şekilde nesne oluşturma örneği;

```

//Dikdortgen dikdortgen1 = new Dikdortgen(10,12,"mavi",100,20);
Dikdortgen dikdortgen1 = new Dikdortgen.DikdortgenBuilder(10,12)
    .renk("mavi")
    .koordinat(100,20)
    .build();

//Dikdortgen dikdortgen2 = new Dikdortgen(10,12,"yeşil");
Dikdortgen dikdortgen2 = new Dikdortgen.DikdortgenBuilder(100,200)
    .renk("yeşil")
    .build();

//Dikdortgen dikdortgen4 = new Dikdortgen(10,12,100,20);
Dikdortgen dikdortgen4 = new Dikdortgen.DikdortgenBuilder(20,100)
    .build();

```

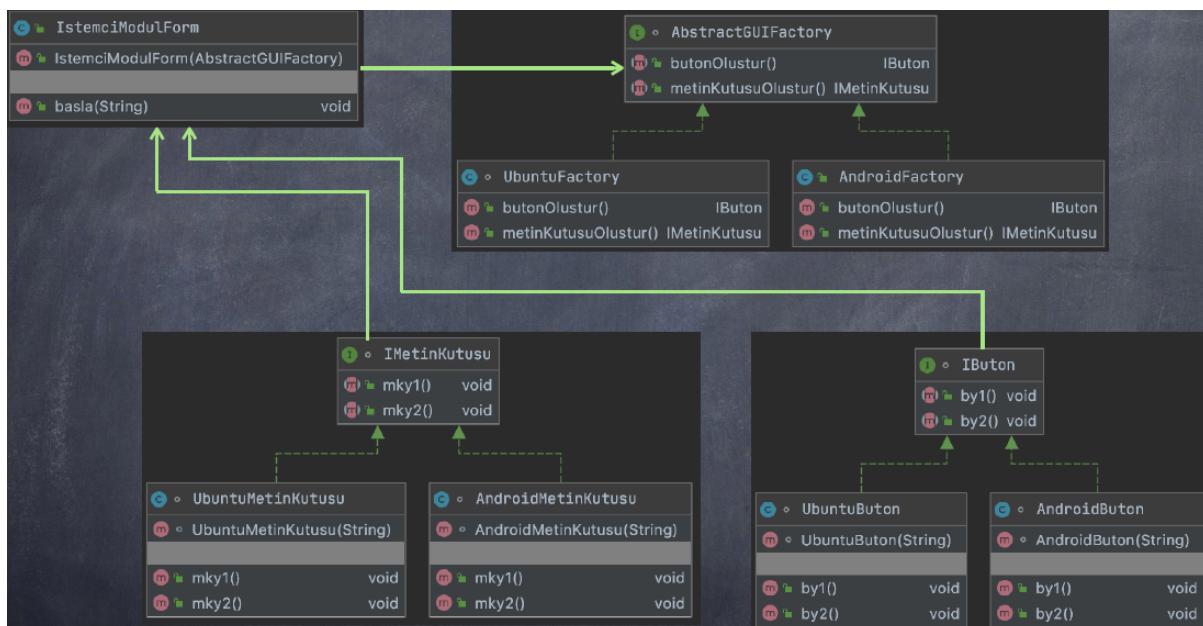
Abstract Factory

İstemci modül içerisinde, somut sınıflarını belirtmeden, birbirleriyle ilgili ya da birbirlerine bağlı nesne aileleri/grupları oluşturmak için kullanılabilir.

Örneğin; bir geliştirme ortamının görünümünü değiştirmek istediğimizde (dark theme gibi) ya da bir yazılımın farklı platformlarda sorunsuz çalışabilmesini istediğimizde, bu deseni kullanabiliriz.

Factory Method'un gelişmiş halidir.

Örneğin bu diyagram için main içerisinde bilgisayarın config dosyasından işletim sistemi türünü okuyup işletim sistemi adına göre o işletim sisteminin Factory'sinden bir nesne oluşturup bu nesneyi de IstemciModulForm nesnesi oluşturup parametre olarak verdiğimizde o kendi içerisinde bu işletim sistemi ailesine bağlı olan MetinKutusu ve Buton'unu oluşturacaktır.



Prototype

Yeni nesne oluşturma işleminin maliyetli olduğu durumlarda, bu desen kullanılarak, mevcut nesnenin kopyası oluşturulabilir.

