Low Level Design (LLD)

Code Docs

Aa Name
Overview
Template
Untitled

Low Level Design (LLD)

SOLID PRINCIPLES

S - Single Responsibility Principle

O - Open Closed Principle

L - Liskov Substitution Principle

I - Interface Segmented Principle

D - Dependency Inversion Principle

Advantages :

1. Avoid Duplicate code
2. Easy to maintain
3. Easy to understand
4. Flexible Software
5. REduces Complexity

Table of Contents

SIngle Responsibility

A class should have only one reason to change.

Before Applying SRP (Violating the Principle)

Book Class

🔴 **Problem:** `Invoice` is responsible for both **calculation** and **printing**, which makes it harder to maintain.

```
public class Book {
    String name;
    double price;

    public Book(String name, double price) {
        this.name = name;
        this.price = price;
    }
}
```

```
public class Invoice {
    Book book;
    int quantity;
    double discountRate; // e.g., 0.1 for 10% discount
    double taxRate;      // e.g., 0.05 for 5% tax
    double total;

    public Invoice(Book book, int quantity, double disc
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = calculateTotal();
    }

    public double calculateTotal() {
        double discountedPrice = (book.price - (book.pr
        return discountedPrice * (1 + taxRate); // Apply t
    }

    public void printInvoice() {
```

```
        System.out.println(quantity + "x " + book.name -
        System.out.println("Discount Rate: " + (discountI
        System.out.println("Tax Rate: " + (taxRate * 100)
        System.out.println("Total: $" + total);
    }
    public void saveToFile(String filename) {
}
```

```
public class Main {
    public static void main(String[] args) {
        Book book = new Book("Clean Code", 50.0);
        Invoice invoice = new Invoice(book, 2, 0.1, 0.05);
        invoice.printInvoice();
    }
}
```

So we do SIngle responsibility principle where -

After Applying SRP

Book Class remains the same , we change the invoice class to handle just one function or rresponsibility and another class to handle another function.

```
public class Invoice {
    Book book;
    int quantity;
    double discountRate;
    double taxRate;
    double total;

    public Invoice(Book book, int quantity, double disc
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = calculateTotal();
    }
```

```
    public double calculateTotal() {
        double discountedPrice = (book.price - (book.pr
        return discountedPrice * (1 + taxRate);
    }
}
```

```
public class InvoicePrinter {
    Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void printInvoice() {
        System.out.println(invoice.quantity + "x " + invoi
        System.out.println("Discount Rate: " + (invoice.d
        System.out.println("Tax Rate: " + (invoice.taxRat
        System.out.println("Total: $" + invoice.total);
    }
}
```

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the i
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Book book = new Book("Clean Code", 50.0);
        Invoice invoice = new Invoice(book, 2, 0.1, 0.05);
```

```
        InvoicePrinter printer = new InvoicePrinter(invoic
        printer.printInvoice();
    }
}
```

Why This is Better (SRP Applied)

✅ **Each class has a single responsibility**

✅ **Easier to modify the printing logic without affecting invoice calculations**

✅ **More maintainable and scalable**

Open Closed Principle

Open for extension but closed for modification.

But how are we going to add new functionality without touching the class, you may ask. It is usually done with the help of interfaces and abstract classes.

Now that we have covered the basics of the principle, let's apply it to our Invoice application.

Let's say our boss came to us and said that they want invoices to be saved to a database so that we can search them easily.

Before OC principle

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
```

```
        // Saves the invoice to database
    }
}
```

After OC Principle

```
interface InvoicePersistence {

    public void save(Invoice invoice);
}
```

```
public class DatabasePersistence implements InvoicePersistence {

    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}
```

```
public class FilePersistence implements InvoicePersistence {

    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

Why is `PersistenceManager` Required?

The `PersistenceManager` **class** is a central **controller** for persisting invoices and books.

Instead of **directly calling** `save()` **on different storage classes**, we use `PersistenceManager` to **handle all persistence operations in one place**.

📌 **Main Purpose:**

- Provides a **single point of access** for saving invoices and books.
- **Decouples business logic** from storage logic.

- Follows **Open/Closed Principle (OCP)** by allowing **new persistence methods without modifying existing code**.

