S.O.L.I.D. Principles of Object-Oriented Programming

Table of Contents:

Single Responsibility Principle (SRP) …….pg: 03

Open/Closed Principle (OCP)……..pg: 05

Liskov substitution Principle (LSP)…….Pg: 07

Interface Segregation Principle (ISP)……..pg: 09

Dependency Inversion Principle (DIP)…….pg: 12

Single Responsibility Principle (SRP):

Bad package:

The bad package contains the following classes:

- **Item**
  This class represents an item in the inventory management with attributes `itemName` and `Count` and also it includes getters and setters for these attributes.
- **InventoryManagement**
  This class initializes an inventory HashMap , and methods for adding and deleting items the inventory displaying item counts and notifying restocks.
- **SRPMain**
  The SRPMain class contains the main method for running the application. And it prompts the user to enter item names, adds these items to the inventory, and then deletes them etc.

Violations in the Bad Package:

The InventoryManagement class is responsible for multiple tasks like:

managing inventory items
displaying item counts
and notifying about restocks.

This violates SRP because a class should only have one reason to change.

Good package:

The Good package contains the following classes:

- **Item**:
  Represents an item with attributes itemName and count. And Includes getters and setters for these attributes.

- **InventoryManagement**:
  Manages the inventory using HashMap and collaborates with AddItemService, DeleteItemService, and DisplayItemService and notification to perform its functions.

- **AddItemService**:
  Responsible for adding items to the inventory.Takes the inventory map as a constructor parameter and performs the add operation.

- **DeleteItemService**:
  Responsible for deleting items from the inventory.Takes the inventory map as a constructor parameter and performs the delete operation.Delegates the notification and display logic to NotificationService and DisplayItemService.

- **DisplayItemService**:
  Responsible for displaying item counts.Takes the inventory map as a constructor parameter and provides a method to display the count of a given item.

- **NotificationService**:
  Responsible for notifying when an item needs to be restocked.Provides a static method for sending restock notifications.

- **SRPMainGood**:
  Contains the `main` method for running the application.Prompts the user to enter item names, adds these items to the inventory, and then deletes them using the refactored classes.

Violations FIX:

The violations in bad package were fixed by splitting tasks into separate service classes in the good package .

So,In the corrected code the InventoryManagement uses

- ✓ AddItemService class for adding items into the inventory
- ✓ DeleteItemService class for deleting items from inventory
- ✓ DisplayItemService class for showing item counts after each addition or deletion of item
- ✓ NotificationService class for restock alerts if the item count is less than 5.

Each class now does one specific job, and adhering to Single Responsibility Principle OF SOLID principles.

And the main method SRPMainGood handles user input and uses these services for inventory tasks and its management.

Open/closed Principle (OCP):

Bad package:

The bad package contains the following classes:

- **Product**
  Represents a product with attributes like name, type represents the categories like electronics,books,clothing etc , and price of the product. And includes methods to calculate discounts, HST, GST, and the final price for the product .

- **OCPMain**
  Demonstrates the usage of product class with different categories types

Violations in the bad package:

The Product class requires modifications if there is change in the pricing strategy ,tax calculation or else adding a new category of products like food, appliances etc. As the modification is required in the product when a change happens so it violates the OCP principle.

Good package:

The good package contains the following classes:

- **Product**:
  Represents a product with a name, price, and uses a PricingStrategy to calculate final prices based on specific discount and tax rules.

- **PricingStrategy**:
  An Interface defining methods for calculating discounts, HST, and GST based on a product's price.

- **ElectronicsPricingStrategy**:
  Implements PricingStrategy for electronics products, defining specific discount, HST, and GST calculations.

- **ClothingPricingStrategy**:
  Implements PricingStrategy for clothing products, defining specific discount, HST, and GST calculations.

- **BooksPricingStrategy**:
  Implements PricingStrategy for books products, defining specific discount, HST, and GST calculations.

- **OCPMainGood**:
  Main class interacts with user to get input to create Product instances with different PricingStrategy implementations

Violations Fix:

- ✓ In the good package Instead of embedding discount and tax calculations directly into the Product class as in bad package.

- ✓ A PricingStrategy interface with methods to calculate discount,Tax(HST and GST) is defined .

- ✓ And each type of pricing strategy like electronics ,clothing,books implements these interface methods by overriding the methods defined in the interface according to their category.

- ✓ This separation allows new types of products and pricing strategies to be added without modifying existing code, thereby adhering to the OCP.

- ✓ The Product class now uses a PricingStrategy object, allowing it to calculate final prices dynamically based on the chosen strategy.

Liskov Substitution Principle (LSP):

Bad package:

The bad package contains the following classes:

- **BankAccount**:
  Represents a basic bank account with operations like deposit, withdrawal, and balance retrieval.

- **CurrentAccount**:
  Extends BankAccount and overrides the calculateInterest method to return zero interest, reflecting the behavior of a current account which typically does not earn interest.

