Palash Bajpai

Low Level Design

Table of Contents

[**1. Introduction to Low-Level Design** 2](#_Toc190333069)

[1. System Design requirements 4](#_Toc190333070)

[**2. SOLID Principles** 5](#_Toc190333071)

[a. Single Responsibility Principle 7](#_Toc190333072)

[b. Open/Closed Principle (OCP) 9](#_Toc190333073)

[c. Liskov Substitution Principle (LSP) 11](#_Toc190333074)

[d. Interface Segregation Principle (ISP) 13](#_Toc190333075)

[e. Dependency Inversion Principle (DIP) 15](#_Toc190333076)

[**3.** **Design Patterns** 17](#_Toc190333077)

[**4.** **Singleton Pattern** 18](#_Toc190333078)

[**5.** **Factory Design Pattern** 21](#_Toc190333079)

<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, and various actual logic of different components.

Let’s see a real world example of planning a birthday party and how designing fits in here:

1. **Inviting Guests:**

In software development, inviting guests to a birthday party is similar to inviting users to use your software application. You might send out invitations (marketing emails or social media posts) to let people know about your app and encourage them to try it out.

1. **Choosing a Theme:**

This is like deciding on the overall design and aesthetic of your software application. You might choose a theme (such as minimalist, playful, or professional) that reflects the personality and purpose of your app.

1. **Buying Supplies:**

Buying supplies for a birthday party is akin to acquiring resources and tools for software development. You might purchase software licenses, development kits, or other resources needed to build and test your application.

1. **Setting Up Decorations:**

Setting up decorations for the party is similar to designing the user interface (UI) of your software application. You want to create an attractive and visually appealing interface that enhances the user experience and makes navigation intuitive.

1. **Preparing Food and Drinks:**

This is like implementing the functionality of your software application. You need to develop features and functionalities that fulfill user needs and provide value, whether it's processing transactions, managing data, or facilitating communication.

1. **Entertaining Guests:**

Entertaining guests at a birthday party is akin to providing support and engagement for users of your software application. You want to ensure that users have a positive experience with your app and that their needs are met through timely support, updates, and feedback channels.

Though basic building blocks of all systems are same, namely servers, databases, cache, application, message queue etc. All systems are combined in different ways to serve different users. Let’s see one example how same components help to create different applications and learn about system design in technical aspects.

E-commerce Platform vs. Social Media Platform:

**E-commerce Platform:**

1. In an e-commerce platform like Amazon, servers host the website and handle user requests, while databases store product information, user profiles, and order history.
2. Caches are used to store frequently accessed data such as product listings or user session information, improving performance and reducing latency.
3. Applications manage the business logic, including product search, shopping cart functionality, and payment processing.
4. Message queues are employed for order processing, inventory management, and notifications, ensuring reliability and scalability of backend processes.

**Social Media Platform:**

1. In a social media platform like Facebook, servers host the website and handle user interactions, while databases store user profiles, posts, comments, and media content.
2. Caches are utilized for storing frequently accessed content such as profile information, news feed updates, and image thumbnails, improving responsiveness and user experience.
3. Applications manage complex features such as news feed algorithms, content recommendation systems, and real-time notifications, leveraging machine learning and data analytics.
4. Message queues facilitate real-time communication between users, handling instant messaging, comments, likes, and notifications, ensuring timely delivery and scalability of messaging features.

So here design is important, to use same components but serve different purposes.

## 1. System Design requirements

Categorizing requirements into functional and non-functional types is a common and effective approach. Here's how you can classify different types of requirements:

1. **Non-Functional Requirements:**

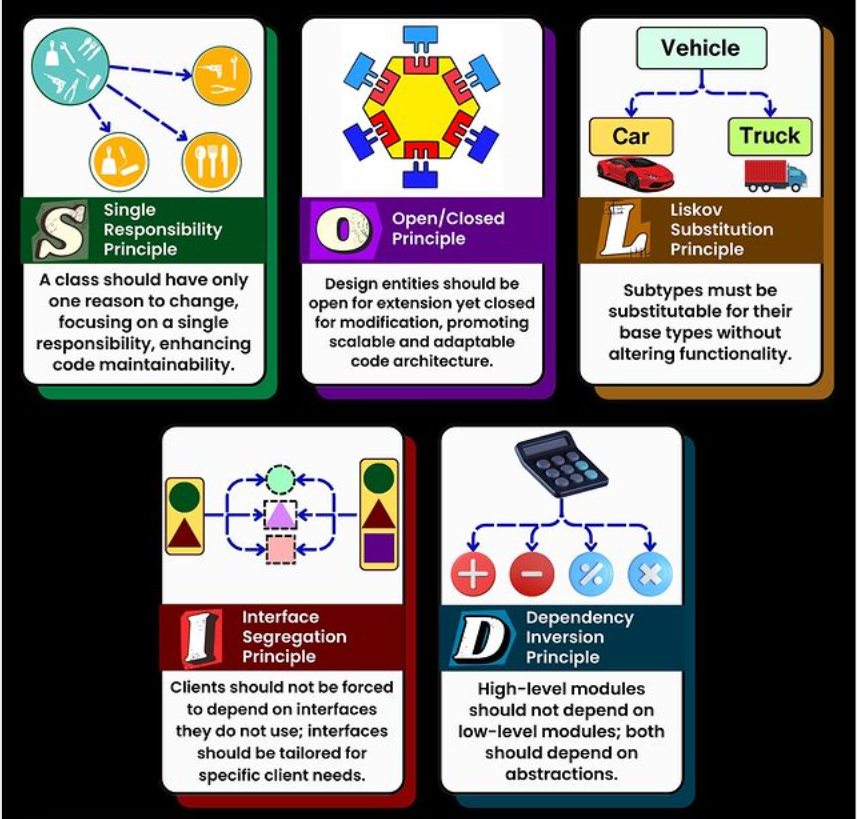
* These specify the quality attributes, constraints, and characteristics that the system must possess, beyond its core functionality.
* Non-functional requirements address aspects such as performance, reliability, security, usability, scalability, maintainability, and compatibility.
* Examples include response time targets, availability goals, security controls, usability guidelines, scalability limits, maintainability standards, and compatibility with external systems or standards.

