COMP 354
Design Document for myMoney

Team PA-PK

March 14, 2018

Table 1: Team

Name	ID Number
Anne-Laure Ehresmann	27858906
Marc-Antoine Dube	40029307
Kadeem Caines	26343600
Abdel Rahman Jawhar	27192142
Keith Dion	40036340
Hrachya Hakobyan	40041555
Andrew-Smith	40034936
Dongyu Chen	27241909
Yauheni Karaniuk	40005680
Renny Xu	40005262
Wei Wang	40041116

Contents

List of Figures

List of Tables

1 Introduction and Purpose

The goal of this document is to define the design for the desktop application myMoney. The majority of the design decisions have been taken with the Requirements document in mind, one may thus want to look at this document first to have a clear picture of the problem in mind as well as the requirements demanded for the solution. This document presents an implementation of a possible solution to answer this problem. Its design is is outlined through an Architectural Design (AD), a Detailed design (DD) and Dynamic Design Scenarios (DDS) for the application. The AD focuses on high-level project decomposition, the DD describes the overarching system design (which includes the UML design, divided into multiple subsections), and the DDS displays how the subsystems interact with one another in order to produce system-level services. This document may thus be used to plan, coordinate, and guide the development of the software, estimate and allocate necessary resources for proper execution, and then actually implement the software for the system. It seeks, above all, to serve as a precise and stable reference throughout the development.

2 Scope

This document contains everything to do with the development decisions and design of the system, all of which are derived from the requirements, which are not described in this document. Also not included in here is any testing of the system, which verifies that the requirements are met. It is merely a blueprint for a system that should, in theory, successfully pass any tests that would be done in correspondence with the requirements.

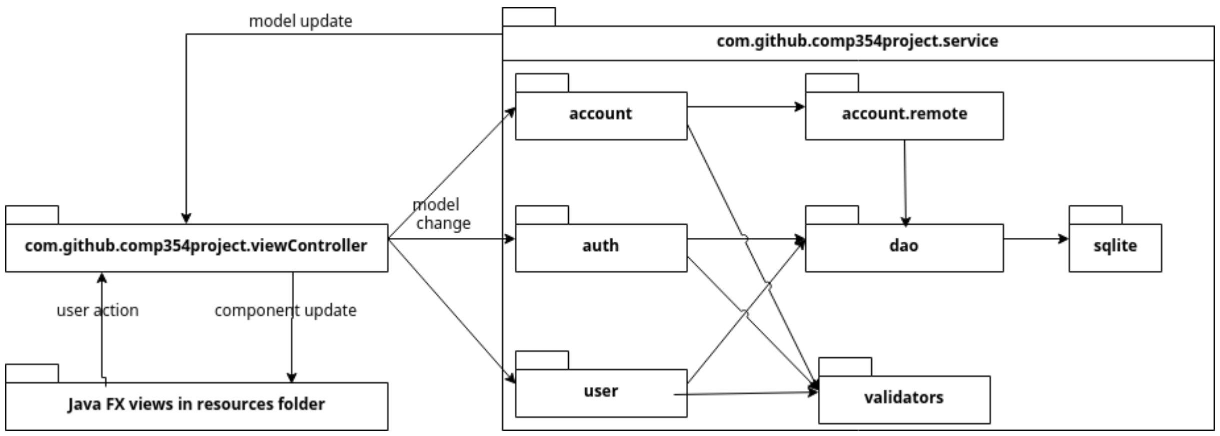
3 Architectural Design

The myMoney application uses the Model-View-Controller (MVC) pattern to read, validate and modify the objects. The view is implemented through a JAVAFX front-end interface. This view then reports any events triggered by the user to the view controllers, which access the back-end and SQLite database to modify the model. Once updated, the model then passes its changes back to the view, to update the information displayed to the user. A more in-depth view of the interaction between the user and the view can be seen in the dynamic models, in section 5.

The controllers are implemented through a series of function calls to services which handle different layers of the application. All services perform their own validation, whether it is for implementing the business rules, or simply ensuring expected application behaviour (no null objects, caught exceptions...). Once this validation has been verified, the controllers passes the actually requested model change to the model. Examples of such services, their intercommunication, and their validation, is explained more in-depth in section 3.2.

The model, which includes the back-end connection to the database, is the more complex part of the system. Lower layer services use the data access objects (DAO) to apply edits to the database, after the upper layers have verified the validity of the calls. Our system actually employs two databases; The first, a local database holding user info, bank account info, and transaction info, and the second, a ”remote” database (also local, but acts as if it were remote) used to simulate the bank institutions’ servers. When the user first adds a bank account, our view receives his input, passes it to the controls, which then makes requests for information to the ”remote” database through the use of the services handling remote communication. These services receive a serialisation of the ”remote” account, which it then translates into usable data for the local database. Once this has been successfully executed, the model triggers a view update, wherein the user can see his newly requested additions. All other events triggered by the user have no need of the remote database, and simply employ a series of communications between the controllers and the services handling local database. See section 3.2 for more details on these services and the databases.

3.1 Architectural Diagram


model update
com.github.comp354project.service
account
account.remote
model change
com.github.comp354project.viewController
auth
dao
sqlite
user action
component update
user
validators
Java FX views in resources folder

This design represents the MVC pattern discussed previously. The big advantage of this design is that everything is separated with interfaces which makes it easier to use different implementations, modify the features and create or mock tests.

The com.github.comp354project.service package is the main subsystem where most of the logic happens. The data validation and processing is done there. It connects to an SQLite database to persist the data. The com.github.comp354project.viewController calls this package to update the view and the model.

The com.github.comp354project.service package.account.remote package is a subsystem to our services which is meant to mock an API call to systems outside of ours like banks or credit card companies. Because we don’t have access to these APIs for real, obviously, the data is persisted in the same SQLite database as the rest of the system.

