**LOW LEVEL DESIGN**

**Solid Principle’s**

**SOLID PRINCIPLE’S**

Certainly, Ajay. The **SOLID** principles are the five core principles of object-oriented design and **Low-Level Design (LLD)**. These principles help developers create systems that are easy to maintain, extend, and refactor. They were introduced by **Robert C. Martin (Uncle Bob)** and are widely used in software engineering to write clean, scalable, and robust code.

**🔑 What is SOLID?**

**SOLID** is an acronym:

* **S** – Single Responsibility Principle (SRP)
* **O** – Open/Closed Principle (OCP)
* **L** – Liskov Substitution Principle (LSP)
* **I** – Interface Segregation Principle (ISP)
* **D** – Dependency Inversion Principle (DIP)

Each of these principles tackles a specific design issue.

**1️⃣ Single Responsibility Principle (SRP)**

A class should have only **one reason to change**.

**✅ Meaning:**

A class should focus on doing **only one job**. If it’s doing multiple things, it becomes harder to modify, test, and understand.

**❌ Bad Example:**

class Invoice {

public void calculateTotal() { /\* logic \*/ }

public void printInvoice() { /\* printing logic \*/ }

public void saveToDatabase() { /\* DB logic \*/ }

}

This class has **three responsibilities**: calculation, printing, and persistence.

**✅ Good Example:**

class Invoice {

public void calculateTotal() { /\* logic \*/ }

}

class InvoicePrinter {

public void print(Invoice invoice) { /\* printing logic \*/ }

}

class InvoiceRepository {

public void save(Invoice invoice) { /\* DB logic \*/ }

}

Each class has a single reason to change.

**2️⃣ Open/Closed Principle (OCP)**

Software entities (classes, modules, functions) should be **open for extension** but **closed for modification**.

**✅ Meaning:**

You should be able to add new functionality without modifying existing code.

**❌ Bad Example:**

class Payment {

public void makePayment(String type) {

if (type.equals("credit")) {

// credit payment logic

} else if (type.equals("debit")) {

// debit payment logic

}

}

}

Adding new payment types requires modifying this class.

**✅ Good Example (Using Polymorphism):**

interface Payment {

void pay();

}

class CreditCardPayment implements Payment {

public void pay() { /\* credit logic \*/ }

}

class DebitCardPayment implements Payment {

public void pay() { /\* debit logic \*/ }

}

class PaymentService {

public void makePayment(Payment payment) {

payment.pay();

}

}

You can add new types by extending Payment without modifying PaymentService.

**3️⃣ Liskov Substitution Principle (LSP)**

Subtypes must be substitutable for their base types without altering the correctness of the program.

**✅ Meaning:**

Objects of a subclass should behave in the same way as objects of the superclass.

**❌ Bad Example:**

class Bird {

void fly() {}

}

class Ostrich extends Bird {

void fly() {

throw new UnsupportedOperationException();

}

}

Here, Ostrich is a Bird, but it **cannot fly** – violating LSP.

**✅ Good Example:**

Use composition or a better hierarchy:

interface Bird {}

interface FlyingBird extends Bird {

void fly();

}

class Sparrow implements FlyingBird {

public void fly() {}

}

class Ostrich implements Bird {

// no fly method

}

Now, only flying birds implement fly.

**4️⃣ Interface Segregation Principle (ISP)**

No client should be forced to depend on methods it does not use.

**✅ Meaning:**

Don’t create **fat interfaces**. Instead, prefer **multiple smaller interfaces**.

**❌ Bad Example:**

interface Worker {

void work();

void eat();

}

class Robot implements Worker {

public void work() {}

public void eat() {} // irrelevant

}

**✅ Good Example:**

interface Workable {

void work();

}

interface Eatable {

void eat();

}

class HumanWorker implements Workable, Eatable {

public void work() {}

public void eat() {}

}

class RobotWorker implements Workable {

public void work() {}

}

Clients now implement only what they need.

**5️⃣ Dependency Inversion Principle (DIP)**

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

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

**✅ Meaning:**

Instead of tightly coupling classes, depend on interfaces or abstract classes.

