Split Wise Demo

It is a simulation of the Bill Splitting App using JAVA.

The complete app is made using JAVA, and no other software was used except VS Code and Eclipse. The app uses Command Line Interface.

Primary Functions of the App:

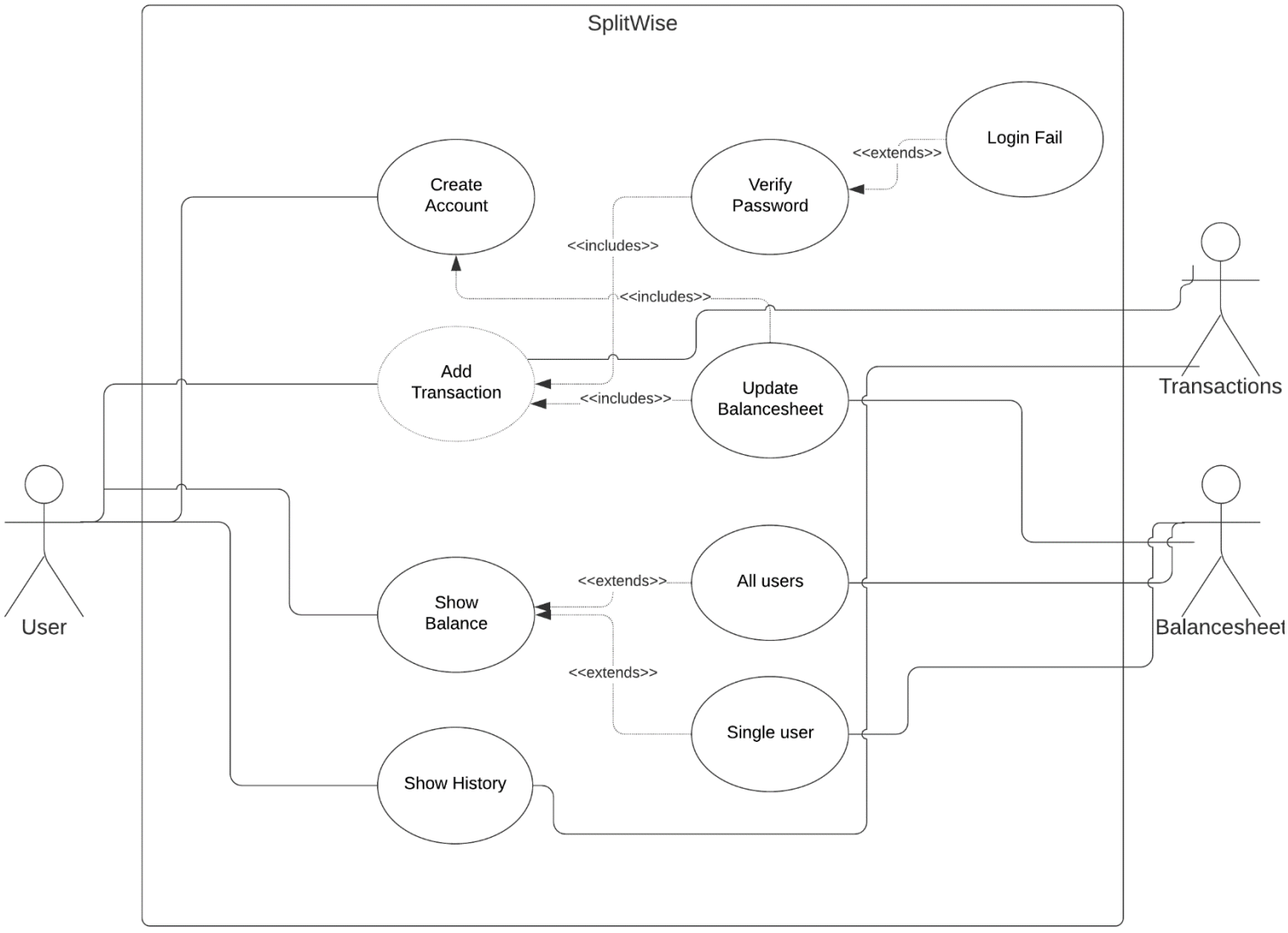
- Add users to form a group to facilitate an easy way to keep track of balances.
- Splitting the balances into three types, i.e., Equal, Exact and Percent.
- Storing the balances and showing them whenever required.
- Storing the transaction history.
- A way to update the balance sheet when a certain amount of due is paid back.
- Ability to make some basic user customization like password change and name change.

