CSCI 5308

Advanced Topics in Software Development

Assignment-2
S.O.L.I.D. Principles of Object-Oriented Programming

Prepared By
Bhavisha Oza (B00935827)

Name: Oza Bhavisha Himanshu

CSID: boza

Banner Id: B00935827

Gitlab repository link of individual assignment: https://git.cs.dal.ca/courses/2023-summer/csci-5308/assignment2

Bad Package code explanation:

1) Single responsibility principle (Package name: solid.bad.s)

a) Description of the classes and code:

The given package consists of two classes: “SRPViolation” and “RunApp”.

i) SRPViolation.java:

This class represents a violation of the Single Responsibility Principle (SRP). It contains multiple methods that handle different responsibilities such as handling orders, preparing, cooking, and serving items, calculating prices, and printing receipts.

The class includes the following methods:

- setPrices(): Sets the prices for various food items.
- getPrice(String itemName): Retrieves the price of an item by its name.
- addOrder(List<String> orderItems): Adds an order to the list of order items.
- removeOrder(): Removes all items from the current order.
- processOrders(): Processes all orders by preparing, cooking, and serving them.
  (a) prepare(): Prepares the order by performing necessary actions.
  (b) cook(): Cooks the order.
  (c) serve(): Serves the order.
- printReceipt(): Prints receipts for the order.
- calculateTotalSales(): Calculates and prints the total sales from all orders.
- calculateTotalPrice(): Calculates the total price of the current order.
- Item(String name, double price): Sets the name and price of an item.
- getName(): Returns the name of the item.
- getPrice(): Returns the price of the item.

ii) RunApp.java:

This class serves as the entry point for running the program. It creates an instance of the SRPViolation class and demonstrates its usage by performing various operations like setting prices, adding orders, processing orders, printing receipts, etc.

b) Why the code violates the principle?

The code violates the Single Responsibility Principle (SRP) because the SRPViolation class has **multiple responsibilities**. It handles order management, item preparation, cooking, serving, price calculations, and receipt printing. Each of these responsibilities should ideally be separated into distinct classes, following the principle of having a single reason to change. By combining multiple responsibilities in a single class, the code becomes tightly coupled and difficult to maintain and modify.

Good Package code explanation:

1) Single responsibility principle (Package name: solid.good.s)

a) <u>Description of the classes and code:</u>

The given package consists of four classes: “FoodMenu”, “OrderProcessor”, “ReceiptPrinter” and “RunApp”.

i) FoodMenu.java:

This class now represents a food menu in the restaurant. It contains methods related to managing the menu and orders. The key changes to fix the SRP violation are as follows:

- The responsibilities of preparing, cooking, and serving orders, as well as printing receipts, have been delegated to separate classes, namely OrderProcessor and ReceiptPrinter. This separation of concerns ensures that each class is responsible for a single task, adhering to the SRP.
- The processOrders() method now takes an OrderProcessor as a parameter, which will handle the preparation, cooking, and serving of orders.
- The printReceipt() method now takes a ReceiptPrinter as a parameter, which will handle the printing of receipts.
- The setPrices() method is now only responsible for setting food prices, which is a single responsibility.
- The Item() method has been removed as it was not serving any useful purpose.

ii) OrderProcessor.java:

This class represents an order processor that handles the preparation, cooking, and serving of orders. The changes made to this class are as follows:

- The methods prepare(), cook(), and serve() are now responsible for their respective tasks. Each method takes a list of order items and prints the appropriate messages for each step in the process.

iii) ReceiptPrinter.java:

This class represents a receipt printer that prints the receipt for an order. The changes made to this class are as follows:

- The printReceipt() method now takes the orderItems and foodPrices as parameters and prints the receipt accordingly. It calculates the total price based on the food prices, which is its single responsibility.

iv) RunApp.java:

This class serves as the entry point for running the program.

b) <u>How are the violations fixed?</u>

By separating the responsibilities of order processing and receipt printing into distinct classes (OrderProcessor and ReceiptPrinter, respectively), the SRP violation has been fixed. Each class now has a single responsibility, making the code more maintainable and easier to understand. The FoodMenu class is responsible for managing the food menu, setting prices, adding orders, and calculating total prices, while the order processing and receipt printing tasks are handled by separate specific classes.

Bad Package code explanation:

2) Open-closed principle (Package name: solid.bad.o)

To demonstrate a violation of the Open/Closed Principle, I've used the same code base as it is done in the good package of Single Responsibility package. The requirement was introduced is like to apply a discount to certain food items. So modified the existing code to accommodate this new requirement which violates the Open/Closed Principle

a) **Description of the classes and code:**

Five classes: "OrderProcessor", "ReceiptPrinter", "RunApp", "FoodMenu", "DiscountedReceiptPrinter"

i) **OrderProcessor.java:**
This class represents an order processor that prepares, cooks, and serves orders. It has methods to prepare, cook, and serve a list of order items.

ii) **ReceiptPrinter.java:**
This class represents a receipt printer that prints the receipt for an order. It has a method printReceipt that takes a list of order items and a map of food prices as input and prints the receipt with item names and their corresponding prices, as well as the total price.