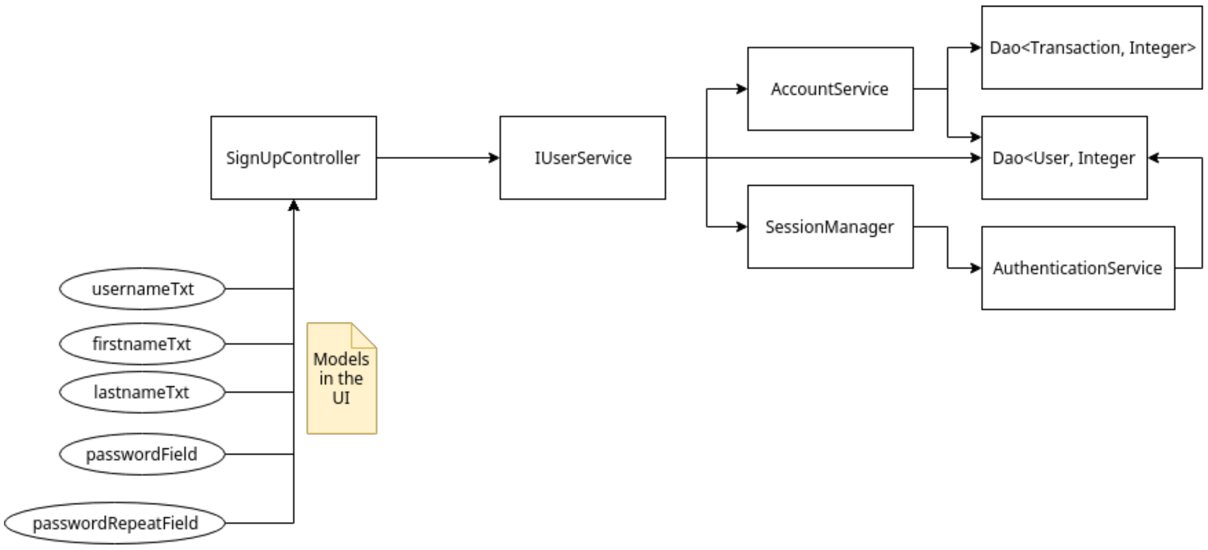
3.2 Subsystem Interface Specifications

Specification of the software interfaces between the subsystems, i.e. specific messages (or function calls) that are exchanged by the subsystems. These are also often called "Module Interface Specifications". Description of the parameters to be passed into these function calls in order to have a service fulfilled, including valid and invalid ranges of values. Each subsystem interface must be presented in a separate subsection.

*Note: The above is a description of what to provide. Need to edit into our own

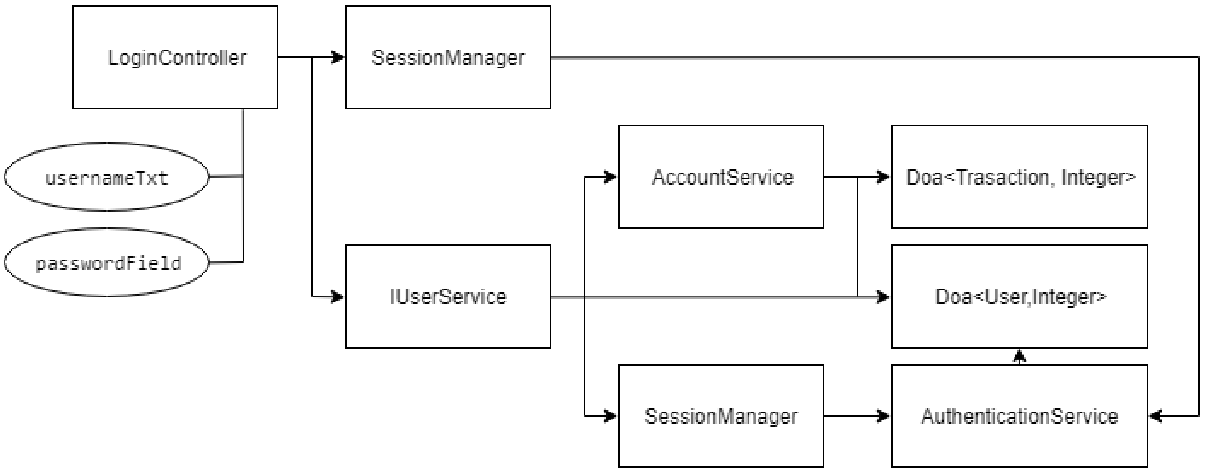
SignUpController Interfaces

Below are the different models and services used in the SignUpController view.



