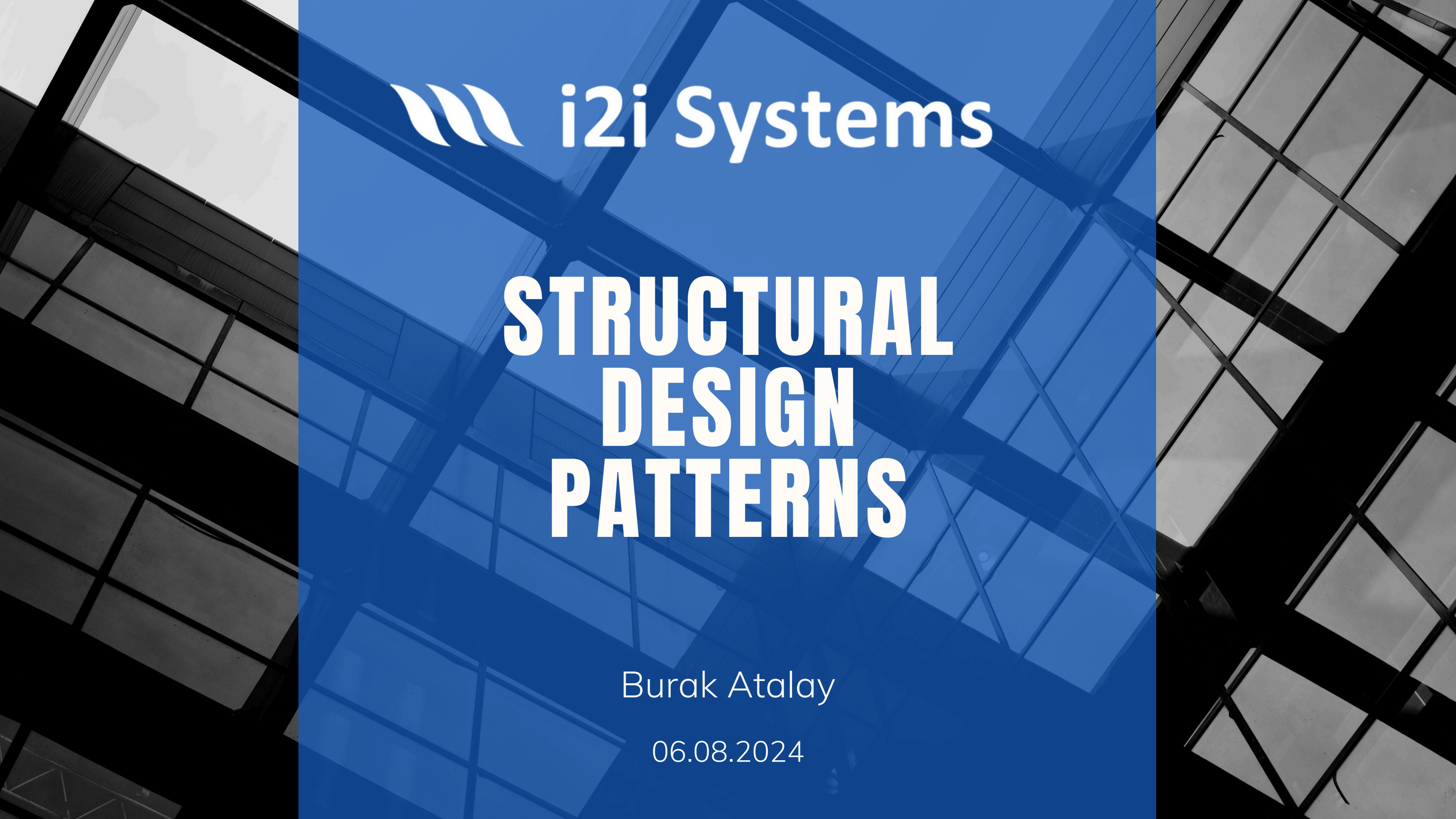


i2i Systems

STRUCTURAL DESIGN PATTERNS

Burak Atalay

06.08.2024

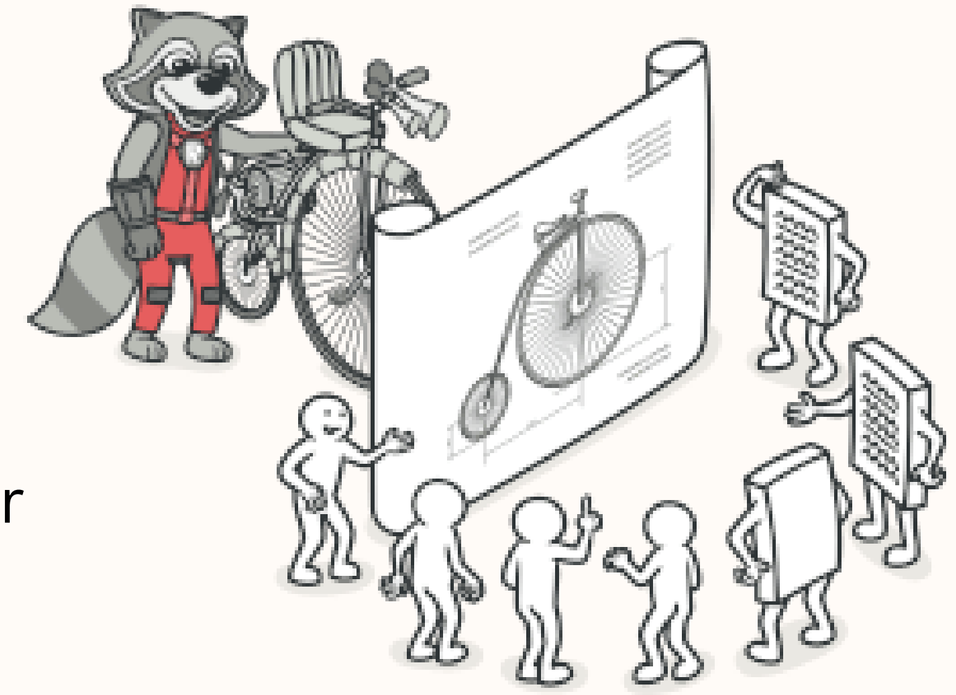
i2i Systems

OVERVIEW

i2i Systems

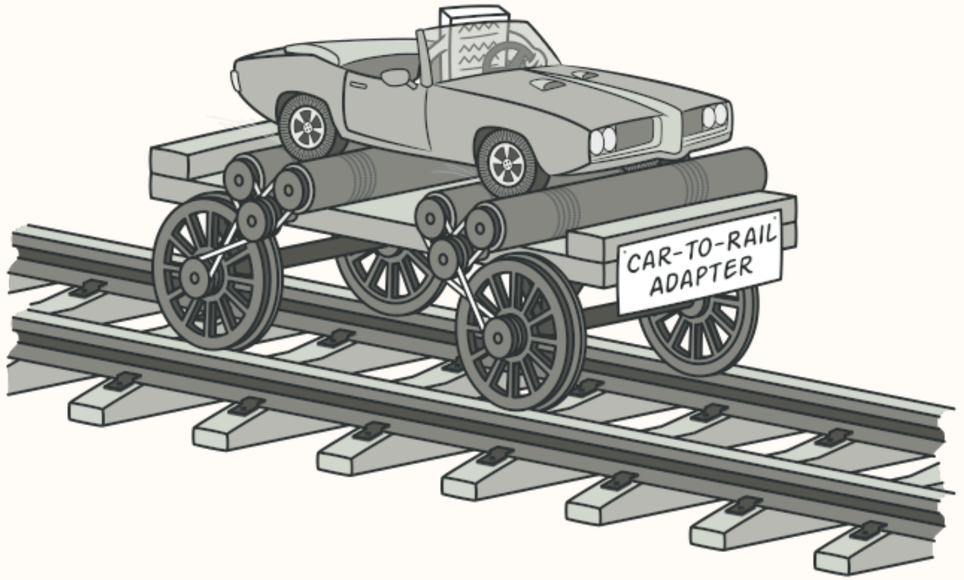
INTRODUCTION

- Design patterns are solutions to commonly occurring problems in software design.
- The pattern is not a specific piece of code, but a general concept that you can apply it in your software
- Structural patterns explains how to assemble objects and classes into larger structures by using composition, while keeping these structures flexible and efficient