iii) **RunApp.java:**
This class contains the main method to run the application.

iv) **FoodMenu.java:**
This class represents a food menu in a restaurant. It has methods to set the prices of food items, get the price of an item, add orders, remove orders, process orders, print receipts, calculate the total price of the current order, and retrieve the order items and food prices.

v) **DiscountedReceiptPrinter.java:**
It calculates and prints receipts with discounts applied to specific food items. This violates the Open/Closed Principle because I had to modify the existing code and add a new class to accommodate the new requirement. The original ReceiptPrinter class is not open for extension and requires modification to support the new functionality.

b) **Why the code violates the principle?**

The code violates the Open/Close Principle because it is not easily extensible for adding new functionality without modifying the existing code. If a new type of receipt printer or order processor needs to be added, the code would require modifications in multiple classes (FoodMenu and RunApp) to accommodate the new functionality. This violates the principle of Open/Close

Good Package code explanation:

2) Open-closed principle (Package name: solid.good.o)

a) **<u>Description of the classes and code:</u>**
Four classes and Two Interfaces: “OrderProcessorInterfaceImplementation”, “ReceiptPrinterInterfaceImplementation”, “RunApp”, “FoodMenu”, “OrderProcessor”, “ReceiptPrinter”

   i) **OrderProcessor.java:**
   This interface represents an order processor that prepares, cooks, and serves orders. It defines methods for preparing, cooking, and serving the items in an order.

   ii) **ReceiptPrinter.java:**
   This interface represents a receipt printer that prints the receipt for an order. It defines a method for printing the receipt.

   iii) **RunApp.java:**
   This class contains the main method to run the application.

   iv) **FoodMenu.java:**
   This class represents a food menu in a restaurant. It has methods to set prices for food items, retrieve prices, add orders, remove orders, process orders, print receipts, and calculate the total price of the order. The violations in this class were fixed by introducing interfaces and delegating the responsibilities of order processing and receipt printing to separate classes.

   v) **OrderProcessorInterfaceImplementation.java:**
   This class implements the OrderProcessor interface. It provides the concrete implementation for preparing, cooking, and serving orders. By separating the order processing logic into a separate class and interface, the Open/Closed Principle is adhered to, allowing for the extension of new processors without modifying the existing code.

   vi) **ReceiptPrinterInterfaceImplementation.java**
   This class implements the ReceiptPrinter interface. It provides the concrete implementation for printing the receipt. By abstracting the printing logic into a separate class and interface, the code is open for extension with new printers without modifying the existing code.

b) **<u>How are the violations fixed?</u>**

   i) Introduced interfaces (OrderProcessor and ReceiptPrinter) to abstract the responsibilities of order processing and receipt printing.

   ii) Created separate implementation classes (OrderProcessorInterfaceImplementation and ReceiptPrinterInterfaceImplementation) that implement the respective interfaces and provide the concrete implementation for order processing and receipt printing.

   iii) Modified the FoodMenu class to use the interfaces (OrderProcessor and ReceiptPrinter) instead of the concrete implementation classes directly.

This way, the code is open for extension (new processors and printers can be added by implementing the interfaces) and closed for modification (existing classes don't need to be modified when new processors or printers are added). This gives the solution to the Open/Closed Principle violation.

Bad Package code explanation:

3) Liskov Substitution Principle (Package name: solid.bad.l)

a) <u>Description of the classes and code:</u>

Five classes: “ContractEmployee”, “Employee”, “EmployeeManagementSystem”, “RunApp”, “RegularEmployee”

i) ContractEmployee.java:
This class represents a contract employee and extends the Employee class. It overrides the calculateSalary() method to calculate the salary based on contract employee rules.

ii) Employee.java:
This abstract class represents an employee. It has private fields for name, ID, and salary, along with getter and setter methods for these fields. It declares an abstract method calculateSalary() that must be implemented by subclasses.

iii) EmployeeManagementSystem.java:
It manages a list of employees and provides operations to add employees, calculate salaries, and display employee information.

iv) RegularEmployee.java:
This class represents a regular employee and extends the Employee class. It overrides the calculateSalary() method to calculate the salary based on regular employee rules

v) RunApp.java:
This class contains the main method to run the Employee Management System.

b) <u>Why the code violates the principle?</u>

In the given code, there is a violation of the Liskov substitution principle because the behavior of the calculateSalary() method in the ContractEmployee and RegularEmployee subclasses differs from the behavior described in the superclass Employee. In the Employee class, the calculateSalary() method is declared as an abstract method, indicating that subclasses must provide their own implementation. However, the subclasses ContractEmployee and RegularEmployee override the method and change the salary calculation logic according to their specific rules.

