**Design Patterns**

● Design patterns are the foundation of good software design. They help you solve

● A design pattern is a general, reusable solution to a common problem within a

specific context in software design.

● Patterns are blueprints that can be applied in multiple situations, but they are

not direct pieces of code.

● They focus on object interactions, system architecture, and how

classes/objects communicate.

Each pattern describes a problem which occurs over and over

again in our environment, and then describes the core of the

solution to that problem, in such a way that you can use this

solution a million times over, without ever doing it the same way

twice

Benefits  
● Code Reusability: Patterns provide proven solutions that can be applied in

multiple projects.

● Maintainability: Encourages clean, understandable, and structured code,

which is easier to maintain and extend.

● Communication: Design patterns act as a common language between

developers, improving communication and understanding.

● Scalability: Design patterns help in structuring code to support future scaling

with fewer refactors.

● Efficiency: Avoids reinventing the wheel by using well-tested and widely

accepted solutions.

Types of patterns  
**Behavioral Patterns:** Focus on communication between objects and responsibilities.

Examples: *Observer Pattern, Strategy Pattern, Command Pattern*

*● Behavioral Example: A notification system using the Observer*

*Pattern, where various modules subscribe to system events.*

**Creational Patterns:** Deal with object creation mechanisms, trying to create objects

in a manner suitable to the situation.

Examples: *Factory Pattern, Singleton Pattern, Builder Pattern*

*● Creational Example: A factory that produces different types of*

*documents (Word, PDF, Excel) depending on the input parameters.*

**Structural Patterns:** Deal with the composition of objects or classes to form larger

structures.

Examples: *Adapter Pattern, Composite Pattern, Decorator Pattern ● Structural Example: Using the Adapter Pattern to integrate a new API into an existing codebase.*

**Behavioral Pattern**

* Behavioral patterns focus on how objects communicate and interact, managing the flow of information between entities.
* They simplify complex control flow by defining clear communication and behavior among objects.
* They provide solutions for managing object relationships and communication protocols to promote loose coupling and enhance flexibility.

Common Applications:

● Coordinating interactions between objects.

● Managing state transitions and communication efficiently.

 **Chain of Responsibility**

* Passes a request along a chain of handlers until one of them handles it.
* Example: Customer support system → Level 1 → Level 2 → Level 3.

 **Command**

* Encapsulates a request as an object, allowing undo/redo and queuing.
* Example: Remote control buttons → each button press is a command object.

 **Interpreter**

* Defines a grammar and uses it to interpret sentences.
* Example: SQL or Regex engines that parse and evaluate expressions.

 **Iterator**

* Provides a way to access elements of a collection sequentially without exposing its underlying representation.
* Example: Iterator in Java to loop through ArrayList.

 **Mediator**

* Defines an object that centralizes communication between many objects.
* Example: Air Traffic Control tower → coordinates all planes instead of planes talking directly.

 **Memento**

* Captures and restores an object’s internal state without exposing details.
* Example: “Undo” in text editors.

 **Observer**

* Defines a one-to-many dependency so when one object changes state, all dependents are notified.
* Example: News agency notifies multiple subscribers.

 **State**

* Allows an object to change its behavior when its internal state changes.
* Example: A traffic light switching between Red, Yellow, Green states.

 **Strategy**

* Defines a family of algorithms and makes them interchangeable.
* Example: Payment method selection → Credit Card, PayPal, UPI.

 **Template Method**

* Defines the skeleton of an algorithm, letting subclasses fill in specific steps.
* Example: Cooking recipe → fixed steps (boil, cook, serve), but details (ingredients) vary.

 **Visitor**

* Separates an algorithm from the object structure it operates on.
* Example: Tax calculator visiting different item types (Food, Electronics, Clothing).

### ****Quick Analogy****

Think of behavioral patterns as **“rules of engagement”** for objects:

* Chain of Responsibility = “Passing the buck”
* Command = “Command queue”
* Interpreter = “Language translator”
* Iterator = “Playlist navigator”
* Mediator = “Middleman manager”
* Memento = “Time machine (Undo)”
* Observer = “News subscription”
* State = “Mood swings”
* Strategy = “Different game plans”
* Template Method = “Blueprint with steps”
* Visitor = “Guest performing actions in different houses”

**1) Memento Pattern**

Problem: How to provide undo/redo functionality or state restoration without exposing the object's internal state and breaking encapsulation.

