# Design Pattern

BCS1430

Dr. Ashish Sai

📅 Week 5 Lecture 1

💻 BCS1430.ashish.nl

📍 EPD150 MSM Conference Hall

## Structural Design Patterns

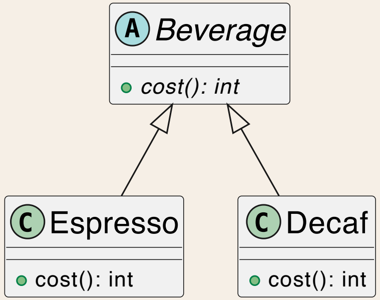
Structural patterns focus on assembling objects and classes into larger structures while maintaining efficiency and flexibility.

## All Structural Patterns

| **Pattern** | **Description** | **Covered** |
|---|---|---|
| Adapter | Allows the interface of an existing class to be used as another interface. | ✅ |
| Bridge | Separates an object's abstraction from its implementation so that they can vary independently. | ❌ |
| Composite | Composes objects into tree structures to represent part-whole hierarchies. | ❌ |
| Decorator | Attaches additional responsibilities to an object dynamically. | ✅ |
| Facade | Provides a unified interface to a set of interfaces in a subsystem. | ✅ |
| Flyweight | Uses sharing to support a large number of fine-grained objects efficiently. | ❌ |
| Proxy | Provides a surrogate or placeholder for another object to control access to it. | ✅ |

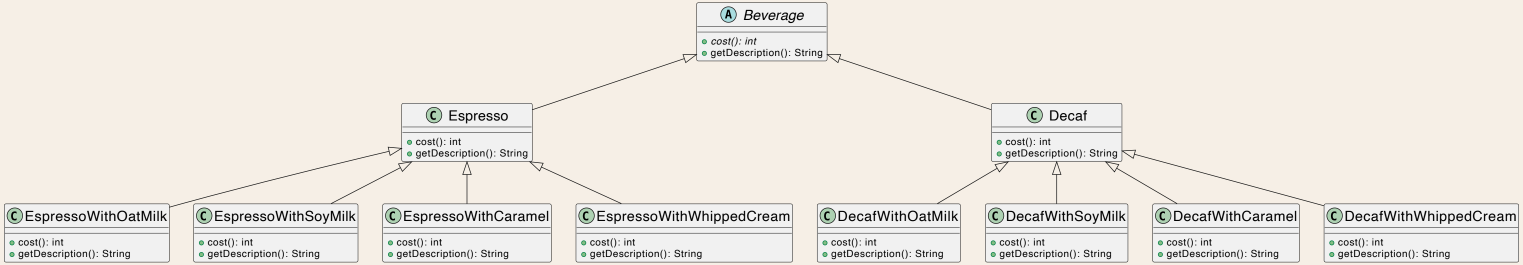
# Decorator Pattern

# Espresso Yourself Café




- You want to offer an option of add-ons (caramel, soy milk etc)

# Oh no!



Beverage
cost(): int
getDescription(): String
Espresso
cost(): int
getDescription(): String
Decaf
cost(): int
getDescription(): String
EspressoWithOatMilk
cost(): int
getDescription(): String
EspressoWithSoyMilk
cost(): int
getDescription(): String
EspressoWithCaramel
cost(): int
getDescription(): String
EspressoWithWhippedCream
cost(): int
getDescription(): String
DecafWithOatMilk
cost(): int
getDescription(): String
DecafWithSoyMilk
cost(): int
getDescription(): String
DecafWithCaramel
cost(): int
getDescription(): String
DecafWithWhippedCream
cost(): int
getDescription(): String

