KIRKLARELİ ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
YAZILIM MÜHENDİSLİĞİ BÖLÜMÜ

YAZ20411
YAZILIM MİMARİSİ VE TASARIMI

Hazırlayan: Öğr. Gör. Dr. Fatih BAL

Kasım 2023

İÇİNDEKİLER

BÖLÜM - I

1. NESNEYE YÖNELİK PROGRAMLAMA

"*Nesneye Yönelik Programlama*" (*Object-Oriented Programming* veya *kısaca OOP*), yazılım geliştirmeye ve tasarımına bir yaklaşımı ifade eder. Bu yaklaşım, programlamada kullanılan nesneleri merkeze alarak, bunların birbirleriyle etkileşimini ve ilişkilerini modellemeyi amaçlar. Temel olarak, OOP, gerçek dünyadaki nesnelerin ve ilişkilerin bir bilgisayar programında nasıl temsil edilebileceği üzerine odaklanır.

OOP'nin temel prensipleri şunlardır:

1. 1. SINIFLAR VE NESNELER

1. 1. 1. SINIF (CLASS)

Bir nesnenin tasarımını sağlayan şablondur. Bir sınıf, nesnenin özelliklerini (veri alanları) ve davranışlarını (metotlar) tanımlar. Bir sınıf, nesnelerin (objects) tasarımını ve yapısını tanımlayan bir şablondur. Sınıflar, bir nesnenin sahip olabileceği veri alanlarını (attributes veya properties) ve bu veri üzerinde çalışacak metodları (methods) içerir. Sınıf, bir nesnenin temel özelliklerini ve davranışlarını tanımlayan bir tasarım planı gibidir. Örneğin, bir "Araba" sınıfı aşağıdaki **Şekil-1**'deki gibi görünebilir:

```
public class Araba {
    // Veri alanları (properties)
    3 usages
    private String marka;
    3 usages
    private String model;
    1 usage
    private String renk;

    // Kurucu metod (constructor)
    no usages
    public Araba(String marka, String model, String renk) {
        this.marka = marka;
        this.model = model;
        this.renk = renk;
    }

    // Metodlar
    no usages
    public void calistir() {
        System.out.println(marka + " " + model + " çalıştırılıyor.");
    }

    no usages
    public void durdur() {
        System.out.println(marka + " " + model + " durduruldu.");
    }
}
```

Şekil 1. Araba Sınıfı

Şekil-1'deki Java sınıfı, "Araba" adında bir sınıfı temsil eder. Sınıf, üç özel veri alanı (marka, model, renk) ve bir kurucu metot ile iki davranışı (calistir, durdur) içerir.

1. 1. 2. NESNE (OBJECT)

Bir sınıftan türetilen örnekleri oluşturur. Nesneler, sınıf tarafından tanımlanan özelliklere ve davranışlara sahiptir. **Şekil-2**'de yer alan Java programı, "Araba" sınıfından iki nesne oluşturur (araba1 ve araba2) ve bu nesnelerin metotlarını çağırarak çıktıları ekrana yazdırır. Bu, Java'da sınıf ve nesne kavramlarının nasıl kullanıldığını gösteren basit bir örnektir.

```
public class ArabaTest {
    public static void main(String[] args) {
        // "Araba" sınıfından nesneler oluşturuyoruz
        Araba araba1 = new Araba( marka: "Toyota",  model: "Corolla",  renk: "Mavi");
        Araba araba2 = new Araba( marka: "Honda",  model: "Civic",  renk: "Kırmızı");

        // Nesnelerin metodlarını çağırma
        araba1.calistir();  // Çıktı: Toyota Corolla çalıştırılıyor.
        araba2.durdur();    // Çıktı: Honda Civic durduruldu.
    }
}
```

Şekil 2. Araba Test Sınıfı

2. ENCAPSULATION (KAPSÜLLEME):

Veri ve işlevselliği bir araya getirme ilkesidir. "*Encapsulation*" (Kapsülleme) kavramı, Nesneye Yönelik Programlama (OOP) paradigması içinde önemli bir ilkedir. Bu ilke, veriyi ve onunla ilgili işlevselliği bir kapsül içinde birleştirerek, bu kapsülü dış dünyadan gelen erişimlere karşı koruma amacını taşır. Yani, sınıf içindeki veri ve metotların dışarıdan gelen müdahalelere karşı gizlenmesi ve kontrol altında tutulmasıdır. Verinin dış dünyadan gizlenmesine ve sadece belirli metotlar aracılığıyla erişilebilir olmasına olanak tanır.