```
public interface BookPersistence {
    void save(Book book);
}

public interface BookPersistence {
    void save(Book book);
}
```

```
import java.io.FileWriter;
import java.io.IOException;

public class FileInvoicePersistence implements InvoicePersistence {
    @Override
    public void save(Invoice invoice) {
        String filename = "invoice.txt";
        try (FileWriter writer = new FileWriter(filename)) {
            writer.write(invoice.quantity + "x " + invoice.book.name + " @ $" + invoice.book.price
            writer.write("Discount Rate: " + (invoice.discountRate * 100) + "%\n");
            writer.write("Tax Rate: " + (invoice.taxRate * 100) + "%\n");
            writer.write("Total: $" + invoice.total + "\n");
            System.out.println("Invoice saved to file: " + filename);
        } catch (IOException e) {
            System.out.println("Error saving invoice: " + e.getMessage());
        }
    }
}
```

```
public class DatabaseInvoicePersistence implements InvoicePersistence {
    @Override
    public void save(Invoice invoice) {
        System.out.println("Invoice saved to database:");
        System.out.println("Book: " + invoice.book.name);
        System.out.println("Quantity: " + invoice.quantity);
        System.out.println("Total: $" + invoice.total);
    }
}
```

```
import java.io.FileWriter;
import java.io.IOException;

public class FileBookPersistence implements BookPersistence {
    @Override
    public void save(Book book) {
        String filename = "books.txt";
        try (FileWriter writer = new FileWriter(filename, true)) {
            writer.write("Book: " + book.name + " | Price: $" + book.price + "\n");
            System.out.println("Book saved to file: " + filename);
        } catch (IOException e) {
            System.out.println("Error saving book: " + e.getMessage());
        }
    }
}
```

```
public class DatabaseBookPersistence implements BookPersistence {
    @Override
    public void save(Book book) {
        System.out.println("Book saved to database:");
        System.out.println("Book: " + book.name);
        System.out.println("Price: $" + book.price);
    }
}
```

```
public class PersistenceManager {
    InvoicePersistence invoicePersistence;
    BookPersistence bookPersistence;

    public PersistenceManager(InvoicePersistence invoicePersistence,
                              BookPersistence bookPersistence) {
        this.invoicePersistence = invoicePersistence;
        this.bookPersistence = bookPersistence;
    }

    public void saveInvoice(Invoice invoice) {
        invoicePersistence.save(invoice);
    }
```

```
    public void saveBook(Book book) {
        bookPersistence.save(book);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create book and invoice
        Book book = new Book("Clean Code", 50.0);
        Invoice invoice = new Invoice(book, 2, 0.1, 0.05); // 10% discount, 5% tax

        // Create different persistence implementations
        InvoicePersistence fileInvoiceSaver = new FileInvoicePersistence();
        BookPersistence databaseBookSaver = new DatabaseBookPersistence();

        // Create Persistence Manager with chosen strategies
        PersistenceManager persistenceManager = new PersistenceManager(fileInvoiceSaver, d

        // Save invoice to file and book to database
        persistenceManager.saveInvoice(invoice);
        persistenceManager.saveBook(book);
    }
}
```

Liskov Substitution Principle

If Class B is subtype of class A then we should be able to replace object of class A with class B and vice versa without breaking the behavior of the code meaning subclass should extend the capability of parent class and not narrow it down.

Before LSP

```
// Parent class: Bike
public class Bike {
    public void ride() {
```

```
        System.out.println("Riding the bike...");
    }

    public void pedal() {  // 🚨 Problem! Not all bikes have pedals (e.g., motorcycles)
        System.out.println("Pedaling the bike...");
    }
}

// Subclass: Motorcycle
public class Motorcycle extends Bike {
    @Override
    public void pedal() {  // 🚨 This is incorrect because motorcycles don't have pedals!
        throw new UnsupportedOperationException("Motorcycles do not have pedals.");
    }
}
```

After LSP

Solution:

- **Separate** the `Bike` class into `Vehicle` and use a better hierarchy.
- **Avoid forcing** `Motorcycle` **to implement** `pedal()`

```
// Parent class: Vehicle (applies to all vehicles)
public class Vehicle {
    public void ride() {
        System.out.println("Riding the vehicle...");
    }
}
```

```
// Bike class extends Vehicle and has pedals
public class Bike extends Vehicle {
    public void pedal() {
        System.out.println("Pedaling the bike...");
    }
}
```

```
// Motorcycle extends Vehicle but does not have pedals
public class Motorcycle extends Vehicle {
    public void startEngine() {
        System.out.println("Starting the motorcycle engine...");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Vehicle myBike = new Bike();
        myBike.ride(); // ✅ Works
        ((Bike) myBike).pedal(); // ✅ Works because it's a Bike

        Vehicle myMotorcycle = new Motorcycle();
        myMotorcycle.ride(); // ✅ Works
        // ((Motorcycle) myMotorcycle).pedal(); ❌ Error! Motorcycles don't have pedals
        ((Motorcycle) myMotorcycle).startEngine(); // ✅ Works correctly
    }
}
```

Interface Segmented Principle

Interfaces should be such that clients should not implement unnescsary functions that they don't need.

Before ISP

