CEN206 Nesne Yönelimli Programlama

Yapısal Tasarım Desenleri

Yazar: Öğr. Gör. Dr. Uğur CORUH

Şekil Listesi

Tablo Listesi

CEN206 Nesne Yönelimli Programlama

Hafta-10 (Yapısal Tasarım Desenleri)

Bahar Dönemi, 2025-2026 İndir DOC-PDF[1], DOC-DOCX[2], SLIDE[3]

Yapısal Tasarım Desenleri

Haftalık Ana Hatlar

- **Modül A:** Yapısal Desenlere Giriş
- **Modül B:** Adapter (Adaptör) Deseni
- **Modül C:** Bridge (Köprü) Deseni
- **Modül D:** Composite (Bileşik) Deseni
- **Modül E:** Decorator (Dekoratör) Deseni
- **Modül F:** Facade (Cephe) Deseni
- **Modül G:** Flyweight (Sinek Siklet) Deseni
- **Modül H:** Proxy (Vekil) Deseni

---

[1] ce204-week-10.tr.md_doc.pdf
[2] ce204-week-10.tr.md_word.docx
[3] ce204-week-10.tr.md_slide.pdf

Modül A: Yapısal Desenlere Giriş

Modül A: Ana Hatlar

- Yapısal Tasarım Desenleri Nedir?
- Yapısal Desenlerin Sınıflandırılması
- Yedi Yapısal Desenin Genel Görünümü
- Yapısal Desenleri Ne Zaman Kullanmalı
- Modül Özeti

Yapısal Tasarım Desenleri Nedir?

Yapısal tasarım desenleri, nesnelerin ve sınıfların **daha büyük yapılar halinde nasıl birleştirileceğini** açıklarken, bu yapıların **esnek ve verimli** kalmasını sağlar.

Ele aldıkları konular:

- **Sınıf bileşimi**: Sınıfların birbirinden nasıl miras aldığı
- **Nesne bileşimi**: Nesnelerin daha büyük yapılar oluşturmak için nasıl birleştirildiği
- **Arayüz basitleştirme**: Karmaşık alt sistemlerin istemcilere nasıl sunulduğu
- **Bellek optimizasyonu**: Paylaşılan durumun kaynak tüketimini nasıl azalttığı

Kaynak: refactoring.guru[4]

Yapısal Desenlerin Önemi

- **Ölçeklenebilir mimariler** oluşturmanıza yardımcı olurlar; bileşenler arasında net ilişkiler tanımlarlar
- Etkileşen sınıflar arasında **gevşek bağlılığı** teşvik ederler
- Çalışma zamanında nesnelerin **esnek bileşimini** sağlarlar
- **Sınıf hiyerarşileri** ve **nesne ilişkilerini** organize etmek için kanıtlanmış çözümler sunarlar
- Kalıtım yerine bileşim yoluyla **kod tekrarını** azaltırlar

---

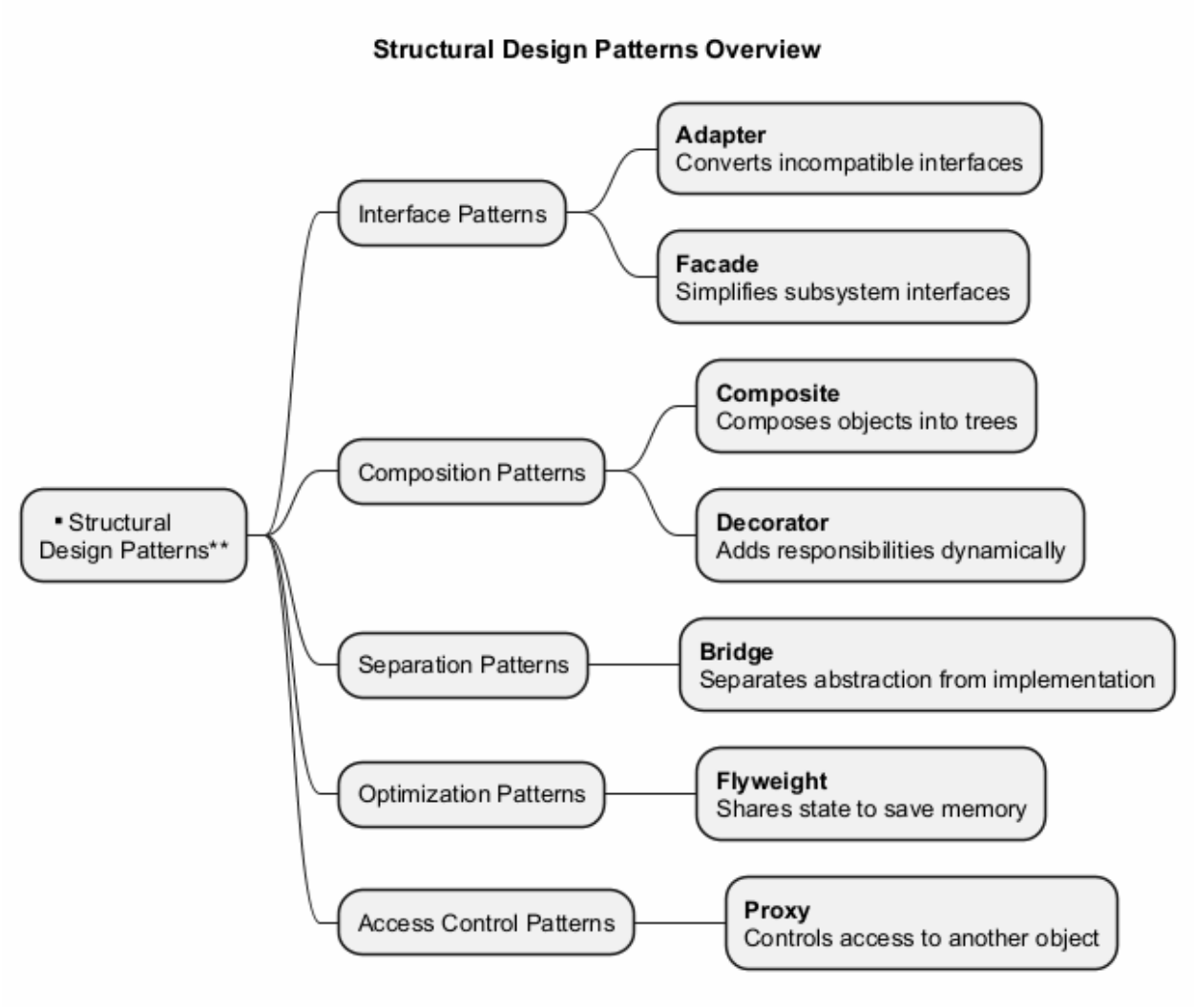
Yedi Yapısal Desen

Desen	Amacı
Adapter (Adaptör)	Uyumsuz arayüzlerin birlikte çalışmasını sağlar
Bridge (Köprü)	Soyutlamayı uygulamadan ayırır
Composite (Bileşik)	Nesneleri ağaç yapılarına dönüştürür
Decorator (Dekoratör)	Sorumlulukları dinamik olarak ekler
Facade (Cephe)	Karmaşık alt sistem arayüzlerini basitleştirir
Flyweight (Sinek Siklet)	Çok sayıda nesneyi desteklemek için durumu paylaşır
Proxy (Vekil)	Başka bir nesneye erişimi kontrol eder

Yapısal Desenler Genel Görünümü

---

[4] https://refactoring.guru/design-patterns/structural-patterns



Şekil 1: center

Amaca Göre Sınıflandırma

Arayüz Desenleri - Arayüzleri uyarlama veya basitleştirme ile ilgilenir: - Adapter (Adaptör), Facade (Cephe)

Bileşim Desenleri - Nesneleri yapılar halinde birleştirme ile ilgilenir: - Composite (Bileşik), Decorator (Dekoratör)

Ayrıştırma Desenleri - Kaygıları ayırma ile ilgilenir: - Bridge (Köprü)

Optimizasyon Desenleri - Kaynak yönetimi ile ilgilenir: - Flyweight (Sinek Siklet)

Erişim Kontrolü Desenleri - Nesne erişimini kontrol etme ile ilgilenir: - Proxy (Vekil)

Modül A: Özet

- Yapısal desenler, sınıfların ve nesnelerin daha büyük yapılar oluşturmak için nasıl **birleştirildiğini** ele alır
- Yapıların **esnek** ve **verimli** kalmasını sağlarlar
- Her biri ayrı bir yapısal problemi çözen **yedi** yapısal desen vardır
- Bu desenleri anlamak, **sürdürülebilir** nesne yönelimli sistemler oluşturmak için gereklidir

---

Modül B: Adapter (Adaptör) Deseni

Modül B: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Adapter (Adaptör): Amaç

Adapter (Adaptör), **uyumsuz arayüzlere** sahip nesnelerin birlikte çalışmasını sağlayan yapısal bir tasarım desenidir.

- Ayrıca şöyle de bilinir: **Wrapper (Sarmalayıcı)**
- Bir sınıfın arayüzünü, istemcilerin beklediği başka bir arayüze dönüştürür
- Uyumsuz arayüzler nedeniyle birlikte çalışamayacak sınıfların birlikte çalışmasını sağlar

Kaynak: refactoring.guru[5]

---

[5] https://refactoring.guru/design-patterns/adapter

Adapter (Adaptör): Problem

Bir **borsa izleme uygulaması** oluşturduğunuzu hayal edin. Uygulama, birden fazla kaynaktan **XML formatında** hisse senedi verilerini indirir.

Bir noktada, akıllı bir **üçüncü taraf analiz kütüphanesi** entegre ederek uygulamayı iyileştirmeye karar verirsiniz. Ancak bir sorun vardır: analiz kütüphanesi yalnızca **JSON formatındaki** verilerle çalışır.

Analiz kütüphanesini "olduğu gibi" kullanamazsınız çünkü uygulamanızla uyumsuz bir formatta veri bekler.

Adapter (Adaptör): Problem (devam)

Kütüphaneyi XML ile çalışacak şekilde değiştirebilirsiniz, ancak: - Bu, kütüphaneye bağımlı olan **mevcut kodu bozabilir** - Kütüphanenin **kaynak koduna erişiminiz olmayabilir**

Bu durum, **üçüncü taraf** veya **eski** kodu entegre ederken sıkça ortaya çıkar.

Adapter (Adaptör): Çözüm

Bir **adaptör** oluşturabilirsiniz – bir nesnenin arayüzünü başka bir nesnenin anlayabileceği şekilde dönüştüren özel bir nesne.

- Adaptör, perde arkasında gerçekleşen dönüşümün karmaşıklığını gizlemek için nesnelerden birini **sarar**
- Sarılan nesne adaptörün farkında bile değildir
- Adaptör çağrıları bir formatta alır ve bunları sarılan nesnenin anlayabileceği çağrılara çevirir

---

Adapter (Adaptör): Çözüm (devam)

Adaptörlerin çalışma şekli:

1. Adaptör, mevcut nesnelerden biriyle uyumlu bir arayüze sahip olur
2. Bu arayüzü kullanarak, mevcut nesne adaptörün yöntemlerini güvenle çağırabilir
3. Bir çağrı aldığında, adaptör isteği ikinci nesneye iletir, ancak ikinci nesnenin beklediği format ve sırada

Bazen her iki yönde de çağrıları dönüştürebilen **çift yönlü bir adaptör** oluşturmak bile mümkündür.

Adapter (Adaptör): Gerçek Dünya Benzetmesi

ABD'den Avrupa'ya ilk kez seyahat ettiğinizde, dizüstü bilgisayarınızı şarj etmeye çalışırken sürprizle karşılaşabilirsiniz. **Elektrik fişi ve priz** standartları farklı ülkelerde farklıdır.

Bir **elektrik fişi adaptörü** bu sorunu çözer. Bir ucunda Amerikan tarzı priz ve diğer ucunda Avrupa tarzı fiş bulunur; iki uyumsuz arayüz arasında çevirmen görevi görür.

Adapter (Adaptör): Yapı (Nesne Adaptörü)

Nesne Adaptörü bileşimi kullanır:

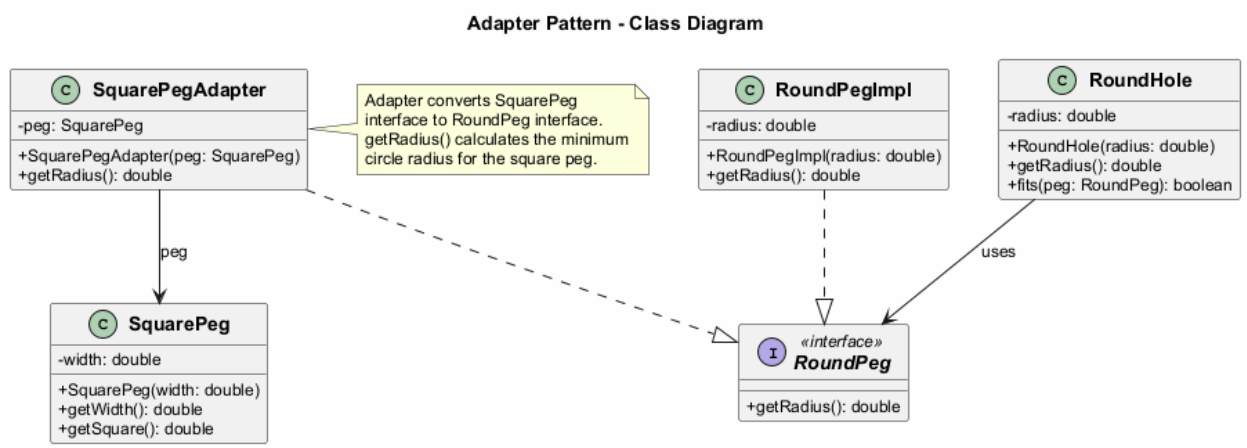
1. **İstemci** - Programın mevcut iş mantığını içeren bir sınıf
2. **İstemci Arayüzü** - İstemci koduyla işbirliği yapabilmek için diğer sınıfların uyması gereken bir protokol
3. **Servis** - Genellikle üçüncü taraf veya eski olan yararlı bir sınıf. İstemci bu sınıfı doğrudan kullanamaz çünkü uyumsuz bir arayüze sahiptir
4. **Adaptör** - Hem istemci hem de servis ile çalışabilen bir sınıf. Servis nesnesini sararken istemci arayüzünü uygular

Adapter (Adaptör): Yapı (Sınıf Adaptörü)

Sınıf Adaptörü çoklu kalıtım kullanır:

- Adaptör, hem istemci hem de servis sınıflarından **aynı anda** arayüzleri miras alır
- Bu yaklaşım yalnızca **çoklu kalıtımı** destekleyen dillerde mümkündür (örn., C++)
- Sarmalama gerekmez; uyarlama geçersiz kılınan yöntemlerde gerçekleşir
- Adaptör her iki taraftaki yöntemleri geçersiz kılarak aralarında çeviri yapar

Adapter - Sınıf Diyagramı



Şekil 2: center

Adapter (Adaptör): Java Kod Örneği - Arayüzler

```java
// The target interface that the client expects
public interface RoundPeg {
    double getRadius();
}

// Concrete implementation of round peg
public class RoundPegImpl implements RoundPeg {
    private double radius;

    public RoundPegImpl(double radius) {
        this.radius = radius;
    }

    @Override
    public double getRadius() {
        return radius;
    }
}
```

Adapter (Adaptör): Java Kod Örneği - Servis

```java
// The incompatible class (service) that we want to adapt
public class SquarePeg {
    private double width;

    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

    public double getSquare() {
        return Math.pow(this.width, 2);
    }
}
```

Adapter (Adaptör): Java Kod Örneği - Adaptör

```java
// The adapter makes SquarePeg compatible with RoundPeg
public class SquarePegAdapter implements RoundPeg {
    private SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        // Calculate the minimum circle radius that can
        // fit this square peg
        return (Math.sqrt(Math.pow(peg.getWidth(), 2) * 2)) / 2;
    }
}
```

Adapter (Adaptör): Java Kod Örneği - Yuvarlak Delik

```java
// The client class that works with RoundPeg interface
public class RoundHole {
    private double radius;

    public RoundHole(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public boolean fits(RoundPeg peg) {
        return this.getRadius() >= peg.getRadius();
    }
}
```

Adapter (Adaptör): Java Kod Örneği - İstemci

```
public class AdapterDemo {
    public static void main(String[] args) {
        RoundHole hole = new RoundHole(5);
        RoundPegImpl roundPeg = new RoundPegImpl(5);
        System.out.println(hole.fits(roundPeg)); // true

        SquarePeg smallSquarePeg = new SquarePeg(5);
        SquarePeg largeSquarePeg = new SquarePeg(10);

        // hole.fits(smallSquarePeg); // Won't compile!

        // Use adapter to make square pegs compatible
        SquarePegAdapter smallAdapter =
            new SquarePegAdapter(smallSquarePeg);
        SquarePegAdapter largeAdapter =
            new SquarePegAdapter(largeSquarePeg);

        System.out.println(hole.fits(smallAdapter)); // true
        System.out.println(hole.fits(largeAdapter)); // false
    }
}
```

Adapter - Sıralama Diyagramı

Adapter (Adaptör): Uygulanabilirlik

Adapter (Adaptör) desenini şu durumlarda kullanın:

- Mevcut bir sınıfı kullanmak istediğinizde, ancak arayüzü kodunuzun geri kalanıyla **uyumlu olmadığında**
- İlgisiz veya öngörülemeyen, uyumlu arayüzlere sahip olmayan sınıflarla işbirliği yapan **yeniden kullanılabilir bir sınıf** oluşturmak istediğinizde
- Birkaç mevcut alt sınıfı kullanmanız gerektiğinde, ancak her birinin arayüzünü alt sınıflayarak uyarlamak pratik olmadığında. Bir **nesne adaptörü** üst sınıfının arayüzünü uyarlayabilir