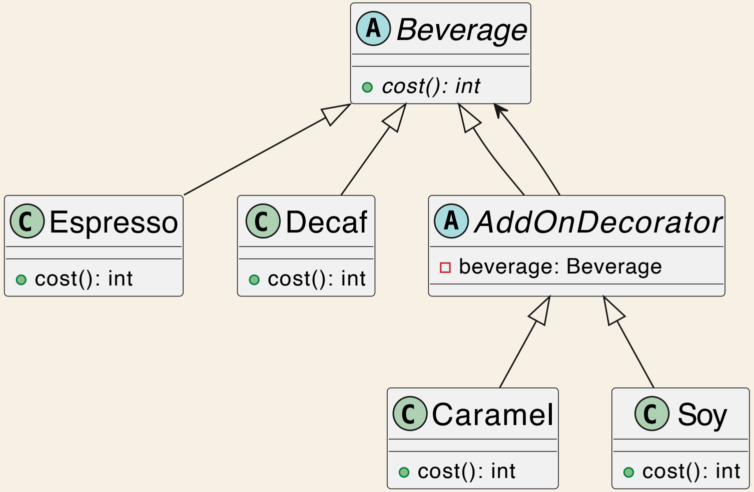
## The Problem Statement

- The need to extend the functionality of objects dynamically.
- Avoiding “class explosion” for similar yet distinct objects.
- Example: Different types of coffee in a coffee house application.

# Class Explosion Problem

- Multiple classes for each combination of coffee and add-ons (e.g., Espresso with Caramel, Decaf with Soy, etc.).
- Results in a large, unmanageable number of subclasses.

# How to solve it?



# Introduction to the Decorator Pattern

- Purpose: Dynamically adds behaviors to objects without modifying their structure.
- Key Concept: Wraps additional behaviors around objects to enhance or modify their functionality.

## Decorator Pattern: Intent

- Attach new behaviors to objects dynamically.
- Provide a flexible alternative to subclassing for extending functionality.

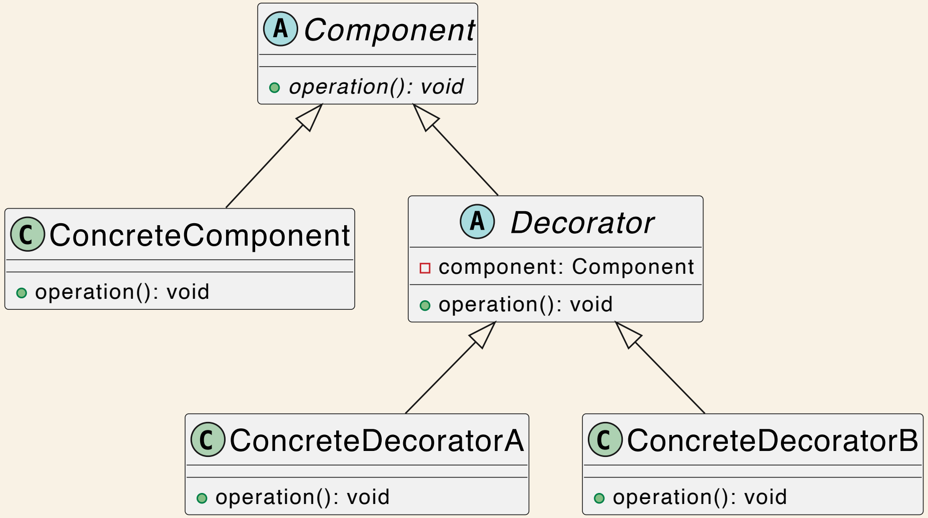
## Decorator Pattern: Issues with Subclassing

- `Combinatorial explosion of subclasses.`
- `Inflexibility to combine multiple behaviors dynamically.`

## How It Works

- Wraps the original object inside objects containing new behaviors.
- Each wrapper (decorator) adds its behavior either before or after delegating to the wrapped object.

# Decorator Pattern Structure



# Applying Decorator Pattern - Example

- Decorators for each add-on (e.g., caramel, soy milk).

```java
public abstract class Beverage {
    public abstract int cost();
}

public class Espresso extends Beverage {
    public int cost() {
        return 1; // Base cost for Espresso
    }
}

public abstract class AddOnDecorator extends Beverage {
    protected Beverage beverage;
}
```

