

Yapısal Tasarım Kalıpları

Burak Atalay

Bilgisayar Mühendisliği

Yıldız Teknik Üniversitesi

İstanbul, Türkiye

burak.atalay@example.com

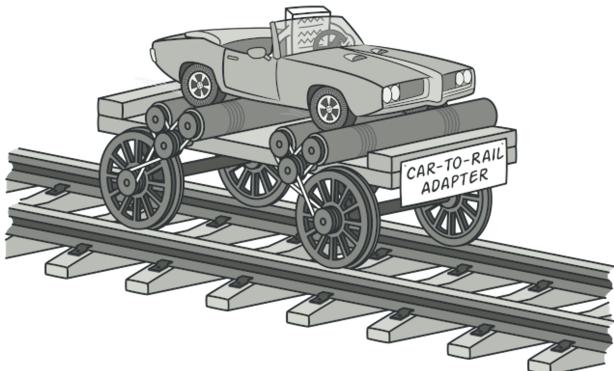
Abstract—Tasarım kalıpları, yazılım mühendisliğinde tekrar eden sorunlara çözüm sunan önemli araçlardır. Bu makale, nesnelerin ve sınıfların daha büyük, verimli ve esnek yapımlara nasıl organize edileceğini açıklayan yapısal tasarım kalıplarını ele almaktadır. Her bir kalıp, örneklerle birlikte incelenerek yazılım tasarımında uygulamaları ve etkinlikleri gösterilmektedir.

Index Terms—Yapısal Tasarım Kalıpları, Adapter, Bridge, Decorator, Composite, Flyweight, Proxy, Facade.

I. GİRİŞ

Tasarım kalıpları, yazılım tasarımında sık karşılaşılan problemlere standartlaştırılmış çözümler sunar. Yapısal kalıplar, bu kalıpların bir composition'dan yararlanılarak nesnelerin ve sınıfların nasıl daha büyük yapımlara dönüştürüleceğini ele alır ve bu yapımları esnek ve verimli tutmayı hedefler. Bu makale, Adapter, Bridge, Decorator, Composite, Flyweight, Proxy ve Facade gibi yedi ana yapısal kalıbı kapsamaktadır ve bunların kavramları ile pratik uygulamalarını detailandırmaktadır.

II. ADAPTER KALIBI



Şekil 1.

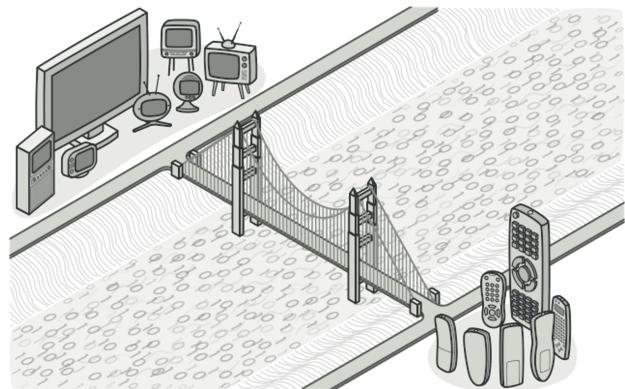
Adapter kalıbı, uyumsuz arayzlere sahip nesnelerin birlikte çalışmasını sağlar [2]. Bu kalıp, mevcut bir arayüzü başka bir arayüze dönüştürerek nesneler arasındaki iletişimini mümkün kılar. Bu kalıbın yazılım geliştirme süreçlerinde entegrasyon problemlerini çözmek için sıkça kullanıldığı gözlemlenmiştir.

Örneğin bir e-ticaret sisteminde farklı bir implementasyona sahip yeni bir ödeme yöntemini mevcut interface'e entegre etmek için Şekil 8'deki gibi bir adapter kullanılarak uyumluluk sağlanabilir. Bu örnekte PaymentGateway interface'sini implemente eden GateWayA gibi ödeme yöntemlerimiz

varken SOLID prensiplerine uygun şekilde polymorphism'den yararlanılarak istenilen yöntem interface referansıyla kullanılabilir.