**❌ Bad Example:**

class MySQLDatabase {

public void save(String data) { /\* save logic \*/ }

}

class Application {

private MySQLDatabase db = new MySQLDatabase();

public void saveData(String data) {

db.save(data);

}

}

Application is tightly coupled with MySQLDatabase.

**✅ Good Example:**

interface Database {

void save(String data);

}

class MySQLDatabase implements Database {

public void save(String data) { /\* save logic \*/ }

}

class Application {

private Database db;

public Application(Database db) {

this.db = db;

}

public void saveData(String data) {

db.save(data);

}

}

Now you can use different databases (like MongoDB) without changing the Application class.

**✅ Benefits of SOLID in LLD:**

* Easier to **maintain**
* Easier to **test**
* Promotes **reusability**
* Encourages **clean architecture**
* Reduces **tight coupling**

**✅ Real-World Analogy**

Let’s say you have a **Restaurant System**:

| **Feature** | **SOLID Principle Applied** |
| --- | --- |
| Menu class handles food, print, and billing | Violates SRP – split into Menu, Printer, and BillingService |
| Adding a new delivery type requires modifying existing logic | Violates OCP – use Delivery interface with multiple implementations |
| “Staff” base class has a method cookFood(), and cleaner subclass throws error | Violates LSP – cleaner shouldn’t be forced to cook |
| A Worker interface with methods clean(), cook(), serve() | Violates ISP – split into separate interfaces |
| Restaurant class creates an instance of MySQLService | Violates DIP – should depend on DatabaseService interface |

If you'd like, I can provide **Java code examples for each SOLID principle** using a consistent theme (like a ride-hailing app, library system, or food delivery system). Just say the word!

**HIGH LEVEL MODULE AND LOW LEVEL MODULE**

In software design, especially when discussing modular or layered architecture, the terms high-level module and low-level module refer to different layers of abstraction and responsibility in the system.

🔹 High-Level Module

* Definition: A high-level module is a part of the system that contains business logic, policies, or high-level operations.
* Responsibility: Defines what the system does.
* Abstraction Level: High (more abstract).
* Examples:
  + A PaymentService class that manages how payments are processed.
  + A UserManager class that handles user account operations.
* Key Point: It should be independent of the technical details of how tasks are performed.

🔹 Low-Level Module

* Definition: A low-level module performs specific tasks or operations—usually involving technical or hardware-related details.
* Responsibility: Defines how the system does something.
* Abstraction Level: Low (closer to hardware or technical details).
* Examples:
  + A MySQLDatabaseConnector class that interacts directly with the database.
  + A SendEmailSMTP class that sends emails via SMTP.
* Key Point: It contains the implementation detail for services or functionalities.

🔄 Relation and Dependency (as per Dependency Inversion Principle - part of SOLID):

* Traditionally: High-level modules depend on low-level modules.
* Better design (as per DIP): Both should depend on abstractions (interfaces) so that high-level modules don’t break when low-level modules change.

💡 Real-World Analogy:

Imagine a car:

* High-level module: The driver interface (steering wheel, pedals)—the user decides *what* to do (go forward, turn, stop).
* Low-level module: The engine, transmission, brakes—they perform *how* the action is done.

✅ Summary Table:

| Feature | High-Level Module | Low-Level Module |
| --- | --- | --- |
| Abstraction | High | Low |
| Focus | What to do (logic/flow) | How to do (implementation) |
| Example | OrderProcessor, AuthService | EmailSender, DBConnector |
| Should depend on | Abstractions (Interfaces) | Abstractions (Interfaces) |

**🔹 What is the Single Responsibility Principle (SRP)?**

**✅ Definition (as per Robert C. Martin - “Uncle Bob”):**

"**A class should have one, and only one, reason to change.**"

**🔍 In Simple Words:**

Each class, module, or function should do **only one job** and do it **well**. If a class is responsible for **more than one thing**, it becomes harder to maintain, test, and modify.

**📦 Real-World Analogy**

Imagine a **restaurant manager** who:

* Takes customer orders.
* Cooks the food.
* Serves the dish.
* Handles billing.
* Cleans the tables.