Good Package code explanation:

3) Liskov Substitution Principle (Package name: solid.good.l)

a) **Description of the classes and code:**

Five classes: "ContractEmployee", "Employee", "EmployeeManagementSystem", "RunApp", "RegularEmployee"

i) **ContractEmployee.java:**
This class represents a contract employee and extends the Employee class. It overrides the getSalary() and setSalary() methods from the parent class to handle contract employee-specific salary operations.

ii) **Employee.java:**
This abstract class represents an employee. It has private fields for name, ID, and salary. The getSalary() and setSalary() methods are declared as abstract, allowing subclasses to provide their own implementation.

iii) **EmployeeManagementSystem.java:**
This class manages a list of employees and provides operations to add employees, calculate salaries, and display employee information.

iv) **RegularEmployee.java:**
This class represents a regular employee and extends the Employee class. It overrides the getSalary() and setSalary() methods from the parent class to handle regular employee-specific salary operations.

v) **RunApp.java:**
This class contains the main method to run the Employee Management System

b) **How are the violations fixed?**

i) The getSalary() and setSalary() methods in the Employee class were changed from concrete methods to abstract methods. This change allows subclasses to provide their own implementation of these methods, ensuring that the behavior can be specialized for different types of employees.

ii) The calculateSalary() method in the Employee class was also changed to an abstract method. This enforces that subclasses must implement their own salary calculation logic, which can vary depending on the type of employee.

By making these changes, the modified code solves the Liskov Substitution Principle because the subclasses (ContractEmployee and RegularEmployee) can be used interchangeably with the parent class (Employee) without causing unexpected behavior or violating the contract defined by the parent class.

Bad Package code explanation:

4) Interface segregation principle (Package name: solid.bad.i)

a) <u>Description of the classes and code:</u>

Four classes: "GraduateStudent", "UndergraduateStudent", "Student", and "RunApp"

i) GraduateStudent.java:

Represents an undergraduate student by implementing the Student interface. It consists of a name and a list of courses and Provides implementations for the registerCourse, deregisterCourse, and showCourses methods. Also contains an additional method, printScholarshipEligibility, which prints the scholarship eligibility of the undergraduate student.

ii) UndergraduateStudent.java:

Represents a graduate student by implementing the Student interface. It consists of a name and a list of courses and Provides implementations for the registerCourse, deregisterCourse, and showCourses methods. Also contains an additional method, printResearchTopic, which prints the research topic being conducted by the graduate student.

iii) Student.java:

Represents the Student. Defines three methods: registerCourse, deregisterCourse, and showCourses.

iv) RunApp.java:

Contains the main method to run the program.

b) <u>Why the code violates the principle?</u>

The given code violates the Interface Segregation Principle (ISP) because the Student interface is not properly segregated. In this code, both UndergraduateStudent and GraduateStudent are forced to implement all the methods defined in the Student interface, even though each student type requires different behavior for certain methods.

Specifically, the methods printScholarshipEligibility and printResearchTopic are not applicable to both undergraduate and graduate students, respectively. Yet, they are included in the Student interface, which forces both classes to provide empty implementations for the methods that are not relevant to them.

Good Package code explanation:

4) Interface segregation principle (Package name: solid.good.i)

a) <u>Description of the classes and code:</u>

Three classes and Three Interfaces: “GraduateStudent”, “UndergraduateStudent”, “Student”, and “RunApp”

i) GraduateStudent.java:

Represents an undergraduate student by implementing the Student interface. It consists of a name and a list of courses and Provides implementations for the registerCourse, deregisterCourse, and showCourses methods. Also contains an additional method, printScholarshipEligibility, which prints the scholarship eligibility of the undergraduate student.

ii) UndergraduateStudent.java:

Represents a graduate student by implementing the Student interface. It consists of a name and a list of courses and Provides implementations for the registerCourse, deregisterCourse, and showCourses methods. Also contains an additional method, printResearchTopic, which prints the research topic being conducted by the graduate student.

iii) Student.java:

Represents the Student. Defines three methods: registerCourse, deregisterCourse, and showCourses

iv) RunApp.java:

Contains the main method to run the program.

b) <u>How are the violations fixed?</u>

i) The Student interface was divided into two separate interfaces: ResearchTopic and ScholarshipEligibility. This segregation allowed each class to implement only the relevant interfaces.

ii) The GraduateStudent class implemented both the Student and ResearchTopic interfaces, while the UndergraduateStudent class implemented both the Student and ScholarshipEligibility interfaces. This way, each class had access to only the methods that were specific to its type, eliminating the need to implement irrelevant methods.

Each class now depends only on the interfaces containing methods that are relevant to its behavior, resulting in better separation of concerns and improved maintainability of the codebase.

Bad Package code explanation:

5) Dependency inversion principle (Package name: solid.bad.d)

a) **<u>Description of the classes and code:</u>**
Four classes: "StudentRepository", "StudentService", "Student", and "RunApp"

   i) **StudentRepository.java:**
   The StudentRepository class acts as a repository for managing students. It maintains a list of students and provides methods to add, delete, and retrieve students.

   ii) **StudentService.java:**
   The StudentService class provides services related to student management. It has methods to add a new student, get a list of all students, delete a student, update student banner ID, and update student age.

   iii) **Student.java:**
   This class represents a Student entity with attributes such as studentName, bannerId, and studentAge. It provides getters and setters to access and modify these attributes.

   iv) **RunApp.java:**
   Contains the main method to run the program.

b) **<u>Why the code violates the principle?</u>**
The code violates the Dependency Inversion Principle because the high-level module (RunApp) depends on the low-level modules (StudentService and StudentRepository). According to DIP, both high-level and low-level modules should depend on abstractions, not concretions.

   i) RunApp directly depends on the concrete implementation of StudentService and StudentRepository classes. It creates an instance of StudentService and calls its methods directly.

   ii) If in the future, there is a need to change the implementation of StudentService or StudentRepository (e.g., using a different data storage mechanism), RunApp would also need to be modified. This tight coupling makes the system less flexible and harder to maintain.