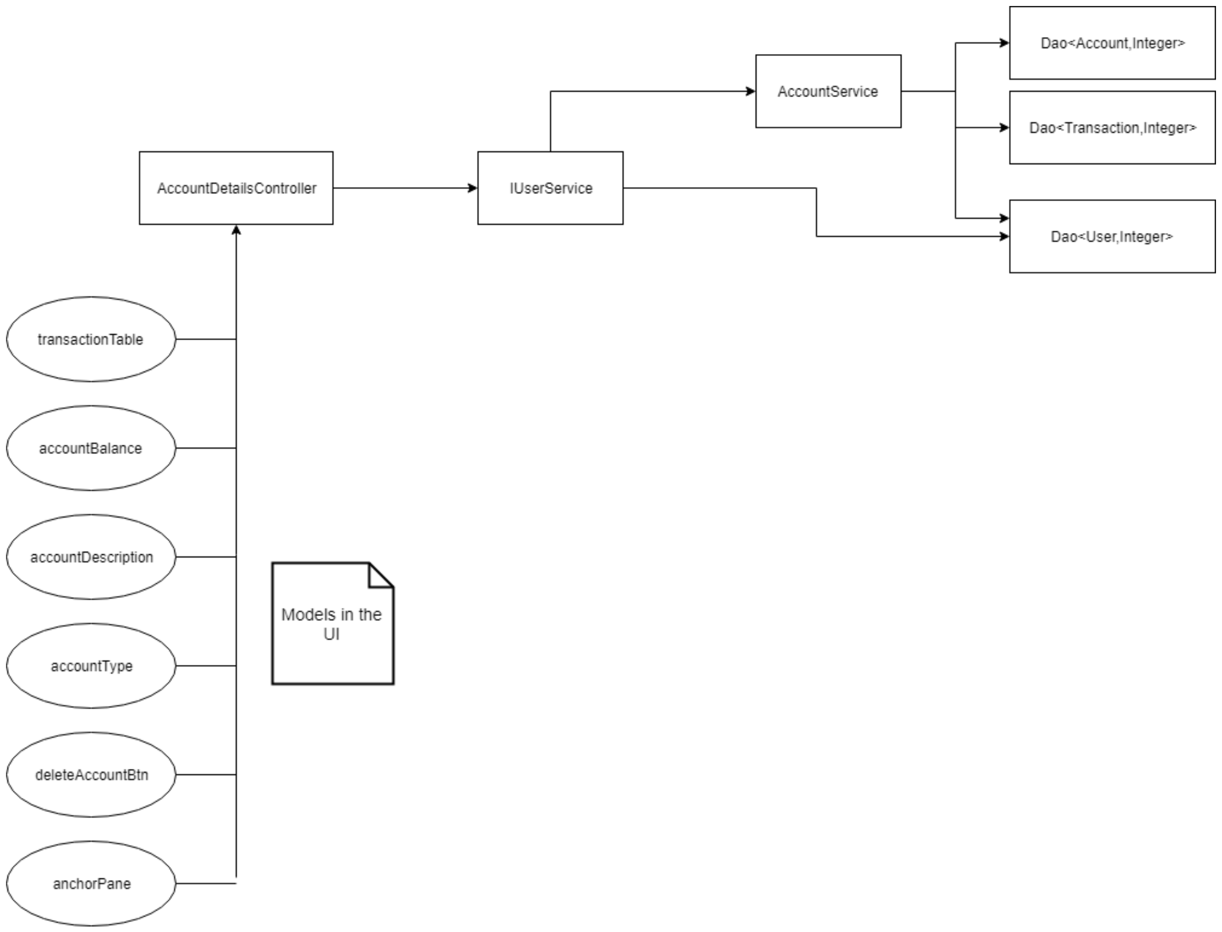
LoginController Interfaces

Below are the different models and services used in the LoginController view.



AcountDetailsController Interfaces

Below are the different models and services used in the AccountDetailsController view.



4 Detailed Design

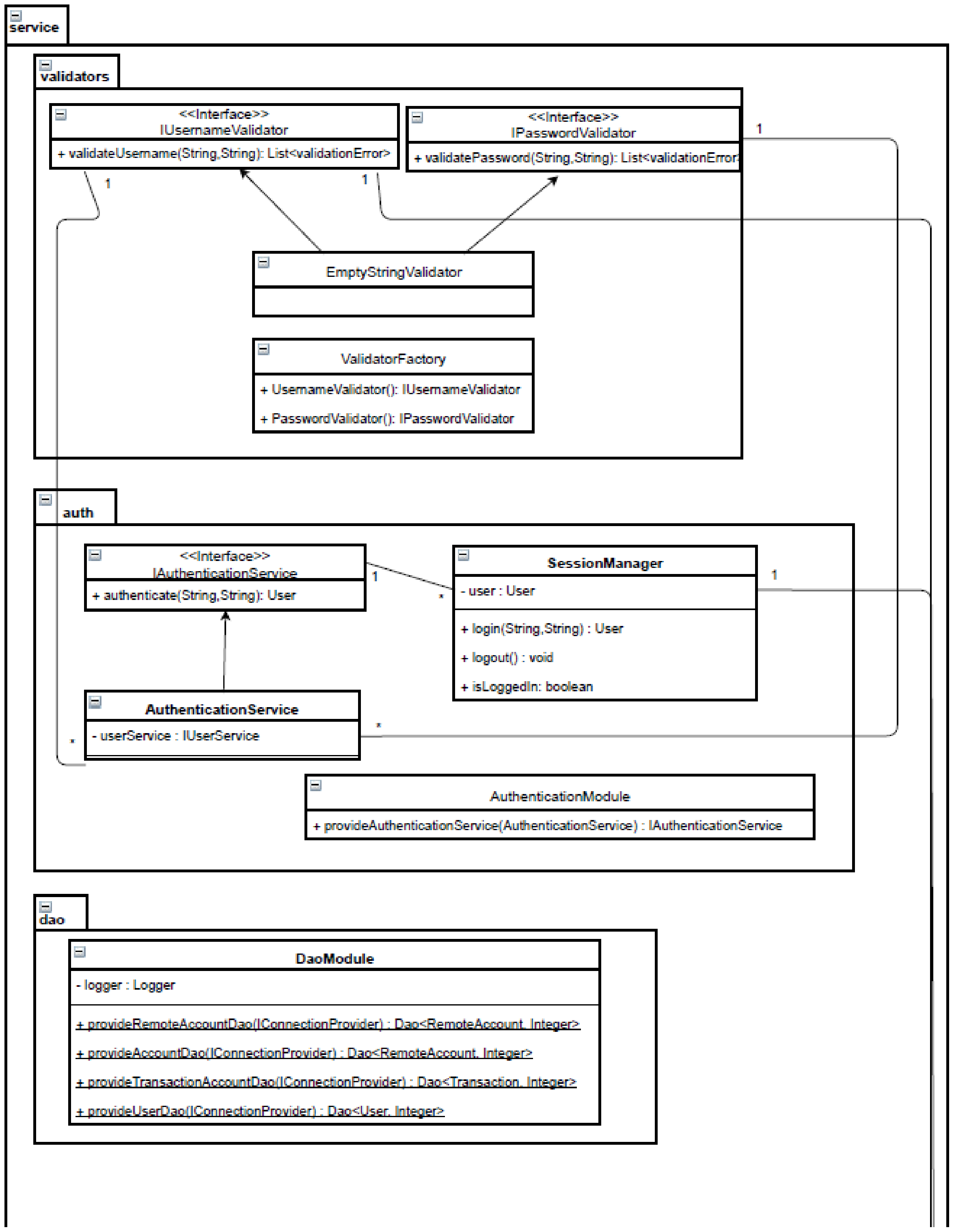
The myMoney system architecture is designed to be easily modified because of the low coupling between the modules. This was done with interfaces and auto injection of dependencies in classes. Each service package has a Module class designed to bind and provide an implementation to an interface. This way, classes are never instantiated directly into each other, but injected. This design pattern is useful because a change in implementation is as simple as creating a new class and change the module binding. The classes that use it and the tests should in no way be changed. Mocking classes for test purposes is also much easier.

