

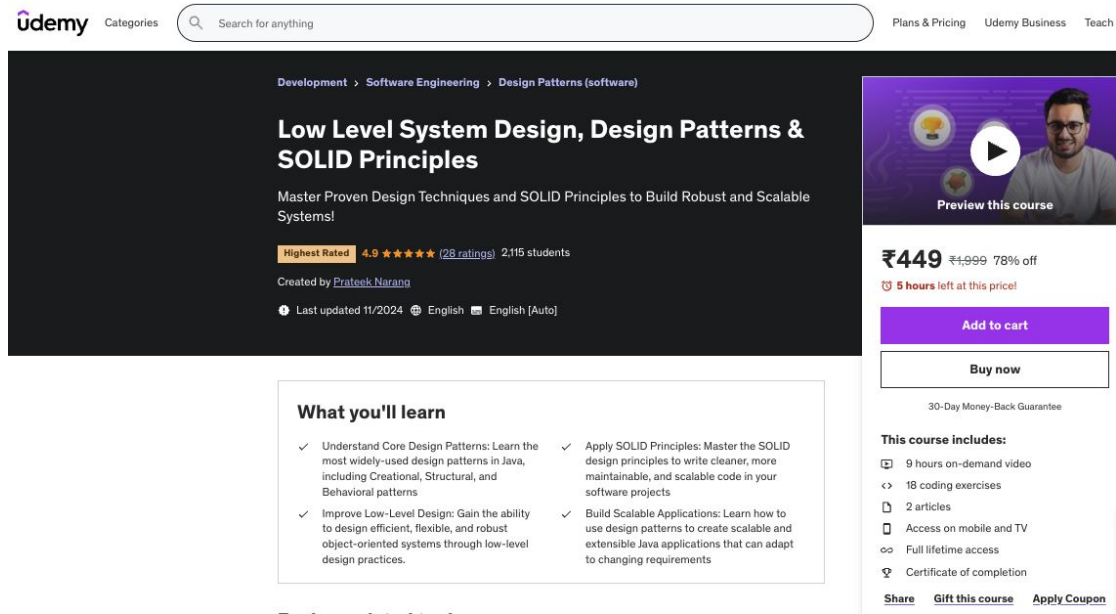
Low Level System Design, Design Patterns & SOLID Principles

Prateek Narang



PDF Notes

Course Link



The screenshot shows the Udeemy course page for "Low Level System Design, Design Patterns & SOLID Principles" by Prateek Narang. The course is categorized under Development > Software Engineering > Design Patterns (software). It has a rating of 4.9 stars from 28 ratings and 2,115 students. The price is ₹449, which is 78% off from ₹1,999. There are 5 hours left at this price. The course includes 9 hours of on-demand video, 18 coding exercises, 2 articles, access on mobile and TV, full lifetime access, and a certificate of completion. The page also features a "What you'll learn" section with four bullet points.

Development > Software Engineering > Design Patterns (software)

Low Level System Design, Design Patterns & SOLID Principles

Master Proven Design Techniques and SOLID Principles to Build Robust and Scalable Systems!

Highest Rated 4.9 ★★★★★ (28 ratings) 2,115 students

Created by [Prateek Narang](#)

🕒 Last updated 11/2024 🗣️ English 📄 English [Auto]

What you'll learn

- ✓ Understand Core Design Patterns: Learn the most widely-used design patterns in Java, including Creational, Structural, and Behavioral patterns
- ✓ Improve Low-Level Design: Gain the ability to design efficient, flexible, and robust object-oriented systems through low-level design practices.
- ✓ Apply SOLID Principles: Master the SOLID design principles to write cleaner, more maintainable, and scalable code in your software projects
- ✓ Build Scalable Applications: Learn how to use design patterns to create scalable and extensible Java applications that can adapt to changing requirements

₹449 ₹1,999 78% off

🕒 5 hours left at this price!

[Add to cart](#)

[Buy now](#)

30-Day Money-Back Guarantee

This course includes:

- 📺 9 hours on-demand video
- 🔗 18 coding exercises
- 📄 2 articles
- 📱 Access on mobile and TV
- 🕒 Full lifetime access
- 📄 Certificate of completion

[Share](#) [Gift this course](#) [Apply Coupon](#)

For any suggestions, discrepancy in the notes
please reach me at prateeknarang111@gmail.com

Design Patterns Overview

- Design patterns are the foundation of good software design. They help you solve recurring problems and improve the structure and quality of your code.
- This course offers hands-on examples and real-world scenarios to help you understand and implement these patterns effectively.
- Whether you're preparing for technical interviews or aiming to write better software, this course will give you the practical skills and confidence you need to level up your design capabilities.

Learning Objectives

- Understand the **fundamental design patterns** used in software engineering and how to apply them effectively in Java.
- Explore **creational, structural, and behavioral patterns**, and their practical use cases.
- Write **cleaner, more efficient code** by leveraging well-established design principles.
- Learn how to **improve low-level design** for building scalable and maintainable applications.
- Solve complex design problems with ease using **proven design techniques**.

Who is this course for?

For everyone interested in Computer Science and Software Engineering, this is a MUST Do course!

- Software Engineers, Junior Engineers and Developers
- Students and Fresh Graduates:
- System Architects and Tech Leads

Design Patterns for Software Engineers

Build Scalable, Extensible & Maintainable Object Oriented Software

Prateek Narang



Pre-requisites

- Basic knowledge of programming languages & OOPS concepts preferably Java
- Basic problem solving skills & logical thinking
- Familiarity with Java IDEs (IntelliJ IDEA, Eclipse etc)

Course Logistics

- Github Repository
 - <https://github.com/prateek27/design-patterns-java>
- Slides/PDF Notes
- Coding Exercises
- Quizzes
- Q/A & Discussion Section

Course Structure

1. OOP Recap
2. SOLID Principles
3. Design Patterns
 - Behavioural
 - Creational
 - Structural
4. Project

Module 1

Object Oriented Programming Recap

Prateek Narang



Keys Concepts in OOPS

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Encapsulation

- Binding data (variables) and methods (functions) together into a single unit, usually a class.
- Control **access to data** via access modifiers (e.g., private, public).
- Example: **Getters and setters** in a class to access and update private data members.