ADAPTER

- It allows objects with incompatible interfaces to collaborate
- As the name suggests, it acts as an intermediary to convert an otherwise incompatible interface to one that a client expects



Suppose you have a Retail or E-commerce system that has different payment gateways and you have implemented your first gateway (GatewayA) by implementing an interface

```
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);
    }
}
```

Now imagine you want to add another gateway which has a different implementation. You want to make your second gateway work with the old interface you defined before so you create an adapter which implements the interface.

```
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);
    }
}
```

```
Processing payment with gateway A: $100.0
Charging payment with GatewayB: $100.0
```

i2i Systems

Pros

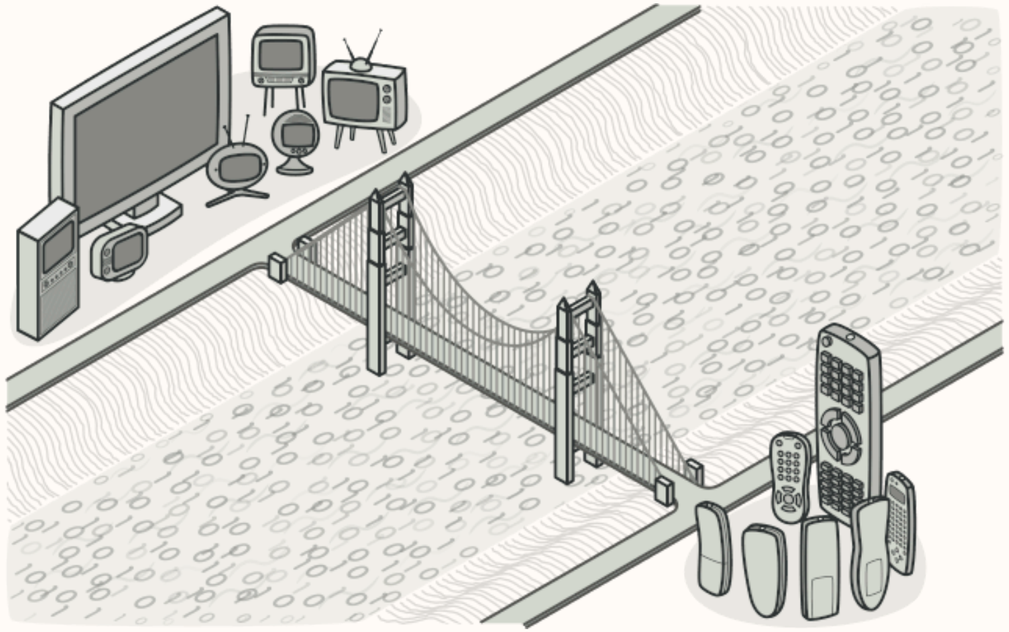
You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface

Cons

The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code

BRIDGE

The Bridge pattern is a structural pattern that decouples an abstraction from its implementation, allowing them to vary independently but still have a way, or bridge, to interact



```
abstract class Animal {
    protected BreathingStyle breathingStyle;

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

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

```
interface BreathingStyle {
    void breathe();
}
```

- In this code example, Animal will be Abstraction part of the Bridge pattern while BreathingStyle will be Implementor part.
- Animal abstract class will store the reference of the BreathingStyle interface and subclasses of Animal class can do breathing by using this reference.

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

```java
class SkinBreathing implements BreathingStyle {
    @Override
    public void breathe() {
        System.out.println("Breathing through the skin.");
    }
}
```