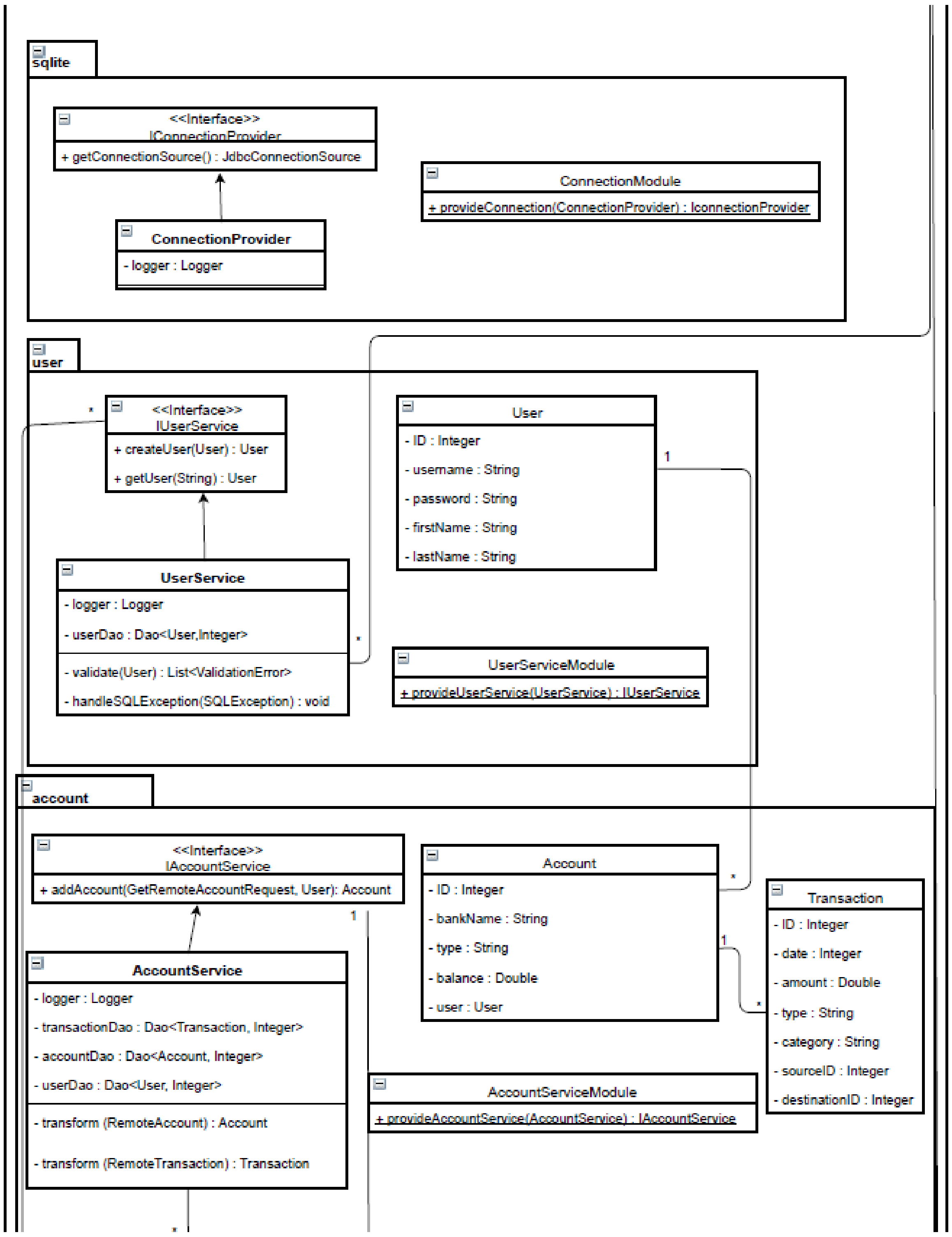
As a side note, we noticed that merge conflicts using git were much less likely to happen because we can each work on different parts of the system without modifying another module.

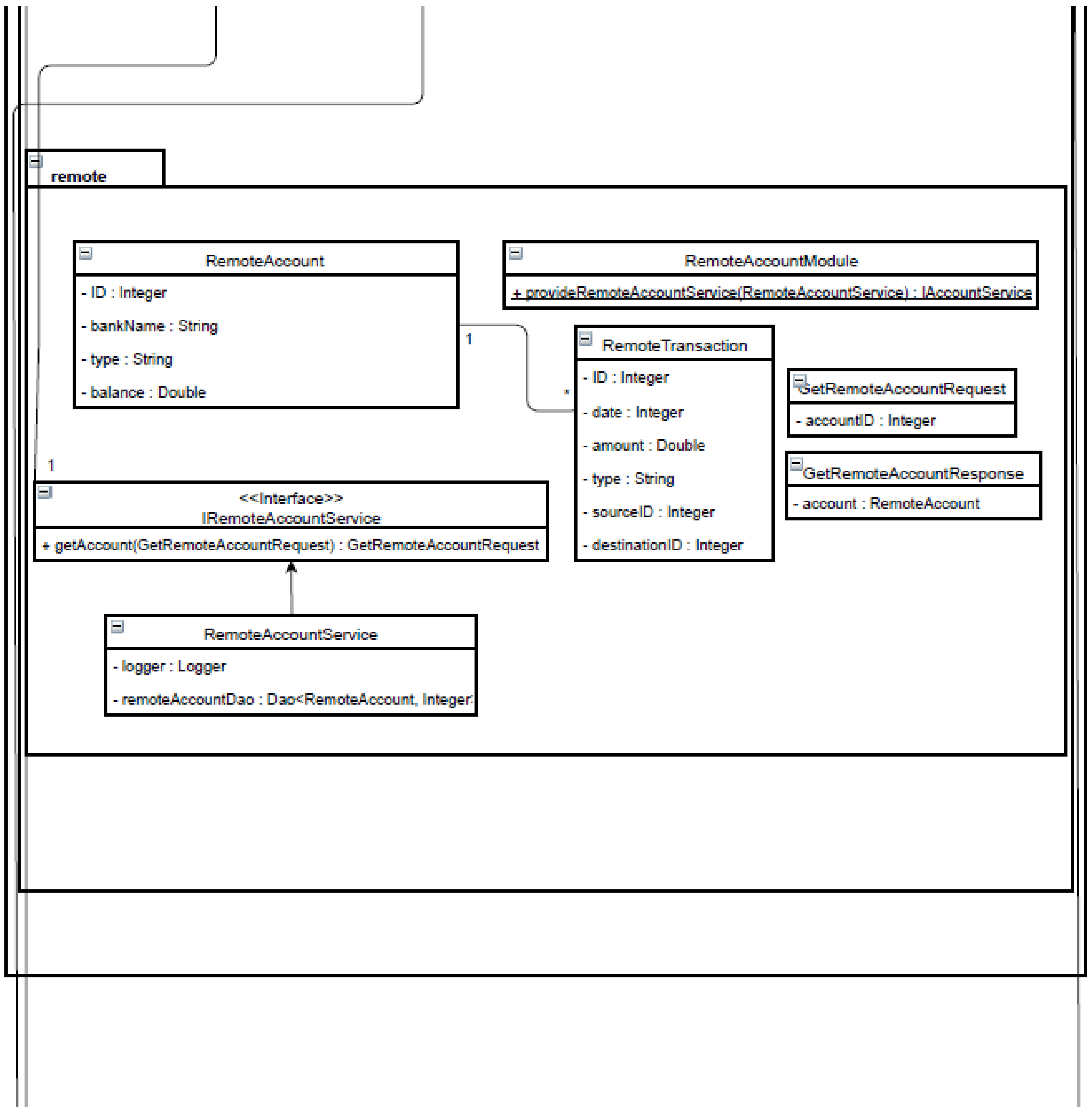
The tool used for this purpose is Dagger version 2.