Abstraction

- Hiding unnecessary details and showing only the essential information.
- Focus on **what an object does**, not **how it does it**.
- Example: Defining a method `draw()` in a `Shape` class without revealing the drawing logic (implemented by subclasses like `Circle`, `Rectangle`).

Inheritance

- Mechanism for creating new classes based on existing classes.
- Helps in code **reuse** and establishing relationships between objects.
- Example: `Car` class inherits from `Vehicle` class and adds specific behavior like `startEngine()`.

Polymorphism

- Allows one interface to be used for a general class of actions.
- Achieved through **method overriding** (runtime polymorphism) or **method overloading** (compile-time polymorphism).
- Example: A single method `area()` can be used for different objects like `Circle`, `Rectangle`, with different implementations.

Design a Payment Service

For this example, we can assume payment service should be responsible for storing details of Payments Methods (Credit Card, Debit Card etc) that belong to a user and also have a *makePayment()* Method.

UML - Unified Modeling Language

UML Diagrams

- Unified Modeling Language to model systems
- The idea is to have a uniform way to represent the classes, objects, relationships and interactions within simple or complex systems to make it easier for developers and stakeholders to understand and communicate about the system.

UML Diagrams Importance

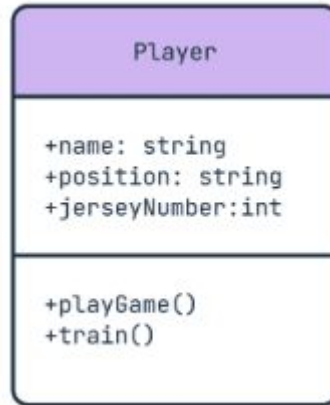
- **Visualization:** UML diagrams provide a visual representation of a system, making it easier to understand the structure, relationships, and interactions between components.
- **Documentation:** They serve as detailed documentation for the software architecture, which is useful for maintaining and scaling the system.
- **Communication:** UML diagrams are a common language for software designers, developers, and stakeholders, facilitating communication about the design.
- **Standardization:** Since UML is a standardized language, it ensures that everyone involved understands the system's design in the same way.

Basic Elements

1. **Class:** A blueprint for objects, defined with a name, attributes, and methods.
 - Example: `class User { name, age, login() }`
2. **Interface:** A contract that defines methods that a class must implement.
 - Example: `interface Loginable { login() }`
3. **Object:** An instance of a class at runtime.
4. **Association:** A relationship between two classes that represents interactions between objects.
5. **Inheritance:** Represents an "is-a" relationship, where a subclass inherits from a superclass.
6. **Composition:** A stronger association where one object is part of another and cannot exist independently.
7. **Aggregation:** A weaker form of association where one object contains another, but they can exist independently.

Class Diagrams

Class Diagram: Represents the static structure of a system, showing classes, attributes, methods, and the relationships between them (inheritance, association, etc.).



Association

Association represents a relationship between two or more classes. In this case, each object in one class is associated with one or more objects of another class.

Code

Aggregation

Aggregation is a weak "has-a" relationship where one class contains objects of another class. However, the contained objects can exist independently of the container object.

Code

Aggregation Example

In this example, a **Department** is composed of multiple **Professors**, but the professors exist independently. Even if the department is dissolved, the professors can still exist.

Composition

Composition is a strong "has-a" relationship, where one class owns objects of another class. If the container object is destroyed, the contained objects are destroyed as well.

Code

Composition Example

House and Rooms

Inheritance

Inheritance defines an "is-a" relationship where a subclass inherits properties and behaviors (methods) from a superclass.

Dependency

This is a relationship where one class relies on another in some way, often through method parameters, return types or temporary associations.

Summary

- **Association:** Objects are related but can exist independently.
- **Aggregation:** A weak "has-a" relationship where the contained objects can exist independently.
- **Composition:** A strong "has-a" relationship where the contained objects cannot exist without the container.
- **Inheritance:** A subclass inherits from a superclass (is-a relationship).
- **Dependency:** One class depends on another for its functionality.
- **Realization:** A class implements the behavior defined by an interface.

How OOP Concepts Influence Design Patterns

Additional Notes-I

- **Encapsulation and Design Patterns:**
 - Encapsulation forms the backbone of **Creational Patterns** like **Singleton** and **Builder**, where object creation details are hidden.
 - Example: Singleton encapsulates the creation and access control to a single instance of a class.
- **Abstraction and Design Patterns:**
 - Many patterns rely on abstraction, such as **Factory Pattern** and **Abstract Factory**, where the client interacts with abstract interfaces.
 - Example: Factory Pattern abstracts the creation of objects by delegating it to subclasses.

Additional Notes-II

Inheritance and Design Patterns:

- Inheritance is key to **Structural Patterns** like **Adapter** and **Decorator**, where behavior is added or adapted using subclassing.
- Example: **Decorator Pattern** uses inheritance to add responsibilities to objects at runtime.

Polymorphism and Design Patterns:

- **Behavioral Patterns** like **Strategy** and **Command** use polymorphism to define interchangeable behaviors.
- Example: Strategy Pattern allows different algorithms (polymorphic behavior) to be swapped dynamically during runtime.

SOLID Principles

SOLID Principles

The **SOLID** principles are a set of five design principles in object-oriented programming and software design. They aim to make software more understandable, flexible, and maintainable by following a structured approach

SOLID Principles

S – Single Responsibility Principle (SRP)

O – Open/Closed Principle (OCP)

L – Liskov Substitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

S-Single Responsibility Principle

A class should have only one reason to change, meaning it should only have **one responsibility**.

Example: A `User` class should only handle user-related logic, while database-related operations should be handled by a separate `UserRepository` class.

O – Open/Close Principle

- Software entities (classes, modules, functions) should be **open for extension** but **closed for modification**.
- Example: Adding new functionality to a system using **inheritance** or **composition** without modifying existing code.

L – Liskov Substitution Principle

- The **Liskov Substitution Principle (LSP)** states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. It ensures that a subclass can stand in for its parent class and function correctly in any context that expects the parent class.

I – Interface Segregation Principle

- The **Interface Segregation Principle** ensures that classes are not burdened with methods they don't need. It promotes better design by breaking large, general-purpose interfaces into smaller, more specific ones.