Şimdi farklı bir implementasyona sahip olan ödeme yöntemi GatewayB'nin bize hazır olarak geldiğini düşünelim. PaymentGateway interface'ine uyumlu hale getirmek için uzun bir mesai harcamak gerekiyor. Bunu yapmak yerine adapter görevi görecek PaymentGateway'i implemente eden bir sınıf yaratırsak ve composition'dan yararlanarak GatewayB'nin referansını bu adapter sınıfında tutarsak override edilmiş metod içerisinde GatewayB'nin metodunu çağırabiliriz ve bu adapter sınıfını yeni ödeme yöntemimiz gibi kullanabiliriz.

III. BRIDGE KALIBI



Şekil 2.

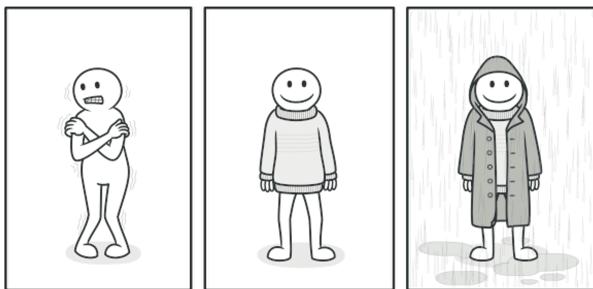
Bridge kalıbı, soyutlama ile uygulamayı birbirinden ayırarak bağımsız şekilde gelişmelerine olanak tanır [1][s. 151]. Bu yapı, kodun daha esnek ve genişletilebilir olmasını sağlar.

Örneğin, Şekil 9 ve Şekil 10'deki gibi farklı hayvanların farklı nefes alma yöntemleri olduğu bir sistemde *Animal* sınıfı (Abstraction), bir *BreathingStyle* (Implementation) referansı tutarak alt sınıfların belirli davranışları tanımlamasını sağlar. Bu, yazılımın yeniden kullanımını artıran ve bakımını kolaylaştıran bir yaklaşımındır.

Bu desenin sağladığı avantajları şöyle düşünebiliriz: Diyalim ki bu kalıbı kullanmayarak implementation tarafını (*BreathingStyle*) doğrudan abstract bir metod olarak tanımladık. *Animal* sınıfından türeyen her bir alt sınıf için parent'taki abstract metodu override etmek zorunda kalirdık. Her çocuk bu metodu farklı override ediyorsa sorun yok

ancak eğer ortak kullanım sayısı fazlaysa bridge kalıbından yararlanmak mantıklı hale geliyor. Çünkü aynı şekilde ciğerle nefes alan memeli ve kuş sınıfları *LungBreathing* sınıfının nesnesini *Animal* sınıfındaki *BreathingStyle* referansına atayarak bir kere override edilmiş metodu kullanabilirler. Yani aynı şekilde kullanılacak bir metodu bir kez override ederek kod tekrarını önlemiş olduk.

IV. DECORATOR KALIBI

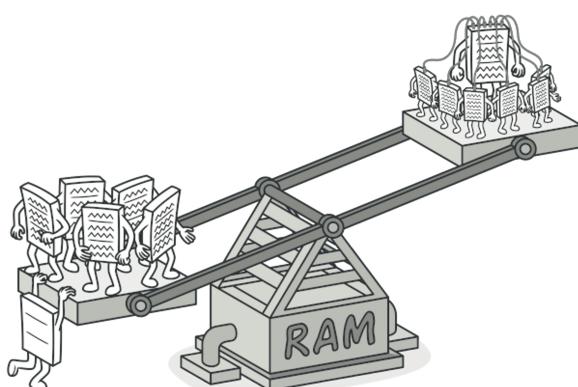


Şekil 3.

Decorator kalabı, nesnelerin davranışlarını değiştirmeden veya mevcut sınıfı değiştirmeden dinamik olarak eklemeler yapar veya işlevselliklerini artırır [4]. Bu kalıp, bir nesneye nesnede değişiklik yapmadan yeni özellikler eklemek için çok uygundur. Bu kalıp sayesinde karmaşık kalıtım hiyerarşileri önlenir.

