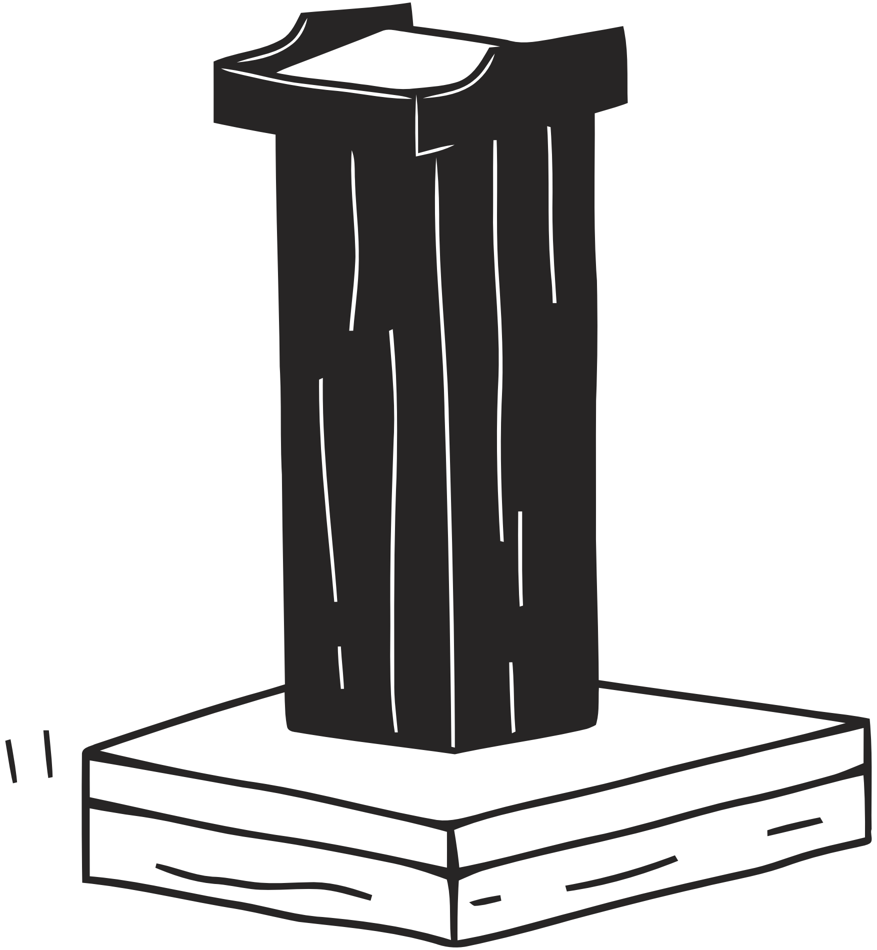
Advanced Programming Techniques – Week 3

Structural Design Patterns

Overview



Structural design patterns focus on how classes and objects are composed

These patterns simplify **object relationships** and improve the organization of code in a scalable way.

Topics Covered:

1. **Adapter** – Converts one interface into another
2. **Bridge** – Decouples abstraction from implementation
3. **Composite** – Treats a group of objects as a single entity
4. **Decorator** – Dynamically extends object behavior
5. **Facade** – Provides a simplified interface to a complex system
6. **Proxy** – Controls access to another object
7. **Flyweight** – Optimizes memory by sharing objects

Introduction to Structural Design Patterns

Definition:

Structural design patterns focus on composing **classes and objects** to create large, scalable architectures while maintaining **flexibility** and **efficiency**.

Why Use Structural Patterns?

✅ Improve **code reuse** and **modularity**

✅ Enhance **scalability** and **flexibility**

✅ Reduce **tight coupling** between classes

✅ Promote separation of concerns

1️⃣ Adapter Pattern

Purpose

The Adapter pattern allows objects with **incompatible interfaces** to work together. It acts as a **middle layer** that translates one interface into another.

🔹 When to Use Adapter?

- When integrating **legacy code** into a new system.
- When using **third-party libraries** with different interfaces.