That’s **too many responsibilities** for one person!

🟢 Better: Assign **one responsibility per role**:

* **Chef** → cooks food
* **Waiter** → serves dishes
* **Cashier** → handles payments

This makes the system manageable, efficient, and scalable.

**🚫 Bad Example: Violating SRP**

**Scenario: A Report class that handles report data, formatting, and printing**

class Report {

private String title;

private String content;

public Report(String title, String content) {

this.title = title;

this.content = content;

}

public String getContent() {

return title + "\n" + content;

}

// ❌ Mixing logic

public String formatReport() {

return "\*\*\* " + title.toUpperCase() + " \*\*\*\n" + content;

}

public void printReport() {

System.out.println(formatReport());

}

}

**❌ Problems:**

* Report has **3 responsibilities**: managing data, formatting, and printing.
* If formatting logic changes, this class must change.
* If printing logic changes, this class must also change.
* Violates SRP → harder to test and maintain.

**✅ Good Example: Applying SRP Properly**

**✅ Break into separate classes:**

* Report: holds data
* ReportFormatter: handles formatting
* ReportPrinter: handles printing

// Responsibility 1: Hold data

class Report {

private String title;

private String content;

public Report(String title, String content) {

this.title = title;

this.content = content;

}

public String getTitle() { return title; }

public String getContent() { return content; }

}

// Responsibility 2: Format report

class ReportFormatter {

public String format(Report report) {

return "\*\*\* " + report.getTitle().toUpperCase() + " \*\*\*\n" + report.getContent();

}

}

// Responsibility 3: Print report

class ReportPrinter {

public void print(String formattedReport) {

System.out.println(formattedReport);

}

}

// Main usage

public class Main {

public static void main(String[] args) {

Report report = new Report("Sales Report", "Total sales: $5000");

ReportFormatter formatter = new ReportFormatter();

ReportPrinter printer = new ReportPrinter();

String formatted = formatter.format(report);

printer.print(formatted);

}

}

**✅ Benefits:**

* Each class has **only one reason to change**.
* **Report** changes only if the data model changes.
* **Formatter** changes if formatting rules change.
* **Printer** changes if output device or method changes (e.g., to PDF or file).
* Easy to **test**, **maintain**, and **extend**.

**💡 More Real-World Examples**

**✅ Example 1: Bank Account**

**❌ Bad Design (Violates SRP)**

class BankAccount {

double balance;

public void deposit(double amount) {

balance += amount;

}

public void printStatement() {

System.out.println("Balance: " + balance);

}

}

**✅ Good Design (SRP)**

class BankAccount {

double balance;

public void deposit(double amount) {

balance += amount;

}

public double getBalance() {

return balance;

}

}

class StatementPrinter {

public void print(double balance) {

System.out.println("Balance: " + balance);

}

}

**🧠 Summary**

| **Aspect** | **Explanation** |
| --- | --- |
| 📖 **Principle** | A class should have one reason to change |
| 🔄 **Reason to Change** | Responsibility = Reason to change |
| 🚫 **Bad** | Mixing unrelated behavior (data + logic + output) |
| ✅ **Good** | Each class/module has one focused job |
| 🛠️ **Benefits** | Easy maintenance, readability, testability, reuse |

Would you like a **UML diagram**, **JavaScript/Python version**, or an **example using design patterns** like Factory or Strategy next?

**🔷 What is the Open/Closed Principle?**

**✅ Definition (by Bertrand Meyer):**

"**Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.**"

**🔍 In Simple Terms:**

You should be able to **add new features or behavior** to a class **without modifying its existing code**.  
Instead of editing existing classes, you **extend them** using abstraction like interfaces or inheritance.

**🍽️ Real-World Analogy: Restaurant Menu**

Imagine a restaurant system that calculates discounts:

* At first, there's a **"Flat 10%"** discount.
* Later, we add:
  + **"Buy 1 Get 1"**
  + **"Festival 20%"**

If your original discount logic is tightly coupled in one class, you’ll **keep modifying** it every time a new offer is added → 🔥 risk of breaking old logic.

