AMSS Lecture 11: Design Patterns (II)

Traian-Florin Șerbănuță

2025

Agenda

1. Recall
   - 1.1 What are Design Patterns?
   - 1.2 Classification of Patterns
2. Patterns
   - 2.1 **Visitor**
   - 2.2 **Mediator**
   - 2.3 **Bridge**
   - 2.4 **Adapter**
   - 2.5 **Decorator**
   - 2.6 **Proxy**
   - 2.7 **Composite**
3. Wrap-up

What Are Design Patterns?

Definition

Reusable solutions to common software design problems.

Origin

Popularized by the "Gang of Four" (Gamma, Helm, Johnson, Vlissides, 1994).

Purpose

- Provide shared vocabulary
- Improve code maintainability
- Promote reusability and clarity

Example

Instead of reinventing how to traverse a collection, we apply the **Iterator** pattern.

Pattern Classification

Design patterns are typically grouped into three main categories:

Category	Description	Example Patterns
Creational	How objects are created	Builder, Factory, Singleton
Structural	How classes and objects are composed	Adapter, Bridge, Composite, Decorator, Proxy
Behavioral	How objects interact and communicate	Visitor, Mediator, State

Visitor Pattern

Type

Behavioral pattern

Intent

- Separate an algorithm from the object structure it operates on
  - Allow new operations without modifying existing classes.

Problem Solved

- How to add new operations to a set of related classes?
  - without changing their source code?

Solution

- Define a *Visitor* interface with visit methods for each element type.
- *Elements* accept a visitor and delegate the operation to it.

Visitor Pattern - key ideas

Elements Objects you want to operate on (e.g., nodes in an AST, shapes in a graphics editor).

Each element implements an accept(visitor) method.

Visitor An object that implements different operations for each concrete element type.

Example methods: `visitCircle(circle)`, `visitSquare(square)`, `visitTriangle(triangle)`

Double Dispatch A crucial mechanism:

The element calls `visitor.visitXYZ(this)`

The visitor chooses the correct method based on the element type

This avoids type-checking or if/instanceof cascades.

Visitor Pattern — concrete example scenario

Problem

- You maintain a graphics library containing shapes like Circle and Rectangle.
- You frequently need to add new operations—such as area calculation, exporting, rendering—
  - but you want to avoid modifying the existing shape classes every time.