# Concrete Decorators

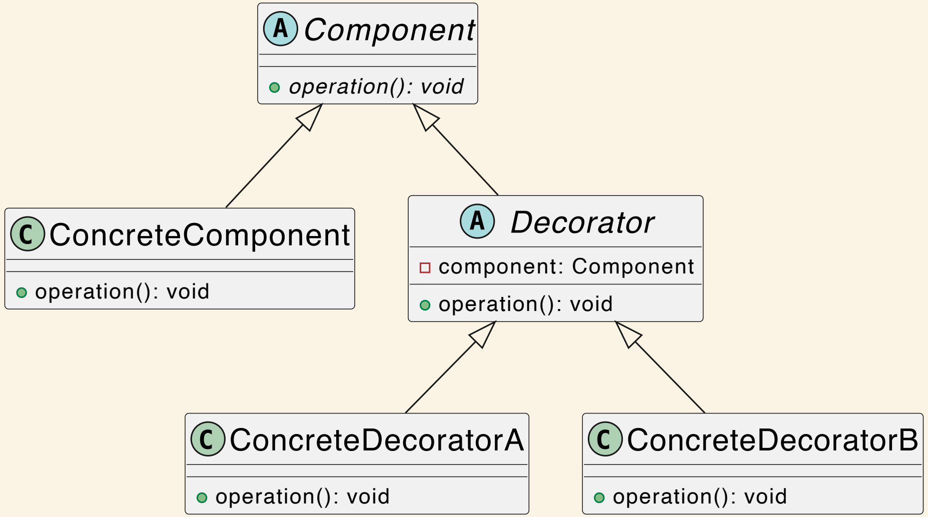
```java
public class CaramelDecorator extends AddOnDecorator {
    public CaramelDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    public int cost() {
        return beverage.cost() + 2; // Adding cost of caramel
    }
}

public class SoyDecorator extends AddOnDecorator {
    public SoyDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    public int cost() {
        return beverage.cost() + 1; // Adding cost of soy
    }
}
```

# Decorator Pattern in Action

- Creating a coffee with add-ons.
- Calculating the total cost dynamically.

```java
public class CoffeeShop {
    public static void main(String[] args) {
        Beverage beverage = new Espresso();
        beverage = new CaramelDecorator(beverage);
        beverage = new SoyDecorator(beverage);

        System.out.println("Total Cost: " + beverage.cost());
    }
}
```

## Decorator Pattern: Decorator Pattern Structure




- **Component:** Common interface for both wrappers and wrapped objects.
- **Concrete Component:** The object being wrapped.
- **Decorator:** Base class for all decorators with a reference to a Component.
- **Concrete Decorators:** Classes that add new behaviors.

## Decorator Pattern: Implementing the Decorator Pattern

- Steps for Implementation

1. Define the component interface.
2. Create a concrete component class.
3. Develop a base decorator class.
4. Implement concrete decorators.

## Decorator Pattern: Key Considerations

- Ensure all components and decorators implement the component interface.
- Decorators should delegate to the wrapped object and add their behavior.

## Another Problem

- You have a different types of data sources (such as Text Files or Database)
- You want to Encrypt the data yous tore or Compress it.

# Java Example - Base Component and Decorator

```java
// Component Interface
interface DataSource {
    void writeData(String data);
    String readData();
}

// Concrete Component
class FileDataSource implements DataSource {
    // Implementation details...
}

// Base Decorator
class DataSourceDecorator implements DataSource {
    protected DataSource wrappee;

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

    public void writeData(String data) {
```

- The base decorator delegates all work to the wrapped component.

# Decorator Pattern: Java Example - Concrete Decorators

```java
// Encryption Decorator
class EncryptionDecorator extends DataSourceDecorator {
    EncryptionDecorator(DataSource source) {
        super(source);
    }

    public void writeData(String data) {
        // Encrypt and write data
    }

    public String readData() {
        // Read and decrypt data
        return "decrypted data";
    }
}

// Compression Decorator
class CompressionDecorator extends DataSourceDecorator {
    CompressionDecorator(DataSource source) {
        super(source);
```

- Each decorator adds its behavior either

# Decorator Pattern: Using the Decorator Pattern

The Decorator Pattern allows stacking multiple decorators to add several behaviors.

- Dynamic Behavior Addition

```
DataSource basicData = new FileDataSource("data.txt");
DataSource encrypted = new EncryptionDecorator(basicData);
DataSource encryptedCompressed = new CompressionDecorator(encrypted);

// Now 'encryptedCompressed' has both encryption and compression capabilities.
```

- The client code can combine decorators in various configurations at runtime.