🟢 Instead, if you design it to **plug in new discount types**, the original logic stays untouched. That's **OCP in action**.

**🚫 Bad Example: Violating Open/Closed Principle**

**❌ One class keeps getting modified for every new feature**

class DiscountCalculator {

public double calculateDiscount(String discountType, double amount) {

if (discountType.equals("FLAT10")) {

return amount \* 0.10;

} else if (discountType.equals("FESTIVAL20")) {

return amount \* 0.20;

} else if (discountType.equals("BOGO")) {

return amount \* 0.50;

}

return 0;

}

}

**❌ Problems:**

* Every time a new discount is added, you must **modify** this class.
* **Violates OCP** — Not closed for modification.
* Leads to **code bloat** and **regression bugs**.

**✅ Good Example: Following Open/Closed Principle**

**✅ Solution: Use Abstraction + Polymorphism**

// Common interface

interface Discount {

double apply(double amount);

}

// Implementations

class Flat10Discount implements Discount {

public double apply(double amount) {

return amount \* 0.10;

}

}

class Festival20Discount implements Discount {

public double apply(double amount) {

return amount \* 0.20;

}

}

class BuyOneGetOneDiscount implements Discount {

public double apply(double amount) {

return amount \* 0.50;

}

}

// Calculator uses abstraction

class DiscountCalculator {

public double calculate(Discount discount, double amount) {

return discount.apply(amount);

}

}

// Client code

public class Main {

public static void main(String[] args) {

DiscountCalculator calculator = new DiscountCalculator();

Discount flat = new Flat10Discount();

Discount fest = new Festival20Discount();

Discount bogo = new BuyOneGetOneDiscount();

System.out.println("Flat 10: " + calculator.calculate(flat, 100));

System.out.println("Festival 20: " + calculator.calculate(fest, 100));

System.out.println("BOGO: " + calculator.calculate(bogo, 100));

}

}

**✅ Benefits:**

| **Point** | **Benefit** |
| --- | --- |
| 🔄 **Open for extension** | You can add new discount types without touching existing code |
| 🛠️ **Closed for modification** | Existing classes like DiscountCalculator remain unchanged |
| 🔌 **Plug-and-play logic** | New strategies can be added like plugins |
| 🧪 **Easier to test and debug** | Isolated changes reduce risk |

**🔍 Another Real-World Example: Notification System**

**❌ Without OCP**

class NotificationService {

public void sendNotification(String type, String message) {

if (type.equals("EMAIL")) {

// send email

} else if (type.equals("SMS")) {

// send SMS

} else if (type.equals("PUSH")) {

// send push notification

}

}

}

**✅ With OCP**

interface Notification {

void send(String message);

}

class EmailNotification implements Notification {

public void send(String message) {

System.out.println("Email: " + message);

}

}

class SMSNotification implements Notification {

public void send(String message) {

System.out.println("SMS: " + message);

}

}

class PushNotification implements Notification {

public void send(String message) {

System.out.println("Push: " + message);

}

}

class NotificationService {

public void send(Notification notification, String message) {

notification.send(message);

}

}

✔️ Add new notification types (e.g., WhatsApp) **without modifying** existing code.

**🧠 Summary**

| **Aspect** | **Explanation** |
| --- | --- |
| 📖 Principle | Classes should be open for extension, closed for modification |
| ❌ Bad Code | Uses if-else or switch to handle variations |
| ✅ Good Code | Uses polymorphism (interfaces/inheritance) to handle new features |
| ⚙️ How | Define interfaces, implement different behaviors in subclasses |
| 🧪 Benefit | Flexible, testable, robust, and future-proof design |

Would you like me to extend this to use **Factory Pattern**, **Enum Strategy**, or show a **C++ version** as well?

**🔷 What is Liskov Substitution Principle (LSP)?**

**✅ Definition (by Barbara Liskov):**

"**Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.**"

**🔍 In Simple Terms:**

If class S is a subclass of class T, then you should be able to use an object of type S in place of T **without breaking the program**.

So,

* Subclasses must **honor the behavior** of their parent class.
* They should not **violate expected behavior**, throw errors, or change outcomes.

