Advance Topics in Software Development (Summer 2023)

Assignment 2

Submitted by: Arihant Dugar (B00917961)

GitLab repo: https://git.cs.dal.ca/courses/2023-summer/csci-5308/assignment2/dugar

Application of S.O.L.I.D Principles :

- **Single Responsibility Principle (Calculator Example):**

  Every class should have a single responsibility. There should never be more than one reason for a class to change.

  BAD Package :

  The **Calculator** class in the example code violates the **Single Responsibility Principle** (SRP) because it takes on multiple responsibilities that should be separated into distinct classes. SRP states that a class should have only one reason to change, meaning it should have a single responsibility or purpose.

  In the code, the **Calculator** class has several methods related to mathematical operations (add, subtract, multiply, divide, etc.), as well as utility methods (generateFibonacciSequence, isPrime, etc.). These responsibilities should ideally be separated into different classes to achieve better separation of concerns. Any changes related to logging, such as modifying the logging mechanism or adding additional logging features, would require modifying the Calculator class.

  GOOD Package:

  - **Calculator:** This class is responsible for performing basic mathematical operations, such as addition, subtraction, multiplication, and division.

    Violations addressed: The Calculator class now focuses solely on mathematical calculations, separating it from other concerns. The code has been refactored to remove unrelated functionality, such as logging and advanced statistical calculations.

  - **CalculatorLogger:** This class is responsible for logging operations performed by the calculator.

Violations addressed: The logging functionality has been moved to the CalculatorLogger class. By extracting the logging responsibility into its own class, the Calculator class is no longer burdened with unrelated responsibilities.

- **NumberCalculator:** This class is responsible for performing mathematical operations specifically related to numbers, such as calculating the least common multiple, and determining whether a number is even or odd etc.

  Violations addressed: The code related to number-specific operations has been moved to the NumberCalculator class. By separating number-related operations into their own class, the Calculator class becomes more focused on its core responsibility of basic mathematical calculations.

- **StatisticsCalculator:** This class is responsible for performing statistical calculations, such as calculating the average, and standard deviation of a set of numbers.

  Violations addressed: The statistical calculations have been moved to the StatisticsCalculator class. By separating the statistical functionality into its own class, the Calculator class is relieved from handling unrelated responsibilities.

The violations of SRP have been addressed by separating concerns into distinct classes. The Calculator class now focuses solely on basic mathematical calculations, while the CalculatorLogger, NumberCalculator, and StatisticsCalculator classes handle logging, number-specific operations, and statistical calculations, respectively. This refactoring improves code organization, enhances maintainability, and allows each class to have a clear and single responsibility, aligning with the principles of SRP.

- **Open-Closed Principle (Car Manager Example):**

  Software entities should be open for extension, but closed for modification.

  BAD Package :

  The CarManager class in the example code violates the Open-Closed Principle (OCP) because it is not closed for modification and does not provide an easy way to extend its behaviour without modifying its source code. OCP states that software entities (classes, modules, etc.) should be open for extension but closed for modification. Here's a description of the CarManager class and an explanation of why it violates the OCP:

  - **CarManager:** This class manages car-related operations, including fetching car details based on the car model.

    Violations of OCP:

a. Conditional statements: The methods uses conditional statements to determine the car model and return the corresponding details. This design makes the class less extensible as any addition or modification of car models requires modifying the existing code.

b. Lack of abstraction: The class directly deals with specific car models (Sedan, SUV, Hatchback) and their details. Adding a new car model would require modifying the existing code, violating the OCP.

The violations occur because the CarManager class is tightly coupled to specific car models and their details, making it difficult to extend without modifying the class itself. This violates the OCP as it is not open for extension without modifying its code.

GOOD Package:

- **Car interface:** This interface defines the contract for car objects and declares common methods related to car details.

  Violations addressed: By introducing the Car interface, the code establishes an abstraction that allows the class to interact with cars through the interface without knowing the specific implementations. This enables the classes to be open for extension to support new car models without modifying its existing code.

- **Hatchback:** This class represents a hatchback car and implements the Car interface.

  Violations addressed: The Hatchback class provides its own implementation of the methods declared in the Car interface, such as getModel(), getColor(), and getSeating(). Adding a new car model, such as a sedan or an SUV, can be done by creating a new class that implements the Car interface, ensuring the main class remains closed for modification and open for extension.

- **Sedan:** This class represents a sedan car and implements the Car interface.

  Violations addressed: Similar to the Hatchback class, the Sedan class provides its own implementation of the methods declared in the Car interface. The class can be extended or modified independently to support new features or variations of sedan cars without affecting the main class.

- **SUV:** This class represents an SUV car and implements the Car interface.

  Violations addressed: The SUV class adheres to the same principles as the Hatchback and Sedan classes. It provides its own implementation of the methods defined in the Car interface, enabling the main class to interact with SUV objects without requiring changes to its code.

By introducing the Car interface and implementing it in separate classes for different car models, the code adheres to the Open-Closed Principle.

- **Liskov's Substitution Principle (Plant with fruits and leaves Example):**

  Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

  BAD Package :

  Tree class extends the **Plant** class and adds additional methods specific to trees, such as growLeaves(), shedLeaves(), and bearFruit().

  However, this violates the **Liskov Substitution Principle** because not all plants necessarily have leaves or bear fruit. For instance, if we introduce a new subclass **Cactus** that extends **Plant**, the methods growLeaves() and bearFruit() would be irrelevant and potentially cause issues.