- **FixedDepositAccount**:
  Extends BankAccount and overrides both calculateInterest and withdraw methods to handle fixed deposit-specific interest calculations like for terms of 18 months,6 months ,12 months etc and penalties for early withdrawals, respectively.

- **SavingsAccount**:
  Extends BankAccount and overrides the calculateInterest method to apply a fixed interest rate, typical for savings accounts.

Violations in bad package:

- In the bad package the FixedDepositAccount class extends BankAccount and this alters the behavior of the withdraw method by introducing a penalty calculation (double penalty = balance * 0.05;) for early withdrawals.

- This modification violates LSP because it changes the expected behavior defined in the superclass (BankAccount).

- Substituting FixedDepositAccount for BankAccount in client code could lead to unexpected results, as clients relying on standard BankAccount behavior (without penalties) would encounter behavior that does not conform to their assumptions.

Good Package:

The Good package contains the following classes:

- **BankAccount**:
  Abstract class representing a generic bank account with methods for depositing, withdrawing, and calculating interest.
  Subclasses (CurrentAccount, FixedDepositAccount, SavingsAccount) provide specific implementations of interest calculation and withdrawal behavior.

- **CurrentAccount**:
  Extends BankAccount and overrides the calculateInterest method to return zero interest, reflecting the behavior of a current account.

- **FixedDepositAccount**:
  Extends BankAccount and overrides both calculateInterest to compute interest based on the deposit term and withdraw to include a penalty for early withdrawals from fixed deposits.

- **SavingsAccount**:
  Extends BankAccount and overrides calculateInterest to apply a fixed interest rate typical for savings accounts.

Violations Fix:

- ✓ The Liskov Substitution Principle (LSP) violation is addressed by introducing an abstract BankAccount class that serves as the base for all types of bank accounts.

- ✓ This abstract class defines common operations like deposit, withdraw, and getBalance, ensuring consistency in behavior across subclasses.

- ✓ Each specific type of account (CurrentAccount, FixedDepositAccount, SavingsAccount) extends BankAccount and implements its unique behavior through methods like calculateInterest, getbalance and withdraw.

- ✓ By structuring the hierarchy this way, subclasses adhere strictly to their defined roles without altering inherited methods in ways that could surprise client code.

- ✓ For example, FixedDepositAccount correctly overrides calculateInterest to handle interest rates based on deposit terms and adds a penalty for early withdrawals, staying within the bounds set by BankAccount without introducing unexpected behaviors.

- ✓ This approach ensures that all subclasses maintain substitutability and uphold the principle

Interface segregation Principle:

Bad Package:

The bad package contains the following classes:

- **Administrator**:
  Implements Healthcare with methods for scheduling appointments, prescribing medications, but throws unsupported exceptions for surgery, insurance management, and patient billing, which administrators typically don't handle.

- **AdministratorManagementSystem**:
  Manages administrators using a Healthcare interface, attempting to call methods for surgery, insurance management, and patient billing that administrators cannot perform, potentially leading to runtime errors or unexpected behavior.

- **Doctor**:
  Implements Healthcare with methods for scheduling appointments, prescribing medications, performing surgery, managing insurance, and billing patients, reflecting typical doctor responsibilities in healthcare.

- **Nurse**:
  Implements Healthcare with methods for scheduling appointments and prescribing medications but throws unsupported exceptions for surgery, insurance management, and patient billing, tasks typically not within a nurse's scope.

- **PatientManagementSystem**:
  Manages patients using a Healthcare interface, potentially encountering issues when calling methods like surgery, insurance management, and patient billing that nurses and administrators cannot perform.

Violations in bad package:

- In the bad.I package, the Healthcare interface contains methods such as performSurgery, manageInsurance, and billPatient. These methods are implemented by classes like Administrator, Doctor, and Nurse.

- However, administrators and nurses throw UnsupportedOperationExceptions for methods like performSurgery, manageInsurance, and billPatient because these tasks are typically beyond their roles and responsibilities.

- This violates the Interface Segregation Principle (ISP) because it forces classes to implement methods that are irrelevant or unsupported for their specific roles.

Good package:

The good package contains the following classes:

- **AdministratorManagementSystem**:
  Manages administrators using an AdministratorService which implements AppointmentScheduler, allowing administrators to schedule appointments.

- **AdministratorService**:
  Implements AppointmentScheduler with logic specific to scheduling appointments for administrators.

- **Doctor**:
  Implements multiple interfaces (AppointmentScheduler, MedicationPrescriber, Surgeon, InsuranceManager, PatientBiller) for scheduling appointments, prescribing medication, performing surgery, managing insurance, and billing patients.

- **AppointmentScheduler**:
  Interface defining the method scheduleAppointment used by administrators and nurses to schedule appointments.

- **InsuranceManager**:
  Interface defining the method manageInsurance used by doctors to manage insurance tasks.