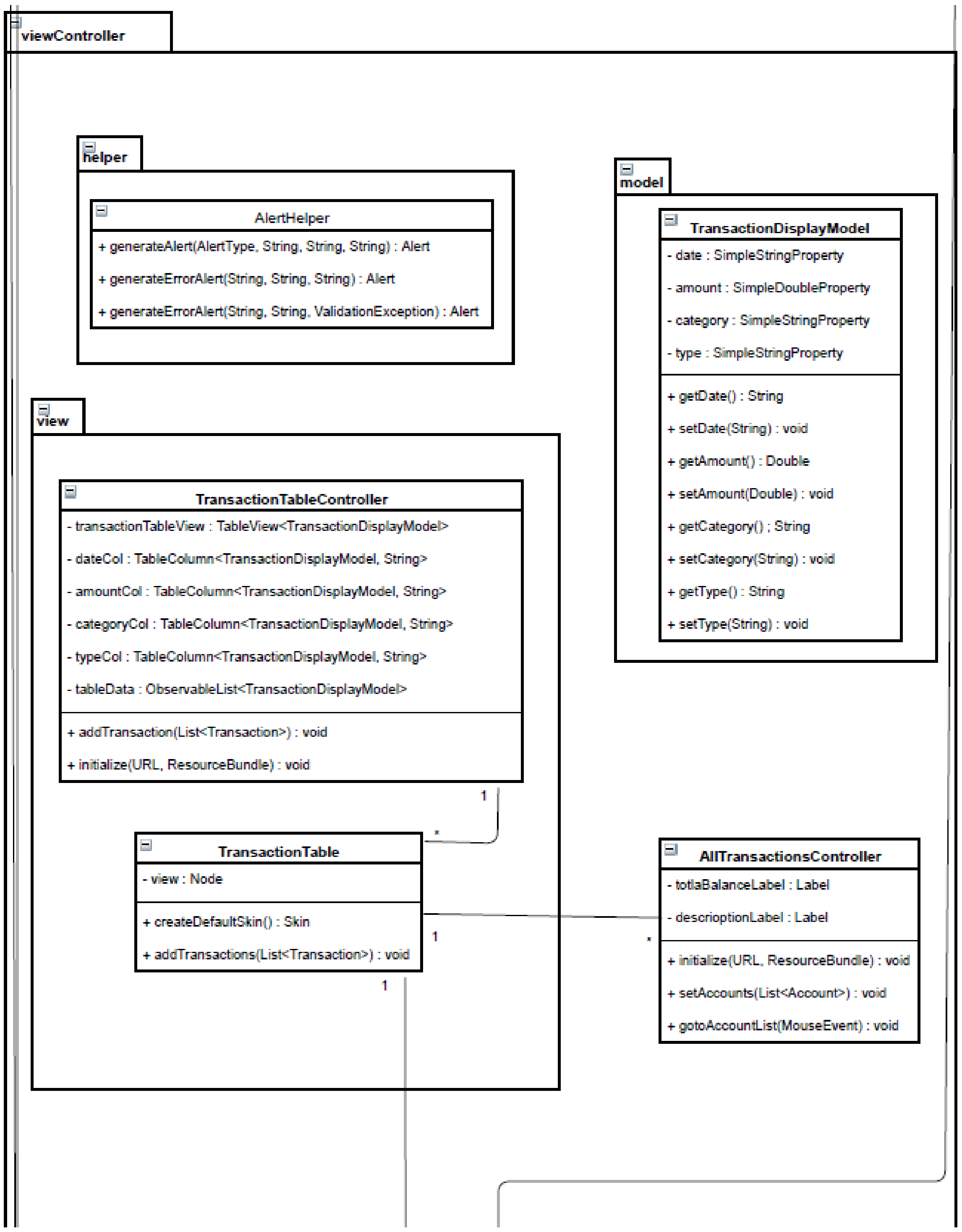
4.1 Class Diagram

In this section we provide the class diagram of our system, useful for the system developers and testers.This is an in depth look at all of the classes within our system see figure 1 below If a term is unclear, view section 4.2 for the glossary.