Good Package code explanation:

5) Dependency inversion principle (Package name: solid.good.d)

a) <u>Description of the classes and code:</u>

Four classes, One Interface: “StudentRepository”, “StudentService”, “Student”, “ManageStudentRepository” and “RunApp”

i) StudentRepository.java:
This interface defines the contract for a student repository that manages students. It declares methods for adding a student, getting a list of all students, and deleting a student by banner ID.

ii) StudentService.java:
This class provides services related to student management. It takes a StudentRepository as a dependency through its constructor. It has methods to add a student, get all students, delete a student, update student banner ID, and update student age.

iii) Student.java:
This class represents a student with properties like studentName, bannerId, and studentAge. It provides getter and setter methods to access and modify these properties.

iv) ManageStudentRepository.java:
This class implements the StudentRepository interface and serves as the implementation of the repository. It uses an ArrayList to store and manage the list of students. It provides methods to add a student, retrieve all students, and delete a student using the banner ID.

v) RunApp.java:
Contains the main method to run the progra

b) <u>How are the violations fixed?</u>

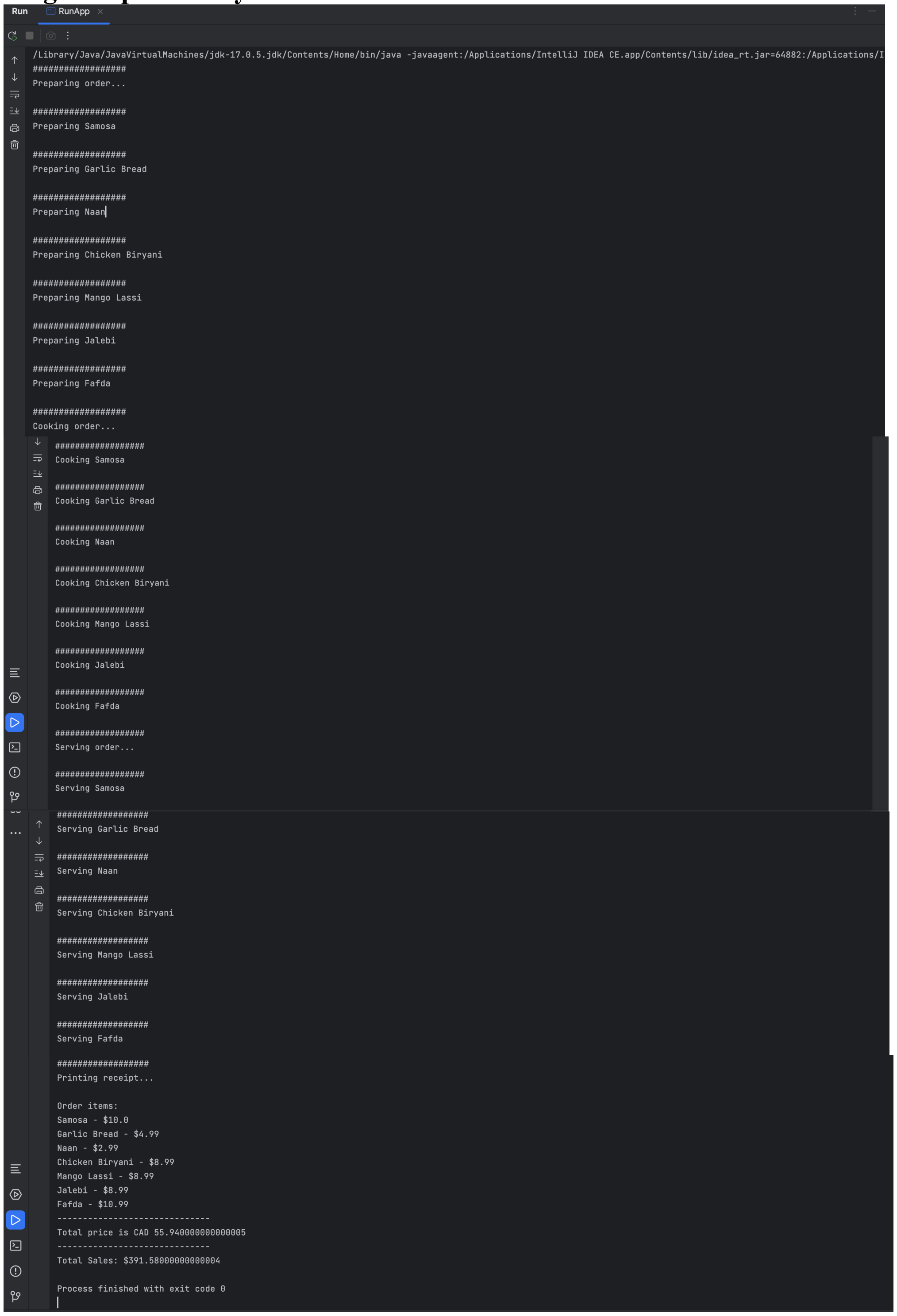
i) The code solves the Dependency Inversion principle by introducing the StudentRepository interface, which acts as an abstraction for the student repository. The StudentService class depends on this interface, rather than a specific implementation like ManageStudentRepository.

