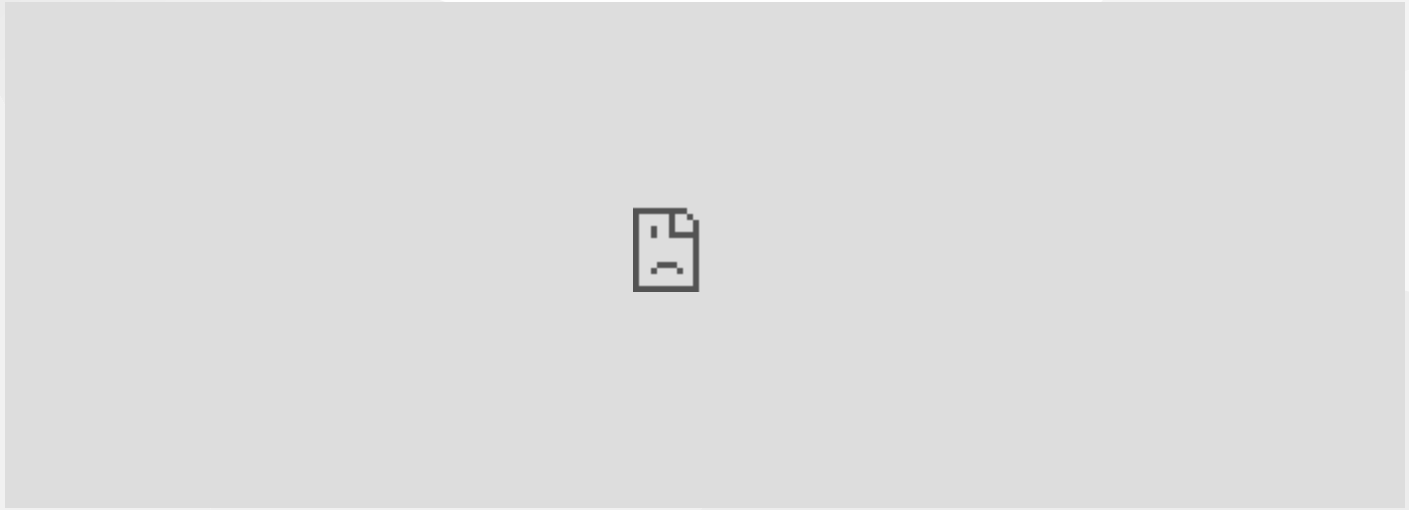
CEN206 Nesne Yönelimli Programlama

Hafta-9 (Nesne Yönelimli Tasarım İlkeleri ve Tasarım Desenleri)

Bahar Dönemi, 2024-2025

İndir BELGE-PDF, BELGE-DOCX, SLAYT, PPTX



Nesne Yönelimli Tasarım İlkeleri ve Tasarım Desenleri

Ana Hatlar

- Tasarım Desenleri
- SOLID İlkeleri
- Bağımlılık Enjeksiyonu ve Kontrolün Tersine Çevrilmesi
- Java'da Pratik Uygulamalar

Tasarım Desenlerine Giriş

Tasarım desenleri, yazılım tasarımındaki yaygın problemlere tipik çözümlerdir. Deneyimli yazılım geliştiricileri tarafından zaman içinde geliştirilmiş en iyi uygulamaları temsil ederler.

- **Tanım**: Yaygın tasarım problemleri için yeniden kullanılabilir çözüm şablonu
- **Faydalar**: Geliştirmeyi hızlandırır, kod kalitesini ve bakım yapılabilirliğini artırır
- **Kökenler**: Mimari desenlerden esinlenilmiştir (Christopher Alexander)

Mimaride ilk Tasarım Deseni kitabı:
https://www.amazon.com/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199

Dörtlü Çete (Gang of Four - GoF) Kitabı

Tasarım desenleri alanındaki temel eser, Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides (Dörtlü Çete) tarafından yazılan "Design Patterns: Elements of Reusable Object-Oriented Software" kitabıdır.

