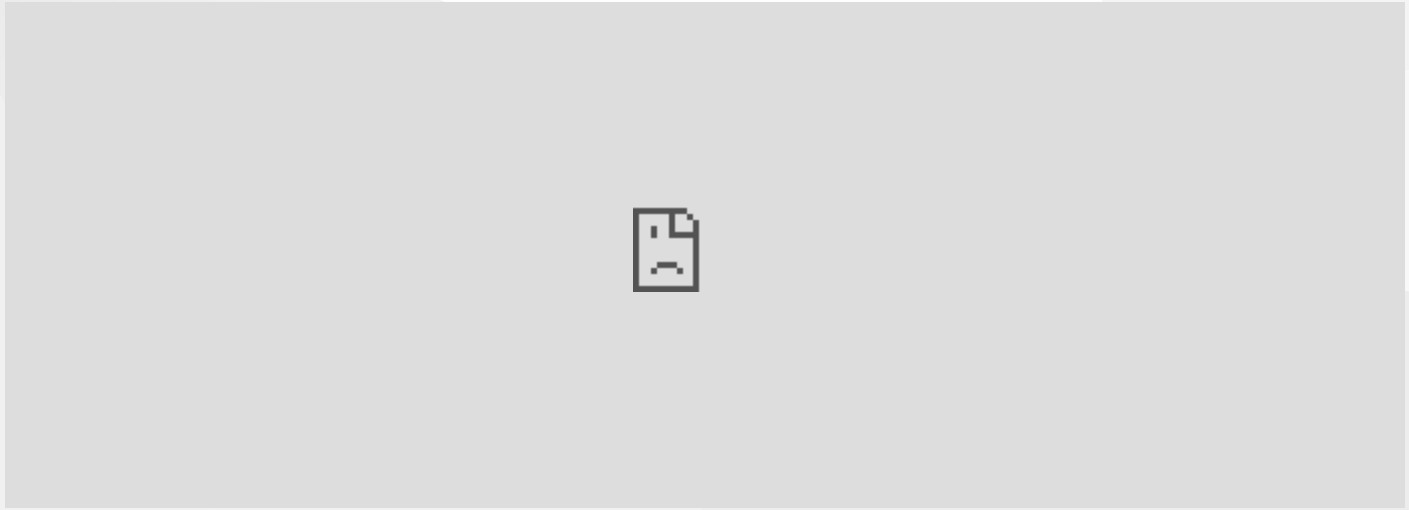
# CEN206 Nesne Yönelimli Programlama

## Hafta-11 (Gelişmiş Tasarım Desenleri & Pratik Uygulamalar)

### Bahar Dönemi, 2024-2025

İndir DOC-PDF, DOC-DOCX, SLIDE, PPTX



# Gelişmiş Tasarım Desenleri & Pratik Uygulamalar

## Ana Hatlar

- Ek Tasarım Desenleri
  - Kompozit (Composite) Deseni
  - Façade (Cephe) Deseni
  - Proxy (Vekil) Deseni
  - Command (Komut) Deseni
  - Template Method (Şablon Metot) Deseni
- Tasarım Desenlerinin Gerçek Dünya Uygulamaları
- Birden Fazla Desenin Birleştirilmesi
- Java Framework Vaka Çalışmaları

# Kompozit (Composite) Deseni

Nesneleri ağaç yapılarında bütün-parça hiyerarşilerini temsil edecek şekilde düzenlemenize olanak tanır. Kompozit, müşterilerin tek tek nesneleri ve nesne kompozisyonlarını aynı şekilde ele almasını sağlar.

```
// Bileşen (Component)
interface Component {
    void operation();
}

// Yaprak (Leaf)
class Leaf implements Component {
    private String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void operation() {
        System.out.println("Yaprak " + name + " işlemi");
    }
}

// Kompozit (Composite)
class Composite implements Component {
    private List<Component> children = new ArrayList<>();
    private String name;

    public Composite(String name) {
        this.name = name;
    }

    public void add(Component component) {
        children.add(component);
    }

    public void remove(Component component) {
        children.remove(component);
    }

    @Override
    public void operation() {
        System.out.println("Kompozit " + name + " işlemi");
        for (Component component : children) {
            component.operation();
        }
    }
```