ii) This abstraction allows for flexibility and extensibility. If we need to switch to a different implementation of the student repository in the future, we can create a new class that implements the StudentRepository interface without modifying the existing code in StudentService.

This approach decouples the high-level StudentService from the low-level ManageStudentRepository, enabling easier maintenance and testing

Output of each package:

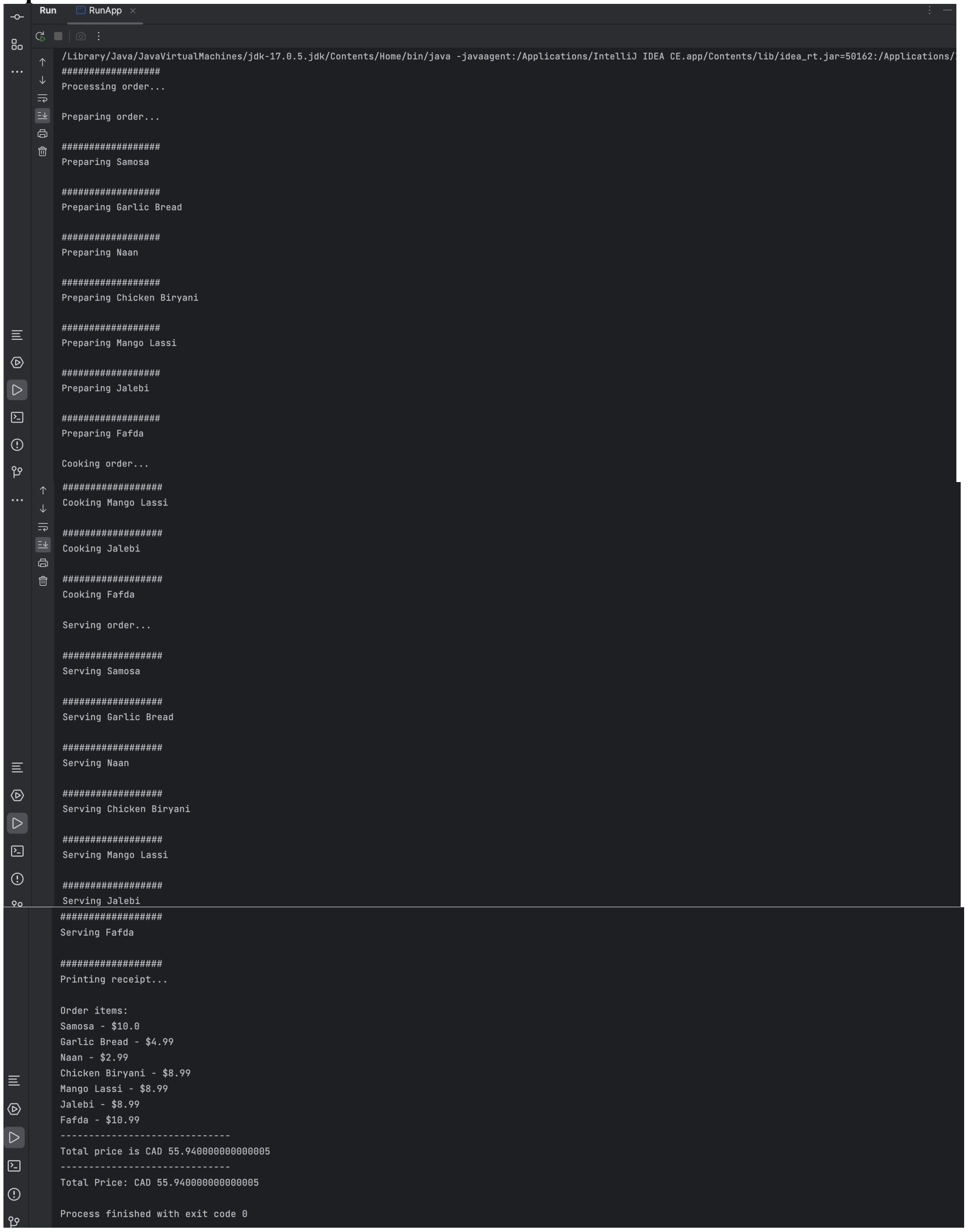
1) Single responsibility:



```
/Library/Java/JavaVirtualMachines/jdk-17.0.5.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64882:/Applications/I
##################
Preparing order...

##################
Preparing Samosa

##################
Preparing Garlic Bread

##################
Preparing Naan

##################
Preparing Chicken Biryani

##################
Preparing Mango Lassi

##################
Preparing Jalebi

##################
Preparing Fafda

##################
Cooking order...
##################
Cooking Samosa

##################
Cooking Garlic Bread

##################
Cooking Naan

##################
Cooking Chicken Biryani

##################
Cooking Mango Lassi

##################
Cooking Jalebi

##################
Cooking Fafda

##################
Serving order...

##################
Serving Samosa
##################
Serving Garlic Bread

##################
Serving Naan

##################
Serving Chicken Biryani

##################
Serving Mango Lassi

##################
Serving Jalebi

##################
Serving Fafda

##################
Printing receipt...

Order items:
Samosa - $10.0
Garlic Bread - $4.99
Naan - $2.99
Chicken Biryani - $8.99
Mango Lassi - $8.99
Jalebi - $8.99
Fafda - $10.99
------------------------------
Total price is CAD 55.940000000000005
------------------------------
Total Sales: $391.58000000000004

Process finished with exit code 0
```