Encapsulation'ın temel avantajları şunlardır:

Gizlilik (Privacy): Sınıf içindeki veri alanları genellikle özel (private) olarak işaretlenir, bu da demektir ki bu verilere doğrudan erişim dış sınıflardan engellenir. Bu, sınıfın içyapısının dışarıdan gizli kalmasını sağlar ve sadece belirli metotlar aracılığıyla bu verilere erişilebilir.

Güvenlik (Security): Veri güvenliği ve bütünlüğü, doğrudan erişime karşı korunduğu için artar. Sınıfın içindeki veri, sınıfın sunduğu metotlar üzerinden kontrol edilebilir ve manipülasyonlara karşı korunabilir.

Esneklik (Flexibility): Sınıfın içyapısını değiştirmek, güncellemek veya iyileştirmek, sınıfın dışındaki kodları etkilemeden gerçekleştirilebilir. Bu, sınıflar arasındaki bağımlılığı azaltır ve kodun daha esnek ve bakımı kolay hale gelmesini sağlar.

Kod Organizasyonu (Code Organization): Kapsülleme, büyük projelerde kodun daha düzenli ve anlaşılır olmasına katkı sağlar. Sınıfların içyapısı gizlenirken, sadece dışarıya sunulan arayüzler önemlidir.

Şekil-3'te yer alan örnekte, marka, model ve renk veri alanları private olarak işaretlenmiş ve bu verilere erişim için get ve set metodları tanımlanmıştır. Bu şekilde, Araba sınıfının iç yapısı gizlenirken, dışarıdan kontrol edilebilmesi sağlanmış olur.

```
public class Araba {
    // Veri alanları private olarak işaretlenmiştir
    5 usages
    private String marka;
    5 usages
    private String model;
    3 usages
    private String renk;
    // Kurucu metod (constructor)
    2 usages
    public Araba(String marka, String model, String renk) {
        this.marka = marka;
        this.model = model;
        this.renk = renk;
    }
    // Veri alanlarına erişim sağlayan get ve set metodları
    no usages
    public String getMarka() {
        return marka;
    }
    no usages
    public void setMarka(String marka) {
        this.marka = marka;
    }
    no usages
    public String getModel() {
        return model;
    }
    no usages
    public void setModel(String model) {
        this.model = model;
    }
    no usages
    public String getRenk() {
        return renk;
    }
    no usages
    public void setRenk(String renk) {
        this.renk = renk;
    }
```

Şekil 3. Araba Sınıfına Ait Encaspulation Örneği

Java programlama dilinde kullanılan erişim kontrol belirleyicileri (Access Modifiers), sınıfların ve sınıf üyelerinin diğer sınıflar tarafından nasıl erişilebileceğini düzenler. Bu belirleyiciler, sınıflar arasındaki ilişkileri ve kodun güvenliğini yönetmeye yardımcı olur. Java'da üç temel erişim kontrol belirleyicisi bulunmaktadır: ***public***, ***private***, ve ***protected***.

2. 1. PUBLIC

Public belirleyici, sınıf veya sınıf üyelerinin herkes tarafından erişilebilir olduğunu gösterir. Bir sınıf veya sınıf üyesi public olarak işaretlenirse, bu sınıfa başka paketlerden ve sınıflardan erişim sağlanabilir.

2. 2. PRIVATE

Private belirleyici, sınıf veya sınıf üyelerinin sadece tanımlandıkları sınıf içinde erişilebilir olduğunu gösterir. Bu, sınıfın içyapısının dışarıya gizlenmesini ve sınıfın kontrolü altında tutulmasını sağlar.

2. 3. PROTECTED

Protected belirleyici, sınıf içinde ve aynı paket içinde erişilebilir. Ayrıca, alt sınıflardan erişim sağlanabilir. Bu belirleyici, sınıf kalıtımında kullanılırken önemli bir rol oynar.

2. 4. GET VE SET METOTLARI

GET ve SET metotları, genellikle "Accessor" ve "Mutator" olarak da adlandırılır ve Nesneye Yönelik Programlama (OOP) paradigmalarında Encapsulation (Kapsülleme) prensibini desteklemek amacıyla kullanılır. Bu metotlar, sınıf içindeki private veri alanlarına erişim ve bu veri alanlarını güncelleme işlemlerini düzenler.

2. 4. 1. GET METODU (ACCESSOR)

GET metodu, bir sınıftaki private bir veri alanının değerini döndüren bir metottur. Bu metotlar genellikle isminin önüne "get" eklenerek adlandırılır (örneğin, *getMarka()*). GET metotları, sınıfın dışındaki diğer kodların private veriye okuma erişimi sağlar. **Şekil-3**'te yer alan örnekte, *getMarka()* metodu, marka adlı private bir veri alanının değerini döndürür.