Visitor Pattern — concrete example scenario

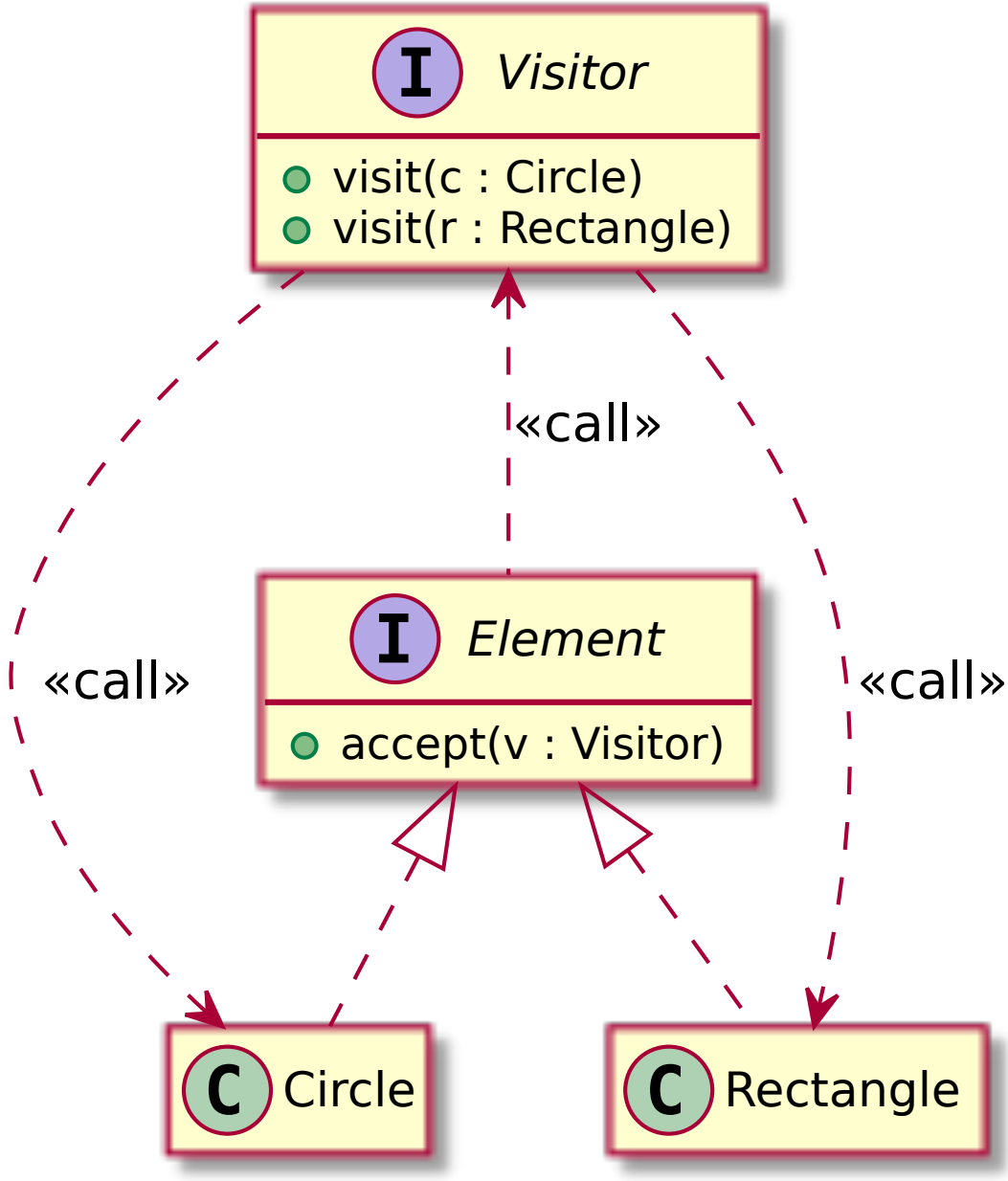
Problem

- You maintain a graphics library containing shapes like Circle and Rectangle.
- You frequently need to add new operations—such as area calculation, exporting, rendering—
  - but you want to avoid modifying the existing shape classes every time.

Solution

The Visitor pattern lets you add new operations by creating new Visitor classes, while the shapes themselves remain unchanged and simply “accept” visitors.

Visitor Pattern — UML class diagram



Visitor Pattern — concrete example code

Source file

```
// Element
interface Shape {
    void accept(Visitor v);
}

// Concrete Elements
class Circle implements Shape {
    double radius = 5;
    public void accept(Visitor v) { v.visit(this); }
}

class Rectangle implements Shape {
    double width = 4, height = 3;
    public void accept(Visitor v) { v.visit(this); }
}

// Visitor
```

Visitor Pattern - benefits and drawbacks

Benefits

- Easy to add new operations
  - Add a new visitor class, no changes to element classes
- Keeps element classes small
  - Offloads complex logic
- Great for tree-like structures
  - Compilers, interpreters, document processors, etc

Drawbacks

- Hard to add new element types
  - Every visitor must be updated
- Increased coupling
  - Visitors need access to element internals
- More boilerplate
  - Especially in statically typed languages

Visitor Pattern Exercise

Task

- Design a system for processing elements in an online editor:
  - Paragraph;
  - Image;
  - Table.
- Define two possible operations:
  - spell-checking;
  - exporting to HTML.
- How would the Visitor pattern let you add these operations?
  - without modifying the element classes

Mediator Pattern

Type

Behavioral pattern

Intent

- Define an object encapsulating how a set of objects interact
  - promoting loose coupling

Problem Solved

How to reduce direct dependencies and complex communication between many interacting objects?

Solution

- Create a `Mediator` object that centralizes communication logic.
- Colleagues communicate only through the mediator.

Mediator Pattern — key concepts

Mediator (Interface / Abstract Class) Defines how components communicate through the mediator.

Concrete Mediator Implements coordination logic.

Receives events from components and decides how to react.

Colleague Components Objects that interact only through the mediator.

They notify the mediator when something happens instead of contacting each other directly.

Mediator Pattern — concrete example scenario

Problem

- In a chat application, every user needs to send messages to others.
- If each user communicated directly with every other user, the system would become highly coupled and difficult to maintain.