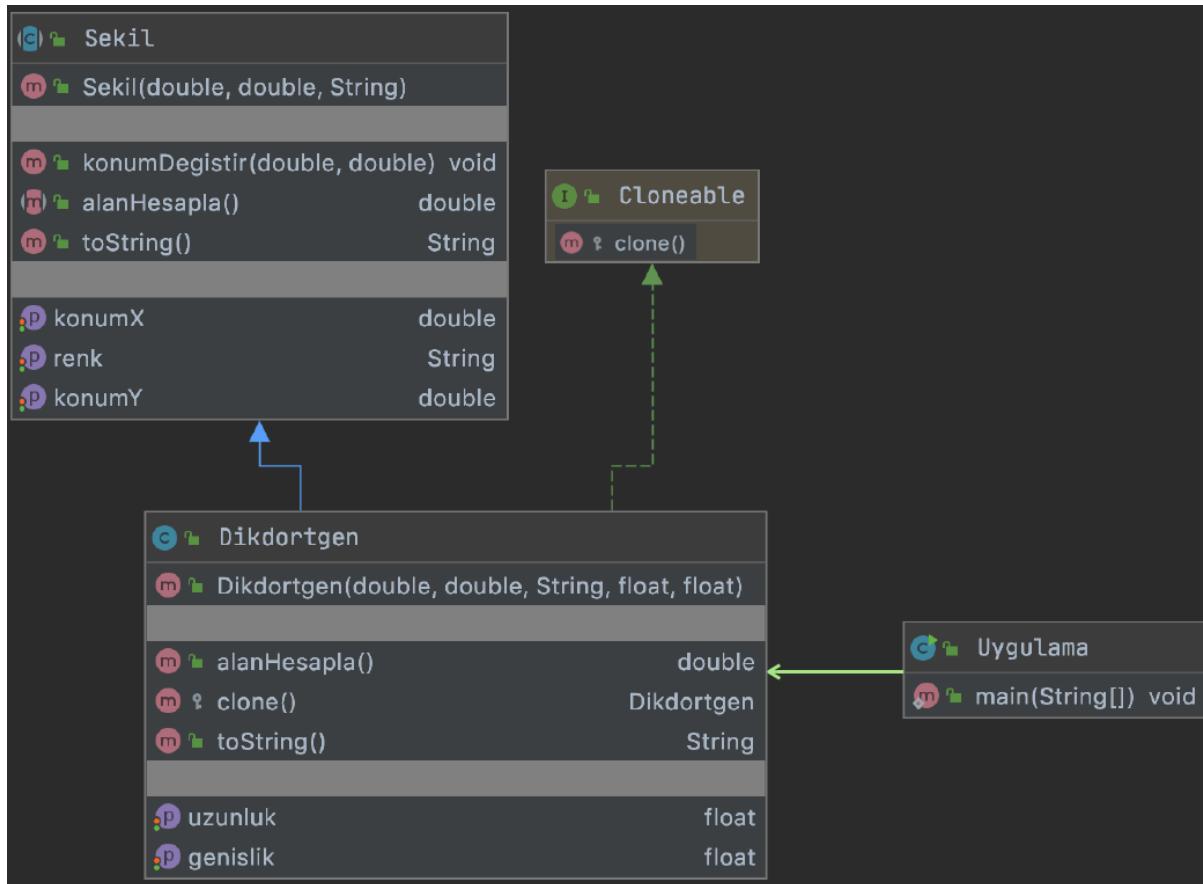
Karmaşık işlemler sonucu oluşan nesneler (örneğin, veritabanı sorguları ya da diğer sistemlerden gelecek verilerle oluşturulacak nesneler) "cache" içerisinde saklanır ve bu nesnelere ihtiyaç duyulduğunda, önce "cache" içerisinde aranır. Bulunursa kopyası oluşturulur ve veritabanı sorgusuna gerek kalmaz.

Nesneler, prototype (cloneable) arayüzü içerisinde tanımlı `clone()` yöntemini gerçekleyerek, (`new` komutu ile) kendi kopyalarını oluştururlar.

Örneğin bu diyagram için Sekil sınıfından ve Java'ya ait Cloneable arayüzünden kalıtım alan Dikdortgen sınıfı içerisinde bir `clone` metodu var ve bu metot bu方法 çağırılan

dikdörtgen nesnesinin aynı özelliklerine sahip yeni bir dikdörtgen nesnesi döndürüyor yani onun bir klonunu yaratıyor. Örn koddan görüleceği üzere dikdortgen2, dikdortgen1'in aynısı oldu. Yani bir nesneyi en baştan oluşturmuyoruz, varolan bir nesneyi kopyalıyoruz. Dolayısıyla yeni bir nesne oluşturma maliyetinden kurtulmuş olduk.

```
Dikdortgen dikdortgen1=new Dikdortgen(100,200,"sarı",3,4);
Dikdortgen dikdortgen2=dikdortgen1.clone();
```



Burada mühim olan şey eğer clone'unu oluşturduğumuz nesne içerisinde başka sınıfa ait nesneler de mevcutsa şayet, bu durumda biz klonlama işlemi yaptığımızda bizim nesnemizi klonlar ve bu yeni bir ramde başka bir yeri gösteren nesne olur fakat klonlanılan nesnenin içerisinde klonlanılan nesnenin gösterdiği diğer bir nesnelerin gösterdikleri yeri aynen geçirir yani klonlama işlemi yaparken bizim için o nesnenin içindeki diğer sınıflara ait nesneleri yeniden oluşturmaz, klonlanılan nesnenin gösterdiği nesneleri gösterir. Buna Shallow Copy denir. Eğer biz bu tarz durumlar için içerisindeki nesneleri de yeniden oluşturmak istiyorsak o nesnelere ait sınıflardan da Cloneable arayüzünden kalıtım alıp clone metodunu oluşturup, asıl klonlayacağımız nesnenin sınıfının clone metodunda ise bu nesneleri de klonlamalıyız ve son olarak nesneyi döndürmeden önce set metodlarıyla bu yeni nesneleri yeni oluşturduğumuz nesneye veririz. Buna da Deep Copy denir.

Tasarım Kalıplarını anlatan iyi bir YouTube kanalı;

<https://www.youtube.com/c/SadıkBahadırMemiş>

Bu kanaldaki yazılan kodlar;

<https://github.com/sbahadirm/TasarimKaliplari/tree/master/src/main/java/designpatterns>