2. 4. 2. SET METODU (MUTATOR)

SET metodu, bir sınıftaki private bir veri alanının değerini güncelleyen bir metottur. Bu metotlar genellikle isminin önüne "set" eklenerek adlandırılır (örneğin, *setMarka(String yeniMarka)*). SET metotları, sınıfın dışındaki diğer kodların private veriye yazma erişimini düzenler. **Şekil-3**'te yer alan örnekte, *setMarka(String yeniMarka)* metodu, marka adlı private bir veri alanının değerini günceller.

3. INHERITANCE (MİRAS ALMA VEYA KALITIM)

"Inheritance" (Miras Alma), Nesneye Yönelik Programlama (OOP) paradigmalarından biridir ve sınıflar arasındaki bir ilişkiyi ifade eder. Bu kavram, bir sınıfın başka bir sınıftan özellikleri ve davranışları miras almasını sağlar. Bu, kodun yeniden kullanılabilirliğini artırır ve sınıflar arasında hiyerarşik ilişkiler kurulmasına olanak tanır.

Bir sınıf, başka bir sınıftan özellikler ve davranışlar miras alabilir. Miras veren sınıfa "üst sınıf" veya "ana sınıf" denir. Terminolojide, Base Class, Super Class veya Parent Class olarak adlandırılır. Miras alan sınıfa ise "alt sınıf" denir. Terminolojide, Sub-Class veya Child Class olarak adlandırılır. Miras alma "*extends*" sözcüğü ile ifade edilir. **Extends** sözcüğü, bir sınıfın başka bir sınıftan miras alacağını belirtir. Bu, üst sınıfın özelliklerinin ve davranışlarının alt sınıfa geçmesini sağlar. **Şekil-5**'te yer alan **Araba** sınıfı, **Şekil-4**'te yer alan **Arac** sınıfından kalıtım yoluyla türetilmiştir.

```
public class Arac {
    2 usages
    private String marka;

    no usages
    public void setMarka(String marka) {
        this.marka = marka;
    }

    no usages
    public String getMarka() {
        return marka;
    }
}
```

Şekil 4. Arac Sınıfı

```
public class Araba extends Arac {
    2 usages
    private int kapiSayisi;

    no usages
    public void setKapiSayisi(int kapiSayisi) {
        this.kapiSayisi = kapiSayisi;
    }

    no usages
    public int getKapiSayisi() {
        return kapiSayisi;
    }
}
```

Şekil 5. Arac Sınıfından Türetilmiş Araba Sınıfı

Kalıtımın bazı avantajları bulunmaktadır. Bu avantajlar şunlardır:

- Kodun Yeniden Kullanılabilirliği: Ortak özellikler ve davranışlar, bir üst sınıfta tanımlanabilir ve alt sınıflar bu özellikleri doğrudan miras alabilir. Bu, kodun tekrar kullanılabilirliğini artırır.
- Kodun Daha Organize Olması: İlgili sınıflar arasında hiyerarşik bir düzen kurmak, kodun daha düzenli ve anlaşılır olmasını sağlar.
- Alt Sınıflarda Özellik Ekleme ve Değiştirme: Alt sınıflar, miras aldıkları özellikleri kullanırken, ihtiyaçlarına göre bu özellikleri genişletebilir veya değiştirebilirler. Bu, esneklik sağlar.

NOT

Miras alınan sınıf, miras alan sınıfın bir türü olmalıdır. Bu nedenle, "**IS-A**" ilişkisi kurulmalıdır. Örneğin, "*Araba bir araçtır*" ifadesi geçerli olmalıdır.

Kapsülleme ve erişim kontrolü, Java'da sınıflar arasındaki ilişkileri düzenleyen önemli bir konsepttir. Base class (üst sınıf veya ana sınıf) içinde tanımlanan public, private ve protected değişkenler ve metotlar, alt sınıflar (sub-classes veya alt sınıflar) tarafından nasıl kullanılabilir, aşağıda detaylı bir şekilde açıklanmıştır:

- "*public*" olarak tanımlanan değişkenler ve metotlar, alt sınıflar tarafından doğrudan erişilebilir ve kullanılabilir. Bu, alt sınıfın bu özelliklere ve metotlara sahipmiş gibi davranabileceği anlamına gelir.

- "*private*" olarak tanımlanan değişkenler ve metotlar, sadece tanımlandıkları sınıf içinde erişilebilir ve kullanılabilir. Alt sınıflar, bu özelliklere ve metotlara doğrudan erişemezler.
- "*protected*" olarak tanımlanan değişkenler ve metotlar, tanımlandıkları sınıf içinde, aynı paket içinde ve alt sınıflar tarafından erişilebilir ve kullanılabilir.