Solution: The Memento Pattern captures the internal state of an object in a

memento, allowing the object to restore its state later on without revealing internal details.

Momento Pattern structure -   
● Components:

○ Originator: The object whose state needs to be saved and restored.

○ Memento: Captures and stores the internal state of the originator.

○ Caretaker: Manages and stores the mementos, without modifying them.

Applications -

* Undo/Redo in Applications: Commonly used in text editors, drawing applications, or
* any system that requires history management.
* State Restoration: Used in scenarios where you need to periodically save system
* states (e.g., games, data recovery) and allow users to return to previous states.

Use Cases:

● Games: Saving the game state for load/reload functionality.

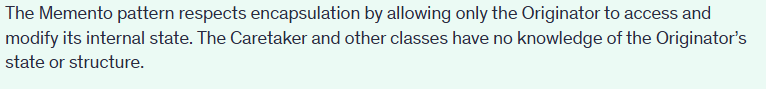
● Document Editors: Undo/redo functionality to navigate through document

changes.

CODE : <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/MementoPattern/all%20in%20one>

Using Memento Design Pattern, we separate concerns:

1. TextEditor (Originator) → Only knows how to save/restore its own state.
2. TextMemento → A simple snapshot object, no one can modify it.
3. History (Caretaker) → Manages undo/redo without knowing the internals of TextEditor.



Why this helps:

* Encapsulation preserved → TextEditor’s internals (StringBuilder, styles, cursor, etc.) are hidden.
* Reusability → History logic works for any object that supports Memento, not just text editor.
* Flexibility → You can change the editor’s internal representation tomorrow (say from StringBuilder to a rich document model), and undo/redo will still work — because Caretaker only stores/uses Memento, not internals.
* Cleaner design → Each class does one job.

Using a Memento object gives you a few benefits:

1. Encapsulation (Core idea of Memento pattern)

The TextEditor is the only class that knows what its “state” means.

The History doesn't need to know it’s a String. It just stores and retrieves TextMemento objects.

This makes your code more maintainable and less tightly coupled.

2. Flexibility for future changes

Today the state is just a String. Tomorrow it might include formatting, cursor position, undo stacks, etc. If you return a TextMemento, the state structure can change without affecting the History.

3. Clear responsibilities

TextEditor: responsible for its own state and how to save/restore it.

TextMemento: just holds a snapshot of the state.

History: just manages a list of previous states.

This separation follows SOLID principles, especially Single Responsibility.

Real Trouble Without Memento

Imagine your text editor evolves:

* You add bold/italic formatting,
* Then cursor position,
* Then syntax highlighting.

Without Memento:

* You’d expose and manage all these states manually in history.
* Undo logic becomes tightly coupled to editor internals.
* Any internal refactor breaks undo.

With Memento:

* Just save/restore a snapshot (TextMemento).
* Caretaker doesn’t care what’s inside — it’s still just “a state”.
* Undo keeps working no matter how complex the editor gets.

So the Memento pattern is about protecting encapsulation while still allowing save/restore functionality.Would you like me to show you a before-and-after comparison with formatting + cursor so you can see how Memento keeps things clean?

**2) Observer Pattern**

**Problem**: There is a need to notify multiple objects about a change in state without

tightly coupling them.

**Solution**: The Observer Pattern defines a **one-to-many dependency**. When one

object (the **Subject**) changes its state, all its dependents (the **Observers**) are notified

and updated automatically. This pattern is widely used for implementing distributed

event-handling systems, also known as the **publish-subscribe** pattern

OBSERVER PATTERN STRUCTURE:  
**Subject**: Maintains a list of observers and notifies them of changes.

**Observers**: Receive updates from the subject.

Benefits:  
● **Loose Coupling**: The subject (e.g., WeatherStation) doesn’t need to know

about the specific observers. It just notifies them.

● **Scalability**: New observers (e.g., new display devices) can easily be added

without changing the subject.

● **Flexibility**: Observers can be dynamically added or removed at runtime.  
  