- **ISPMainGood**:
  Main class demonstrating usage of interfaces (AppointmentScheduler, MedicationPrescriber, Surgeon, InsuranceManager, PatientBiller) to manage patients with different healthcare roles (Doctor, Nurse, AdministratorService).

- **MedicationPrescriber**:
  Interface defining the method prescribeMedication used by doctors and nurses to prescribe medications.

- **Nurse**:
  Implements AppointmentScheduler and MedicationPrescriber for scheduling appointments and prescribing medications.

- **PatientBiller**:
  Interface defining the method billPatient used by doctors to handle patient billing.

- **PatientManagementSystem**:
  Manages patients using interfaces (AppointmentScheduler, MedicationPrescriber, Surgeon, InsuranceManager, PatientBiller) to schedule appointments, prescribe medication, perform surgery (if applicable), manage insurance (if applicable), and bill patients (if applicable).

Violations Fix:

- ✓ In the good.I package, the Interface Segregation Principle (ISP) violation is fixed by breaking down the monolithic Healthcare interface from the bad.I package into smaller and more focused interfaces such as AppointmentScheduler, MedicationPrescriber, Surgeon, InsuranceManager, PatientBiller.

- ✓ Each interface now represents a specific role or capability within the healthcare system.

- ✓ Classes now implement only the interfaces that are relevant to their responsibilities, ensuring that they do not inherit or implement unnecessary methods.

- ✓ For example, Doctor implements AppointmentScheduler, MedicationPrescriber, Surgeon, InsuranceManager, and PatientBiller, reflecting their comprehensive role, while Nurse implements AppointmentScheduler and MedicationPrescriber, aligning with their more focused duties.

Dependency Inversion Principle (DIP):

Bad Package:

The bad package contains these following classes:

- **Book:**
  Represents a book with attributes for title and author. Provides getters for accessing these attributes.

- **BookManager:**
  Manages the issuing and returning of books for users, printing details such as the book title, user, and dates. Directly interacts with LibraryDatabase for updating issued and returned books.

- **DIPMain:**
  Contains the main method to demonstrate book management functionalities like issuing, returning, sending notifications, calculating fines, and searching books, all directly instantiated within the class.

- **FineManager:**
  Calculates fines for users based on the number of late days after returning a book, printing the fine amount. Depends directly on User objects to perform the calculation.

- **LibraryDatabase:**
  Handles database operations for issuing and returning books, printing messages about updating the database with book details for specific users.

- **Notification:**
  Sends notifications to users with custom messages, printing details about the user and the notification message sent.

- **Search:**
  Searches for books in the library database based on a provided query, printing details about the search operation and the query used.

- **User:**
  Represents a library user with a username attribute. Provides a getter method to retrieve the username.

Violations:

- In the bad.D package, violations of the Dependency Inversion Principle (DIP) occur due to direct dependencies between high-level modules and their concrete implementations.

- Specifically, BookManager directly creates instances of LibraryDatabase, Notification, FineManager, and Search, rather than depending on abstractions or interfaces.

- This direct instantiation and usage tightly couple BookManager to specific implementations, making it difficult to swap implementations or extend functionality without modifying BookManager itself.

- Similarly, FineManager directly relies on User details for calculating fines, rather than abstracting this dependency through an interface or higher-level abstraction.

Good Package:

The Good package contains the following classes:

- **Book**:
  Represents a book with properties for title and author. Provides getters and setters for accessing and modifying these properties.

- **BookManager**:
  Implements BookManagerInterface to manage book issuance and return operations. It depends on LibraryDatabaseInterface for updating book statuses in the database.

- **DIPMainGood**:
  Contains the main method for demonstrating dependency inversion principles. It uses interfaces such as BookManagerInterface, NotificationInterface, FineInterface, SearchInterface to interact with various functionalities such as issuing books, sending notifications, calculating fines, and searching books.
- **Fine**:
  Implements FineInterface to calculate fines based on late return days for books.

- **LibraryDatabase**:
  Implements LibraryDatabaseInterface to update the status of issued and returned books in the database.

- **Notification**:
  Implements NotificationInterface to send notifications to users with custom messages.

- **Search**:
  Implements SearchInterface to search for books in the library database based on a query.

- **User**:
  Represents a library user with a username and provides methods to get and set the username.

Violations Fix:

- In the good.D package, the violations of the DIP seen in the bad.D package are rectified through defining interfaces and then implementing the interfaces specific functionality.

- Firstly, interfaces BookManagerInterface, LibraryDatabaseInterface, NotificationInterface, FineInterface, SearchInterface are introduced to define contracts for different functionalities, ensuring high-level modules depend on abstractions rather than concrete implementations.

- Classes like BookManager now accept dependencies through constructor injection, enabling flexibility and allowing different implementations of LibraryDatabaseInterface, NotificationInterface, FineInterface, and SearchInterface to be used interchangeably without modifying BookManager itself.

- This approach reduces coupling between components and makes the application decoupled or lightly coupled.