Software design specification document

2023

Project Team

ID	Name	Email
20210428	Nour Muhammed	nourmuhammad835@gamil.com
20210387	Mariam Haitham	haithammariam36@gmail.com
20210133	Rana Essam	ranaessam03@gmail.com
20210499	Noor Eyad	nooreyadd39@gmail.com

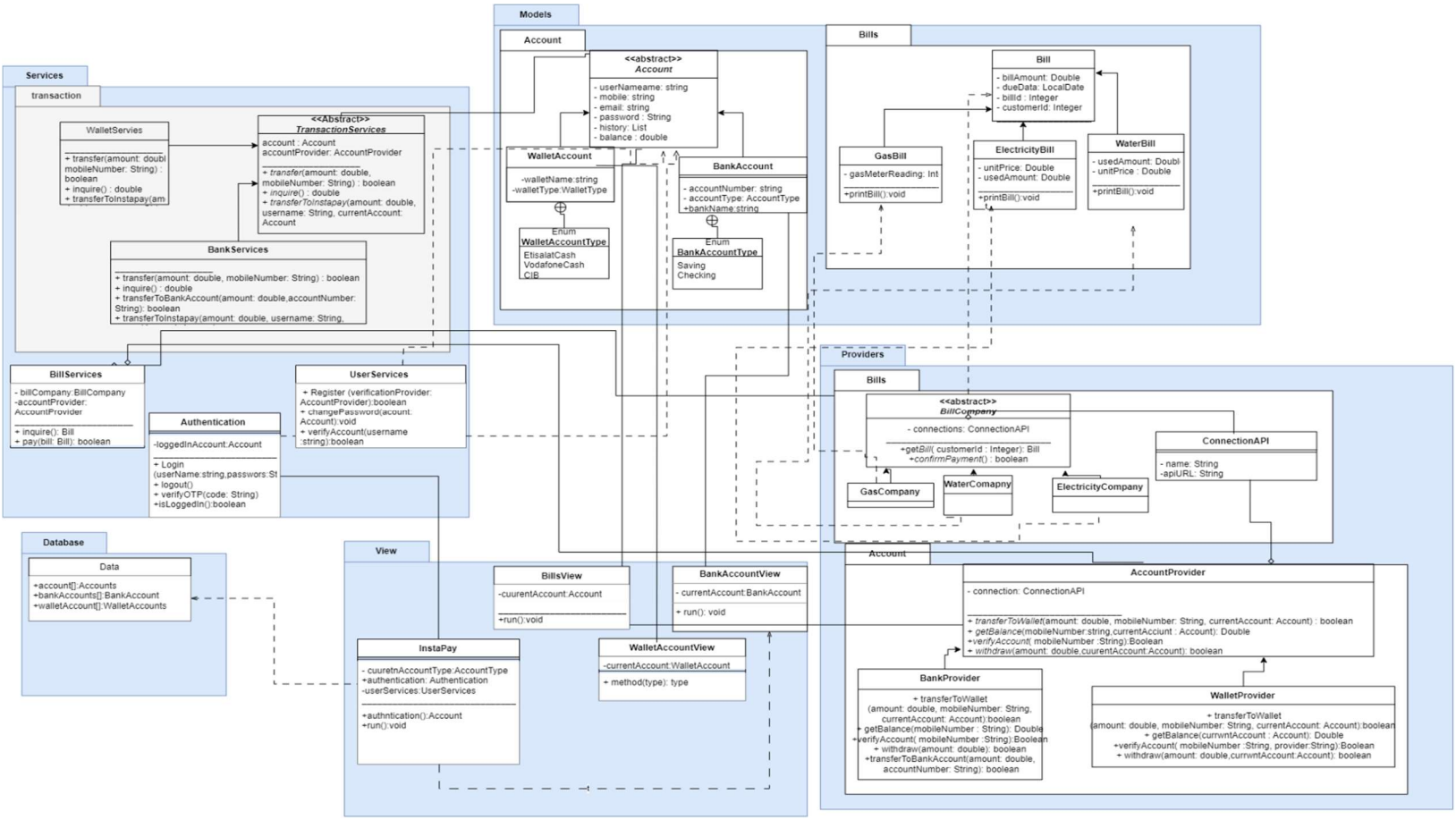
CS352: Sprint SDS – NRM - Online Payment System

SDS document

Contents

SDS document

Class diagram design



SDS document

Class diagram Explanation

1. Factory Method Design Pattern:

As we know, factory pattern aims to solve a fundamental problem in instantiation (the creation of a concrete object of a class) in OOP. In principle, creating an object directly within the class that needs or should use this object is possible, but very inflexible. It binds the class to this object and makes it impossible to change the instantiation independently of the class. This kind of code is avoided in the factory pattern approach by first defining a separate operation for creating the object – the factory method. As soon as this is called up, it generates the object instead of the class constructor already mentioned.