Usecases  
● **Event Listeners**: GUI frameworks often use the Observer Pattern to

implement event listeners for handling button clicks, input changes, etc.

● **Stock Price Monitoring**: When a stock price changes, multiple subscribers

(like investors or systems) can be notified of the change.

● **News Publishing Systems**: News articles are published (subject), and

subscribers (users) are notified whenever a new article is available.

● **Social Media Notifications**: Users can subscribe to updates from specific

accounts, and when an account posts (subject), all followers (observers) are

notified.

● **Logging Systems**: Different logging handlers can observe events and log

them as needed, such as to the console, file, or remote server.  
  
CODE: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/ObserverPattern/ObserverPatternExample.java>

**3) Strategy Pattern**

**Motivations**Let's consider a simple payment system where users can pay using different

methods like credit cards or PayPal. Without the Strategy Pattern, you might use

if-else conditions to handle the different payment methods, leading to less

maintainable and flexible code.if in future you want to add a new payment option you need to modify the payment service where these if else are present which is violation of Open and Closed Principle.

**Problems in Code**

● The PaymentService class has multiple responsibilities (deciding the

payment type and processing it).

● Adding a new payment method requires modifying the PaymentService

class.

● The use of if-else conditions can make the code harder to maintain as more

payment types are added.

**With the Strategy Pattern, the logic for each payment type is encapsulated in separate strategy**

**classes, and the PaymentService (context class) delegates the task of payment processing to one**

**of these strategies at runtime.**

**Problem**: Hardcoded algorithms in classes lead to:

○ Code **duplication**.

○ Increased **maintenance complexity** when switching between algorithms.

○ **Violation of Open/Closed Principle**: Modifications are required every

time a new algorithm is introduced.

**Solution**: The Strategy Pattern decouples the algorithm implementation from the

client, allowing easy **switching of algorithms** without altering the client code.

Strategy Pattern Structure:

**Context**: The client class that uses a strategy to perform an operation.

**Strategy Interface**: Defines the operations that all concrete strategies must

implement.

**Concrete Strategy**: Implements the actual algorithms, interchangeable based on the

context.

USECASE:  
**When to Use the Strategy Pattern:**

● When multiple algorithms need to be used interchangeably.

● To avoid conditional statements (if-else or switch-case) in the client

code.

● When a class has multiple behaviors, which can vary independently.

CODE:<https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/StrategyPattern/StrategyPattern.java>

**4) Command Pattern**

The **Command Design Pattern** is a **behavioral design pattern** that turns a **request** into a **standalone object** that contains all the information about the request. This allows you to **parameterize methods with different requests**, **queue or log requests**, and **support undoable operations**.

**Structure**:

● **Command**: Interface for executing operations.

● **Invoker**: Sends the command.

● **Receiver**: Performs the operation.

**Basic Idea**

* You encapsulate a request as an object.
* You decouple the object that invokes the operation from the one that knows how to perform it.

**Participants**

1. **Command** – declares an interface for executing operations.
2. **ConcreteCommand** – defines a binding between a Receiver and an action.
3. **Receiver** – knows how to perform the operations.
4. **Invoker** – asks the command to carry out the request.
5. **Client** – creates and configures ConcreteCommand objects.

**Real-Life Analogy**

Think of a **TV remote control**:

* The remote is the **Invoker**.
* Each button press is a **Command**.
* The **TV** is the **Receiver**.
* You can change which command is assigned to which button (decoupling logic).

**Java Example**

Let’s implement a simple **Remote Control** that turns a light **on** and **off**.

CODE:<https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/CommandPattern/allInOne>

**Advantages**

● **Decoupling of Invoker and Receiver**: The button (invoker) doesn't know the

details of the TextEditor (receiver), making the system more flexible and

reusable.

● **Command History and Undo**: Commands can be logged for undo/redo

functionality.

● **Task Queuing**: Commands can be stored in a queue and executed later,

making it useful for task scheduling.

● **Extensibility**: New commands can be added easily without modifying existing

code. For example, adding a ChangeColorCommand only requires creating a new command class.

* Decouples the sender and receiver.
* Makes it easy to add new commands.
* Supports **undo**, **redo**, **logging**, **macro commands**.