Figure 1: Output of Good SRP

2) Open-closed

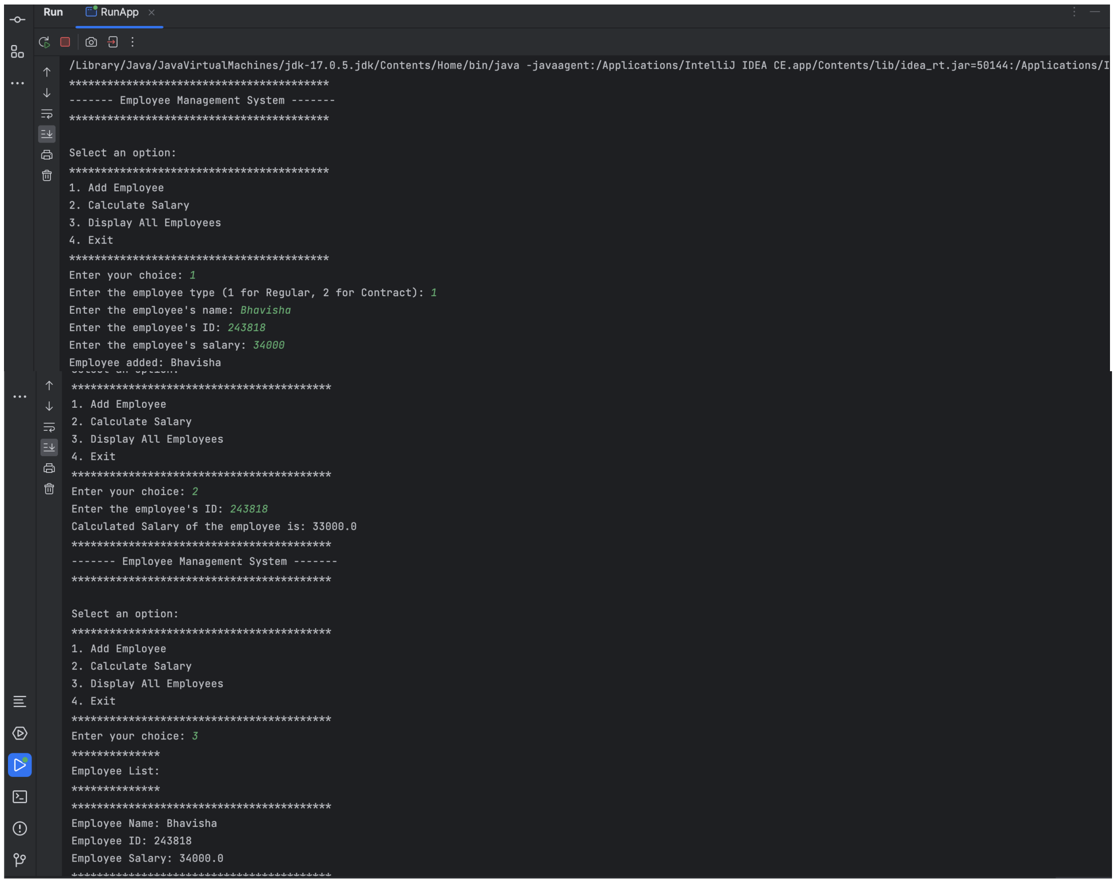


Run RunApp

```
/Library/Java/JavaVirtualMachines/jdk-17.0.5.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50162:/Applications/
##################
Processing order...

Preparing order...

##################
Preparing Samosa

##################
Preparing Garlic Bread

##################
Preparing Naan

##################
Preparing Chicken Biryani

##################
Preparing Mango Lassi

##################
Preparing Jalebi

##################
Preparing Fafda

Cooking order...

##################
Cooking Mango Lassi

##################
Cooking Jalebi

##################
Cooking Fafda

Serving order...

##################
Serving Samosa

##################
Serving Garlic Bread

##################
Serving Naan

##################
Serving Chicken Biryani

##################
Serving Mango Lassi

##################
Serving Jalebi
##################
Serving Fafda

##################
Printing receipt...

Order items:
Samosa - $10.0
Garlic Bread - $4.99
Naan - $2.99
Chicken Biryani - $8.99
Mango Lassi - $8.99
Jalebi - $8.99
Fafda - $10.99
------------------------------
Total price is CAD 55.940000000000005
------------------------------
Total Price: CAD 55.940000000000005

Process finished with exit code 0
```