It improves **maintainability**, **flexibility**, and **testability** by ensuring that classes only have the dependencies they actually require.

D – Dependency Inversion Principle

- High-level modules should not depend on low-level modules; both should depend on **abstractions**.

DRY Principle

DRY (Don't Repeat Yourself)

Avoid code duplication by **reusing** existing code and organizing it into reusable functions, classes, and modules.

Why It Matters: Repeated code increases the risk of bugs, makes maintenance difficult, and increases the chance of inconsistencies.

Example:

Instead of writing the same database connection logic in multiple places, use a **DatabaseConnection** class that handles this logic, ensuring **code reuse**.

What are Design Patterns?

Design Patterns

- 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.

Benefits of Design Patterns

- **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.

Historical Context

- **Origin (1970s-1980s):** Early ideas of patterns began with **Christopher Alexander** in architecture.
 - His concepts of patterns in building design inspired the software industry.
 - 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

Historical Context

Gang of Four (1994): The most famous book on design patterns, "**Design Patterns: Elements of Reusable Object-Oriented Software**", was written by **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides** (GoF). Introduced **23 classical design patterns**.

- **Importance:**
 - Promoted **modular, maintainable, and scalable code**.
 - Encouraged **reusability** of solutions across different projects.
- **GoF Patterns:** Many of the 23 patterns introduced in the Gang of Four book are still widely used today, e.g., **Factory, Singleton, Observer**.

Types of Design Patterns

Behavioral Patterns: Focus on communication between objects and responsibilities.

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

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*

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

Examples: *Adapter Pattern, Composite Pattern, Decorator Pattern*

Example use-cases

- **Creational Example:** A factory that produces different types of documents (Word, PDF, Excel) depending on the input parameters.
- **Structural Example:** Using the Adapter Pattern to integrate a new API into an existing codebase.
- **Behavioral Example:** A notification system using the Observer Pattern, where various modules subscribe to system events.

Module 1

Behavioural Design Patterns

Prateek Narang



Agenda

- Overview of Behavioral Design Patterns
- Observer Pattern
- Strategy Pattern
- Command Pattern
- Template Method Pattern
- Iterator Pattern
- State Pattern
- Mediator Pattern
- Memento Pattern

Behavioral Patterns

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.

Behavioral Patterns

Common Applications:

- Coordinating **interactions** between objects.
- Managing **state transitions** and **communication** efficiently.

Memento Pattern

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.

Memento 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.

Memento Pattern Structure

- **Components:**

- **Originator:** The object whose state needs to be saved and restored. (Editor)
- **Memento:** Captures and stores the internal state of the originator. (Editor Memento)
- **Caretaker:** Manages and stores the mementos, without modifying them. (Caretaker)



Memento Pattern 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.

Observer Pattern

Motivation

Suppose we have a weather station that records temperature and multiple devices (e.g., display units) want to show the latest temperature. Without using the observer pattern, the weather station would have to explicitly inform each device about the temperature change, which results in tight coupling between the station and devices.

Code

Problems with Implementation

The `WeatherStation` class is tightly coupled to the `DisplayDevice` class.

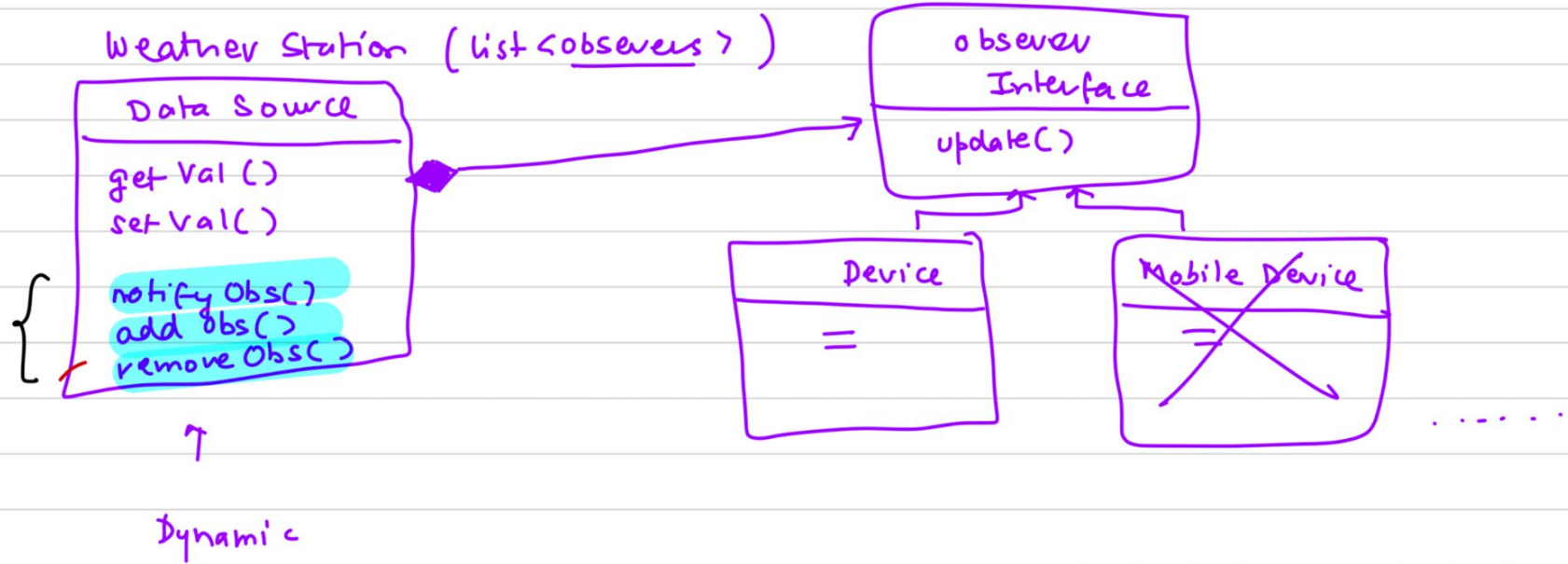
If we need multiple devices (e.g., mobile, tablet) to display the temperature, we need to modify the `WeatherStation` class, leading to poor scalability and flexibility.

