Java ile Tasarım Prensipleri ve Tasarım Örüntüleri

İyi tasarımın amacı nedir? Neden iyi bir tasarıma sahip olmak isteriz? İyi tasarımın amacı ileride karşımıza çıkacak olası değişiklikleri en kolay şekilde ele almamızı sağlamaktır.

Peki, herhangi bir yazılım sisteminde neler değişir? Gereksinimler...Gereksinimlerin değişmesinin pek çok değişik nedeni vardır. Projenin başında tam manası ile tespit edilememiş olabilirler. Eksik, yanlış veya yanıltıcı ifade edilebilirler. Sistemin geliştirilmesi süresince ve sistem kullanıma alındıktan sonra da sürekli olarak değişirler. Bu değişikler ekip üyelerinin ve kullanıcıların yeni olasılıkları görmelerinden, geliştircilerin sistemi kavrayışlarının değişmesinden, yazılım sisteminin geliştirildiği ortamın veya iş süreçlerinin değişmesinden kaynaklanabilir. Ancak, gereksinimlerin değişmesinden şikayet etmek anlamsızdır. Yapılması gereken gereksinimlerdeki değişmelerin önüne geçmekten çok, değişiklikleri daha kolay biçimde ele almamızı sağlayacak bir sistem tasarımına sahip olmaya çalışmaktır.

Bu noktada da aklımıza şöyle bir soru gelebilir. Peki, iyi tasarımı, kötü tasarımdan nasıl ayırt edebiliriz? Neyin iyi bir tasarım kararı olduğunu, hangi kararın ise ileride karşımıza çıkabilecek muhtemel değişiklikleri ele almamızı kolaylaştıracağını veya zorlaştıracağını önceden nasıl anlayabiliriz?

Kötü Tasarımın Belirtileri

Kötü tasarımın dört temel belirtisi vardır. Bu belirtiler birbirleri ile bağlantılıdır ve kötü bir tasarımın ve mimarinin işaretleridir.

1. Rigidity (Esnemezlik)
2. Fragility (Kırılganlık)
3. Immobility (Taşınamamazlık)
4. Viscosity (Akışkanlık)

Şimdi bu dört temel kötü tasarım belirtisine daha detaylı bakalım.

Rigidity (Esnemezlik)

Bu problemi barındıran sistemlerde değişiklik yapmak çok zordur. En basit değişiklikler bile pek çok başka değişiklikleri gerektirebilir. Bir gün süreceği düşünülen bir iş, haftalar alabilir. İşin ne zaman biteceği kestirilemez. Bu nedenle kritiklik arz etmeyen problemlerin düzeltilmesi sürekli olarak ertelenir. Bu durum sistemde bir süre sonra kırık pencereler sendromuna yol açacaktır. Tasarımsal problem, yönetimsel bir hal alır.

Fragility (Kırılganlık)

Bir önceki problem ile yakından alakalıdır. Bir değişikliğin, ilgili ilgisiz sistemin diğer pek çok başka yerinde problemlere yol açmasıdır. Çoğunlukla da bu yerlerin asıl değişikliğin yapıldığı yerle doğrudan bir bağlantısı yoktur, yada ekip üyeleri böyle düşünmektedir. Sonuç olarak, ekip üyelerinde bir süre sonra yapılacak her yeni değişiklikte çalışan başka bir yeri bozma korkusu ortaya çıkar. Bakım neredeyse imkansız hale gelir. Her fix, problem hanesine yeni bug'ların eklenmesine neden olur.

Immobility (Taşınamazlık)

Kısacası herhangi bir yazılım modülünün veya daha önce ki bir proje için geliştirilmiş bir çözümün yeniden kullanılamamasıdır. Yazılım projelerinde genellikle daha önceden yapılmış bir takım çalışmaların, önceki projelerde geliştirilmiş modüllerin veya kütüphanelerin yeni projede de kullanılabileceği ön görülür. Ancak bu modül veya kütüphanenin yeni proje ile alakalı olmayan diğer bir takım modüllere veya kütüphanelere bağımlı olduğu tespit edilir. Bir süre sonra yazılım geliştiriciler modülü yeniden kullanma fikrinden vazgeçip, sıfırdan yazmaya karar verirler.

Viscosity (Akışkanlık)

Herhangi bir yazılım problemi için çoğunlukla birden fazla çözüm üretilebilir. Bu çözümlerden bazıları sistemin genel tasarımına ve mimarisine uygun iken, diğer bazıları ise kestirme yol veya “hack” olarak tabir edilebilirler. Eğer tasarıma uygun çözümlerin hayata geçirilmesi, uygulanması zor ve zaman alıyor, geliştiriciler daha çok hack diye tabir edilen çözümlere yöneliyor ise böyle bir

sistemde tasarımsal viscosity problemi var demektir. Tasarımsal viscosity probleminin yüksek olduğu sistemlerde yanlışı yapmak daha kolaydır.

Diğer bir tür viscosity problemi ise ortamsal viscosity'dir. Örneğin, bir projenin geliştirme ortamında derleme ve checkin zamanları çok uzun olabilir. Böyle bir durumda mühendisler uygun olmasa bile derleme ve checkin gerektirmeyecek çözüm yollarını tercih edeceklerdir.

Nesne Yönelimli Analiz ve Tasarım Nasıl Olmalı?

Muhtemel değişikliklere karşı esnek ve uzun ömürlü bir yazılım tasarımı ve mimarisi oluşturmada nesne yönelimli analiz ve tasarım yöntemleri önemli rol oynamaktadır. Değişikliklere kolay adapte olabilen, esnek ve fonksiyonel bir nesne yönelimli tasarım için öncelikle probleme olan bakış açımızı değiştirmeliyiz.

Nesne yönelimli programlama öğretilirken çoğunlukla nesne = veri + metot benzetmesi yapılır. C, Pascal gibi yapısal dilleri bilen öğrencilerin yeni kavramları anlaması için bu benzetmeler bir açıdan öğrenme sürecini kolaylaştırsa da, nesne yönelimli programlama bakış açısının sağlıklı temeller üzerine bina edilmemesine de neden olmaktadır. Nesne = veri + metot gözlüğü ile nesne dünyasına bakan bir yazılım geliştiricinin elinden çıkan bir tasarımda nesneler, herhangi bir akıllı davranış sergileme kabiliyetinden yoksun güdük veri yapılarına benzemektedirler. Klasik yaklaşım olarak da tanımlanan bu bakış açısı Martin Fowler'in UML Distilled kitabında “implementation” perspektifi olarak tanımlanmaktadır. Yapısal programlama ve problem çözme bakış açılarından arındırılmış modernist yaklaşım da ise ile nesneler sorumlulukları olan ve belirli bir davranış sergileyen olgulardır. Bu yaklaşımda tasarımcı problemi analiz etme safhasında nesnenin içerisinde ne olduğu ile ilgilenmez. Nesne yönelimli sistem olgularla “kavramsal” düzeyde iletişimde olmalıdır. Başka bir deyişle nesnelere ne yapmaları gerektiği söylenmelidir, nasıl yapmaları değil. Bir olgunun herhangi bir işi gerçekleştirmesi kendine özel bir işlemdir. Bu işlemlerin nasıl gerçekleştirileceği “implementation” aşamasında nesnenin içerisinde halledilecek bir konudur. Dış dünyanın bunun nasıl yapıldığı ile ilgili bilgi sahibi olmasına gerek yoktur.

