DALHOUSIE UNIVERSITY

MASTER of APPLIED COMPUTER SCIENCE

Assignment 2

CSCI5308 - Advanced Software Development Concepts

Aharnish Maheshbhai Solanki

(Banner ID: B00933563, CSID: asolanki, Email: ah910744@dal.ca)

Link to Gitlab Repo : Courses / 2023-Summer / CSCI 5308 / Assignment2 / asolanki · GitLab (dal.ca)

Single Responsibility Principle

Violation:

In bad.s package there are two classes namely Course.java and CourseManagement.java.

Classes :

1. **Course.java** -: This class is an object class with three private instance variables: course_ID, courseName, and Credits. There are getter setter methods as well as empty and parameterized constructor.

2. **CourseManagement.java** :- This is the driving class that acts as a course management system. It includes various methods for managing a list of courses. It has a List<course> variable that stores all the course objects.

    - initialCourseList() , this method is called to initialize the List and fill it with the course objects.
    - In main() method, user is prompter to choose between options provided such as to see the courses, add new courses update the course details and delete a course. Based on the choice, function calls are made to the static methods present in the class to complete the operation.
    - All the business login is implemented in this class.

The Single Responsibility Principle states that a class should have only one reason to change. In the given code, the class CourseManagement violates the Single Responsibility Principle **because it is responsible for both managing the list of courses (CourseList) and handling the user interface logic for interacting with the user.**

Adherence:

In good.s package there are 4 classes namely, course.java , CourseData.java, CourseManager.java and Main.java

Classes :

1. Course.java: (same, no changes)
2. CourseData.java : This class is responsible for managing the list of courses basically maintains the data and performs operations directly. This is the business logic class. It provides methods to add, update, delete, and retrieve courses from the list. By isolating the course data management operations into a separate class, it adheres to SRP by having the responsibility of managing the course data.
3. CourseManager.java: This class acts as a mediator between the CourseData class and other parts of the application. It provides higher-level operations related to course management, such as handling user input. CourseManager adheres to SRP by having the responsibility of managing the course-related actions.
4. Main.java: This class contains the main method and serves as the entry point of the program. It is responsible for initializing the CourseManager and handling the user interface logic. By separating the user interface logic from the course data management, Main adheres to SRP by having the responsibility of handling the program's execution flow and user interaction.

   With this refactoring, each class has a single responsibility:

   Course.java: Represents a course entity.
   CourseData.java: Manages the course data (add, update, delete, retrieve).
   CourseManager.java: Coordinates course-related actions and presents the data.
   Main.java: Handles program execution and user interaction.

   Thus by creating individual class to handle operations we achieved SRP.

Open / Closed Principle

Violation:

In bad.o package there are 3 classes namely OCP_bad.java, Student_bad_OCP and Course_OCP_bad.java.

Classes :

1. **Student_bad_OCP.java** :- This class is an entity class with three private instance variables: course_ID, courseName, and Credits. There are getter setter methods as well as empty and parameterized constructor.

2. **Course_OCP_bad.java** :- This class is an entity class with three private instance variables: studentid, Name, and batchyear. There are getter setter methods as well as empty and parameterized constructor.

3. **OCP_bad.java :-** It has the main() method that controls the flow of the program , interating with the User and calling the necessary functions. It has 4 hashmap which are used to store the course and the student data.It has 4 methods

    - ViewStudents :- to view the students
    - ViewCourse:- to view the course
    - sortCourses:- to sort the courses based on their courseID
    - sortStudents:- to sort the students based on their id.

    The Open/Closed Principle (OCP) states that classes should be open for extension but closed for modification. In the given code, the OCP_bad class violates the Open/Closed Principle **because if we want to add more methods or functionalities we will have to make changes in the OCP_bad.java file.**

Adherence

In good.o package there are 9 classes namely

1. **OCP_good.java** : This class serves as the main entry point and manages the interaction with users. It contains the core functionality of the program, including the main method and user interface.
2. **interface Printer<T>** : generic interface
3. **interface Sorter<T> :** generic interface
4. **StudentPrinter.java** : implementing printer interface methods
5. **StudentSorter.java :** implementing sorter interface methods
6. **CoursePrinter.java :** implementing printer interface methods
7. **CourseSorter.java :** implementing sorter interface methods
8. **Student_OCP_good.java :** entity class
9. **Course_OCP_good.java :** entity class