Observer Pattern

Problem: There is a need to notify multiple objects about a change in state without tightly coupling them.

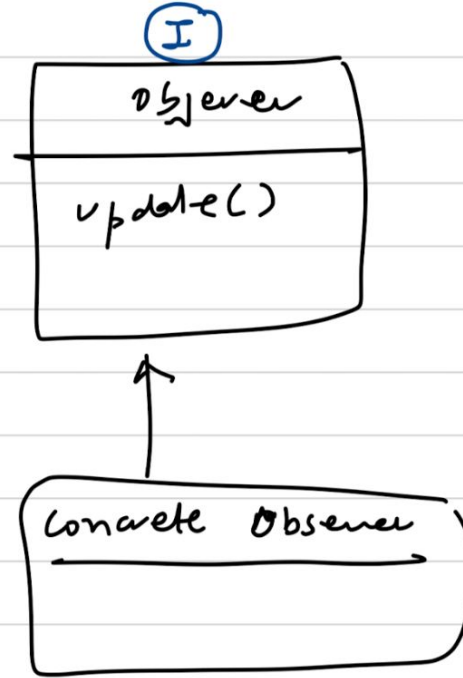
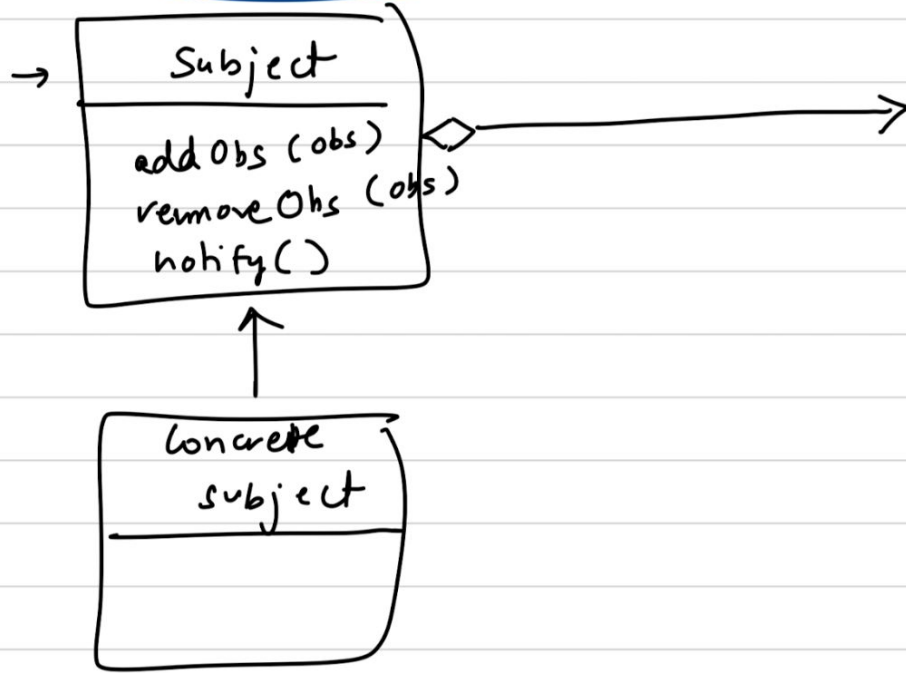
Solution: The Observer Pattern defines a **one-to-many dependency**. When one object (the **Subject**) changes its state, all its dependents (the **Observers**) are notified and updated automatically. This pattern is widely used for implementing distributed event-handling systems, also known as the **publish-subscribe** pattern.

Observer Pattern



Publisher - Subscribers Pattern

Abstract class



Observer Pattern Structure

Subject: Maintains a list of observers and notifies them of changes.

Observers: Receive updates from the subject.

Observer Pattern 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.

Observer Pattern Use Cases

- **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.

Strategy Pattern

Motivation

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.

Code

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.

Strategy Pattern

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.

Strategy Pattern Use Case

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.

Command Pattern

Motivation

Imagine you're developing a basic text editor with buttons for **bold**, **italic**, and **underline** text formatting.

Without the Command Pattern, the buttons directly interact with the `TextEditor` class, and you'd end up hardcoding behavior into the UI classes, making them tightly coupled.

Code

Problems in Code

- Each button class is tightly coupled with the `TextEditor`. If the action changes, all button classes need to be modified.
- It's harder to extend with new commands or add functionality such as undo/redo or logging.

Command Pattern

By introducing the Command Pattern, we can decouple the actions (bold, italic, underline) from the UI components (buttons), making the design more flexible and maintainable. The buttons no longer need to know about the editor directly but instead work with generic **Command** objects.

Command Pattern Structure

Structure:

- **Command:** Interface for executing operations.
- **Invoker:** Sends the command.
- **Receiver:** Performs the operation.

UML Diagram

Command Pattern Benefits

- **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.

Command Pattern Use Cases

GUI Applications:

- Commands can be associated with buttons, menus, and keyboard shortcuts in applications like text editors, spreadsheets, or drawing software.

Task Scheduling:

- Commands can be placed in a queue and executed later, useful in batch processing or deferred task execution.

Command Pattern Use Cases

Undo/Redo Functionality:

- Commands can be stored and rolled back to provide undo and redo capabilities, especially in applications like IDEs, word processors, or graphics software.

Macro Recording:

- Actions performed by the user can be recorded as a series of commands, which can then be played back as macros.

Command Pattern Drawbacks

- **Increased Complexity:** Introducing the Command Pattern can lead to more classes and complexity, especially when there are many different commands.
- **Overhead:** Each operation becomes an object, which may add memory and performance overhead in systems with large numbers of commands.

Template Method Pattern

Motivation Problem

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 our 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.

Template Method Pattern

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

Template Method Pattern

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.

Use Cases of Template Method

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

Iterator Pattern Motivation

Suppose you have a collection, such as an array or list, and you need to provide a mechanism for accessing its elements. Without the iterator pattern, the client code needs to understand how the collection is structured, and different collections would require different methods to traverse them.

Code Demo

Problems in our Code

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.

Iterator Pattern

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.

Iterator Pattern Drawbacks

1. **Additional Complexity:** Implementing the iterator pattern can add extra layers of abstraction, especially for small or simple collections where direct traversal is sufficient.
2. **Increased Overhead:** For small collections or when the structure is unlikely to change, the overhead of creating iterators may not be justified.