```java
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.");
    }
}
```

```
class Amphibian extends Animal {
    public Amphibian(BreathingStyle breathingStyle) {
        super(breathingStyle);
    }

    @Override
    void display() {
        System.out.println("I am an amphibian.");
    }
}
```

i2i Systems

Pros

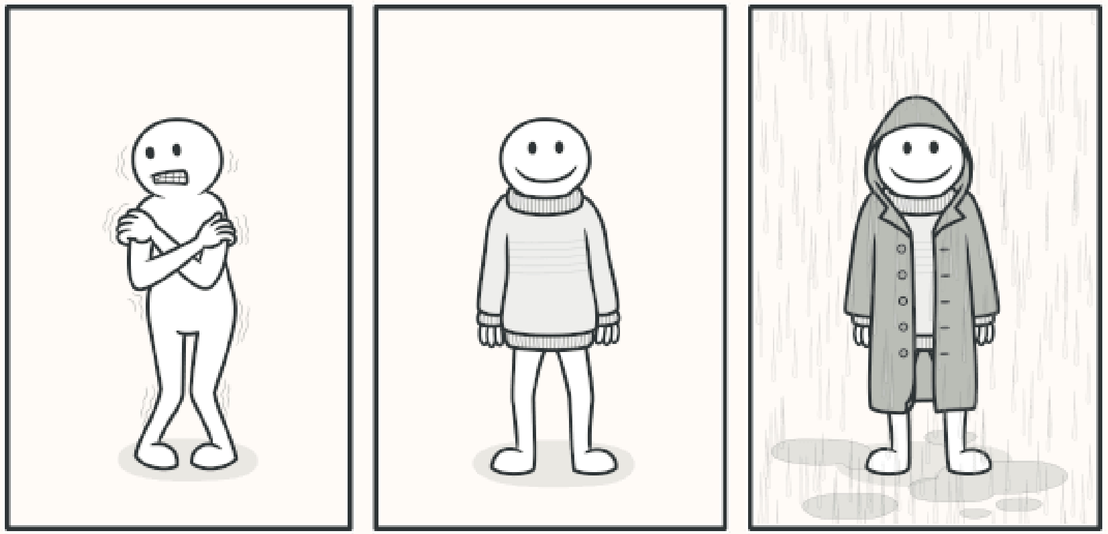
Let’s consider our Animal example; if we do not use Bridge pattern and use directly inheritance, for each combination of animal type and breathing style, a new class would need to be created. If new animal types or new breathing styles need to be added, the number of classes grows exponentially. For instance, adding a Reptile type with both LungBreathing and SkinBreathing would require additional classes

Cons

The Bridge pattern introduces additional layers of abstraction, which can add complexity to the system's design. This added complexity can make the codebase harder to understand and maintain, especially for developers who are not familiar with the pattern

DECORATOR

- Allows for dynamically adding or modifying behaviors of objects at runtime by wrapping them with decorator objects
- Instead of relying on a fixed inheritance hierarchy, the Decorator pattern uses composition. It enables combining objects to extend their functionality without altering the object's class