Mediator Pattern — concrete example scenario

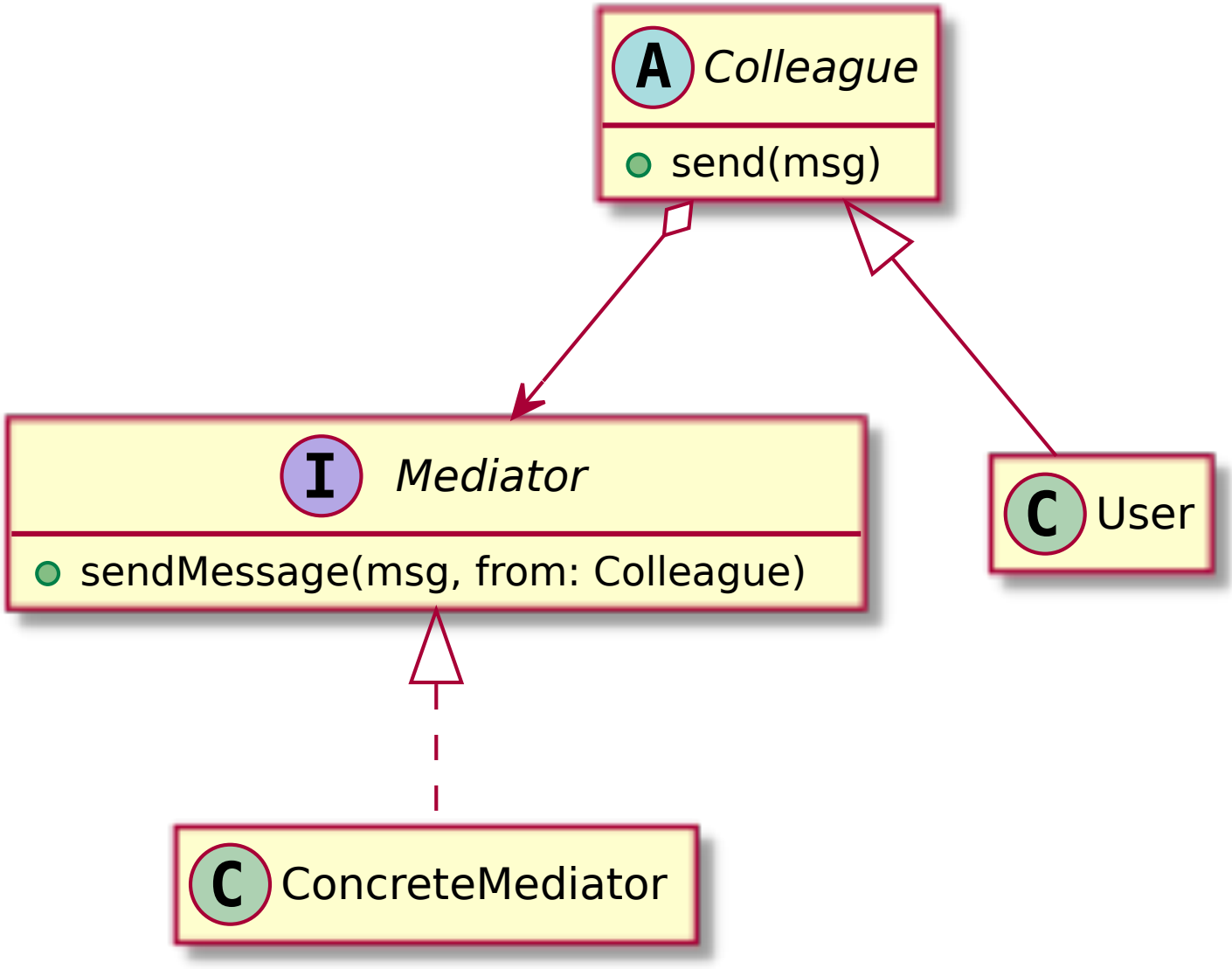
Problem

- In a chat application, every user needs to send messages to others.
- If each user communicated directly with every other user, the system would become highly coupled and difficult to maintain.

Solution

- The Mediator pattern introduces a central ChatRoom that manages all communication.
- Users send messages through the mediator, drastically simplifying interaction.

Mediator Pattern — UML class diagram



Mediator Pattern — concrete example code

Source file

```
// Mediator
interface ChatMediator {
    void sendMessage(String msg, User user);
}

// Concrete Mediator
class ChatRoom implements ChatMediator {
    public void sendMessage(String msg, User user) {
        System.out.println(user.getName() + ": " + msg);
    }
}

// Colleague
abstract class User {
    protected ChatMediator mediator;
    protected String name;
    User(String name, ChatMediator mediator) {
```

Mediator Pattern — Benefits & Tradeoffs

Benefits

- Loose coupling
  - Components don't depend on each other's implementation.
- Centralized control
  - Collaboration logic is in one place.
- Easier maintenance
  - Changes affect fewer places.

Drawbacks

- Mediator can become too large ("God Object")
  - If it takes on too much logic, the pattern backfires.
- May hide complexity instead of removing it.

Mediator Pattern Exercise

Task

- Imagine a smart home system where devices (lights, thermostat, alarm, blinds) must coordinate actions (e.g., “away mode”).
- Design a Mediator that centralizes communication so devices do not directly reference or call each other.
- Outline the mediator role and how devices interact with it.

Bridge Pattern

Type

Structural pattern

Intent

- Decouple an abstraction from its implementation
  - so that the two can vary independently.

Problem Solved

How to avoid a class explosion caused by combining multiple abstractions with multiple implementations?

Solution

- Split abstraction and implementation into separate class hierarchies
  - connecting them via a bridge interface.

Bridge Pattern — key components

Abstraction Defines high-level control logic.

Maintains a reference to an implementation.

Refined Abstraction Specialized abstractions that extend the base abstraction.

Implementor (Interface or Abstract Class) Defines low-level platform-specific operations.

Concrete Implementor Actual implementation details.

Bridge Pattern — concrete example scenario

Problem

- You want to build a universal remote-control system that works with different devices like TVs, Radios, and projectors.
- If you directly subclass for every combination (e.g., AdvancedTVRemote, BasicRadioRemote), you get class explosion.