State Pattern

State Pattern

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**

Code Demo

Problems with our Code

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).

Problems with our Code

Code Duplication and Reduced Maintainability:

- Similar code blocks for different transportation modes may lead to **code duplication**, making the system less maintainable and more error-prone.

Scalability Issues:

- As more features or transportation modes are added, the class becomes **bulky**, impacting scalability and readability.

State Pattern: Structure

Structure:

- **Context:** Holds a reference to the current state.
- **State:** Interface for state-specific behavior.
- **Concrete State:** Specific implementations of the **State** interface that represent a particular state of the context object.

State Pattern: Example

UI Navigation

- **Scenario:** A mobile app UI where the navigation behavior changes based on whether the user is logged in or not.

Example:

- States: **LoggedInState**, **LoggedOutState**.
- Context: The app's navigation system switches between these states.

State Pattern Use Cases

1. **UI Components:** Buttons that change behavior based on state (enabled, disabled, pressed).
2. **Vending Machines:** States like waiting for money, dispensing product, or out of stock.
3. **TCP Connections:** Changing behavior based on connection state (listening, connected, closed).

Mediator Pattern

Motivation

We want to build a chat system with multiple participants where each user can send messages to all other users. If users send messages to each other directly, the complexity increases as more users are added. Each user must know about every other user, creating a complex web of communication and dependencies.

Problems with Code

- As more users are added, each user needs to manage direct communication with all others, leading to high coupling.
- If a new communication rule is introduced (e.g., message logging), it would need to be added to all users.

Mediator Pattern

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.

Mediator Pattern Code

1. **ChatMediator Interface:** Declares the `sendMessage` method, which all mediators must implement.
2. **Concrete Mediator:**
 - The `ChatRoom` class implements the `ChatMediator` interface. It holds a list of users and handles message broadcasting.
3. **User Class:**
 - Each `User` object represents a participant in the chat. When a user sends a message, the `sendMessage` method in the `ChatRoom` mediator is called, which distributes the message to all users except the sender.
4. **Communication:**
 - Users interact only with the `ChatRoom` (mediator), which facilitates communication between them, removing direct dependencies between individual users.

Mediator Pattern 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.

Mediator Pattern in Real World

Air Traffic Control:

Airplanes communicate through a central control tower (mediator) instead of coordinating directly with each other.

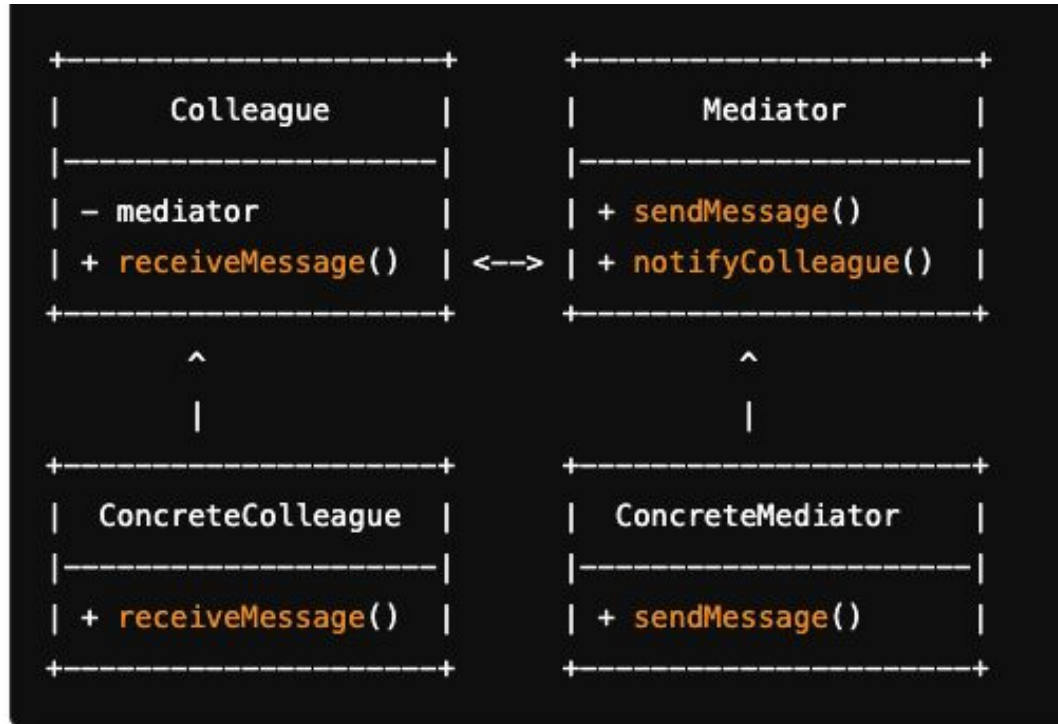
GUI Component Coordination:

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.

Workflow Systems:

In a business process management system, a mediator can coordinate various activities across multiple systems or departments.

UML Diagram



Summary

Summary

This module covers key **Behavioral Design Patterns** that help define how objects interact and manage state transitions within a system. From **observer-based event notification** to **decoupling algorithms with strategies**, each pattern offers a solution to common communication challenges in software design.

Module

Creational Design

Patterns

Prateek Narang



Agenda

- Overview of Creational Design Patterns
- Singleton Pattern
- Builder Pattern
- Prototype Pattern
- Factory Pattern
- Abstract Factory Pattern

Overview

Purpose: Creational patterns are focused on **object creation mechanisms**, aiming to optimize the creation process while ensuring flexibility.

Goal: They abstract the instantiation process to make systems **more flexible and reusable**.

Problem They Solve: Prevent tight coupling between code and object creation logic, simplifying the management of new object creation, especially in complex systems.

Example:

- In a system where different types of documents (PDF, Word, Excel) are created, instead of using `new` everywhere, a **Factory Pattern** can centralize object creation.

Overview

Application: They are applied in scenarios where:

- You want to **separate the instantiation** process from usage.
- The exact type of object needed can vary depending on runtime conditions.