Figure 2: Output of Good OCP

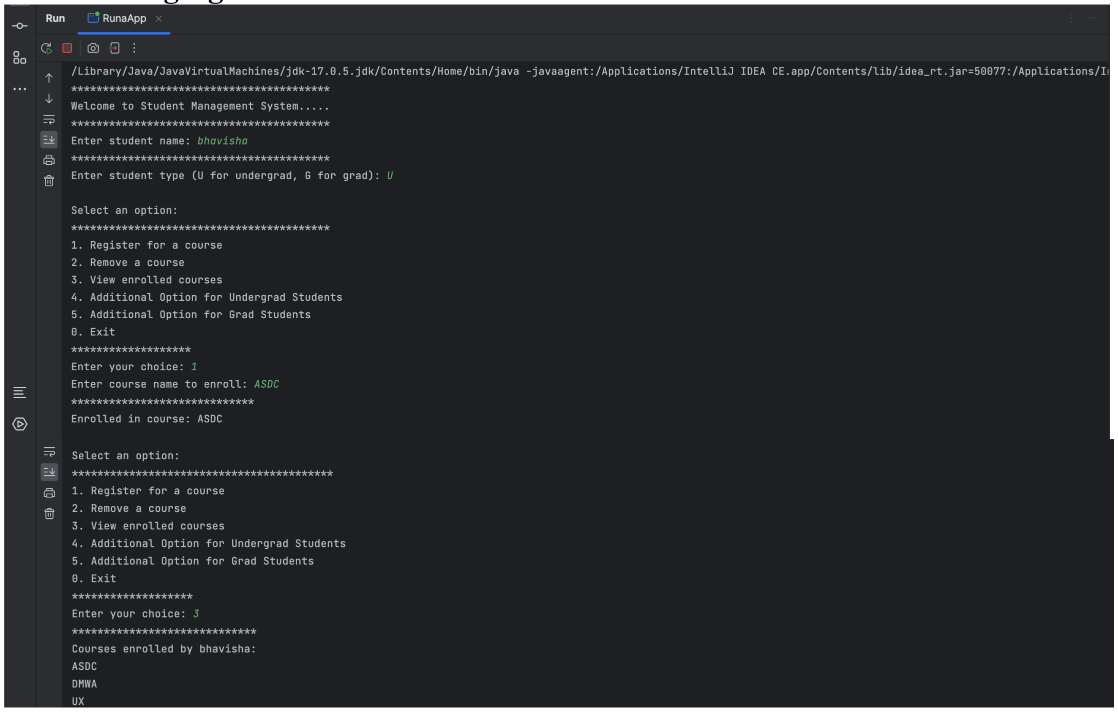
3) Liskov substitution



```
/Library/Java/JavaVirtualMachines/jdk-17.0.5.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50144:/Applications/I
******************************************
------- Employee Management System -------
******************************************

Select an option:
******************************************
1. Add Employee
2. Calculate Salary
3. Display All Employees
4. Exit
******************************************
Enter your choice: 1
Enter the employee type (1 for Regular, 2 for Contract): 1
Enter the employee's name: Bhavisha
Enter the employee's ID: 243818
Enter the employee's salary: 34000
Employee added: Bhavisha
******************************************
1. Add Employee
2. Calculate Salary
3. Display All Employees
4. Exit
******************************************
Enter your choice: 2
Enter the employee's ID: 243818
Calculated Salary of the employee is: 33000.0
******************************************
------- Employee Management System -------
******************************************

Select an option:
******************************************
1. Add Employee
2. Calculate Salary
3. Display All Employees
4. Exit
******************************************
Enter your choice: 3
**************
Employee List:
**************
******************************************
Employee Name: Bhavisha
Employee ID: 243818
Employee Salary: 34000.0
```

