***DAY 20 HOME TASKS***

### ***Home Task 01: SRP Violation vs Implementation***

**Problem:**

The original Book class violates the **Single Responsibility Principle** because it:

* Manages book data (title, author, price).
* Formats the book title (getFormattedTitle()).
* Calculates price-related logic (calculateDiscountedPrice()).

**SRP Implementation**

**BookDetails.java** Handles only book-related data.

public class BookDetails {

private String title;

private String author;

private double price;

public BookDetails(String title, String author, double price) {

this.title = title;

this.author = author;

this.price = price;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public double getPrice() {

return price;

}

}

**BookFormatter.java** Handles **only formatting**.

public class BookFormatter {

public String formatTitle(BookDetails book) {

return "Title: " + book.getTitle().toUpperCase();

}

}

**PriceCalculator.java** Handles **price calculations**.

public class PriceCalculator {

public double calculateDiscountedPrice(BookDetails book, double discountPercentage) {

return book.getPrice() \* (1 - discountPercentage);

}

}

**TestSRP.java** Bringing it all together.

public class TestSRP {

public static void main(String[] args) {

BookDetails book = new BookDetails("Clean Code", "Robert C. Martin", 500);

BookFormatter formatter = new BookFormatter();

PriceCalculator priceCalc = new PriceCalculator();

System.out.println(formatter.formatTitle(book));

System.out.println("Discounted Price: " + priceCalc.calculateDiscountedPrice(book, 0.1));

}

}

### ***Home Task 02: SRP for Customer***

**SRP Violation**

Your Customer class is doing **two things**:

1. Storing customer data.
2. Handling file saving (I/O operations).

**SRP Implementation**

**Customer.java** Handles only customer-related data.

public class Customer {

private String name;

private String custID;

public Customer(String name, String custID) {

this.name = name;

this.custID = custID;

}

public String getName() {

return name;

}

public String getCustID() {

return custID;

}

}

**ManagingFiles.java** Handles **file operations**.

import java.io.FileWriter;

import java.io.IOException;

public class ManagingFiles {

public void saveData(Customer customer) {

try (FileWriter fw = new FileWriter(customer.getName() + ".txt")) {

fw.write("Customer Name: " + customer.getName() + "\n");

fw.write("Customer ID: " + customer.getCustID() + "\n");

System.out.println("Data saved successfully in " + customer.getName() + ".txt");

} catch (IOException ex) {

ex.printStackTrace();

}

}

}

**SRP\_Imple.java**

public class SRP\_Imple {

public static void main(String[] args) {

Customer cobj = new Customer("Prasunamba", "C001");

ManagingFiles mobj = new ManagingFiles();

mobj.saveData(cobj);

}

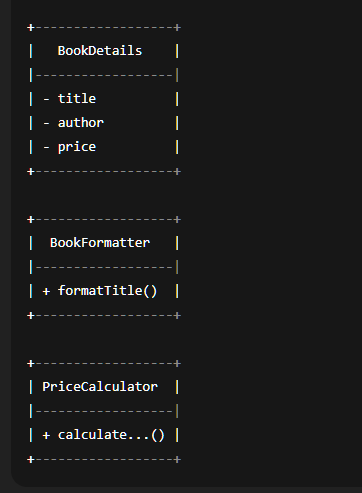
}

### ***Task 03: Structural Diagrams***

Structural diagrams in UML include:

* **Class Diagram** (shows classes, attributes, methods, and relationships).
* **Object Diagram** (instances of classes at runtime).
* **Component Diagram** (shows components and their interactions).
* **Package Diagram** (shows grouping of classes/modules).
* **Deployment Diagram** (shows hardware and software nodes).

For SRP examples, a **class diagram** would be:



***DAY 21 HOME TASKS***

### ***Task 01: Dependency***

**Definition:**

* A class **depends** on another when it uses it temporarily (e.g., as a method parameter or local variable).
* Represented with a **dashed arrow** in UML.