Bridge Pattern — concrete example scenario

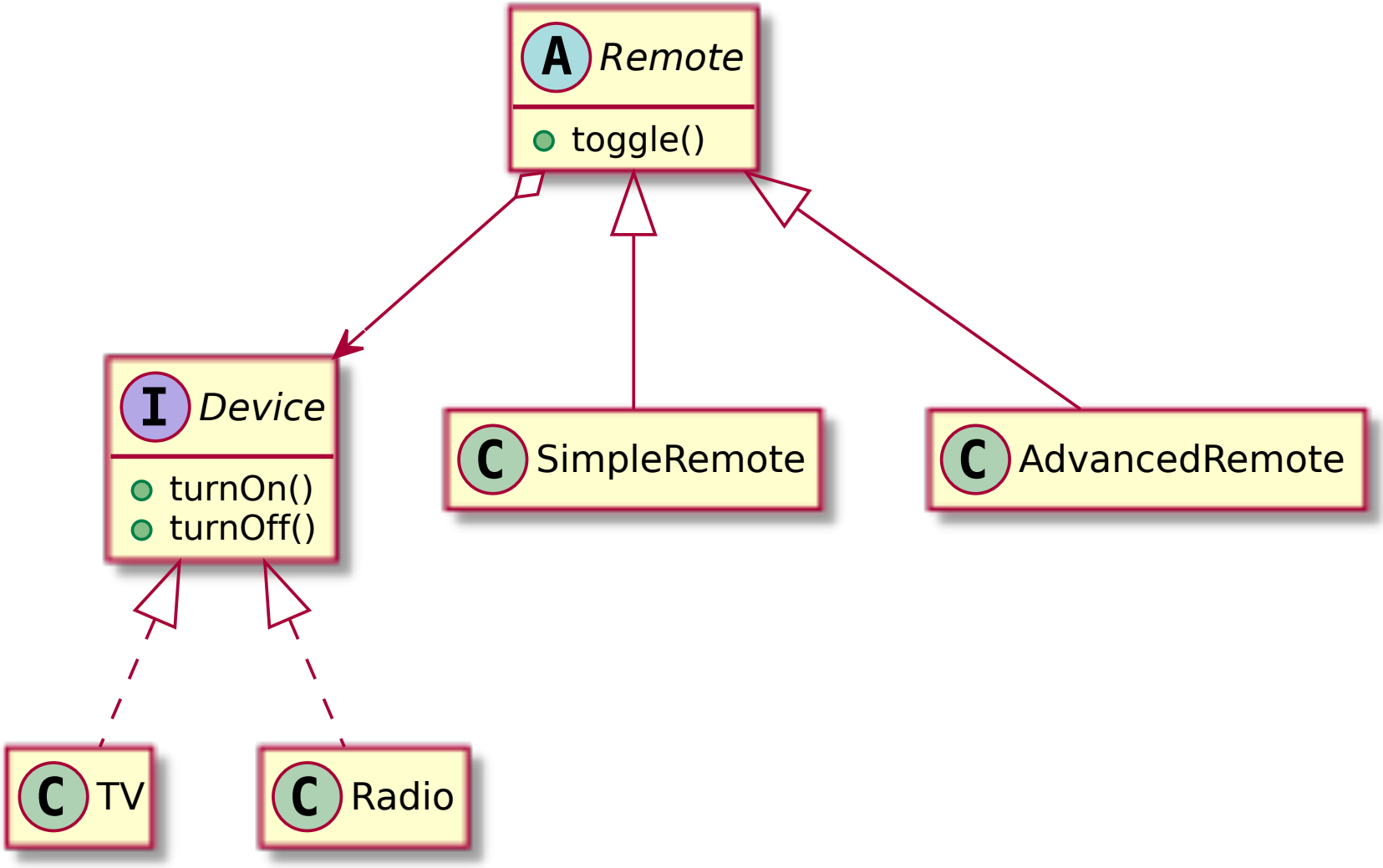
Problem

- You want to build a universal remote-control system that works with different devices like TVs, Radios, and projectors.
- If you directly subclass for every combination (e.g., AdvancedTVRemote, BasicRadioRemote), you get class explosion.

Solution

- The Bridge pattern separates the abstraction (Remote) from the implementation (Device), allowing each to evolve independently and avoiding unnecessary subclasses.

Bridge Pattern — concrete example diagram



Bridge Pattern — concrete example code

Source file

```
// Implementor
interface Device {
    void turnOn();
    void turnOff();
}

// Concrete Implementors
class TV implements Device {
    public void turnOn() { System.out.println("TV ON"); }
    public void turnOff() { System.out.println("TV OFF"); }
}

// Abstraction
abstract class Remote {
    protected Device device;
    Remote(Device d) { this.device = d; }
    abstract void toggle();
```

Bridge Pattern — benefits and drawbacks

Benefits

- Decouples abstraction from implementation
  - They evolve independently
- Avoids class explosion
  - You don't need a subclass for every combination
- Improves extensibility
  - Add new abstractions or new implementations without touching the other side
- Follows the Open/Closed Principle
  - Add features without modifying existing code

Drawbacks

- Architecture more complex than necessary for simple cases.
- Adds layers of indirection you may not always need.

Bridge Pattern Exercise

Task

You are building a drawing tool with two dimensions of variability:

- Shapes (Circle, Rectangle, Line);
- Rendering Methods (OpenGL, SVG).

Explain how to apply the Bridge pattern so all shapes can be rendered with any rendering method without class explosion.

Goal

Identify separate dimensions of change and design a usable abstraction/ implementation split.

