Palash Bajpai

Low Level Design

Table of Contents

[**1. Introduction to Low-Level Design** 2](#_Toc191561376)

[**2. SOLID Principles** 3](#_Toc191561377)

[a. Single Responsibility Principle 5](#_Toc191561378)

[b. Open/Closed Principle (OCP) 7](#_Toc191561379)

[c. Liskov Substitution Principle (LSP) 9](#_Toc191561380)

[d. Interface Segregation Principle (ISP) 11](#_Toc191561381)

[e. Dependency Inversion Principle (DIP) 13](#_Toc191561382)

[**3.** **Design Patterns** 15](#_Toc191561383)

[**4.** **Singleton Pattern** 17](#_Toc191561384)

[**5.** **Factory Design Pattern** 20](#_Toc191561385)

[**6.** **Abstract Factory Design Pattern** 23](#_Toc191561386)

[**7.** **Builder Design Pattern** 28](#_Toc191561387)

[**8.** **Prototype Design Pattern** 31](#_Toc191561388)

[**9.** **Adapter Pattern** 35](#_Toc191561389)

[**10. Decorator Pattern** 39](#_Toc191561390)

[**11. Proxy Pattern** 43](#_Toc191561391)

<https://roadmap.sh/system-design>

<https://roadmap.sh/software-design-architecture>

<https://media.geeksforgeeks.org/courses/syllabus/b16669e7619419215993fe653e0b2fee.pdf>

https://www.geeksforgeeks.org/complete-roadmap-to-learn-system-design/

# **1. Introduction to Low-Level Design**

A system is a set of interconnected components working together to achieve a common goal, while system design is the process of defining and implementing the architecture, components, and interfaces of a system to fulfill specific requirements.

1. High Level Design (HLD): describe system architecture details, database design, service and processes, relationship between different modules and features.
2. Low Level Design (LLD): describe design of each component in detail, classes, interfaces, relationship between them.

Low Level Design

Low-Level Design (LLD) is the process of designing software at the component level. It provides detailed descriptions of modules, classes, methods, and interactions between system components. LLD focuses on implementation details, object-oriented design principles, and design patterns.

🔹 **Key Characteristics of LLD**

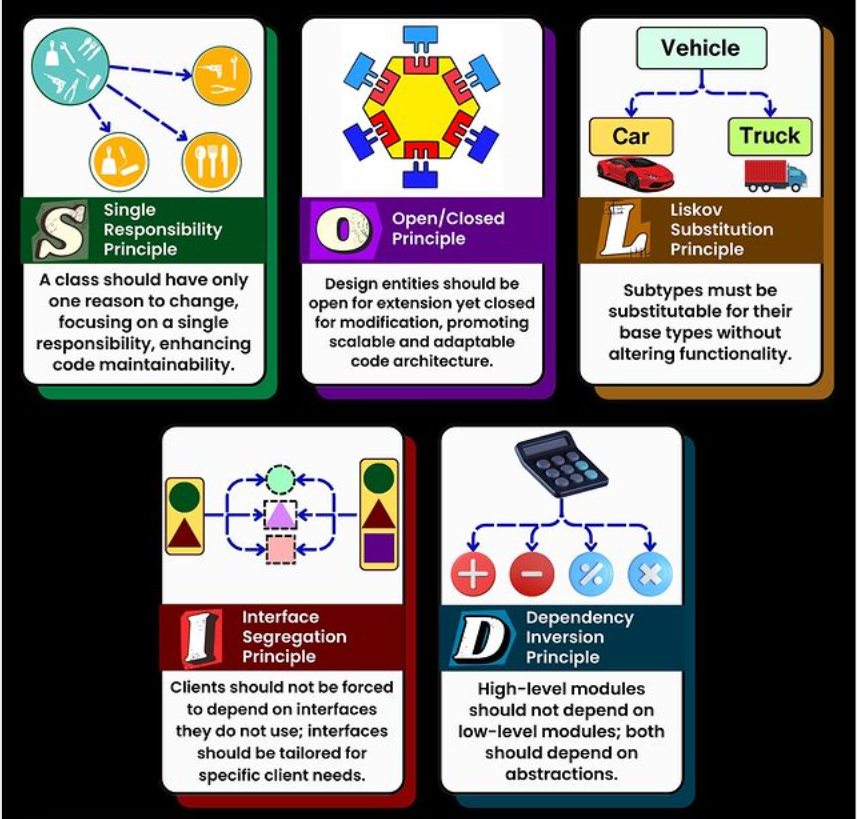
* Defines the structure of classes and objects.
* Includes UML diagrams such as Class Diagrams, Sequence Diagrams, and Component Diagrams.
* Follows Object-Oriented Programming (OOP) principles.
* Uses design patterns to solve common software design problems.

🔹 **Importance of LLD**

* Improved Code Maintainability: Well-structured code is easier to update and debug.
* Better Collaboration: LLD serves as a blueprint for developers, reducing misunderstandings.
* Optimized Performance: Proper design improves performance by reducing redundancy and increasing reusability.
* Enhanced Scalability: Well-designed software can be extended without major refactoring.
* Promotes Best Practices: Encourages the use of SOLID principles and design patterns.

# **2. SOLID Principles**

SOLID is a set of five design principles that help software developers write maintainable, scalable, and robust code in object-oriented programming. These principles were introduced by Robert C. Martin (Uncle Bob) to improve software design and ensure that systems are easy to extend and modify.­



| **Principle** | **Description** |
| --- | --- |
| **S** - Single Responsibility Principle (SRP) | A class should have only **one reason to change**. |
| **O** - Open/Closed Principle (OCP) | Code should be **open for extension but closed for modification**. |
| **L** - Liskov Substitution Principle (LSP) | Derived **classes** should be substitutable for their **base classes.** |
| **I** - Interface Segregation Principle (ISP) | Avoid forcing classes to implement **unnecessary methods** of an **interface**. |
| **D** - Dependency Inversion Principle (DIP) | High-level modules should not depend on low-level modules. Both should **depend on abstractions** |

Advantages of using SOLID principle:-

* **Better Maintainability:** Each class has a single responsibility, making code easier to understand and modify.
* **Scalability:** New features can be added without modifying existing code.
* **Loose Coupling:** Dependency Inversion and Interface Segregation help reduce code dependencies.
* **Easy Testing:** Code becomes modular, allowing better unit testing and mocking.
* **Better Collaboration:** Developers can work on different parts of the system without causing conflicts.

Eg: - Consider building an e-commerce platform, if you follow SOLID principle

1. Adding a new payment method (e.g., UPI) doesn't require modifying the entire payment system.
2. Changing the notification system from SMS to Email won't break existing code.

## Single Responsibility Principle

The **Single Responsibility Principle (SRP)** states that **a class should have only one reason to change**. This means that a class should focus on **only one functionality** and should not mix multiple concerns. If a class handles multiple responsibilities, changes in one part of the system might affect unrelated parts, leading to **fragile and tightly coupled** code.

🔹 **Key Takeaways of SRP**

* A class should have **one and only one responsibility**.
* Changes in functionality should not impact unrelated features.
* Code should be **modular, reusable, and easy to maintain**.

Example: Book Management System:-

Imagine we are designing a **Book Management System**. A naive approach might be to create a single class that: Stores book details, Prints book details, Saves book details to a database

This **violates SRP** because the class is responsible for multiple things:  
❌ Managing book data  
❌ Handling printing logic  
❌ Dealing with persistence (saving to the database)

//Bad Code

class Book {  
 private String title;  
 private String author;  
  
 public Book(String title, String author) {  
 this.title = title;  
 this.author = author;  
 }  
  
 // This method handles business logic (responsibility 1)  
 public void printBook() {  
 System.*out*.println("Title: " + title + ", Author: " + author);  
 }  
  
 // This method handles persistence (responsibility 2)  
 public void saveToDatabase() {  
 System.*out*.println("Saving book to database...");  
 }  
}

// Good Code  
// Book class - Handles only book data

class Book {  
 private String title;  
 private String author;  
  
 public Book(String title, String author) {  
 this.title = title;  
 this.author = author;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
}

// Separate class for printing  
class BookPrinter {  
 public void printBook(Book book) {  
 System.*out*.println("Title: " + book.getTitle() + ", Author: " + book.getAuthor());  
 }  
}

// Separate class for saving to database  
class BookRepository {  
 public void saveToDatabase(Book book) {  
 System.*out*.println("Saving book to database: " + book.getTitle());  
 }  
}

Why is this better?