**Common Use Cases**

* GUI buttons and menu items.
* Task scheduling.
* Transaction management (undo/redo).
* Macro recording in applications.

Let me know if you want an **undo** example or want it applied to a **real-world system** like a text editor or home automation.

**5) Template Method Pattern**

Consider a scenario where you have different data parsers (e.g., CSV, XML,

and JSON). Each parser follows the same steps: **open file, parse data, and**

**close file**.

Without the Template Method Pattern, you might end up duplicating the

common steps in each parser class.

Problems in code

● Code duplication: The openFile() and closeFile() methods are

duplicated in both parsers.

● Any changes to the common logic would require changes in every parser,

violating the DRY (Don’t Repeat Yourself) principle.

**Problem**: Different parts of an algorithm may need to vary in subclasses, but the

overall structure should remain consistent.

**Solution**: The Template Method Pattern defines the **skeleton** of an algorithm in a

base class and lets subclasses override specific steps.

**Structure**:

● **Abstract Class**: Defines the algorithm skeleton. (open file closed file will be with implementation and process method will be abstract method that can be implemented in classes which will extend this class)

● **Concrete Subclasses**: Override specific steps of the algorithm.

CODE: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/TemplateMethodPattern/WithTemplatePattern.java>

Benefits  
● **Code Reuse**: Common code is moved to the parent class, promoting reuse

and reducing duplication.

● **Flexibility**: Subclasses can vary certain steps in the algorithm, while keeping

the overall structure intact.

● **Consistency**: Ensures that the high-level structure of the algorithm remains

consistent, even when subclass behavior differs.  
  
**Usecases**  
**UI Frameworks**: Rendering a UI element might follow a fixed set of steps (initialize, draw, finish),

but the details of how each element is drawn are left to subclasses.

**Document Processing**: A framework might define the skeleton for reading, processing, and

saving documents, while specific formats (e.g., Word, PDF) provide their own processing logic.

**Game Development**: A game loop (initialize, update, render) can be defined in a base class, with

specific games implementing their own logic for updating and rendering.

The **Template Method Pattern** is ideal for situations where a common algorithm exists, but some

steps may need to be redefined by subclasses. It helps enforce structure and promotes

reusability, while allowing flexibility where needed.

**Iterator Pattern**

Think of it like a TV remote:

* You don’t need to know how the TV stores channels internally.
* You just use **next/previous buttons** to iterate through them.

Key Components in Java

**Iterator interface**

Defines how to traverse a collection.

Example methods: hasNext(), next().

**Concrete Iterator**

Implements the Iterator interface for a specific collection.

**Aggregate (Collection)**

Defines a method to return an iterator.

**Concrete Aggregate**

A real collection (like BookCollection) that creates the iterator.

**Problems**:

● The client needs to know the internal structure of the collection (array in this

case).

● If we change the collection type (e.g., from an array to a linked list), we would

need to modify the client code.

● It’s harder to implement different traversal strategies.

**Problem**: How to access elements in a collection without exposing its internal

representation.

**Solution**: The Iterator Pattern provides a way to **traverse** a collection without

revealing its underlying structure, offering a uniform interface for traversal.

**Structure**:

● **Iterator**: Interface for traversing a collection.

● **Collection**: Holds the elements and provides an iterator.

Iterator pattern benefits  
1. **Separation of Concerns**: The traversal logic is separated from the collection

itself, allowing you to change one without affecting the other.

2. **Uniform Interface**: The same interface (Iterator) is used to traverse

different types of collections, making the code more flexible.

3. **Simplified Client Code**: The client doesn’t need to know the underlying data

structure, reducing coupling and making the code easier to maintain.

4. **Multiple Traversal Strategies**: You can implement multiple types of iterators

(e.g., forward, backward, filtered) without changing the collection.

Iterator pattern use cases   
  
1. Java Collections Framework:

○ The Java Collections Framework (e.g., ArrayList, HashSet) uses the iterator pattern

to provide a common interface (Iterator) for traversing different types of collections.

2. Database Cursors:

○ In database programming, cursors are used to iterate over result sets. The iterator

pattern can abstract this traversal, making it easier to work with data from a database

without exposing the underlying query mechanism.