Singleton Design Pattern

Motivation

In certain situations, such as managing a database connection, logging, or configuration settings, you want to ensure that only one instance of a class is created throughout the application's lifecycle. If multiple instances were created, it could lead to issues like:

- **Inconsistent state:** If multiple instances represent the same concept, they may hold different data.
- **Resource conflicts:** If multiple instances of a resource-heavy class are created, it can lead to performance degradation.

Singleton Design Pattern

The Singleton pattern is used when exactly **one instance** of a class is required to coordinate actions across the system.

When to Use

- **Global resource management** (e.g., managing database connections, logging).
- **Configuration settings** in applications that need to be shared.

Singleton Design Code

- **Private Constructor:** The constructor is private so that no other class can instantiate `AppSettings` directly.
- **Singleton Access:** The `getInstance()` method is used to access the single instance of the settings.
- **Global Access:** Any part of the application can access the settings using `AppSettings.getInstance()`.

Thread Safety in Singleton DP

Multi-threading issues: Without thread safety, multiple instances of the Singleton class could be created in a multi-threaded environment.

Solutions:

- **Lazy Initialization with Synchronization:** Synchronize the method that creates the instance.
- **Double-Checked Locking:** Optimize the performance by only locking when necessary.
- **Bill Pugh Singleton Design:** Uses an inner static helper class to ensure thread safety and lazy loading.

Code Demo

Builder Design Pattern

Builder Design Pattern

When an object requires many parameters, especially optional ones, the constructor can become hard to use or maintain. This issue can lead to:

1. Long constructor parameter lists.
2. Difficulty in understanding which values are optional or required.
3. Lack of flexibility when it comes to setting only some values.

For example, constructing an object with multiple optional parameters without the Builder pattern can look like this:

Code Demo

Builder Design Pattern

Problem: When a class constructor has too many parameters, the **Builder Pattern** allows step-by-step construction of complex objects.

Solution: Separates the construction of an object from its representation, offering a fluent interface for creating complex objects

Code Demo

Prototype Pattern

Problem

Consider a board game where you need to save the current state of the game at various checkpoints. Instead of manually creating new board objects and copying all the pieces or their states (which could be costly if the board is large and has many game pieces), we can use the Prototype Pattern to clone the board.

We will use the prototype pattern allows us to make a copy of the current board, including all its pieces and their states, without the need for deeply recreating each part of the board.

Code Demo

Solution – Prototype Pattern

- The **Prototype Pattern** can be extremely useful in a board game when you want to save the current state of the game (including the board layout and the position of pieces) for undo/redo functionality, checkpoints, or simply making a copy of the board for a new player.
- Each piece or game element can provide its own `clone` method, allowing the entire board to be easily cloned with all its current state.

By using the Prototype pattern, we decouple the complexity of cloning the board from the client, ensuring that each object knows how to clone itself, making the system flexible and easier to maintain.

Shallow vs Deep Copy

Shallow Copy: Creates a new object but **does not clone the objects** that the original object refers to.

Deep Copy: Clones the original object and all the objects it refers to (nested objects)

Example:

- **Shallow Copy:** Cloning an object with references to other objects (only the outer object is copied).
- **Deep Copy:** Cloning the entire object graph, including any objects the original refers to.

Benefits

- **Simplifies Object Creation:** Instead of manually copying each object, the `clone` method in each object simplifies the creation of copies.
- **Avoids Subclassing:** The pattern relies on delegation to the `clone` method, allowing the class itself to handle object creation, avoiding the need for subclassing.
- **Shallow or Deep Copy:** Depending on the use case, you can either implement a shallow copy (copying references) or a deep copy (cloning objects) based on the specific requirements.
- **Efficient Creation:** When creating objects with a complex structure or when performance is a concern, the Prototype pattern allows you to efficiently replicate objects.
- **Consistency:** Ensures that all properties of the object are consistently copied, avoiding errors associated with manual copying.

Use Cases

- **Use Cases:**
 - Cloning large objects where construction is costly (e.g., deep configuration settings).
 - When system resources (time, memory) are limited and re-creating objects is expensive.

Factory Design Pattern

Motivation

Consider an example of a transportation service app where users can request different types of transport vehicles (e.g., Car, Bike, Bus). You might initially create separate classes for each type, and create instances like this:

```
Car car = new Car();  
Bike bike = new Bike();
```

But as the system evolves, managing object creation directly like this can become complex, especially when adding new types of vehicles.

Code

Problems:

1. The client code (i.e., `TransportService`) is tightly coupled to concrete classes (`Car`, `Bike`, `Bus`).
2. Adding new transport types requires modifying client code.

Factory Design Pattern

The **Factory Pattern** helps centralize the creation logic and delegates the responsibility of creating objects to factory classes, which decide the specific class to instantiate. This allows the code to adhere to the **Open/Closed Principle** by letting new types of vehicles be added without modifying the existing code.

Factory Design Pattern

1. **Factory Class:** The `TransportFactory` class contains the logic to create different types of transport based on the input string. This abstracts the creation logic and makes it easier to add or change transport types.
2. **Decoupling:** The `TransportService` class (client) no longer needs to know the details of how `Car`, `Bike`, or `Bus` are created. It simply calls the factory method.
3. **Flexibility:** Adding a new transport type (e.g., `Truck`) only requires modifying the factory, not the client code.

Factory Design Benefits

Benefits of Factory Pattern:

1. **Loose Coupling:** The client is decoupled from the specifics of object creation.
2. **Single Responsibility Principle:** The factory class handles the responsibility of object creation.
3. **Open/Closed Principle:** We can easily add new transport types without changing the client code, making the system open to extension and closed to modification.

Real World Use Cases

- **GUI Frameworks:** When the type of button or widget to be created is determined at runtime based on the platform (e.g., Windows, macOS, Linux).
- **Database Connectivity:** When choosing different types of databases (e.g., SQL, NoSQL) based on configuration.
- **Document Conversion Tools:** Where the type of file (e.g., PDF, Word, HTML) to be created depends on user input or settings.

The **Factory Design Pattern** is a fundamental tool to reduce coupling and centralize object creation logic, especially in systems that need to support multiple types of objects.

Abstract Factory Pattern

Motivation

Consider an application that needs to support multiple UI themes (e.g., Windows, macOS). Each theme has its own set of UI components such as buttons, scrollbars, and windows. The challenge is to create an architecture that allows switching between these themes without changing the client code that uses the UI components.