* Each class has a single responsibility
* Changes in one part won’t affect unrelated functionalities
* Code is modular, reusable, and easy to test

## Open/Closed Principle (OCP)

**A class should be open for extension but closed for modification.** This means that we should be able to extend a class’s behavior without modifying its existing code. By doing this, we avoid introducing bugs into existing, tested code while still allowing enhancements.

🔹 **Why OCP is important?**

* Enhances maintainability – No need to modify existing classes when adding new features.
* Prevents unintended side effects – Since we don’t modify existing code, there's no risk of breaking functionality.
* Encourages scalability – New features can be added easily by extending the behavior.

Example: E-commerce platform:-

Imagine you are building an e-commerce system that processes payments.

❌ Bad Approach: A PaymentProcessor class handles different payment methods:

//Bad design

class PaymentProcessor {  
 public void processPayment(String paymentType) {  
 if (paymentType.equals("CreditCard")) {  
 System.*out*.println("Processing credit card payment...");  
 } else if (paymentType.equals("PayPal")) {  
 System.*out*.println("Processing PayPal payment...");  
 } else {  
 System.*out*.println("Invalid payment method.");  
 }  
 }  
}

* Every time a new payment method is added (e.g., Google Pay, Apple Pay), we need to modify processPayment().
* This violates OCP because the class is not closed for modification—we must edit the existing logic whenever a new type of payment is introduced.
* Use interfaces or abstract classes to allow extension.

🔹 **Solution using OCP:**

Instead of modifying the PaymentProcessor, we use **polymorphism and abstraction** to extend functionality without modifying the existing code.

// Step 1: Create an abstract Payment class  
abstract class Payment {  
 abstract void process();  
}

// Step 2: Implement different payment types  
class CreditCardPayment extends Payment {  
 @Override  
 void process() {  
 System.*out*.println("Processing credit card payment...");  
 }  
}  
  
class PayPalPayment extends Payment {  
 @Override  
 void process() {  
 System.*out*.println("Processing PayPal payment...");  
 }  
}  
  
class GPay extends Payment {  
 @Override  
 void process() {  
 System.*out*.println("Processing PayPal payment...");  
 }  
}

// Step 3: PaymentProcessor now works with any new payment type without modification  
class PaymentProcessor {  
 public void processPayment(Payment payment) {  
 payment.process(); // Uses polymorphism to process payment  
 }  
}

// Usage  
public class Main {  
 public static void main(String[] args) {  
 PaymentProcessor processor = new PaymentProcessor();  
  
 Payment creditCard = new CreditCardPayment();  
 Payment paypal = new PayPalPayment();  
  
 processor.processPayment(creditCard);  
 processor.processPayment(paypal);  
 }  
}

## Liskov Substitution Principle (LSP)

**"Subtypes must be substitutable for their base types without altering the correctness of the program."** This means that if class B is a subclass of class A, then B should be usable in place of A without unexpected behavior.

🔹 **Why LSP is important?**

* Ensures that inheritance is used correctly and meaningfully.
* Helps avoid breaking polymorphism by preventing unexpected behavior.
* Promotes robust and maintainable code.

Example:

Imagine we are building a Bird Hierarchy for a wildlife simulation

❌ If we replace a Bird object with a Penguin, it violates expectations (penguins can’t fly!).

❌ The fly() method is not applicable to all subclasses of Bird.

❌ If client code calls fly(), it could crash unexpectedly(runtime exception)

//Bad Example  
  
class Bird {  
 void fly() {  
 System.*out*.println("This bird can fly.");  
 }  
}  
  
class Sparrow extends Bird {  
 // Sparrow can fly - fine  
}  
  
class Penguin extends Bird {  
 // But penguins cannot fly!  
 @Override  
 void fly() {  
 throw new UnsupportedOperationException("Penguins cannot fly!");  
 }  
}

🔹 **Solution using LSP:**

We should separate birds that can fly and those that cannot.

// Base interface for all birds  
interface Bird {  
 void eat();  
}  
  
// Separate interface for birds that can fly  
interface FlyingBird extends Bird {  
 void fly();  
}

// Sparrow implements FlyingBird because it can fly  
class Sparrow implements FlyingBird {  
 @Override  
 public void eat() {  
 System.*out*.println("Sparrow is eating.");  
 }  
  
 @Override  
 public void fly() {  
 System.*out*.println("Sparrow is flying.");  
 }  
}  
  
// Penguin implements only Bird, not FlyingBird  
class Penguin implements Bird {  
 @Override  
 public void eat() {  
 System.*out*.println("Penguin is eating.");  
 }  
}

//LSP ensures that subclasses can replace their parent classes without breaking functionality.  
// Correctly designed client code  
class BirdSanctuary {  
 public void letBirdFly(FlyingBird bird) {  
 bird.fly();  
 }  
}

## Interface Segregation Principle (ISP)

It states that **“do not force any client to implement an interface which is irrelevant to them“.** This means that instead of having large, general-purpose interfaces, we should create smaller, more specific interfaces that contain only the methods relevant to a particular client. This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle but specific to interfaces.

🔹 **Difference between LSP and ISP**

| **Principle** | **Focus** | **Problem It Solves** | **Solution** |
| --- | --- | --- | --- |
| **Liskov Substitution Principle (LSP)** | Ensures that subclasses can replace parent classes without breaking behavior. | Inheritance misuse where a subclass changes the behavior of the base class. | Use **correct abstraction** to ensure child classes fully comply with the parent class’s expected behavior. |
| **Interface Segregation Principle (ISP)** | Ensures that clients do not depend on methods they do not use. | Large, “fat” interfaces that force classes to implement unnecessary methods. | **Split interfaces** into smaller, more specific ones so that classes implement only what they need. |

👉 LSP is about subclass relationships, while ISP is about breaking down interfaces properly.

Example: ❌ Bad Example

//Bad example

interface Printer {  
 void print();  
 void scan();  
 void fax();  
}

* A basic printer may support only print().
* A multifunction printer (MFP) may support print(), scan(), and fax().
* Forcing all printers to implement scan() and fax() even if they don’t support these features violates ISP.

🔹 **Solution using ISP:**

Split the interface

// Interface for basic printing functionality  
interface Printer {  
 void print();  
}  
  
// Interface for scanning functionality  
interface Scanner {  
 void scan();  
}  
  
// Interface for fax functionality  
interface Fax {  
 void fax();  
}

// Basic printer only implements Printer  
class BasicPrinter implements Printer {  
 @Override  
 public void print() {  
 System.*out*.println("Printing document...");  
 }  
}

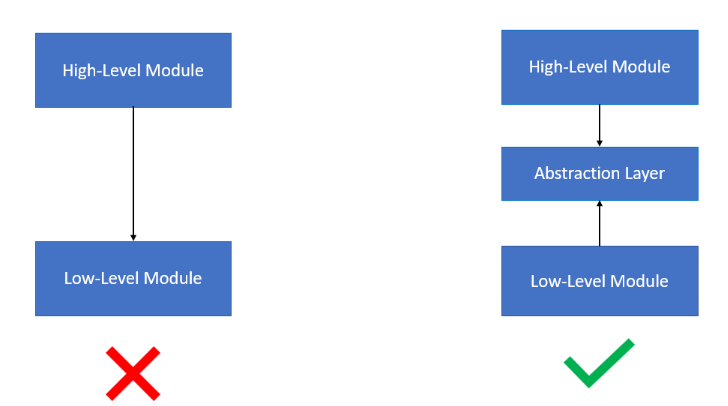
// Multifunction printer implements all three interfaces  
class MultiFunctionPrinter implements Printer, Scanner, Fax {  
 @Override  
 public void print() {  
 System.*out*.println("Printing document...");  
 }  
  
 @Override  
 public void scan() {  
 System.*out*.println("Scanning document...");  
 }  
  
 @Override  
 public void fax() {  
 System.*out*.println("Sending fax...");  
 }  
}

## Dependency Inversion Principle (DIP)

**"High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces or abstract classes)."**

**"Abstractions should not depend on details. Details should depend on abstractions."**

* In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
* This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.