Other possible features:

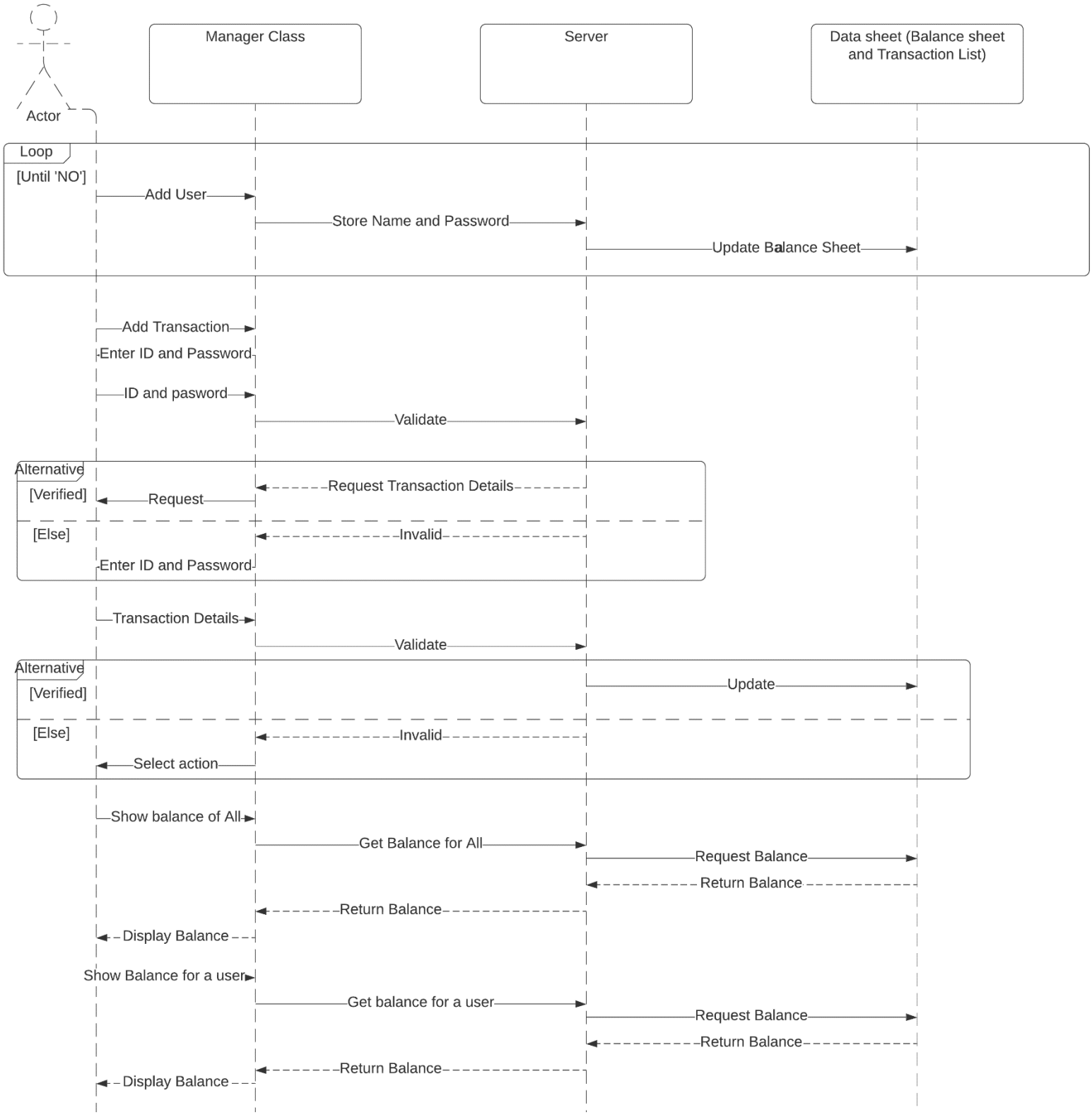
- A GUI for better user experience.
- Integrated payment system to pay off the dues.
- A method to simplify the balances, like if User1 owes $50 to User2 and User3 owes $50 to User1, we can simplify it to the statement User3 owes $50 to User2 to reduce the no. of transactions.