Adapter Pattern

Type

Structural pattern

Intent

Convert the interface of one class into another interface clients expect.

Problem Solved

How to make incompatible interfaces work together without changing existing code?

Solution

Create an `Adapter` that wraps an existing class and exposes the desired target interface.

Adapter Pattern — key components

Target	The interface your code expects and uses.
Adaptee	The existing class with an incompatible interface.
Adapter	The wrapper that Implements the Target interface
	Internally calls the Adaptee method(s), translating data or behavior

Adapter Pattern — concrete example scenario

Problem

- Your app expects a *MediaPlayer* interface with a `play()` method,
  - but your *LegacyPlayer* only supports `playMp3()`.
- You cannot modify the legacy system
  - but you must integrate it

Adapter Pattern — concrete example scenario

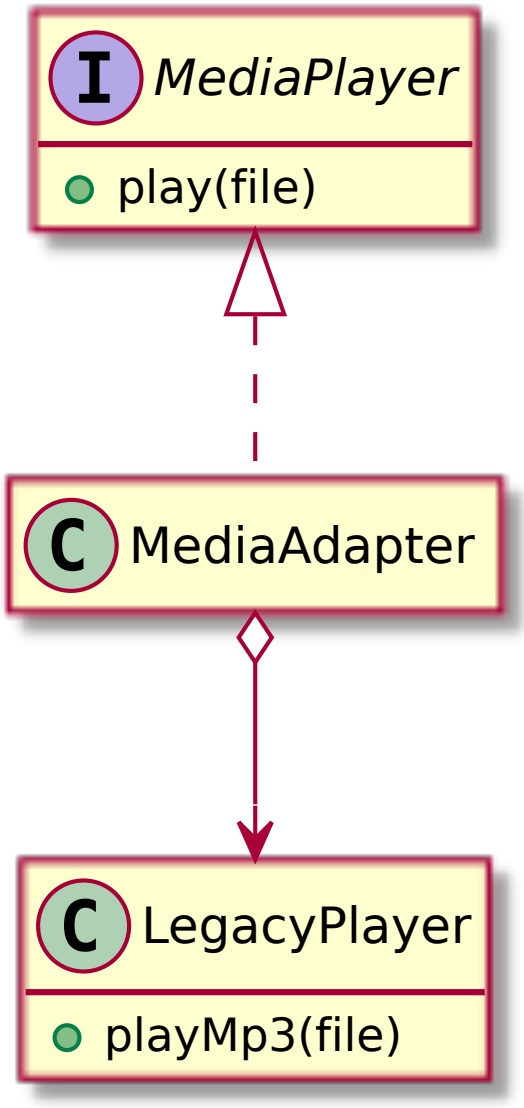
Problem

- Your app expects a *MediaPlayer* interface with a `play()` method,
  - but your *LegacyPlayer* only supports `playMp3()`.
- You cannot modify the legacy system
  - but you must integrate it

Solution

The Adapter pattern wraps the incompatible class and exposes the interface the client expects, allowing the two systems to work together seamlessly.

Adapter Pattern — UML class diagram



Adapter Pattern — concrete example code

Source file

```
// Target interface
interface MediaPlayer {
    void play(String file);
}

// Adaptee
class LegacyPlayer {
    void playMp3(String filename) {
        System.out.println("Playing MP3: " + filename);
    }
}

// Adapter
class MediaAdapter implements MediaPlayer {
    private LegacyPlayer legacy = new LegacyPlayer();
    public void play(String file) { legacy.playMp3(file); }
}
```

Adapter Pattern — benefits and drawbacks

Benefits

- Reuses existing code without modification
- Decouples client code from concrete implementations
- Makes third-party, legacy, or low-level APIs easier to use
- Improves testability by exposing a clean interface

Drawbacks

- Adds an extra layer of indirection
- Can proliferate adapters if many mismatched types exist
- If misused, may hide architectural inconsistencies

Adapter Pattern Exercise

Task

- A new external weather service provides data in a completely different format from your current WeatherData interface.
- Design an Adapter that lets your system continue using WeatherData
  - while seamlessly integrating the new provider.

Decorator Pattern

Type

Structural pattern

Intent

Attach additional responsibilities to an object dynamically without modifying its class.

Problem Solved

How to add flexible, combinable features to objects without subclass explosion?

Solution

Wrap objects with decorator classes that implement the same interface and add behavior before/after delegating calls.

Decorator Pattern — key components

Component (interface or abstract class) Defines the main operations.

Concrete Component The core object you want to decorate.

Decorator (abstract class) Wraps a component and delegates calls to it.

Concrete Decorators Add additional behavior before or after delegating to the wrapped object.

Decorator Pattern — concrete example scenario

Problem

- A beverage ordering system needs to allow customers to add ingredients
  - like milk, sugar, or whipped cream to drinks.
- Creating a subclass for every combination (CoffeeWithMilkAndSugar, etc.) would cause a combinational explosion.