Problemden Çözüme Nasıl Gidilir?

Herhangi bir yazılım problemini ele aldığımızda gerçek dünyadaki problemden, yazılım sistemi olarak ifade edilecek çözüme nasıl gidileceği, hangi yol ve yordamların kullanılacağı da önemli bir konudur. Nesne yönelimli analiz, tasarım ve kodlama süreçleri sistematik bir yol ile gerçekleştirilmelidir. Problem alanındaki olguları, olgular arasındaki ilişkileri tespit etmek için izlenebilecek çeşitli yollar mevcuttur. İsim-fiil analizi bunlardan birisidir. Kısaca kesin olmasa da genellikle isimler olguları, fiiller ise olgulardaki davranışları işaret ederler. Ancak gelecekte ortaya çıkacak yeni gereksinimleri esnek biçimde ele almayı sağlayacak bir nesne modele sahip olmak için tasarımın değişmeyen temel noktaları ile birlikte yeni ihtiyaçların ve değişikliklerin ele alınmasını sağlayacak genişleme (extension) noktalarını da tespit etmek önemlidir. Bunun için uygulanan yöntemlerden birisi de ortaklık ve değişkenlik analizidir.

Ortaklık/Değişkenlik Analizi

Bu yöntem ile problem alanındaki ortak yapılar ve olgular ile değişenler, varyasyonlar tespit edilir. Ortaklık analizi zaman içerisinde çok sık değişmeyecek kısımları tespit etmeyi sağlar. Bir bakıma nesne modelin iskeletini oluşturmaya yardımcı olur. Değişkenlik analizi ise sistem içerisinde sıklıkla değişecek noktaları arar. Biraz önce belirttiğimiz nesne modelin iskeletindeki hareketli, değişken noktaları belirlemeye yardımcı olur. Ortaklık ve değişkenlik analizleri birbirlerini tamamlayan süreçlerdir. Değişkenlik ancak mevcut bir ortaklık içerisinde anlam kazanır.

Mimarisel perspektiften bakılırsa, ortaklık analizi mimariye uzun ömürlülük katar. Değişkenlik analizi ise tasarıma kullanım kolaylığı getirir. Genel bir ifade ile nesne modeldeki soyut sınıflar ortaklık analizi ile tespit edilen olgulardır. Concrete sınıflar ise varyasyonlara karşılık gelir.

Nesne yönelimli tasarımda ortak ve değişken yapıları bir araya getirmede encapsulation kritik önem arz etmektedir. Klasik söylemde encapsulation sıklıkla verinin gizlenmesi olarak anlatılır. Bu yanlış bir ifade değildir. Ancak oldukça eksik bir ifadedir. Encapsulation herhangi tür birşeyin gizlenmesi için uygulanabilir. Encapsulation aynı zamanda concrete sınıfların soyut sınıflar ile gizlenmesidir. Değişen bir davranış da encapsule edilebilir.

Öncelikle tasarım içerisinde neyin değişken olduğu, neyin tekrar tasarım yapmadan değişebileceği tespit edilmelidir. Burada değişen olgu veya konsept üzerine odaklanılır ve encapsule edilir.

Altın Değerinde İki Tasarım Kuralı

Gelecekte karşımıza çıkabilecek muhtemel yeni gereksinimleri ve sistemle ilgili değişiklikleri kolay biçimde ele almamızı sağlayacak bir nesne yönelimli tasarım ortaya çıkarmak için iki kurala mümkün olduğunca uymak gerekir.

- **Kural 1:** Değişen ne var ise bul ve encapsule et.
- **Kural 2:** Composition'ı inheritance'a tercih et.

Geliştirme sürecinde analiz ve tasarım aşamalarında tespit edilen olguların sınıflara dönüştürülmesi noktasında izlenebilecek birtakım temel tasarım prensipleri de vardır.

Açıklık Kapalılık Prensibi (Open Closed Principle) (OCP)

Bir modül veya sınıf genişlemeye açık, değişikliklere kapalı olmalıdır. Nesne yönelimli programlamanın en temel prensibidir. İlk olarak Bertrand Meyer tarafından ortaya konulmuştur. Mümkün olduğunca modülleri extend edilebilir biçimde yazmalıyız. Modüller değişikliklere kapalı olmalıdırlar. OCP'nin anahtar kelimesi soyutlamadır.

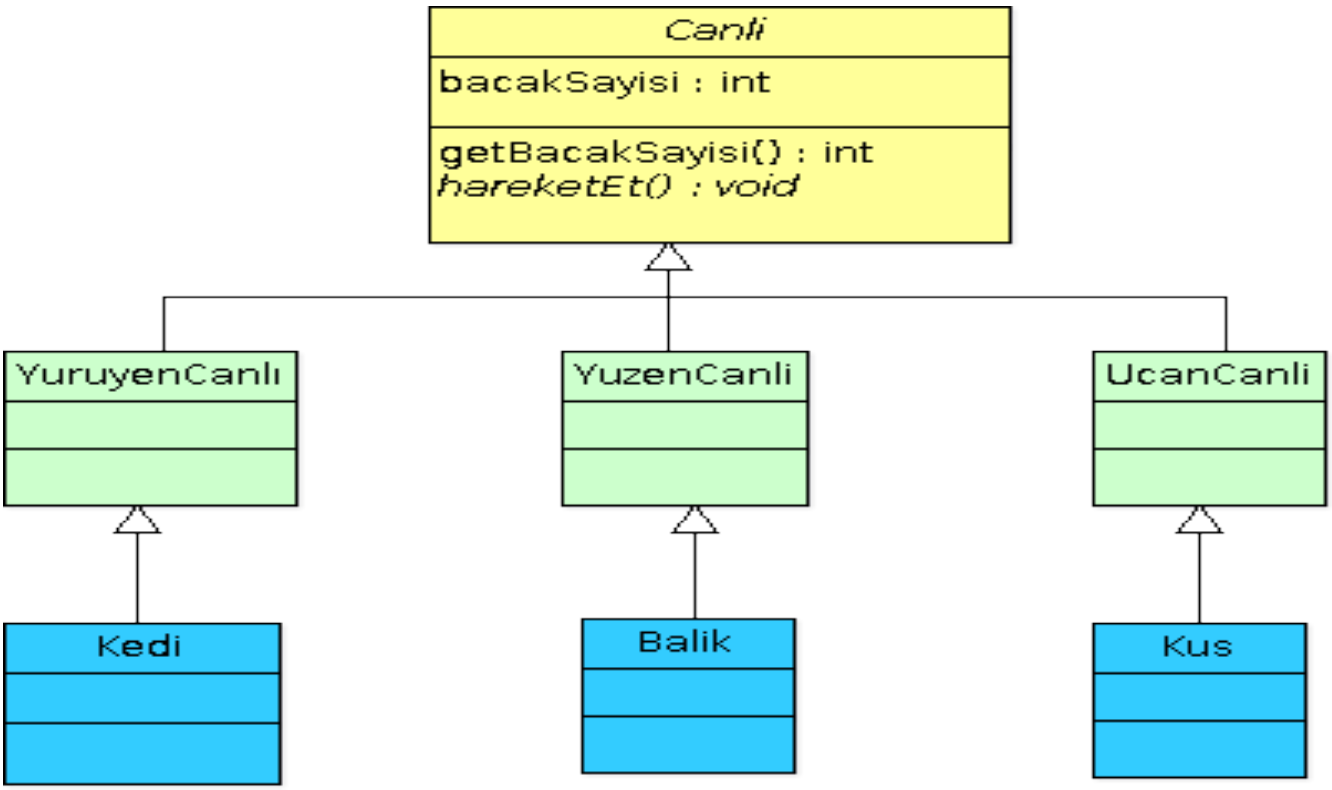
Tersine Bağımlılık Prensibi (Depedency Inversion Principle) (DIP)