Bu kitap tasarım desenlerini şu kategorilere ayırır:

- **Yaratımsal Desenler (Creational Patterns)**: Nesne oluşturma mekanizmaları
- **Yapısal Desenler (Structural Patterns)**: Nesne kompozisyonu ve ilişkileri
- **Davranışsal Desenler (Behavioral Patterns)**: Nesne etkileşimi ve sorumluluk dağıtımı

Referans:
https://www.amazon.com/gp/product/0201633612/

Fabrika Metodu Deseni (Factory Method Pattern)

Fabrika Metodu, nesneler oluşturmak için bir arayüz tanımlar ancak hangi sınıfların örnekleneceğine alt sınıfların karar vermesini sağlar.

```
// Ürün arayüzü
interface Urun {
    void islem();
}

// Somut ürünler
class SomutUrunA implements Urun {
    @Override
    public void islem() {
        System.out.println("SomutUrunA işlemi");
    }
}

// Yaratıcı soyut sınıf
abstract class Yaratici {
    public abstract Urun urunOlustur();

    public void birIslemYap() {
        Urun urun = urunOlustur();
        urun.islem();
    }
}

// Somut yaratıcı
class SomutYaratici extends Yaratici {
    @Override
    public Urun urunOlustur() {
        return new SomutUrunA();
    }
}
```

SOLID İlkeleri

SOLID, yazılım tasarımlarını daha anlaşılır, esnek ve bakımı yapılabilir hale getirmeye yardımcı olan beş tasarım ilkesidir.

Bu beş ilke şunlardır:

1. **T**ek Sorumluluk İlkesi (Single Responsibility Principle)
2. **A**çık/Kapalı İlkesi (Open/Closed Principle)
3. **L**iskov Yerine Geçme İlkesi (Liskov Substitution Principle)
4. **A**rayüz Ayrımı İlkesi (Interface Segregation Principle)
5. **B**ağımlılığın Tersine Çevrilmesi İlkesi (Dependency Inversion Principle)

Kaynaklar:

- https://www.monterail.com/blog/solid-principles-cheatsheet-printable
- https://www.monterail.com/hubfs/PDF content/SOLID_cheatsheet.pdf
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

> "Bir sınıfın değişmesi için yalnızca bir nedeni olmalıdır."

Her sınıfın tek bir sorumluluğu veya amacı olmalıdır. Yazılımın yalnızca bir yönünü kapsüllenmelidir.

```
// SRP'yi ihlal eder
class Calisan {
    public void maasHesapla() { /* ... */ }
    public void veritabaninaKaydet() { /* ... */ }
    public void raporOlustur() { /* ... */ }
}

// SRP'yi takip eder
class Calisan {
    private String ad;
    private double maas;
    // Sadece çalışan özellikleri ve davranışları
}

class MaasHesaplayici {
    public double maasHesapla(Calisan calisan) { /* ... */ }
}

class CalisanDeposu {
    public void kaydet(Calisan calisan) { /* ... */ }
}

class RaporOlusturucu {
    public void raporOlustur(Calisan calisan) { /* ... */ }
```

Açık/Kapalı İlkesi (OCP)

> "Yazılım varlıkları genişletilmeye açık, değiştirilmeye kapalı olmalıdır."