```
public interface RestaurantWorker {
    void takeOrder();  // 🚨 Chefs do not take orders
    void serveFood();  // 🚨 Chefs do not serve food
    void cookFood();   // 🚨 Waiters do not cook
}
```

```
public class Waiter implements RestaurantWorker {
    @Override
    public void takeOrder() {
        System.out.println("Taking the customer's order...");
```

```
    }

    @Override
    public void serveFood() {
        System.out.println("Serving food to customers...");
    }

    @Override
    public void cookFood() {  // ❌ Waiters do not cook!
        throw new UnsupportedOperationException("Waiters do not cook.");
    }
}
```

```
public class Chef implements RestaurantWorker {
    @Override
    public void takeOrder() {  // ❌ Chefs do not take orders!
        throw new UnsupportedOperationException("Chefs do not take orders.");
    }

    @Override
    public void serveFood() {  // ❌ Chefs do not serve food!
        throw new UnsupportedOperationException("Chefs do not serve food.");
    }

    @Override
    public void cookFood() {
        System.out.println("Cooking the food...");
    }
}
```

After ISP

```
// Interface for Waiters
public interface WaiterInterface {
    void takeOrder();
    void serveFood();
}

// Interface for Chefs
```

```
public interface ChefInterface {
    void cookFood();
}
```

```
public class Waiter implements WaiterInterface {
    @Override
    public void takeOrder() {
        System.out.println("Taking the customer's order...");
    }

    @Override
    public void serveFood() {
        System.out.println("Serving food to customers...");
    }
}
```

```
public class Chef implements ChefInterface {
    @Override
    public void cookFood() {
        System.out.println("Cooking the food...");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        WaiterInterface waiter = new Waiter();
        waiter.takeOrder(); // ✅ Works
        waiter.serveFood(); // ✅ Works

        ChefInterface chef = new Chef();
        chef.cookFood(); // ✅ Works
    }
}
```

Dependency Inversion Principle

Class should depend on interfaces rather than concrete class.

Before DIP

COncrete classes

```
public class AppleKeyboard {
    public void type() {
        System.out.println("Typing on Apple Keyboard...");
    }
}

public class AppleMouse {
    public void click() {
        System.out.println("Clicking with Apple Mouse...");
    }
}
```

```
public class MacBook {
    private AppleKeyboard keyboard;
    private AppleMouse mouse;

    public MacBook() {
        this.keyboard = new AppleKeyboard(); // ❌ Direct dependency on AppleKeyboard
        this.mouse = new AppleMouse();       // ❌ Direct dependency on AppleMouse
    }

    public void start() {
        keyboard.type();
        mouse.click();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MacBook macBook = new MacBook();
        macBook.start();
    }
}
```

Problems in This Design

1. **MacBook cannot use a different keyboard or mouse** (e.g., Logitech).
2. **Tightly coupled** to `AppleKeyboard` and `AppleMouse`.
3. **Modifying** `MacBook` **is required if a new input device is introduced**.

After DIP

```
// Keyboard abstraction
public interface Keyboard {
    void type();
}

// Mouse abstraction
public interface Mouse {
    void click();
}
```

```
public class AppleKeyboard implements Keyboard {
    @Override
    public void type() {
        System.out.println("Typing on Apple Keyboard...");
    }
}

public class LogitechKeyboard implements Keyboard {
    @Override
    public void type() {
        System.out.println("Typing on Logitech Keyboard...");
    }
}
```

```
public class AppleMouse implements Mouse {
    @Override
    public void click() {
        System.out.println("Clicking with Apple Mouse...");
    }
}
```

```
public class LogitechMouse implements Mouse {
    @Override
    public void click() {
        System.out.println("Clicking with Logitech Mouse...");
    }
}
```

```
public class MacBook {
    private Keyboard keyboard;
    private Mouse mouse;

    // Constructor Injection (MacBook doesn't know which brand is used)
    public MacBook(Keyboard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }

    public void start() {
        keyboard.type();
        mouse.click();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Using Apple Keyboard and Apple Mouse
        Keyboard appleKeyboard = new AppleKeyboard();
        Mouse appleMouse = new AppleMouse();
        MacBook macBook1 = new MacBook(appleKeyboard, appleMouse);
        macBook1.start();

        // Using Logitech Keyboard and Logitech Mouse
        Keyboard logitechKeyboard = new LogitechKeyboard();
        Mouse logitechMouse = new LogitechMouse();
        MacBook macBook2 = new MacBook(logitechKeyboard, logitechMouse);
        macBook2.start();
    }
}
```