UML Diagram

```
classDiagram
class Target { +request() }
class Adapter { +request() }
class Adaptee { +specificRequest() }
Target <|.. Adapter
Adapter ..|> Adaptee
```

📌 Java Example

```java
// Old interface (Legacy Code)
class OldSystem {
  void oldMethod() { System.out.println("Using the old system"); }
}

// New interface (Client expects this)
interface NewInterface { void newMethod(); }

// Adapter to make OldSystem work with NewInterface
class Adapter implements NewInterface {
  private OldSystem oldSystem;

  public Adapter(OldSystem oldSystem) { this.oldSystem = oldSystem; }

  @Override
  public void newMethod() { oldSystem.oldMethod(); }
}

// Client Code
public class AdapterExample {
  public static void main(String[] args) {
    NewInterface adapter = new Adapter(new OldSystem());
    adapter.newMethod(); // Works with the new interface
  }
}
```

Adapter Pattern: Pros and Cons

Advantages

- Enables **compatibility** between incompatible interfaces
- Encourages **code reuse**

Pitfalls

- Can introduce **additional complexity**
- May lead to performance overhead

2️⃣ Bridge Pattern

Purpose

The Bridge pattern **separates abstraction from implementation**, allowing them to be developed independently.

UML Diagram (Mermaid.js)

```
classDiagram
class Abstraction { +Implementation impl +operation() }
class Implementation { +operationImpl() }
class ConcreteImplementationA { +operationImpl() }
class ConcreteImplementationB { +operationImpl() }
Abstraction --> Implementation
Implementation <|-- ConcreteImplementationA
Implementation <|-- ConcreteImplementationB
```

Bridge Pattern Example

```
interface Color { void applyColor(); }

class Red implements Color {
  public void applyColor() { System.out.println("Applying red color"); }
}

abstract class Shape {
  protected Color color;
  public Shape(Color color) { this.color = color; }
  abstract void draw();
}

class Circle extends Shape {
  public Circle(Color color) { super(color); }

  public void draw() {
    System.out.print("Drawing Circle with ");
    color.applyColor();
  }
}

public class BridgeExample {
  public static void main(String[] args) {
    Shape redCircle = new Circle(new Red());
    redCircle.draw();
  }
}
```