3. Tree Traversals:

○ In tree data structures, the iterator pattern can be used to traverse nodes using different

strategies like depth-first or breadth-first, without exposing the tree's internal structure.

4. File Systems:

○ File systems can use the iterator pattern to traverse directories and files without exposing

the internal details of how files and folders are stored.  
  
CODE: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/IteratorPattern/allInOne>

**State Pattern**

The **State Pattern** is a **behavioral design pattern** that allows an object to **change its behavior when its internal state changes**.

* Instead of using **if-else or switch statements** everywhere, the object **delegates the behavior to different state classes**.
* Each state class represents a **particular behavior** of the context.

Think of it like a **traffic light**:

* When it’s green, cars move.
* When it’s red, cars stop.
* The **TrafficLight object** doesn’t need to check colors itself; each **State object** handles the behavior.

## Key Components

1. **Context**
   * Maintains an instance of a State object.
   * Delegates behavior to the current state.
2. **State Interface**
   * Defines the methods that each concrete state must implement.
3. **Concrete States**
   * Implement behavior specific to a particular state.

You are tasked with building a **DirectionService** class for a navigation app. This class calculates the

**estimated time of arrival (ETA)** and provides **directions** between two points. The ETA and direction differ

based on the mode of transportation, which can be one of the following:

● **Walking**

● **Cycling**

● **Car**

● **Train**

**Problems if we use if else if   
Tight Coupling and Complex Conditional Logic**:

● The DirectionService likely uses **conditional statements** (if-else or switch-case) based

on transportation mode enums to determine how to calculate ETA and provide directions.

● As the number of transportation modes increases, the conditional logic becomes **more complex**

**and harder to maintain**.

**Violation of the Open/Closed Principle**:

● **Adding new transportation modes** (e.g., Airplane, Boat) requires modifying the existing

DirectionService class, which goes against the **Open/Closed Principle** (classes should be

open for extension but closed for modification).

CODE: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/StatePattern/Solution/allInOne>

**Mediator Pattern**

## 💡 What is the Mediator Design Pattern?

The **Mediator Pattern** is a **behavioral design pattern** that **reduces the complexity of communication between multiple objects (colleagues) by introducing a mediator object**.

* Instead of objects referring to each other directly, they **communicate through a mediator**.
* This reduces **tight coupling** and makes the system easier to maintain.

Think of it like an **air traffic controller**:

* Planes (colleagues) don’t talk directly to each other.
* The **controller (mediator)** coordinates takeoffs and landings.

## 🔑 Key Components

1. **Mediator Interface**
   * Declares methods for communication between colleagues.
2. **Concrete Mediator**
   * Implements the communication logic.
3. **Colleague Classes**
   * Know the mediator and communicate **only through it**.

## 📘 Example: Chat Room

**Problem**: Objects in a system need to communicate, but direct communication leads to tight

coupling and complexity.

**Solution**: The Mediator Pattern introduces a **mediator** object that handles all

communication between objects, reducing direct dependencies and coupling.

In our chat app, by introducing a **Mediator** object, we will decouple the users from knowing

about each other directly. The **Mediator** handles all communication, and the users

(colleagues) only interact with the **Mediator**. This simplifies the interaction and reduces

dependencies.

Benefits  
● **Reduces Complexity**: The mediator centralizes communication, reducing direct dependencies

between objects.

● **Loose Coupling**: Colleagues only interact with the mediator, making them easier to manage,

extend, and maintain.

● **Single Responsibility**: The mediator handles complex communication logic, allowing colleagues

to focus on their own behavior.

● **Centralized Control**: Changes to communication rules can be made in the mediator without

affecting the colleagues.

**Real world usecases**

**Air Traffic Control:**

Airplanes communicate through a central control tower (mediator) instead of coordinating directly

with each other.

**GUI Component Coordination:**

In GUI applications, multiple UI components may need to interact. For example, when a dropdown

changes, it may trigger updates to text fields, buttons, etc. A mediator can handle this interaction

logic instead of having the components know about each other directly.