Bu prensibe göre sınıflar sadece arayüzlere veya soyut sınıflara bağımlı olmalıdırlar. Mümkün olduğunca concrete sınıflara doğru bir bağımlılık olmamalıdır. Tasarımdaki bütün bağımlılıklar da soyut olgulara doğru olmalıdır. COM, CORBA, EJB gibi bileşen teknolojilerinin, Spring gibi framework'lerin dayandığı temel prensiptir.

Örnek Problem: Simülasyon Programı

Faz 1

"Doğadaki **canlılar** hareket kabiliyetlerini sahip oldukları **bacakları** vasıtası ile sağlamaktadır. Her türün **farklı sayıda** bacakları olabilir. Canlılar karada **yürüyebilir**, denizde **yüzebilir**, havada ise **uçabilir**ler. Farklı canlı türlerinin **hareket şekilleri**ni modelleyen bir **simülasyon programı** yazılması istenmektedir. Simülasyon programında farklı canlı türlerini temsil etmek için **kedi**, **kuş** ve **balık** türleri kullanılabilir."



Şekil 1

Yukarıdaki probleme benzer pek çok problemde çoğu yazılımcı klasik yaklaşımda bacakSayisini doğrudan encapsule eden ve hareket şekline karşılık gelen soyut bir metoda sahip bir Canli sınıfı oluşturarak işe başlar. Çoğunlukla da sınıflar arasındaki hiyeyarşiler 3-4 ara sınıf içeren derin yapılar halini alır. (Şekil-1)

Bizim tasarımımızda da Canli soyut sınıfından türeyen YuruyenCanli, YuzenCanli, UcanCanli şeklinde ara sınıflar tanımlanmış, bu ara sınıflarda da farklı hareket şekilleri kodlanmıştır. Kedi, Kus ve Balık sınıflarının constructor'larında ise bacak sayıları farklı değerlerle initialize edilmiştir. (Kod-1, Kod-2)

```
public abstract class Canli {

   private int bacakSayisi;

   public int getBacakSayisi() {
      return bacakSayisi;
   }

   public void setBacakSayisi(int bacakSayisi) {
      this.bacakSayisi = bacakSayisi;
   }

   public abstract void hareketEt();

}
```

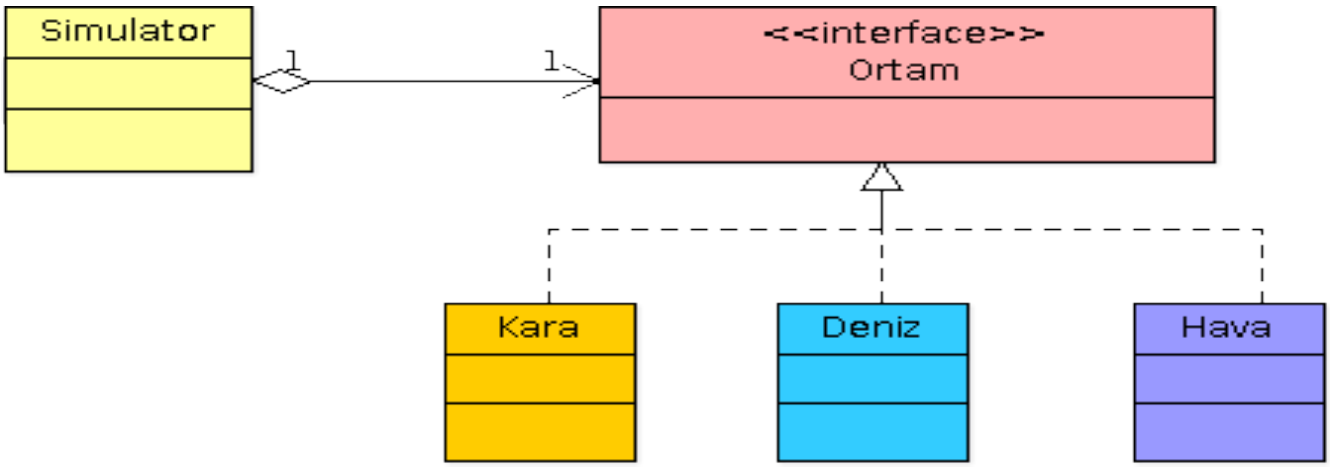
Kod 1

```
public class YuzenCanli extends Canli {

   @Override
   public void hareketEt() {
      System.out.println("yüzüyor...");
   }

}
```

```
public class Balik extends YuzenCanli {
   public Balik() {
      setBacakSayisi(0);
   }
}
```

Kod 2

Ortam bilgisi ise hemen bir arayüz ile ifade edilmiştir, ancak içerisinde herhangi bir davranış söz konusu değildir. Bir nevi Ortam nesnesi sadece belirli bir durumu işaret eden veri yapısı rolündedir. (Şekil-2, Kod-3)



Şekil 2

```
public interface Ortam {

}
```

```
public class Deniz implements Ortam {

}
```

Kod 3

Simulator sınıfı ise temel olarak hareketEttir metodunda kendisine input argüman olarak verilen bir grup Canli nesneyi, hareketEt metotlarını çağırarak hareket ettirmektedir. (Kod-4)

```
public class Simulator {
   private Ortam ortam;

   public Ortam getOrtam() {
      return ortam;
   }

   public void setOrtam(Ortam ortam) {
      this.ortam = ortam;
   }

   public void hareketEttir(Canli...canlilar) {
      for(Canli c:canlilar) {
         c.hareketEt();
      }
   }
}
```

Kod 4

Son olarak, test amaçlı yazdığımız Main sınıfı da, main metodunda Simulator nesnesine üç farklı türde Canli nesnesi oluşturarak simülasyon tstini gerçekleştirmektedir. (Kod-5)

```
public class Main {

public static void main(String[] args) {
   Simulator simulator = new Simulator();

   simulator.hareketEttir(new Kedi(),new
Kus(),new Balik());
   }

}
```