# Kompozit Deseninin Kullanım Alanları

## Uygulama Senaryoları

- Dosya ve klasör yapıları
- GUI hiyerarşileri ve konteynerlar
- Organizasyon hiyerarşileri
- Alt menülerle menü sistemleri
- Bireysel öğelerin ve grupların aynı arayüze ihtiyaç duyduğu herhangi bir ağaç benzeri yapı

## Uygulama Hususları

- Hem yaprak hem de kompozit öğeler için ortak bir arayüz tanımlayın
- Ebeveyn referansları dahil edilip edilmeyeceğini düşünün
- Şeffaflık ve güvenlik yaklaşımları arasında seçim yapın
- Çocuk yönetimi işlemlerinin nasıl ele alınacağını belirleyin

# Façade (Cephe) Deseni

Bir alt sistemdeki arayüzler grubuna birleşik bir arayüz sağlar. Façade, alt sistemi kullanmayı kolaylaştıran daha üst düzey bir arayüz tanımlar.

```
// Karmaşık alt sistem sınıfları
class CPU {
    public void freeze() { System.out.println("CPU: Dondurma"); }
    public void jump(long position) { System.out.println("CPU: " + position + " konumuna atlama"); }
    public void execute() { System.out.println("CPU: Yürütme"); }
}

class Memory {
    public void load(long position, byte[] data) {
        System.out.println("Bellek: " + position + " konumundan yükleme");
    }
}

class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("Sabit Disk: " + lba + " LBA'dan " + size + " boyutunda okuma");
        return new byte[size];
    }
}

// Façade (Cephe)
class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void start() {
        cpu.freeze();
        memory.load(0, hardDrive.read(0, 1024));
        cpu.jump(0);
        cpu.execute();
    }
}
```

# Façade Deseninin Kullanım Alanları

## Uygulama Senaryoları

- Karmaşık alt sistemleri basitleştirme
- İstemci kodu için daha temiz bir API oluşturma
- İstemci kodunu alt sistem uygulama detaylarından ayırma
- Bir sistem yeniden düzenlendiğinde ve API'lerin gelişmesi gerektiğinde
- Uygulamanızın farklı katmanlarına giriş noktaları oluşturma

## Uygulama Hususları

- Façade, hafif bir sarmalayıcı olmalıdır
- Aynı alt sistem için birden fazla façade oluşturulabilir
- Mümkün olduğunda alt sistem sınıflarını paket-özel (package-private) yapmayı düşünün
- Façade'ı çok fazla sorumlulukla aşırı yüklemeyin

# Proxy (Vekil) Deseni

Başka bir nesneye erişimi kontrol etmek için bir vekil veya yer tutucu sağlar.

```
// Konu (Subject) arayüzü
interface Image {
    void display();
}

// Gerçek Konu (Real Subject)
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println(filename + " yükleniyor");
    }

    @Override
    public void display() {
        System.out.println(filename + " görüntüleniyor");
    }
}

// Proxy (Vekil)
class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

## Sanal Proxy (Virtual Proxy)

- Pahalı nesneleri talep üzerine oluşturur (tembel yükleme)
- Örnek: Bir belgedeki görüntü yükleme

## Koruma Proxy (Protection Proxy)

- Orijinal nesneye erişimi kontrol eder
- Örnek: Erişim kontrol sistemleri

## Uzak Proxy (Remote Proxy)

- Farklı bir adres alanında bulunan bir nesneyi temsil eder
- Örnek: Java'da RMI (Uzak Metot Çağrısı)

## Akıllı Proxy (Smart Proxy)

- Nesneye erişildiğinde ek işlemler gerçekleştirir

# Command (Komut) Deseni

Bir isteği nesne olarak kapsüller, böylece müşterileri farklı isteklerle parametreleştirmenize, istekleri sıraya koymanıza veya günlüğe kaydetmenize ve geri alınabilir işlemleri desteklemenize olanak tanır.