## Benefits of Decorator Pattern

- Flexibility in adding new functionality.
- Avoids class explosion by using composition over inheritance.
- Easier to maintain and extend.

## Limitations of Decorator Pattern

- Can lead to complex code structures.
- Difficulty in debugging, as it introduces layers of abstraction.
- Potential performance issues due to increased object creation.

## Decorator Pattern vs Subclassing

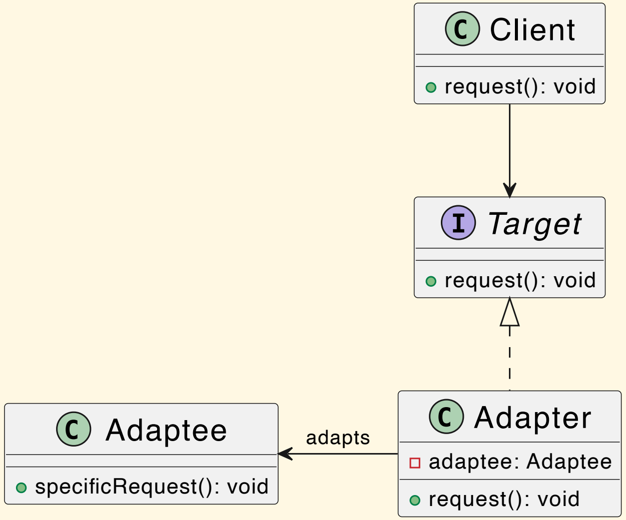
- Decorator Pattern allows for more flexibility than subclassing.
- Avoids rigid class hierarchy.
- Promotes loose coupling and adherence to the Open-Closed Principle.

# Adapter Design Pattern

## The Adapter Pattern

- Purpose: To make two incompatible interfaces compatible.
- Also known as a "wrapper."
- Use Case: Connecting new code to legacy code or third-party libraries.

# Adapter Pattern UML Diagram



# Adapter Pattern Java Example

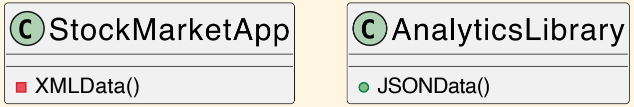
```java
public class Client {
  Target target = new Adapter (new Adaptee());
  target.request();
}

public interface Target {
  void request();
}

class Adapter implements Target {
public Adapter (Adaptee a){
  this.adaptee = a;
}
public void request(){
  this.adaptee.SpecialRequest();
}

}

class Adaptee {
```

## Adapter Pattern: Intent of Adapter Pattern

- The Adapter pattern allows objects with incompatible interfaces to work together.
- It acts as a bridge between two incompatible interfaces, effectively allowing them to communicate.

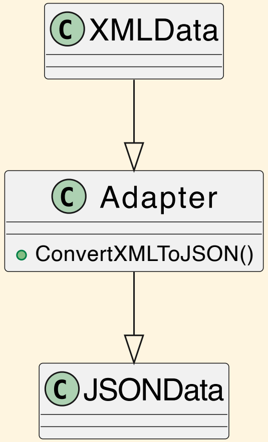
# Adapter Pattern: The Problem

- **Scenario:** Stock market monitoring app downloads data in XML.
- **Challenge:** Integration with a 3rd-party analytics library requiring JSON.
- **Problem:** Incompatibility between data formats (XML vs. JSON).



# Adapter Pattern: The Solution

Enables collaboration by converting XML interface for Analytics Library compatibility.



## Adapter Pattern: Applicability of Adapter Pattern

- **Purpose:** To use an existing class whose interface is incompatible with your code.
- Ideal For:
  - Integrating new classes with old ones.
  - Working with 3rd-party libraries.

# Adapter Pattern: Pros and Cons

**Pros:**

- Single Responsibility Principle: Separates the interface conversion code from the primary business logic.
- Open/Closed Principle: Allows introducing new types of adapters without breaking existing client code.

**Cons:**

- Increases overall complexity due to the introduction of new interfaces and classes.