4. POLYMORPHISM (ÇOK BİÇİMLİLİK)

"Polymorphism" (Çok Biçimlilik), Nesneye Yönelik Programlama (OOP) paradigmalarından biridir ve aynı isme sahip ancak farklı davranışlara sahip nesnelerin aynı şekilde kullanılabilmesini ifade eder. Polimorfizm, kodun daha esnek ve yeniden kullanılabilir olmasını sağlar. Polimorfizmin iki temel türü vardır: *Compile-Time Polymorphism* (Derleme Zamanı Çok Biçimlilik) ve *Run-Time Polymorphism* (Çalışma Zamanı Çok Biçimlilik):

4. 1. COMPILE-TIME POLYMORPHISM

Compile-time polymorphism, aynı isimdeki bir metodun farklı parametre listeleriyle birden fazla kez tanımlanmasıdır. Bu duruma aynı zamanda "*method overloading*" denir. Hangi metodu çağıracağınız derleme zamanında belirlenir. **Şekil-6**, Matematik sınıfında yer alan aynı isimde fakat farklı türde iki metodu içermekte ve **Şekil 7**'de kullanım örneği yer almaktadır.

```
public class Matematik {
    1 usage
    public int topla(int sayi1, int sayi2) {
        return sayi1 + sayi2;
    }

    1 usage
    public double topla(double sayi1, double sayi2) {
        return sayi1 + sayi2;
    }
}
```

Şekil 6. Matematik Sınıfı

```
public class Main {
    public static void main(String[] args) {
        Matematik matematik = new Matematik();
        int sonuc1 = matematik.topla(3, 4);          // topla(int, int) metodu çağrılır
        double sonuc2 = matematik.topla(2.5, 3.5);   // topla(double, double) metodu çağrılır
    }
}
```

Şekil 7. Matematik Sınıfı Kullanım Örneği

4. 2. RUN-TIME POLYMORHISM

Run-time polymorphism, bir alt sınıfın üst sınıf referansı ile kullanılabilmesini ifade eder. Bu duruma aynı zamanda "*method overriding*" denir. Hangi metodu çağıracağınız çalışma zamanında belirlenir. **Şekil-8**, **Şekil-9** ve **Şekil-10** Run-time polymorphism örneğini içermektedir.

```
class Arac {
    2 usages  1 override
    void hareketEt() {
        System.out.println("Araç hareket ediyor.");
    }
}
```

Şekil 8. Matematik Sınıfı

```
class Araba extends Arac {
    2 usages
    void hareketEt() {
        System.out.println("Araba hareket ediyor.");
    }

    1 usage
    void kornaCal() {
        System.out.println("Araba korna çalıyor.");
    }
}
```

Şekil 9. Matematik Sınıfı

```
public class Main {
    public static void main(String[] args) {
        Arac arac1 = new Arac(); // Üst sınıf referansı ile üst sınıf nesnesi
        Arac arac2 = new Araba(); // Üst sınıf referansı ile alt sınıf nesnesi

        arac1.hareketEt(); // "Araç hareket ediyor."
        arac2.hareketEt(); // "Araba hareket ediyor."

        // arac2.kornaCal(); // Bu satır hata verir, çünkü Arac referansı ile Araba nesnesi kullanılıyor

        // Ancak, Araba tipinde bir referans ile Araba nesnesini kullanabiliriz
        Araba araba = new Araba();
        araba.kornaCal(); // "Araba korna çalıyor."
    }
}
```

Şekil 10. Matematik Sınıfı

5. ABSTRACTION (SOYUTLAMA)

Abstraction (Soyutlama), Nesneye Yönelik Programlama (OOP) paradigmalarından biridir ve karmaşık sistemleri basitleştirmek, temel özellikleri vurgulamak ve gereksiz ayrıntıları gizlemek amacıyla kullanılır. Bu, programlamada bir tasarım ilkesidir ve karmaşıklığı yönetilebilir hale getirir. Soyutlama kavramının temel ilkeleri şunlardır:

- Gizleme (Encapsulation): Soyutlama, veri ve işlevselliği bir kapsül içinde gizleme ilkesine dayanır. Sınıf içindeki detaylar sınıfın dışındaki kodlardan gizlenir ve sadece belirlenen arayüz üzerinden erişilebilir.
- Soyut Sınıflar ve Arabirimler: Soyutlama genellikle soyut sınıflar ve arayüzler (interfaces) aracılığıyla gerçekleştirilir. Soyut sınıflar, ortak özelliklere ve davranışlara sahip alt sınıfların ortak bir tabanını tanımlar. Arabirimler ise bir grup metodu tanımlar ve bu metotların alt sınıflarca implemente edilmesini bekler.

```
abstract class Arac {
    2 usages
    private String marka;

    1 usage
    public Arac(String marka) {
        this.marka = marka;
    }

    // Soyut bir metod
    1 usage  1 implementation
    abstract void hareketEt();

    2 usages
    public String getMarka() {
        return marka;
    }
}
```