Örneğin Şekil 11'deki kodörneğinde sıradan bir araba olan *BasicCar* sınıfı var. Bu sınıfı değişiklik yapmadan sıradan bir arabanın özelliklerine de sahip bir spor araba *SportCar* sınıfı yaratmak istiyoruz. Bunu yine composition'dan yararlanarak decorator görevi gören bir sınıf implemente ederek yapabiliriz. Bu sınıfı *BasicCar* sınıfının referansını tutacağız. *SportCar* sınıfı da bu decorator sınıfından türeyecektir. Bu sayede constructor'ında üst sınıfı *BasicCar* nesnesi sağlayarak sıradan arabanın özelliklerini parent'ı üzerinden kullanabilir.

V. FLYWEIGHT KALIBI



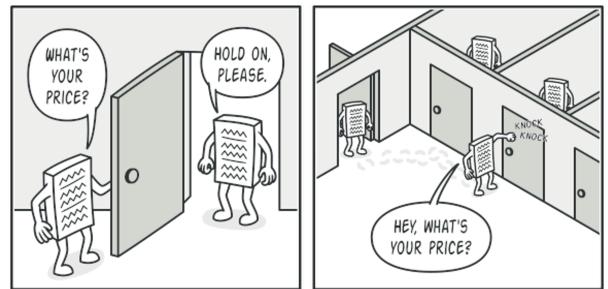
Şekil 4.

Flyweight kalabı, nesneler arasındaki ortak verileri paylaşarak bellek kullanımını minimize eder [1][s. 195].

Özellikle büyük ölçekli sistemlerde bellek tasarrufu sağlamak için bu kalıbın kullanımı yaygındır. Ortak veri için her seferinde yeni nesne yaratmak yerine nesnelere bir kere yaratılmış bu ortak verinin referansı verilir.

Örneğin Şekil 12'deki gibi kullanıcı oturumlarını verimli bir şekilde yönetmek için ortak oturum verilerinin yeniden kullanılması, bu kalıbin pratik bir örneğidir. Örnekte her session için *UserSession* nesnesi oluşturulan *UserSessionFactory* sınıfında *UserSession* nesnelerinde ortak bulunan nesne bir kez yaratılıp oluşturulacak nesnelere bu yaratılan nesnenin referansı veriliyor. Bu sayede büyük bir bellek tasarrufu sağlanıyor.

VI. COMPOSITE KALIBI



Şekil 5.

Composite kalabı, nesneleri bir ağaç yapısında düzenleyerek bütün-parça hiyerarşilerini temsil eder [3]. Bu kalıp, benzer nesne gruplarını tek bir yerden yönetebilmeyi sağlar.

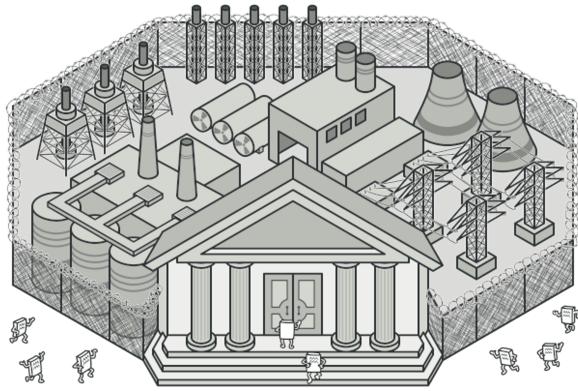
- **Component:** Composition'daki tüm nesneler için base interface.
- **Leaf:** Component'i implemente eder. Implemente eden diğer nesnelerin referanslarını içermez.
- **Composite:** Leaf nesnelerin referanslarını tutar. Component'i implemente eder ve leaf'lerle ilgili operasyonları barındırır.
- **Client:** Composite nesnesini kullanarak leaf nesnelerle ilgili operasyonlarını gerçekleştirir.