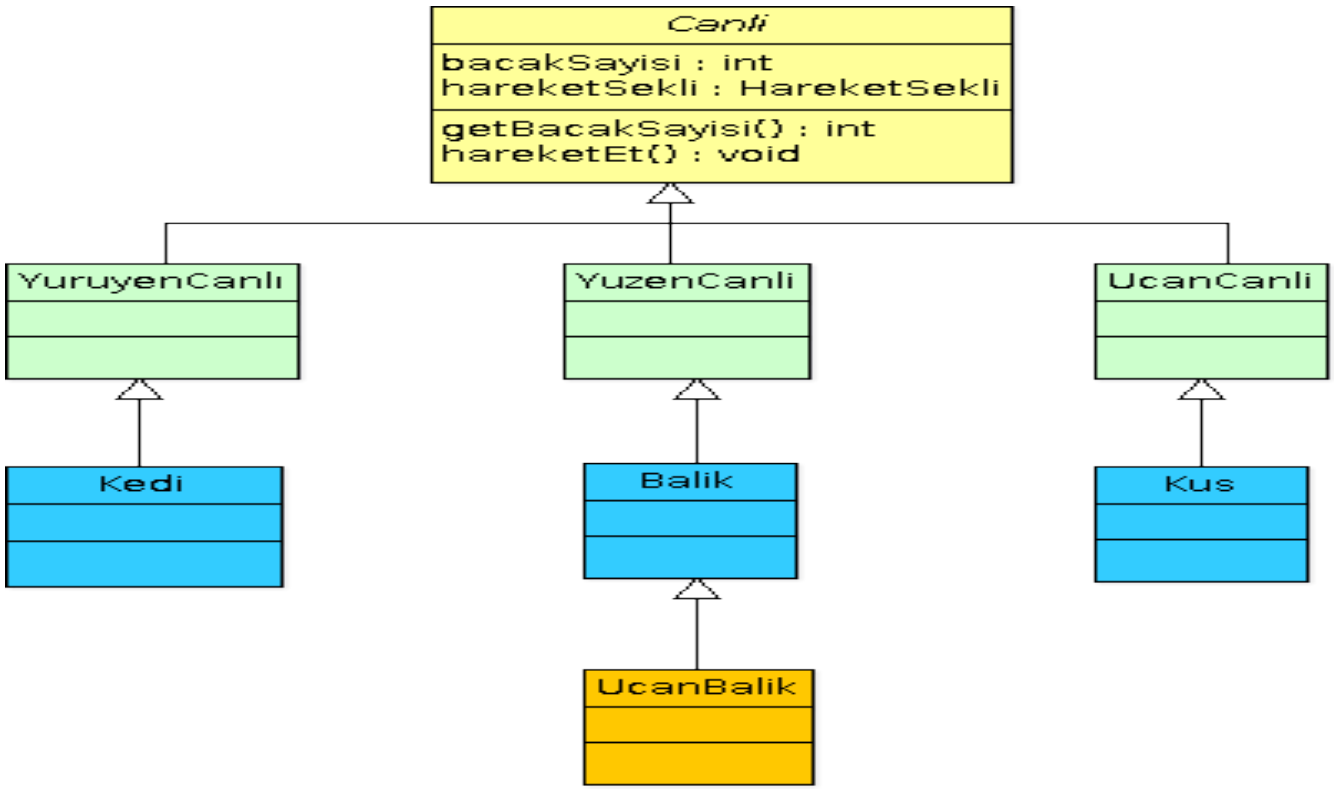
```
yürüyor...
uçuyor...
yüzüyor...
```

Kod 5

Şimdi bu tasarımın gelecekte karşımıza çıkacak yeni gereksinimleri ne kadar kolay ele alıp alamayacağını örneğin devamına bakarak inceleyelim.

Faz 2

"Bazı canlılar tek bir hareket şekline sahip iken, diğer bazıları ise **farklı ortamlarda farklı hareket şekilleri**ne sahip olabilirler. Örneğin, kuşlar karada yürüme, havada ise uçma kabiliyetine sahiptirler. Farklı bir balık türü ise denizde yüzebilirken, belirli süre deniz yüzeyinin üzerinden havalanarak uçabilmektedir. Simülasyon programı, canlı türün **hareket şeklinin ortama göre değişiklik göstermesi**ni de desteklemelidir."



Şekil 3

Örneğin, farklı bir balık türü denizde yüzerken, zaman zaman deniz dışında havalanarak birkaç metre uçabilmektedir. Sistemimizin bu tür balıkların ortama göre farklı davranış şekillerini destekleyecek biçimde genişletilmesi gerekmektedir. Bu noktada ilk yapacağımız şey muhtemelen sınıf hiyerarşimize yeni bir alt sınıf ekleyerek bu davranış farklılığını alt sınıfta ele almak şeklinde olacaktır. (Şekil-3)

```
public class UcanBalik extends Balik {
    private boolean uc = false;

    public boolean isUc() {
        return uc;
    }

    public void setUc(boolean uc) {
        this.uc = uc;
    }

    @Override
    public void hareketEt() {
        if(uc) {
            System.out.println("uçuyor");
        } else {
            super.hareketEt();
        }
    }
}
```

If-else ifadesi bir algoritmik Varyasyon işaretçisidir

Kod 6

Yukarıda da görüldüğü gibi UcanBalik sınıfında hareketEt metodu "override" edilecek ve dışarından set edilecek bir "flag" yardımı ile balığın yüzme veya uçma davranışlarından herhangi birisini sergilemesi sağlanacaktır. Genel bir kötü tasarım sinyali olarak if-else ifadeleri, kod içerisinde düzgün biçimde ele alınmayan bir takım "algoritmik varyasyonların" işaretçisidir. (Kod-6) Burada da flag değerine göre iki farklı algoritmadan uygun olanının tetiklenmesi söz konusudur. Tasarımımızdaki bir diğer problem de bu balık türünün günün birinde karada da yürümesi söz konusu olursa, mevcut kodumuzun içerisinde bu durumu da ele almak için bir değişiklik yapmamız gerekecektir. Bu da biraz evvel bahsettiğimiz temel nesne yönelimli tasarım prensiplerinden "açıklık kapalılık prensibi"nin ihlali demek olacaktır. Eğer kod içerisinde herhangi bir değişiklik yapıyorsak, yeni davranışı implement ederken mevcut davranışlarla ilgili "bug"lara yol açma ihtimalimiz de her zaman için söz konusudur. O nedenle tercih edilen, yeni davranışların sisteme "kod modifikasyonu" yolu ile değil, mevcut koda dokunmayarak "extension" yöntemi ile kazandırılmasıdır.