```
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.");
        }
}
```

i2i Systems

Pros

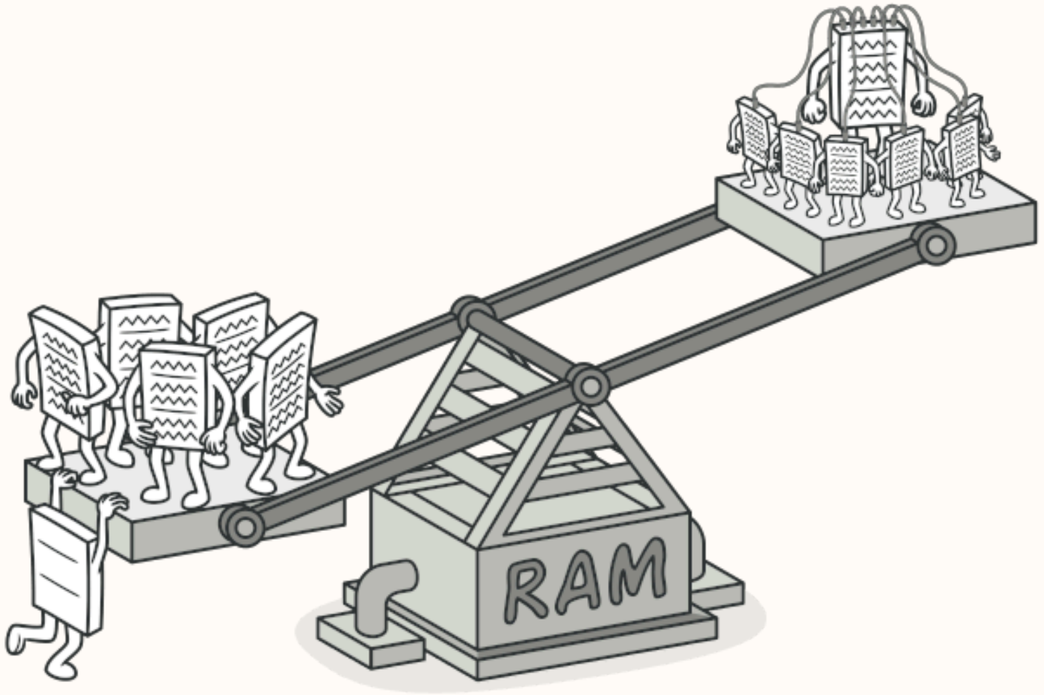
The Decorator pattern allows for dynamically adding or modifying behavior of objects at runtime. This flexibility means that you can easily enhance or change an object's functionality without modifying its class or resorting to a complex inheritance hierarchy

Cons

When multiple decorators are applied to an object, it can be challenging to follow the flow of execution and understand where specific behaviors are being added or altered

FLYWEIGHT

Aims to minimize the memory usage and improve the performance of an application by sharing the common data between objects instead of creating a new object of the common data for each object



This code example is about managing user sessions efficiently with help from Flyweight pattern.
Instead of creating the common session information for each user, a factory class will pass the reference of the common information to the each user.

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