service
validators
<<Interface>>
IUsernameValidator
+ validateUsername(String,String): List<validationError>
<<Interface>>
IPasswordValidator
+ validatePassword(String,String): List<validationError>
1
1
1
EmptyStringValidator
ValidatorFactory
+ UsernameValidator(): IUsernameValidator
+ PasswordValidator(): IPasswordValidator
auth
<<Interface>>
IAuthenticationService
+ authenticate(String,String): User
1
*
SessionManager
- user : User
+ login(String,String) : User
+ logout() : void
+ isLoggedIn: boolean
1
AuthenticationService
- userService : IUserService
*
*
AuthenticationModule
+ provideAuthenticationService(AuthenticationService) : IAuthenticationService
dao
DaoModule
- logger : Logger
+ provideRemoteAccountDao(IConnectionProvider) : Dao<RemoteAccount, Integer>
+ provideAccountDao(IConnectionProvider) : Dao<RemoteAccount, Integer>
+ provideTransactionAccountDao(IConnectionProvider) : Dao<Transaction, Integer>
+ provideUserDao(IConnectionProvider) : Dao<User, Integer>


sqlite
<<Interface>>
IConnectionProvider
+ getConnectionSource() : JdbcConnectionSource
ConnectionProvider
- logger : Logger
ConnectionModule
+ provideConnection(ConnectionProvider) : IconnectionProvider
user
<<Interface>>
IUserService
+ createUser(User) : User
+ getUser(String) : User
UserService
- logger : Logger
- userDao : Dao<User,Integer>
- validate(User) : List<ValidationError>
- handleSQLException(SQLException) : void
User
- ID : Integer
- username : String
- password : String
- firstName : String
- lastName : String
UserServiceModule
+ provideUserService(UserService) : IUserService
account
<<Interface>>
IAccountService
+ addAccount(GetRemoteAccountRequest, User): Account
AccountService
- logger : Logger
- transactionDao : Dao<Transaction, Integer>
- accountDao : Dao<Account, Integer>
- userDao : Dao<User, Integer>
- transform (RemoteAccount) : Account
- transform (RemoteTransaction) : Transaction
Account
- ID : Integer
- bankName : String
- type : String
- balance : Double
- user : User
AccountServiceModule
+ provideAccountService(AccountService) : IAccountService
Transaction
- ID : Integer
- date : Integer
- amount : Double
- type : String
- category : String
- sourceID : Integer
- destinationID : Integer
1
*
1
*
1
*
*
*


remote
RemoteAccount
- ID : Integer
- bankName : String
- type : String
- balance : Double
RemoteAccountModule
+ provideRemoteAccountService(RemoteAccountService) : IAccountService
RemoteTransaction
- ID : Integer
- date : Integer
- amount : Double
- type : String
- sourceID : Integer
- destinationID : Integer
GetRemoteAccountRequest
- accountID : Integer
GetRemoteAccountResponse
- account : RemoteAccount
<<Interface>>
IRemoteAccountService
+ getAccount(GetRemoteAccountRequest) : GetRemoteAccountRequest
RemoteAccountService
- logger : Logger
- remoteAccountDao : Dao<RemoteAccount, Integer>
1
*
1


viewController
helper
AlertHelper
+ generateAlert(AlertType, String, String, String) : Alert
+ generateErrorAlert(String, String, String) : Alert
+ generateErrorAlert(String, String, ValidationException) : Alert
model
TransactionDisplayModel
- date : SimpleStringProperty
- amount : SimpleDoubleProperty
- category : SimpleStringProperty
- type : SimpleStringProperty
+ getDate() : String
+ setDate(String) : void
+ getAmount() : Double
+ setAmount(Double) : void
+ getCategory() : String
+ setCategory(String) : void
+ getType() : String
+ setType(String) : void
view
TransactionTableController
- transactionTableView : TableView<TransactionDisplayModel>
- dateCol : TableColumn<TransactionDisplayModel, String>
- amountCol : TableColumn<TransactionDisplayModel, String>
- categoryCol : TableColumn<TransactionDisplayModel, String>
- typeCol : TableColumn<TransactionDisplayModel, String>
- tableData : ObservableList<TransactionDisplayModel>
+ addTransaction(List<Transaction>) : void
+ initialize(URL, ResourceBundle) : void
1
*
TransactionTable
- view : Node
+ createDefaultSkin() : Skin
+ addTransactions(List<Transaction>) : void
1
1
*
AllTransactionsController
- totlaBalanceLabel : Label
- descrioptionLabel : Label
+ initialize(URL, ResourceBundle) : void
+ setAccounts(List<Account>) : void
+ gotoAccountList(MouseEvent) : void