**🍽️ Real-World Analogy: Bird Example**

Suppose we have a class Bird with a method fly().

class Bird {

void fly() {

System.out.println("Bird is flying");

}

}

If we now create:

class Sparrow extends Bird {

void fly() {

System.out.println("Sparrow is flying");

}

}

Substituting Sparrow where a Bird is expected will work just fine. ✅ This **follows LSP**.

But if we create:

class Ostrich extends Bird {

void fly() {

throw new UnsupportedOperationException("Ostrich can't fly");

}

}

Now substituting Ostrich for Bird and calling fly() will cause an error. ❌ This **violates LSP**.

**🔥 Why LSP Matters**

Violating LSP causes:

* **Unexpected behavior** at runtime
* **Errors in polymorphic code**
* **Loss of trust** in base class contracts

**🚫 Bad Example: LSP Violation**

**❌ Problem: Subclass alters behavior drastically**

class Bird {

void fly() {

System.out.println("Bird is flying");

}

}

class Ostrich extends Bird {

@Override

void fly() {

throw new UnsupportedOperationException("Ostrich can't fly");

}

}

public class Main {

public static void makeBirdFly(Bird bird) {

bird.fly();

}

public static void main(String[] args) {

Bird ostrich = new Ostrich();

makeBirdFly(ostrich); // Runtime Exception!

}

}

**❌ Why It Violates LSP:**

* Bird promises fly() is safe.
* Ostrich breaks that promise by throwing an exception.

**✅ Good Example: Respecting LSP**

**🛠 Refactor with Interfaces (use behavior-specific design)**

interface Bird {

void eat();

}

interface FlyingBird extends Bird {

void fly();

}

class Sparrow implements FlyingBird {

public void eat() {

System.out.println("Sparrow eats seeds.");

}

public void fly() {

System.out.println("Sparrow flies.");

}

}

class Ostrich implements Bird {

public void eat() {

System.out.println("Ostrich eats plants.");

}

}

public class Main {

public static void feedBird(Bird bird) {

bird.eat();

}

public static void flyBird(FlyingBird bird) {

bird.fly();

}

public static void main(String[] args) {

Bird sparrow = new Sparrow();

Bird ostrich = new Ostrich();

feedBird(sparrow); // Works

feedBird(ostrich); // Works

flyBird((FlyingBird) sparrow); // Safe

// flyBird((FlyingBird) ostrich); ❌ Not allowed at compile time!

}

}

✔️ No runtime errors  
✔️ Each class behaves exactly as expected  
✔️ Client code is protected from invalid assumptions

**🏦 Another Real-World Example: Bank Accounts**

**❌ LSP Violation Example**

class Account {

public void withdraw(double amount) {

System.out.println("Withdrawn: " + amount);

}

}

class FixedDepositAccount extends Account {

@Override

public void withdraw(double amount) {

throw new UnsupportedOperationException("Withdrawals not allowed!");

}

}

Calling withdraw() on a FixedDepositAccount will break the system.

**✅ LSP-Compliant Refactor**

interface Withdrawable {

void withdraw(double amount);

}

class SavingsAccount implements Withdrawable {

public void withdraw(double amount) {

System.out.println("Withdrawn: " + amount);

}

}

class FixedDepositAccount {

// No withdraw method at all

}

Now, only the right accounts are allowed to be withdrawn from, and the base type doesn't promise unsupported behavior.

**🧠 LSP Checklist**

| **Rule** | **Good Practice** |
| --- | --- |
| ✅ Don't override a method to throw unexpected exceptions |  |
| ✅ Don't change input/output contracts |  |
| ✅ Don’t remove base class behavior |  |
| ✅ Subclass should enhance, not break, superclass behavior |  |

**🎯 Summary**

| **Aspect** | **Description** |
| --- | --- |
| 📖 Principle | Subclasses must be substitutable for their base classes |
| 🚫 Violation | Subclass breaks or changes base behavior (e.g., throws exception) |
| ✅ Compliance | Subclass behaves according to expectations |
| 🎁 Benefit | Predictable polymorphism, safe substitutions |
| 🛠 Fix | Use composition, interfaces, or better hierarchy |

**✅ LSP in One Line:**

**"If it looks like a duck, quacks like a duck, but throws an exception when asked to swim — it’s not a duck."**

Would you like to see this applied in **real-world projects** like e-commerce, ride-sharing, or ATM systems with UML diagrams and patterns?