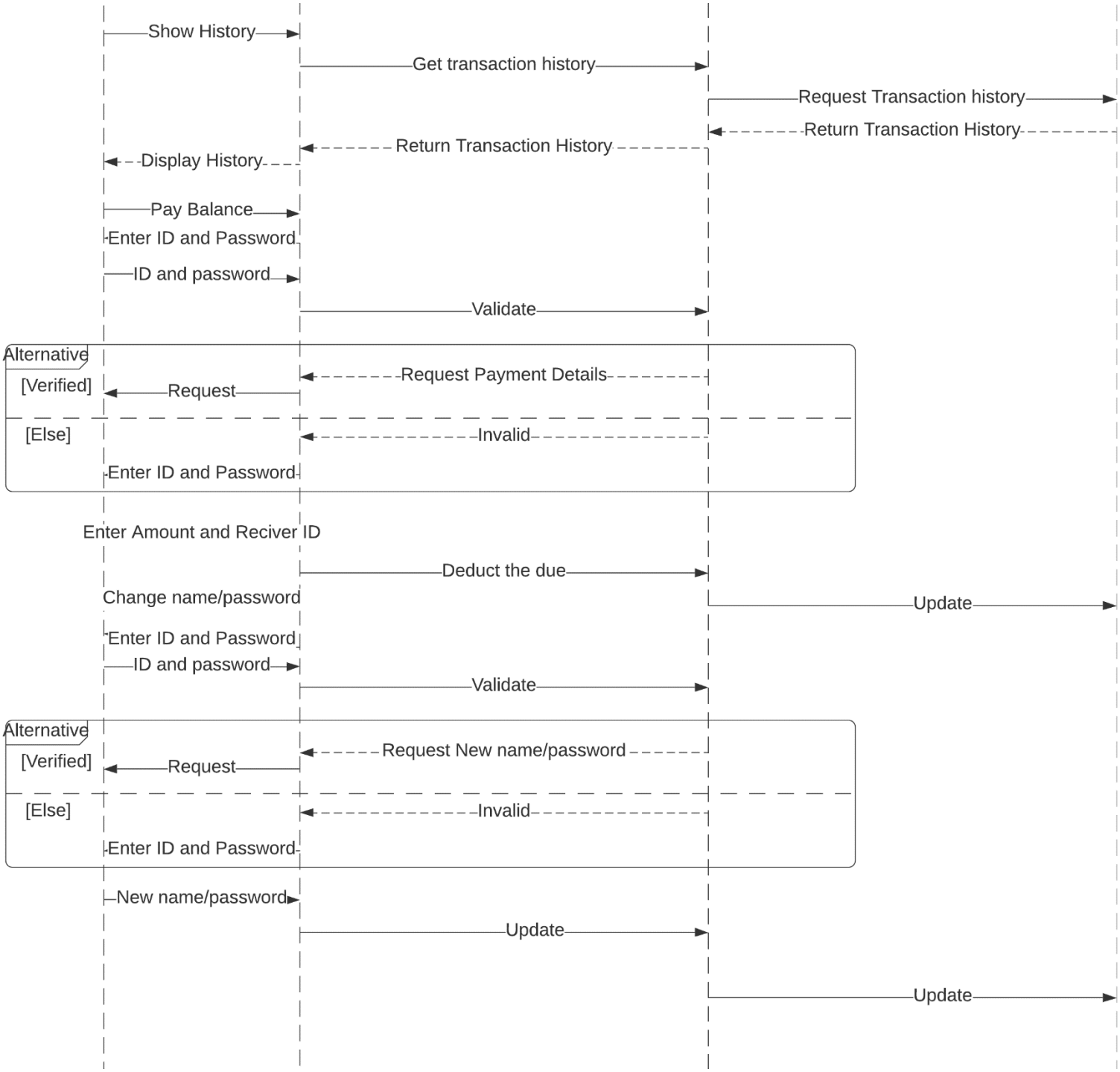
UML class diagram

Use case diagram



UML Sequence diagram




Show History
Get transaction history
Request Transaction history
Return Transaction History
Return Transaction History
Display History
Pay Balance
Enter ID and Password
ID and password
Validate
Alternative
[Verified]
Request Payment Details
Request
[Else]
Invalid
Enter ID and Password
Enter Amount and Reciver ID
Deduct the due
Change name/password
Update
Enter ID and Password
ID and password
Validate
Alternative
[Verified]
Request New name/password
Request
[Else]
Invalid
Enter ID and Password
New name/password
Update
Update

OOP principles:

1. Encapsulation:
   In this project, the primary thing changing was the 'split type'. There are three types of splits allowed, namely, Equal, Exact and Percent. The three categories work fine for now, but if we decide to add/remove any category, we must change the complete code, including the one in Driver and Manager class. One can form the different split types into subclasses of the transaction class, making it possible to add/remove categories easily.

2. Composition over inheritance:
   This project can pass this test as the User reference in the transaction class was used instead of inheriting it. Using this composition helps as the transaction class and user class methods are now independent of each other yet accessible in both classes.

3. Programme to interface not implementation
   Again when creating the transaction class, we can instead create a transaction class with multiple subclasses.

4. Loose coupling between objects:
   The objects/classes are tightly coupled, as is evident by the lack of interfaces to interact. The problem can be fixed by adding a few interfaces for each class that can initialize using the class's constructors implementing them.

5. Classes – open for expansion and closed for modification:
   The principle that code must be added to add features instead of changing the old code is not followed thoroughly. One can argue that adding extra features to the Main Menu is just an extra switch case. Also, adding extra split types with just extra code in the validate method of the Transaction class can work. However, if we need to add a unique feature like the 'simplify method' mentioned earlier, it will be difficult without altering the

current code. Still, one can argue that the code follows the property for all the known types of additions.

6. Depend on abstraction, not on a concrete class:
   This property is not followed as all the classes present are concrete. The classes are instantiated directly using concrete class references. There is no abstract reference.

Let us look at one of the Design Patterns.

Abstract Factory and Factory Method: It is a creational pattern, i.e., it handles the creation of new classes. For this code, the possible way to implement it was with the creation of transactions. It also ensures encapsulation and decoupling of the superclass from the subclass during object creation. Instead of creating the 'splitType' as a string datatype, we could give it the status of a particular class with different types being its subclass. E.g.:

```
Class Transaction{
        SplitType splitType;
        splitType = defineType(type);
        abstract SplitType(string type);
}
```

Now, this would instantiate splitType to different classes based on the type of split performed. The exact logic behind the code could be kept static, and we can add/remove types permissible in our app.

GitHub link: https://github.com/Parzival-009/SplitWise_Demo

Video link: https://drive.google.com/drive/folders/1EYDegH5beUiiLH4THKtOwJ92s_-8yksB?usp=sharing

Test Case:

User1

111111

123

Y

User2

222222

123

Y

User3

333333

123

Y

User4

444444

123

NO

1

111111

1000

4

111111

222222

333333
444444
EQUAL
111111
123
2
1
1
444444
500
2
111111
222222
300
200
444444
123
2
1
3
4
111111
123
300
444444
2
2

111111

5

1

111111

123

Alex

2

1

1

123456

1

111111

500

2

222222

333333

PERCENT

50

40

111111

123

1

222222

100

2

111111

222222

EQUAL

222222

147

6

0

Aditya Sheth

2020S7PS1511P

Project 17 Split Wise