Şekil 11. Abstract Arac Sınıfı

```java
class Araba extends Arac {
    2 usages
    private int hiz;

    1 usage
    public Araba(String marka, int hiz) {
        super(marka);
        this.hiz = hiz;
    }

    // Soyut metodu implemente et
    1 usage
    void hareketEt() {
        System.out.println(getMarka() + " arabası hareket ediyor. Hız: " + hiz + " km/saat");
    }

    // Araba sınıfına özgü bir metot
    1 usage
    void kornaCal() {
        System.out.println(getMarka() + " arabası korna çalıyor.");
    }
}
```

Şekil 12. Arac Sınıfından Türetilen Araba Sınıfı

```java
public class Main {
    public static void main(String[] args) {
        Arac arac = new Araba( marka: "Toyota",  hiz: 80);

        arac.hareketEt();  // "Toyota arabası hareket ediyor. Hız: 80 km/saat"

        // arac.kornaCal(); // Bu satır hata verir, çünkü Arac referansı ile Araba sınıfına özgü bir metot kullanılamaz

        // Ancak, Araba tipinde bir referans ile Araba sınıfının özel metotlarına erişebiliriz
        Araba araba = (Araba) arac;
        araba.kornaCal();  // "Toyota arabası korna çalıyor."
    }
}
```

Şekil 13. Arac Sınıfının Kullanım Örneği

Yukarıdaki örnekte, **Şekil-11**'de yer alan **Arac** soyut sınıfı, **Şekil-12**'de yer alan **Araba** sınıfı tarafından genişletiliyor. Arac soyut sınıfı, soyut bir *hareketEt* metodu ve somut bir *getMarka* metodu içeriyor. Araba sınıfı, Arac sınıfının soyut metodunu implemente ediyor ve kendi özel metotlarından biri olan *kornaCal* metodu bulunuyor.

Soyutlama, Arac sınıfı üzerinden Araba sınıfının kullanımını sağlar ve Arac sınıfı, Araba sınıfının içyapısını gizler.

6. INTERFACE

Interface (Arayüz), Java'da bir soyutlama mekanizmasıdır ve bir sınıfın hangi metotları implement etmesi gerektiğini belirler. Bir sınıf, birden fazla arayüzü implement edebilir, bu da çoklu kalıtımın bir türünü sağlar (Java'da sınıflar sadece bir tane super classı extend edebilir, ancak birden fazla arayüzü implement edebilir).

```
public interface Arac {
    1 usage
    void hareketEt();
    1 usage
    void dur();
}
```

Şekil 14. Abstract Arac Sınıfı

```
// Alt Sınıf
3 usages
class Araba implements Arac {
    4 usages
    private String marka;

    1 usage
    public Araba(String marka) {
        this.marka = marka;
    }

    // Arayüzdeki soyut metodları implemente et
    1 usage
    public void hareketEt() {
        System.out.println(marka + " arabası hareket ediyor.");
    }

    1 usage
    public void dur() {
        System.out.println(marka + " arabası durdu.");
    }

    // Araba sınıfına özgü bir metot
    1 usage
    public void kornaCal() {
        System.out.println(marka + " arabası korna çalıyor.");
    }
}
```

Şekil 15. Abstract Arac Sınıfı

```java
public class Main {
    public static void main(String[] args) {
        Arac arac = new Araba( marka: "Toyota");

        arac.hareketEt();  // "Toyota arabası hareket ediyor."
        arac.dur();        // "Toyota arabası durdu."

        // arac.kornaCal(); // Bu satır hata verir, çünkü Arac referansı ile Araba sınıfına özgü bir metot kullanılamaz

        // Ancak, Araba tipinde bir referans ile Araba sınıfının özel metotlarına erişebiliriz
        Araba araba = (Araba) arac;
        araba.kornaCal();  // "Toyota arabası korna çalıyor."
    }

}
```

Şekil 16. Abstract Arac Sınıfı

Yukarıdaki örnekte, **Şekil 14**'te **Arac** adında bir arayüz tanımlanmış ve bu arayüz, *hareketEt* ve *dur* adında iki soyut metot içeriyor. Daha sonra, **Şekil-15**'te **Araba** sınıfı, Arac arayüzünü implement ediyor. Araba sınıfı, Arac arayüzündeki metodları implement etmek zorunda ve aynı zamanda kendi özel metotlarına sahip olabilir. Arac arayüzü, Araba sınıfını ve muhtemelen başka sınıfları, aynı arayüz üzerinden kullanmamıza olanak tanımaktadır.