**🔷 What is Interface Segregation Principle (ISP)?**

**✅ Definition (by Robert C. Martin):**

"**Clients should not be forced to depend on interfaces they do not use.**"

**🔍 In Simple Terms:**

Instead of having **one fat interface** with lots of methods, **split** it into smaller, **role-specific interfaces**.  
So that **implementing classes only need to implement what they actually use**.

**🍽️ Real-World Analogy: Multifunction Device**

Imagine you have a **Multifunction Printer Interface**:

interface Machine {

void print(Document d);

void scan(Document d);

void fax(Document d);

}

Now, a simple **InkJetPrinter** that can only print is forced to implement scan() and fax() too—**even if it doesn’t support them**.

**🚫 Bad Example: Violates ISP**

interface Machine {

void print(Document d);

void scan(Document d);

void fax(Document d);

}

class OldPrinter implements Machine {

public void print(Document d) {

System.out.println("Printing document...");

}

public void scan(Document d) {

throw new UnsupportedOperationException("Scan not supported");

}

public void fax(Document d) {

throw new UnsupportedOperationException("Fax not supported");

}

}

**❌ Problems:**

* OldPrinter has to implement methods it **doesn’t need**.
* Throws runtime exceptions → **dangerous**
* Code becomes **brittle**, **confusing**, and **hard to maintain**

**✅ Good Example: Follows ISP**

**➤ Break the fat interface into smaller ones:**

interface Printer {

void print(Document d);

}

interface Scanner {

void scan(Document d);

}

interface Fax {

void fax(Document d);

}

Now implement only what is needed:

class OldPrinter implements Printer {

public void print(Document d) {

System.out.println("Printing from OldPrinter...");

}

}

class AdvancedPrinter implements Printer, Scanner, Fax {

public void print(Document d) {

System.out.println("Printing...");

}

public void scan(Document d) {

System.out.println("Scanning...");

}

public void fax(Document d) {

System.out.println("Faxing...");

}

}

**✅ Benefits:**

* Classes implement only what they **actually support**
* Reduces **side effects**
* Improves **code readability**, **testability**, and **maintainability**

**🏦 Real-World Example: Online Payment Gateway**

**❌ Bad Design (Fat Interface)**

interface PaymentGateway {

void payByCreditCard();

void payByUPI();

void payByWallet();

}

class UPIOnlyApp implements PaymentGateway {

public void payByCreditCard() {

throw new UnsupportedOperationException();

}

public void payByUPI() {

System.out.println("Paid via UPI");

}

public void payByWallet() {

throw new UnsupportedOperationException();

}

}

**✅ Good Design (Interface Segregation)**

interface CreditCardPayment {

void payByCreditCard();

}

interface UPIPayment {

void payByUPI();

}

interface WalletPayment {

void payByWallet();

}

class UPIOnlyApp implements UPIPayment {

public void payByUPI() {

System.out.println("Paid via UPI");

}

}

class FullPaymentApp implements UPIPayment, CreditCardPayment, WalletPayment {

public void payByCreditCard() {

System.out.println("Paid via Credit Card");

}

public void payByUPI() {

System.out.println("Paid via UPI");

}

public void payByWallet() {

System.out.println("Paid via Wallet");

}

}

✔️ Now each class only depends on what it actually uses.

**✅ When to Use Interface Segregation**

Use ISP when:

* You see classes **throwing unimplemented methods**
* An interface has **too many unrelated responsibilities**
* Clients are **forced to depend** on unused methods

**🧠 Summary**

| **Concept** | **Description** |
| --- | --- |
| 📖 Principle | Clients should not be forced to implement unused methods |
| ❌ Violation | Classes throw UnsupportedOperationException |
| ✅ Compliance | Split interfaces into smaller, focused roles |
| 🧰 Tools | Interface segregation + composition |
| 🎁 Benefits | Maintainable, testable, and cleaner code |

**📌 TL;DR**

**"Many client-specific interfaces are better than one general-purpose interface."**

Would you like me to explain how **Interface Segregation** is used in **design patterns** (e.g., Strategy, Adapter), or in **large-scale systems** like microservices or e-commerce?