---

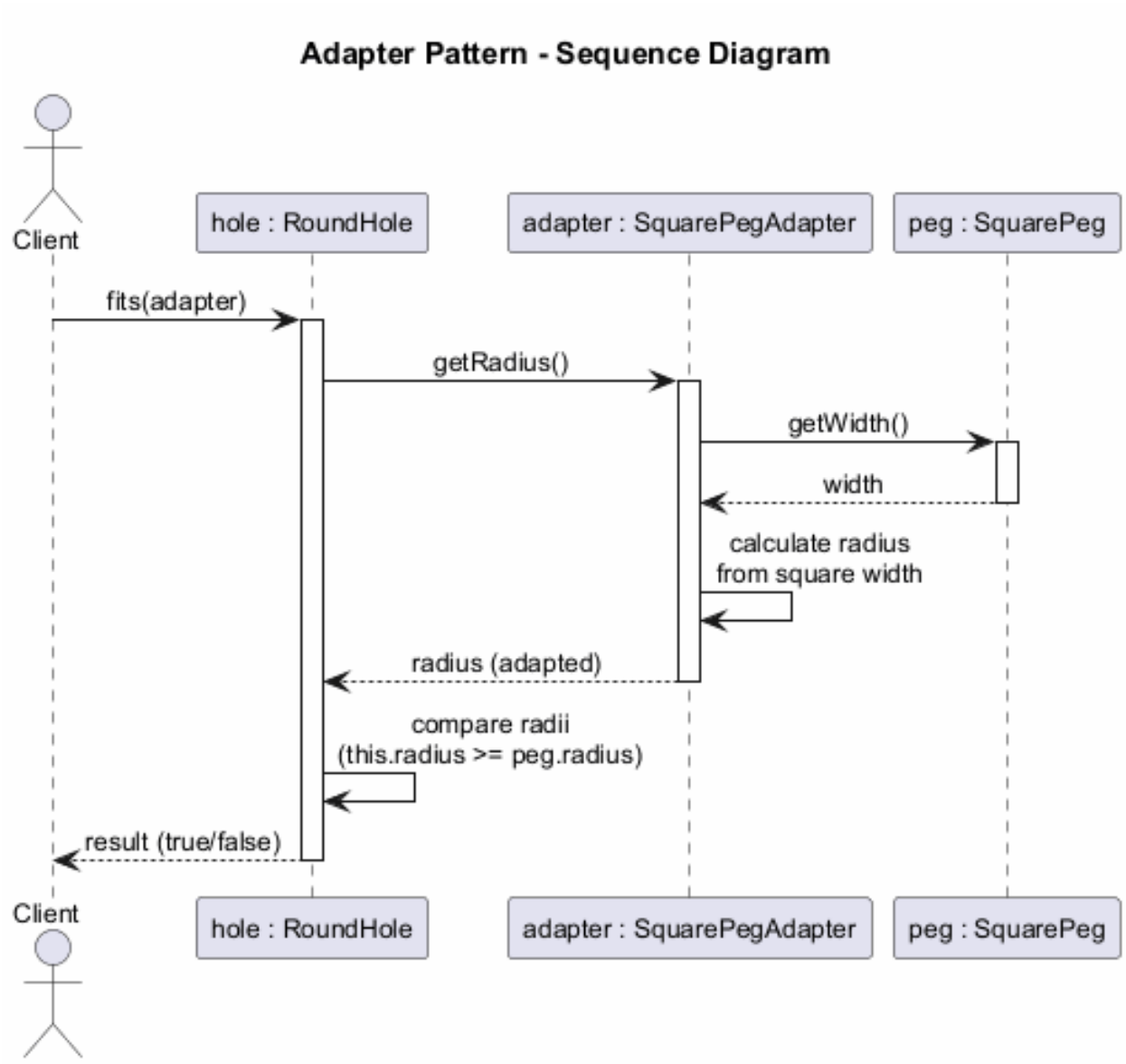
Adapter (Adaptör): Uygulanabilirlik (devam)

- Kodunuz ile eski bir sınıf, üçüncü taraf sınıf veya garip bir arayüze sahip başka bir sınıf arasında çevirmen görevi görecek bir **ara katman sınıfına** ihtiyacınız olduğunda
- Üst sınıfa eklenemeyen bazı ortak işlevlerden yoksun olan **mevcut alt sınıfları yeniden kullanmak** istediğinizde

---

Adapter (Adaptör): Nasıl Uygulanır

1. En az iki sınıfın **uyumsuz arayüzlere** sahip olduğundan emin olun: yararlı bir servis sınıfı ve servis sınıfını kullanmaktan fayda sağlayacak bir veya daha fazla istemci sınıfı
2. **İstemci arayüzünü** tanımlayın ve istemcilerin servis ile nasıl iletişim kurduğunu açıklayın



Şekil 3: center

3. **Adaptör sınıfını** oluşturun ve istemci arayüzünü uygulamasını sağlayın. Şimdilik tüm yöntemleri boş bırakın

---

Adapter (Adaptör): Nasıl Uygulanır (devam)

4. Adaptör sınıfına servis nesnesine referans saklamak için bir **alan ekleyin**. Yaygın uygulama bu alanı yapıcı aracılığıyla başlatmaktır, ancak bazen adaptörün yöntemlerini çağırırken iletmek daha uygundur
5. İstemci arayüzünün tüm yöntemlerini adaptör sınıfında tek tek **uygulayın**. Adaptör, gerçek işin çoğunu servis nesnesine devretmeli, yalnızca arayüz veya veri formatı dönüşümünü ele almalıdır
6. İstemciler adaptörü **istemci arayüzü** üzerinden kullanmalıdır. Bu, istemci kodunu etkilemeden adaptörleri değiştirmenize veya genişletmenize olanak tanır

---

Adapter (Adaptör): Avantajlar ve Dezavantajlar

Avantajlar: - **Tek Sorumluluk İlkesi**: Arayüz veya veri dönüştürme kodunu programın birincil iş mantığından ayırabilirsiniz - **Açık/Kapalı İlkesi**: İstemci arayüzü üzerinden adaptörlerle çalıştıkları sürece, mevcut istemci kodunu bozmadan programa yeni adaptör türleri ekleyebilirsiniz

Dezavantajlar: - Bir dizi yeni arayüz ve sınıf tanıtmanız gerektiğinden kodun genel karmaşıklığı artar. Bazen servis sınıfını kodunuzun geri kalanıyla eşleşecek şekilde değiştirmek daha basittir

---

Adapter (Adaptör): Diğer Desenlerle İlişkiler

- **Bridge (Köprü)** genellikle baştan tasarlanır ve bir uygulamanın parçalarını bağımsız olarak geliştirmenize olanak tanır. **Adapter (Adaptör)** genellikle mevcut bir uygulamayla birlikte kullanılır ve aksi takdirde uyumsuz olan sınıfların birlikte çalışmasını sağlar
- **Adapter (Adaptör)** mevcut bir nesnenin arayüzünü değiştirirken, **Decorator (Dekoratör)** arayüzünü değiştirmeden bir nesneyi geliştirir. Decorator (Dekoratör) özyinelemeli bileşimi desteklerken, Adapter (Adaptör) ile bu mümkün değildir
- **Adapter (Adaptör)** sarılan nesneye farklı bir arayüz sağlarken, **Proxy (Vekil)** aynı arayüzü sağlar ve **Decorator (Dekoratör)** geliştirilmiş bir arayüz sağlar

---

Adapter (Adaptör): Diğer Desenlerle İlişkiler (devam)

- **Facade (Cephe)** mevcut nesneler için yeni bir arayüz tanımlarken, **Adapter (Adaptör)** mevcut arayüzü kullanılabilir hale getirmeye çalışır. Adapter (Adaptör) genellikle yalnızca bir nesneyi sararken, Facade (Cephe) bütün bir nesne alt sistemiyle çalışır
- **Bridge (Köprü)**, **State**, **Strategy** (ve bir dereceye kadar **Adapter (Adaptör)**) çok benzer yapılara sahiptir. Tüm bu desenler, işi başka nesnelere devreden bileşime dayanır. Ancak hepsi farklı problemleri çözer

---

Modül B: Özet

- Adapter (Adaptör) deseni, iki uyumsuz arayüz arasında bir **köprü** görevi görür
- Mevcut bir sınıfı, istemci beklentileriyle uyumlu hale getirmek için yeni bir arayüzle sarar
- **Eski** veya **üçüncü taraf** kodu sisteminize entegre etmeniz gerektiğinde kullanın
- **Tek Sorumluluk** ve **Açık/Kapalı** ilkelerini takip eder
- İki varyantı vardır: **Nesne Adaptörü** (bileşim) ve **Sınıf Adaptörü** (çoklu kalıtım)

Modül C: Bridge (Köprü) Deseni

Modül C: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Bridge (Köprü): Amaç

Bridge (Köprü), büyük bir sınıfı veya birbiriyle yakından ilişkili bir dizi sınıfı **iki ayrı hiyerarşiye** – soyutlama ve uygulama – ayırmanızı sağlayan ve bunların birbirinden **bağımsız olarak** geliştirilebileceği yapısal bir tasarım desenidir.

Kaynak: refactoring.guru[6]

Bridge (Köprü): Problem

Diyelim ki bir geometrik `Shape` sınıfınız ve bir çift alt sınıfınız var: `Circle` ve `Square`. Bu sınıf hiyerarşisini renkleri de dahil edecek şekilde genişletmek istiyorsunuz, bu yüzden `Red` ve `Blue` şekil alt sınıfları oluşturmayı planlıyorsunuz.

Ancak zaten iki alt sınıfınız olduğundan, `BlueCircle` ve `RedSquare` gibi **dört** sınıf kombinasyonu oluşturmanız gerekecektir.

Bridge (Köprü): Problem (devam)

Hiyerarşiye yeni şekil türleri ve renkler eklemek hiyerarşiyi **üstel olarak** büyütecektir. Örneğin: - Bir **Triangle** şekli eklemek 2 yeni alt sınıf gerektirir (her renk için bir tane) - Bir **Green** renk eklemek 3 yeni alt sınıf gerektirir (her şekil için bir tane)

Bu sorun, şekil sınıflarını **iki bağımsız boyutta** genişletmeye çalışmamızdan kaynaklanır: biçim ve renk. Bu, sınıf kalıtımında çok yaygın bir sorundur.

Bridge (Köprü): Çözüm

Bridge (Köprü) deseni, **kalıtımdan nesne bileşimine** geçerek bu sorunu çözmeye çalışır. Bu, boyutlardan birini ayrı bir sınıf hiyerarşisine çıkarmanız anlamına gelir; böylece orijinal sınıflar, tüm durumu ve davranışları tek bir sınıfta tutmak yerine yeni hiyerarşideki bir nesneye referans verir.

---

[6] https://refactoring.guru/design-patterns/bridge

Bridge (Köprü): Çözüm (devam)

Bu yaklaşımı izleyerek, renkle ilgili kodu kendi sınıfına çıkarabiliriz: `Red` ve `Blue` olmak üzere iki alt sınıfla. `Shape` sınıfı daha sonra renk nesnelerinden birine işaret eden bir referans alanı alır.

Artık şekil, renkle ilgili herhangi bir işi bağlantılı renk nesnesine **devredebilir**. Bu referans, `Shape` ve `Color` sınıfları arasında bir **köprü** görevi görecektir.

Bundan sonra, yeni renkler eklemek şekil hiyerarşisini değiştirmeyi gerektirmeyecektir ve bunun tersi de geçerlidir.

Bridge (Köprü): Soyutlama ve Uygulama

Soyutlama (arayüz olarak da adlandırılır), bazı varlıklar için üst düzey kontrol katmanıdır. Bu katmanın kendi başına gerçek bir iş yapması beklenmez. İşi **uygulama** katmanına (platform olarak da adlandırılır) devretmelidir.

Not: Bunlar programlama dilinizdeki arayüzler veya soyut sınıflarla aynı şey değildir. Genel kavramlardır.

- **Soyutlama**: Bir uygulamanın GUI katmanı (örn., uzaktan kumanda)
- **Uygulama**: İşletim sistemi API'si, cihaza özgü kod (örn., TV, Radyo)

Bridge (Köprü): Gerçek Dünya Benzetmesi

Bir **uzaktan kumanda** ve bir **cihaz** (TV, Radyo) düşünün.

- Uzaktan kumanda **soyutlamadır** – üst düzey kontroller sağlar (güç, ses, kanal)
- Cihaz **uygulamadır** – bu kontrollerin gerçekte nasıl çalıştığını tanımlar
- Farklı uzaktan kumandaları farklı cihazlarla eşleştirebilirsiniz
- Cihaz sınıflarını değiştirmeden ekstra düğmelere sahip gelişmiş bir uzaktan kumanda oluşturabilirsiniz

---

Bridge (Köprü): Yapı

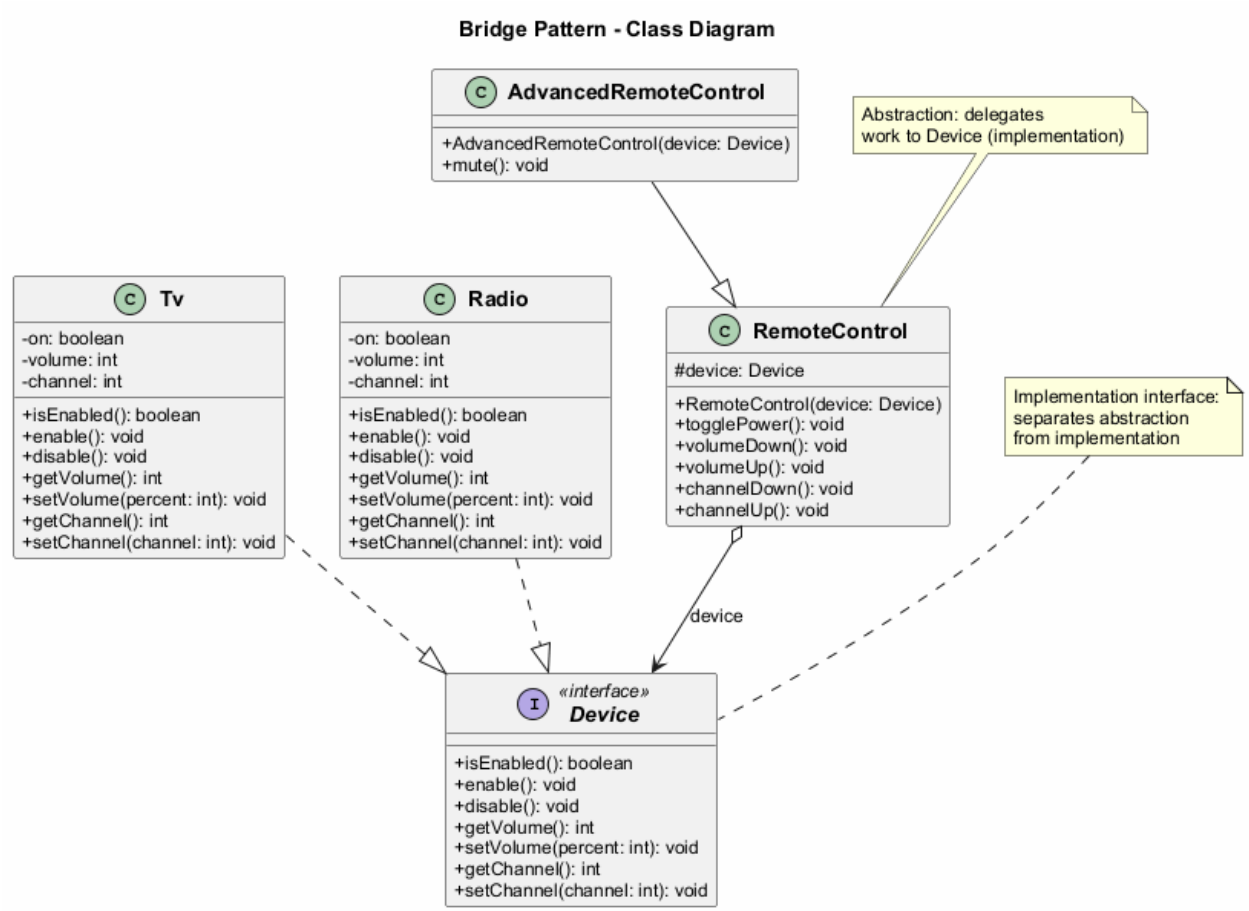
1. **Soyutlama** – Üst düzey kontrol mantığı sağlar. Gerçek alt düzey işi yapmak için uygulama nesnesine güvenir
2. **Uygulama** – Tüm somut uygulamalar için ortak olan arayüzü bildirir. Bir soyutlama, uygulama nesnesiyle yalnızca burada bildirilen yöntemler aracılığıyla iletişim kurabilir
3. **Somut Uygulamalar** – Platforma özgü kod içerir
4. **İyileştirilmiş Soyutlamalar** – Kontrol mantığının varyantlarını sağlar. Ebeveynleri gibi, farklı uygulamalarla genel uygulama arayüzü üzerinden çalışırlar
5. **İstemci** – Bir soyutlama nesnesini uygulama nesnelerinden biriyle bağlar

---

Bridge - Sınıf Diyagramı

Bridge (Köprü): Java Kod Örneği - Uygulama Arayüzü

```java
// The implementation interface declares methods common to all
// concrete implementation classes. It does not have to match the
// abstraction's interface.
public interface Device {
    boolean isEnabled();
    void enable();
    void disable();
```



Şekil 4: center

```
    int getVolume();
    void setVolume(int percent);
    int getChannel();
    void setChannel(int channel);
}
```

Bridge (Köprü): Java Kod Örneği - Somut Uygulamalar

```
public class Tv implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() { return on; }
    @Override
    public void enable() { on = true; }
    @Override
    public void disable() { on = false; }
    @Override
    public int getVolume() { return volume; }
    @Override
    public void setVolume(int percent) {
        this.volume = Math.max(0, Math.min(100, percent));
    }
    @Override
    public int getChannel() { return channel; }
    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }
}
```

Bridge (Köprü): Java Kod Örneği - Somut Uygulamalar (devam)

```
public class Radio implements Device {
    private boolean on = false;
    private int volume = 20;
    private int channel = 1;

    @Override
    public boolean isEnabled() { return on; }
    @Override
    public void enable() { on = true; }
    @Override
    public void disable() { on = false; }
    @Override
    public int getVolume() { return volume; }
    @Override
    public void setVolume(int percent) {
        this.volume = Math.max(0, Math.min(100, percent));
    }
    @Override
    public int getChannel() { return channel; }
```

```
    @Override
    public void setChannel(int channel) {
        this.channel = channel;
    }
}
```

Bridge (Köprü): Java Kod Örneği - Soyutlama

```
// The abstraction defines the interface for the "control" part
// of the two class hierarchies. It maintains a reference to an
// object of the implementation hierarchy and delegates all of
// the real work to this object.
public class RemoteControl {
    protected Device device;

    public RemoteControl(Device device) {
        this.device = device;
    }

    public void togglePower() {
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    public void volumeDown() {
        device.setVolume(device.getVolume() - 10);
    }

    public void volumeUp() {
        device.setVolume(device.getVolume() + 10);
    }

    public void channelDown() {
        device.setChannel(device.getChannel() - 1);
    }

    public void channelUp() {
        device.setChannel(device.getChannel() + 1);
    }
}
```

Bridge (Köprü): Java Kod Örneği - İyileştirilmiş Soyutlama

```
// You can extend the abstraction hierarchy independently
// from device classes.
public class AdvancedRemoteControl extends RemoteControl {
    public AdvancedRemoteControl(Device device) {
        super(device);
    }

    public void mute() {
```

```
        device.setVolume(0);
    }
}
```

Bridge (Köprü): Java Kod Örneği - İstemci

```
public class BridgeDemo {
    public static void main(String[] args) {
        // Pair a basic remote with a TV
        Device tv = new Tv();
        RemoteControl remote = new RemoteControl(tv);
        remote.togglePower();
        remote.volumeUp();
        System.out.println("TV Volume: " + tv.getVolume());

        // Pair an advanced remote with a Radio
        Device radio = new Radio();
        AdvancedRemoteControl advancedRemote =
            new AdvancedRemoteControl(radio);
        advancedRemote.togglePower();
        advancedRemote.mute();
        System.out.println("Radio Volume: " +
            radio.getVolume()); // 0
    }
}
```

Bridge - Sıralama Diyagramı

Bridge (Köprü): Uygulanabilirlik

Bridge (Köprü) desenini şu durumlarda kullanın:

- Bazı işlevselliğin birkaç varyantına sahip olan bir **monolitik sınıfı** bölmek ve organize etmek istediğinizde (örn., sınıf çeşitli veritabanı sunucularıyla çalışabiliyorsa)
- Bir sınıfı birkaç **ortogonal (bağımsız) boyutta** genişletmeniz gerektiğinde
- Çalışma zamanında **uygulamaları değiştirebilmeniz** gerektiğinde