We can see the application of the factory pattern in our code in bills section. The factory abstract class being the BillComapny class, which has 3 different classes – GasCompany, WaterCompany, ElectricityCompany - that inherits from it to implement the getBill and ConfirmPayment functions according to the classes themselves and how they use them, all three of these classes representing the concrete factory classes. As for the abstract product class, it is represented by the Bill class, which has 3 different classes – GasBill, WaterBill, ElectricityBill – that inherits from it to create a bill according to the type the user wants and implement the printBill function that differs from a bill to another, all three of these classes represent the concrete product classes.

2. Strategy Pattern:

We're familiar with the strategy pattern and that it separates the behavior of an object from the object itself. The behavior is encapsulated into different strategies, each with its own implementation of the behavior. The context maintains a reference to a strategy object and interacts with it through a common interface. At runtime, the context can swap the current strategy with another one, effectively changing the object's behavior, which ensures flexibility and maintainability of the code.

This pattern is seen in our code in 2 main parts:

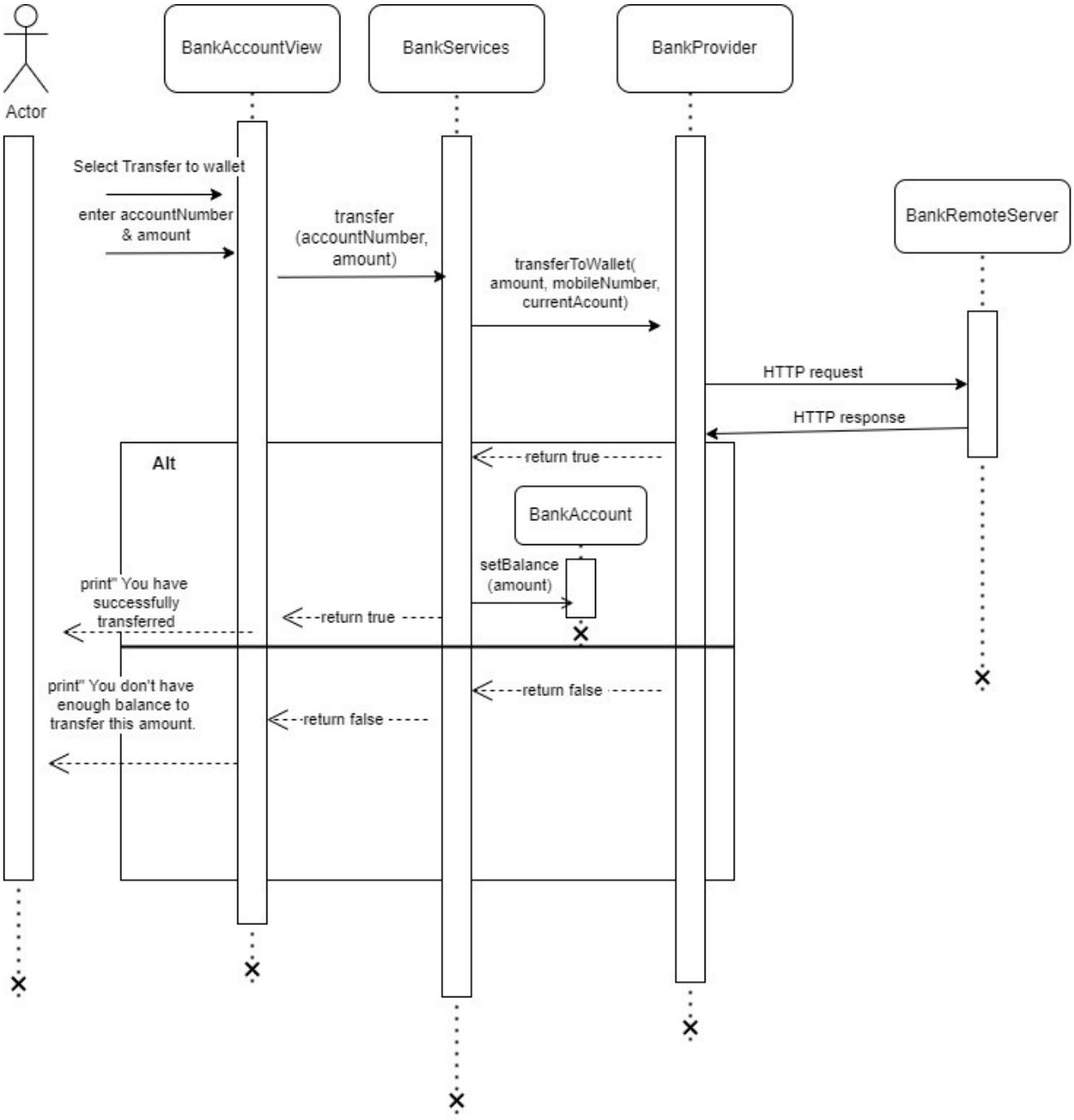
a. The first time is in AccountProvoder class, it being the strategy interface, and then the concrete strategies of it are the WalletProvider and BankProvider, them implementing verifyAccount, getBalance, withdraw and transferToWallet functions, all these functions being in both concrete classes, and an extra function – transferToBank - in the BankProvider class.