1. **Functional Requirements:**

* These describe what the system should do and define specific functions, features, and behaviors that the system must exhibit.
* Examples include user authentication, data validation, report generation, transaction processing, and user interface interactions.
* Functional requirements are typically expressed as user stories, use cases, or specific system functionalities.

# **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...");  
 }  
}

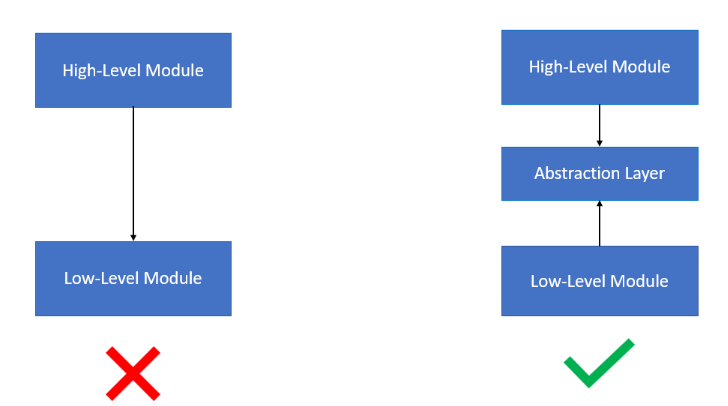
// Multifunction printer implements all three interfaces  
class MultiFunctionPrinter implements Printer, Scanner, Fax {  
 @Override  
 public void print() {  
 System.*out*.println("Printing document...");  
 }  
  
 @Override  
 public void scan() {  
 System.*out*.println("Scanning document...");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

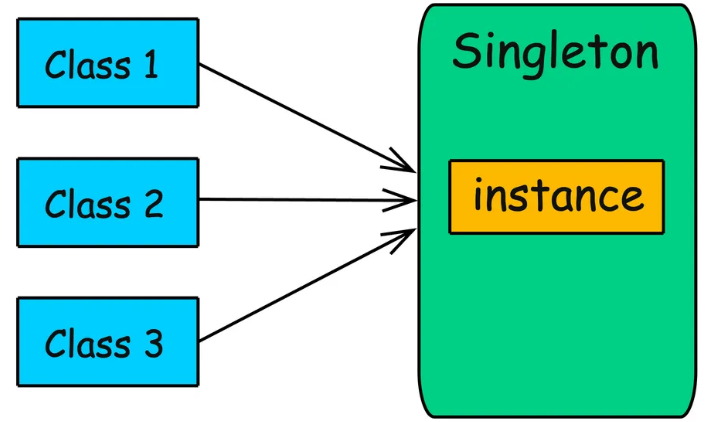
// Implement PayPal payment processing  
class PayPalPayment implements PaymentProcessor {  
 @Override  
 public void processPayment(double amount) {  
 System.*out*.println("Processing payment of $" + amount + " via PayPal.");  
 }  
}  
// Implement Stripe payment processing  
class StripePayment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

# **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 method).

🔹 **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)**

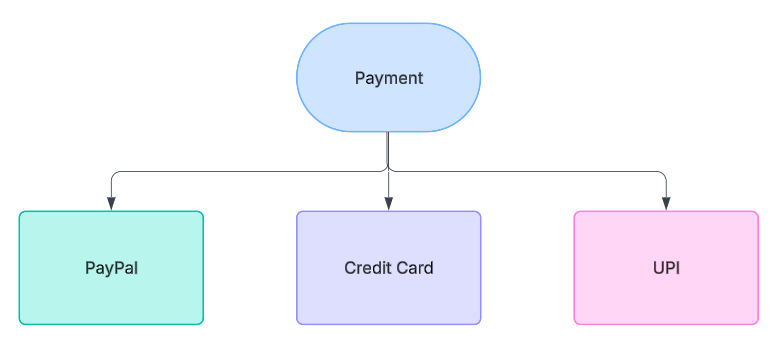
**❌ Without Factory Pattern (Direct Instantiation)**

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() {  
 System.*out*.println("Processing Credit Card Payment");  
 }  
}  
  
class PayPalPayment implements Payment {  
 public void process() {  
 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

class PaymentService {  
 void makePayment(String type) {  
 Scanner scan = new Scanner(System.*in*);  
 String paymentType = scan.next();  
 Payment payment = PaymentFactory.*getPaymentMethod*(paymentType);  
 payment.process();  
 }  
}

🔹 **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.