Bridge Pattern: Pros and Cons

1

Advantages

Decouples abstraction from implementation

2

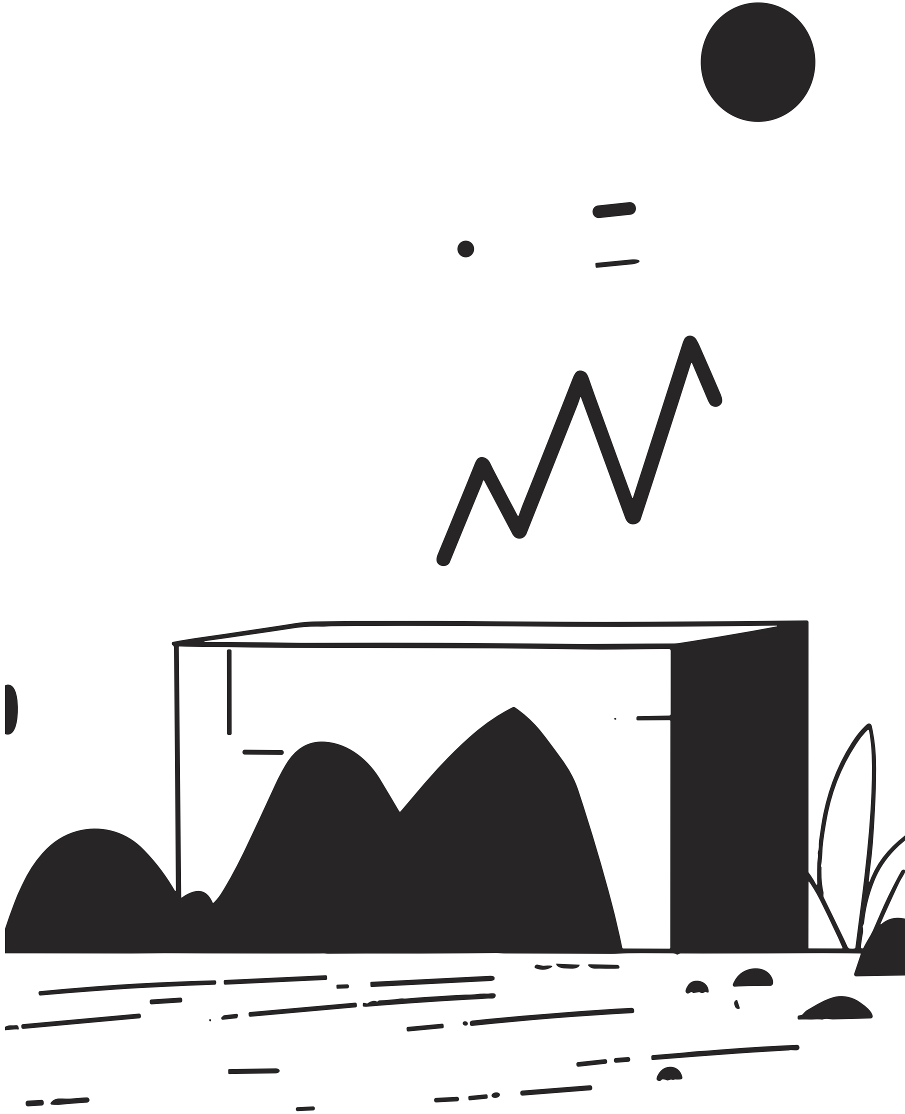
Benefits

Makes systems **scalable and maintainable**

1

Pitfalls

Complexity may increase if not necessary



3️⃣ Composite Pattern

📌 Purpose

The Composite pattern allows treating **a group of objects the same way as individual objects**.

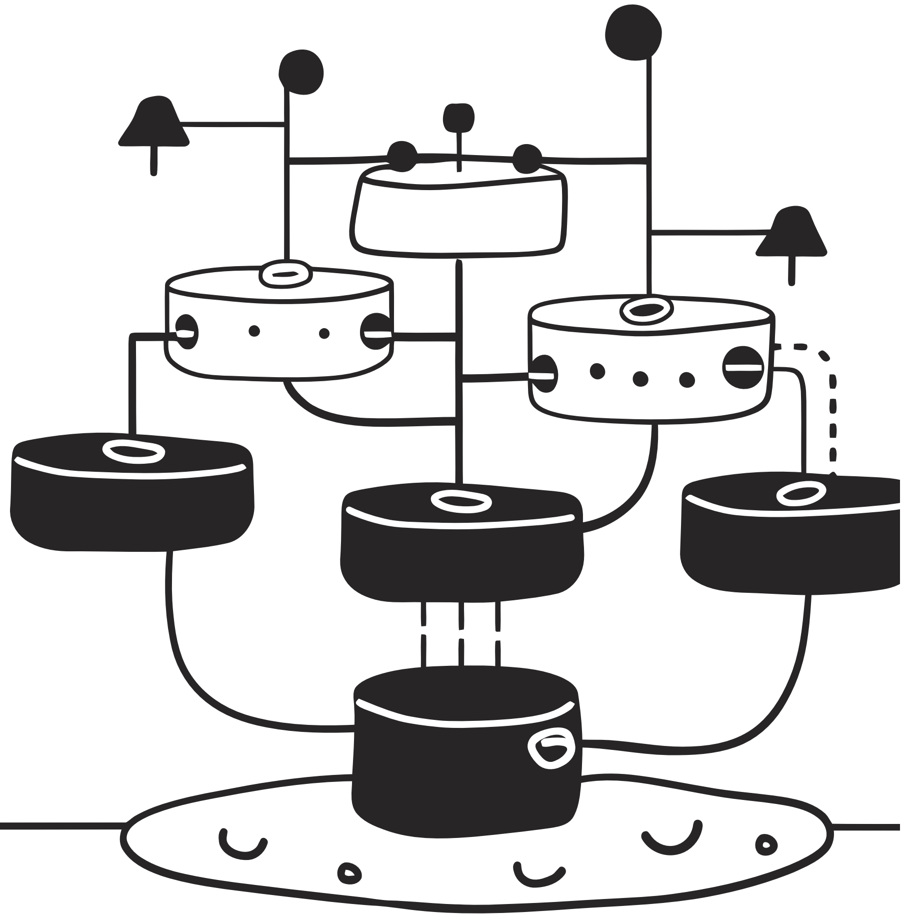
📌 UML Diagram (Mermaid.js)

```
classDiagram
class Component { +operation() }
class Leaf { +operation() }
class Composite {
  +operation()
  +add(Component)
  +remove(Component)
}
Component <|-- Leaf
Component <|-- Composite
```