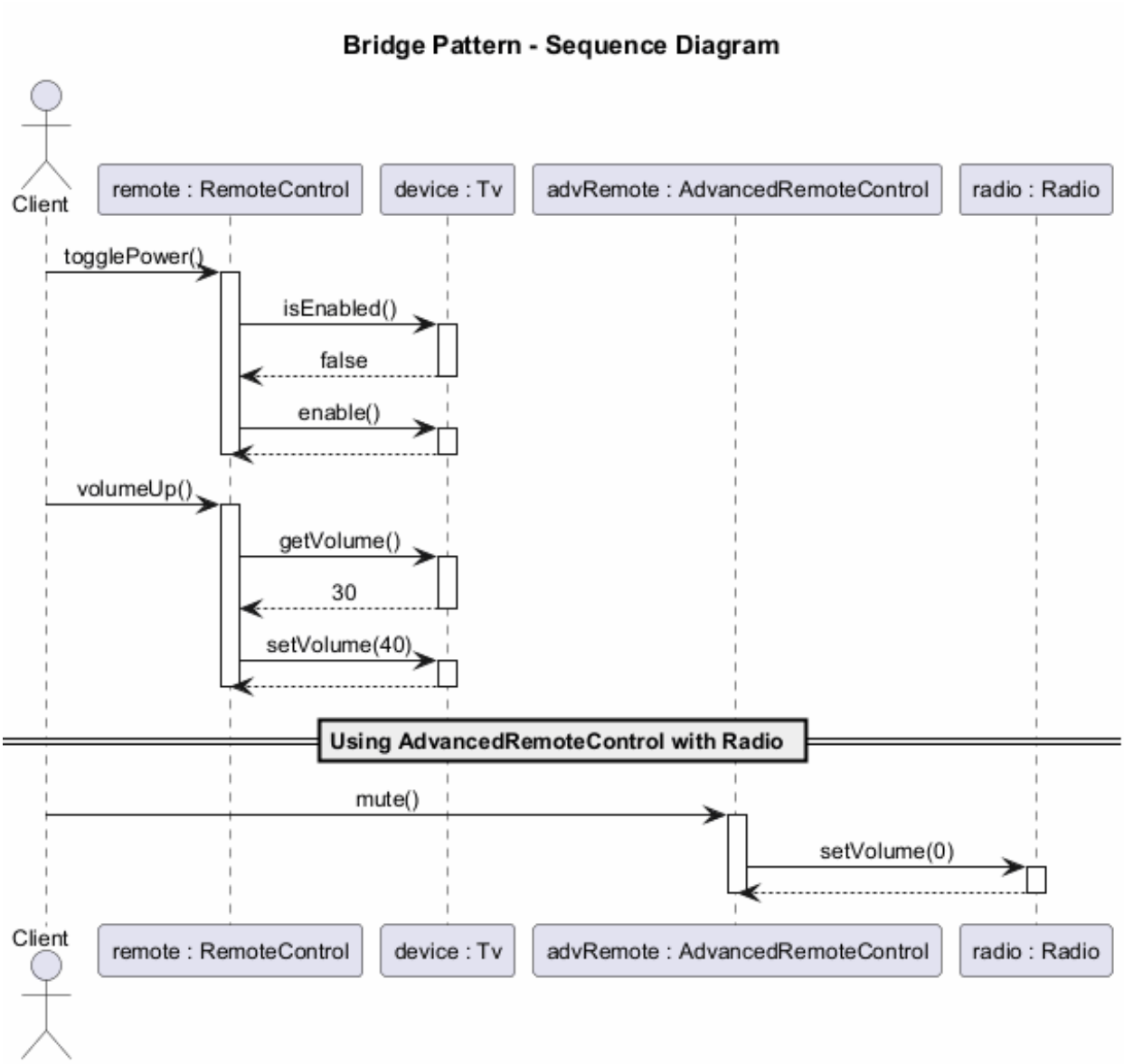
---

Bridge (Köprü): Uygulanabilirlik (devam)

Monolitik sınıf yeniden yapılandırma: Bir sınıf ne kadar büyürse, nasıl çalıştığını anlamak o kadar zorlaşır ve bir değişiklik yapmak o kadar uzun sürer. Bridge (Köprü), monolitik sınıfı her biri bağımsız olarak değiştirilebilen birkaç sınıf hiyerarşisine ayırmanıza olanak tanır.

Ortogonal boyutlar: Her boyut için ayrı bir sınıf hiyerarşisi çıkarın. Orijinal sınıf, her şeyi kendi başına yapmak yerine, ilgili işi bu hiyerarşilere ait nesnelere devreder.

Çalışma zamanı değiştirme: İsteğe bağlı olmakla birlikte, Bridge (Köprü) deseni, soyutlama içindeki uygulama nesnesini çalışma zamanında değiştirmenize olanak tanır. Bu, bir alana yeni bir değer atamak kadar kolaydır.



Şekil 5: center

Bridge (Köprü): Nasıl Uygulanır

1. Sınıflarınızdaki **ortogonal boyutları** belirleyin. Bu bağımsız kavramlar şunlar olabilir: soyutlama/platform, alan/altyapı, ön yüz/arka yüz veya arayüz/uygulama
2. **İstemcinin hangi işlemlere ihtiyaç duyduğunu** belirleyin ve bunları temel soyutlama sınıfında tanımlayın
3. **Tüm platformlarda** mevcut olan işlemleri belirleyin. Soyutlamanın ihtiyaç duyduğu işlemleri genel uygulama arayüzünde bildirin

---

Bridge (Köprü): Nasıl Uygulanır (devam)

4. Alanınızdaki tüm platformlar için **somut uygulama sınıfları** oluşturun, ancak hepsinin uygulama arayüzünü takip ettiğinden emin olun
5. Soyutlama sınıfının içine, uygulama türü için bir **referans alanı** ekleyin. Soyutlama, işin çoğunu bu alanda referans verilen uygulama nesnesine devreder
6. Birkaç **üst düzey mantık** varyantınız varsa, temel soyutlama sınıfını genişleterek her varyant için **iyileştirilmiş soyutlamalar** oluşturun
7. **İstemci kodu**, birini diğeriyle ilişkilendirmek için soyutlamanın yapıcısına bir uygulama nesnesi iletmelidir. Bundan sonra istemci, uygulamayı unutabilir ve yalnızca soyutlama nesnesiyle çalışabilir

---

Bridge (Köprü): Avantajlar ve Dezavantajlar

Avantajlar: - **Platformdan bağımsız** sınıflar ve uygulamalar oluşturabilirsiniz - İstemci kodu üst düzey soyutlamalarla çalışır ve **platform detaylarına maruz kalmaz** - **Açık/Kapalı İlkesi**: Yeni soyutlamaları ve uygulamaları birbirinden bağımsız olarak tanıtabilirsiniz - **Tek Sorumluluk İlkesi**: Soyutlamadaki üst düzey mantığa ve uygulamadaki platform detaylarına odaklanabilirsiniz

Dezavantajlar: - Deseni yüksek uyumlu bir sınıfa uygulayarak kodu **daha karmaşık** hale getirebilirsiniz

---

Bridge (Köprü): Diğer Desenlerle İlişkiler

- **Bridge (Köprü)** genellikle baştan tasarlanır ve bir uygulamanın parçalarını bağımsız olarak geliştirmenize olanak tanır. Öte yandan, **Adapter (Adaptör)** genellikle mevcut bir uygulamayla birlikte kullanılır ve aksi takdirde uyumsuz olan sınıfların birlikte çalışmasını sağlar
- **Bridge (Köprü)**, **State**, **Strategy** (ve bir dereceye kadar **Adapter (Adaptör)**) çok benzer yapılara sahiptir. Tüm bu desenler bileşime dayanır. Ancak hepsi farklı problemleri çözer
- **Abstract Factory**'yi **Bridge (Köprü)** ile birlikte kullanabilirsiniz. Bu eşleşme, Bridge (Köprü) tarafından tanımlanan bazı soyutlamaların yalnızca belirli uygulamalarla çalışabildiği durumlarda faydalıdır
- **Builder**'ı **Bridge (Köprü)** ile birleştirebilirsiniz: yönetici sınıf soyutlama rolünü oynarken, farklı inşacılar uygulama görevi görür

---

Modül C: Özet

- Bridge (Köprü) deseni, **soyutlamayı uygulamadan ayırır** ve her ikisinin bağımsız olarak değişebilmesini sağlar
- Sınıf hiyerarşilerindeki **kombinatoryal patlamayı** çözer
- Farklı boyutları birleştirmek için kalıtım yerine **bileşim** kullanır
- **Açık/Kapalı** ve **Tek Sorumluluk** ilkelerini takip eder
- Sonradan uyarlamak yerine (bunun için Adapter (Adaptör) kullanılır) **baştan tasarlayın**

Modül D: Composite (Bileşik) Deseni

Modül D: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Composite (Bileşik): Amaç

Composite (Bileşik), nesneleri **ağaç yapıları** halinde birleştirmenize ve ardından bu yapılarla sanki **bireysel nesneler** gibi çalışmanıza olanak tanıyan yapısal bir tasarım desenidir.

- Ayrıca şöyle de bilinir: **Object Tree (Nesne Ağacı)**

Kaynak: refactoring.guru[7]

Composite (Bileşik): Problem

Composite (Bileşik) desenini kullanmak, yalnızca uygulamanızın temel modeli bir **ağaç** olarak temsil edilebildiğinde anlamlıdır.

Örneğin, iki tür nesneniz olduğunu düşünün: `Products` ve `Boxes`. Bir `Box`, birkaç `Products` ve birkaç daha küçük `Boxes` içerebilir. Bu küçük `Boxes` da bazı `Products` veya daha küçük `Boxes` içerebilir, vb.

Composite (Bileşik): Problem (devam)

Bu sınıfları kullanarak bir sipariş sistemi oluşturmaya karar verdiğinizi düşünün. Siparişler, herhangi bir ambalaj olmadan basit ürünlerin yanı sıra ürünlerle ve diğer kutularla doldurulmuş kutuları da içerebilir.

Böyle bir siparişin **toplam fiyatını** nasıl belirlersiniz?

Doğrudan bir yaklaşım deneyebilirsiniz: tüm kutuları açın, tüm ürünleri gözden geçirin ve ardından toplamı hesaplayın. Ancak bu, belirli sınıfları ve iç içe geçme yapısını önceden bilmeyi gerektirir, bu da kodu yapıya sıkıca bağlar.

[7] https://refactoring.guru/design-patterns/composite

Composite (Bileşik): Çözüm

Composite (Bileşik) deseni, `Products` ve `Boxes` ile toplam fiyatı hesaplamak için bir yöntem bildiren **ortak bir arayüz** üzerinden çalışmanızı önerir.

- Bir **ürün** için basitçe ürünün fiyatını döndürür
- Bir **kutu** için kutunun içerdiği her öğeyi gözden geçirir, fiyatını sorar ve ardından bu kutu için toplamı döndürür

En büyük fayda, ağacı oluşturan nesnelerin somut sınıflarını bilmenize gerek olmamasıdır. Hepsini ortak arayüz üzerinden aynı şekilde işleyebilirsiniz.

Composite (Bileşik): Çözüm (devam)

Bir yöntem çağırdığınızda, nesneler isteği ağaçta aşağıya iletir. Bir nesne ağacının tüm bileşenleri üzerinde **özyinelemeli olarak bir davranış çalıştırabilirsiniz**.

Bu yaklaşım şu nedenle çalışır: - Bir ürün fiyatını doğrudan döndürür - Bir kutu içeriğini yineler ve her öğe üzerinde fiyat yöntemini çağırır - Bir öğe başka bir kutuysa, o kutu da içeriğini yineler, vb. - Sonuç olarak tüm ağaç yapısı otomatik olarak dolaşılır

---

Composite (Bileşik): Gerçek Dünya Benzetmesi

Çoğu ülkenin **orduları** hiyerarşiler olarak yapılandırılmıştır. Bir ordu birkaç tümenden oluşur; bir tümen bir dizi tugaydır ve bir tugay bölüklerden oluşur, bunlar da takımlara ayrılabilir. Son olarak, bir takım küçük bir gerçek asker grubudur.

Emirler hiyerarşinin tepesinde verilir ve her seviyeye iletilerek her askerin ne yapması gerektiğini bilmesini sağlar. Bu, Composite (Bileşik) deseninin işi ağaç boyunca nasıl devrettiğini yansıtır.

Composite (Bileşik): Yapı

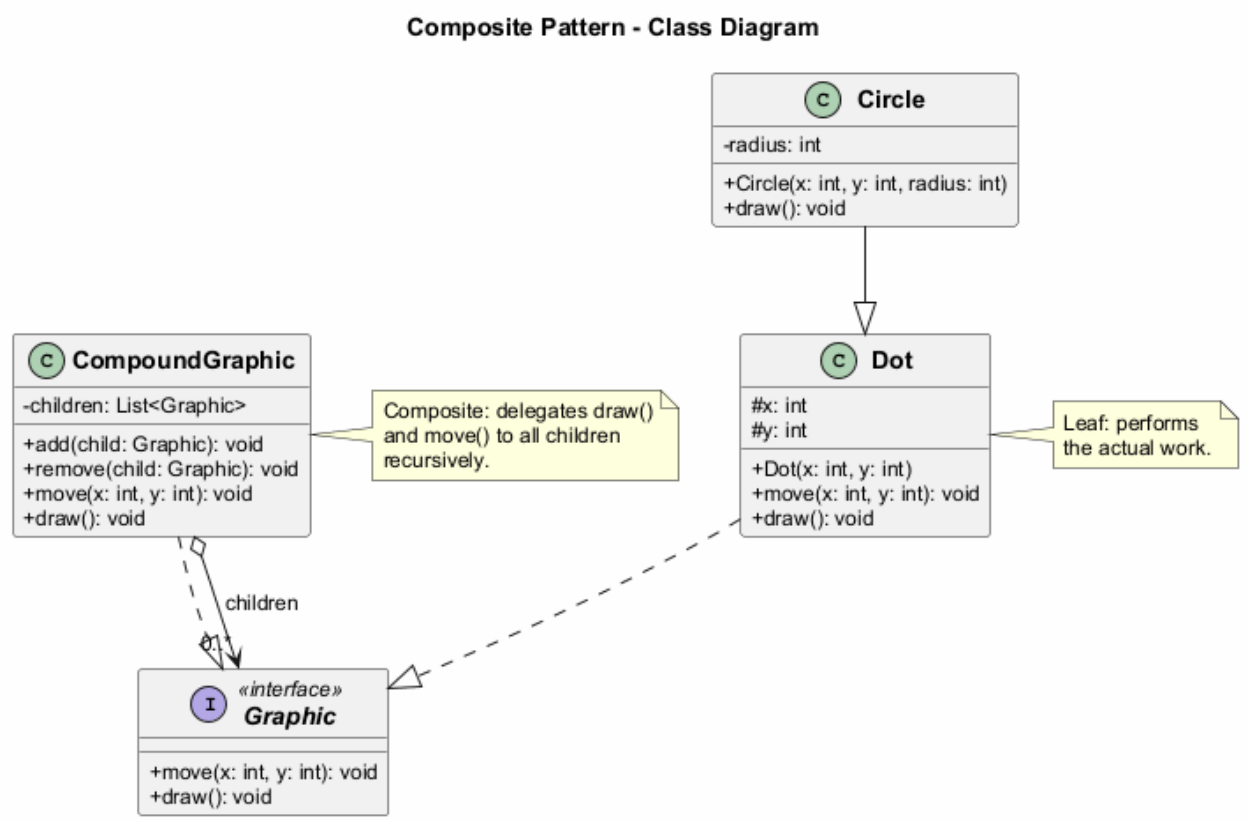
1. **Bileşen** – Ağacın hem basit hem de karmaşık elemanlarında ortak olan işlemleri tanımlayan arayüz
2. **Yaprak** – Alt elemanları olmayan, ağacın temel bir elemanı. Genellikle yaprak bileşenler, işi devredecek kimseleri olmadığından gerçek işin çoğunu yapar
3. **Kapsayıcı (Bileşik)** – Alt elemanlara sahip bir eleman: yapraklar veya diğer kapsayıcılar. Bir kapsayıcı, alt elemanlarının somut sınıflarını bilmez. Tüm alt elemanlarla yalnızca bileşen arayüzü üzerinden çalışır
4. **İstemci** – Tüm elemanlarla bileşen arayüzü üzerinden çalışır. Sonuç olarak istemci, ağacın hem basit hem de karmaşık elemanlarıyla aynı şekilde çalışabilir

---

Composite - Sınıf Diyagramı

Composite (Bileşik): Java Kod Örneği - Bileşen

```java
// The component interface declares common operations for
// both simple and complex objects of a composition.
public interface Graphic {
    void move(int x, int y);
    void draw();
}
```



Şekil 6: center

Composite (Bileşik): Java Kod Örneği - Yapraklar

```
// Leaf classes represent end objects of a composition.
// A leaf object cannot have any sub-objects.
public class Dot implements Graphic {
    protected int x, y;

    public Dot(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public void move(int x, int y) {
        this.x += x;
        this.y += y;
    }

    @Override
    public void draw() {
        System.out.println("Draw dot at (" + x + ", " + y + ")");
    }
}
```

Composite (Bileşik): Java Kod Örneği - Yapraklar (devam)

```java
public class Circle extends Dot {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Draw circle at (" + x + ", " + y
            + ") with radius " + radius);
    }
}
```

Composite (Bileşik): Java Kod Örneği - Bileşik

```java
import java.util.ArrayList;
import java.util.List;

// The Composite class represents complex components that may
// have children. It delegates work to its children and then
// aggregates the result.
public class CompoundGraphic implements Graphic {
    private List<Graphic> children = new ArrayList<>();

    public void add(Graphic child) {
        children.add(child);
    }

    public void remove(Graphic child) {
        children.remove(child);
    }

    @Override
    public void move(int x, int y) {
        for (Graphic child : children) {
            child.move(x, y);
        }
    }

    @Override
    public void draw() {
        System.out.println("---- CompoundGraphic ----");
        for (Graphic child : children) {
            child.draw();
        }
        System.out.println("---- End Compound ----");
    }
}
```

Composite (Bileşik): Java Kod Örneği - İstemci

```java
public class CompositeDemo {
    public static void main(String[] args) {
        // Create simple components
        Dot dot1 = new Dot(1, 2);
        Dot dot2 = new Dot(3, 4);
        Circle circle = new Circle(5, 6, 10);

        // Create a composite and add components
        CompoundGraphic group1 = new CompoundGraphic();
        group1.add(dot1);
        group1.add(circle);

        // Create another composite containing the first
        CompoundGraphic group2 = new CompoundGraphic();
        group2.add(dot2);
        group2.add(group1); // Nesting composites

        // Draw the entire tree
        group2.draw();

        // Move all elements
        group2.move(10, 10);
        group2.draw();
    }
}
```

Composite - Sıralama Diyagramı

Composite (Bileşik): Uygulanabilirlik

Composite (Bileşik) desenini şu durumlarda kullanın:

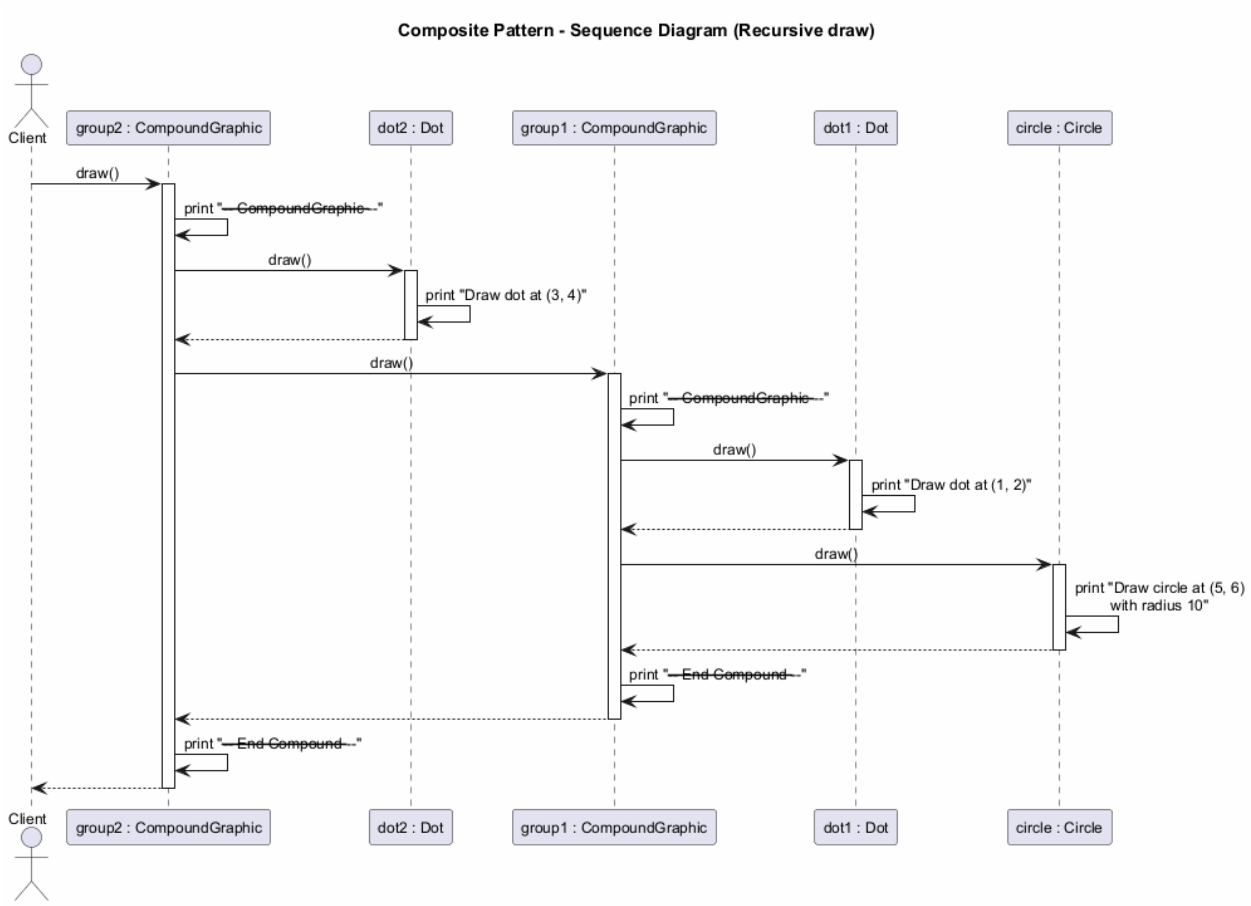
- Bir **ağaç benzeri nesne yapısı** uygulamanız gerektiğinde. Composite (Bileşik) deseni, ortak bir arayüzü paylaşan iki temel eleman türü sağlar: basit yapraklar ve karmaşık kapsayıcılar
- İstemci kodunun hem basit hem de karmaşık elemanları **aynı şekilde** ele almasını istediğinizde. Composite (Bileşik) deseni tarafından tanımlanan tüm elemanlar ortak bir arayüzü paylaşır, böylece istemci çalıştığı nesnelerin somut sınıfı hakkında endişelenmek zorunda kalmaz

---

Composite (Bileşik): Nasıl Uygulanır

1. Uygulamanızın temel modelinin bir **ağaç yapısı** olarak temsil edilebildiğinden emin olun. Basit elemanlara ve kapsayıcılara ayırmaya çalışın. Kapsayıcıların hem basit elemanları hem de diğer kapsayıcıları içerebilmesi gerektiğini unutmayın
2. Hem basit hem de karmaşık bileşenler için anlamlı yöntemlerin bir listesiyle **bileşen arayüzünü** bildirin
3. Basit elemanları temsil edecek bir **yaprak sınıfı** oluşturun. Bir programda birden fazla farklı yaprak sınıfı olabilir

---



Şekil 7: center

Composite (Bileşik): Nasıl Uygulanır (devam)

4. Karmaşık elemanları temsil edecek bir **kapsayıcı sınıfı** oluşturun. Bu sınıfta, alt elemanlara referansları saklamak için bir dizi alanı sağlayın. Dizi hem yaprakları hem de kapsayıcıları saklayabilmelidir, bu nedenle bileşen arayüzü türüyle bildirildiğinden emin olun
5. Kapsayıcının yöntemlerini uygularken, bir kapsayıcının **işin çoğunu alt elemanlara devretmesi** gerektiğini unutmayın
6. Son olarak, kapsayıcıda alt eleman **ekleme ve kaldırma** yöntemlerini tanımlayın. Bu işlemlerin bileşen arayüzünde bildirilebileceğini, bunun Arayüz Ayrıştırma İlkesini ihlal edeceğini, ancak istemcinin tüm elemanları eşit olarak ele almasına olanak tanıyacağını unutmayın

---

Composite (Bileşik): Avantajlar ve Dezavantajlar

Avantajlar: - Karmaşık ağaç yapılarıyla daha rahat çalışabilirsiniz: **polimorfizm ve özyinelemeyi** avantajınıza kullanın - **Açık/Kapalı İlkesi**: Nesne ağacıyla çalışan mevcut kodu bozmadan uygulamaya yeni eleman türleri ekleyebilirsiniz

Dezavantajlar: - İşlevsellikleri çok farklı olan sınıflar için **ortak bir arayüz** sağlamak zor olabilir. Bazı senaryolarda bileşen arayüzünü aşırı genelleştirmeniz gerekebilir, bu da onu anlamayı zorlaştırır

---

Composite (Bileşik): Diğer Desenlerle İlişkiler

- Karmaşık Composite (Bileşik) ağaçları oluştururken **Builder** kullanabilirsiniz çünkü inşa adımlarını özyinelemeli olarak çalışacak şekilde programlayabilirsiniz
- **Chain of Responsibility** genellikle Composite (Bileşik) ile birlikte kullanılır. Bir yaprak bileşen bir istek aldığında, bunu nesne ağacının köküne kadar tüm üst bileşenler zincirinden geçirebilir
- Composite (Bileşik) ağaçlarını dolaşmak için **Iterators** kullanabilirsiniz
- Tüm Composite (Bileşik) ağacı üzerinde bir işlem yürütmek için **Visitor** kullanabilirsiniz

---

Composite (Bileşik): Diğer Desenlerle İlişkiler (devam)

- Composite (Bileşik) ağacının paylaşılan yaprak düğümlerini RAM'den tasarruf etmek için **Flyweight (Sinek Siklet)** olarak uygulayabilirsiniz
- **Composite (Bileşik)** ve **Decorator (Dekoratör)** benzer yapı diyagramlarına sahiptir çünkü her ikisi de özyinelemeli bileşime dayanır. Ancak bir Decorator (Dekoratör), yalnızca bir alt bileşene sahip bir Composite (Bileşik) gibidir. Decorator (Dekoratör) sarılan nesneye ek sorumluluklar eklerken, Composite (Bileşik) yalnızca alt elemanlarının sonuçlarını "toplar"
- Composite (Bileşik) ve Decorator (Dekoratör)'ü yoğun olarak kullanan tasarımlar genellikle **Prototype** kullanmaktan fayda sağlayabilir. Deseni uygulamak, karmaşık yapıları sıfırdan yeniden oluşturmak yerine klonlamanıza olanak tanır

---

Modül D: Özet

- Composite (Bileşik) deseni, nesneleri **ağaç yapıları** halinde birleştirmenize ve bunları aynı şekilde ele almanıza olanak tanır
- Hem basit (yaprak) hem de karmaşık (bileşik) elemanlar için **ortak bir arayüz** kullanır
- Desen, kapsayıcıların hem yaprakları hem de diğer kapsayıcıları tutabildiği **özyinelemeli bileşimi** sağlar
- **Parça-bütün hiyerarşilerini** temsil etmek için idealdir
- Mevcut kodu değiştirmeden yeni eleman türlerine izin vererek **Açık/Kapalı İlkesini** takip eder

---

Modül E: Decorator (Dekoratör) Deseni

Modül E: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Decorator (Dekoratör): Amaç

Decorator (Dekoratör), nesneleri davranışları içeren özel **sarmalayıcı nesnelerin** içine yerleştirerek nesnelere **yeni davranışlar** eklemenize olanak tanıyan yapısal bir tasarım desenidir.

- Ayrıca şöyle de bilinir: **Wrapper (Sarmalayıcı)**

Kaynak: refactoring.guru[8]

Decorator (Dekoratör): Problem

Diğer programların kullanıcılarını önemli olaylar hakkında bilgilendirmesine olanak tanıyan bir **bildirim kütüphanesi** üzerinde çalıştığınızı hayal edin.

Kütüphanenin ilk sürümü, yalnızca birkaç alanı, bir yapıcıyı ve tek bir `send` yöntemi olan `Notifier` sınıfına dayanıyordu. Yöntem, bir istemciden mesaj argümanı alabilir ve mesajı bir e-posta listesine gönderebilirdi.

Decorator (Dekoratör): Problem (devam)

Bir noktada, kütüphanenin kullanıcılarının yalnızca e-posta bildirimlerinden fazlasını beklediğini fark edersiniz. Birçoğu kritik sorunlar hakkında **SMS** almak ister. Diğerleri **Facebook**'ta bilgilendirilmek ister. Ve kurumsal kullanıcılar **Slack** bildirimleri almayı çok ister.

Bu ne kadar zor olabilir? `Notifier` sınıfını genişletir ve ek bildirim yöntemlerini yeni alt sınıflara koyarsınız. Artık istemcinin istediği bildirim sınıfını örneklemesi ve tüm sonraki bildirimler için kullanması beklenir.

Decorator (Dekoratör): Problem (devam)

Ama sonra birisi sorar: "Neden **aynı anda birden fazla bildirim türü** kullanamıyorsunuz?" Örneğin, eviniz yanıyorsa, muhtemelen her kanal üzerinden bilgilendirilmek istersiniz.

Her kombinasyon için özel alt sınıflar oluşturarak bunu çözmeye çalışırsınız. Ancak bu yaklaşımın kodu aşırı şişirdiği çabucak ortaya çıkar – alt sınıfların **kombinatoryal patlaması** kod tabanını bakım yapılamaz hale getirir.

---

[8] https://refactoring.guru/design-patterns/decorator

Decorator (Dekoratör): Çözüm

Bir nesnenin davranışını değiştirmeniz gerektiğinde, akla gelen ilk şey bir sınıfı genişletmektir. Ancak kalıtımın birkaç ciddi sakıncası vardır:

- Kalıtım **statiktir** – çalışma zamanında mevcut bir nesnenin davranışını değiştiremezsiniz
- Alt sınıfların yalnızca **bir üst sınıfı** olabilir (çoğu dilde)

Alternatiflerden biri, Kalıtım yerine **Bileşim/Birleştirme** kullanmaktır.

Decorator (Dekoratör): Çözüm (devam)

Bir **sarmalayıcı**, bazı **hedef** nesnelerle bağlantılı olabilen bir nesnedir. Sarmalayıcı, hedefle aynı yöntem kümesini içerir ve aldığı tüm istekleri hedefe devreder. Ancak sarmalayıcı, isteği hedefe iletmeden **önce veya sonra** bir şeyler yaparak sonucu değiştirebilir.

Basit bir sarmalayıcı ne zaman gerçek bir dekoratör olur? Sarmalayıcı, sarılan nesneyle **aynı arayüzü** uygular. Bu nedenle istemcinin bakış açısından bu nesneler aynıdır.

Decorator (Dekoratör): Çözüm (devam)

Birden fazla sarmalayıcı tek bir nesnenin etrafına **yığılarak** tüm davranışları birleştirebilir:

```
Encryption(Compression(FileDataSource))
```

İstemci, aynı arayüzü takip eden herhangi bir dekoratörle nesneyi sarabilir. Tüm sarmalayıcılar, yeni alt sınıflar oluşturmadan davranışlarını birleştirir.

Decorator (Dekoratör): Gerçek Dünya Benzetmesi

Giysi giymek dekoratör kullanımının bir örneğidir. Üşüdüğünüzde kendinizi bir kazağa sararsınız. Hâlâ üşüyorsanız üzerine bir ceket giyebilirsiniz. Yağmur yağıyorsa bir yağmurluk giyebilirsiniz.

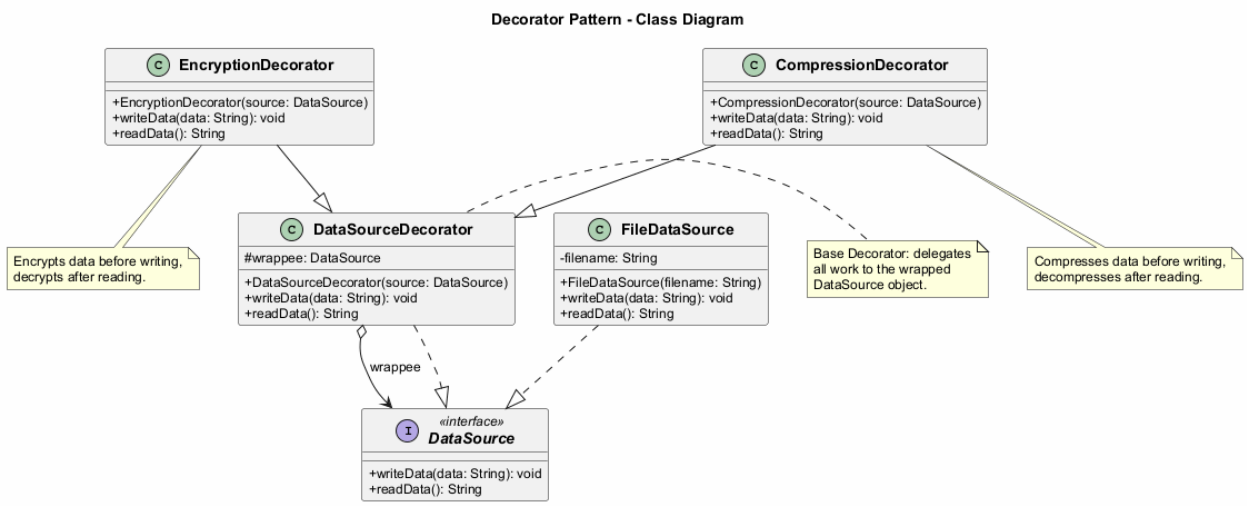
Tüm bu giysiler temel davranışınızı **genişletir** ancak sizin parçanız değildir ve ihtiyacınız olmadığında herhangi bir giysi parçasını kolayca **çıkarabilirsiniz**.

Decorator (Dekoratör): Yapı

1. **Bileşen** – Hem sarmalayıcılar hem de sarılan nesneler için ortak arayüzü bildirir
2. **Somut Bileşen** – Sarılan nesnelerin sınıfı. Dekoratörler tarafından değiştirilebilen temel davranışı tanımlar
3. **Temel Dekoratör** – Sarılan bir nesneye referans tutan bir alana sahip sınıf. Alanın türü, hem somut bileşenleri hem de dekoratörleri içerebilmesi için bileşen arayüzü olarak bildirilmelidir. Temel dekoratör tüm işlemleri sarılan nesneye devreder
4. **Somut Dekoratörler** – Bileşenlere dinamik olarak eklenebilen ekstra davranışları tanımlar. Temel dekoratörün yöntemlerini geçersiz kılar ve davranışlarını üst yöntem çağrısından önce veya sonra yürütür
5. **İstemci** – Bileşenleri birden çok dekoratör katmanıyla sarabilir, tüm nesnelerle bileşen arayüzü üzerinden çalıştığı sürece

---

Decorator - Sınıf Diyagramı



Şekil 8: center

Decorator (Dekoratör): Java Kod Örneği - Bileşen Arayüzü

```
// The component interface defines operations that can be
// altered by decorators.
public interface DataSource {
    void writeData(String data);
    String readData();
}
```

Decorator (Dekoratör): Java Kod Örneği - Somut Bileşen

```
import java.io.*;

// Concrete components provide default implementations
// of the operations.
public class FileDataSource implements DataSource {
    private String filename;

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

    @Override
    public void writeData(String data) {
        try (FileWriter writer = new FileWriter(filename)) {
            writer.write(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String readData() {
        try (BufferedReader reader =
                new BufferedReader(new FileReader(filename))) {
            StringBuilder sb = new StringBuilder();
```

```
            String line;
            while ((line = reader.readLine()) != null) {
                sb.append(line);
            }
            return sb.toString();
        } catch (IOException e) {
            return "";
        }
    }
}
```

Decorator (Dekoratör): Java Kod Örneği - Temel Dekoratör

```
// The base decorator class follows the same interface as
// the other components. The primary purpose of this class
// is to define the wrapping interface for all concrete
// decorators. The default implementation of the wrapping
// code might include a field for storing a wrapped
// component and the means to initialize it.
public class DataSourceDecorator implements DataSource {
    protected DataSource wrappee;

    public DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

Decorator (Dekoratör): Java Kod Örneği - Somut Dekoratörler

```
import java.util.Base64;

// Concrete decorators must call methods on the wrapped
// object, but may add something of their own to the result.
public class EncryptionDecorator extends DataSourceDecorator {
    public EncryptionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        // Encrypt data before writing
        String encrypted = Base64.getEncoder()
            .encodeToString(data.getBytes());
        super.writeData(encrypted);
```

```
    }

    @Override
    public String readData() {
        // Decrypt data after reading
        String data = super.readData();
        return new String(Base64.getDecoder().decode(data));
    }
}
```

Decorator (Dekoratör): Java Kod Örneği - Somut Dekoratörler (devam)

```
import java.io.ByteArrayOutputStream;
import java.util.zip.*;

public class CompressionDecorator extends DataSourceDecorator {
    public CompressionDecorator(DataSource source) {
        super(source);
    }

    @Override
    public void writeData(String data) {
        // Compress data before writing
        String compressed = compress(data);
        super.writeData(compressed);
    }

    @Override
    public String readData() {
        // Decompress data after reading
        String data = super.readData();
        return decompress(data);
    }

    private String compress(String data) {
        // Compression logic here
        return data; // Simplified
    }

    private String decompress(String data) {
        // Decompression logic here
        return data; // Simplified
    }
}
```

Decorator (Dekoratör): Java Kod Örneği - İstemci

```
public class DecoratorDemo {
    public static void main(String[] args) {
        String salaryRecords =
            "Name,Salary\nJohn Smith,100000\nSteven Jobs,912000";