```
// Komut arayüzü
interface Command {
    void execute();
}

// Alıcı (Receiver)
class Light {
    public void turnOn() {
        System.out.println("Işık açık");
    }

    public void turnOff() {
        System.out.println("Işık kapalı");
    }
}

// Somut Komutlar
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}
```

```
// Çağırıcı (Invoker)
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

// İstemci kodu
// Light light = new Light();
// Command lightOn = new LightOnCommand(light);
// Command lightOff = new LightOffCommand(light);
// RemoteControl remote = new RemoteControl();
// remote.setCommand(lightOn);
// remote.pressButton();
// remote.setCommand(lightOff);
// remote.pressButton();
```

# Geri Alma İşlevi ile Komut Deseni

```
interface Command {
    void execute();
    void undo();
}

class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }

    @Override
    public void undo() {
        light.turnOff();
    }
}

class RemoteControlWithUndo {
    private Command command;
    private Stack<Command> history = new Stack<>();

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
        history.push(command);
    }

    public void pressUndo() {
        if (!history.isEmpty()) {
            Command lastCommand = history.pop();
            lastCommand.undo();
        }
    }
}
```

# Template Method (Şablon Metot) Deseni

Bir algoritmanın iskeletini bir metotta tanımlar, bazı adımları alt sınıflara bırakır. Şablon Metot, alt sınıfların algoritmanın yapısını değiştirmeden algoritmanın belirli adımlarını yeniden tanımlamasına olanak tanır.

```
// Şablon metot ile soyut sınıf
abstract class DocumentProcessor {
    // Şablon metot
    public final void processDocument() {
        openDocument();
        content = readContent();
        processContent(content);
        saveDocument();
        closeDocument();
    }

    // Alt sınıflar tarafından uygulanacak metotlar
    protected abstract String readContent();
    protected abstract void processContent(String content);

    // Varsayılan uygulamalara sahip ortak işlemler
    protected void openDocument() {
        System.out.println("Belge açılıyor");
    }

    protected void saveDocument() {
        System.out.println("Belge kaydediliyor");
    }

    protected void closeDocument() {
        System.out.println("Belge kapatılıyor");
    }
```

```java
// Somut uygulama
class PDFProcessor extends DocumentProcessor {
    @Override
    protected String readContent() {
        return "PDF içeriği";
    }

    @Override
    protected void processContent(String content) {
        System.out.println("PDF içeriği işleniyor: " + content);
    }
}

class WordProcessor extends DocumentProcessor {
    @Override
    protected String readContent() {
        return "Word belgesi içeriği";
    }

    @Override
    protected void processContent(String content) {
        System.out.println("Word içeriği işleniyor: " + content);
    }

    @Override
    protected void saveDocument() {
        System.out.println("Word belgesi özel formatla kaydediliyor");
    }
}
```

## Java Collections Framework

- **Iterator (Yineleyici) Deseni**: `java.util.Iterator`
- **Composite (Kompozit) Deseni**: Swing'deki bileşen hiyerarşileri
- **Strategy (Strateji) Deseni**: `java.util.Comparator`
- **Adapter (Adaptör) Deseni**: `java.io.InputStreamReader`, `OutputStreamWriter`

## Spring Framework

- **Factory (Fabrika) Deseni**: `BeanFactory`
- **Singleton (Tekil) Deseni**: Varsayılan bean kapsamı
- **Proxy (Vekil) Deseni**: AOP uygulaması
- **Template Method (Şablon Metot) Deseni**: `JdbcTemplate`, `HibernateTemplate`

## Android Geliştirme

- **Builder (İnşaatçı) Deseni**: `AlertDialog.Builder`
- **Observer (Gözlemci) Deseni**: Olay dinleyicileri

## Model-Görünüm-Kontrolör (MVC)

- **Observer (Gözlemci) Deseni**: Görünümler Modeli gözlemler
- **Composite (Kompozit) Deseni**: Görünümler alt görünümler içerebilir
- **Strategy (Strateji) Deseni**: Farklı kontrolörler farklı stratejiler uygular
- **Factory (Fabrika) Deseni**: Genellikle görünüm veya kontrolör oluşturmak için kullanılır

## Örnek: E-ticaret Sistemi

- **Factory/Builder (Fabrika/İnşaatçı)**: Ürün nesneleri oluştur
- **Decorator (Dekoratör)**: Ürünlere özellikler ekle (garanti, hediye paketi)
- **Observer (Gözlemci)**: Kullanıcıları fiyat değişiklikleri hakkında bilgilendir
- **Strategy (Strateji)**: Farklı ödeme yöntemleri
- **Command (Komut)**: Sipariş işleme düzeni

## Swing'de Desen Kullanımı

- **Composite (Kompozit)**: `JComponent` hiyerarşisi
- **Decorator (Dekoratör)**: Kenarlıklar gibi görsel dekorasyonlar
- **Observer (Gözlemci)**: Olay dinleyicileri
- **Strategy (Strateji)**: Düzen yöneticileri (BorderLayout, FlowLayout, vb.)
- **Factory Method (Fabrika Metodu)**: UI öğesi oluşturma

## Kod Örneği: Swing'de Observer (Gözlemci) Deseni