Simulator sınıfının hareketEttir metodunda ise Ortam tipinin ve Canli türünün kontrolü gerekmektedir. Ayrıca UcanBalik nesnesine uçması gerektiği flag değeri set edilerek söylenmektedir. Bunun için de Canli nesnesinin UcanBalik alt sınıfına dönüştürülmesi (downcast) şarttır. Downcast işlemleri de kod içerisinde çoğu zaman için OCP ve DIP prensiplerinin ihlalinin işaretçisidir. Burada hepimizin gördüğü gibi hem açıklık kapalılık (OCP), hem de tersine bağımlılık (DIP) prensipleri ihlal edilmektedir. DIP'e göre bağımlılıklar sadece soyut sınıflara veya arayüzlere doğru olmalıdır. Oysa Simulator sınıfı ilk başta bu kurala uyarken, yeni gereksinimleri karşılayabilmek için concrete Hava ve UcanBalik sınıflarına bağımlı hale gelmiştir. İleride ortaya çıkabilecek farklı yeni türler için benzer davranışlar ilave etmek için yeni değişiklikler yapmamız gerekiği açıktır. (Kod-7)

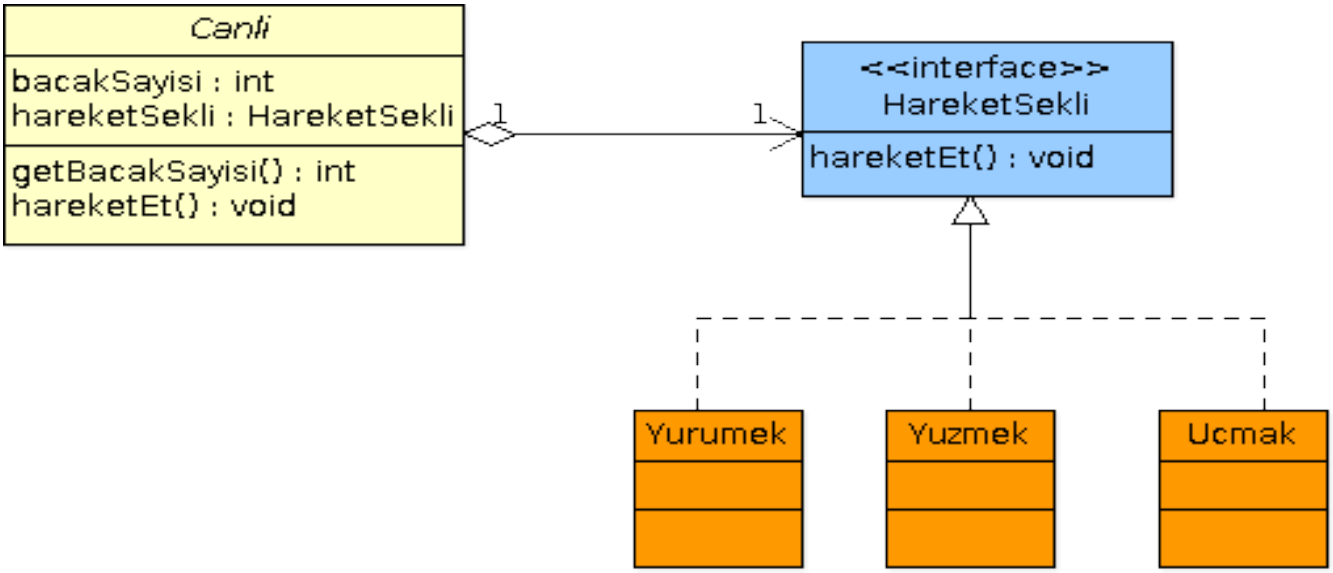
Bu noktada tasarımımızı daha önce bahsettiğimiz iki temel kuralı; "değişen ne ise bul ve encapsule et" , "composition'ı inheritance'a tercih et" ve OCP ile DIP prensiplerini göz önüne alarak sil baştan ele alalım.

```
public class Simulator {
    private Ortam ortam;

    public Ortam getOrtam() {
        return ortam;
    }

    public void setOrtam(Ortam ortam) {
        this.ortam = ortam;
    }

    public void hareketEttir(Canli...canlilar) {
        for(Canli c:canlilar) {
            if(ortam instanceof Hava && c instanceof
UcanBalik) {
                ((UcanBalik)c).setUc(true);
            }
            c.hareketEt();
        }
    }
}
```

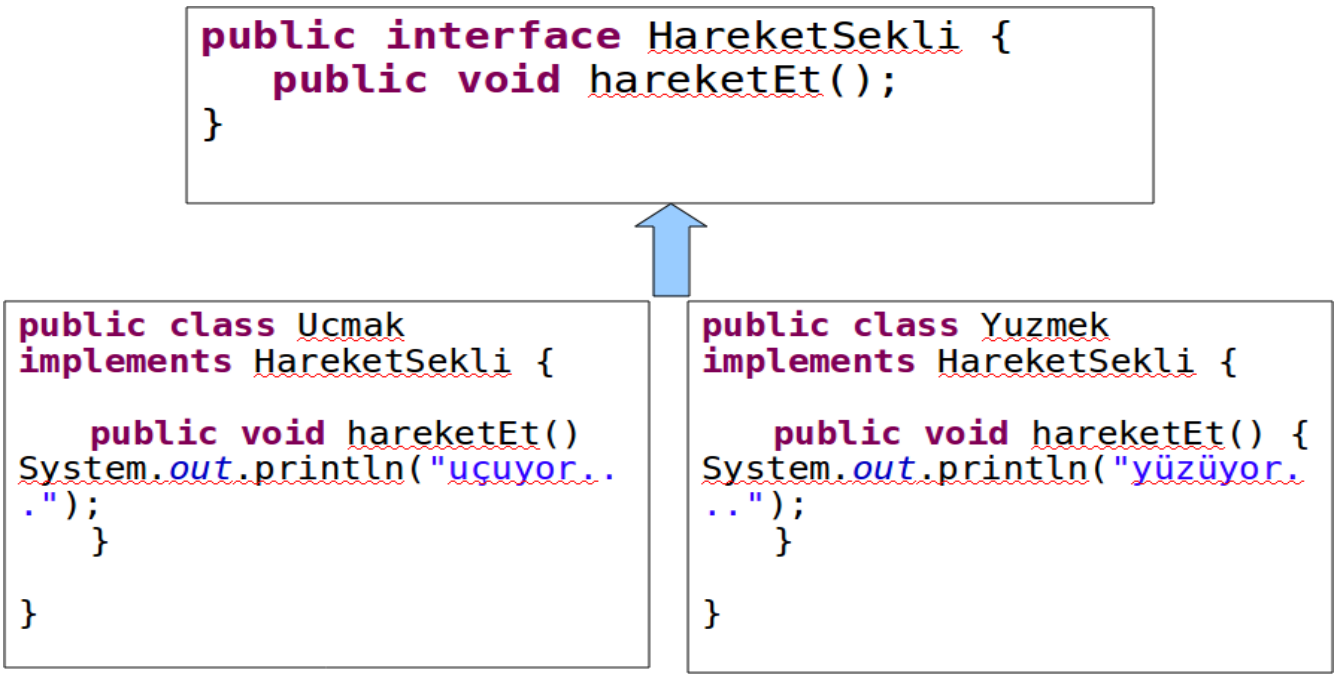
OCP, DIP prensipleri ihlal ediliyor

Kod 7

Faz 2'de, "...farklı ortamlarda farklı hareket şekillerine sahip olabilirler. ...canlı türün hareket şeklinin ortama göre değişiklik göstermesini de desteklemelidir" ifadeleri bize "hareket şekli"nin değiştiğini ve aynı zamanda hareket şeklinin belirlenmesinde ortamın da rol oynadığını anlatmaktadır. Ortaklık/değişkenlik analizine göre, canlılar için ortak olan şey hareket etmeleri iken, değişen şey ise bu hareketin şeklidir. Bu durumda "hareket şekli" encapsulation'a tabi tutulmalıdır. (Şekil-4)