7. OOP İLE UYGULAMALAR

Bu bölümde anlatılan konular ile ilgili uygulamalara https://github.com/balfatih/YAZ16303_Yazilim_Mimarisi_ve_Tasarimi GITHUB reposundan erişebilirsiniz.

BÖLÜM - II

1. SOFTWARE DESIGN PATTERNS

Software Design Patterns (Yazılım Tasarım Desenleri), tekrar eden sorunlara karşı genel ve tekrar kullanılabilir çözümler sağlamak amacıyla geliştirilmiş genel tasarım yönergeleridir. Bu desenler, yazılım geliştirme sürecinde karşılaşılan yaygın sorunlara karşı test edilmiş ve etkili çözümler sunar. Desenler, yazılım geliştiricilere belirli bir sorunla başa çıkmak için kullanılabilecek bir çerçeve veya yaklaşım sağlar.

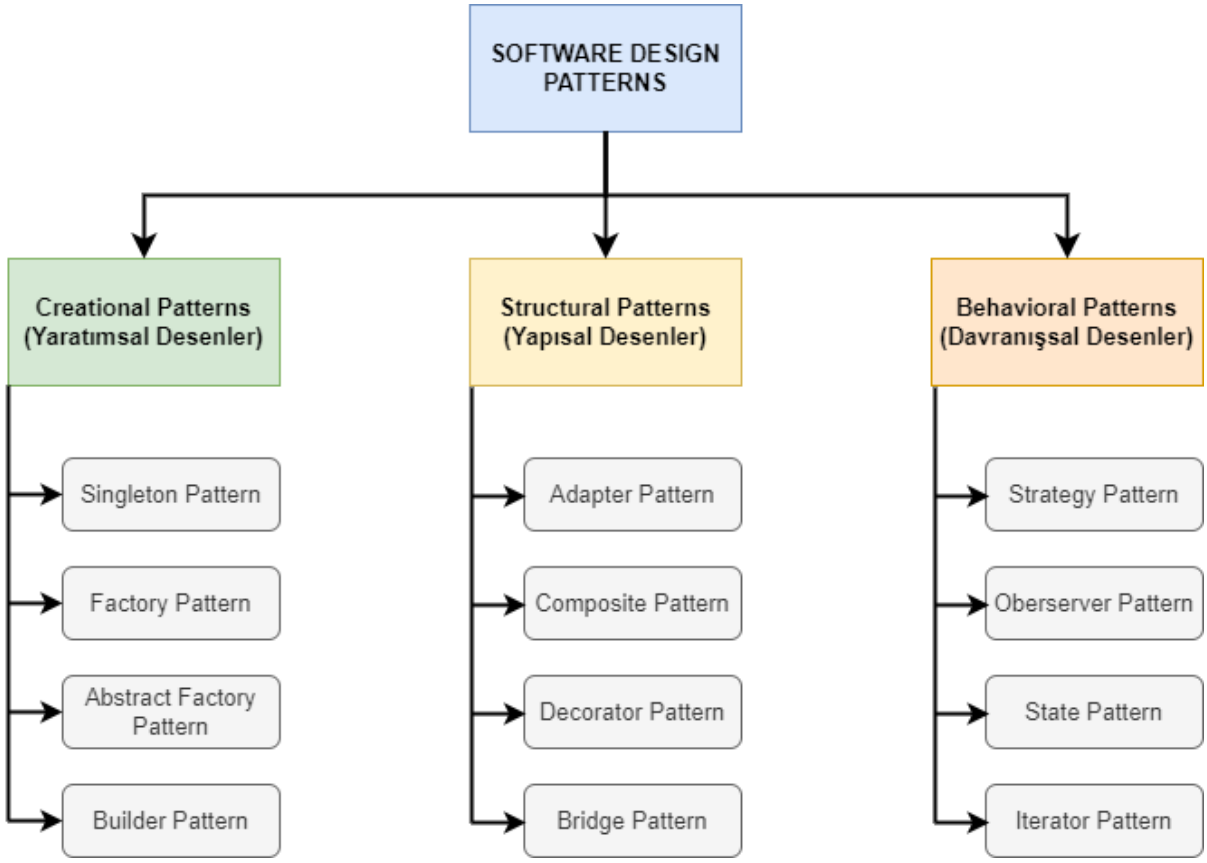
Bir yazılım tasarım deseni genellikle bir isme, bir sorun tanımına, bir niyet (amaca) ve bir çözüm tarifine sahiptir. Bu desenler, yazılım tasarımını daha anlaşılır, esnek ve sürdürülebilir hale getirerek, geliştirme sürecini iyileştirmeyi amaçlar.

