Final Project Report

SWE 4301 - Object Oriented Concepts II

Submitted by:

Adid-Al-Mahamud Shazid

Student ID: **210042172**
Programme: **SWE**
Department: **CSE**

Islamic University of Technology

January 19, 2024

SplitWise

SplitWise is a comprehensive expense management platform designed to simplify the way users handle their personal and group finances. With a user-friendly interface, SplitWise enables users to sign up, add friends, and effortlessly log expenses, whether shared among a group or involving individual friends. The system provides a transparent overview of balances, facilitating easy debt settlement and accurate tracking of financial interactions. SplitWise empowers users with a versatile menu interface, allowing seamless navigation through features such as adding and removing friends, as well as managing both group and non-group expenses. It stands as an intuitive solution, offering users a straightforward and organized approach to managing their financial transactions within their social circles.

Features:

1. **User Management:**
    a. Users can sign up with a username and password.
    b. Users can log in to their accounts.
    c. The system maintains a list of users.
2. **Expense Tracking:**
    a. Users can add both group and non-group expenses.
    b. Group expenses involve multiple users, each contributing to a shared expense.
    c. Non-group expenses involve one user and one friend, where they share the expense.
3. **Friend Management:**
    a. Users can add and remove friends.
    b. Friend relationships are maintained both in a common friends list and a group friends list.
4. **Balance Tracking:**
    a. The system calculates and displays balances for both group and non-group expenses.
    b. For group expenses, it shows who paid and how much each person owes.
    c. For non-group expenses, it shows individual balances for each user involved.

How I Implemented SOLID Principles:

1. Single Responsibility Principle (SRP):

A class should have only one reason to change, meaning that a class should have only one responsibility or job. Each class or module should be responsible for a single aspect or functionality of the software. This makes the software more modular, easier to understand, and less prone to bugs when changes are made.

The Main class of un-refactored code had multiple responsibilities such as managing the user interface, handling user input, and interacting with user-related functionalities like login, sign-up, etc. So, I Separated these responsibilities into distinct classes to ensure each class has a single responsibility. In my refactored code:

- **User** class represents a user and maintains details such as name, password, groupAmount, and lists of commonFriends, groupFriends, commonExpenses, and groupExpenses.

- **Expenses** class holds information about an expense, such as friendName, description, whoPaid, myAmount, friendAmount, and totalAmount.

- Operation classes (**GroupExpenses, NonGroupExpenses, GroupOperations**, **NonGroupOperations**, **ManageFriends**, **UserMenu**, **SignUp**, **Login**, **HomeMenu**) handle specific functionalities, like adding expenses, managing friends, handling user menus, and authentication.

2. Open/Closed Principle (OCP):

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. Existing code should not be modified when new functionality is added. Instead, new functionality should be added through the extension of existing code, typically by using interfaces or abstract classes. This promotes code reuse and makes the system more adaptable to change.

In the Main class of un-refactored code, the menu options and related logic are hardcoded within the methods. Adding a new menu option or modifying existing behavior would require changing the existing code. After Refactoring, I can easily introduce new methods in my code.

I have introduced an Interface **IExpenses** which leads me to use the **addExpense()** and **balances()** method anywhere I want and modification is not necessary as well. Also, In the **groupExpenses()** method, the functionality is organized into a switch statement where each case corresponds to a specific behavior. Now, let's say I want to add a new feature, such as displaying statistics for group expenses. I can easily create a new method for this feature without modifying the existing **UserMenu** class.

Thus, I have made the code Open for Extension and closed for modification.

3. Liskov Substitution Principle (LSP):

Subtypes must be substitutable for their base types without altering the correctness of the program. Derived classes or subclasses should be able to replace their base classes without affecting the correctness of the program. This ensures that objects of a base class and its subclasses can be used interchangeably without causing errors.

I didn’t make any use of LSP in my code.

4. Interface Segregation Principle (ISP):

A class should not be forced to implement interfaces it does not use. Clients should not be forced to depend on interfaces they do not use. Instead, large interfaces should be broken down into smaller, more specific interfaces, and classes should only implement the interfaces that are relevant to their behavior. This prevents unnecessary dependencies and ensures that classes only depend on what they need.

The **IExpenses** interface is used to segregate methods related to expenses. Classes that need to handle expenses implement this interface, and they only need to provide implementations for the methods relevant to their responsibilities (**addExpenses()** and **balances()**).