Bir sınıfın davranışını değiştirmeden genişletebilmelisiniz.

```
// OCP'yi ihlal eder
class Dikdortgen {
    public double genislik;
    public double yukseklik;
}

class AlanHesaplayici {
    public double alanHesapla(Object sekil) {
        if (sekil instanceof Dikdortgen) {
            Dikdortgen dikdortgen = (Dikdortgen) sekil;
            return dikdortgen.genislik * dikdortgen.yukseklik;
        }
        // Yeni şekiller için daha fazla koşul ekle
        return 0;
    }
}

// OCP'yi takip eder
interface Sekil {
    double alanHesapla();
}

class Dikdortgen implements Sekil {
    private double genislik;
    private double yukseklik;

    @Override
    public double alanHesapla() {
        return genislik * yukseklik;
    }
}

class Daire implements Sekil {
    private double yaricap;

    @Override
    public double alanHesapla() {
        return Math.PI * yaricap * yaricap;
    }
}
```

"Alt tipler, temel tiplerinin yerini alabilir olmalıdır."

Bir üst sınıfın nesneleri, programın doğruluğunu etkilemeden alt sınıf nesneleriyle değiştirilebilir olmalıdır.

```
// LSP'yi ihlal eder
class Dikdortgen {
    protected int genislik;
    protected int yukseklik;

    public void genislikAyarla(int genislik) {
        this.genislik = genislik;
    }

    public void yukseklikAyarla(int yukseklik) {
        this.yukseklik = yukseklik;
    }

    public int alanGetir() {
        return genislik * yukseklik;
    }
}

class Kare extends Dikdortgen {
    @Override
    public void genislikAyarla(int genislik) {
        this.genislik = genislik;
        this.yukseklik = genislik;  // Kare her iki boyutu da değiştirir
    }

    @Override
    public void yukseklikAyarla(int yukseklik) {
        this.genislik = yukseklik;  // Kare her iki boyutu da değiştirir
        this.yukseklik = yukseklik;
    }
}

// LSP ihlali örneği
void dikdortgenTest(Dikdortgen d) {
    d.genislikAyarla(5);
    d.yukseklikAyarla(4);
    assert d.alanGetir() == 20; // Kare için başarısız olur
}
```

Arayüz Ayrımı İlkesi (ISP)

> "İstemciler kullanmadıkları arayüzlere bağlı olmaya zorlanmamalıdır."

Birçok istemciye özel arayüz, genel amaçlı bir arayüzden daha iyidir.

```
// ISP'yi ihlal eder
interface Calisan {
    void calis();
    void ye();
    void uyu();
}

class Robot implements Calisan {
    public void calis() { /* ... */ }
    public void ye() { /* Uygulanamaz */ }
    public void uyu() { /* Uygulanamaz */ }
}

// ISP'yi takip eder
interface Calisabilir {
    void calis();
}

interface Yiyebilir {
    void ye();
}

interface Uyuyabilir {
    void uyu();
}

class Insan implements Calisabilir, Yiyebilir, Uyuyabilir {
    public void calis() { /* ... */ }
    public void ye() { /* ... */ }
    public void uyu() { /* ... */ }
}

class Robot implements Calisabilir {
    public void calis() { /* ... */ }
}
```

> "Üst düzey modüller alt düzey modüllere bağlı olmamalıdır. Her ikisi de soyutlamalara bağlı olmalıdır."

> "Soyutlamalar ayrıntılara bağlı olmamalıdır. Ayrıntılar soyutlamalara bağlı olmalıdır."

```
// DIP'yi ihlal eder
class Ampul {
    public void ac() {
        // Işığı aç
    }

    public void kapa() {
        // Işığı kapa
    }
}

class Anahtar {
    private Ampul ampul;

    public Anahtar() {
        this.ampul = new Ampul();
    }

    public void calistir() {
        // Anahtarı çalıştırma mantığı
        ampul.ac();
    }
}

// DIP'yi takip eder
interface Acilabilir {
    void ac();
    void kapa();
}

class Ampul implements Acilabilir {
    public void ac() {
        // Işığı aç
    }

    public void kapa() {
        // Işığı kapa
    }
}

class Vantilatör implements Acilabilir {
    public void ac() {
        // Vantilatörü aç
    }

    public void kapa() {
        // Vantilatörü kapa
    }
}

class Anahtar {
    private Acilabilir cihaz;
    public Anahtar(Acilabilir cihaz) {
        this.cihaz = cihaz;
    }
```

Kontrolün Tersine Çevrilmesi (IoC) ve Bağımlılık Enjeksiyonu (DI)

Kontrolün Tersine Çevrilmesi, bir programın özel olarak yazılmış kısımlarının kontrol akışını genel bir çerçeveden almasını sağlayan bir tasarım ilkesidir.