Şekil 4



```
public interface HareketSekli {
   public void hareketEt();
}
```

```
public class Ucmak
implements HareketSekli {

   public void hareketEt()
System.out.println("uçuyor..
.");
   }

}
```

```
public class Yuzmek
implements HareketSekli {

   public void hareketEt() {
System.out.println("yüzüyor.
..");
   }

}
```

Kod 8

Bunun için HareketSekli arayüzü oluşturarak, hareketEt metodu tanımlayabiliriz. HareketSekli arayüzü Yurumek, Yuzmek ve Ucmak gibi farklı sınıflar tarafından implement edilir. Bu sınıflar farklı davranış şekillerine, başka bir ifade ile algoritmik varyasyonlara karşılık gelmektedir. (Kod-8)

Canli sınıfı ise içerisinde HareketSekli değişkeni barındırmakta ve kendi hareketEt metodunda ise asıl işi o andaki HareketSekli nesnesine havale etmektedir. (Kod-9)

```
public abstract class Canli {

...

private HareketSekli hareketSekli;

    public void hareketEt() {
        hareketSekli.hareketEt();
    }

    public void setHareketSekli(HareketSekli
hareketSekli) {
        this.hareketSekli = hareketSekli;
    }
}
```

Davranışın Encapsule Edilmesi

Kod 9

Bu noktada YuzenCanli ve UcanBalik sınıflarına baktığımızda, üst sınıfta hareket şeklinin yüzmek olarak belirlendiği, alt sınıfta ise flag değerine göre mevcut ortamın yada Ucmak nesnesinin hareket şeklinin kullanıldığı görülmektedir. (Kod-10)

```
public class YuzenCanli extends Canli {
    public YuzenCanli() {
        setHareketSekli(new Yuzmek());
    }
}
```

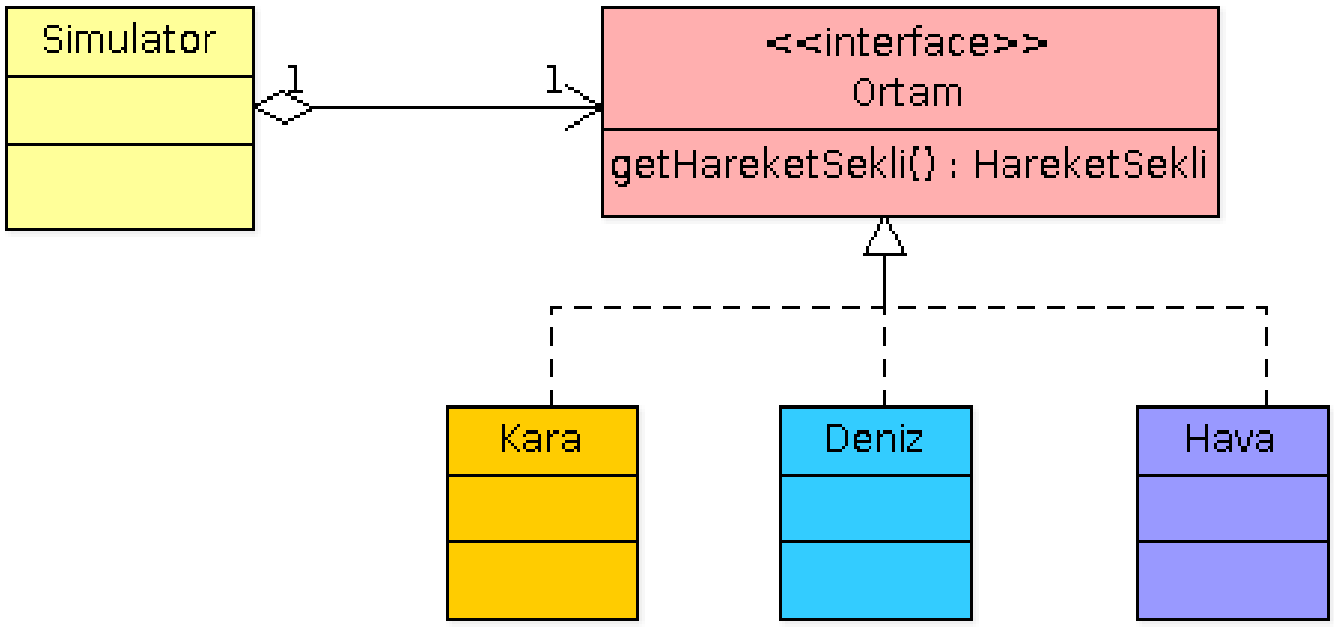
```
public class UcanBalik extends Balik {
...
    @Override
    public void hareketEt() {
        if(uc) {
            new Ucmak().hareketEt();
        } else {
            super.hareketEt();
        }
    }
}
```

OCP ve DIP problemleri Devam ediyor!

Kod 10

Durum bir önceki tasarımımıza göre bir nebze iyileşme göstermiş ise de UcanBalik ve Simulator sınıflarında "hareketin ortam tarafından belirlenmesi" gereksiniminden ötürü OCP ve DIP ihlalleri devam etmektedir. (Kod-10)

"Hareket ortama göre değişiyor" ise Simulator nesnesinin o andaki mevcut Ortam bilgisinden Canli nesneyi haberdar etmesi, Canli nesnenin de hareket şeklini bu Ortam nesnesininden elde etmesi en doğru çözüm olacaktır. (Şekil-5, Kod-11)


Simulator
1
1
<<interface>>
Ortam
getHareketSekli() : HareketSekli
Kara
Deniz
Hava

Şekil 5

```
public interface Ortam {
   public HareketSekli getHareketSekli();
}
```

```
public class Deniz implements Ortam {

   public HareketSekli getHareketSekli() {
      return new Yuzmek();
   }

}
```

Kod 11

Canli sınıfının hareketEt metodu, input argüman olarak Ortam nesnesi alacak biçimde değiştirilebilir. Varsayılan durumda hareketEt metodu, işi Canli nesnenin içindeki HareketSekli nesnesine havale etmektedir. Ancak herhangi bir alt sınıf hareketEt metodunu override ederek Ortam nesnesinden aldığı HareketSekli'ni kullanarak davranışını değiştirebilir. (Kod-12)

```
public abstract class Canli {
...
    private HareketSekli hareketSekli;

    public void hareketEt(Ortam ortam) {
        hareketSekli.hareketEt();
    }
}
```