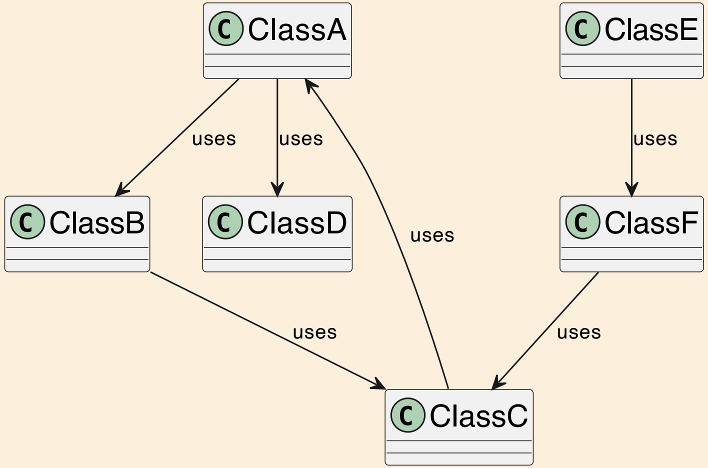
# The Facade Pattern

## The Facade Pattern - Overview

- **Definition:** Simplifies complex system interactions
- **Purpose:** Provide a unified interface to a set of interfaces in a subsystem
- **Key Principle:** High-level abstraction over complex subsystems
- Example: Starting a car (Key Turn → Engine Start, Lights On, etc.)

# Understanding System Complexity

- Multiple classes with intricate interactions
- **Challenge:** Managing complex dependencies and interactions



## The Client’s Perspective

- **Client:** User of a piece of code, not end-user
- **Problem:** Need to interact with complex subsystems

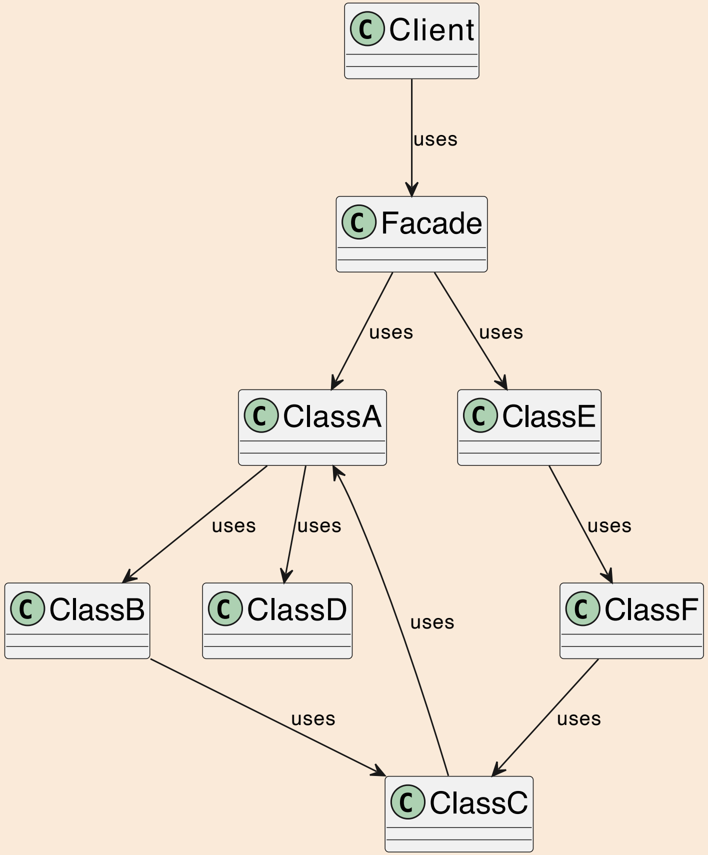
## The Need for the Facade Pattern

- **Complexity:** High due to multiple, interdependent classes
- **Solution:** Simplify interaction using a facade

## Facade Pattern: Real-World Analogy

Consider an operator in a shop as a facade. They provide you with a simple interface to various services and departments of the shop, hiding the complexities of the subsystems behind the scenes.