Decorator Pattern — concrete example scenario

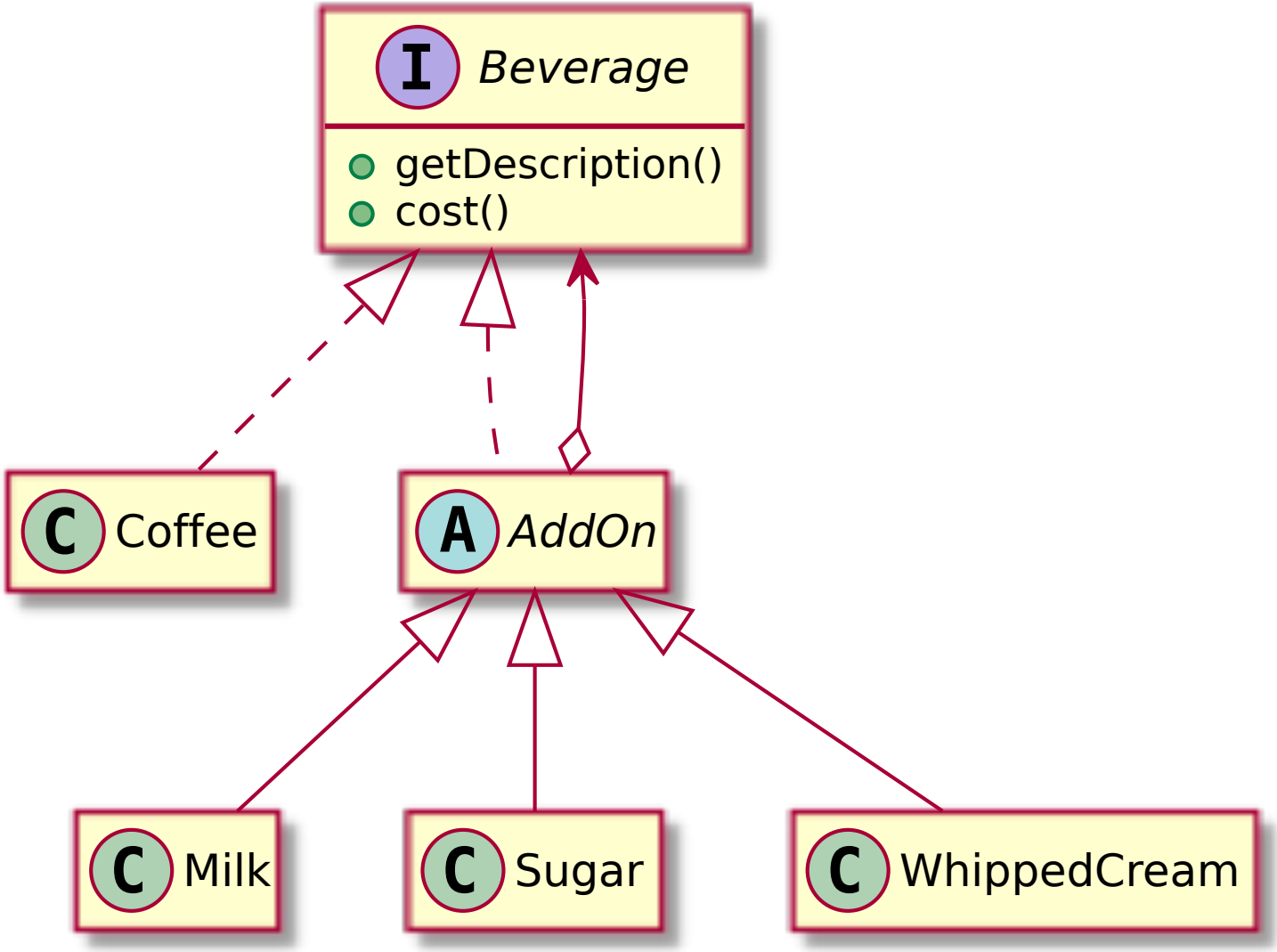
Problem

- A beverage ordering system needs to allow customers to add ingredients
  - like milk, sugar, or whipped cream to drinks.
- Creating a subclass for every combination (CoffeeWithMilkAndSugar, etc.) would cause a combinational explosion.

Solution

- The Decorator pattern lets you dynamically wrap beverages with add-ons,
  - mixing and matching features without modifying existing code.

Decorator Pattern — UML class diagram



Decorator Pattern — concrete example code

Source file

```
// Component
interface Beverage {
    String getDescription();
    double cost();
}

// Concrete Component
class Coffee implements Beverage {
    public String getDescription() { return "Coffee"; }
    public double cost() { return 2.0; }
}

// Decorator
abstract class AddOn implements Beverage {
    protected Beverage beverage;
    AddOn(Beverage b) { beverage = b; }
}
```

Decorator Pattern — benefits and drawbacks

Benefits

- Add responsibilities at runtime
- Combine decorators in flexible ways
- Open/Closed Principle
  - add behavior without modifying existing code
- Avoids deep subclass hierarchies

Drawbacks

- Lots of small classes
- Behavior can become harder to trace after many layers
- Debugging can be trickier

Decorator Pattern Exercise

Task

- Consider an online text editor where users can apply features such as:
  - Bold, Italic, Underline, Syntax Highlighting.
- Describe how you could use Decorators to apply multiple text styles to a plain Text object at runtime
  - without creating many subclasses.