🔹 **Why DIP is important?**

* **Without DIP, changes in low-level modules break high-level modules.**
* **Hard dependencies make it difficult to modify or extend functionality.**
* **Following DIP makes code more maintainable, modular, and adaptable to future changes.**

Bad Example: ❌Payment Processing Example

* OrderService is tightly coupled to PayPalPayment.
* If we want to add StripePayment or GooglePayPayment, we must modify OrderService, breaking the Open/Closed Principle (OCP).
* Testing is difficult because OrderService cannot work with mock payment processors.

class PayPalPayment {  
 void processPayment(double amount) {  
 System.*out*.println("Processing payment of $" + amount + " via PayPal.");  
 }  
}  
  
class OrderService {  
 private PayPalPayment paymentProcessor = new PayPalPayment(); // Direct dependency  
  
 void checkout(double amount) {  
 paymentProcessor.processPayment(amount);  
 }  
}

🔹 **Solution using DIP:**

// Define an abstraction (interface)  
interface PaymentProcessor {  
 void processPayment(double amount);  
}

// Implement PayPal payment processing  
class PayPalPayment implements PaymentProcessor {  
 @Override  
 public void processPayment(double amount) {  
 System.*out*.println("Processing payment of $" + amount + " via PayPal.");  
 }  
}  
// Implement Stripe payment processing  
class StripePayment implements PaymentProcessor {  
 @Override  
 public void processPayment(double amount) {  
 System.*out*.println("Processing payment of $" + amount + " via Stripe.");  
 }  
}

// High-level module depends on abstraction  
class OrderService {  
 private final PaymentProcessor paymentProcessor;  
  
 // Inject dependency via constructor  
 public OrderService(PaymentProcessor paymentProcessor) {  
 this.paymentProcessor = paymentProcessor;  
 }  
 void checkout(double amount) {  
 paymentProcessor.processPayment(amount);  
 }  
}

# **Design Patterns**

Design patterns are proven solutions to common software design problems. They offer best practices to structure and optimize software development, helping developers create scalable, maintainable, and reusable code. They are not strict rules but guidelines that facilitate effective software design.

🔹 **Why use design pattern?**

* **Improve Code Maintainability** - Design patterns encourage a structured and organized approach, making it easier to maintain and modify code over time.
* **Increase Code Reusability** - Patterns allow components to be reused across different projects, reducing redundancy.
* **Enhance Scalability and Flexibility** - Well-structured code ensures that applications can scale efficiently without major changes.
* **Reduce Development Time** - Patterns provide pre-defined solutions, allowing developers to focus on business logic rather than reinventing the wheel.

Types of Design Pattern:

1. **Creational Design Patterns**

Creational patterns deal with the efficient creation of objects while hiding the instantiation logic.

| **Pattern** | **Purpose** | **Example** |
| --- | --- | --- |
| **Singleton** | Ensures only **one instance** of a class exists. | Logger, Database Connection |
| **Factory Method** | Delegates object creation to **subclasses**. | ShapeFactory returning Circle or Rectangle |
| **Abstract Factory** | Creates **related objects** without specifying concrete classes. | GUIFactory producing WindowsButton or MacButton |
| **Builder** | **Step-by-step** creation of complex objects. | MealBuilder for creating VegMeal or NonVegMeal |
| **Prototype** | Creates new objects by **cloning an existing object**. | Document.clone() method |

1. **Structural Design Patterns**

Structural patterns help in organizing classes and objects to create flexible and scalable architectures.

| **Pattern** | **Purpose** | **Example** |
| --- | --- | --- |
| **Adapter** | Converts one interface into another. | USB-to-Ethernet Adapter |
| **Bridge** | Decouples abstraction from implementation. | RemoteControl → TV (SonyRemote, LGRemote) |
| **Composite** | Treats **individual and composite objects** uniformly. | Folder containing Files and SubFolders |
| **Decorator** | Adds functionality **dynamically** without modifying the original class. | BufferedReader wrapping FileReader |
| **Facade** | Provides a **simplified interface** to a complex system. | JDBC Connection |
| **Flyweight** | Shares objects to **reduce memory usage**. | Character Objects in a Text Editor |
| **Proxy** | Controls access to another object. | Virtual Proxy, Security Proxy |

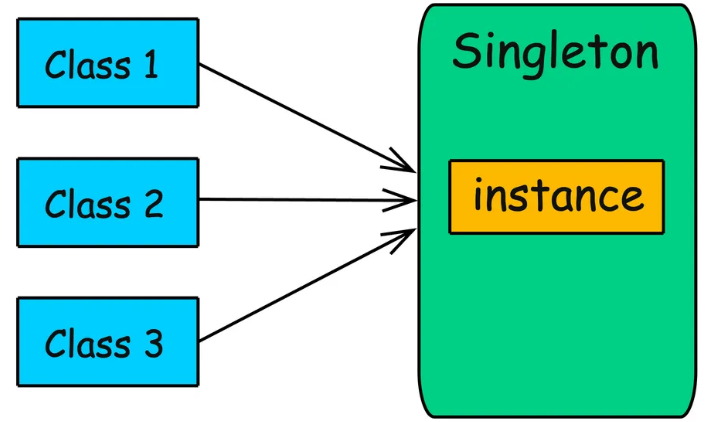
1. **Behavioral Design Patterns**

Behavioral patterns define communication and responsibilities among objects..

| **Pattern** | **Purpose** | **Example** |
| --- | --- | --- |
| **Chain of Responsibility** | Passes requests along a chain of handlers. | Logging System (Error → Warning → Info) |
| **Command** | Encapsulates a request as an object. | Undo/Redo System in Text Editors |
| **Interpreter** | Defines a grammar for language interpretation. | SQL Query Parser |
| **Iterator** | Sequentially accesses elements of a collection. | Java Iterator, Python Itertools |
| **Mediator** | Defines a **central object** for object communication. | Chat Room, Traffic Controller |
| **Memento** | Saves object states for rollback. | Undo feature in MS Word |
| **Observer** | Notifies dependent objects of state changes. | Event Listeners in UI |
| **State** | Allows an object to change behavior based on its state. | Vending Machine States |
| **Strategy** | Defines a family of algorithms and selects one at runtime. | Sorting Algorithms (MergeSort, QuickSort) |
| **Template Method** | Defines the skeleton of an algorithm and allows subclasses to fill in details. | Data Processing Pipelines |
| **Visitor** | Allows adding operations to objects **without modifying their structure**. | AST (Abstract Syntax Tree) Traversal |

# **Singleton Pattern**

The Singleton pattern restricts the instantiation of a class to a single object and provides a way to access it globally. It is used when only one instance of a class is needed to coordinate actions across a system.



🔹 **When to use:**

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.

🔹 **Implementation of Singleton Pattern:**

class Singleton {  
 private static Singleton *instance*;  
  
 //make constructor private, so no other class can create object directly  
 private Singleton() { }  
  
 //The getInstance() method is used to access the single instance  
 public static Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

* **Lazy initialization** in the Singleton pattern ensures that the instance is created only when it is needed for the first time, rather than at the time of class loading
* In a multithreaded environment, the Singleton pattern often uses synchronized methods to ensure that only one instance is created, even if multiple threads attempt to create an instance at the same time.
* To ensure that a Singleton object cannot be cloned, you should override the clone() method and throw CloneNotSupportedException to prevent the creation of multiple instances.

🔹 **Example: Design a logging service that allows multiple parts of an application to log messages but ensures all logs are managed from a single instance**

// The Logger class implements the Singleton Pattern to provide a single point of access for logging messages throughout the application.(Logger.java)  
  
package Singleton;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class Logger {  
  
 private static Logger *instance*;  
 private Logger() {}  
  
 public static synchronized Logger getInstance() {  
 if (*instance* == null) {  
 *instance* = new Logger();  
 }  
 return *instance*;  
 }  
  
 public void info(String message) {  
 log("INFO", message);  
 }  
  
 public void warn(String message) {  
 log("WARN", message);  
 }  
  
 public void error(String message) {  
 log("ERROR", message);  
 }  
  
 private void log(String level, String message) {  
 String timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());  
 System.*out*.println(String.*format*("%s [%s]: %s", timestamp, level, message));  
 }  
}

Class to use it

package Singleton;  
import java.util.Scanner;  
  
public class Exercise {  
 public void run() {  
 Logger logger = Logger.getInstance();  
 Scanner sc = new Scanner(System.*in*);  
  
 //Log the info message using the appropriate logging method.  
 System.*out*.print("Enter an info message: ");  
 String infoMessage = sc.nextLine();  
 logger.info(infoMessage);  
  
 // Log the warn message using the appropriate logging method.  
 System.*out*.print("Enter a warning message: ");  
 String warnMessage = sc.nextLine();  
 logger.warn(warnMessage);  
  
 //Log the error message using the appropriate logging method.  
 System.*out*.print("Enter an error message: ");  
 String errorMessage = sc.nextLine();  
 logger.error(errorMessage);  
  
 sc.close();  
 }  
}

# **Factory Design Pattern**

The Factory Pattern is a creational design pattern that provides a way to create objects without specifying the exact class that should be instantiated.