Without the Abstract Factory Pattern, the client code would be tightly coupled with the concrete implementations of buttons, scrollbars, etc., and switching between themes would require modifying the client code.

Problems in Code

1. **Tight coupling:** The client code is tightly coupled to specific implementations of UI components (WindowsButton, WindowsScrollBar).
2. **Hard to extend:** If we want to add support for macOS UI components, we would need to modify the client code to create instances of `MacOSButton` and `MacOSScrollBar`.

Abstract Factory Pattern

Problem It Solves: Provides an interface for creating **families of related objects** without specifying their concrete classes.

Structure:

- **Abstract Factory:** Interface for creating abstract products.
- **Concrete Factory:** Implements the abstract factory and creates concrete products.

Using the Abstract Factory Pattern, you create interfaces for each product (e.g., Button, ScrollBar, Window) and provide a family of concrete implementations for each theme (e.g., WindowsButton, MacOSButton, etc.). The Abstract Factory provides a way to create a suite of related objects without knowing the exact type of objects that will be created.

Factory vs Abstract Factory

Factory Method: Defines a method in the base class but lets subclasses override it to specify the type of objects that will be created.

- Example: A **DocumentFactory** subclass that creates either PDF or Word documents.

Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- Example: A **GUIFactory** that creates related objects like buttons, checkboxes, and text fields depending on the operating system (Windows or Mac).

Code Demo

Module 4

Structural Design Patterns

Prateek Narang



Agenda

- Overview of Structural Design Patterns
- Adapter Pattern
- Decorator Pattern
- Proxy Pattern
- Composite Pattern
- Facade Pattern
- Bridge Pattern
- Flyweight Pattern

Overview

Structural patterns are design patterns that deal with how objects and classes are **composed** to form larger structures, making the system more flexible and easy to understand.

- They simplify the design of complex systems by creating **relationships** between objects.
- They help in **organizing class hierarchies** and allow for easy modification and extension.

Adapter Design Pattern

Adapter Pattern

The **Adapter Pattern** is a structural design pattern that allows objects with incompatible interfaces to work together.



Problem Statement

Imagine you are working on an **e-commerce application** that needs to send **email notifications** to customers. Currently, you are using your own in-house `EmailNotificationService`, but now you want to integrate a popular third-party email service like **SendGrid**. However, the interface for your in-house system and the SendGrid service are incompatible.

Let's see how the **Adapter Pattern** can solve this problem.

Adapter Pattern

- **Problem:** When two systems or components have incompatible interfaces, they cannot work together directly.
- **Solution:** The Adapter Pattern bridges the gap by **converting the interface** of one class into another that the client expects.

Real-world analogy:

- A **power adapter** that allows a device with a US power plug to fit and work with a European power socket

Code Demo

Adapter Pattern Examples

- **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.

The Adapter Pattern is a versatile solution to ensure incompatible interfaces work together seamlessly, making it invaluable in software integration and legacy system migration.

Adapter Pattern Benefits

- **Reusability:** You can reuse existing code even when its interface is not compatible with the current system.
- **Flexibility:** It helps integrate classes from different libraries or systems that were not designed to work together.
- **Decoupling:** The Adapter decouples the client from the specific implementation of the Adaptee.

Decorator Pattern

Decorator Pattern

You need to add functionality to an object at runtime, but subclassing would lead to an explosion of subclasses or is impractical.

Decorator Pattern

Let's say we have a simple Pizza ordering system. Initially, we just have plain pizza but now we want to add options such as cheese, olives, tomatoes and mushrooms without modifying the existing pizza class or creating multiple subclasses.

Let's use the **Decorator Pattern** using a Pizza making example where different toppings (like cheese, olives, tomatoes, etc.) can be dynamically added to the base pizza.

Decorator Pattern

Problem: In a system where you need to **dynamically extend or add behavior** to objects, inheritance can lead to inflexible and tightly coupled code.

Solution: The Decorator Pattern allows you to **add new functionality to objects at runtime** by wrapping them with decorator classes, providing flexibility

Real-world analogy:

- A **pizza order** where you start with a basic pizza and add toppings dynamically (e.g., cheese, pepperoni) without modifying the original pizza class.

Code Demo

Decorator Pattern

Flexible and Scalable: You can add as many toppings as needed by simply creating a decorator for each topping. No need for subclassing every combination.

Single Responsibility Principle: Each decorator class has one responsibility — to add a specific topping.

Open/Closed Principle: The `BasicPizza` class remains unchanged, and new features (toppings) can be added by creating new decorators.

Dynamically Changeable: Decorators can be added or removed dynamically at runtime.

Combinatorial Freedom: You can mix and match toppings in any order, making the system more flexible and reusable.

Decorator vs Inheritance

- **Inheritance** is static and applied at compile-time, leading to tight coupling between base and derived classes.
- **Decorator Pattern** provides a more flexible way to add or remove behavior at **runtime** without changing the underlying object.

Proxy Pattern

Proxy Pattern

- **Problem:** Sometimes, direct access to an object might not be desirable due to reasons such as security, resource optimization, or controlled access.
- **Solution:** The Proxy Pattern provides a **surrogate** or placeholder for another object to control access to it.

Proxy Pattern Motivation

Imagine a system where we want to load a heavy object like a large image from disk. It might take time to load the image, and we don't want to load it until it's necessary. Without using a proxy, the application would load the image every time it's needed, even if not displayed, wasting resources.

Code Demo

Proxy Design Pattern

- **Virtual Proxy:** Delays the creation of expensive objects until they are needed (lazy initialization).
- **Protection Proxy:** Controls access based on permissions.

Example

A **large image** loading in an application is expensive, so a virtual proxy is used to load the image only when it's actually needed (on-demand).

Proxy Pattern Benefits

1. **Lazy Initialization:** Objects are loaded only when necessary, saving memory and CPU cycles.
2. **Access Control:** You can control access to the real object (e.g., based on user permissions).
3. **Additional Behavior:** Proxies can add additional functionalities like logging, access control, or caching without modifying the real object.
4. **Separation of Concerns:** The real object only deals with its core responsibilities, while the proxy handles ancillary operations like initialization or security.