        // Simple writing without any decorators
        DataSource source = new FileDataSource("output.txt");
```

```
        source.writeData(salaryRecords);

        // Writing with encryption
        DataSource encrypted = new EncryptionDecorator(
            new FileDataSource("encrypted.txt"));
        encrypted.writeData(salaryRecords);

        // Writing with compression AND encryption
        // Note how decorators are stacked
        DataSource compressedAndEncrypted =
            new EncryptionDecorator(
                new CompressionDecorator(
                    new FileDataSource("compressed.txt")));
        compressedAndEncrypted.writeData(salaryRecords);

        // Reading decrypts and decompresses automatically
        System.out.println(compressedAndEncrypted.readData());
    }
}
```

Decorator - Sıralama Diyagramı

Decorator (Dekoratör): Uygulanabilirlik

Decorator (Dekoratör) desenini şu durumlarda kullanın:

- Bu nesneleri kullanan kodu bozmadan **çalışma zamanında nesnelere ekstra davranışlar** atayabilmeniz gerektiğinde
- **Kalıtım** kullanarak bir nesnenin davranışını genişletmek garip veya imkansız olduğunda (örn., `final` sınıflar)
- Bir nesneyi birden fazla dekoratörle sararak **birkaç davranışı birleştirmeniz** gerektiğinde

---

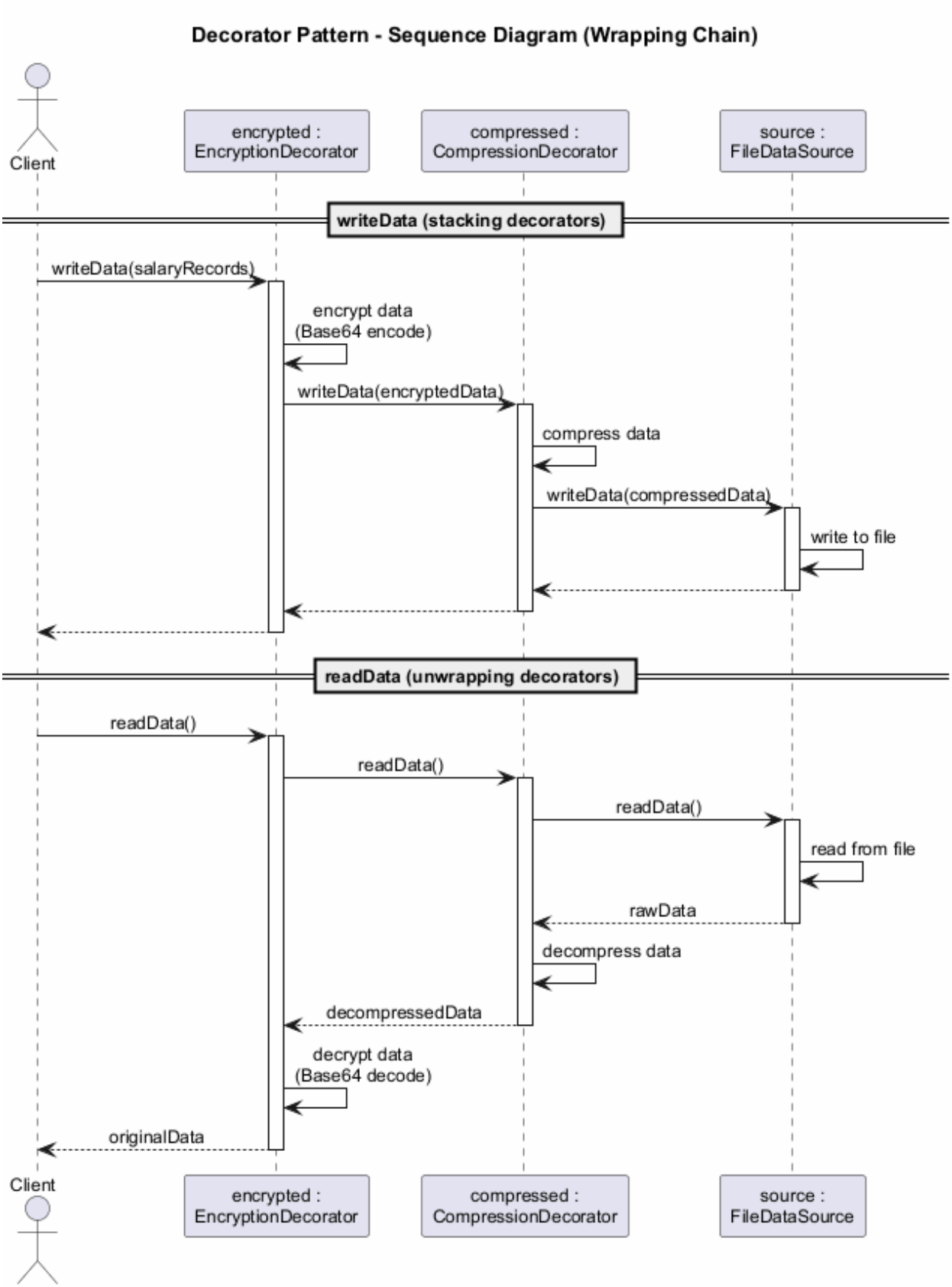
Decorator (Dekoratör): Uygulanabilirlik (devam)

Decorator (Dekoratör), iş mantığınızı **katmanlar** halinde yapılandırmanıza, her katman için bir dekoratör oluşturmanıza ve çalışma zamanında bu mantığın çeşitli kombinasyonlarıyla nesneleri birleştirmenize olanak tanır. İstemci kodu, hepsi ortak bir arayüzü takip ettiği için tüm bu nesneleri aynı şekilde ele alabilir.

Decorator (Dekoratör): Nasıl Uygulanır

1. İş alanınızın, üzerinde birden fazla isteğe bağlı katman bulunan bir **birincil bileşen** olarak temsil edilebildiğinden emin olun
2. Hem birincil bileşen hem de isteğe bağlı katmanlar için hangi yöntemlerin ortak olduğunu belirleyin. Bir **bileşen arayüzü** oluşturun ve bu yöntemleri orada bildirin
3. Bir **somut bileşen** sınıfı oluşturun ve temel davranışı içinde tanımlayın

---



Şekil 9: center

Decorator (Dekoratör): Nasıl Uygulanır (devam)

4. Bir **temel dekoratör** sınıfı oluşturun. Sarılan bir nesneye referans saklamak için bir alanı olmalıdır. Alan, hem somut bileşenlere hem de dekoratörlere bağlantı kurulmasına izin vermek için bileşen arayüzü türüyle bildirilmelidir. Temel dekoratör tüm işi sarılan nesneye devretmelidir
5. Tüm sınıfların **bileşen arayüzünü** uyguladığından emin olun
6. Temel dekoratörden genişleterek **somut dekoratörler** oluşturun. Somut bir dekoratör, davranışını üst yöntem çağrısından (her zaman sarılan nesneye devreder) önce veya sonra yürütmelidir
7. **İstemci kodu**, dekoratörleri oluşturmaktan ve bunları istemcinin ihtiyaç duyduğu şekilde birleştirmekten sorumlu olmalıdır

---

Decorator (Dekoratör): Avantajlar ve Dezavantajlar

Avantajlar: - Bir nesnenin davranışını **yeni bir alt sınıf oluşturmadan** genişletebilirsiniz - **Çalışma zamanında** bir nesneden sorumluluklar ekleyebilir veya kaldırabilirsiniz - Bir nesneyi birden fazla dekoratörle sararak birkaç davranışı **birleştirebilirsiniz** - **Tek Sorumluluk İlkesi**: Birçok olası davranış varyantını uygulayan monolitik bir sınıfı birkaç daha küçük sınıfa bölebilirsiniz

Dezavantajlar: - Sarmalayıcılar yığınından **belirli bir sarmalayıcıyı kaldırmak** zordur - Davranışı dekoratörler yığınındaki **sıraya bağlı olmayan** bir dekoratör uygulamak zordur - Katmanların ilk **yapılandırma kodu** oldukça çirkin görünebilir

---

Decorator (Dekoratör): Diğer Desenlerle İlişkiler

- **Adapter (Adaptör)**, mevcut bir nesneye erişmek için tamamen farklı bir arayüz sağlar. Decorator (Dekoratör) ile arayüz ya aynı kalır ya da genişletilir. Ayrıca Decorator (Dekoratör), Adapter (Adaptör) kullandığınızda mümkün olmayan özyinelemeli bileşimi destekler
- **Adapter (Adaptör)** mevcut bir nesnenin arayüzünü değiştirirken, **Decorator (Dekoratör)** arayüzünü değiştirmeden bir nesneyi geliştirir
- **Proxy (Vekil)** genellikle servis nesnesinin tam yaşam döngüsünü kendi başına yönetirken, **Decorator (Dekoratör)**'lerin bileşimi her zaman istemci tarafından kontrol edilir

---

Decorator (Dekoratör): Diğer Desenlerle İlişkiler (devam)

- **Chain of Responsibility** ve **Decorator (Dekoratör)** çok benzer sınıf yapılarına sahiptir. Her iki desen de bir dizi nesne üzerinden yürütmeyi iletmek için özyinelemeli bileşime dayanır. Ancak birçok önemli fark vardır. CoR işleyicileri birbirinden bağımsız olarak rastgele işlemler yürütebilir. Ayrıca herhangi bir noktada isteği iletmeyi durdurabilirler. Dekoratörler, temel arayüzle tutarlı tutarak nesnenin davranışını genişletir
- **Composite (Bileşik)** ve **Decorator (Dekoratör)** benzer yapı diyagramlarına sahiptir çünkü her ikisi de özyinelemeli bileşime dayanır. Bir Decorator (Dekoratör), yalnızca bir alt bileşene sahip bir Composite (Bileşik) gibidir. Decorator (Dekoratör) sarılan nesneye ek sorumluluklar eklerken, Composite (Bileşik) yalnızca alt elemanlarının sonuçlarını "toplar"
- **Strategy** bir nesnenin içini değiştirmenize olanak tanırken, **Decorator (Dekoratör)** dışını değiştirmenize olanak tanır

---

Modül E: Özet

- Decorator (Dekoratör) deseni, sarmalama yoluyla nesnelere **dinamik olarak** davranış eklenmesine olanak tanır
- Kalıtımın neden olduğu alt sınıfların **kombinatoryal patlamasını** önler

- Birden fazla dekoratör davranışları birleştirmek için **yığılabilir**
- **Tek Sorumluluk** ve **Açık/Kapalı** ilkelerini takip eder
- Davranışların **çalışma zamanında** eklenmesi gerektiğinde kalıtım yerine bileşimi tercih edin

---

Modül F: Facade (Cephe) Deseni

Modül F: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Facade (Cephe): Amaç

Facade (Cephe), bir kütüphaneye, bir çerçeveye veya başka herhangi bir karmaşık sınıf kümesine **basitleştirilmiş bir arayüz** sağlayan yapısal bir tasarım desenidir.

Kaynak: refactoring.guru[9]

Facade (Cephe): Problem

Kodunuzun gelişmiş bir kütüphane veya çerçeveye ait geniş bir nesne kümesiyle çalışmasını sağlamanız gerektiğini hayal edin. Normalde, tüm bu nesneleri **başlatmanız**, **bağımlılıkları** takip etmeniz, yöntemleri **doğru sırada** çalıştırmanız gerekir, vb.

Sonuç olarak, sınıflarınızın iş mantığı üçüncü taraf sınıfların uygulama detaylarına **sıkıca bağlı** hale gelir, bu da anlamayı ve bakımı zorlaştırır.

Facade (Cephe): Çözüm

Bir cephe, çok sayıda hareketli parça içeren karmaşık bir alt sisteme **basit bir arayüz** sağlayan bir sınıftır. Bir cephe, alt sistemle doğrudan çalışmaya kıyasla sınırlı işlevsellik sağlayabilir. Ancak yalnızca **istemcilerin gerçekten önem verdiği** özellikleri içerir.

Düzinelerce özelliğe sahip gelişmiş bir kütüphaneyle uygulamanızı entegre etmeniz gerektiğinde, ancak işlevselliğinin yalnızca küçük bir kısmına ihtiyacınız olduğunda bir cepheye sahip olmak kullanışlıdır.

---

[9] https://refactoring.guru/design-patterns/facade

Facade (Cephe): Çözüm (devam)

Örneğin, sosyal medyaya kedilerle ilgili kısa komik videolar yükleyen bir uygulama potansiyel olarak profesyonel bir video dönüştürme kütüphanesi kullanabilir. Ancak gerçekten ihtiyaç duyduğu tek şey, tek bir `encode(filename, format)` yöntemine sahip bir sınıftır.

Böyle bir sınıf oluşturup video dönüştürme kütüphanesine bağladıktan sonra, ilk **cephenize** sahip olursunuz.

Facade (Cephe): Gerçek Dünya Benzetmesi

Telefonla sipariş vermek için bir mağazayı aradığınızda, bir **operatör** mağazanın tüm hizmet ve bölümlerine cephenizdir. Operatör, sipariş sistemi, ödeme geçitleri ve çeşitli teslimat hizmetlerine basit bir sesli arayüz sağlar.

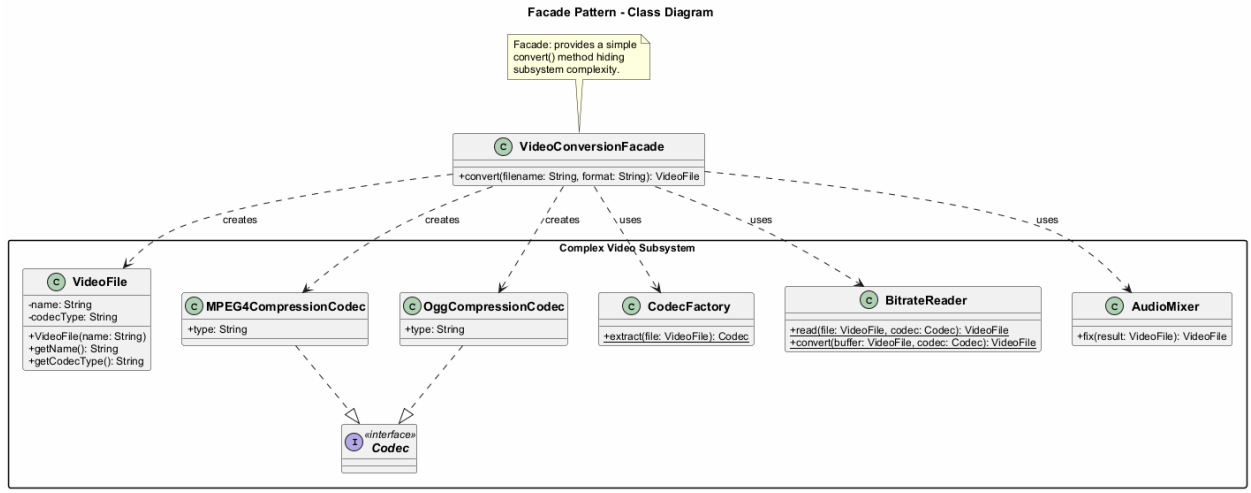
Mağazanın dahili sistemlerinde kendiniz gezinmeniz gerekmez – operatör basit bir arayüz üzerinden her şeyi halleder.

Facade (Cephe): Yapı

1. **Cephe** – Alt sistemin işlevselliğinin belirli bir bölümüne uygun erişim sağlar. İstemcinin isteğini nereye yönlendireceğini ve tüm hareketli parçaları nasıl çalıştıracağını bilir
2. **Ek Cephe** – Tek bir cepheyi ilgisiz özelliklerle kirletmeyi önlemek için oluşturulabilir, aksi takdirde onu başka bir karmaşık yapı haline getirir. Ek cepheler hem istemciler hem de diğer cepheler tarafından kullanılabilir
3. **Karmaşık Alt Sistem** – Düzinelerce çeşitli nesneden oluşur. Hepsinin anlamlı bir şey yapmasını sağlamak için alt sistemin uygulama detaylarına derinlemesine dalmanız gerekir. Alt sistem sınıfları cephenin varlığından haberdar değildir
4. **İstemci** – Alt sistem nesnelerini doğrudan çağırmak yerine cepheyi kullanır

---

Facade - Sınıf Diyagramı



Şekil 10: center

Facade (Cephe): Java Kod Örneği - Alt Sistem Sınıfları

```
// These are some of the classes of a complex third-party
// video conversion framework.
public class VideoFile {
    private String name;
    private String codecType;

    public VideoFile(String name) {
        this.name = name;
        this.codecType = name.substring(
            name.indexOf(".") + 1);
    }

    public String getName() { return name; }
    public String getCodecType() { return codecType; }
}

public interface Codec { }

public class MPEG4CompressionCodec implements Codec {
    public String type = "mp4";
}

public class OggCompressionCodec implements Codec {
    public String type = "ogg";
}
```

Facade (Cephe): Java Kod Örneği - Alt Sistem Sınıfları (devam)

```
public class CodecFactory {
    public static Codec extract(VideoFile file) {
        String type = file.getCodecType();
        if (type.equals("mp4")) {
            System.out.println("CodecFactory: extracting " +
                "MPEG4 audio...");
            return new MPEG4CompressionCodec();
        } else {
            System.out.println("CodecFactory: extracting " +
                "Ogg audio...");
            return new OggCompressionCodec();
        }
    }
}

public class BitrateReader {
    public static VideoFile read(VideoFile file, Codec codec) {
        System.out.println("BitrateReader: reading file...");
        return file;
    }

    public static VideoFile convert(VideoFile buffer,
                                    Codec codec) {
        System.out.println("BitrateReader: writing file...");
        return buffer;
    }
```