```java
// 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, sharedSess
            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();
        }
    }
}
```

i2i Systems

Pros

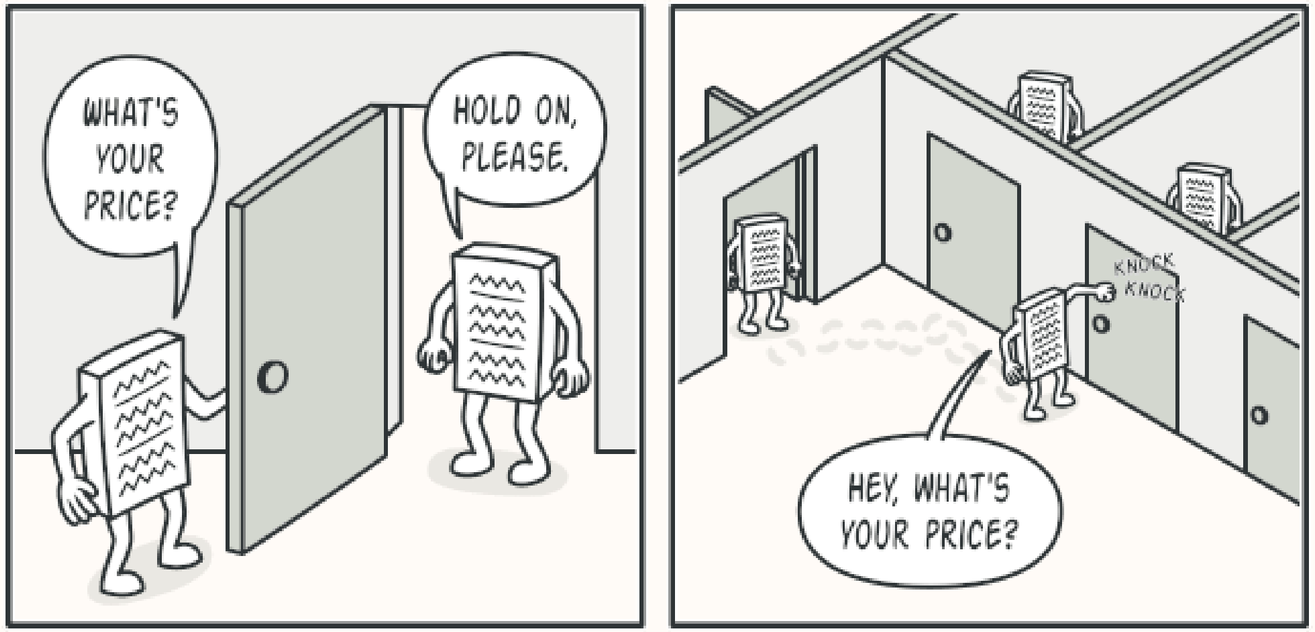
The Flyweight pattern reduces memory usage by sharing common data among multiple objects. Instead of storing the same data in each object, a shared flyweight object holds the intrinsic state, which is used by many clients. This is particularly useful when many similar objects are needed

Cons

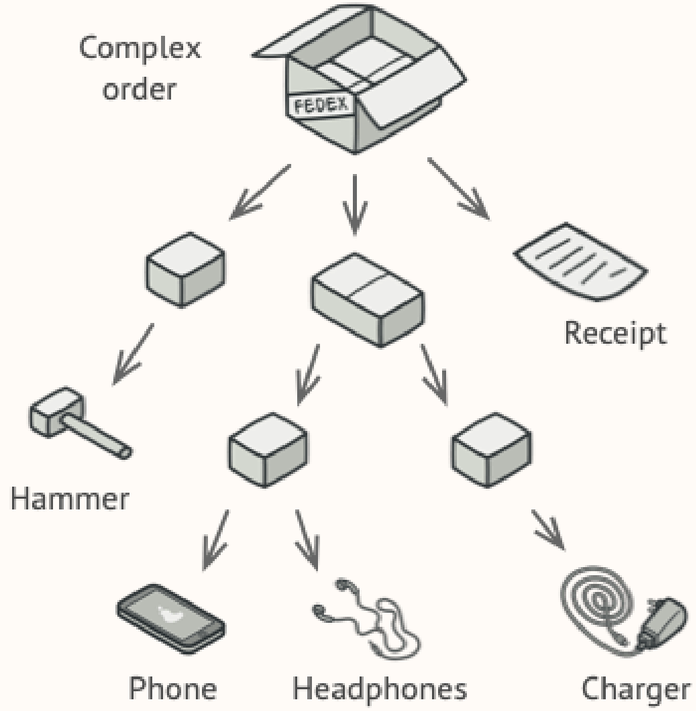
The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way

COMPOSITE

The Composite Pattern lets you compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.



- component -> is the base interface for all the objects in the composition.
- leaf -> implements the default behavior of the base component. It doesn’t contain a reference to the other objects.
- composite -> has leaf elements. It implements the base component methods and defines the child-related operations.
- client -> has access to the composition elements by using the base component object.



This toy example for the Composite pattern is about managing all the bank accounts of a user easily. Account abstract class will be the Component, different account types that inherits from Account will be the Leafs and Portfolio class will be the Composite.

```
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;
    }
}
```

```
class InvestmentAccount extends Account {
    private double balance;

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

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

```java
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());
    }
}
```