Refactoring the code demonstrates the Open/Closed Principle by allowing the extension of the behavior without modifying the existing code. Here's how it adheres to the principle:

1. The Sorter interface defines a contract for sorting operations, and it's implemented by the StudentSorter and CourseSorter classes. These classes provide the sorting logic for students and courses, respectively.
2. The Printer interface defines a contract for printing operations, and it's implemented by the StudentPrinter and CoursePrinter classes. These classes handle the printing functionality for students and courses, respectively.
3. The code supports the addition of new sorting strategies or printing formats by introducing new classes that implement the Sorter and Printer interfaces without modifying the existing code. This makes the code open for extension.
4. The sort() method in the OCP_good class takes advantage of the Map<Integer, Object> type parameter to enable sorting of both students and courses using a single method. It uses a TreeMap to perform the sorting operation, leveraging the natural ordering of the keys.

By following these principles, the code allows for the addition of new sorting algorithms or printing formats without modifying the existing code. The existing code is closed for modifications but open for extension, making it easier to maintain and expand the system's functionality.

Liskov Substitution Principle

Violation:

In bad.l package there are 2 classes namely,

1. Datarepo_LSP_bad.java : This class is the data repo which holds the data and does all the manipulation to the data stored.

2. LSP_bad.java : This contains the main method that interacts with the user prompting them with various operations to be performed on the List data structure which holds Integers.

The LSP Principle states that objects of a superclass should be substitutable with objects of its subclasses without affecting the correctness of the program.

This program violates this because in the program ArrayList has been used to initiate an object, if later due to some condition this had to be changed , all the instances would have to be reomoved.

Adherence

In good.l package there are same 2 classes namely,

1. 1. Datarepo_LSP_bad.java : This class is the data repo which holds the data and does all the manipulation to the data stored.

2. LSP_bad.java : This contains the main method that interacts with the user prompting them with various operations to be performed on the List data structure which holds Integers.

Refactoring the code demonstrates the LSP principle, **Arraylist has been replaced with it's supertype class List which comes in java.collection package.** Now List can be instantiated with either ArrayList / Stack / Vector / LinkedList. Thus implementing LSP principle.

Integration Segregation

Violation:

In bad.i package there are 3 classes namely,

1. **ISP_bad.java :** contains main methods which controls the flow controls user interface.
2. **interface StringMethods.java:** This interface has all the methods to perform operation on Strings.
3. **StringMethodsIMP.java** : This method implements the stringmethods interface.

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they do not use. It emphasizes the importance of creating specific interfaces for clients, tailored to their needs, instead of having a single monolithic interface. **In the given code, the StringMethods interface violates the ISP by implementing all the methods even if not necessary.**

Adherence

Classes :

1.Reversal interface:

2. String Comparison interface ; has all the methods where strings are compared

3.String Manipulation interface; has all the method where stirngs are manipulated

4. String Traversal interface; has all the methods where string traversal is done

5.reversal imp class ; implements the interface

6. manipulation iml class : implements the interface

7.comparision imp class : implements the interface

Thus creating separate interfaces with methods allows us to implement specific methods and not to implement methods that are not necessary

Dependency Injection

Violation:

Classes:

1. Bubble sort : has sorting functionality of bubble sort
2. Insertion sort : has sorting functionality of insertion sort
3. Quick sort : has sorting functionality of quick sort
4. Selection sort: has sorting functionality of selection sort
5. Merge sort : has sorting functionality of merge sort
6. DIP_bad.java : contains main method that drives the functions.

The class interact directly with the main class, thus violates the DP principle.

Adherence

New Classes:

1. SortingAlgorithm interface : interface which has sort method
2. SortingAlgoController : implements the interface and it's metod sort.

The SortingAlgorithm interface acts as a meditor between the main() method and the sorting implementation methods. Thus the main() method which is a High-level module does not depend on low-level modules such as sorting algos. Both now depends on abstractions by SortingAlgorithm interface. Thus creating an interface helped us inject dependency and by this implemented Dependency Principle