Object-Oriented Design Patterns

Intro

- Design patterns are reusable, tested soln^s for common software design problems, helping in efficient problem-solving.
- A-P-I-E (Abstraction, Polymorphism, Inheritance, Encapsulation) forms the foundation of OOP, while design patterns acts as blueprints for scalable & maintainable software.

Creational Design

- Prototype
- Factory Method
- Singleton
- Builder

Factory

- A way to create objects without specifying the exact class.
- keeps code flexible & avoids hardcoding object creation.

- Simple Factory creates object in one place ;
  Factory method lets subclass decide.
- Makes code scalable, modular & easier to maintain.
- Used in UI components, db conn., & plugin systems.

Singleton

- One instance, Global Access.
- Private Constructor - Multiple objects create hone se rokta hai.
- Static Method (getInstance()) - Ek hi instance return karta h.
- Thread-Safety - Double-checked Locking use karo taaki multi-threading issues na aaye.
- Lazy Initialization - Jab zaroorat ho tab instance create hota h, memory bachat h.
- Prevent Reflection Attack - Constructor me check lagao.
- Prevent Serialization Issue - readResolve() implement karo taaki deserialization

naye objects na banaye.

- Prevent cloning :- clone() method override karke exception throw karo.
- Use cases :
  - Logging, Configuration Management, Thread Pools, Caching db conn pooling.

[Singleton pattern ek hi instance ka control deta h, but misuse se avoid karo (reflection, serialization, multithreading issues)].

Builder

- Step-by-step object construction for complex objects.
- Avoids telescoping constructors, improves readability & flexibility.
- Uses a builder class to set properties & a build() method to create the object.
- Clean, flexible, readable., supports method chaining, predefine configurations via Director
- Use when there is complex object creation, immutability, multiple configurations.

[Use Builder when object has many optional fields or step-by-step creation is needed.]

Prototype

- Clone Instead of Create - Duplicate existing objects using clone(), avoiding costly instantiation.
- Handles Complex objects - Useful for duplicating objects with many properties (e.g., documents, game assets).
- Shallow vs. Deep Copy - Shallow copies share references, deep copies create independent objects.
- Fast & efficient - Saves time & resources compared to rebuilding objects from scratch.
- Use Cases - Undo/Redo, game development, UI elements & prototype-based programming.

Structural Patterns

- Adapter
- Decorater
- Facade

Adapter

- Bridge b/w Incompatible Interfaces : Converts one class's interface into another, so they can work together without modifying existing code.
- Used for Legacy Code Integration : Helps connect old systems with new ones without rewriting everything.
- 2 types : Class (Inheritance-based), Object (Composition)
  - Class → Uses multiple inheritance
  - Object → Uses object composition (preferred)
- e.g., HDMI to VGA converter, API wrappers, Charging adapters.
- Uses Code Reusability : Makes different systems communicate efficiently without modifying their core logic.

Decorator

- Enhances Behavior Dynamically - Adds new functionality to objects at runtime without modifying their structure.
- Uses Composition instead of inheritance - Avoids subclass explosion by wrapping objects instead of extending classes.
- Flexible & Scalable - Multiple decorators can be stacked to add different behaviors without altering the core object.
- e.g., Adding features to UI components, logging in functions, data compression, & encryption layers.
- Follows Open/Closed Principle (OCP) - Objects are open for extension but closed for modification, improving maintainability.

Facade

- Provides a unified, easy-to-use interface to a set of complex subsystems.
- Reduce Coupling: Hides the internal details of subsystems, making client

interactions simpler & more maintanable.

- Improves Readability & Usability : Clients interact with a single high-level API instead of multiple subsystem calls.
- Used in Frameworks & Libraries : Commonly found in APIs, db conn., logging systems, & UI frameworks.
- Follows the Single Responsibility Principle (SRP) : keeps subsystems separate while offering a clear access point for clients.

Behavioural Patterns
- Strategy
- Observer
- State

Strategy

- Encapsulates Algorithms - Defines a common interface for interchangeable behaviors or algorithms.
- Dynamic Selection - Allows switching strategies at runtime based on context or conditions.
- Promotes Reusability - Seperate algorithm implementation from client code, enhancing modularity.
- Simplifies Maintenance - Adheres to open/closed principle; new strategies can be added without altering existing code.
- e.g., used in sorting, payment processing, navigation systems, & discount calculations.

Observer

- Observers are automatically notified when the subject's state changes.
- Loose Coupling: Subjects & observers remain independent, promoting flexibility & easier maintenance.
- Dynamic Relationships: Observers can be added or removed at runtime without affecting the subject.
- Event-Driven Architecture: Ideal for real-time systems like notifications, stock updates, or UI event handling.
- Enhances Modularity: Separates the core logic (subject) from response actions (observers), adhering to the SRP.

State

- Dynamic Behavior Change: Object behavior changes based on its internal state without complex conditionals.
- Encapsulated States: Each state is implemented in its own ~~state~~ class, keeping behavior modular & organized.

- Eliminates Conditional Logic : Removes extensive if/else or switch-case blocks by delegating tasks to state objects.
- Open for extension : New states can be added easily without modifying existing code, following the Open/Closed Principle.
- e.g., used in media players, ATMs, traffic lights & other processing systems.

Notes Date 28/03/25

DESIGN PATTERNS - Cheatsheet

Creational

- Singleton → One global instance for shared resources.
- Factory → Creates objects without specifying the exact class.
- Builder → Constructs complex objects step-by-step.
- Prototype → Clones existing objects for efficient duplication.

Structural

- Adapter → Converts one interface to another (bridging incompatibility).
- Decorator → Adds dynamic behavior to objects without altering their structure.
- Facade → Provides a simple interface to complex subsystem.

Behavioral

- Strategy → Encapsulates interchangable algorithms; choose at runtime.
- Observer → Notifies multiple objects automatically when state changes.
- State → Changes object behavior based on its internal state.