**👉** 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.

**👉** Key Idea: Instead of using new to create objects in the client code (service call), we delegate object creation to a dedicated method (factory class).

🔹 **Why to use:**

* **Decouples object creation from implementation**
* **Promotes reusability and maintainability**
* **Improves code readability by abstracting instantiation logic**
* **Provides flexibility in object creation (e.g., dynamic selection at runtime)**

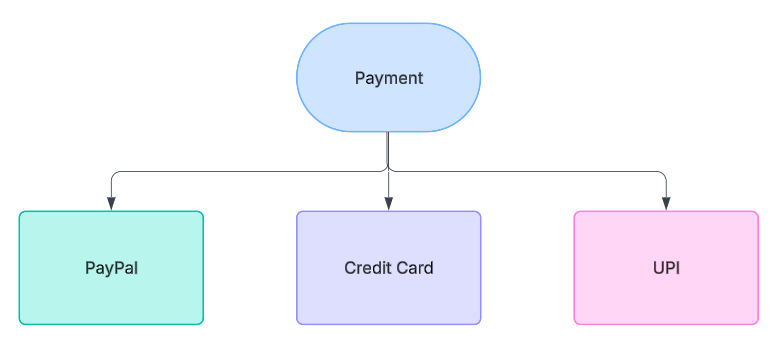
**❌ Without Factory Pattern (Direct Instantiation)**

class PaymentService {  
 void makePayment(String type) {  
 if (type.equals("CreditCard")) {  
 CreditCardPayment payment = new CreditCardPayment();  
 payment.process();  
 } else if (type.equals("PayPal")) {  
 PayPalPayment payment = new PayPalPayment();  
 payment.process();  
 }  
 }  
}

**Problem with above code:**

* Tightly coupled: The PaymentService class (Client code) knows about every payment method.
* Adding new payment methods requires modifying PaymentService (violation of open/close principle)
* Hard to test because objects are directly created inside the method.

🔹 **Implementation with Factory Pattern**



interface Payment {  
 void process();  
}  
  
class CreditCardPayment implements Payment {  
 public void process() {

//Large code to handle credit card payment  
 System.*out*.println("Processing Credit Card Payment");  
 }  
}  
  
class PayPalPayment implements Payment {  
 public void process() {

//Large code to handle Paypal payment  
 System.*out*.println("Processing PayPal Payment");  
 }  
}

//Factory Class (handles object creation)  
//Can use if-else too  
class PaymentFactory {  
 public static Payment getPaymentMethod(String type) {  
 switch (type.toLowerCase()){  
 case "creditcard":  
 return new CreditCardPayment();  
 case "paypal":  
 return new PayPalPayment();  
 default:  
 throw new IllegalArgumentException("Unknown Payment type: " + type);   
 }  
 }  
}

//Service class(client code)  
//in place of taking type input from user, it may be coming via api or from UI input or select field

public class PaymentService {  
 public void makePayment(String type){  
 Payment payment = PaymentFactory.*getPaymentMethod*(type);  
 payment.process();  
 //notification  
 //logging  
 }  
}

//client code  
public class PaymentClient {  
 public static void main(String[] args) {  
 PaymentService ps = new PaymentService();  
 ps.makePayment("creditcard");  
 }  
}

🔹 **Benefits of using Factory Pattern**

* Encapsulation – Object creation logic is centralized in PaymentFactory.
* Easier to extend – Add a new payment method without modifying PaymentService.
* Loose coupling – PaymentService only depends on the Payment interface

**✅** Use the Factory Pattern when:

1. You need multiple implementations of an interface and want to decouple object creation.
2. Object creation is complex and involves configuration settings or dependencies.
3. You want to centralize object creation logic instead of scattering new keywords across your codebase.
4. You need flexibility in choosing an implementation at runtime.

🚫 Avoid Factory Pattern if:

* You have a small number of classes and no need for dynamic object creation.
* Object creation is simple and does not require additional logic.

# **Abstract Factory Design Pattern**

The Abstract Factory Pattern is a creational design pattern that provides an interface for **creating families of related or dependent objects** without specifying their concrete classes..

**👉** Instead of creating objects directly, the client interacts with an abstract factory that provides methods to create different objects.

**👉** This ensures that related objects are created consistently, reducing tight coupling.

🔹 **Why to use:**

* **Encapsulates object creation and provides a consistent way to create related objects.**
* **Decouples client code from the specific classes that implement the objects.**
* **Ensures compatibility among objects created within a family**
* **Easier to extend by adding new factories without modifying existing code.**

**❌ Without Factory Pattern (Direct Instantiation) :**

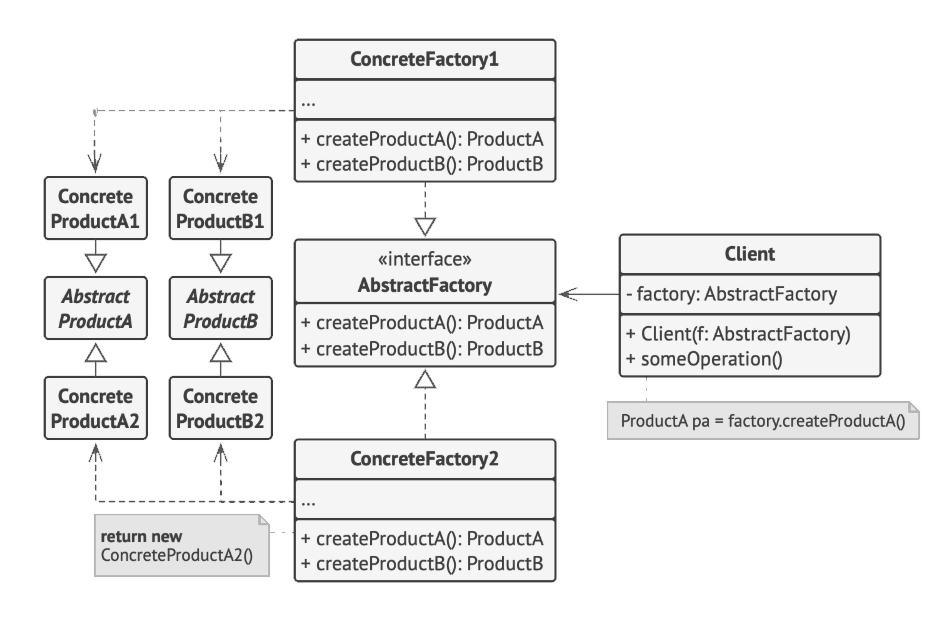
**Imagine we are building a cross-platform UI library that needs to create buttons and checkboxes differently for Windows and MacOS.**

class Application {  
 void createUI(String osType) {  
 if (osType.equals("Windows")) {  
 WindowsButton button = new WindowsButton();  
 WindowsCheckbox checkbox = new WindowsCheckbox();  
 button.render();  
 checkbox.render();  
 } else if (osType.equals("MacOS")) {  
 MacOSButton button = new MacOSButton();  
 MacOSCheckbox checkbox = new MacOSCheckbox();  
 button.render();  
 checkbox.render();  
 }  
 }  
}

**Problem with above code:**

* Code duplication due to if else logic
* Difficult to extend, as if new OS is added, client code needs to be modified
* No code is stopping to create MacOSCheckboc with WindowsButton, which are incompatible with each other.
* Application class directly depends on concrete classes like WindowsButton.