Figure 3: Output of Good LSP

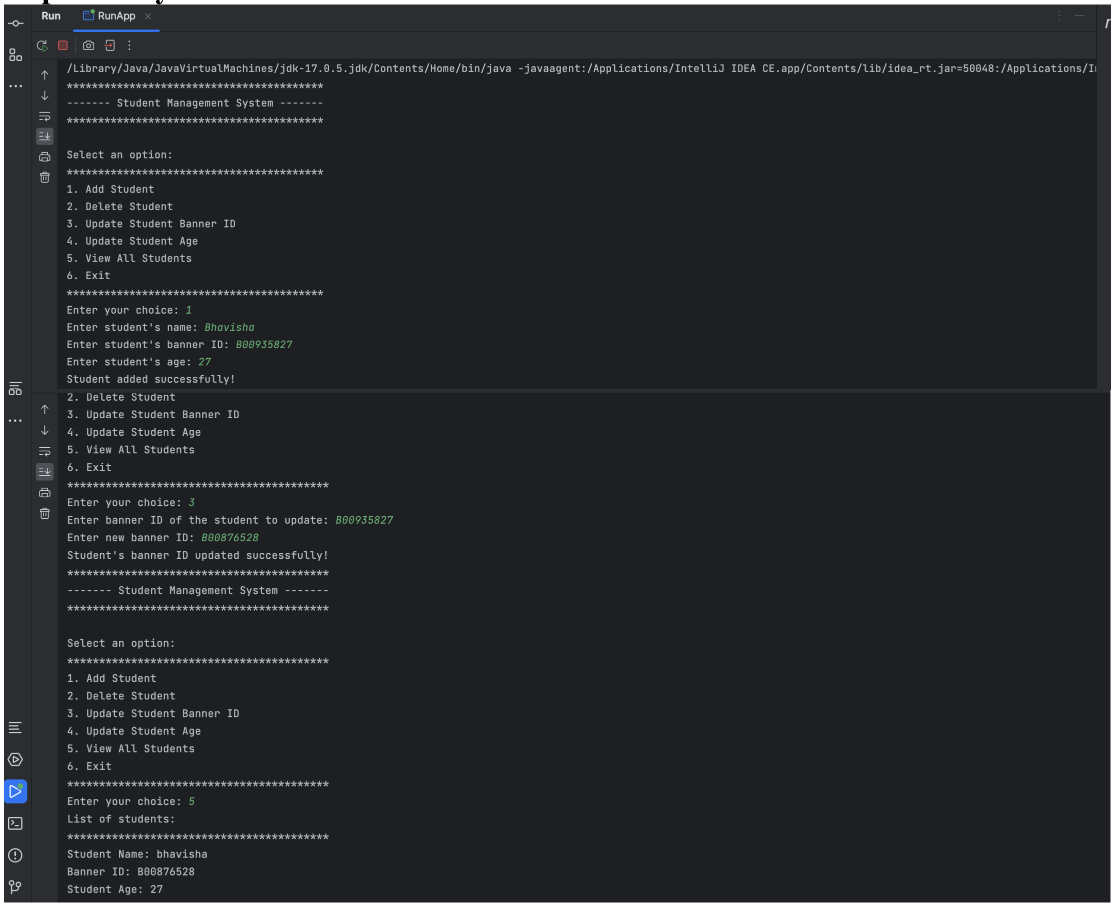
4) Interface segregation



```
/Library/Java/JavaVirtualMachines/jdk-17.0.5.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50077:/Applications/I
******************************************
Welcome to Student Management System.....
******************************************
Enter student name: bhavisha
******************************************
Enter student type (U for undergrad, G for grad): U

Select an option:
******************************************
1. Register for a course
2. Remove a course
3. View enrolled courses
4. Additional Option for Undergrad Students
5. Additional Option for Grad Students
0. Exit
******************
Enter your choice: 1
Enter course name to enroll: ASDC
****************************
Enrolled in course: ASDC

Select an option:
******************************************
1. Register for a course
2. Remove a course
3. View enrolled courses
4. Additional Option for Undergrad Students
5. Additional Option for Grad Students
0. Exit
******************
Enter your choice: 3
****************************
Courses enrolled by bhavisha:
ASDC
DMWA
UX
```

Figure 4: Output of Good ISP

5) Dependency inversion



```
/Library/Java/JavaVirtualMachines/jdk-17.0.5.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=50048:/Applications/I
*****************************************
------- Student Management System -------
*****************************************

Select an option:
*****************************************
1. Add Student
2. Delete Student
3. Update Student Banner ID
4. Update Student Age
5. View All Students
6. Exit
*****************************************
Enter your choice: 1
Enter student's name: Bhavisha
Enter student's banner ID: B00935827
Enter student's age: 27
Student added successfully!
```

```
2. Delete Student
3. Update Student Banner ID
4. Update Student Age
5. View All Students
6. Exit
*****************************************
Enter your choice: 3
Enter banner ID of the student to update: B00935827
Enter new banner ID: B00876528
Student's banner ID updated successfully!
*****************************************
------- Student Management System -------
*****************************************

Select an option:
*****************************************
1. Add Student
2. Delete Student
3. Update Student Banner ID
4. Update Student Age
5. View All Students
6. Exit
*****************************************
Enter your choice: 5
List of students:
*****************************************
Student Name: bhavisha
Banner ID: B00876528
Student Age: 27
```

Figure 5: Output of Good DIP

References:

[1] “Week 6 S.O.L.I.D. (Jun 6,8,9),” *Brightspace Dalhousie University* [Online]. Available: https://dal.brightspace.com/d2l/le/content/271676/Home .[Accessed: June 28, 2023].

[2] S. Desai, “All you Need to Know About Solid Principles in Java,” *edureka* [Online]. Available: https://www.edureka.co/blog/solid-principles-in-java/ , 2021. [Accessed: July 02, 2023].