CODE:<https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/MediatorPattern/allInOne>

**Structual Patterns**

**Structural Design Patterns**

Structural design patterns are a category of design patterns in software engineering that deal with **how classes and objects are composed to form larger structures**. Their main focus is on **simplifying the relationships** between entities to make the system more flexible and easier to maintain.

They help in:

* Defining **clear relationships** between objects.
* Reducing **complexity** in object composition.
* Promoting **code reusability**.

**Main Structural Design Patterns**

Here are the key ones:

1. **Adapter Pattern**
   * Converts the interface of a class into another interface the client expects.
   * Helps classes work together that couldn’t otherwise because of incompatible interfaces.
   * Example: A power plug adapter that lets a 2-pin plug fit into a 3-pin socket.
2. **Bridge Pattern**
   * Separates abstraction from its implementation, so both can evolve independently.
   * Example: Remote control (abstraction) works with different types of devices like TV, Radio (implementation).
3. **Composite Pattern**
   * Treats individual objects and groups of objects uniformly.
   * Example: A tree structure like a folder containing files and subfolders.
4. **Decorator Pattern**
   * Dynamically adds new behavior to objects without modifying their code.
   * Example: Adding extra features to a car (like sunroof, sports exhaust) without changing the car’s base class.
5. **Facade Pattern**
   * Provides a simplified interface to a larger and more complex system.
   * Example: Using a single “HomeTheaterFacade” class to control multiple subsystems like DVD player, projector, speakers.
6. **Flyweight Pattern**
   * Reduces memory usage by sharing as much data as possible with similar objects.
   * Example: A text editor that reuses character objects instead of creating new ones for every occurrence.
7. **Proxy Pattern**
   * Provides a placeholder or substitute for another object to control access.
   * Example: Virtual proxy for loading images only when needed, or a security proxy that checks access before invoking methods.

### ****Quick Analogy****

Think of structural patterns as **architectural blueprints** for software:

* Adapter = “Translator”
* Bridge = “Connector”
* Composite = “Tree structure”
* Decorator = “Add-ons”
* Facade = “Front desk”
* Flyweight = “Shared resource pool”
* Proxy = “Middleman”

**Adapter Pattern**

### ****Definition****

The **Adapter Pattern** is used to **bridge the gap between two incompatible interfaces**.  
It lets classes work together that otherwise couldn’t, because their interfaces don’t match.

### ****Real-World Analogy****

* Think about a **power plug adapter**:
  + Your laptop has a **3-pin charger**.
  + The wall socket in another country may only support **2-pin**.
  + An **adapter** converts the **3-pin plug** into a **2-pin** so you can still use your laptop without changing either device.

### ****When to Use****

* When you want to use an existing class, but its interface does not match the one you need.
* To reuse legacy code without modifying it.
* To integrate third-party libraries into your application.

### ****Types of Adapter****

1. **Class Adapter (using inheritance)**
   * Adapter extends the existing class and implements the target interface.
   * Works only in single-inheritance languages like Java.
2. **Object Adapter (using composition)**
   * Adapter has a reference to the adaptee and delegates calls.
   * More flexible and preferred in most cases.

// Target interface

interface MediaPlayer {

void play(String filename);

}

// Adaptee (existing incompatible class)

class VlcPlayer {

public void playVlcFile(String filename) {

System.out.println("Playing VLC file: " + filename);

}

}

// Adapter class (bridges MediaPlayer and VlcPlayer)

class MediaAdapter implements MediaPlayer {

private VlcPlayer vlcPlayer;

public MediaAdapter(VlcPlayer vlcPlayer) {

this.vlcPlayer = vlcPlayer;

}

@Override

public void play(String filename) {

vlcPlayer.playVlcFile(filename); // Delegate call

}

}

// Client code

public class AdapterPatternDemo {

public static void main(String[] args) {

MediaPlayer player = new MediaAdapter(new VlcPlayer());

player.play("song.vlc");

}

}

### ✅ ****Benefits****

* Reuses existing classes without modification.
* Promotes flexibility (works with third-party libraries).
* Decouples client from concrete implementations.

👉 The **gap** (incompatibility) is:

* The client expects to call play(String filename) (defined by MediaPlayer).
* But the VlcPlayer class provides only playVlcFile(String filename).
* So, without an adapter, the client **cannot directly use** VlcPlayer where a MediaPlayer is required.