```
// ActionListener ile Observer deseni kullanımı
JButton button = new JButton("Tıkla Bana");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Düğmeye tıklandı!");
    }
});

// Composite deseni kullanımı
JPanel panel = new JPanel();
```

# Vaka Çalışması: Java Stream API

## Stream'lerde Desen Kullanımı

- **Builder (İnşaatçı) Deseni**: Stream oluşturma ve yapılandırma
- **Iterator (Yineleyici) Deseni**: Altta yatan dolaşma mekanizması
- **Strategy (Strateji) Deseni**: Farklı işlemler (map, filter, vb.)
- **Decorator (Dekoratör) Deseni**: İşlemleri zincirleme

## Kod Örneği

```
List<String> names = Arrays.asList("Ali", "Burak", "Canan", "Deniz");

// Birden çok desenin birlikte kullanımı
names.stream()
     .filter(name -> name.length() > 4)  // Strateji deseni
     .map(String::toUpperCase)           // Başka bir strateji
     .sorted()                           // Başka bir strateji daha
     .forEach(System.out::println);      // Terminal işlemi
```

# Alıştırma: Mini Belge Yönetim Sistemi Uygulaması

## Gereksinimler

1. Farklı belge türlerini (PDF, Word, Metin) işleyebilen bir sistem oluşturun
2. Belgeleri açma, okuma, düzenleme ve kaydetme gibi işlemleri destekleyin
3. Farklı kullanıcı türleri için erişim kontrolü uygulayın
4. Belge kompozisyonunu etkinleştirin (belgeler diğer belgeleri içerebilir)
5. Düzenlemeler için geri alma/yineleme işlevini destekleyin

## Uygulanacak Desenler

- **Factory (Fabrika)**: Belge oluşturma için
- **Strategy (Strateji)**: Farklı belge işleyicileri için
- **Composite (Kompozit)**: Belge kompozisyonu için
- **Command (Komut)**: Geri alma/yineleme ile işlemler için
- **Proxy (Vekil)**: Erişim kontrolü için

# Alıştırma Çözüm Taslağı

```
// Belge oluşturma için Factory deseni
class DocumentFactory {
    public Document createDocument(String type, String name) {
        switch (type.toLowerCase()) {
            case "pdf": return new PDFDocument(name);
            case "word": return new WordDocument(name);
            case "text": return new TextDocument(name);
            default: throw new IllegalArgumentException("Bilinmeyen belge türü");
        }
    }
}

// Belge yapısı için Composite deseni
interface Document {
    void open();
    void save();
    String getName();
}

class CompositeDocument implements Document {
    private String name;
    private List<Document> children = new ArrayList<>();

    // Uygulama detayları...
}
```

# Kaynaklar


- Freeman, E., Robson, E., Bates, B., Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Bloch, J. (2018). Effective Java (3rd Edition). Addison-Wesley.
- Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices. Pearson.
- Refactoring Guru: https://refactoring.guru/design-patterns
- Java Design Patterns: https://java-design-patterns.com/
- Spring Framework Reference: https://docs.spring.io/spring-framework/reference/

# Gelecek Hafta

UML (Unified Modeling Language - Birleşik Modelleme Dili) ve UMPLE'ı keşfedeceğiz, nesne yönelimli sistemleri modellemeye ve bu modellerden otomatik olarak kod üretmeye odaklanacağız.