Pros

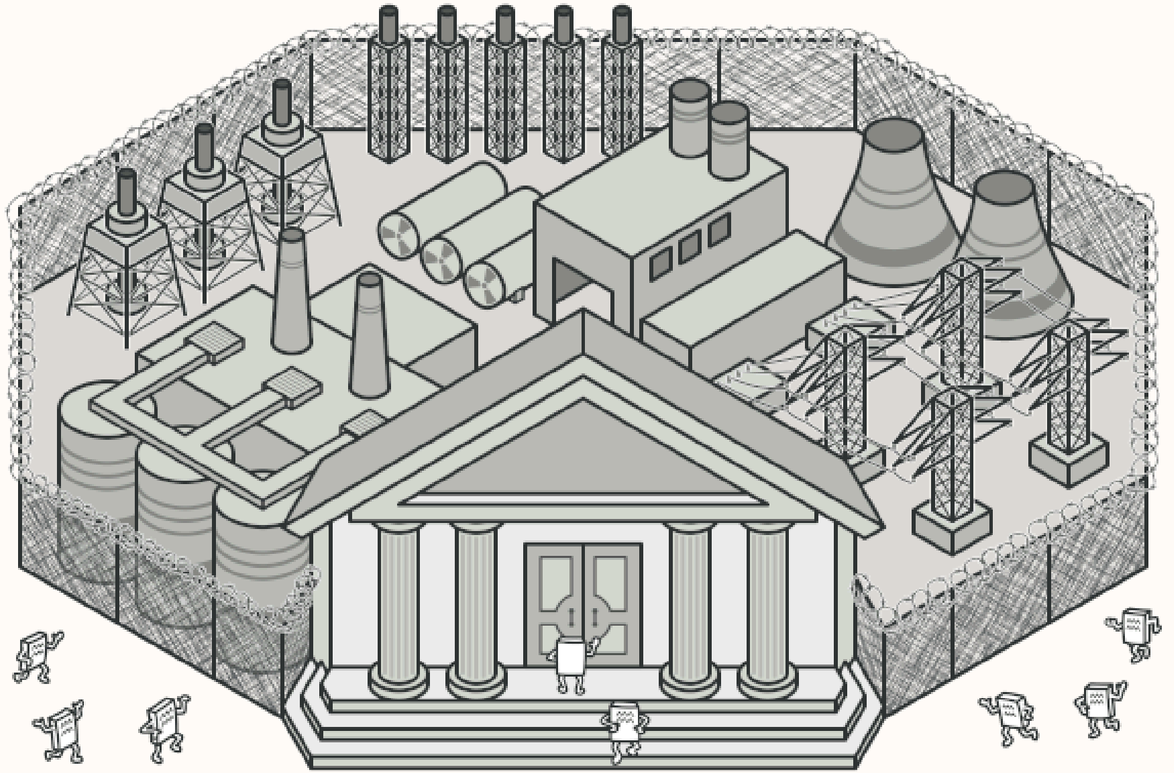
As in the picture you saw in the beginning slide of this pattern, Composite class will take care of it regardless of whether
you are dealing with one specific account of the user or a all the accounts

Cons

It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend

FACADE

- Provides a simplified interface to a complex subsystem
- It acts as a front door and hides the complexity of the subsystem and provides a unified interface to interact with it



This code is a toy example for an online shopping system. It composes all the complex subsystem into a basic interface.

```
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);
    }
}
```

```java
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);
    }
}
```

```java
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.");
        }
    }
}
```

```java
// 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);
    }
}
```