Composite Pattern Example

```
interface Component { void showDetails(); }

class Leaf implements Component {
  private String name;

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

  public void showDetails() {
    System.out.println(name);
  }
}
```

```
class Composite implements Component {
  private List children = new ArrayList<>();

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

  public void showDetails() {
    for (Component component: children) {
      component.showDetails();
    }
  }
}

public class CompositeExample {
  public static void main(String[] args) {
    Composite folder = new Composite();
    folder.add(new Leaf("File 1"));
    folder.add(new Leaf("File 2"));
    folder.showDetails();
  }
}
```



Composite Pattern: Pros and Cons

✅ Advantages

- Simplifies working with **hierarchical structures**.
- Supports **recursive operations**.

⚠️ Pitfalls

- Can **complicate debugging**.

4️⃣ Decorator Pattern

📌 Purpose

The Decorator pattern allows you to **dynamically extend the functionality of an object** without modifying its structure.

🔹 When to Use Decorator?

- When you need to **add responsibilities dynamically** to objects.
- When subclassing would create **too many subclasses**.

📌 UML Diagram (Mermaid.js)

```
classDiagram
class Component { +operation() }
class ConcreteComponent { +operation() }
class Decorator { +Component component +operation() }
class ConcreteDecoratorA { +operation() }
class ConcreteDecoratorB { +operation() }
Component <|-- ConcreteComponent
Component <|-- Decorator
Decorator <|-- ConcreteDecoratorA
Decorator <|-- ConcreteDecoratorB
Decorator --> Component
```

Decorator Pattern Example

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

class ConcreteComponent implements Component {
  public void operation() {
    System.out.println("Basic operation");
  }
}

class Decorator implements Component {
  protected Component component;

  public Decorator(Component component) {
    this.component = component;
  }

  public void operation() {
    component.operation();
  }
}
```

```
class ConcreteDecoratorA extends Decorator {
  public ConcreteDecoratorA(Component component) {
    super(component);
  }

  public void operation() {
    super.operation();
    System.out.println("Adding feature A");
  }
}

public class DecoratorExample {
  public static void main(String[] args) {
    Component decorated = new ConcreteDecoratorA(new ConcreteComponent());
    decorated.operation();
  }
}
```

✅ Advantages

Adds behavior dynamically at runtime.

Avoids **subclass explosion**.

⚠️ Pitfalls

May lead to **complexity if overused**.

5️⃣ Facade Pattern

📌 Purpose

The Facade pattern **hides the complexity** of a subsystem and provides a **simplified interface** for clients.

📌 UML Diagram (Mermaid.js)

```
classDiagram
class Facade { +operation() }
class SubsystemA { +operationA() }
class SubsystemB { +operationB() }
Facade --> SubsystemA
Facade --> SubsystemB
```

Facade Pattern Example