Yazılım tasarım desenleri, Gang of Four (GoF) olarak bilinen dört kişilik bir grup tarafından popüler hale getirilmiştir. *"Design Patterns: Elements of Reusable Object-Oriented Software*" adlı kitaplarında, Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides, bir dizi temel tasarım deseni tanımlamış ve bu desenlerin kullanımını açıklamışlardır.

Design patterns yani tasarım kalıpları, yazılım geliştirmede yaygın olarak karşılaşılan zorluklar için geliştirilen, test edilmiş, kendini kanıtlamış kod tasarımları, programlama şablonlarıdır. Tasarım kalıpları bir algoritma, framework veya kod değildir. Ayrıca belli dile özgü değildir, dilden bağımsızdır. Genellikle nesneler arası ilişkiler UML diyagramları ile gösterilir. Design Pattern yani tasarım kalıpları genellikle OOP yani Nesne Tabanlı Programlama'da kullanılır.

Tasarım desenleri 3 ana başlık altında incelenir.

1. Creational Patterns (Yaratımsal Kalıplar)
2. Structural Patterns (Yapısal Kalıplar)
3. Behavioral Patterns (Davranışsal Kalıplar)



Şekil 1. Software Design Patterns

Şekil-1'de yer alan grafik gösteriminde Yaratımsal, Yapısal ve Davranışsal Tasarım Desenlerine ait dağılım görülmektedir. Ders dönemi boyunca tüm bu tasarım desenleri Java dilinde uygulamalı olarak gösterilecektir.

2. CREATIONAL DESIGN PATTERNS

Creational Design Patterns (Yaratımsal/Oluşturucu Tasarım Desenleri) adını verdikleri bir kategori içerir. Bu desenler, nesnelerin oluşturulmasıyla ilgili sorunları ele alır ve nesne oluşturma sürecini daha esnek, ölçeklenebilir ve bağımsız hale getirmeyi amaçlar. Önemli Yaratımsal Tasarım Desenlerinden Singleton, Factory, Abstract Factory ve Builder tasarım desenlerini inceleyeceğiz.

2. 1. SINGLETON PATTERN

Singleton tasarım deseni, bir sınıfın yalnızca bir örneğine sahip olmasını ve bu örneğe genel bir erişim noktası sağlamayı amaçlayan bir creational tasarım desenidir. Bu desen, bir uygulama içinde belirli bir sınıfın tek bir örneğini paylaşarak, bu örneğe global erişim sağlamak veya kaynak kullanımını optimize etmek için kullanılır.

Singleton tasarım deseninin temel unsurları şunlardır:

Özel Constructor (Yapıcı Metod): Singleton sınıfının bir nesnesinin oluşturulmasını kontrol etmek için genellikle özel bir constructor (yapıcı metot) kullanılır. Bu, sınıfın dışından doğrudan nesne oluşturulmasını engeller.

Private Instance (Özel Örnek): Singleton sınıfının yalnızca bir örneğine sahip olması için genellikle bir özel (private) sınıf örneği kullanılır.

Static Method (Statik Metod): Singleton sınıfının tek bir örneğine erişmek için genellikle bir statik metot kullanılır. Bu metot, sınıfın içindeki örneğe erişim sağlar ve eğer örnek henüz oluşturulmamışsa onu oluşturur.

2. 1. 1. SINGLETON PATTERN UYGULAMASI

Singleton Tasarım Deseni ile ilgili uygulamaya https://github.com/balfatih/YAZ16303_Yazilim_Mimarisi_ve_Tasarimi GITHUB reposundan erişebilirsiniz.

2. 2. FACTORY PATTERN

Factory Design Pattern (Fabrika Tasarım Deseni), bir creational tasarım desenidir. Bir nesne oluşturma sürecini gizlemek ve alt sınıfların nesne oluşturma sürecini belirlemesine olanak tanımak amacıyla kullanılan bir tasarım desenidir. Bu desen, bir arayüz üzerinden bir nesne oluşturmayı ve alt sınıfların bu arayüzü uygulayarak nesne oluşturma sorumluluğunu üstlenmelerini sağlar.

Bu desenin temel avantajlarından biri, istemcilerin nesnelerin nasıl oluşturulduğu konusunda bilgi sahibi olmalarına gerek olmamasıdır. İstemci sadece belirli bir türdeki nesneyi yaratmak için fabrikaya ihtiyaç duyar ve bu fabrika, uygun nesneyi oluşturma sorumluluğunu üstlenir.