i2i Systems

Pros

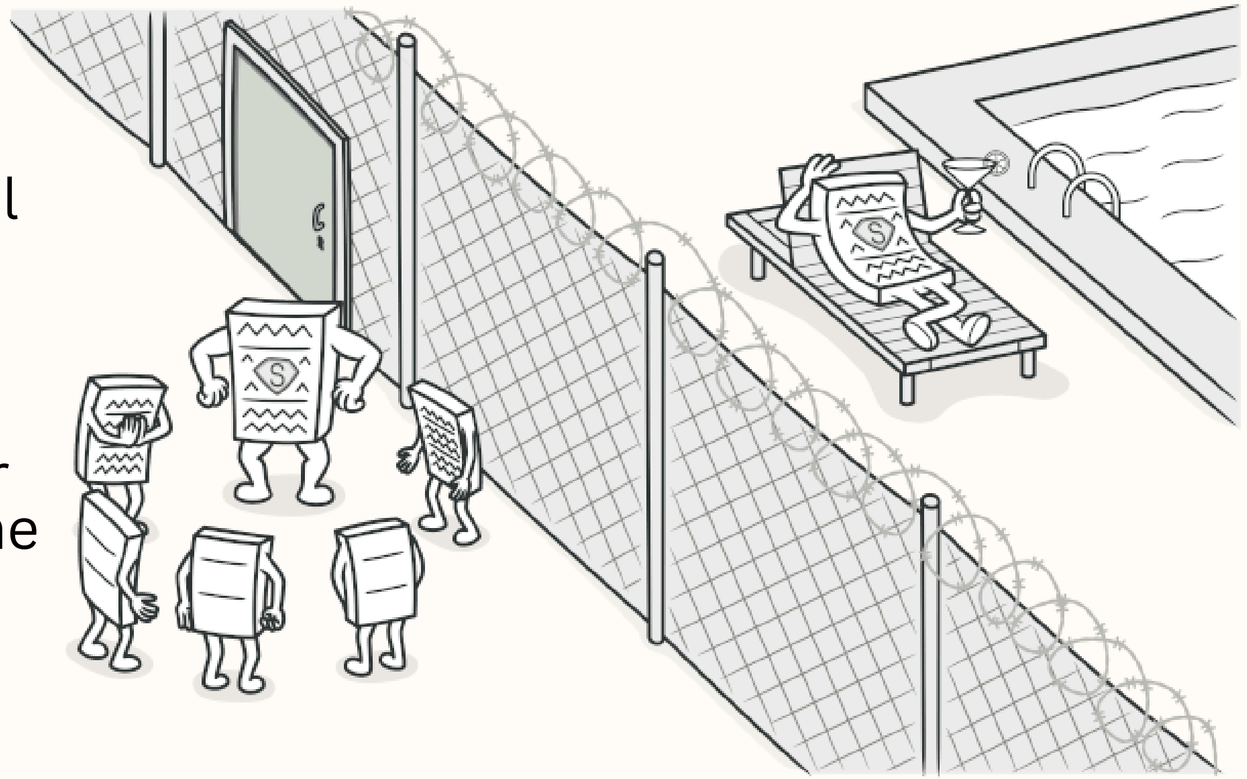
The Facade pattern provides a simplified interface to a complex subsystem, making it easier for clients to use. By offering a higher-level interface, it reduces the number of methods and interactions the client needs to understand and manage

Cons

A Facade can sometimes evolve into a "God Object," which is an anti-pattern where a single class becomes overly large and complex. This occurs if the Facade starts handling too many responsibilities, not just simplifying access but also managing complex logic, coordinating numerous subsystems, and containing too much knowledge about the subsystems

PROXY

- The Proxy Pattern is used to control access to an object
- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object



A common example is a virtual proxy for loading images on a web page. The proxy loads the real image only when it’s requested by the user, saving bandwidth and resources

```
interface Image {
    void display();
}
```

```java
// 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);
    }
}
```

```
// 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();
    }
}
```

i2i Systems

Pros

A common example is the use of the @Lazy annotation in Spring Framework, which delays the initialization of a bean until it is needed. This can improve application startup time and reduce memory usage by not creating objects until they are actually required

Cons

When using the @Lazy annotation, the actual creation and initialization of the object are deferred until it is first accessed. This can lead to delays in response time the first time the object is needed, as the system must perform the initialization at that point. In a scenario where a service call requires the lazy-initialized object, the initial request might experience an unexpected delay

REFERENCES

01 Gang of Four - Design Patterns, Elements of Reusable Object Oriented Software

02 https://refactoring.guru/design-patterns/structural-patterns

03 https://www.initgrep.com/posts/design-patterns/structural-design-patterns-java

04 https://medium.com/javarevisited/top-structural-design-patterns-with-real-examples-in-java-7eede31bde45


i2i Systems
THANK YOU FOR LISTENNING
https://www.linkedin.com/in/burak-atalay-/
https://github.com/burakataly