```
}

public class AudioMixer {
    public VideoFile fix(VideoFile result) {
        System.out.println("AudioMixer: fixing audio...");
        return result;
    }
}
```

Facade (Cephe): Java Kod Örneği - Cephe Sınıfı

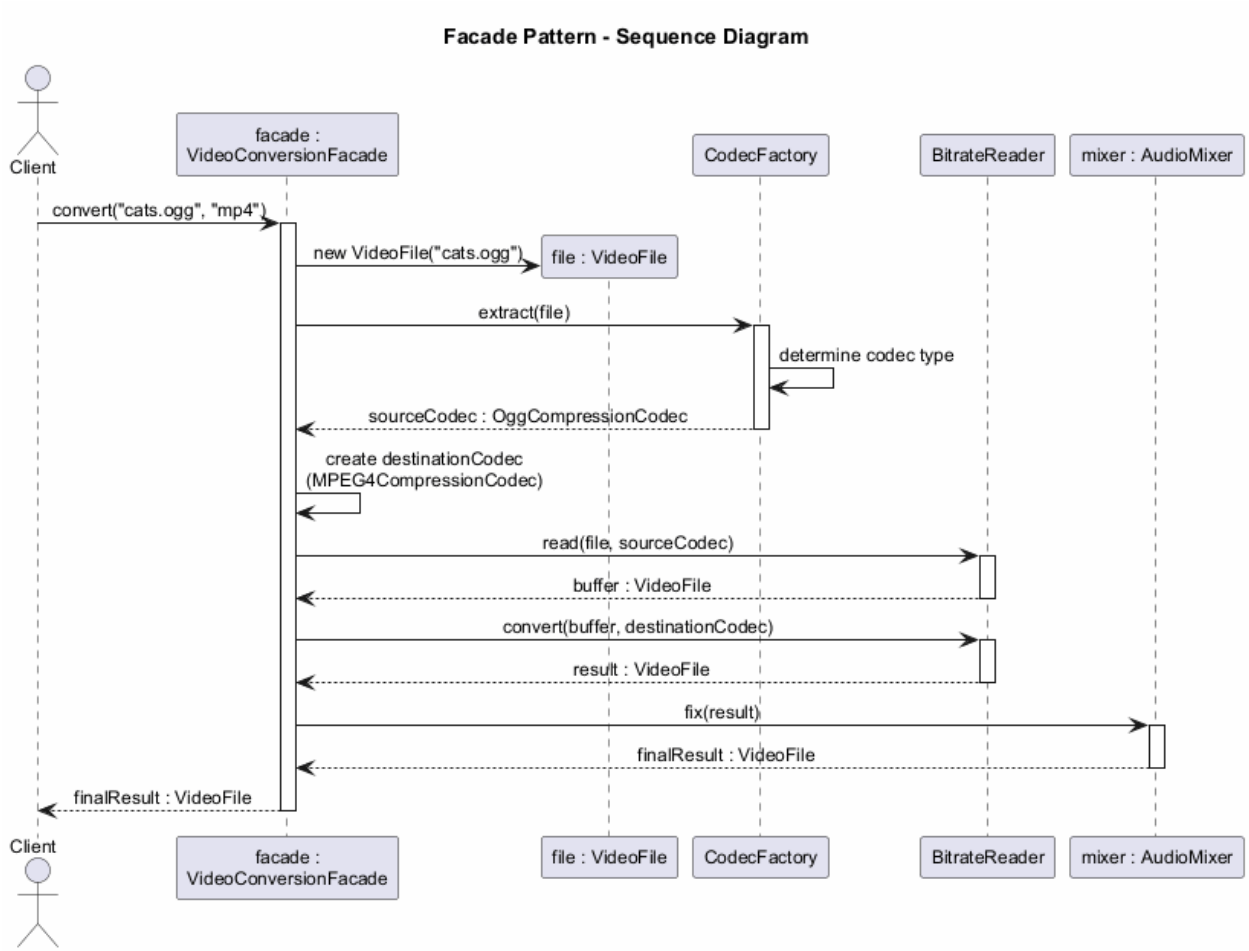
```
// The Facade class provides a simple interface to the
// complex logic of one or several subsystems. The Facade
// delegates the client requests to the appropriate objects
// within the subsystem and is also responsible for managing
// their lifecycle.
public class VideoConversionFacade {

    public VideoFile convert(String filename, String format) {
        System.out.println("VideoConversionFacade: " +
            "conversion started.");
        VideoFile file = new VideoFile(filename);
        Codec sourceCodec = CodecFactory.extract(file);

        Codec destinationCodec;
        if (format.equals("mp4")) {
            destinationCodec = new MPEG4CompressionCodec();
        } else {
            destinationCodec = new OggCompressionCodec();
        }

        VideoFile buffer = BitrateReader.read(file,
            sourceCodec);
        VideoFile result = BitrateReader.convert(buffer,
            destinationCodec);
        result = new AudioMixer().fix(result);

        System.out.println("VideoConversionFacade: " +
            "conversion completed.");
        return result;
    }
}
```

Facade (Cephe): Java Kod Örneği - İstemci

```
// The client code does not depend on any subsystem classes.
// Any changes to the subsystem's code will not affect the
// client code. You would only need to update the Facade.
public class FacadeDemo {
    public static void main(String[] args) {
        VideoConversionFacade converter =
            new VideoConversionFacade();

        VideoFile mp4Video = converter.convert(
```

```
            "funny-cats-video.ogg", "mp4");

        System.out.println("Converted: " +
            mp4Video.getName());

        // The client only interacts with the simple facade
        // interface, without knowing about dozens of
        // subsystem classes.
    }
}
```

Facade - Sıralama Diyagramı



Şekil 11: center

Facade (Cephe): Uygulanabilirlik

Facade (Cephe) desenini şu durumlarda kullanın:

- Karmaşık bir alt sisteme **sınırlı ama basit bir arayüze** sahip olmanız gerektiğinde. Genellikle alt sistemler zamanla daha karmaşık hale gelir. Tasarım desenlerini uygulamak bile genellikle daha fazla sınıf oluşturmaya yol açar. Bir alt sistem çeşitli bağlamlarda daha esnek ve yeniden kullanılabilir

hale gelebilir, ancak istemciden talep ettiği yapılandırma ve kalıp kod miktarı giderek büyür. Facade (Cephe), alt sistemin en çok kullanılan özelliklerine bir kısayol sağlamaya çalışır

---

Facade (Cephe): Uygulanabilirlik (devam)

- **Bir alt sistemi katmanlar halinde yapılandırmak** istediğinizde. Bir alt sistemin her seviyesine giriş noktaları tanımlamak için cepheler oluşturun. Birden fazla alt sistem arasındaki bağlılığı, yalnızca cepheler aracılığıyla iletişim kurmalarını gerektirerek azaltabilirsiniz. Bu yaklaşım Mediator desenine benzer

---

Facade (Cephe): Nasıl Uygulanır

1. Mevcut bir alt sistemin zaten sağladığından **daha basit bir arayüz** sağlamanın mümkün olup olmadığını kontrol edin. Bu arayüz, istemci kodunu alt sistemin birçok sınıfından bağımsız kılıyorsa doğru yoldasınız
2. Bu arayüzü yeni bir **cephe sınıfında** bildirin ve uygulayın. Cephe, istemci kodundan gelen çağrıları alt sistemin uygun nesnelerine yönlendirmelidir. Cephe, istemci kodu bunu zaten yapmıyorsa, alt sistemi başlatmaktan ve daha sonraki yaşam döngüsünü yönetmekten sorumlu olmalıdır

---

Facade (Cephe): Nasıl Uygulanır (devam)

3. Desenden tam fayda sağlamak için, tüm istemci kodunun alt sistemle **yalnızca cephe üzerinden** iletişim kurmasını sağlayın. Artık istemci kodu, alt sistem kodundaki herhangi bir değişiklikten korunur. Örneğin, bir alt sistem yeni bir sürüme yükseltildiğinde, yalnızca cephedeki kodu değiştirmeniz gerekir
4. Cephe **çok büyük** olursa, davranışının bir kısmını yeni, iyileştirilmiş bir cephe sınıfına çıkarmayı düşünün

---

Facade (Cephe): Avantajlar ve Dezavantajlar

Avantajlar: - Kodunuzu bir alt sistemin karmaşıklığından **izole** edebilirsiniz - Karmaşık bir sisteme **basit** bir arayüz sağlarsınız - İstemci ile alt sistem arasında **gevşek bağlılığı** teşvik edersiniz

Dezavantajlar: - Bir cephe, uygulamanın tüm sınıflarına bağlı bir **tanrı nesnesi** haline gelebilir

---

Facade (Cephe): Diğer Desenlerle İlişkiler

- **Facade (Cephe)** mevcut nesneler için yeni bir arayüz tanımlarken, **Adapter (Adaptör)** mevcut arayüzü kullanılabilir hale getirmeye çalışır. Adapter (Adaptör) genellikle yalnızca bir nesneyi sararken, Facade (Cephe) bütün bir nesne alt sistemiyle çalışır
- **Abstract Factory**, alt sistem nesnelerinin istemci kodundan nasıl oluşturulduğunu gizlemek istediğinizde **Facade (Cephe)**'ye alternatif olarak hizmet edebilir
- **Flyweight (Sinek Siklet)** çok sayıda küçük nesne yapmanın nasıl olduğunu gösterirken, **Facade (Cephe)** tüm bir alt sistemi temsil eden tek bir nesne yapmanın nasıl olduğunu gösterir

---

Facade (Cephe): Diğer Desenlerle İlişkiler (devam)

- **Facade (Cephe)** ve **Mediator** benzer görevlere sahiptir: birçok sıkıca bağlı sınıf arasındaki işbirliğini organize etmeye çalışırlar. Facade (Cephe), bir nesne alt sisteme basitleştirilmiş bir arayüz tanımlar,

ancak herhangi bir yeni işlevsellik tanıtmaz. Alt sistemin kendisi cepheden habersizdir. Mediator, sistemin bileşenleri arasındaki iletişimi merkezileştirir. Bileşenler yalnızca mediator nesnesini bilir
- Bir **Facade (Cephe)** sınıfı genellikle bir **Singleton**'a dönüştürülebilir çünkü çoğu durumda tek bir cephe nesnesi yeterlidir
- **Facade (Cephe)**, **Proxy (Vekil)**'e benzer; her ikisi de karmaşık bir varlığı tamponlar ve kendi başına başlatır. Facade (Cephe)'den farklı olarak, Proxy (Vekil) servis nesnesiyle aynı arayüze sahiptir, bu da onları değiştirilebilir kılar

---

Modül F: Özet

- Facade (Cephe) deseni, karmaşık bir alt sisteme **basitleştirilmiş bir arayüz** sağlar
- **Yeni işlevsellik eklemez** ancak mevcut işlevselliği erişmeyi kolaylaştırır
- İstemci kodu ile alt sistem sınıfları arasında **gevşek bağlılığı** teşvik eder
- **Karmaşık kütüphaneler** veya **eski sistemlerle** entegrasyon sırasında faydalıdır
- Cephenin bir **tanrı nesnesi** haline gelmemesine dikkat edin

---

Modül G: Flyweight (Sinek Siklet) Deseni

Modül G: Ana Hatlar

- Amaç
- Problem
- Çözüm
- İçsel ve Dışsal Durum
- Gerçek Dünya Benzetmesi
- Yapı (UML)
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Flyweight (Sinek Siklet): Amaç

Flyweight (Sinek Siklet), her nesnede tüm verileri tutmak yerine, birden fazla nesne arasında durumun ortak kısımlarını paylaşarak mevcut **RAM** miktarına daha fazla nesne sığdırmanıza olanak tanıyan yapısal bir tasarım desenidir.

- Ayrıca şöyle de bilinir: **Cache (Önbellek)**

Kaynak: refactoring.guru[10]

Flyweight (Sinek Siklet): Problem

Uzun çalışma saatlerinden sonra biraz eğlenmek için basit bir video oyunu oluşturmaya karar verirsiniz: oyuncular bir haritada dolaşır ve birbirlerine ateş eder. Özelliklerinden biri olarak gerçekçi bir **parçacık**

[10] https://refactoring.guru/design-patterns/flyweight

sistemi uygularsınız. Çok miktarda mermi, füze ve patlamalardan çıkan şarapnel parçaları haritanın her yerinde uçarak heyecan verici bir deneyim sunmalıdır.

Son commit'i gönderdikten, oyunu derledikten ve test için bir arkadaşınıza gönderdikten sonra, oyun makinenizde sorunsuz çalışmasına rağmen arkadaşınız birkaç dakika oynadıktan sonra oyun sürekli **çöker**.

Flyweight (Sinek Siklet): Problem (devam)

Hata ayıklama günlüklerinde saatlerce araştırma yaptıktan sonra, oyunun **yetersiz RAM** nedeniyle çöktüğünü keşfedersiniz. Arkadaşınızın bilgisayarının sizinkinden çok daha az güçlü olduğu ve sorunun onun makinesinde bu kadar çabuk ortaya çıkmasının nedeni budur.

Asıl sorun parçacık sistemiyle ilgiliydi. Mermi, füze veya şarapnel parçası gibi her parçacık, bol miktarda veri içeren ayrı bir nesneyle temsil ediliyordu. Bir noktada, oyuncunun ekranındaki kıyam doruğa ulaştığında, yeni oluşturulan parçacıklar artık kalan RAM'e sığmıyordu ve program çöktü.

Flyweight (Sinek Siklet): Problem (devam)

Her parçacık nesnesi şunları içerir: - **Renk** (tüm mermiler için aynı) - **Sprite/doku** (tüm mermiler için aynı) - **Koordinatlar** (her parçacık için benzersiz) - **Yön vektörü** (her parçacık için benzersiz) - **Hız** (her parçacık için benzersiz)

Renk ve sprite verileri binlerce parçacık arasında **tekrarlanır**, bu da büyük miktarda bellek israfına neden olur.

Flyweight (Sinek Siklet): Çözüm - İçsel ve Dışsal Durum

Flyweight (Sinek Siklet) deseni, **dışsal durumu** nesnenin içinde saklamayı bırakmanızı önerir. Bunun yerine, bu durumu ona bağımlı olan belirli yöntemlere iletmelisiniz. Yalnızca **içsel durum** nesnenin içinde kalır ve farklı bağlamlarda yeniden kullanmanıza olanak tanır.

İçsel Durum – Bir nesnenin sabit verileri. Diğer nesneler bunu yalnızca okuyabilir, değiştiremez. Bu veri flyweight nesnesi içinde yaşar ve oluşturulduktan sonra **değiştirilemez** kalır. - Örnekler: renk, sprite, doku

Dışsal Durum – Her nesne örneğine özgü, değişen bağlamsal veri. - Örnekler: koordinatlar, yön, hız

---

Flyweight (Sinek Siklet): Çözüm (devam)

Sonuç olarak, bu nesnelerden **daha azına** ihtiyacınız olacaktır çünkü yalnızca içsel durumda farklılık gösterirler ve bu durumun dışsal durumdan çok daha az varyasyonu vardır.

Parçacık örneğinde, binlerce tekrarlanan veri içeren parçacık nesnesi yerine, benzersiz konum ve hareket verilerini içeren birçok bağlam nesnesi tarafından referans verilen yalnızca **üç flyweight nesnesi** (mermi, füze, şarapnel) olurdu.

Flyweight (Sinek Siklet): Değişmezlik ve Fabrika

Değişmezlik Gereksinimi: - Aynı flyweight nesnesi farklı bağlamlarda kullanılabildiğinden, durumunun **değiştirilememesini** sağlamalısınız - Bir flyweight durumunu yalnızca bir kez, **yapıcı parametreleri** aracılığıyla başlatmalıdır - Diğer nesnelere herhangi bir setter veya public alan açmamalıdır

Fabrika Deseni Entegrasyonu: - Çeşitli flyweight'lere erişimi basitleştirmek için, mevcut flyweight nesnelerinin bir havuzunu yöneten bir **fabrika yöntemi** oluşturun - Fabrika, istenen flyweight'in içsel durumunu

kabul eder, bu durumla eşleşen mevcut bir flyweight arar ve bulunursa döndürür. Bulunamazsa yeni bir flyweight oluşturur ve havuza ekler

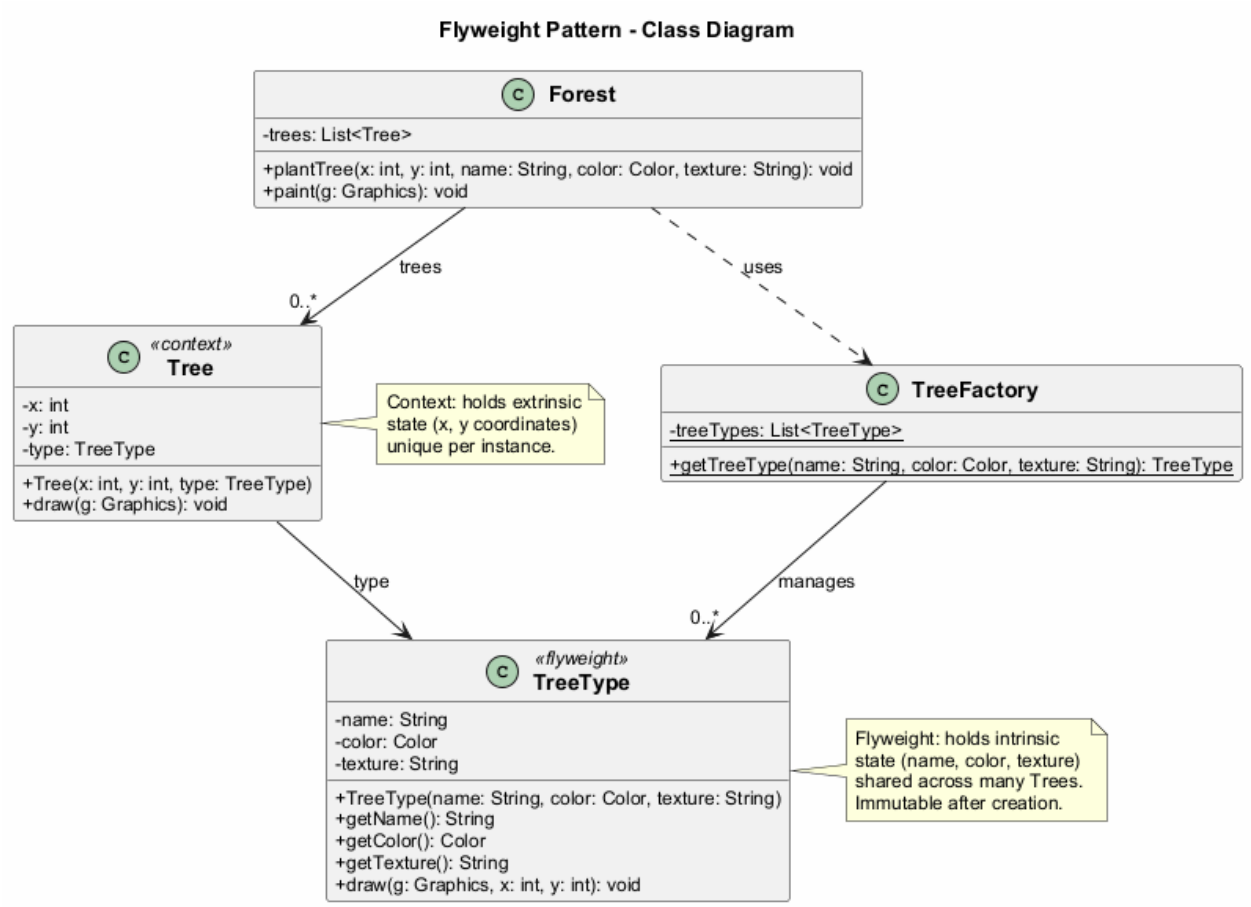
---

Flyweight (Sinek Siklet): Yapı

1. **Flyweight** – Birden fazla nesne arasında paylaşılabilen orijinal nesnenin durumunun bir bölümünü (içsel durum) içerir. Aynı flyweight nesnesi birçok farklı bağlamda kullanılabilir. Flyweight'in içinde saklanan durum içsel olarak adlandırılır. Flyweight'in yöntemlerine iletilen durum dışsal olarak adlandırılır
2. **Bağlam** – Tüm orijinal nesneler arasında benzersiz olan dışsal durumu içerir. Bir bağlam flyweight nesnelerinden biriyle eşleştirildiğinde, orijinal nesnenin tam durumunu temsil eder
3. **Flyweight Fabrikası** – Mevcut flyweight'lerin bir havuzunu yönetir. Fabrika ile istemciler flyweight'leri doğrudan oluşturmaz. Bunun yerine fabrikayı çağırır ve istenen flyweight'in içsel durumunun bitlerini iletir. Fabrika önceden oluşturulmuş flyweight'leri inceler ve mevcut olanı döndürür veya yeni bir tane oluşturur
4. **İstemci** – Flyweight'lerin dışsal durumunu (bağlam) hesaplar veya saklar. İstemcinin bakış açısından, bir flyweight, yöntemlerinin parametrelerine bazı bağlamsal veriler ileterek çalışma zamanında yapılandırılabilen bir şablon nesnedir

---

Flyweight - Sınıf Diyagramı



Şekil 12: center

Flyweight (Sinek Siklet): Java Kod Örneği - Flyweight Sınıfı