Hareket şeklini ortama göre değiştirme imkanı sağlanıyor

```
public class UcanBalik extends Balik {

    @Override
    public void hareketEt(Ortam ortam) {
        ortam.getHareketSekli().hareketEt();
    }
}
```

Kod 12

Simulator sınıfı da hareketEttir metodu içerisinde Canli nesnelerin hareketEt metotlarını çağırırken mevcut Ortam nesnesini input argüman olarak vermektedir. Bu sayede Canli nesneye davranışını ortama göre farklılaştırma şansı sunulmaktadır. (Kod-13)

```
public class Simulator {
    private Ortam ortam;

    public Ortam getOrtam() {
        return ortam;
    }

    public void setOrtam(Ortam ortam) {
        this.ortam = ortam;
    }

    public void hareketEttir(Canli...canlilar) {
        for(Canli c:canlilar) {
            c.hareketEt(getOrtam());
        }
    }
}
```

O anki ortam, hareketEt metoduna input argüman olarak veriliyor

Kod 13

Son olarak da test amaçlı olarak yazılmış olan Main sınıfının main metodunda Simulator nesnesinin ortam bilgisi Deniz'den Hava'ya dönüştürüldüğünde UcanBalik nesnesinin davranışının da başarılı biçimde değiştiği gözlenecektir. (Kod-14)

```
public class Main {

    public static void main(String[] args) {
        Simulator simulator = new Simulator();

        simulator.setOrtam(new Deniz());
        simulator.hareketEttir(new UcanBalik());

        simulator.setOrtam(new Hava());
        simulator.hareketEttir(new UcanBalik());
    }
}
```

```
yüzüyor...
uçuyor...
```

Kod 14

Nesne Yönelimli Yazılım Geliştirmede Tasarım Örüntülerinin Rolü

Tasarım örüntülerinin temelindeki fikir de yazılım sistemlerinin kalitesinin nesnel biçimde değerlendirilip değerlendirilemeyeceğidir. Tasarım örüntüleri kaliteli olarak nitelendirilen, zaman içerisinde ortaya çıkan değişiklik taleplerini esnek biçimde karşılayabilen pek çok yazılım sisteminin incelenmesi sonucu ortaya çıkmış, rafine edilmiş, yeniden kullanılabilecek şekilde ifade edilmiş bilgi yumağıdır. Bu tür yazılım sistemlerinde olup da, kötü veya başarısız olarak nitelendirilmiş tasarımlarda olmayan, ya da kötü tasarımlı sistemlerde olup da, bu tür başarılı sistemlerin uzak durduğu noktalara odaklanılarak tespit edilmişlerdir.

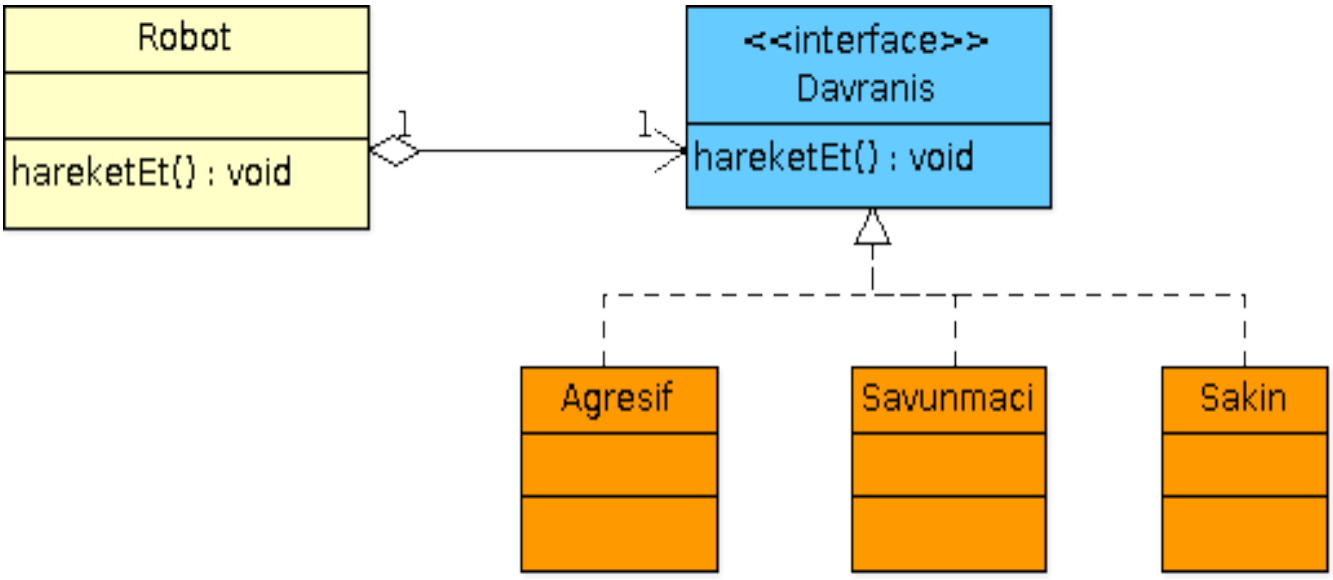
Tasarım Örüntülerinin Faydaları

Tasarım örüntüleri daha önceki çözümlerin, tasarımların ve hali hazırda mevcut bilgi birikiminin yeniden kullanılmasını sağlar. Bu örüntüler zaman içerisinde evrilmiş ve olgunlaşmış

çözümlerdir. Probleme sıfırdan başlamayı ve muhtemelen daha önceki ekiplerin karşılaştığı problemlerle tekrardan uğraşmamayı sağlar. Diğer ekiplerin deneyimlerinden faydalanmak mümkün hale gelir. Ekip içinde ortak bir terminolojinin oluşmasını sağlar, ortak bir bakış açısı getirir. Tasarım ve nesne modelleme sürecine bir üst perspektiften bakmayı sağlar. Bu sayede daha ilk dakikadan itibaren gereksiz detaylar ve ayrıntılar içerisinde boğulmanın önüne geçilebilir. Belirli bir süre ve farklı yerlerde kullanıldıklarından olası yeni gereksinimleri ele alırken üzerlerinde değişiklik yapmak daha kolay ve hızlıdır.

Örnek Bir Örüntü: Strategy

Öncelikle strategy örüntüsünün ihtiyaç duyulduğu örnek problem tanımına bakalım; "Robot davranışları ile ilgili bir simülasyon programı geliştirilecektir. Robotların davranışları agresif, savunmacı ve sakin olarak değişmektedir. Her bir davranış tipine göre robot farklı farklı haraket etmektedir. Robotların davranışları dinamik olarak değişebilmektedir."



Şekil 6

Tasarım örüntülerinin dokümantasyonunda örüntünün hangi probleme çözüm getirdiği, çözümü oluşturan bileşenler ve bunlar arasındaki ilişkiler, çözümün detayları, örüntünün uygulanması ile ortaya

çıkan sonuçlar ve kısıtlardan bahsedilir.