Proxy Pattern

Type

Structural pattern

Intent

Provide a surrogate or placeholder for another object to control access to it.

Problem Solved

How to manage access to a resource-heavy or remote object (e.g., lazy loading, caching, security)?

Solution

Implement a proxy that implements the same interface as the real subject and controls access before forwarding requests.

Proxy Pattern - key components

Subject (interface) Defines the operations available to both Proxy and RealSubject.

RealSubject The actual object that does the real work.

Proxy Implements the same interface but performs extra steps before/after delegating to RealSubject: *Access control*, *Lazy initialization*, *Logging / auditing*, *Remote communication*, *Caching*

Clients cannot tell whether they're talking to the proxy or the real subject.

Proxy Pattern — concrete example scenario

Problem

- Accessing a real database connection is slow and expensive.
- However, you only need the actual connection when a query is executed.

Proxy Pattern — concrete example scenario

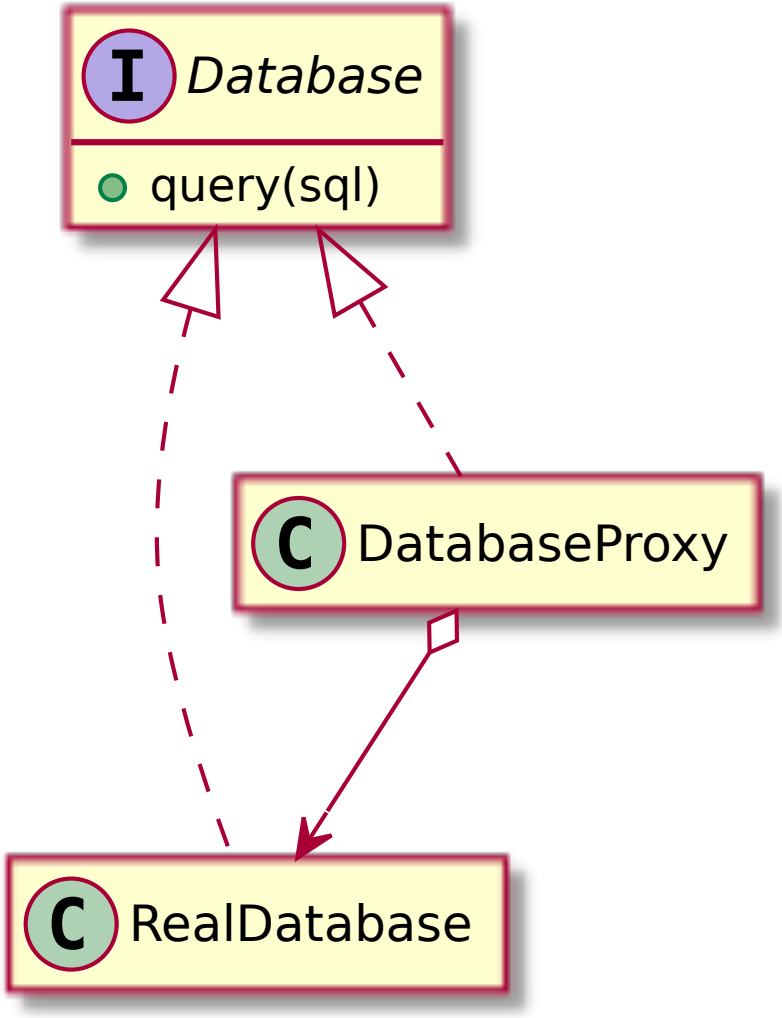
Problem

- Accessing a real database connection is slow and expensive.
- However, you only need the actual connection when a query is executed.

Solution

- The Proxy pattern allows you to create a DatabaseProxy that
  - delays the creation of the RealDatabase until it's truly needed (lazy loading),
  - controlling access and improving performance.

Proxy Pattern — UML class diagram



Proxy Pattern — concrete example code

Source file

```
// Subject
interface Database {
    void query(String sql);
}

// Real Subject
class RealDatabase implements Database {
    public RealDatabase() {
        System.out.println("Connecting to database...");
    }
    public void query(String sql) {
        System.out.println("Executing query: " + sql);
    }
}

// Proxy
class DatabaseProxy implements Database {
```

Proxy Pattern — benefits and drawbacks

Benefits

- Add functionality without changing the real object
- Control expensive or sensitive operations
- Transparent to clients—same interface
- Can optimize performance (caching, lazy loading)
- Supports distributed systems (remote proxy)

Drawbacks

- Adds complexity
- Indirection may hurt performance if misused
- Can hide what's really happening (e.g., network calls look local)

Proxy Pattern Exercise

Task

- Your application accesses remote image files stored on a cloud server.
- Design a Proxy that loads the actual image only when it is displayed
  (for example, when scrolling in a gallery).
- Describe the responsibilities of both the proxy and the real image.

Goal

Identify opportunities for lazy loading, access control, and indirection.

Composite Pattern

Type

Structural pattern

Intent

Compose objects into tree structures to represent part–whole hierarchies.