Bağımlılık Enjeksiyonu, bir sınıfın bağımlılıklarının dışarıdan "enjekte edildiği" özel bir IoC biçimidir.

Kaynaklar:

- http://www.dotnet-stuff.com/tutorials/dependency-ınjection/understanding-and-implementing-inversion-of-control-container-ioc-container-using-csharp
- https://stackify.com/dependency-injection/
- https://www.tutorialsteacher.com/ioc/inversion-of-control
- https://www.wikiwand.com/en/Dependency_injection
- https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

```
    public Servis(Depo depo) {
        this.depo = depo;
    }
}
```

2. Ayarlayıcı Enjeksiyonu (Setter Injection)

Bağımlılıklar setter metotları aracılığıyla sağlanır.

```
class Servis {
    private Depo depo;

    public void depoAyarla(Depo depo) {
        this.depo = depo;
    }
}
```

3. Arayüz Enjeksiyonu (Interface Injection)

Bağımlılıklar bir arayüz metodu aracılığıyla sağlanır.

```
interface DepoEnjektoru {
```

Tasarım Desenleri ve SOLID'in Faydaları

- **Geliştirilmiş Kod Kalitesi**: Daha bakımı yapılabilir, esnek ve sağlam kod
- **Azaltılmış Karmaşıklık**: Karmaşık problemleri daha küçük, yönetilebilir parçalara ayırma
- **Daha İyi İletişim**: Tasarım çözümlerini tartışmak için ortak bir kelime dağarcığı
- **Daha Hızlı Geliştirme**: Yeniden icat etmek yerine kanıtlanmış çözümleri yeniden kullanma
- **Daha Kolay Test Etme**: Daha modüler kod daha kolay test edilebilir
- **Azaltılmış Teknik Borç**: Gelecekteki değişiklikler daha az yeniden çalışma gerektirir

Tasarımda Güvenlik En İyi Uygulamaları

Tasarım desenleri uygularken, güvenlik yönlerini de göz önünde bulundurun:

https://www.cisecurity.org/controls/cis-controls-list

- Kimlik doğrulama ve yetkilendirmenin doğru şekilde kapsüllendiğinden emin olun
- En az ayrıcalık ilkesini uygulayın
- Her sınırda veri doğrulamasını düşünün
- Bilgi sızdırmayan uygun hata işleme uygulayın
- Başlangıçtan itibaren güvenlik için tasarlayın

Kaynaklar

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Martin, R. C. (2003). Agile Software Development, Principles, Patterns, and Practices. Pearson.
- Freeman, E., Robson, E., Bates, B., Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.
- Refactoring Guru. (n.d.). Design Patterns. https://refactoring.guru/design-patterns
- Martin, R. C. (n.d.). The Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Ve sunum boyunca bağlantısı verilen tüm kaynaklar.

Önerilen Alıştırmalar

1. Basit bir uygulamada Fabrika Metodu desenini uygulayın
2. Mevcut bir kod tabanını SOLID ilkelerini uygulamak için yeniden düzenleyin
3. Bağımlılık Enjeksiyonu kullanarak küçük bir uygulama oluşturun
4. Mevcut çerçevelerde (Spring, JavaFX, vb.) tasarım desenlerini belirleyin
5. Belirli desenlerin ne zaman ve neden kullanılacağını açıklamayı alıştırın

Gelecek Hafta

Java'da daha fazla tasarım deseni ve bunların pratik uygulamalarını keşfetmeye devam edeceğiz.