b. The second time is in Account class, it being the strategy interface, and then the concrete strategies of it are the BankAccount and WalletAccountClass, them extending the Account class as an entity/model to make sure that all the attributes that differ between both of them are separate.

SDS document

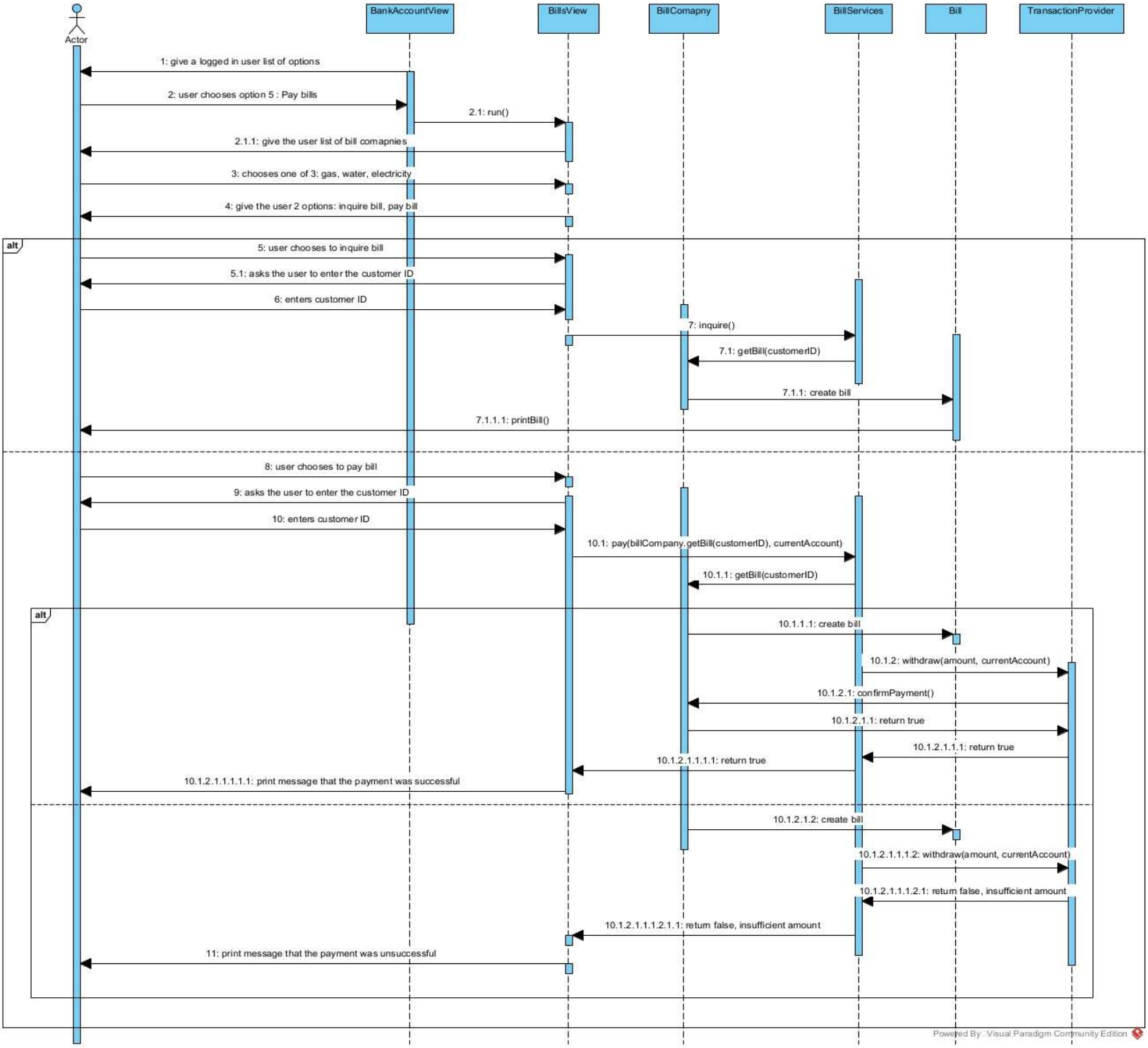
Sequence diagram design

1. Transferring from bank to wallet sequence diagram:



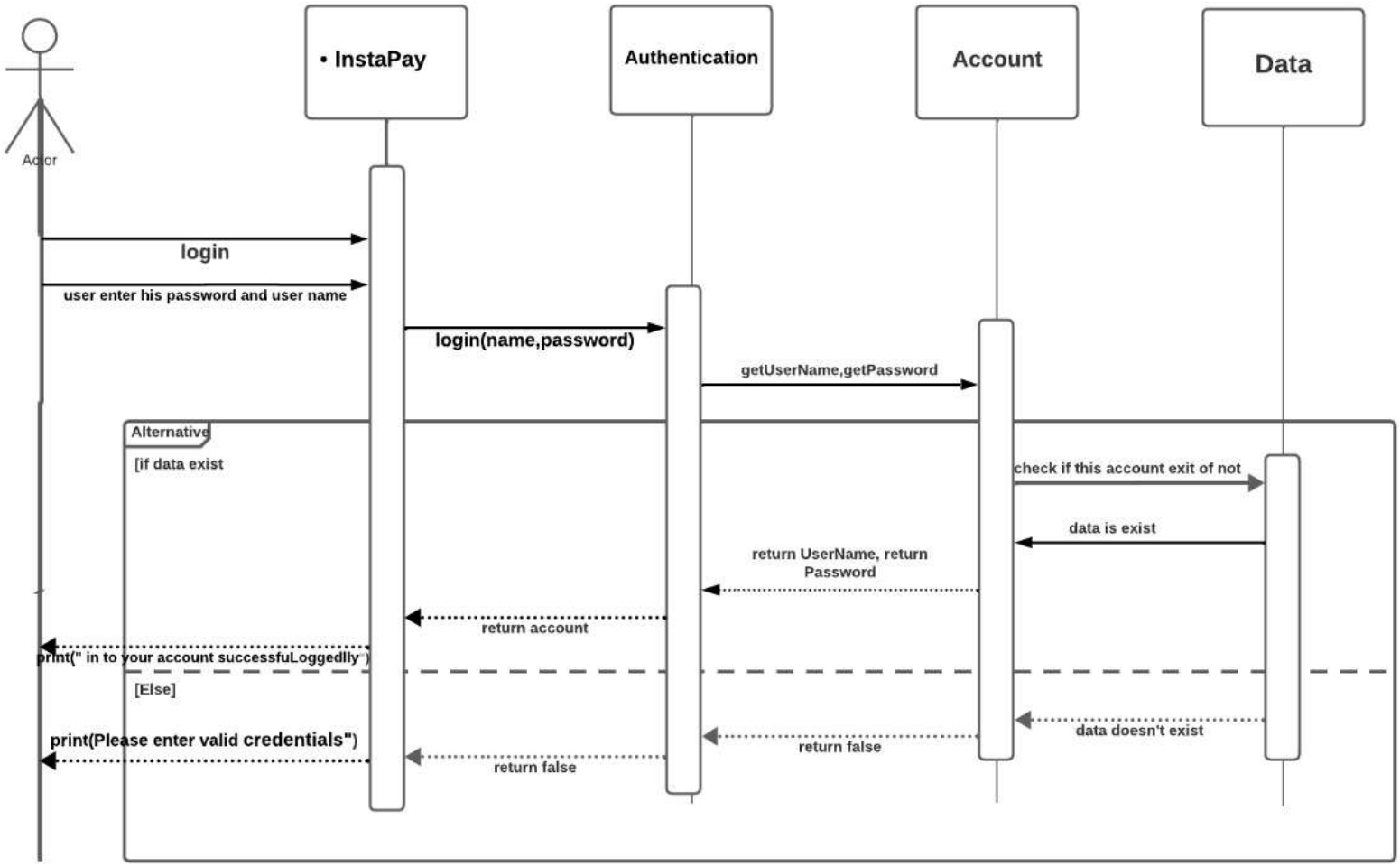