🔹 **Implementation with Abstract Factory Pattern**



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

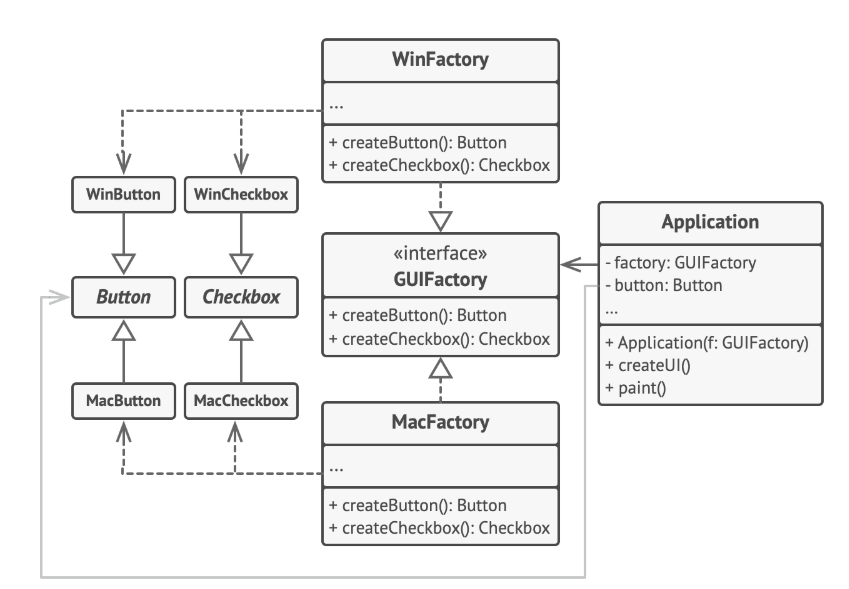
Using the Abstract Factory Pattern, you create interfaces for each product (e.g., Button, ScrollBar, and Window) and provide a family of concrete implementations for each theme (e.g., WindowsButton, MacOSButton, etc.).

// Abstract Button Interface  
interface Button {  
 void render();  
}  
  
// Concrete Implementations for Windows and MacOS  
class WindowsButton implements Button {  
 public void render() {  
 System.*out*.println("Rendering Windows Button");  
 }  
}  
  
class MacOSButton implements Button {  
 public void render() {  
 System.*out*.println("Rendering MacOS Button");  
 }  
}

// Abstract Checkbox Interface  
interface Checkbox {  
 void render();  
}  
  
// Concrete Implementations for Windows and MacOS  
class WindowsCheckbox implements Checkbox {  
 public void render() {  
 System.*out*.println("Rendering Windows Checkbox");  
 }  
}  
  
class MacOSCheckbox implements Checkbox {  
 public void render() {  
 System.*out*.println("Rendering MacOS Checkbox");  
 }  
}

// Abstract Factory Interface  
interface GUIFactory {  
 Button createButton();  
 Checkbox createCheckbox();  
}  
  
// Concrete Factories for Windows and MacOS  
class WindowsFactory implements GUIFactory {  
 public Button createButton() {  
 return new WindowsButton();  
 }  
  
 public Checkbox createCheckbox() {  
 return new WindowsCheckbox();  
 }  
}  
  
class MacOSFactory implements GUIFactory {  
 public Button createButton() {  
 return new MacOSButton();  
 }  
  
 public Checkbox createCheckbox() {  
 return new MacOSCheckbox();  
 }  
}

// Application uses Abstract Factory  
class Application {  
 private Button button;  
 private Checkbox checkbox;  
  
 public Application(GUIFactory factory) {  
 this.button = factory.createButton();  
 this.checkbox = factory.createCheckbox();  
 }  
  
 void renderUI() {  
 button.render();  
 checkbox.render();  
 }  
}  
  
// Client Code  
class Main {  
 public static void main(String[] args) {  
 GUIFactory factory;  
 String osType = "Windows"; // Change to "MacOS" for Mac UI  
  
 if (osType.equals("Windows")) {  
 factory = new WindowsFactory();  
 } else {  
 factory = new MacOSFactory();  
 }  
  
 Application app = new Application(factory);  
 app.renderUI();  
 }  
}

****

🔹 **Benefits of using Factory Pattern**

* Decouples object creation → Application class only depends on GUIFactory, not on specific classes.
* Easy to extend → Adding a new LinuxFactory does not affect existing code.
* Ensures compatibility → A WindowsFactory always provides a Windows button and checkbox, preventing mismatches.

**✅** Use the Factory Pattern when:

1. You need to create related objects that must be used together (e.g., UI components).
2. You want to ensure compatibility between created objects.
3. You need flexibility in choosing between different implementations at runtime.
4. You want to enforce consistency by ensuring that objects from the same family are used together.

🚫 Avoid Factory Pattern if:

1. You don’t need to create families of related objects.
2. Object creation logic is simple and doesn’t require multiple variations.

Difference between Factory and Abstract Factory:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Factory Pattern** | **Abstract Factory Pattern** |
| **Purpose** | Creates objects of a single type | Creates families of related objects |
| **Flexibility** | Less flexible; one factory per type | More flexible; centralizes multiple factories |
| **Scalability** | Limited to a single product type | Supports multiple related products |
| **Example** | Payment processing factory | UI Component factory (Windows/Mac) |

# **Builder Design Pattern**

The Builder Pattern is a creational design pattern that helps in constructing complex objects step by step. It allows you to **separate the construction process from the actual object representation** (following SRP), making object creation more flexible and readable.

**👉** Instead of creating objects with long constructor parameters (telescoping constructors), the Builder pattern provides a fluent API to build objects conveniently.

**❌ Without Builder Pattern (Building a burger)**

class Burger {  
 private String size; //mandatory  
 private String patty; //mandatory  
 private String sauce; //optional  
 private boolean cheese; //optional  
 private boolean tomato; //optional  
   
 // Constructor with many parameters (Telescoping Constructor Problem)  
 public Burger(String size,String patty, String sauce, boolean cheese,boolean tomato) {  
 this.size = size;  
 this.patty = patty;  
 this.sauce = sauce;  
 this.cheese = cheese;  
 this.tomato = tomato;  
 }  
  
 @Override  
 public String toString() {  
 return "Burger{" + "size='" + size + ", patty='" + patty + ", sauce='" + sauce + ", cheese=" + cheese + ", tomato=" + tomato + '}';  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Difficult to understand what parameters mean  
 Burger burger = new Burger("Large", "Aloo", "Mint Mayo", true, false);  
 System.*out*.println(burger);  
 }  
}

**Problem with above code:**

* **Difficult to read** – What does true, false, true, false mean?
* **Unnecessary arguments** – What if the user wants a default burger?
* **Hard to extend** – Adding new ingredients means modifying the constructor.

🔹 **Implementation with Builder Pattern**

class Burger {  
 private String size; //mandatory  
 private String patty; //mandatory  
 private String sauce; //optional  
 private boolean cheese; //optional  
 private boolean tomato; //optional  
  
 // Private constructor to force use of Builder  
 private Burger(BurgerBuilder builder) {  
 this.size = builder.size;  
 this.patty = builder.patty;  
 this.sauce = builder.sauce;  
 this.cheese = builder.cheese;  
 this.tomato = builder.tomato;  
 }  
  
 @Override  
 public String toString() {  
 return "Burger{" + "size='" + size + ", patty='" + patty + ", sauce='" + sauce + ", cheese=" + cheese + ", tomato=" + tomato + '}';  
 }  
  
 // Static Inner Builder Class  
 public static class BurgerBuilder {  
 private String size; //mandatory  
 private String patty; //mandatory  
 private String sauce = "Tamato Sauce"; //giving default value  
 private boolean cheese; //optional, defaults to false  
 private boolean tomato; //optional, , defaults to false  
  
 // Mandatory parameter  
 public BurgerBuilder(String size, String patty) {  
 this.size = size;  
 }

public BurgerBuilder addSauce(String sauce) {  
 this.sauce = sauce;  
 return this;  
 }  
 public BurgerBuilder addCheese(boolean cheese) {  
 this.cheese = cheese;  
 return this;  
 }  
 public BurgerBuilder addTomato(boolean tomato) {  
 this.tomato = tomato;  
 return this;  
 }  
  
 // Build method to create final object  
 public Burger build() {  
 return new Burger(this);  
 }  
 }  
}

//Client code to handle burger creation  
public class Main {  
 public static void main(String[] args) {  
 Burger burger = new Burger.BurgerBuilder("Large","Aloo")  
 .addSauce("Mint Mayo")  
 .addCheese(true) //chaining method call  
 .addTomato(true)  
 .build();  
 System.*out*.println(burger);  
  
 Burger burger2 = new Burger.BurgerBuilder("Small","Chicken")  
 .addTomato(true)  
 .build();  
 }  
}

Explanation: Inside Burger class, we created a nested static class BurgerBuilder, which create the burger. Burger has different set methods, which return BurgerBuilder only, so that while calling, these methods can be chained.