**🔷 Dependency Inversion Principle (DIP) – Definition**

**Definition:**

High-level modules should not depend on low-level modules. Both should depend on **abstractions**.  
Abstractions should not depend on details. Details should depend on **abstractions**.

**✅ Essence of DIP:**

* High-level modules → represent **business logic** (e.g., OrderService)
* Low-level modules → represent **implementation details** (e.g., EmailSender, DatabaseWriter)
* Instead of high-level depending directly on low-level, **both should rely on an interface or abstract class**.

**🔸 Real-World Analogy**

Imagine a **universal power adapter**:

* **High-level module:** Your **Laptop**
* **Low-level module:** Different types of **power sockets** (US, UK, EU)
* Your laptop shouldn't need to change for each type of socket — it uses a **universal adapter interface**.
* DIP ensures **flexibility**: You can change the low-level power source without modifying the laptop.

**🧑‍💼 Real-World Use Case: Notification System**

* Suppose you're building an **AlertService** that can send messages via **Email, SMS, or Slack**.
* You don’t want to tightly couple the business logic to any specific messaging service.

**❌ Without DIP (Bad Design)**

class EmailService {

public void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

}

class Notification {

private EmailService emailService;

public Notification() {

emailService = new EmailService();

}

public void notifyUser(String message) {

emailService.sendEmail(message);

}

}

**Problems:**

* Notification is tightly coupled to EmailService
* Want to switch to SMSService? Modify Notification → violates **Open/Closed Principle** too

**✅ With DIP (Good Design)**

// Abstraction

interface MessageService {

void sendMessage(String message);

}

// Low-level modules

class EmailService implements MessageService {

public void sendMessage(String message) {

System.out.println("Email: " + message);

}

}

class SMSService implements MessageService {

public void sendMessage(String message) {

System.out.println("SMS: " + message);

}

}

// High-level module

class Notification {

private MessageService messageService;

public Notification(MessageService messageService) {

this.messageService = messageService;

}

public void notifyUser(String message) {

messageService.sendMessage(message);

}

}

// Usage

public class Main {

public static void main(String[] args) {

MessageService email = new EmailService();

Notification notifier = new Notification(email);

notifier.notifyUser("Welcome via Email!");

MessageService sms = new SMSService();

Notification notifier2 = new Notification(sms);

notifier2.notifyUser("Welcome via SMS!");

}

}

**Benefits:**

* You can inject any MessageService implementation
* No modification needed in Notification class for adding new types (Slack, WhatsApp etc.)

**💡 Another Example – Logger System**

interface Logger {

void log(String message);

}

class FileLogger implements Logger {

public void log(String message) {

System.out.println("Writing to file: " + message);

}

}

class DatabaseLogger implements Logger {

public void log(String message) {

System.out.println("Writing to database: " + message);

}

}

class App {

private Logger logger;

public App(Logger logger) {

this.logger = logger;

}

public void doSomething() {

logger.log("App did something important.");

}

}

public class Main {

public static void main(String[] args) {

App app = new App(new FileLogger());

app.doSomething(); // Logs to file

App app2 = new App(new DatabaseLogger());

app2.doSomething(); // Logs to DB

}

}

**🧠 Summary of DIP**

| **Aspect** | **Description** |
| --- | --- |
| Goal | Decouple high-level and low-level modules |
| Means | Use interfaces or abstract classes |
| Benefit | Flexibility, testability, scalability |
| Real-world analogies | Universal chargers, plugin-based architecture |

**✅ Key Takeaways**

* DIP promotes **decoupled architecture**
* Improves **testability** (you can mock services in unit tests)
* Makes code **extensible** and **maintainable**
* Encourages **dependency injection** (constructor/setter/field)

Would you like examples of applying DIP in other domains like web frameworks or Android apps?