```java
import java.awt.*;

// The flyweight class contains a portion of the state of a
// tree. These fields store values that are unique for each
// particular tree type. You will not find here the tree
// coordinates. But the texture and colors shared between
// many trees are here. Since this data is usually BIG, you
// would waste a lot of memory by keeping it in each tree
// object. Instead, we extract texture, color, and other
// repeating data into a separate object (TreeType).
public class TreeType {
    private String name;
    private Color color;
    private String texture;

    public TreeType(String name, Color color, String texture) {
        this.name = name;
        this.color = color;
        this.texture = texture;
    }

    public String getName() { return name; }
    public Color getColor() { return color; }
    public String getTexture() { return texture; }

    public void draw(Graphics g, int x, int y) {
        // Draw the tree on canvas at position (x, y)
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);
    }
}
```

Flyweight (Sinek Siklet): Java Kod Örneği - Flyweight Fabrikası

```java
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

// The flyweight factory decides whether to reuse existing
// flyweight or to create a new object.
public class TreeFactory {
    private static List<TreeType> treeTypes = new ArrayList<>();

    public static TreeType getTreeType(String name,
                                       Color color,
                                       String texture) {
        for (TreeType type : treeTypes) {
            if (type.getName().equals(name) &&
                type.getColor().equals(color) &&
                type.getTexture().equals(texture)) {
```

```
                return type;
            }
        }
        TreeType type = new TreeType(name, color, texture);
        treeTypes.add(type);
        return type;
    }
}
```

Flyweight (Sinek Siklet): Java Kod Örneği - Bağlam Sınıfı

```
import java.awt.*;

// The context object contains the extrinsic part of the
// tree state. An application can create billions of these
// since they are pretty small: just two integer coordinates
// and one reference field.
public class Tree {
    private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

Flyweight (Sinek Siklet): Java Kod Örneği - İstemci (Orman)

```
import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import javax.swing.*;

public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name,
                          Color color, String texture) {
        TreeType type = TreeFactory.getTreeType(
            name, color, texture);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }

    @Override
    public void paint(Graphics g) {
        for (Tree tree : trees) {
```

```
            tree.draw(g);
        }
    }

    public static void main(String[] args) {
        Forest forest = new Forest();
        // Plant 1,000,000 trees but only a few TreeType
        // flyweight objects are created
        for (int i = 0; i < 1000000; i++) {
            forest.plantTree(
                (int) (Math.random() * 800),
                (int) (Math.random() * 600),
                "Summer Oak",
                Color.GREEN,
                "oak_texture");
        }
        forest.setSize(800, 600);
        forest.setVisible(true);
    }
}
```

Flyweight - Sıralama Diyagramı

Flyweight (Sinek Siklet): Uygulanabilirlik

Flyweight (Sinek Siklet) desenini **yalnızca** programınızın mevcut RAM'e zar zor sığan çok sayıda nesneyi desteklemesi gerektiğinde kullanın.

Deseni uygulamanın faydası, nasıl ve nerede kullanıldığına büyük ölçüde bağlıdır. En faydalı olduğu durumlar:

- Bir uygulamanın **çok sayıda** benzer nesne oluşturması gerektiğinde
- Bu, hedef cihazda mevcut tüm RAM'i tükettiğinde
- Nesneler, birden fazla nesne arasında çıkarılıp paylaşılabilen **tekrarlanan durum** içerdiğinde

---

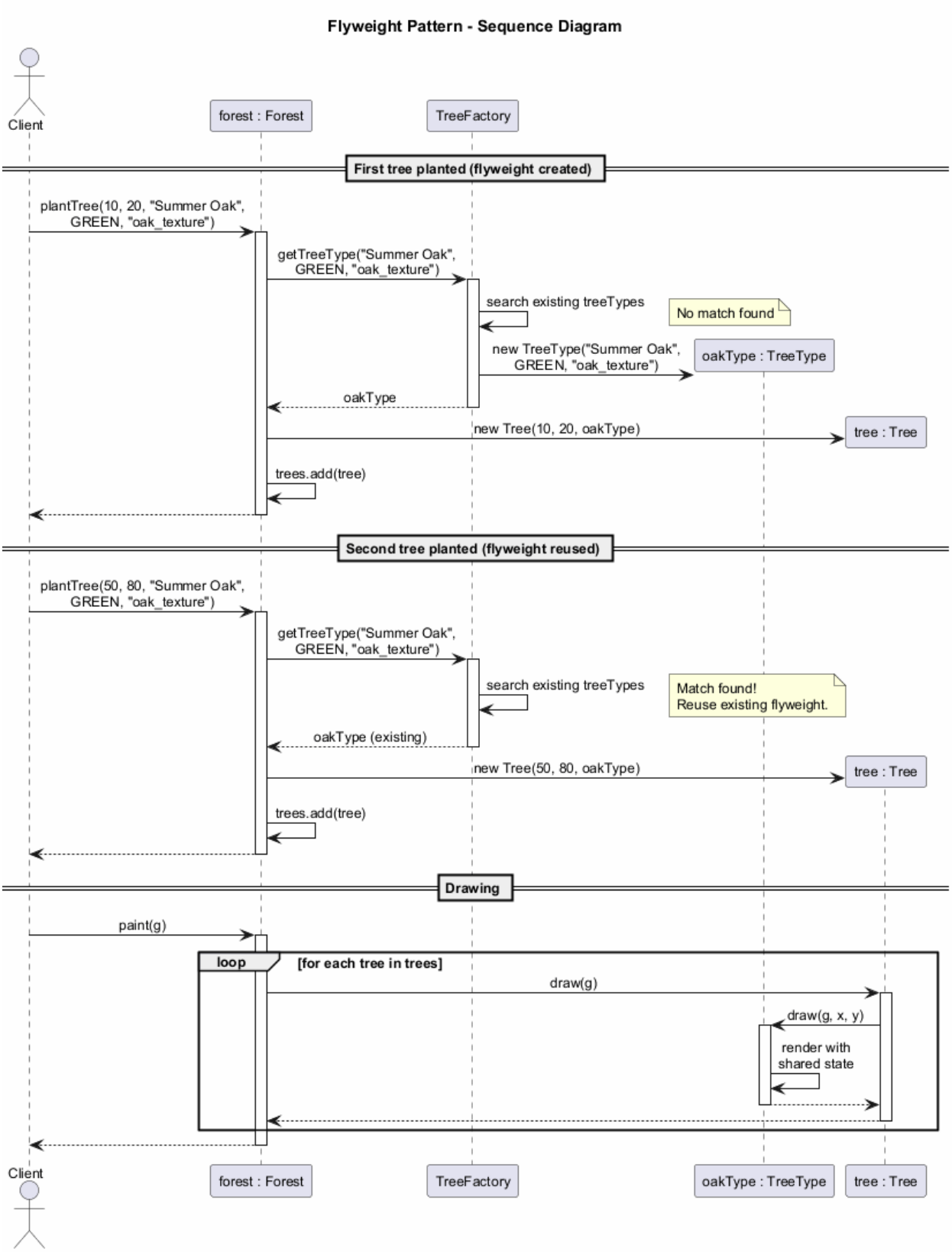
Flyweight (Sinek Siklet): Uygulanabilirlik (devam)

Önemli: Flyweight (Sinek Siklet) deseni yalnızca bir **optimizasyondur**. Uygulamadan önce, programınızın bellekte aynı anda çok sayıda benzer nesneye sahip olmayla ilgili RAM tüketim sorununa gerçekten sahip olduğundan emin olun. Bu sorunun başka anlamlı bir şekilde çözülemeyeceğinden emin olun.

Flyweight (Sinek Siklet): Nasıl Uygulanır

1. Flyweight olacak bir sınıfın alanlarını iki kısma ayırın:
   - **İçsel durum**: birçok nesne arasında tekrarlanan değişmeyen verileri içeren alanlar
   - **Dışsal durum**: her nesneye özgü bağlamsal verileri içeren alanlar
2. **İçsel durumu** temsil eden alanları sınıfta bırakın, ancak değiştirilemez olduklarından emin olun. Başlangıç değerlerini yalnızca **yapıcı** içinde almalıdırlar

---



Şekil 13: center

Flyweight (Sinek Siklet): Nasıl Uygulanır (devam)

3. **Dışsal durumun** alanlarını kullanan yöntemleri gözden geçirin. Yöntemde kullanılan her alan için yeni bir **parametre** tanıtın ve alanın yerine onu kullanın
4. İsteğe bağlı olarak, flyweight'lerin havuzunu yönetmek için bir **fabrika sınıfı** oluşturun. Yeni bir tane oluşturmadan önce mevcut bir flyweight'i kontrol etmelidir. Fabrika yerleştirildikten sonra, istemciler flyweight'leri yalnızca fabrika üzerinden talep etmeli ve içsel durumu fabrikaya iletmelidir
5. İstemci, flyweight nesnelerinin yöntemlerini çağırabilmek için **dışsal durumun** (bağlam) değerlerini saklamalı veya hesaplamalıdır. Kolaylık sağlamak için, dışsal durum flyweight referans alanıyla birlikte ayrı bir **bağlam sınıfına** taşınabilir

---

Flyweight (Sinek Siklet): Avantajlar ve Dezavantajlar

Avantajlar: - Programınızda çok sayıda benzer nesne varsa, çok fazla **RAM** tasarrufu sağlayabilirsiniz - Durum paylaşımı yoluyla bellek ayak izi azaltılır

Dezavantajlar: - Bağlam verilerinin bazılarının her biri flyweight yöntemi çağrıldığında yeniden hesaplanması gerektiğinde **RAM'i CPU döngüleriyle takas** ediyor olabilirsiniz - Kod çok **daha karmaşık** hale gelir. Yeni ekip üyeleri her zaman bir varlığın durumunun neden bu şekilde ayrıldığını merak edecektir - Paylaşılan değiştirilebilir referanslar düzgün eşitlenmezse **iş parçacığı güvenliği** sorunları ortaya çıkabilir

---

Flyweight (Sinek Siklet): Diğer Desenlerle İlişkiler

- **Composite (Bileşik)** ağacının paylaşılan yaprak düğümlerini RAM'den tasarruf etmek için **Flyweight (Sinek Siklet)** olarak uygulayabilirsiniz
- **Flyweight (Sinek Siklet)** çok sayıda küçük nesne yapmanın nasıl olduğunu gösterirken, **Facade (Cephe)** tüm bir alt sistemi temsil eden tek bir nesne yapmanın nasıl olduğunu gösterir
- **Flyweight (Sinek Siklet)**, nesnelerin tüm paylaşılan durumlarını tek bir flyweight nesnesine indirmeyi bir şekilde başarırsanız **Singleton**'a benzerdi. Ancak iki temel fark vardır:
  - Yalnızca bir Singleton örneği olmalıdır, oysa bir Flyweight sınıfı farklı içsel durumlara sahip **birden fazla örneğe** sahip olabilir
  - Singleton nesnesi **değiştirilebilir** olabilir. Flyweight nesneleri **değiştirilemezdir**

Modül G: Özet

- Flyweight (Sinek Siklet) deseni, bellek tasarrufu için birçok nesne arasında **ortak durumu paylaşır**
- **İçsel** (paylaşılan, değiştirilemez) ve **dışsal** (benzersiz, bağlamsal) durum arasında ayrım yapın
- Flyweight nesnelerinin havuzunu yönetmek için bir **fabrika** kullanın
- Bunu yalnızca birçok benzer nesneyle gerçekten bir **bellek sorununuz** olduğunda uygulayın
- Bu bir **optimizasyon desenidir** – erken kullanmayın

Modül H: Proxy (Vekil) Deseni

Modül H: Ana Hatlar

- Amaç
- Problem
- Çözüm
- Gerçek Dünya Benzetmesi

- Yapı (UML)
- Proxy Türleri
- Java Kod Örneği
- Uygulanabilirlik
- Nasıl Uygulanır
- Avantajlar ve Dezavantajlar
- Diğer Desenlerle İlişkiler
- Modül Özeti

Proxy (Vekil): Amaç

Proxy (Vekil), başka bir nesne için bir **yedek veya yer tutucu** sağlamanıza olanak tanıyan yapısal bir tasarım desenidir. Bir proxy, orijinal nesneye erişimi kontrol eder ve isteğin orijinal nesneye ulaşmadan önce veya sonra bir şeyler yapmanıza olanak tanır.

Kaynak: refactoring.guru[11]

Proxy (Vekil): Problem

Bir nesneye erişimi neden kontrol etmek isteyesiniz? İşte bir örnek: büyük miktarda sistem kaynağı tüketen devasa bir nesneniz var. Buna zaman zaman ihtiyacınız var, ancak her zaman değil.

Tembel başlatma uygulayabilirsiniz: bu nesneyi yalnızca gerçekten ihtiyaç duyulduğunda oluşturun. Nesnenin tüm istemcilerinin ertelenmiş başlatma kodunu çalıştırması gerekir. Ne yazık ki, bu muhtemelen çok fazla **kod tekrarına** neden olur.

Proxy (Vekil): Problem (devam)

İdeal bir dünyada, bu kodu doğrudan nesnemizin sınıfına koymak isterdik, ancak bu her zaman mümkün değildir. Örneğin, sınıf kapalı bir **üçüncü taraf kütüphanesinin** parçası olabilir.

Nesnenin sınıfını değiştirmeden veya başlatma mantığını tüm istemcilere tekrarlamadan nesneye erişimi kontrol etmenin bir yoluna ihtiyacımız var.

Proxy (Vekil): Çözüm

Proxy (Vekil) deseni, orijinal servis nesnesiyle **aynı arayüze** sahip yeni bir proxy sınıfı oluşturmanızı önerir. Ardından uygulamanızı, proxy nesnesini orijinal nesnenin tüm istemcilerine iletecek şekilde güncellersiniz.

İstemciden bir istek alındığında, proxy gerçek bir servis nesnesi oluşturur ve tüm işi ona **devreder**.

Proxy (Vekil): Çözüm (devam)

Peki fayda nedir? Sınıfın birincil mantığından önce veya sonra bir şey çalıştırmanız gerekiyorsa, proxy bunu **o sınıfı değiştirmeden** yapmanıza olanak tanır. Proxy, orijinal sınıfla aynı arayüzü uyguladığından, gerçek bir servis nesnesi bekleyen herhangi bir istemciye iletilebilir.

Proxy şunları halledebilir: - Gerçek nesnenin **tembel başlatılması** - Sonuçların **önbelleğe alınması** - **Erişim kontrolü** ve yetkilendirme - İsteklerin **günlüğe kaydedilmesi** - **Kaynak yönetimi** ve temizlik

---

[11] https://refactoring.guru/design-patterns/proxy

Proxy (Vekil): Gerçek Dünya Benzetmesi

Bir **kredi kartı**, bir banka hesabı için bir proxy'dir ve banka hesabı da bir deste nakit için bir proxy'dir. Her ikisi de aynı arayüzü uygular: ödeme yapmak için kullanılabilirler.

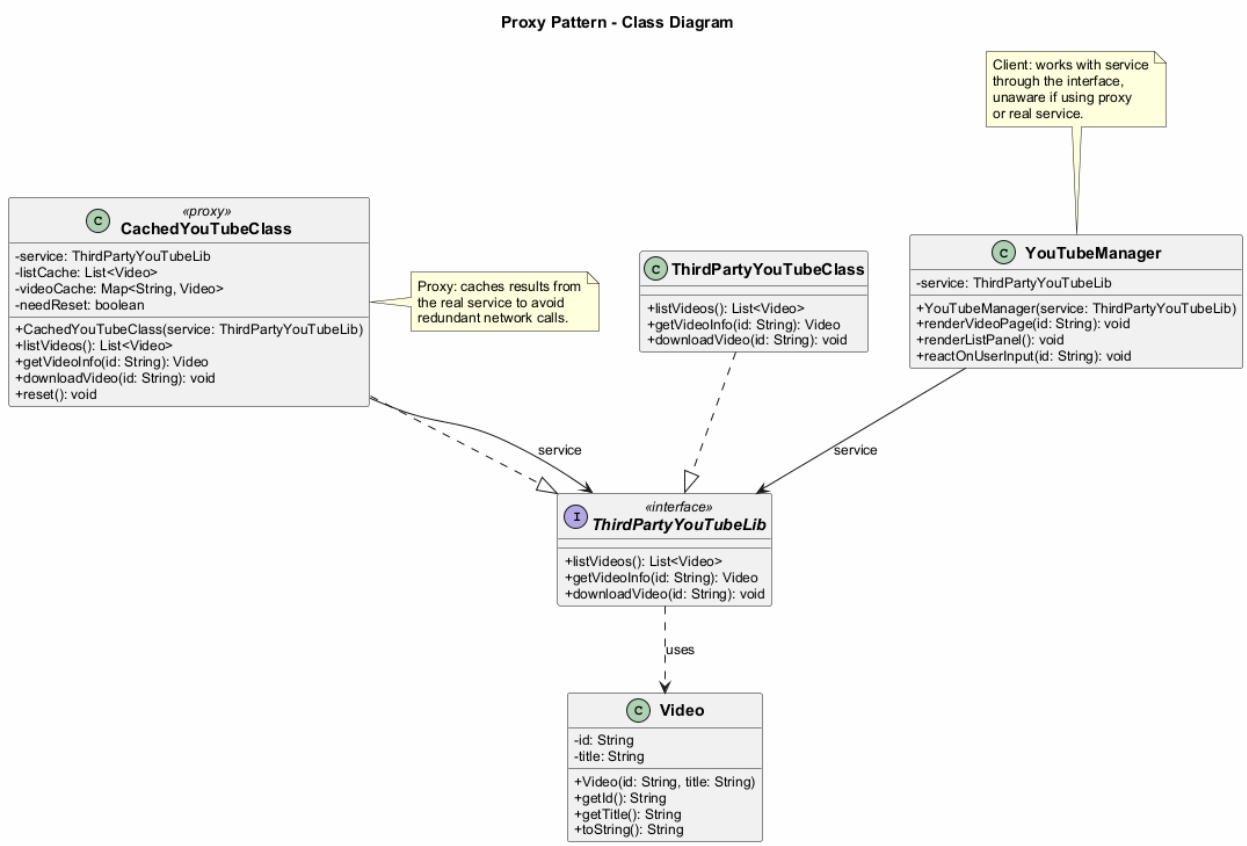
Bir tüketici, etrafta yığınla nakit taşımasına gerek olmadığı için kendini harika hisseder. Bir mağaza sahibi de mutludur çünkü bir işlemden elde edilen gelir, depozitosunu kaybetme veya bankaya giderken soyulma riski olmadan mağazanın banka hesabına elektronik olarak eklenir.