BurgerBuilder also have build method, which at the end calls the private constructor of Burger and returns Burger object.

🔹 **Key Features of builder pattern:**

**✔ Step-by-step object construction**

**✔ Readable and maintainable code**

**✔ Immutability – Objects can be made immutable after creation (by setting values to final)**

**✔ Avoids complex telescoping constructor**

**✔ No need to remember order of values to be set for constructor.**

🔹 **Real world use case:**

HttpRequest request = new HttpRequest.Builder()  
 .setUrl("https://api.example.com")  
 .setMethod("POST")  
 .setHeader("Content-Type", "application/json")  
 .setBody("{ \"name\": \"John\" }")  
 .build();

# **Prototype Design Pattern**

The Prototype Pattern is a creational design pattern that allows **cloning an object instead of creating a new instance from scratch**. It provides a mechanism to create objects by copying an existing object (prototype).

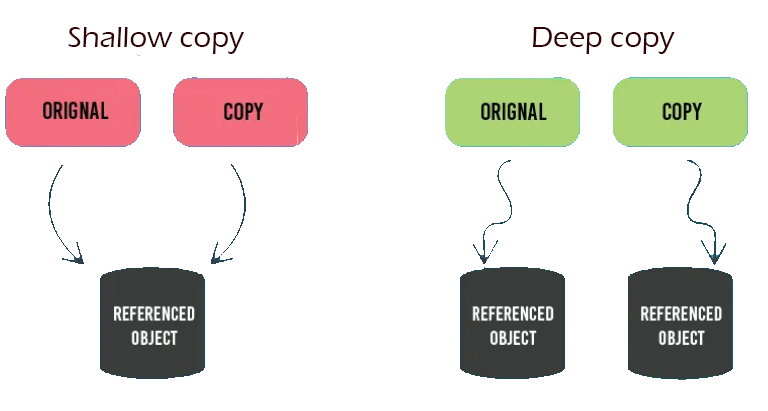
🔹 **When to use Prototype pattern**

| **Scenario** | **Why Prototype Helps** |
| --- | --- |
| Objects are expensive to create | Cloning avoids costly instantiations |
| Need multiple instances of an object with minor changes | Clone the object and modify specific fields |
| Reduce dependency on constructors | Simplifies object creation logic |
| Need to maintain object history (versioning) | Cloning preserves previous states |

🔹 **Shallow Vs Deep copy**

**In most cases, eg for case where we need to store the previous state of anything like a game. So in such cases we need to store deep copy, so that current state don’t change values in previous state.** **By default, Java’s clone() method creates a shallow copy, meaning nested objects are not cloned.**

* **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)**



**❌ Without Prototype Pattern (Storing current state of chess game, for undo redo functionality)**

**Problem with below code:**

* To make deep copy for all pieces, we have to call constructor many times.
* Clone logic is present in client code. Violation of SRP
* Any changes in constructor of piece, will cause changes in client code. Violation of open/closed principle.

public class Piece {  
 private String name;  
 private String position;  
  
 public Piece(String name, String position){  
 this.name = name;  
 this.position = position;  
 }  
   
 public String getName() {  
 return name;  
 }  
  
 public String getPosition() {  
 return position;  
 }  
  
 @Override  
 public String toString() {  
 return "Piece{" + "name='" + name + '\'' + ", position='" + position + '\'' + '}';  
 }  
}

public class ChessBoard {  
 private List<Piece> pieces = new ArrayList<>();  
  
 public void addPiece(Piece piece){  
 pieces.add(piece);  
 }  
  
 public List<Piece> getPieces(){  
 return pieces;  
 }  
  
 public void showBoardState(){  
 for(Piece piece:pieces){  
 System.*out*.println(piece);  
 }  
 }  
}

public class GameWithoutPrototype {  
 public static void main(String[] args) {  
 ChessBoard chessBoard = new ChessBoard();  
 chessBoard.addPiece(new Piece("Rook","a3"));  
 chessBoard.addPiece(new Piece("Bishop","c5"));  
 chessBoard.showBoardState();  
  
 //save the state  
 ChessBoard copiedBoard = new ChessBoard();  
 for(Piece piece: chessBoard.getPieces()){//deep copy  
 copiedBoard.addPiece(new Piece(piece.getName(), piece.getPosition()));  
 }  
   
 System.*out*.println("Copied Board");  
 copiedBoard.showBoardState();  
 }  
}

🔹 **Implementation with Prototype Pattern**

public class Piece {  
 private String name;  
 private String position;  
  
 public Piece(String name, String position){  
 this.name = name;  
 this.position = position;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getPosition() {  
 return position;  
 }  
  
 @Override  
 public String toString() {  
 return "Piece{"+"name='"+ name + '\''+ ",position='"+position+'\''+'}';  
 }  
  
 @Override //creating deep copy of piece  
 public Piece clone() {  
 return new Piece(this.name,this.position);  
 }  
}

public class ChessBoard{  
 private List<Piece> pieces = new ArrayList<>();  
  
 public void addPiece(Piece piece){  
 pieces.add(piece);  
 }  
  
 public List<Piece> getPieces(){  
 return pieces;  
 }  
  
 public void showBoardState(){  
 for(Piece piece:pieces){  
 System.*out*.println(piece);  
 }  
 }  
  
 @Override  
 public ChessBoard clone() {  
 ChessBoard clonedBoard = new ChessBoard();  
 for(Piece piece:pieces){  
 clonedBoard.addPiece(piece.clone());  
 }  
 return clonedBoard;  
 }  
}

public class GameWithPrototype {  
 public static void main(String[] args) {  
 ChessBoard chessBoard = new ChessBoard();  
 chessBoard.addPiece(new Piece("Rook","a3"));  
 chessBoard.addPiece(new Piece("Bishop","c5"));  
 chessBoard.showBoardState();  
  
 //Save this state  
 ChessBoard copiedBoard = chessBoard.clone();  
  
 System.*out*.println("Copied Board");  
 copiedBoard.showBoardState();  
 }  
}

**✅** Key takeaways:

* Prototype Pattern is useful when object creation is expensive.
* Reduces constructor dependency and improves performance.
* Use deep cloning to avoid shared references in complex objects.
* Works well when multiple copies of an object with slight variations are needed.

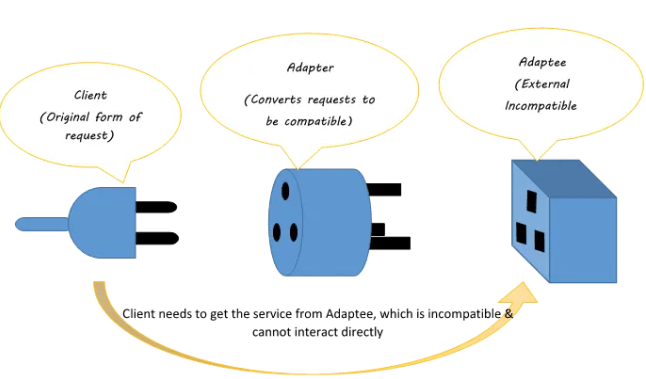
# **Adapter Pattern**

The Adapter Pattern is a **structural design pattern** that **allows incompatible interfaces to work together** by acting as a bridge between them. It enables an existing class with a different interface to be used as if it were a different interface, without modifying its source code.

The Adapter pattern allows for the integration of new interfaces with existing classes, adhering to the Open/Closed Principle by enabling extension of functionality without altering existing code.

🔹 **Working Strategy**

* Convert the interface of a class into another interface that clients expect.
* Bridge the gap between legacy code and new code.
* Facilitate the reuse of existing classes without modifying their implementation.



🔹 **Implementation with Adapter Pattern**

**Scenario:** We have an existing music player that plays only MP3 files. Now, we need to add support for MP4 and VLC files without modifying the existing MediaPlayer class.