Örneğin Şekil 13 ve Şekil 14'deki gibi bir bankacılıkörneğinde *Account* sınıfı Component olarak, bireysel hesaplar Leaf olarak, *Account Portfolio* sınıfı Composite olarak kullanılıp kullanıcı hesapları yönetilebilir. Kullanıcı, Composite görevi gören nesne üzerinden tüm hesaplarını rahatlıkla yönetebilir. Mesela kodörneğinde tüm hesaplarındaki total bakiyesini composite nesnesi üzerinden hesaplayabiliyor.

VII. FACADE KALIBI

Facade kalabı, karmaşık bir alt sisteme basitleştirilmiş bir arayüz sağlar. Bu kalıp aslında aşağıdaki resimde de görüleceği gibi kullanıcı için karmaşık sistemin bir ön kapısı işlevini görür. Kullanıcı bu kapıdan ne istediğini söyler ve ürünü bu kapidan alır [1][s. 201].

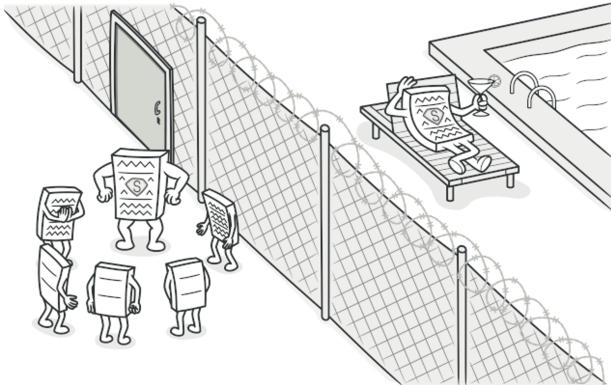
Örneğin Şekil 15 ve Şekil 16'deki gibi bir çevrimiçi alışveriş sistemi; stok takibi (*InventorySystem*), ödeme (*PaymentSystem*) ve kargo takibi (*ShippingSystem*) gibi çeşitli bileşenleri



Şekil 6.

birleştirerek tek bir arayüz (*OrderFacade*) üzerinden kulancıya sunabilir. Bu sayede kullanıcı sadece bu arayüz üzerindeki bir metod üzerinden siparişini verir ve arkada dönen karmaşıklıktan soyutlanır.

VIII. PROXY KALIBI



Şekil 7.

Proxy kalibi, bir nesneye erişimi kontrol ederek bir dolaylılık seviyesi ekler [2]. Kaynak tasarrufu veya güvenlik amacıyla erişim kontrolü sağlamak için kullanılabilir. Mesela Spring Boot'daki Lazy anotasyonu da Proxy kalibi örneğidir Çünkü backend sisteminin ayağa kalkarken yavaşlamasına neden olan veritabanındaki bazı nesnelerin sadece ihtiyaç olduğu anda veritabanından sisteme çekilmesini sağlar.

Örneğin Şekil 17 ve Şekil 18'deki gibi bir web uygulamasında resimlerin yalnızca talep edildiğinde yüklenmesini sağlayan sanal proxy, bant genişliğini ve kaynakları tasarruf etmek için kullanılabilir. Örnekte sadece resime tıklandığında resmin sisteme yüklenmesi sağlanıyor. *ImageProxy* sınıfında resmin dosya yolu var ancak sadece display metodu çağrırlırsa resim sisteme yükleniyor. Bu sayede büyük projelerde kaynak tasarrufu sağlanabilir.

IX. SONUÇ

Yapısal tasarım kalıpları, yazılım mimarisini organize etmek ve optimize etmek için güçlü çözümler sunar. Bu kalıpları

etkili bir şekilde uygulayarak geliştiriciler, hem ölçeklenebilir hem de sürdürülebilir sistemler oluşturabilir. Gelecekteki çalışmalar, giderek daha karmaşık tasarım zorluklarını ele almak için kalıpların birleştirilmesini inceleyebilir.