The **Adapter** (MediaAdapter) **fills the gap**:

### ✅ Why This is an Adapter

* Client knows **only** about MediaPlayer.
* VlcPlayer is **incompatible** because it doesn’t implement MediaPlayer and has a different method.
* MediaAdapter acts as a **bridge**: it **converts the client’s expected interface** into one that the VlcPlayer understands.

**USECASES**

**Adapters in Software Frameworks**: In GUI frameworks, adapters are used to

convert legacy code into newer formats.

**Adapter in Java I/O**: In Java, InputStreamReader works as an adapter to

convert InputStream (byte-based) to Reader (character-based).

**Adapter in APIs**: When integrating external libraries, you often need adapters

to convert data formats or APIs to match your system's requirements.

**Decorator Pattern**

**📌 What is the Decorator Pattern?**

* **Type**: Structural Design Pattern
* **Purpose**: It allows adding **new behaviors or responsibilities** to an object **dynamically** without modifying its original class.
* You “wrap” an object with another object (the decorator) that provides extra functionality.

Think of it like:  
👉 You have a plain coffee ☕.  
👉 You can add **milk**, **sugar**, or **whipped cream** without changing the original coffee class.  
👉 Each addition "decorates" the coffee.

**📌 Structure**

1. **Component (interface/abstract class)**
   * Defines the base behavior.
2. **ConcreteComponent (original class)**
   * The object that we want to decorate.
3. **Decorator (abstract class)**
   * Implements the component interface and holds a reference to a component.
4. **ConcreteDecorator (extra features)**
   * Adds new functionality while still calling the wrapped component.

Code: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/DecoratorPattern/allInOne>

**Proxy Pattern**

**📌 What is the Proxy Pattern?**

* **Type**: Structural Design Pattern
* **Purpose**: Provides a **substitute or placeholder** for another object to control access to it.
* The proxy object “stands in” for the real object and can add:
  + Access control
  + Lazy initialization (load on demand)
  + Logging, caching, security checks, etc.

Think of it like:  
👉 You want to meet a celebrity 🎬.  
👉 Instead of meeting them directly, you must go through their **manager/assistant** who controls access.  
👉 The assistant is the **proxy**.

**📌 Structure**

1. **Subject (interface/abstract class)**
   * Common interface for RealSubject and Proxy.
2. **RealSubject**
   * The actual object that performs the real work.
3. **Proxy**
   * Holds a reference to RealSubject and controls access to it.

**📌 Example in Java**

Let’s make a **proxy for an image viewer** (common example):

CODE : <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/ProxyPattern/allInOne>

**Composite Pattern**

**📌 What is the Composite Pattern?**

* **Type**: Structural Design Pattern
* **Purpose**: It lets you **compose objects into tree structures** to represent part-whole hierarchies.
* It allows clients to **treat individual objects and groups of objects uniformly**.

Think of it like:  
👉 You have a **folder** that contains files and subfolders.  
👉 Each subfolder can again contain files or other folders.  
👉 You can **treat both files and folders the same way** (e.g., open(), delete(), etc.).

**📌 Structure**

1. **Component (interface/abstract class)**
   * Declares operations that can be performed (common for leaf and composite).
2. **Leaf**
   * Represents simple elements (e.g., file).
3. **Composite**
   * Represents complex elements (e.g., folder), which can hold other components (files or folders).

**📌 When to Use Composite Pattern?**

✅ When you have **hierarchical tree structures** (e.g., file systems, menus, organizations).  
✅ When you want clients to treat **single objects and groups of objects the same way**.  
✅ To avoid writing separate code for handling “part” vs “whole.”

**📌 Real-World Examples**

* **File Explorer** (files & folders).
* **GUI frameworks** (buttons, panels, frames are treated uniformly).
* **Company structure** (employees vs managers).
* **Spring Beans hierarchy** (ApplicationContext parent-child).

CODEL <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/CompositePattern/allInOne>

**Facade Pattern**

**Example: API Gateway as a Facade**

● In **microservices architectures**, an **API Gateway** acts as a facade, providing

a simple interface to clients while hiding the complexity of multiple

microservices working behind the scenes.