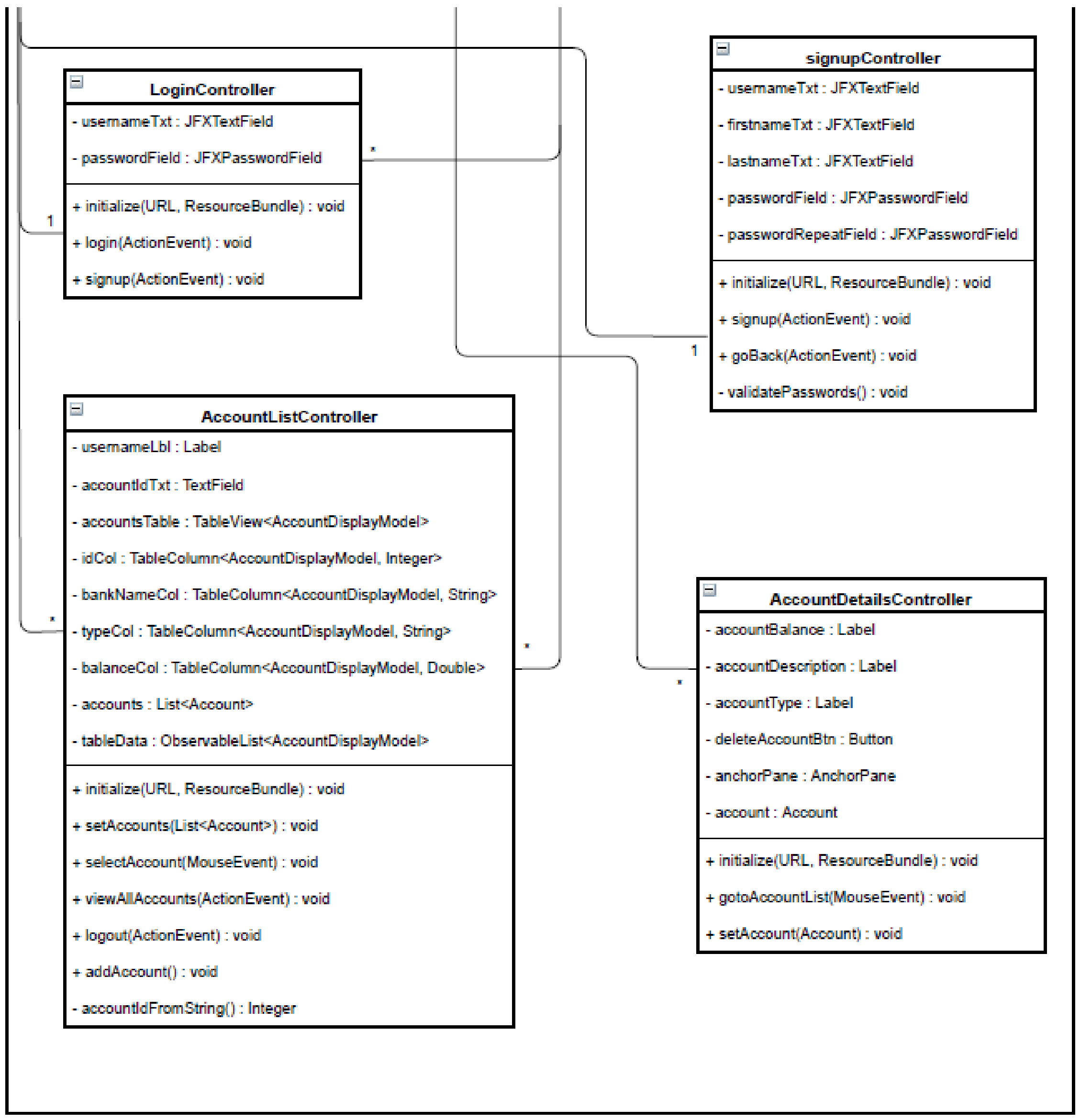

LoginController
- usernameTxt : JFXTextField
- passwordField : JFXPasswordField
+ initialize(URL, ResourceBundle) : void
+ login(ActionEvent) : void
+ signup(ActionEvent) : void
signupController
- usernameTxt : JFXTextField
- firstnameTxt : JFXTextField
- lastnameTxt : JFXTextField
- passwordField : JFXPasswordField
- passwordRepeatField : JFXPasswordField
+ initialize(URL, ResourceBundle) : void
+ signup(ActionEvent) : void
+ goBack(ActionEvent) : void
- validatePasswords() : void
AccountListController
- usernameLbl : Label
- accountIdTxt : TextField
- accountsTable : TableView<AccountDisplayModel>
- idCol : TableColumn<AccountDisplayModel, Integer>
- bankNameCol : TableColumn<AccountDisplayModel, String>
- typeCol : TableColumn<AccountDisplayModel, String>
- balanceCol : TableColumn<AccountDisplayModel, Double>
- accounts : List<Account>
- tableData : ObservableList<AccountDisplayModel>
+ initialize(URL, ResourceBundle) : void
+ setAccounts(List<Account>) : void
+ selectAccount(MouseEvent) : void
+ viewAllAccounts(ActionEvent) : void
+ logout(ActionEvent) : void
+ addAccount() : void
- accountIdFromString() : Integer
AccountDetailsController
- accountBalance : Label
- accountDescription : Label
- accountType : Label
- deleteAccountBtn : Button
- anchorPane : AnchorPane
- account : Account
+ initialize(URL, ResourceBundle) : void
+ gotoAccountList(MouseEvent) : void
+ setAccount(Account) : void
1
*
1
*
*
*

Figure 1: Class Diagram

4.2 Glossary of Domain Concepts

Table 2: Glossary of Domain Concepts

Expression	Definition
User	The person that is using the application and the main provider of requests to the system.
User Account	A data object containing user information. It also contains the various bank accounts that a user may have linked to the system.
Bank Account	A data object containing transactions linked with a specific bank account in a bank institution. One user account may have more than one bank accounts.
Transaction	Any kind of money exchange associated with a bank account.
Transfer	A type of transaction that occurs between two parties.
Deposit	A type of transaction where the owner puts money in his own bank account.
Withdrawal	A type of transaction where the owner of the bank account removes money from his balance.
Database	A local or online container which holds data in an organised, efficient manner.
Server	a computer that is accessible on a network, on which a database and/or system may be hosted. The bank institutions' databases will be hosted on here.
Object-Oriented Programming	A programming paradigm which separates entities into objects, and uses the concept of inheritance of properties, polymorphism of objects, encapsulation of objects. We use this paradigm for its maintainability and structural benefits.
MVC - Model-View-Controller Architecture	An architectural pattern which strictly separates components into the model (manages the data and logic), the view (output of the model), and the controller (handling input and passing it to the model or view).
Interface	A component of a system by which other entities (be it humans or other systems) may engage in an exchange of data with the system in question.
API - Application Programming Interface	A protocol or set of functions which serve as a method of communication to a software system. It is a type of interface, and the one by which our system will communicate with the banking institutions' databases.
DAO - Data access object	An object that provides an abstract interface to some type of database or other persistence mechanism.

4.3 Subsystem X

Detailed Design Diagram

UML class diagram depicting the internal structure of the subsystem, accompanied by a paragraph of text describing the rationale of this design.

*Note: The above is a description of what to provide. Need to edit into our own

Units Description

List each class in this subsystem and write a short description of its purpose, as well as notes or reminders useful for the programmers who will implement them. List all attributes and functions of the class.

*Note: The above is a description of what to provide. Need to edit into our own

5 Dynamic Design Scenarios

Describe some (at least two) important execution scenarios of the system using UML sequence diagrams. These scenarios must demonstrate how the various subsystems and units are interacting to achieve a system-level service. Units and subsystems depicted here must be compatible with the descriptions provided in section 3 and 4.

*Note: The above is a description of what to provide. Need to edit into our own

5.1 Dynamic Models of System Interface

We have chosen 3 major functionalities of the system (also known as use cases) in order to portray the interactions between the classes of the system. By using a sequence diagram, this will display the dynamics visually by showcasing the sequences of method calls when a particular use case begins functioning.