SDS document

2. Choosing to pay a bill:



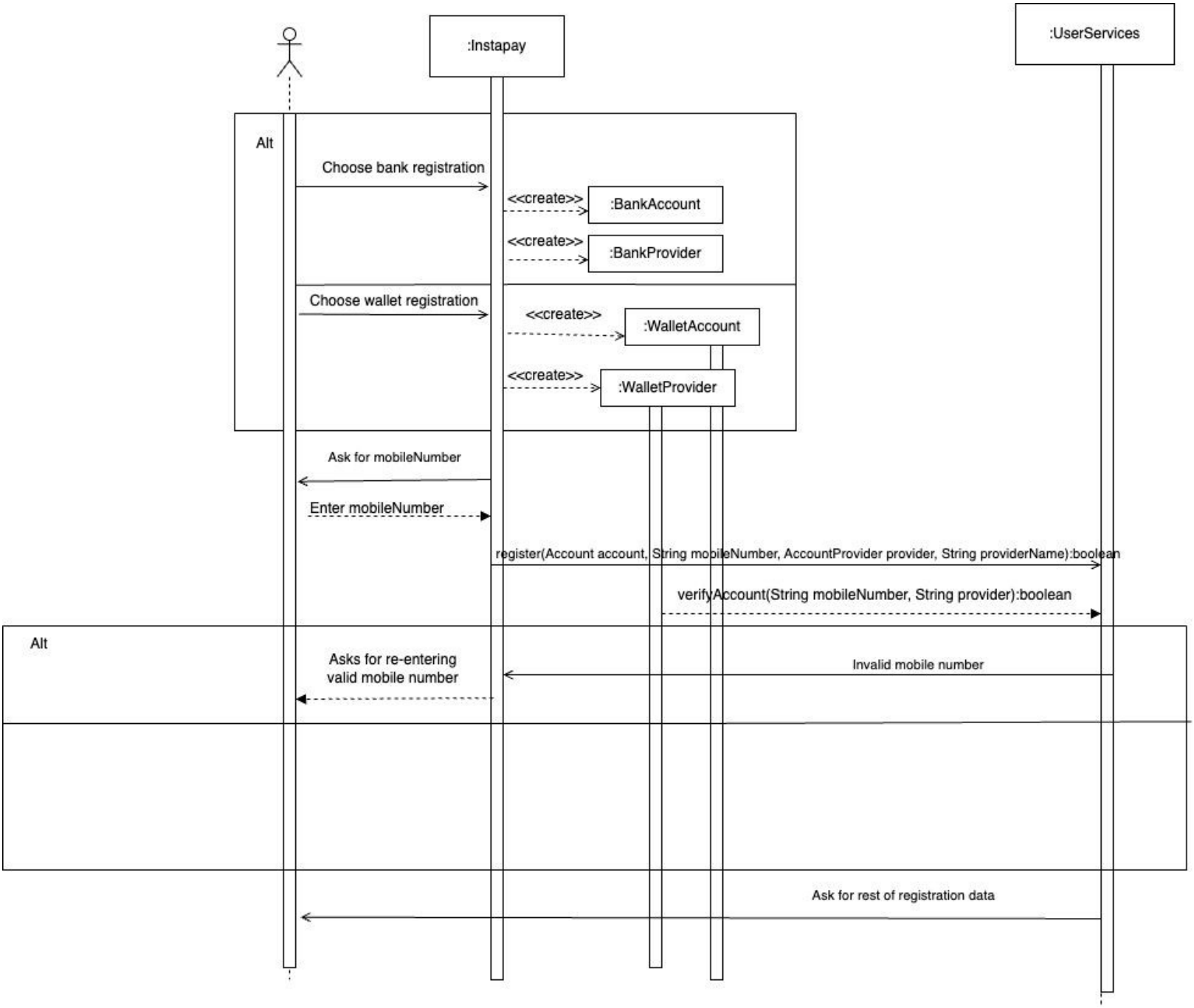
SDS document

3. Log-in:



SDS document

4. Registration



SDS document

GitHub repository link

https://github.com/RanaEssam03/Online-Payment-System