KAYNAKÇA

- [1] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994
- [2] Refactoring.Guru, "Structural Design Patterns", <https://refactoring.guru/design-patterns/structural-patterns>
- [3] InitGrep, "Yapısal Tasarım Kalıpları", <https://www.initgrep.com/posts/design-patterns/structural-design-patterns-java>
- [4] Medium, "Structural Design Patterns with Real Examples in Java", <https://medium.com/javarevisited/top-structural-design-patterns-with-real-examples-in-java-7eede31bde45>

```

public interface PaymentGateway {
    void processPayment(double amount);
}

public class GatewayA implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment with gateway A: $" + amount);
    }
}

public class GatewayB {
    public void charge(double amount) {
        System.out.println("Charging payment with GatewayB: $" + amount);
    }
}

public class GatewayBAdapter implements PaymentGateway {
    private GatewayB gatewayB;

    @Override
    public void processPayment(double amount) {
        gatewayB.charge(amount);
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentGateway gateway1 = new GatewayA();
        PaymentGateway gateway2 = new GatewayBAdapter(new GatewayB());

        double amount = 100.0;

        gateway1.processPayment(amount);
        gateway2.processPayment(amount);
    }
}

```

Şekil 8. adapter kalibi örneği

```

abstract class Animal {
    protected BreathingStyle breathingStyle;

    public Animal(BreathingStyle breathingStyle) {
        this.breathingStyle = breathingStyle;
    }

    abstract void display();
    void breathe() {
        breathingStyle.breathe();
    }
}

interface BreathingStyle {
    void breathe();
}

class LungBreathing implements BreathingStyle {
    @Override
    public void breathe() {
        System.out.println("Breathing with lungs.");
    }
}

```

Şekil 9. bridge kalıbı örneği

```

class SkinBreathing implements BreathingStyle {
    @Override
    public void breathe() {
        System.out.println("Breathing through the skin.");
    }
}
class GillBreathing implements BreathingStyle {
    @Override
    public void breathe() {
        System.out.println("Breathing with gills.");
    }
}
class Mammal extends Animal {
    public Mammal(BreathingStyle breathingStyle) {
        super(breathingStyle);
    }

    @Override
    void display() {
        System.out.println("I am a mammal.");
    }
}
class Fish extends Animal {
    public Fish(BreathingStyle breathingStyle) {
        super(breathingStyle);
    }

    @Override
    void display() {
        System.out.println("I am a fish.");
    }
}

```

Şekil 10. bridge kalıbı örneği devamı

```

public interface Car {
    public void assemble();
}

public class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}

public class CarDecorator implements Car {
    protected Car car;

    public CarDecorator(Car c) {
        this.car = c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }
}

public class SportsCar extends CarDecorator {
    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble() {
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}

```

Şekil 11. decorator kalıbı örneği

```

// Flyweight interface
interface UserSession {
    void displaySessionInfo();
}

// Concrete Flyweight: UserSessionFlyweight
class UserSessionFlyweight implements UserSession {
    private String username;
    private String sessionData; // Common session data shared among users

    public UserSessionFlyweight(String username, String sharedData) {
        this.username = username;
        this.sessionData = sharedData;
    }

    @Override
    public void displaySessionInfo() {
        System.out.println("User: " + username);
        System.out.println("Session Data: " + sessionData);
        System.out.println("-----");
    }
}

// Flyweight Factory: UserSessionFactory
class UserSessionFactory {
    private Map<String, UserSession> userSessions = new HashMap<>();
    private String sharedSessionData = "Shared session data for all users"; //

    public UserSession getUserSession(String username) {
        if (!userSessions.containsKey(username)) {
            // Create and store a new user session if it doesn't exist
            UserSession session = new UserSessionFlyweight(username, sharedSessionData);
            userSessions.put(username, session);
        }
        return userSessions.get(username);
    }
}

// Client
public class BankingApp {
    public static void main(String[] args) {
        UserSessionFactory sessionFactory = new UserSessionFactory();

        // Simulate user sessions
        String[] users = {"User1", "User2", "User3", "User1"};

        for (String user : users) {
            UserSession session = sessionFactory.getUserSession(user);
            session.displaySessionInfo();
        }
    }
}