Composite Pattern

Composite Pattern

Problem: When building systems like a **file directory**, which consist of both individual items (files) and groups of items (directories), managing these with standard object hierarchies can become complex.

Solution: The Composite Pattern allows you to treat **individual objects** and **compositions of objects** uniformly by representing part-whole hierarchies.

Composite Pattern

- In a **file system**, both **File** and **Directory** can be treated as components. A directory can contain both files and other directories, forming a recursive tree structure.

Facade Pattern

Motivation

In software engineering, a common real-world example of the **Facade Pattern** is an **API Gateway** in a microservices architecture.

Problem Without Facade

In a microservices architecture, each microservice can have its own API for specific business logic, such as user management, order processing, and inventory. If the client needs to interact with these microservices, it would need to directly communicate with all the individual services. This would increase the complexity of the client code, tightly couple the client to all the microservices, and expose the inner workings of the system.

Motivation

Solution Using Facade (API Gateway):

The **API Gateway** acts as a facade, providing a unified interface to the client while handling communication with the underlying microservices. It simplifies client interactions, reduces network calls, and abstracts away the complexity of dealing with multiple services.

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.

Facade Pattern

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.

Facade Pattern

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.

Facade Pattern Benefits

- **Decoupling:** The client is decoupled from the internal structure of the microservices. Any changes to the underlying services (e.g., changes in APIs or architecture) can be handled within the facade without impacting the client.
- **Reduced Complexity:** The facade handles the orchestration of complex operations, hiding the complexity of multiple service interactions. This reduces the learning curve for developers working with the system.
- **Centralized Control:** The facade (API Gateway in this case) can enforce security, logging, caching, and rate-limiting centrally. This avoids duplicating these concerns across multiple services or clients.
- **Easier Maintenance:** With a facade in place, changes in one subsystem do not ripple out to clients. For example, if the order service API changes, the facade can adapt without needing to update the client code.

Facade Pattern Benefits

- **Enhanced Performance:** By consolidating multiple requests into a single call through the facade, you can reduce network overhead, especially in distributed systems like microservices. For example, fetching user details, order details, and payment information in a single request rather than three separate requests.
- **Consistency in Interfacing:** The facade ensures a consistent interface for clients, even if the underlying systems evolve over time or become more complex. This helps keep the system modular and more understandable for future changes.

Bridge Pattern

Bridge Pattern

- **Problem:** You need to separate an object's **abstraction** from its **implementation** so they can vary independently.
- **Solution:** The Bridge Pattern decouples abstraction from implementation, allowing the two to evolve separately.

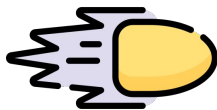
Bridge Pattern

- A **shape abstraction** that can be implemented using different **rendering systems** (e.g., Vector or Raster)

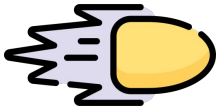
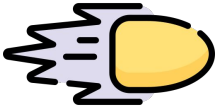
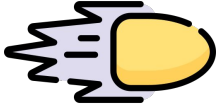
Flyweight Pattern

Flyweight Pattern Motivation

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



Flyweight Pattern Motivation



Flyweight Pattern Solution

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



Flyweight Pattern

Problem: Creating many identical or similar objects in memory-heavy applications can lead to excessive memory consumption.

Solution: By sharing the common state among multiple objects (the **flyweights**), the pattern reduces the memory footprint.

Example Problem:

- Consider a graphics system rendering a large number of trees in a forest, where each tree shares the same image but may vary by position or size.

Flyweight Pattern

The Flyweight Pattern is a structural pattern used to **minimize memory usage** by sharing as much data as possible with similar objects. It separates the **intrinsic state** (shared) from the **extrinsic state** (non-shared).

Purpose: To optimize resource usage in applications that need to handle a large number of similar objects by **reusing** common parts of their data.

When to Use:

- When you need to create a large number of objects, and they share a lot of common data.
- Ideal for memory-constrained applications where the object creation cost is high

Flyweight Pattern Structure

- **Flyweight:** The shared object that stores **intrinsic state** (common data).
- **Concrete Flyweight:** Implements the flyweight interface and shares the intrinsic state.
- **Flyweight Factory:** Manages and creates flyweight objects, ensuring that objects with the same intrinsic state are reused.
- **Client:** Holds and manages the **extrinsic state** (unique information) and delegates behavior to the flyweight.

Flyweight Pattern Example

Scenario: A program that needs to render a large number of trees in a forest.

- **Intrinsic State:** Tree type, shape, texture, and color (shared across trees).
- **Extrinsic State:** Tree position and size (unique per tree).

Without Flyweight: Each tree object would store all properties, leading to **high memory usage**.

With Flyweight: The tree objects share common properties, and only the unique properties (position and size) are stored separately.

Final Project Ride Sharing App

Prateek Narang



Design a Ride Sharing App!

Ride Sharing App

You are tasked with designing and implementing a ride-sharing application where passengers can request rides, and drivers can be matched to them based on proximity. The application should handle different types of vehicles (such as cars, bikes, luxury cars) and support multiple fare calculation strategies. The system must notify both passengers and drivers about ride statuses and calculate the fare based on the type of ride and distance traveled.

Let's Refactor our Code

Coding Ride Sharing App with SOLID Principles

Requirements-I

Ride Request:

- Passengers can request a ride by providing their location and the desired destination.
- The system should calculate the distance between the passenger's location and the driver's location.
- The system must assign the **nearest available driver** to the passenger.

Vehicle Types:

- The system should support different vehicle types (e.g., car, bike, luxury car).
- Each vehicle type should have a different base fare per kilometer.

Fare Calculation:

- The system should use **different fare strategies** (e.g., standard fare, shared fare, luxury fare).

Requirements-II

Ride Status Notifications:

- Both the passenger and the driver should be notified of ride statuses (e.g., ride started, ride completed).
- Use the **Observer Pattern** to notify users about ride status updates.

Ride Matching:

- Drivers should be assigned to passengers based on **proximity**.
- After a ride is completed, the driver becomes available for new ride requests.

Bad Code Example :(

Let's write some code which doesn't follow any principles!

Lets' Refactor!

Lets modify our code using SOLID principles & Design Patterns practices that we have seen.

Thank you!