**Step 1: Define the Target Interface and create an existing class (client code)**

// Target interface that client expects  
interface MediaPlayer {  
 void play(String audioType, String fileName);  
}  
  
// Existing class that can play only MP3 files  
class MP3Player implements MediaPlayer {  
 @Override  
 public void play(String audioType, String fileName) {  
 if (audioType.equalsIgnoreCase("mp3")) {  
 System.*out*.println("Playing MP3 file: " + fileName);  
 } else {  
 System.*out*.println("Invalid media type: " + audioType);  
 }  
 }  
}

**Step 2: Create the Adaptee ( New classes with different interfaces)**

// Adaptee interface (New functionality that needs to be adapted)  
interface AdvancedMediaPlayer {  
 void playMP4(String fileName);  
 void playVLC(String fileName);  
}  
  
// Concrete classes implementing the new functionality  
class MP4Player implements AdvancedMediaPlayer {  
 @Override  
 public void playMP4(String fileName) {  
 System.*out*.println("Playing MP4 file: " + fileName);  
 }  
  
 @Override  
 public void playVLC(String fileName) {  
 // Do nothing, as this player only supports MP4  
 }  
}  
  
class VLCPlayer implements AdvancedMediaPlayer {  
 @Override  
 public void playMP4(String fileName) {  
 // Do nothing, as this player only supports VLC  
 }  
  
 @Override  
 public void playVLC(String fileName) {  
 System.*out*.println("Playing VLC file: " + fileName);  
 }  
}

**Step 3: Modify the Client code to use the adapter**

// Adapter class that converts AdvancedMediaPlayer to MediaPlayer  
class MediaAdapter implements MediaPlayer {  
 private AdvancedMediaPlayer advancedMediaPlayer;  
  
 public MediaAdapter(String audioType) {  
 if (audioType.equalsIgnoreCase("mp4")) {  
 advancedMediaPlayer = new MP4Player();  
 } else if (audioType.equalsIgnoreCase("vlc")) {  
 advancedMediaPlayer = new VLCPlayer();  
 }  
 }  
  
 @Override  
 public void play(String audioType, String fileName) {  
 if (audioType.equalsIgnoreCase("mp4")) {  
 advancedMediaPlayer.playMP4(fileName);  
 } else if (audioType.equalsIgnoreCase("vlc")) {  
 advancedMediaPlayer.playVLC(fileName);  
 }  
 }  
}

**Step 4: Modify the client code to use the Adapter**

// Client code using the adapter pattern  
class AudioPlayer implements MediaPlayer {  
 private MediaAdapter mediaAdapter;  
  
 @Override  
 public void play(String audioType, String fileName) {  
 if (audioType.equalsIgnoreCase("mp3")) {  
 System.*out*.println("Playing MP3 file: " + fileName);  
 } else if (audioType.equalsIgnoreCase("mp4") || audioType.equalsIgnoreCase("vlc")) {  
 mediaAdapter = new MediaAdapter(audioType);  
 mediaAdapter.play(audioType, fileName);  
 } else {  
 System.*out*.println("Invalid media format: " + audioType);  
 }  
 }  
}

**Step 5: Test the client code**

public class AdapterPatternDemo {  
 public static void main(String[] args) {  
 AudioPlayer audioPlayer = new AudioPlayer();  
  
 audioPlayer.play("mp3", "song.mp3");  
 audioPlayer.play("mp4", "movie.mp4");  
 audioPlayer.play("vlc", "video.vlc");  
 audioPlayer.play("avi", "clip.avi"); // Unsupported format  
 }  
}

🔹 **When to use Adapter Pattern**

| **Scenario** | **Why Adapter Helps** |
| --- | --- |
| Working with legacy systems | Helps integrate older classes with a new API |
| Incompatible interfaces | Allows systems with different interfaces to communicate |
| Third-party library integration | Adapts external libraries to match your system's interface |
| Multiple implementations of the same functionality | Provides a common interface for different implementations |

**✅** Key takeaways:

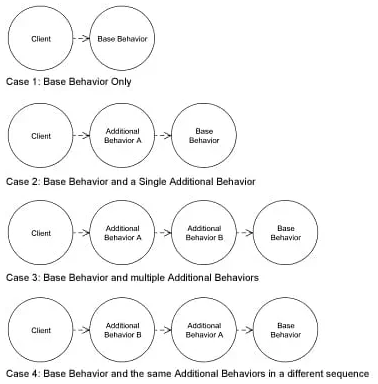
* Adapter Pattern bridges incompatible interfaces without modifying existing code.
* It helps in legacy system integration and third-party API adaptation.
* Works well in multi-technology environments (e.g., different chargers).

# **10. Decorator Pattern**

The Decorator Pattern is a **structural design pattern** that adds new behaviors to objects dynamically without modifying their existing code. Instead of extending a class through inheritance, it relies on **composition** to wrap objects with additional functionality.

🔹**Intent of Decorator Pattern**

* Allows adding new functionalities to an object at runtime.
* Avoids subclass explosion caused by multiple feature combinations.
* Uses wrappers instead of modifying the base class directly.



* In the Decorator pattern, decorators wrap original objects, adding new behavior or responsibilities without altering the original object's code.
* Decorator classes implement the same interface as the components they enhance, allowing clients to use both interchangeably.
* The Decorator pattern allows you to extend an object's functionality at runtime without altering its structure, enabling dynamic behavior enhancements

🔹 **Implementation with Decorator Pattern**

A coffee shop offers different types of coffee. Customers can add milk, sugar, whipped cream, or vanilla flavour. The price should dynamically update based on the selected add-ons.

**Step 1: Define the Base Component**

// Component Interface  
interface Coffee {  
 String getDescription();  
 double getCost();  
}

**Step 2: Implement a Basic Coffee Class**

// Concrete Component: Plain coffee  
class BasicCoffee implements Coffee {  
 @Override  
 public String getDescription() {  
 return "Basic Coffee";  
 }  
  
 @Override  
 public double getCost() {  
 return 50; // Base price  
 }  
}

**Step 3: Create an Abstract Decorator**

// Abstract Decorator  
abstract class CoffeeDecorator implements Coffee {  
 protected Coffee coffee;  
  
 public CoffeeDecorator(Coffee coffee) {  
 this.coffee = coffee;  
 }  
  
 @Override  
 public String getDescription() {  
 return coffee.getDescription();  
 }  
  
 @Override  
 public double getCost() {  
 return coffee.getCost();  
 }  
}

**Step 4: Implement Add-On Decorators**

// Concrete Decorator: Adds Sugar  
class Sugar extends CoffeeDecorator {  
 public Sugar(Coffee coffee) {  
 super(coffee);  
 }  
  
 @Override  
 public String getDescription() {  
 return coffee.getDescription() + ", Sugar";  
 }  
  
 @Override  
 public double getCost() {  
 return coffee.getCost() + 5;  
 }  
}  
  
// Concrete Decorator: Adds Whipped Cream  
class WhippedCream extends CoffeeDecorator {  
 public WhippedCream(Coffee coffee) {  
 super(coffee);  
 }  
  
 @Override  
 public String getDescription() {  
 return coffee.getDescription() + ", Whipped Cream";  
 }  
  
 @Override  
 public double getCost() {  
 return coffee.getCost() + 15;  
 }  
}

**Step 5: Client code**

public class CoffeeShop {  
 public static void main(String[] args) {  
 Coffee coffee = new BasicCoffee();  
 System.*out*.println(coffee.getDescription() + " = $" + coffee.getCost());  
  
 coffee = new Milk(coffee);  
 coffee = new Sugar(coffee);  
 System.*out*.println(coffee.getDescription() + " = $" + coffee.getCost());  
  
 coffee = new WhippedCream(coffee);  
 System.*out*.println(coffee.getDescription() + " = $" + coffee.getCost());  
 }  
}

**Output:**

Basic Coffee = $50.0

Basic Coffee, Milk, Sugar = $65.0

Basic Coffee, Milk, Sugar, Whipped Cream = $80.0

🔹 **When to use Adapter Pattern**

| **Scenario** | **Why Decorator Helps** |
| --- | --- |
| Need to add functionality dynamically | Allows extending objects without modifying their original code |
| Avoid subclass explosion | Replaces multiple subclasses with flexible object composition |
| Open-Closed Principle adherence | Supports extending functionality without modifying existing code |
| Multiple independent features | Allows combining features dynamically at runtime |

🔹 **Key Takeaways**

✔ Adds behavior dynamically without modifying the base class.