```

Şekil 12. flyweight kalıbı örneği

```

abstract class Account {
    public abstract double getBalance();
}

class CheckingAccount extends Account {
    private double balance;

    public CheckingAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public double getBalance() {
        return balance;
    }
}

class SavingsAccount extends Account {
    private double balance;

    public SavingsAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public double getBalance() {
        return balance;
    }
}

```

Şekil 13. composite kalıbı örneği

```

class AccountPortfolio extends Account {
    private List<Account> accounts = new ArrayList<>();

    public void addAccount(Account account) {
        accounts.add(account);
    }

    public void removeAccount(Account account) {
        accounts.remove(account);
    }

    @Override
    public double getBalance() {
        double totalBalance = 0;
        for (Account account : accounts) {
            totalBalance += account.getBalance();
        }
        return totalBalance;
    }
}

public class Main {
    public static void main(String[] args) {
        Account checking = new CheckingAccount(1500.00);
        Account savings = new SavingsAccount(3000.00);
        Account investment = new InvestmentAccount(5000.00);

        AccountPortfolio portfolio = new AccountPortfolio();
        portfolio.addAccount(checking);
        portfolio.addAccount(savings);
        portfolio.addAccount(investment);

        System.out.println("Total portfolio balance: " + portfolio.getBalance());
    }
}

```

Şekil 14. composite kalıbı örneği devamı

```

public class InventorySystem {
    public boolean checkStock(String item) {
        // Check if the item is in stock
        System.out.println("Checking stock for " + item);
        return true; // Assume the item is in stock
    }

    public void reserveItem(String item) {
        System.out.println("Reserving item: " + item);
    }
}

public class PaymentSystem {
    public void processPayment(String paymentDetails) {
        System.out.println("Processing payment with details: " + paymentDetails);
    }
}

// ShippingSystem.java
public class ShippingSystem {
    public void arrangeShipping(String item, String address) {
        System.out.println("Arranging shipping for " + item + " to " + address);
    }
}

```

Şekil 15. facade kalıbı örneği

```

public class OrderFacade {
    private InventorySystem inventorySystem;
    private PaymentSystem paymentSystem;
    private ShippingSystem shippingSystem;

    public OrderFacade() {
        this.inventorySystem = new InventorySystem();
        this.paymentSystem = new PaymentSystem();
        this.shippingSystem = new ShippingSystem();
    }

    public void placeOrder(String item, String paymentDetails, String address) {
        if (inventorySystem.checkStock(item)) {
            inventorySystem.reserveItem(item);
            paymentSystem.processPayment(paymentDetails);
            shippingSystem.arrangeShipping(item, address);
            System.out.println("Order placed successfully for " + item);
        } else {
            System.out.println("Item " + item + " is out of stock.");
        }
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        OrderFacade orderFacade = new OrderFacade();

        String item = "Laptop";
        String paymentDetails = "Credit Card";
        String address = "123 Main St, Cityville";

        // Placing an order using the Facade
        orderFacade.placeOrder(item, paymentDetails, address);
    }
}

```

Şekil 16. facade kalıbı örneği devamı

```

// ImageProxy class acting as a virtual proxy
class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;

    ImageProxy(String filename) {
        this.filename = filename;
    }

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

public class ProxyPatternExample {
    public static void main(String[] args) {
        // Create an image proxy (virtual proxy)
        Image image = new ImageProxy("large_image.jpg");

        // Image is not loaded until it's displayed
        System.out.println("Image proxy created.");

        // Display the image - the real image is loaded here
        image.display();

        // Displaying the image again - this time, it's already loaded
        image.display();
    }
}

```

Şekil 18. proxy kalıbı örneği devamı

```

interface Image {
    void display();
}

// RealImage class representing the actual image
class RealImage implements Image {
    private String filename;

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

    private void loadFromDisk() {
        System.out.println("Loading image: " + filename);
    }

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

```

Şekil 17. proxy kalıbı örneği