5. Dependency Inversion Principle (DIP):

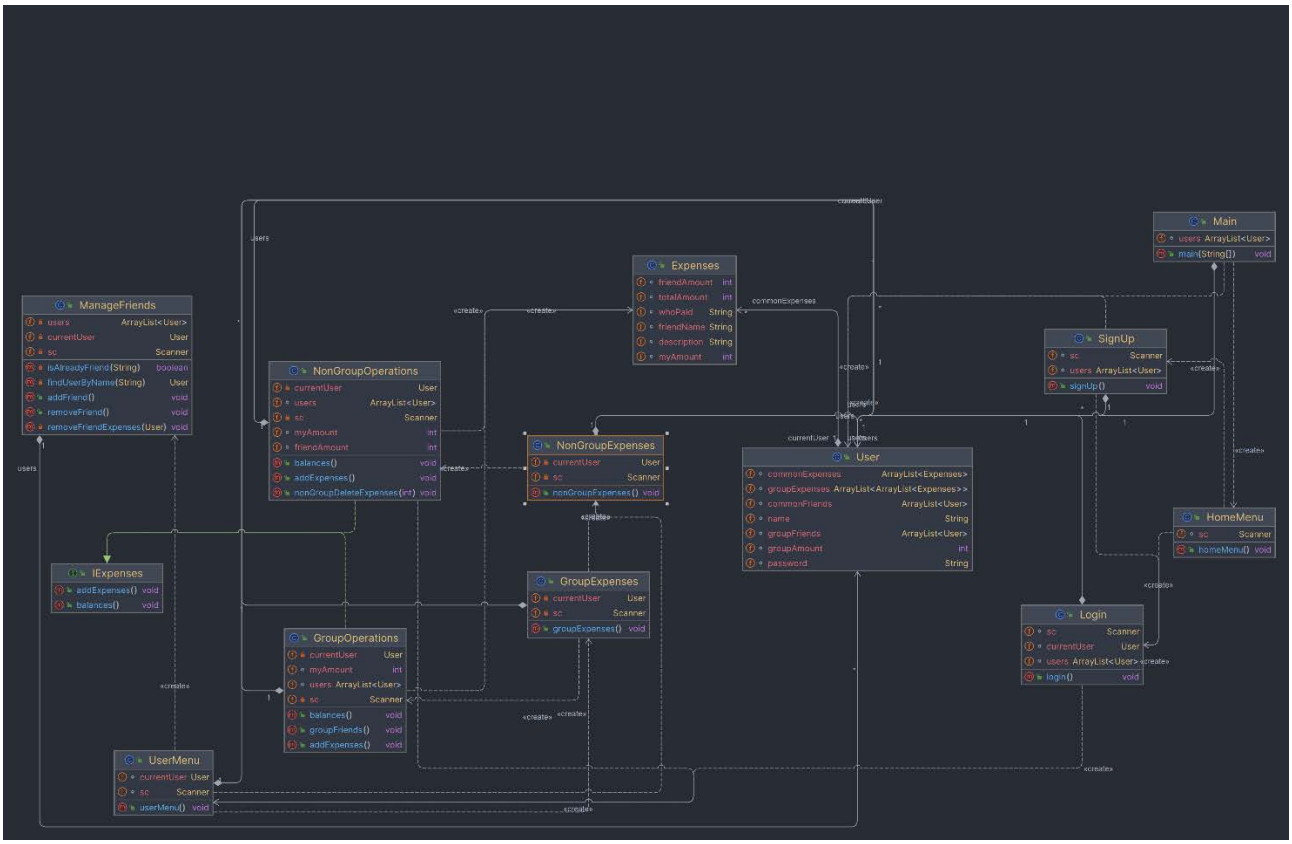
High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. Dependency inversion promotes decoupling between high-level and low-level modules by introducing abstractions. This allows changes in low-level modules to be made without affecting high-level modules and vice versa. Abstractions (interfaces, abstract classes) play a crucial role in achieving this decoupling.

The Main class of un-refactored code is tightly coupled, where direct instantiation of concrete classes like Login, SignUp, UserMenu, etc., is happening. So, I consider using dependency injection to decouple classes and promote better testability and flexibility.

- High level Modules (**GroupExpenses, NonGroupExpenses, GroupOperations, NonGroupOperations, ManageFriends, UserMenu, SignUp, Login, HomeMenu**) are depending on **User** class which is abstraction instead of concrete classes.
- High level modules (**GroupOperations, NonGroupOperations**) are depending on **IExpenses** interface which is also an abstraction.

This allows flexibility in changing or extending implementations without affecting the high-level modules.