Proxy (Vekil): Yapı

1. **Servis Arayüzü** – Servisin arayüzünü bildirir. Proxy'nin kendini bir servis nesnesi olarak gizleyebilmesi için bu arayüzü takip etmesi gerekir
2. **Servis** – Bazı yararlı iş mantığı sağlayan bir sınıf
3. **Proxy** – Proxy sınıfı, bir servis nesnesine işaret eden bir referans alanına sahiptir. Proxy, işlemini tamamladıktan sonra (örn., tembel başlatma, günlüğe kaydetme, erişim kontrolü, önbelleğe alma, vb.), isteği servis nesnesine iletir. Genellikle proxy'ler servis nesnelerinin tam yaşam döngüsünü yönetir
4. **İstemci** – Hem servisler hem de proxy'lerle aynı arayüz üzerinden çalışmalıdır. Bu şekilde bir proxy'yi bir servis nesnesi bekleyen herhangi bir koda iletebilirsiniz

---

Proxy - Sınıf Diyagramı



Şekil 14: center

Proxy (Vekil): Proxy Türleri

- **Sanal Proxy (Tembel Başlatma)** – Ağır bir servis nesnesinin oluşturulmasını gerçekten ihtiyaç duyulana kadar geciktirir
- **Koruma Proxy'si (Erişim Kontrolü)** – İstekleri yalnızca istemcinin kimlik bilgileri belirli kriterleri karşıladığında servis nesnesine iletir
- **Uzak Proxy** – Proxy, istemci isteğini ağ üzerinden iletir ve ağla çalışmanın tüm detaylarını halleder
- **Günlükleme Proxy'si** – Proxy, her isteği servise iletmeden önce günlüğe kaydedebilir
- **Önbellekleme Proxy'si** – Proxy, istemci isteklerinin sonuçlarını önbelleğe alır ve bu önbelleğin yaşam döngüsünü yönetir, özellikle sonuçlar oldukça büyükse
- **Akıllı Referans** – Proxy, istemcinin servis nesnesine referans alıp almadığını izleyebilir ve hiçbir istemci kullanmadığında servisi sonlandırabilir

---

Proxy (Vekil): Java Kod Örneği - Servis Arayüzü

```java
import java.util.List;

// The interface of a remote service.
public interface ThirdPartyYouTubeLib {
    List<Video> listVideos();
    Video getVideoInfo(String id);
    void downloadVideo(String id);
}
```

Proxy (Vekil): Java Kod Örneği - Video Sınıfı

```java
public class Video {
    private String id;
    private String title;

    public Video(String id, String title) {
        this.id = id;
        this.title = title;
    }

    public String getId() { return id; }
    public String getTitle() { return title; }

    @Override
    public String toString() {
        return "Video{id='" + id + "', title='" +
            title + "'}";
    }
}
```

Proxy (Vekil): Java Kod Örneği - Servis

```java
import java.util.*;

// The concrete implementation of a service connector.
// Methods of this class can request information from
// YouTube. The speed of the request depends on a user's
// internet connection as well as YouTube's. The
```

```java
// application will slow down if a lot of requests are
// fired at the same time, even if they all request the
// same information.
public class ThirdPartyYouTubeClass
        implements ThirdPartyYouTubeLib {

    @Override
    public List<Video> listVideos() {
        // Send an API request to YouTube.
        System.out.println("Downloading video list from " +
            "YouTube API...");
        return new ArrayList<>(); // Simplified
    }

    @Override
    public Video getVideoInfo(String id) {
        // Get metadata about some video.
        System.out.println("Downloading info for video: " +
            id);
        return new Video(id, "Sample Video");
    }

    @Override
    public void downloadVideo(String id) {
        // Download a video file from YouTube.
        System.out.println("Downloading video file: " + id);
    }
}
```

Proxy (Vekil): Java Kod Örneği - Proxy (Önbellekleme)

```java
import java.util.*;

// To save some bandwidth, we can cache request results
// and keep them for some time. But it may be impossible
// to put such code directly into the service class. For
// example, it could have been provided as part of a
// third party library and/or defined as final.
public class CachedYouTubeClass
        implements ThirdPartyYouTubeLib {

    private ThirdPartyYouTubeLib service;
    private List<Video> listCache = null;
    private Map<String, Video> videoCache = new HashMap<>();
    private boolean needReset = false;

    public CachedYouTubeClass(ThirdPartyYouTubeLib service) {
        this.service = service;
    }

    @Override
    public List<Video> listVideos() {
        if (listCache == null || needReset) {
            listCache = service.listVideos();
            needReset = false;
        }
```

```
        return listCache;
    }

    @Override
    public Video getVideoInfo(String id) {
        if (!videoCache.containsKey(id) || needReset) {
            videoCache.put(id, service.getVideoInfo(id));
        }
        return videoCache.get(id);
    }

    @Override
    public void downloadVideo(String id) {
        service.downloadVideo(id);
    }

    public void reset() {
        needReset = true;
    }
}
```

Proxy (Vekil): Java Kod Örneği - Yönetici Sınıfı

```
import java.util.List;

// The GUI class, which used to work directly with a
// service object, stays unchanged as long as it works
// with the service object through an interface. We can
// safely pass a proxy object instead of a real service
// object since they both implement the same interface.
public class YouTubeManager {
    private ThirdPartyYouTubeLib service;

    public YouTubeManager(ThirdPartyYouTubeLib service) {
        this.service = service;
    }

    public void renderVideoPage(String id) {
        Video info = service.getVideoInfo(id);
        System.out.println("Rendering video page: " +
            info.getTitle());
    }

    public void renderListPanel() {
        List<Video> list = service.listVideos();
        System.out.println("Rendering video list: " +
            list.size() + " videos");
    }

    public void reactOnUserInput(String id) {
        renderListPanel();
        renderVideoPage(id);
    }
}
```

Proxy (Vekil): Java Kod Örneği - İstemci

```java
public class ProxyDemo {
    public static void main(String[] args) {
        // Create the real service
        ThirdPartyYouTubeLib youtubeService =
            new ThirdPartyYouTubeClass();

        // Create the caching proxy
        ThirdPartyYouTubeLib youtubeProxy =
            new CachedYouTubeClass(youtubeService);

        // The manager works through the interface,
        // so it does not know if it is using the real
        // service or the proxy
        YouTubeManager manager =
            new YouTubeManager(youtubeProxy);

        // First call - fetches from the real service
        manager.reactOnUserInput("video123");

        // Second call - uses cached data
        manager.reactOnUserInput("video123");
    }
}
```

Proxy - Sıralama Diyagramı

Proxy (Vekil): Uygulanabilirlik

Proxy (Vekil) desenini kullanmanın düzinelerce yolu vardır. İşte en yaygın olanları:

- **Tembel başlatma (sanal proxy)**: Her zaman açık olan ve sistem kaynaklarını boşa harcayan ağır bir servis nesneniz olduğunda, yalnızca zaman zaman ihtiyaç duyduğunuzda. Uygulama başlatıldığında nesneyi oluşturmak yerine, nesnenin başlatılmasını gerçekten ihtiyaç duyulduğu zamana erteleyebilirsiniz

---

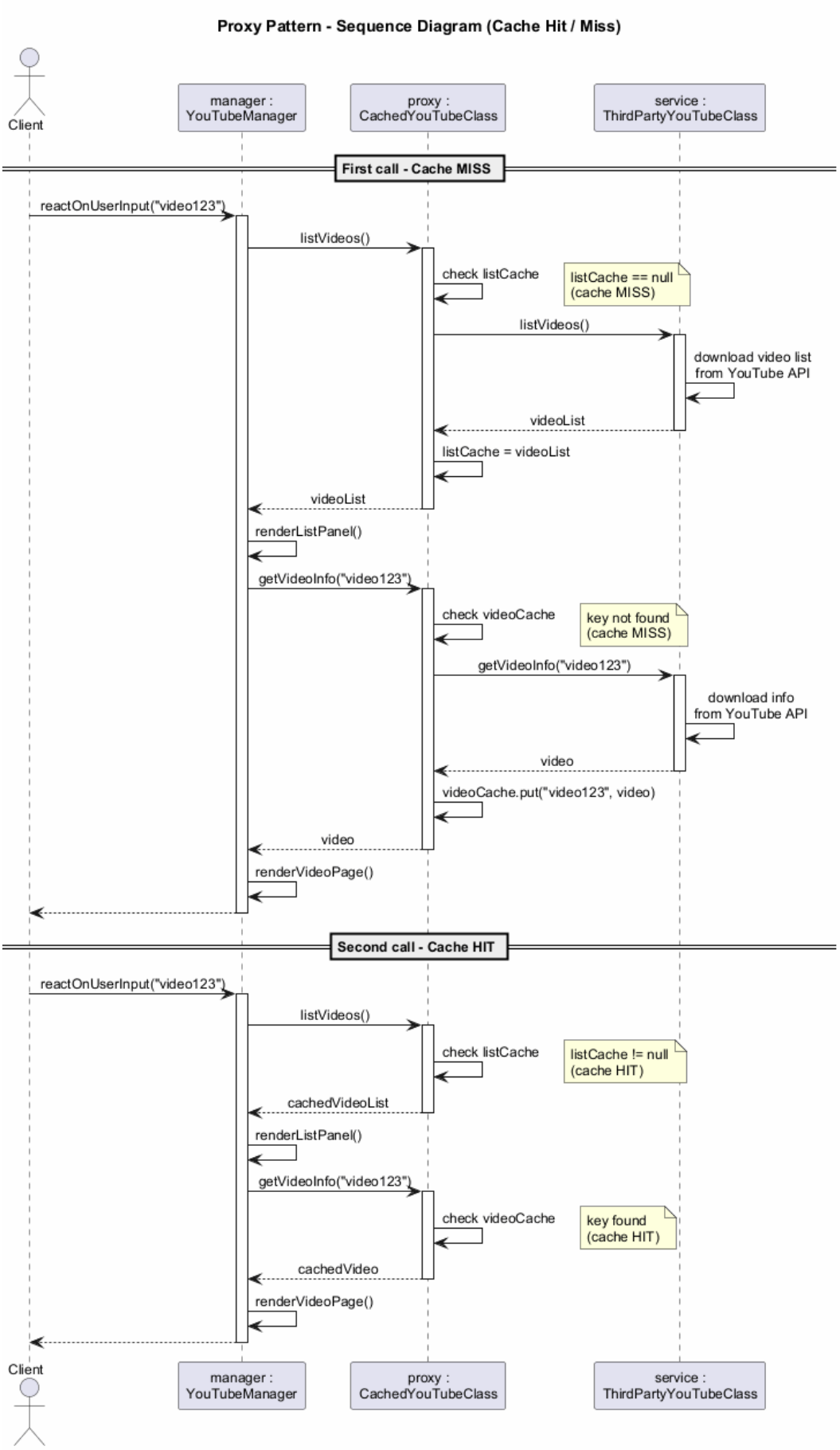
Proxy (Vekil): Uygulanabilirlik (devam)

- **Erişim kontrolü (koruma proxy'si)**: Servis nesnesini yalnızca belirli istemcilerin kullanabilmesini istediğinizde; örneğin, nesneleriniz bir işletim sisteminin kritik parçaları olduğunda ve istemciler çeşitli başlatılan uygulamalar (kötü amaçlı olanlar dahil) olduğunda
- **Uzak servisin yerel yürütülmesi (uzak proxy)**: Servis nesnesi uzak bir sunucuda bulunduğunda. Bu durumda proxy, istemci isteğini ağ üzerinden ileterek ağ ile çalışmanın tüm karmaşık detaylarını halleder

---

Proxy (Vekil): Uygulanabilirlik (devam)

- **İstekleri günlüğe kaydetme (günlükleme proxy'si)**: Servis nesnesine yapılan isteklerin geçmişini tutmak istediğinizde. Proxy, her isteği servise iletmeden önce günlüğe kaydedebilir



Şekil 15: center

- **İstek sonuçlarını önbelleğe alma (önbellekleme proxy'si)**: İstemci isteklerinin sonuçlarını önbelleğe almanız ve bu önbelleğin yaşam döngüsünü yönetmeniz gerektiğinde, özellikle sonuçlar oldukça büyükse. Proxy, her zaman aynı sonuçları veren tekrarlayan istekler için önbelleğe alma uygulayabilir
- **Akıllı referans**: Onu kullanan hiçbir istemci kalmadığında ağır bir nesneyi sonlandırabilmeniz gerektiğinde. Proxy, servis nesnesine veya sonuçlarına referans alan istemcileri takip edebilir. Zaman zaman proxy, istemcileri kontrol edebilir ve hâlâ aktif olup olmadıklarını kontrol edebilir. İstemci listesi boşalırsa, proxy servis nesnesini sonlandırabilir ve altta yatan sistem kaynaklarını serbest bırakabilir

---

Proxy (Vekil): Nasıl Uygulanır

1. Önceden var olan bir **servis arayüzü** yoksa, proxy ve servis nesnelerini değiştirilebilir kılmak için bir tane oluşturun. Arayüzü servis sınıfından çıkarmak her zaman mümkün değildir çünkü servisin tüm istemcilerini bu arayüzü kullanacak şekilde değiştirmeniz gerekir. Bir alternatif, proxy'yi servis sınıfının bir **alt sınıfı** yapmaktır
2. **Proxy sınıfını** oluşturun. Servise referans saklamak için bir alanı olmalıdır. Genellikle proxy'ler servislerinin tüm yaşam döngüsünü oluşturur ve yönetir. Nadir durumlarda, bir servis istemci tarafından yapıcı aracılığıyla proxy'ye iletilir

---

Proxy (Vekil): Nasıl Uygulanır (devam)

3. **Proxy yöntemlerini** amaçlarına göre uygulayın. Çoğu durumda, bazı işler yaptıktan sonra proxy işi servis nesnesine devretmelidir
4. İstemcinin bir proxy mi yoksa gerçek bir servis mi aldığına karar veren bir **oluşturma yöntemi** tanıtmayı düşünün. Bu, proxy sınıfındaki basit bir statik yöntem veya tam teşekküllü bir fabrika yöntemi olabilir
5. Servis nesnesi için **tembel başlatma** uygulamayı düşünün

---

Proxy (Vekil): Avantajlar ve Dezavantajlar

Avantajlar: - Servis nesnesini **istemciler bilmeden** kontrol edebilirsiniz - İstemciler umursamadığında servis nesnesinin **yaşam döngüsünü** yönetebilirsiniz - Proxy, servis nesnesi **hazır olmasa** veya kullanılamasa bile çalışır - **Açık/Kapalı İlkesi**: Servisi veya istemcileri değiştirmeden yeni proxy'ler tanıtabilirsiniz

Dezavantajlar: - Çok sayıda yeni sınıf tanıtmanız gerektiğinden kod **daha karmaşık** hale gelebilir - Servisten gelen yanıt **gecikebilir**

Proxy (Vekil): Diğer Desenlerle İlişkiler

- **Adapter (Adaptör)** sarılan nesneye farklı bir arayüz sağlarken, **Proxy (Vekil)** aynı arayüzü sağlar ve **Decorator (Dekoratör)** geliştirilmiş bir arayüz sağlar
- **Facade (Cephe)**, **Proxy (Vekil)**'e benzer; her ikisi de karmaşık bir varlığı tamponlar ve kendi başına başlatır. Facade (Cephe)'den farklı olarak, Proxy (Vekil) servis nesnesiyle aynı arayüze sahiptir, bu da onları değiştirilebilir kılar
- **Decorator (Dekoratör)** ve **Proxy (Vekil)** benzer yapılara sahiptir, ancak çok farklı amaçları vardır. Her iki desen de bileşim ilkesi üzerine kuruludur; bir nesnenin işin bir kısmını başka bir nesneye devretmesi beklenir. Fark şudur: bir Proxy (Vekil) genellikle servis nesnesinin yaşam döngüsünü kendi başına yönetirken, Decorator (Dekoratör)'lerin bileşimi her zaman istemci tarafından kontrol edilir

---

Modül H: Özet

- Proxy (Vekil) deseni, başka bir nesne için bir **yedek veya yer tutucu** sağlar
- Orijinal servisi **değiştirmeden** erişimi kontrol eder ve davranış ekler
- Birçok türü vardır: **sanal, koruma, uzak, günlükleme, önbellekleme, akıllı referans**
- Mevcut kodu değiştirmeden yeni proxy'lere izin vererek **Açık/Kapalı İlkesini** takip eder
- Yapı olarak Decorator (Dekoratör)'e benzer ancak **amaç ve yaşam döngüsü yönetiminde** farklıdır

---

Özet: Yapısal Tasarım Desenleri Karşılaştırması

Desen	Temel Fikir	Tipik Kullanım
Adapter (Adaptör)	Arayüz dönüştürme	Eski/üçüncü taraf entegrasyonu
Bridge (Köprü)	Soyutlama/uygulama ayırma	Çok boyutlu hiyerarşiler
Composite (Bileşik)	Ağaç yapısı	Parça-bütün hiyerarşileri
Decorator (Dekoratör)	Davranış dinamik ekleme	Katmanlı davranışlar
Facade (Cephe)	Arayüz basitleştirme	Karmaşık alt sistem erişimi
Flyweight (Sinek Siklet)	Durum paylaşımı	Bellek optimizasyonu
Proxy (Vekil)	Erişim kontrolü	Tembel başlatma, önbelleğe alma, güvenlik

---

Yapısal Desenler: Temel İlişkiler

- **Adapter (Adaptör)** ve **Bridge (Köprü)**: Adapter (Adaptör) sonradan uyarlar, Bridge (Köprü) baştan tasarlar
- **Composite (Bileşik)** ve **Decorator (Dekoratör)**: Her ikisi de özyinelemeli bileşim kullanır, ancak farklı amaçlar için
- **Decorator (Dekoratör)** ve **Proxy (Vekil)**: Benzer yapılar, farklı yaşam döngüsü yönetimi
- **Facade (Cephe)** ve **Adapter (Adaptör)**: Facade (Cephe) bir alt sistemi basitleştirir, Adapter (Adaptör) tek bir nesneyi sarar
- **Flyweight (Sinek Siklet)** ve **Singleton**: Flyweight (Sinek Siklet) birden fazla paylaşılan örneğe sahiptir, Singleton bir değiştirilebilir örneğe sahiptir
- **Facade (Cephe)** ve **Mediator**: Her ikisi de işbirliğini organize eder, ancak Facade (Cephe) işlevsellik eklemez

---

Kaynaklar

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Freeman, E., & Robson, E. (2020). *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software* (2nd ed.). O'Reilly Media.
- Refactoring.Guru. *Structural Design Patterns*. https://refactoring.guru/design-patterns/structural-patterns
- Refactoring.Guru. *Adapter*. https://refactoring.guru/design-patterns/adapter
- Refactoring.Guru. *Bridge*. https://refactoring.guru/design-patterns/bridge
- Refactoring.Guru. *Composite*. https://refactoring.guru/design-patterns/composite

---

Kaynaklar (devam)

- Refactoring.Guru. *Decorator*. https://refactoring.guru/design-patterns/decorator
- Refactoring.Guru. *Facade*. https://refactoring.guru/design-patterns/facade

- Refactoring.Guru. *Flyweight*. https://refactoring.guru/design-patterns/flyweight
- Refactoring.Guru. *Proxy*. https://refactoring.guru/design-patterns/proxy
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.

Teşekkürler

Sorular?

CEN206 Nesne Yönelimli Programlama Hafta-10: Yapısal Tasarım Desenleri Bahar Dönemi, 2025-2026

Öğr. Gör. Dr. Uğur CORUH

$End - Of - Week - 10 - Module$