Problem Solved

How to treat individual objects and groups of objects uniformly?

Solution

Define a common component interface

- Leaf objects implement base behavior;
- Composite objects store children and delegate operations recursively.

Composite Pattern — key components

Component (common interface) Defines operations available to both leaf and composite objects.

Leaf A simple object with no children.

Composite A container object that can hold components (both leaves and other composites).

Client Interacts with components uniformly, without caring if they're leaves or composites.

Composite Pattern — concrete example scenario

Problem

- You want to represent a hierarchical file system where folders can contain both files and other folders.
- Clients should treat individual files and folder groups uniformly (e.g., calling show() on either should work).

Composite Pattern — concrete example scenario

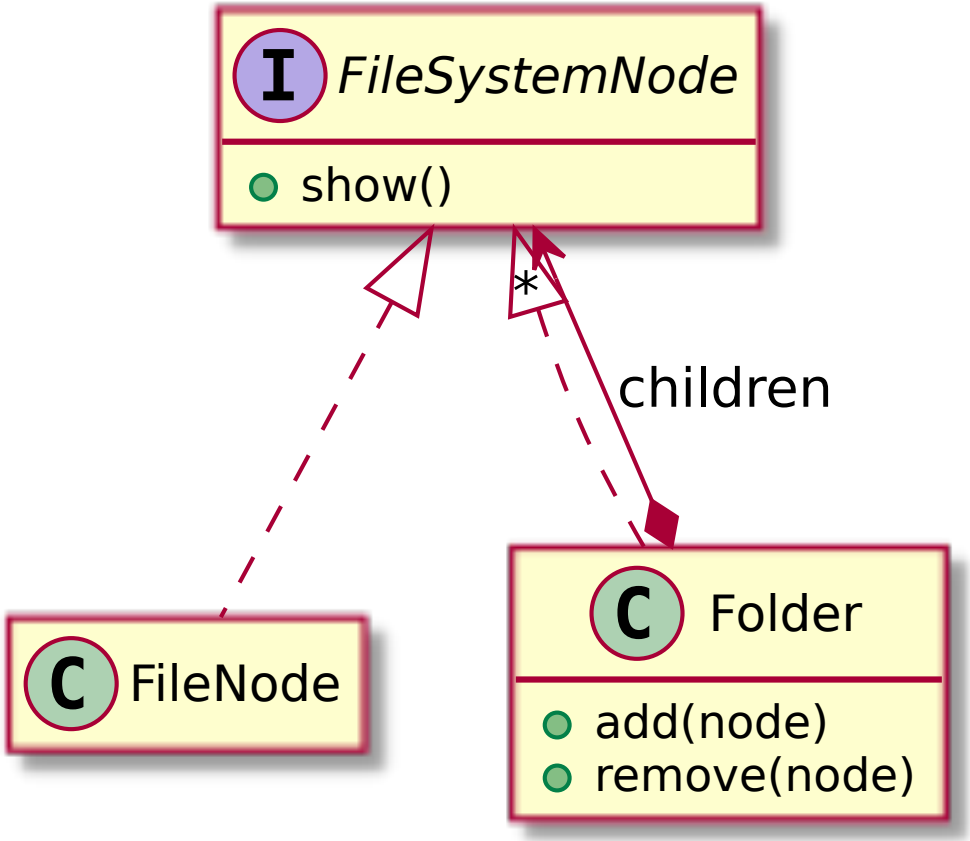
Problem

- You want to represent a hierarchical file system where folders can contain both files and other folders.
- Clients should treat individual files and folder groups uniformly (e.g., calling show() on either should work).

Solution

- The Composite pattern allows you to build tree structures in which both leaf nodes (files) and composite nodes (folders) share the same interface.

Composite Pattern — UML class diagram



Composite Pattern — concrete example code

Source file

```
// Component
interface FileSystemNode {
    void show();
}

// Leaf
class FileNode implements FileSystemNode {
    private String name;
    FileNode(String name) { this.name = name; }
    public void show() { System.out.println("File: " + name
}

// Composite
class Folder implements FileSystemNode {
    private String name;
    private java.util.List<FileSystemNode> children = new
```

Composite Pattern Exercise

Task

- You are modeling hierarchical UI components:
  - Buttons, Labels, TextFields, and Containers that hold other components.
- Explain how the Composite pattern allows you to treat every UI element uniformly (e.g., calling render() or resize()).
- Sketch the component interface and the composite structure.

Wrap-Up — Key Insights

Visitor	Add operations to existing class hierarchies *without modifying them* by externalizing behavior.
Mediator	Reduce tangled, many-to-many communication by centralizing interaction logic inside a mediator object.
Bridge	Separate abstraction from implementation; avoid class explosion and allow sides to vary independently.
Adapter	Make incompatible interfaces work together: wrap one interface to match expectations of another.
Decorator	Dynamically add responsibilities or behavior to objects without subclassing or modifying original classes.
Proxy	Control/enhance access to an object (lazy loading, security, caching) without changing the real object.
Composite	Treat individual items and groups uniformly through a shared component interface.

Wrap-up diagram