UML Diagram:


Main
users ArrayList<User>
main(String[]) void
ManageFriends
users ArrayList<User>
currentUser User
sc Scanner
isAlreadyFriend(String) boolean
findUserByName(String) User
addFriend() void
removeFriend() void
removeFriendExpenses(User) void
Expenses
friendAmount int
totalAmount int
whoPaid String
friendName String
description String
myAmount int
commonExpenses
SignUp
sc Scanner
users ArrayList<User>
signUp() void
NonGroupOperations
currentUser User
users ArrayList<User>
sc Scanner
myAmount int
friendAmount int
balances() void
addExpenses() void
nonGroupDeleteExpenses(int) void
NonGroupExpenses
currentUser User
sc Scanner
nonGroupExpenses() void
User
commonExpenses ArrayList<Expenses>
groupExpenses ArrayList<ArrayList<Expenses>>
commonFriends ArrayList<User>
name String
groupFriends ArrayList<User>
groupAmount int
password String
HomeMenu
sc Scanner
homeMenu() void
IExpenses
addExpenses() void
balances() void
GroupExpenses
currentUser User
sc Scanner
groupExpenses() void
GroupOperations
currentUser User
myAmount int
users ArrayList<User>
sc Scanner
balances() void
groupFriends() void
addExpenses() void
Login
sc Scanner
currentUser User
users ArrayList<User>
login() void
UserMenu
currentUser User
sc Scanner
userMenu() void
«create»
users
currentUser

Code smells that were present in un-refactored Code:

1. **Duplicate Code:**

There was some duplicated code for handling user input and managing menus across different methods. So, I used separate classes to handle common functionality and reduce code duplication.

2. **Conditional Complexity:**

The nested conditional structures in methods like **nonGroupBalances** and **groupBalances** could lead to increased cognitive complexity. So, I Refactored complex conditionals by breaking them down into smaller, more manageable parts or using polymorphism to simplify the logic.

3. **God Class:**

The **Main** class had multiple responsibilities, including managing user input, handling menus, and controlling the overall flow of the program. This violates the Single Responsibility Principle (SRP) and can lead to a monolithic and difficult-to-maintain design. So, I Refactored the **Main** class to delegate responsibilities to separate classes or components.

4. **Inconsistent Naming:**

Some field name was not following proper naming convention. So, I changed the names such that the names are understandable.

5. **Shotgun Surgery:**

Shotgun Surgery occurs when a small change in one part of the codebase requires changes in many other parts. If I decide to change the structure or behavior of the **User** class, it may require modifications in several places throughout the code. For example, changes related to friends, group expenses, and common expenses. So, I Considered encapsulating related functionalities into separate classes to reduce the impact of changes in one area on other parts of the code.

6. **Black Sheep:**

A Black Sheep occurs when a class or module doesn't fit well with the overall design or follows a different pattern compared to the rest of the code. The Expenses class is defined as a static inner class inside the User class, but it is treated differently than other classes. It's directly referenced from methods in Main. So, I made a separate class file for Expenses for providing a good design.

7. **Data Clumps:**

Data Clumps happen when the same group of variables are passed together in multiple places. The code frequently deals with expenses, and related variables like **des**, **totAmt**, and **whoPaid** are often clumped together. So, I created an **Expense** class, to avoid passing multiple variables around.

There are other code smells present as well. I just mentioned all I could find.

Design smells that were present in un-refactored Code:

1. **Rigidity:**

Rigidity occurs when the code is hard to change because a small modification requires changes in many places. The code exhibit rigidity, especially in the **nonGroupExpenses** and **groupExpenses** sections. For example, if I decide to change the structure of expenses or how they are handled, it might require modifications in several parts of the code. So, I considered making several classes to get over this problem.

2. **Fragility:**

Fragility occurs when small changes in the code unintentionally break other parts of the system. The code is fragile, especially in methods like **nonGroupBalances** and **groupBalances**, where there's a potential risk of unintended consequences due to the tight coupling between different components. I just improved encapsulation to get rid of it.

3. **Viscosity:**

Viscosity refers to the effort required to make changes. High viscosity means it's difficult to do the right thing. The code exhibit viscosity, particularly in methods where there is a mix of user input handling, data processing, and interaction with other classes. Separation of concerns ensured a better design pattern.

4. **Immobility:**

Immobility occurs when it's challenging to reuse or move code to other parts of the system. The code is immobile, especially due to the tight coupling between the **Main** class and the **User** class. The **User** class has a lot of responsibilities, making it less reusable in other contexts. I made the functionality smaller and usable.

5. **Needless Complexity:**

Needless Complexity occurs when the code is more complex than necessary, making it harder to understand and maintain. The code has some areas, like the expense handling logic, that might be considered complex and could benefit from simplification. So, I Simplified complex parts of the code, used design patterns where appropriate.

6. **Opacity:**

Opacity refers to code that is difficult to understand, lacking clear documentation or meaningful identifiers. The code could benefit from more meaningful variable names, comments, and documentation, especially in complex sections like the expense handling logic. So, I changed variable and method names to make the code readable.

Github Link:

https://github.com/adid/Code-Smell-Project-OOC-II.git