**Problem**: Large, complex subsystems with many classes and methods can be

difficult to use directly.

**Solution**: The Facade Pattern provides a simple, unified interface to a complex

subsystem, making it easier to interact with.

The **Facade Pattern** is a structural design pattern that provides a simplified interface

to a complex system of classes, libraries, or frameworks. Instead of exposing all the

details of the complex system, the facade offers a higher-level interface, making it

easier to interact with the system. The Facade Pattern is particularly useful when

dealing with large systems that contain many interdependent classes, by reducing the

interaction points for the client.

**📌 What is the Facade Pattern?**

* **Type**: Structural Design Pattern
* **Purpose**: Provides a **simplified interface** to a **complex subsystem**.
* It hides the complexity of the system by exposing only what the client really needs.

Think of it like:  
👉 When you check into a hotel 🏨, you don’t deal with housekeeping, kitchen, maintenance separately.  
👉 You just call the **reception desk (facade)**, and they handle the complex interactions behind the scenes.

**📌 Structure**

1. **Subsystem Classes**
   * Complex classes with many methods.
2. **Facade**
   * Provides a simple interface for clients.
3. **Client**
   * Uses the facade instead of calling subsystem classes directly.

CODE: <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/tree/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/FacadePattern>

**Bridge Pattern**

## it’s about **splitting abstraction from implementation** and connecting them with a **reference (composition)**.

## 📌 What is the Bridge Pattern?

* **Type**: Structural Design Pattern
* **Purpose**: Decouples an **abstraction** from its **implementation**, so the two can evolve **independently**.
* Instead of binding a class permanently with its implementation, we “bridge” them with composition.

Think of it like:  
👉 You have a **remote control** (abstraction).  
👉 It can work with different **devices** like TV, Radio, or Projector (implementations).  
👉 The remote doesn’t care what device it controls — both can change independently.

**📌 Structure**

1. **Abstraction**
   * Defines the high-level control logic (e.g., Remote).
2. **Refined Abstraction**
   * A variant of abstraction (e.g., AdvancedRemote).
3. **Implementor (Interface)**
   * Defines the low-level operations (e.g., Device).
4. **Concrete Implementors**
   * Different implementations of the interface (e.g., TV, Radio).

### 🔑 Key Idea:

The **Bridge Pattern** is not about having multiple abstract classes or interfaces —  
👉 it’s about **splitting abstraction from implementation** and connecting them with a **reference (composition)**.

So yes, a **single abstract class** (or interface) on each side is enough:

Code : <https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/blob/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/BridgePattern>

**Flyweight Pattern**

In game development, especially in scenarios like shooting games where numerous

identical bullets are fired, memory and performance can quickly become an issue if

each bullet object stores redundant data. Using the **Flyweight Pattern**, we can

reduce memory overhead by sharing intrinsic (Connected to the essential unique nature) properties of bullets (like appearance)

while maintaining unique extrinsic properties (like position and velocity).  


In this refactor, the Bullet class contains only **extrinsic** data like position and velocity. The

**Intrinsic** (Not connected to the essential unique nature)data like color is stored in a BulletType class, which is shared across all Bullet

objects. This allows us to manage a large number of bullets efficiently.



# 📌 What is the Flyweight Pattern?

* **Type**: Structural Design Pattern
* **Purpose**: **Minimize memory usage** by sharing as much data as possible between similar objects.
* It separates:
  + **Intrinsic state** → shared (doesn’t change, e.g., tree type).
  + **Extrinsic state** → unique per object (changes, e.g., tree position).

👉 Think of it like:  
In a forest with **1,00,000 trees 🌳**, instead of creating 1,00,000 heavy objects,  
you create one **shared “tree type” object** (shape, color, texture) and only store **position** separately.

# 📌 Structure

1. **Flyweight (interface)** → defines the common methods.
2. **Concrete Flyweight** → shared object containing intrinsic state.
3. **Flyweight Factory** → manages and reuses Flyweights.
4. **Client** → uses Flyweights and supplies extrinsic state.

CODE:<https://github.com/AkshayChouguleac1/LowLevelDesignPatterns/tree/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/FacadePattern>