**Example Code:**

class Printer {

void printDocument(String doc) {

System.out.println("Printing: " + doc);

}

}

class User {

// User depends on Printer for printing documents

void createAndPrintDoc() {

Printer printer = new Printer(); // temporary dependency

printer.printDocument("UML Concepts Notes");

}

}

public class DependencyDemo {

public static void main(String[] args) {

User user = new User();

user.createAndPrintDoc();

}

}

**Key Point:** User **depends on** Printer but does not own or store it permanently.

### ***Task 02: Aggregation***

**Definition:**

* Aggregation is a **"has-a"** relationship where one class holds a reference to another but the contained object **can exist independently**.
* Represented with a **hollow diamond** in UML.

**Example Code:**

class Engine {

void start() {

System.out.println("Engine starts...");

}

}

class Car {

private Engine engine; // Aggregation: Car has an Engine, but Engine can exist without Car

public Car(Engine engine) {

this.engine = engine;

}

void startCar() {

engine.start();

System.out.println("Car is running.");

}

}

public class AggregationDemo {

public static void main(String[] args) {

Engine engine = new Engine(); // Engine can exist independently

Car car = new Car(engine); // Inject engine into car

car.startCar();

}

}

### ***Task 03: Composition***

**Definition:**

* Composition is a **stronger "has-a" relationship** where the lifetime of the part is controlled by the whole.
* Represented with a **filled diamond** in UML.

**Example Code:**

class Heart {

void pump() {

System.out.println("Heart is pumping...");

}

}

class Human {

private Heart heart; // Composition: Heart is part of Human, cannot exist without Human

public Human() {

this.heart = new Heart(); // Created inside Human

}

void startLife() {

heart.pump();

System.out.println("Human is alive.");

}

}

public class CompositionDemo {

public static void main(String[] args) {

Human human = new Human();

human.startLife();

}

}

**Key Point:** If the Human object is destroyed, its Heart also ceases to exist.

### 

### ***Task 04: Interface Segregation with Payment System (Bank & Loan Implementation)***

Key Points from Diagram

* Payment interface:
  + status()
  + getListOfPayment()
* Bank interface:
  + initiatePayment()
  + stopPayment()
* Loan interface:
  + initiateRepayment()
  + initiateFinalSettlement()
* BankPayment class:
  + Implements Payment and Bank.
* LoanPayment class:
  + Implements Payment and Loan.

**Code Implementation**

import java.util.ArrayList;

import java.util.List;

**===================== Interfaces =====================**

interface Payment {

Object status();

List<Object> getListOfPayment();

}

interface Bank {

void initiatePayment();

void stopPayment();

}

interface Loan {

void initiateRepayment();

void initiateFinalSettlement();

}

**===================== BankPayment Class =====================**

class BankPayment implements Payment, Bank {

private List<Object> payments = new ArrayList<>();

@Override

public void initiatePayment() {

System.out.println("Bank payment initiated.");

payments.add("Payment Started");

}

@Override

public void stopPayment() {

System.out.println("Bank payment stopped.");

payments.add("Payment Stopped");

}

@Override

public Object status() {

return payments.isEmpty() ? "No Payment" : payments.get(payments.size() - 1);

}

@Override

public List<Object> getListOfPayment() {

return payments;

}

}

**===================== LoanPayment Class =====================**

class LoanPayment implements Payment, Loan {

private List<Object> loanTransactions = new ArrayList<>();

@Override

public void initiateRepayment() {

System.out.println("Loan repayment initiated.");

loanTransactions.add("Repayment Started");

}

@Override

public void initiateFinalSettlement() {

System.out.println("Loan final settlement done.");

loanTransactions.add("Final Settlement Completed");

}

@Override

public Object status() {

return loanTransactions.isEmpty() ? "No Transactions" : loanTransactions.get(loanTransactions.size() - 1);

}

@Override

public List<Object> getListOfPayment() {

return loanTransactions;

}

}

**===================== Main Class =====================**

public class InterfaceSegregationDemo {

public static void main(String[] args) {

// Working with BankPayment

BankPayment bankPayment = new BankPayment();

bankPayment.initiatePayment();

bankPayment.stopPayment();

System.out.println("Bank Status: " + bankPayment.status());

System.out.println("Bank Payments: " + bankPayment.getListOfPayment());

// Working with LoanPayment

LoanPayment loanPayment = new LoanPayment();

loanPayment.initiateRepayment();

loanPayment.initiateFinalSettlement();

System.out.println("Loan Status: " + loanPayment.status());

System.out.println("Loan Payments: " + loanPayment.getListOfPayment());

}

}