```
class SubsystemA {
  void operationA() {
    System.out.println("Subsystem A operation");
  }
}

class SubsystemB {
  void operationB() {
    System.out.println("Subsystem B operation");
  }
}

class Facade {
  private SubsystemA a = new SubsystemA();
  private SubsystemB b = new SubsystemB();

  public void operation() {
    a.operationA();
    b.operationB();
  }
}
```

```
public class FacadeExample {
  public static void main(String[] args) {
    Facade facade = new Facade();
    facade.operation(); // Simplified interface
  }
}
```

✅ Advantages

Simplifies complex systems.

Reduces dependencies on multiple subsystems.

⚠️ Pitfalls

Hides functionality, which might reduce flexibility.

6️⃣ Flyweight Pattern

📌 Purpose

The Flyweight pattern **optimizes memory usage** by **sharing objects** instead of creating new instances.

📌 UML Diagram (Mermaid.js)

```
classDiagram
class Flyweight { +operation(extrinsicState) }
class ConcreteFlyweight { +operation(extrinsicState) }
class FlyweightFactory { +getFlyweight() }
Flyweight <|-- ConcreteFlyweight
FlyweightFactory --> Flyweight
```

Flyweight Pattern Example

```
interface Flyweight {
  void operation(String extrinsicState);
}

class ConcreteFlyweight implements Flyweight {
  private String intrinsicState;

  public ConcreteFlyweight(String intrinsicState) {
    this.intrinsicState = intrinsicState;
  }

  public void operation(String extrinsicState) {
    System.out.println("Intrinsic: " + intrinsicState +
                   ", Extrinsic: " + extrinsicState);
  }
}
```

```
class FlyweightFactory {
  private Map cache = new HashMap<>();

  public Flyweight getFlyweight(String key) {
    if (!cache.containsKey(key)) {
      cache.put(key, new ConcreteFlyweight(key));
    }
    return cache.get(key);
  }
}

public class FlyweightExample {
  public static void main(String[] args) {
    FlyweightFactory factory = new FlyweightFactory();
    Flyweight shared1 = factory.getFlyweight("SharedState");
    shared1.operation("Instance1");
    Flyweight shared2 = factory.getFlyweight("SharedState");
    shared2.operation("Instance2");
  }
}
```

✅ Advantages

Reduces memory consumption by reusing objects.

⚠️ Pitfalls

Complex to implement.

7️⃣ Proxy Pattern

📌 Purpose

The Proxy pattern provides a **placeholder** for another object to **control access to it**.

📌 UML Diagram (Mermaid.js)

```
classDiagram
class Subject { +request() }
class RealSubject { +request() }
class Proxy { +request() }
Subject <|-- RealSubject
Subject <|-- Proxy
Proxy --> RealSubject
```

Proxy Pattern Example

```
interface Subject {
  void request();
}

class RealSubject implements Subject {
  public void request() {
    System.out.println("RealSubject handling request");
  }
}
```

```
class Proxy implements Subject {
  private RealSubject realSubject;

  public void request() {
    if (realSubject == null) {
      realSubject = new RealSubject();
    }
    System.out.println("Proxy controls access to RealSubject");
    realSubject.request();
  }
}

public class ProxyExample {
  public static void main(String[] args) {
    Subject proxy = new Proxy();
    proxy.request(); // Proxy controls access
  }
}
```

✅ Advantages

Adds security & lazy initialization.

⚠️ Pitfalls

May introduce latency.

Comparison: Facade vs. Proxy Pattern

Aspect	Facade	Proxy
Purpose	Simplifies access to a **complex subsystem** by providing a unified interface.	Controls access to a **real object**, adding security, logging, or lazy loading.
Use Case	When you want to make a system easier to use by **hiding complexity**.	When you need **indirect access** to an object for **security, performance, or logging reasons**.
Structure	Calls multiple subsystems and coordinates their use.	Acts as an **intermediary** to an existing object.
Example	A media library providing a **simple interface** to multiple audio and video processing classes.	A **lazy-loading image proxy** that loads an image **only when needed**.

Use Facade when:

- The system is **too complex**, and you need a **simplified interface**.
- Clients should not interact with **multiple components** directly.

Use Proxy when:

- You need **lazy initialization** (e.g., loading a large object **only when required**).
- You want to add **security, access control, or logging** before an object is accessed.
- The object is **remote** (e.g., a **networked resource**).