  GOOD Package:

  Introduced an abstract **Plant** class that serves as a common interface for plants with fruits and leaves and plants without fruits and leaves. It includes abstract methods growLeaves() and bearFruit() that will be implemented by the concrete subclasses.

  The concrete subclass **PlantWithFruitsAndLeaves** extends **Plant** and provides specific implementations for the growLeaves() and bearFruit() methods, as well as additional methods shedLeaves() that are specific to plants with fruits and leaves.

  Similarly, the concrete subclass **PlantWithoutFruitsAndLeaves** also extends **Plant** and does not provides its implementations for the methods growLeaves() and bearFruit() because they are not applicable to them, and provides implementation for additional methods that are specific to plants without fruits and leaves.

  By introducing the **Plant** abstract class, we create a common interface for both plant types and adhere to the Liskov Substitution Principle. The subclasses can be used interchangeably wherever an instance of **Plant** is expected, allowing for flexibility and avoiding violating the principle. Now, the **Cactus** and **Tree** can extend the classes **PlantWithoutFruitsAndLeaves** and **PlantWithFruitsAndLeaves** respectively.

- **Interface Segregation Principle (Employee Example):**

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

  BAD Package :

The **IEmployee** interface includes the **manageTeam() , provideFeedback()** method, which is not applicable to all employees. The **Manager** class, which represents a manager employee, implements all the methods of the **IEmployee** interface, including the **manageTeam() , provideFeedback()** method.

However, the **Developer** class, representing a regular employee, also has to implement the **manageTeam() , provideFeedback()** method even though it's not relevant to them. In this case, the implementation of the **manageTeam() , provideFeedback()** method in **Developer** throws an **UnsupportedOperationException** to indicate that regular employees do not have the authority to manage a team.

This violates the Interface Segregation Principle because the Employee interface forces the **Developer** class to implement a method that is not applicable to them, leading to unnecessary and incorrect behaviour in the codebase.

GOOD Package:

- **IManager**: The "**IManager**" interface represents the contract for a manager. It declares methods specific to managerial responsibilities, such as **manageTeam()** and **provideFeedback().** By segregating the interface, it ensures that classes implementing this interface will only need to implement the methods relevant to managers.

- **IEmployee**: The "**IEmployee**" interface represents the contract for an employee. It declares methods that are common to all employees, such as **work(), takeBreak()** etc. By segregating the interface, it ensures that classes implementing this interface will only need to implement the methods relevant to regular employees and not have to implement unnecessary methods related to management.

- **Manager**: The "**Manager**" class implements the "**IManager**" interface. It provides the concrete implementation of methods specific to managers, such as **manageTeam()** and **provideFeedback().** By implementing the "**IManager**" interface, the "**Manager**" class adheres to the Interface Segregation Principle (ISP) by implementing only the methods relevant to its role as a manager.

- **Developer**: The "**Developer**" class implements the "**IEmployee**" interface. It provides the concrete implementation of methods common to all employees, such as **work(), takeBreak()** etc. By implementing the "**IEmployee**" interface, the "**Developer**" class adheres to the ISP by implementing only the methods relevant to its role as an employee without having to implement unnecessary methods related to management.

Fixing the Violations:
To fix the violations of the Interface Segregation Principle, the following changes were made:

Interface Segregation: The interfaces **"IManager"** and **"IEmployee"** were segregated based on the different responsibilities and behaviours of managers and employees. This ensures that each interface represents a cohesive set of methods relevant to a specific role, avoiding the implementation of unnecessary methods in implementing classes.

By segregating the interfaces and implementing them in a way that each class only implements the relevant methods, the code adheres to the Interface Segregation Principle, promoting better separation of concerns and reducing unnecessary dependencies between classes.

- **Dependency Inversion Principle (Payment Processor Example):**

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

  BAD Package :

  The **PaymentManager** class directly depends on concrete implementations (**CreditCardProcessor** and **PaypalProcessor**) instead of depending on abstractions (e.g., the **PaymentProcessor** interface). The **PaymentManager** class creates instances of the concrete classes in its constructor, establishing a direct dependency on those implementations.

  The **processCreditCardPayment()** and **processPaypalPayment()** method in the PaymentManager class directly invokes the **processPayment()** methods of both the **CreditCardProcessor** and **PaypalProcessor** classes, violating the Dependency Inversion Principle.

  By relying on concrete implementations instead of abstractions, this code violates the DIP as it tightly couples the high-level module (**PaymentManager**) with the low-level modules (**CreditCardProcessor** and **PaypalProcessor**). This lack of abstraction makes it difficult to extend or modify the payment processing behaviour and increases the risk of code fragility and maintenance issues.

  GOOD Package:

  In this main method, we first create instances of **CreditCardProcessor** and **PaypalProcessor**, which are both implementations of the **PaymentProcessor** interface. Then, we create two instances of the **PaymentManager** class, passing in the respective **PaymentProcessor** implementations.

  By instantiating **PaymentManager** with different implementations of **PaymentProcessor**, we can demonstrate the flexibility and adherence to the

Dependency Inversion Principle. Each **PaymentManager** instance will use the appropriate payment processing implementation specified during construction.

In this logic, **the processPayment()** method is called on both paymentManager1 and **paymentManager2**, showcasing how different payment processors can be used interchangeably without modifying the **PaymentManager** class.

This design allows for easier extensibility and flexibility in the future. If a new payment processor implementation is introduced, we can simply create a new class that implements the **PaymentProcessor** interface and use it with the **PaymentManager** without modifying any existing code. This decoupling of dependencies promotes code reusability and maintainability.