✔ Helps avoid subclass explosion by using composition instead of inheritance.

✔ Used in UI frameworks, logging, data compression, and streaming APIs.

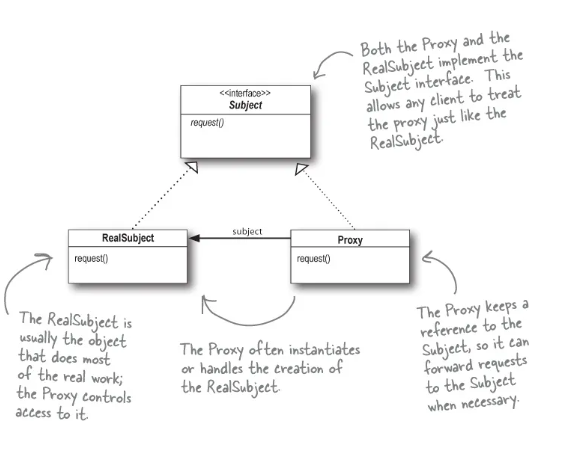
✔ Flexible but can become complex if overused.

# **11. Proxy Pattern**

The Proxy Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. The proxy acts as an intermediary between the client and the real object, adding functionalities like **security, logging, caching**, or **lazy initialization** without modifying the original object.

🔹**Intent of Decorator Pattern**

* Controls access to an object based on certain conditions (e.g., authentication, permissions).
* Adds lazy initialization for resource-intensive objects.
* Implements logging, caching, and request filtering.
* Protects the real object from direct access.



🔹 **Implementation with Proxy Pattern**

Imagine an application that loads high-resolution images. Instead of loading the image immediately, a proxy delays loading until it's actually needed **(Lazy initialization).**

**Step 1: Define a common interface**

// Subject interface  
interface Image {  
 void display();  
}

**Step 2: Implement a Real Object**

// Real Object that loads and displays an image  
class RealImage implements Image {  
 private String fileName;  
  
 public RealImage(String fileName) {  
 this.fileName = fileName;  
 loadFromDisk(); //Heavy operation  
 }  
  
 private void loadFromDisk() {  
 System.*out*.println("Loading image: " + fileName);  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("Displaying image: " + fileName);  
 }  
}

**Step 3: Create a Proxy Object**

// Proxy class that controls access to RealImage  
// Implements same interface as real object  
class ProxyImage implements Image {  
 private RealImage realImage;  
 private String fileName;  
  
 public ProxyImage(String fileName) {  
 this.fileName = fileName;  
 }  
  
 @Override  
 public void display() {  
 if (realImage == null) { //Lazy Loading  
 realImage = new RealImage(fileName); // Load only when needed  
 }  
 realImage.display();  
 }  
}

**Step 4: Client Code**

public class ProxyPatternDemo {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("photo1.jpg");  
 Image image2 = new ProxyImage("photo2.jpg");  
  
 // Image is loaded only when display() is called  
 image1.display();  
 System.*out*.println("------");  
 image1.display(); // This time it is not loaded again  
  
 //Since image2 is never displayed, it is never loaded  
 }  
}

🔹**When and where to use Proxy Pattern**

|  |  |
| --- | --- |
| **Scenario** | **Why Proxy Helps** |
| Restrict access to a resource | Ensures only authorized users can access the object. |
| Lazy Initialization | Delays object creation until it's actually needed (e.g., large database connections). |
| Logging & Monitoring | Keeps track of actions performed on the object. |
| Caching | Stores previous results to optimize performance. |

🔹**Key Takeaways:**

✔ Controls access to objects while adding extra functionalities.

✔ Useful for security, logging, caching, and lazy initialization.

✔ Helps implement network security, database access control, and resource management.

✔ Be careful of unnecessary proxies, as they can slow down performance.

# **12. Composite Pattern**

The Composite Pattern is a structural design pattern that lets you compose objects into tree-like structures to represent part-whole hierarchies. It treats individual objects and compositions of objects uniformly.

🔹**Intent of Composite Pattern**

* Allows clients to interact with individual objects and composite objects in the same way.
* Provides a tree structure to represent complex hierarchies.
* The Composite pattern allows clients to treat individual objects and compositions of objects uniformly, often useful for hierarchical structures like trees.

**❌ Without Composite Pattern (File Management system)**

public class File {  
 private String name;  
  
 public File(String name){  
 this.name = name;  
 }  
  
 public void showDetails(){  
 System.*out*.println("File : " +name);  
 }  
}

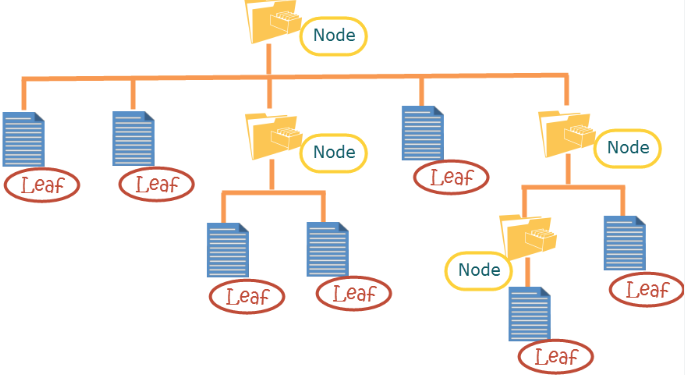
public class Folder {  
 private String name;  
 private List<File> files = new ArrayList<>();  
  
 public Folder(String name){  
 this.name = name;  
 }  
  
 public void addFile(File file){  
 files.add(file);  
 }  
  
 public void showDetails(){  
 System.*out*.println("Folder: " + name);  
 for(File file:files){  
 file.showDetails();  
 }  
 }  
}

**Client code:**

public class FileSystemApp {  
 public static void main(String[] args) {  
 File file1 = new File("File1.txt");  
 File file2 = new File("File2.txt");  
 Folder folder = new Folder("Documents");  
 folder.addFile(file1);  
 folder.addFile(file2);  
  
 folder.showDetails();  
 }  
}

🔹 **Implementation with Composite Pattern**

Issue with above code: For file system we can also have scenarios where we can have sub-folders. To handle that case in above code we will have to write complex if-else conditions. But with Composite pattern, both Files and Folder will be treated uniformly and will implement same interface



**Step 1: Define a common interface**

// Component Interface  
interface FileSystemComponent {  
 void showDetails();  
}

**Step 2: Implement Leaf class(File)**

// Leaf: File (does not contain other components)  
class File implements FileSystemComponent {  
 private String name;  
  
 public File(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void showDetails() {  
 System.*out*.println("File: " + name);  
 }  
}

**Step 3: Implement Composite Class(Folder)**

import java.util.ArrayList;  
import java.util.List;  
  
// Composite: Folder (can contain files and other folders)  
class Folder implements FileSystemComponent {  
 private String name;  
 private List<FileSystemComponent> components = new ArrayList<>();  
  
 public Folder(String name) {  
 this.name = name;  
 }  
  
 public void addComponent(FileSystemComponent component) {  
 components.add(component);  
 }  
  
 @Override   
 public void showDetails() { // DFS (recursive function)  
 System.*out*.println("Folder: " + name);  
 for (FileSystemComponent component : components) {  
 component.showDetails();  
 }  
 }  
}

**Step 4: Client Code**

public class CompositePatternDemo {  
 public static void main(String[] args) {  
 FileSystemComponent file1 = new File("document.txt");  
 FileSystemComponent file2 = new File("photo.jpg");  
  
 Folder folder1 = new Folder("My Documents");  
 folder1.addComponent(file1);  
 folder1.addComponent(file2);  
  
 FileSystemComponent file3 = new File("notes.txt");  
  
 Folder rootFolder = new Folder("Root");  
 rootFolder.addComponent(folder1);  
 rootFolder.addComponent(file3);  
  
 rootFolder.showDetails();  
 }  
}

**Output:**

Folder: Root

Folder: My Documents

File: document.txt

File: photo.jpg

File: notes.txt

🔹**When and Where to use Composite Pattern**

| **Scenario** | **Why Composite Helps** |
| --- | --- |
| **File System (Folders & Files)** | Both files and folders should be handled in the same way. |
| **Organization Hierarchy** | Employees, managers, and departments should be processed uniformly. |
| **UI Components (Buttons, Panels, Containers)** | Allows nesting of UI components inside one another. |