# Facade Pattern



# Facade Pattern Java Example

```java
// Facade Class
public class CarEngineFacade {
    private Ignition ignition;
    private FuelInjector fuelInjector;
    private AirFlowController airFlowController;

    public CarEngineFacade() {
        ignition = new Ignition();
        fuelInjector = new FuelInjector();
        airFlowController = new AirFlowController();
    }

    public void startEngine() {
        fuelInjector.on();
        airFlowController.takeAir();
        ignition.ignite();
        // Other complex interactions
    }

    public void stopEngine() {
```

## Advantages of Facade Pattern

- **Simplicity:** Provides simple interface to complex subsystems
- **Decoupling:** Clients interact with facade rather than direct subsystem
- **Maintainability:** Changes in subsystems less likely to affect clients

## Facade Pattern: Relations with Other Patterns

- **Adapter vs. Facade:** Adapter wraps one object, while Facade works with an entire subsystem of objects.
- **Facade and Singleton:** Often, a single facade object is sufficient, making it a good candidate for a Singleton.

# Proxy Pattern

## Introduction to Proxy Pattern

- Provides a surrogate or placeholder for another object.
- Controls access to the original object.
- Use cases: Security, Remote Object Access etc.

## Intent of Proxy Pattern

- Purpose: Acts as a substitute to control access to another object.
- Use Cases: Ideal for scenarios needing object access management without changing the object’s behavior.

## Proxy Pattern: Real-World Analogy

A real-world analogy for the Proxy pattern is a credit card acting as a proxy for a bank account, which in turn is a proxy for a bundle of cash. Both provide a means for payment but with additional layers of control and convenience.

## Types of Proxy Patterns

1. **Remote Proxy:** Facilitates access to objects located in different address spaces.
2. **Virtual Proxy:** Delays the creation and initialization of expensive objects until needed.
3. **Protection Proxy:** Controls access to an object based on access rights.

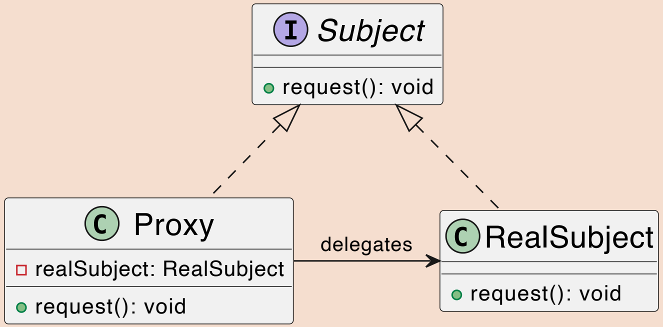
## The Problem

- Access management for resource-intensive objects or services.
- Security assurance complexities.
- Simplification of remote service access.

## The Solution

- Act as an intermediary for controlling access.
- Add Abstraction: Provides a layer to handle security, manage initialization, and simplify remote access.

# Proxy Pattern UML Diagram



# Implementation Examples (Virtual Proxy)

```java
public interface Image {
    void display();
}
public class RealImage implements Image {
    private String fileName;
    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }
    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }
    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}

public class ProxyImage implements Image {
    private RealImage realImage;
```

# Protection Proxy Example in Java

```java
public interface SecureResource {
    void accessResource();
}
public class RealResource implements SecureResource {
    @Override
    public void accessResource() {
        System.out.println("Accessing Secure Resource");
    }
}
public class SecurityProxy implements SecureResource {
    private RealResource realResource;
    private boolean hasAccess;
    public SecurityProxy(boolean hasAccess) {
        this.hasAccess = hasAccess;
        this.realResource = new RealResource();
    }
    @Override
    public void accessResource() {
        if (hasAccess) {
            realResource.accessResource();
```

## Applicability of Proxy Pattern

The Proxy Pattern is highly versatile, applicable in situations requiring:

- Lazy initialization (Virtual Proxy)
- Access control (Protection Proxy)
- Remote object access (Remote Proxy)
- Other use cases like logging, caching, or auditing access

# Proxy Pattern: Pros and Cons

**Pros:**

- Control the service object indirectly.
- Manage the lifecycle and initialization of the service.
- Introduce new proxies without changing the service or clients.

**Cons:**

- Can complicate the code structure with additional classes.
- May introduce latency in the response from the service.

# See you tomorrow!