Örnek bir Factory Design Pattern uygulamasını düşünelim. Örneğin, bir şekil (Shape) arayüzümüz varsa ve bu arayüzü uygulayan farklı alt sınıflarımız varsa (örneğin, Circle ve Rectangle), Factory Design Pattern ile şekil nesnelerini oluşturabiliriz. Bu, istemci kodunun hangi türde bir şekil oluşturulduğunu bilmek zorunda olmadan nesneleri yaratabilmesine olanak tanır.

2. 2. 1. FACTORY PATTERN UYGULAMASI

Factory Tasarım Deseni ile ilgili uygulamaya https://github.com/balfatih/YAZ16303_Yazilim_Mimarisi_ve_Tasarimi GITHUB reposundan erişebilirsiniz.

2. 3. ABSTRACY FACTORY PATTERN

Abstract Factory, creational (yaratımsal) bir tasarım desenidir ve birbirine bağımlı veya birbiriyle ilişkili nesne ailelerini oluşturmak ve kullanmak için kullanılır. Bu desen, bir nesne ailesinin oluşturulmasını, kullanılmasını ve değiştirilmesini soyutlar, böylece farklı bir nesne ailesini kullanmak istendiğinde sistemde değişiklik yapmak daha kolay hale gelir.

Abstract Factory deseni, Factory Method deseninin genişletilmiş bir versiyonudur. Factory Method deseni, tek bir üst sınıf içindeki alt sınıfların nesne oluşturma sorumluluğunu üstlenirken, Abstract Factory deseni birden fazla ilgili nesne grubu üreten ayrı ayrı Factory Methodlar içeren bir arayüz sağlar.

Bu desenin temel amacı, birbirleriyle uyumlu nesnelerden oluşan bir aile oluşturmak ve bu aileyi değiştirmenin kolay olmasını sağlamaktır. Bu, nesne ailesinin bir bütün olarak kullanılmasını sağlar ve bir parçasının değiştirilmesi durumunda sistemde minimum değişiklik yapılmasını sağlar.

ÖNEMLİ NOT:

Ders sırasında gerçekleştirdiğimiz örnekte, Factory Design Pattern yapısında, birbirleri ile benzer nesnelerin üretimini tek bir sınıfa bağlayarak ve tek bir arayüz üzerinden işlemleri gerçekleştirdik. Abstract Factory yapısında ise benzer nesne üretimlerini gerçekleştirebilmek adına her nesne için ayrı bir fabrika sınıfı oluşturmamız gerekmektedir. Ayrıca bu desen, birden fazla Interface kullanımı ile gerçekleştirilir. Factory Design Pattern'de bulunan koşullu durum(if/else if/else if…) problemi Abstract Factory'de çözülür.

2. 3. 1. ABSTRACT FACTORY PATTERN UYGULAMASI

Abstract Factory Tasarım Deseni ile ilgili uygulamaya https://github.com/balfatih/YAZ16303_Yazilim_Mimarisi_ve_Tasarimi GITHUB reposundan erişebilirsiniz.

2. 4. BUILDER DESIGN PATTERN

Builder tasarım deseni, bir nesnenin karmaşık bir yapıya sahip olduğu durumlarda nesneyi adım adım oluşturmayı ve temsil etmeyi amaçlayan bir tasarım desenidir. Bu desen, bir nesnenin farklı parçalarını oluşturan ve bir araya getiren bir "director" ve bu parçaları temsil eden bir dizi "builder" sınıfından oluşur.

Genellikle şu bileşenlere sahiptir:

Product (Ürün): Oluşturulacak nesneyi temsil eder.

Builder (Oluşturucu): Soyut bir sınıftır ve ürünün farklı parçalarını oluşturan metodları içerir.

ConcreteBuilder (Somut Oluşturucu): Builder sınıfını implemente eder ve belirli bir ürünü oluşturan metotları sağlar.

Director (Yönetici): Builder sınıflarını kullanarak bir ürünü oluşturan bir sınıftır. Yönetici sınıf, hangi adımların hangi sırayla çağrılacağını belirler.

Client (Müşteri): Builder desenini kullanarak nesneleri oluşturan sınıftır.

Builder deseni, özellikle bir nesnenin farklı varyasyonlarını oluşturmak istediğinizde veya bir nesnenin karmaşık bir yapıya sahip olduğu durumlarda kullanışlıdır. Bu desen, nesne oluşturma sürecini parçalara ayırarak, nesnenin oluşturulma sürecini daha esnek ve modüler hale getirir.

2. 4. 1. BUILDER DESIGN PATTERN UYGULAMASI

Builder Tasarım Deseni ile ilgili uygulamaya https://github.com/balfatih/YAZ16303_Yazilim_Mimarisi_ve_Tasarimi GITHUB reposundan erişebilirsiniz.