Burada örnek olarak üzerinde durulan strategy örüntüsü ile sistemin ortama göre dinamik olarak farklı davranış sergilemesi veya farklı bir algoritma çalıştırabilmesi mümkün kılınmaktadır. Kullanılacak algoritma istemciye veya eldeki veriye göre değişiklik gösterebilir. İstemcinin algoritmanın detayını bilmesine gerek yoktur. Bu sayede zaman içerisinde ortaya çıkabilecek yeni gereksinimler ve değişiklikler mevcut kod üzerinde değişikliğe gitmeden, composition yöntemi ile sistem genişletilerek sağlanabilir. Algoritmanın seçimi ile implemantasyonu birbirinden ayrı tutulur. Algoritma seçimi bağlama göre dinamik yapılabilir.

Switch ve şartlı ifadeler ortadan kaldırılır. Algoritma değişiklikleri için inheritance ile alt sınıf oluşturmaya iyi bir alternatiftir. Bütün algoritmalar aynı şekilde çağrılmalıdır. Zaman zaman strategy ile bağlam arasında etkileşim gerekebilir.

Sonuç

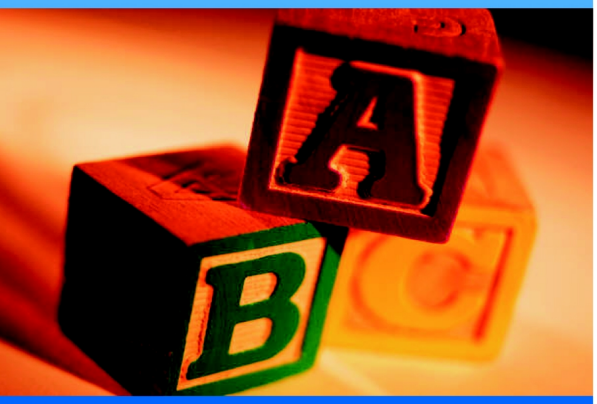
Kaliteli bir tasarımın önemi çoğunlukla yazılım sistemleri işletime alınıp bir süre kullanıldıktan sonra ortaya çıkmaktadır. Zaman içerisinde sistemin bakım ve idame süresince karşılaşılan hataların düzeltilmesi, yeni gereksinimlerin sisteme eklenmesi veya mevcutlarda yapılacak değişikliklerin ele alınması eğer sistemimiz esnek bir tasarıma ve mimariye sahip ise oldukça kolay olacaktır. Yeniden kullanılabilir sistemlerin ve modüllerin ortaya çıkartılması uzun vadede yeni sistemlerin geliştirme maliyetlerine de olumlu etkilerde bulunacaktır.

Kaliteli bir tasarıma götüren yol ve yordamları yazılım geliştiricilerin çok iyi etüd ederek günlük geliştirme pratikleri arasına katmaları ve tasarım ve mimari çalışmalarını bahsettiğimiz kural ve prensipler etrafında yürütmeleri uzun ömürlü yazılım sistemleri ortaya koymalarını sağlayacaktır.

Encapsulation, inheritance, abstraction ve composition gibi nesne yönelimli kavramların atomik birimler olduğu, bu atomik parçaların belirli kural ve prensipler ışığında bir araya getirilerek daha büyük sistem parçaları ortaya çıkatmak gerektiği unutulmamalıdır. Bu sistem parçaları da bir nevi moleküllere benzetilebilir. Bu moleküllerin bir araya gelmesi ile tasarım örüntüleri ortaya çıkmakta ve birbirleri ile ilişkili tasarım örüntülerinin oluşturduğu dil ile de bir sistem bütün olarak ifade edilebilmektedir.

Ürün ve Hizmetlerimiz

Speedy Framework (Model Güdümlü Çevik Yazılım Geliştirme Platformu)
Kurumsal Uygulama Geliştirme
Teknoloji Danışmanlığı ve Koçluk
Kurumsal Java Eğitimleri



Speedy Framework

(Model Güdümlü Çevik Yazılım Geliştirme Platformu)

Tamamen açık kaynak kodlu sistemler üzerine bina ettiğimiz Speedy Framework ile kurumsal yazılım geliştirme sürecini daha sistematik, otomatize, hızlı ve verimli bir hale getiriyor, yazılım sistemlerinin geliştirme, bakım ve idame maliyetlerini önemli ölçüde azaltıyoruz.

Speedy Framework'ün Faydaları

- Orta katman hizmetlerinin hazır olarak sunulması, uygulama geliştirmede ilk andan itibaren mimari yapının oturmuş olması ile yazılım geliştirme sürecinde önemli ölçüde bir hızlanma olur.
- Geliştirilen uygulamaların kalite düzeyinin uygulama genelinde aynı olması sağlanır.
- Uygulamalardaki kalitenin düzeyi uygulama geliştiricilerden bağımsız hale gelir.
- Model güdümlü yazılım geliştirme yaklaşımı, mimarisel yapının hazır olması, tekrar kullanılabilir servisler sayesinde uygulamada ortaya çıkabilecek hata sayısında da hissedilir bir düşüş söz konusu olur.
- Kullanıcı ara yüzleri bütün ekranlarda ve senaryolarda bir standarda sahip olduğu için müşterinin ve son kullanıcıların sisteme adaptasyonu kolaylaşıyor, sistemi öğrenme süreleri oldukça kısalır.
- Projelerin geliştirme sürecinden, bakım ve idame dönemlerine kadar bütün evrelerinde maliyetler azalır.

Daha fazla bilgi için: **http://www.speedyframework.com**

Kurumsal Uygulama Geliştirme

Kurumsal uygulama geliştirme faaliyetlerimizle kurumunuzun ihtiyaç duyduğu her türlü yazılım ihtiyacını karşılayacak, süreçlerinizi daha verimli hale getirecek çözümler üretiyoruz. Uzun yıllar boyunca pekçok kurumsal yazılım projesinde elde ettiğimiz bilgi ve tecrübemizle ihtiyacınız olan çözümleri en uygun şekilde hizmetinize sunuyoruz.

Teknoloji Danışmanlığı ve Koçluk

Java teknolojileri, kurumsal yazılım geliştirme, nesne yönelimli analiz, tasarım ve modelleme, yazılım mimarileri konularında bire bir koçluk ve proje danışmanlığı hizmetlerimizle yazılım geliştirme faaliyetlerinizin her adımında size destek oluyoruz.

Kurumsal Java Eğitimleri

JAVA Eğitimleri

Java Programlama Dili, Spring Application ve Security Framework, Hibernate, Object Oriented Analiz Tasarım, Tasarım Örüntüleri (Design Patterns), Aspect Oriented Programlama konularında verdiğimiz eğitimlerin kurumunuza ve çalışanlarınıza kesinlikle artı değer katacağına eminiz. Kurumlara özel ve genel katılıma açık olarak düzenlediğimiz eğitimlerle bilişim sektörümüze rafine bilgi ve tecrübeyi aktarıyoruz.

Daha fazla bilgi için: **http://www.java-egitimleri.com**