Use Case 1: Create User Account

The following scenario depicts the actions that occur when a user clicks the sign up button in order to create their account. Firstly , the SignUpController (which represents the controller part of the MVC architecture) is called to handle the sign up event. The SignUpController then calls the validatePassword() method in order to check if the password chosen fits the required format. Next, the signupcontroller sends a message to the User class in order to start its builder method. The builder method is filled out with the info needed to sign up. The user class then sends a message to the IUserService class in order to finally create a user within the database using the information entered. Finally, the IUserService class calls the MyMoneyApplication's displayLogin() method in order to show the user the success of creating an account (The View)

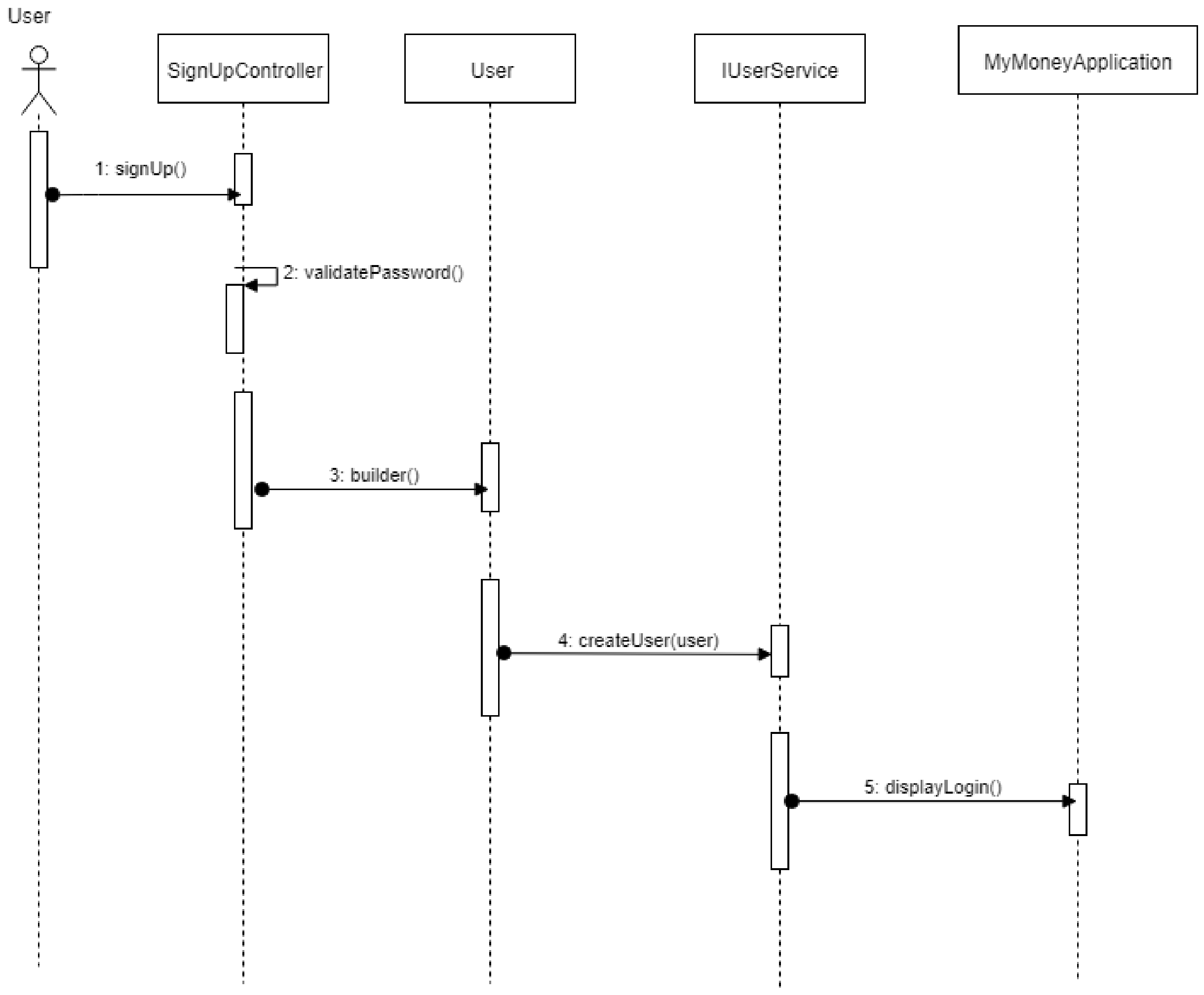


Figure 2: Use case 1 Sequence Diagram

Use Case 3: Add Bank Account to a User Account

The following scenario describes the actions that occur when a user clicks the add button in the account list view.

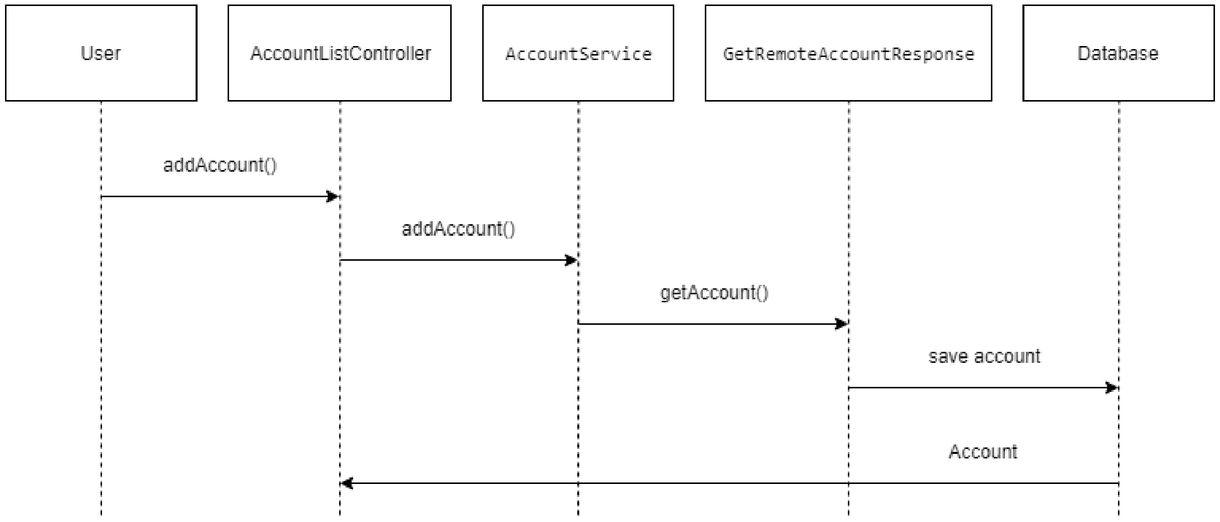


Figure 3: Use case 3 Sequence Diagram

Use Case 5: View Transactions for Specific Bank Account

The following scenario describes the actions occur when user click the button¡view transactions¿ for a specific bank account.

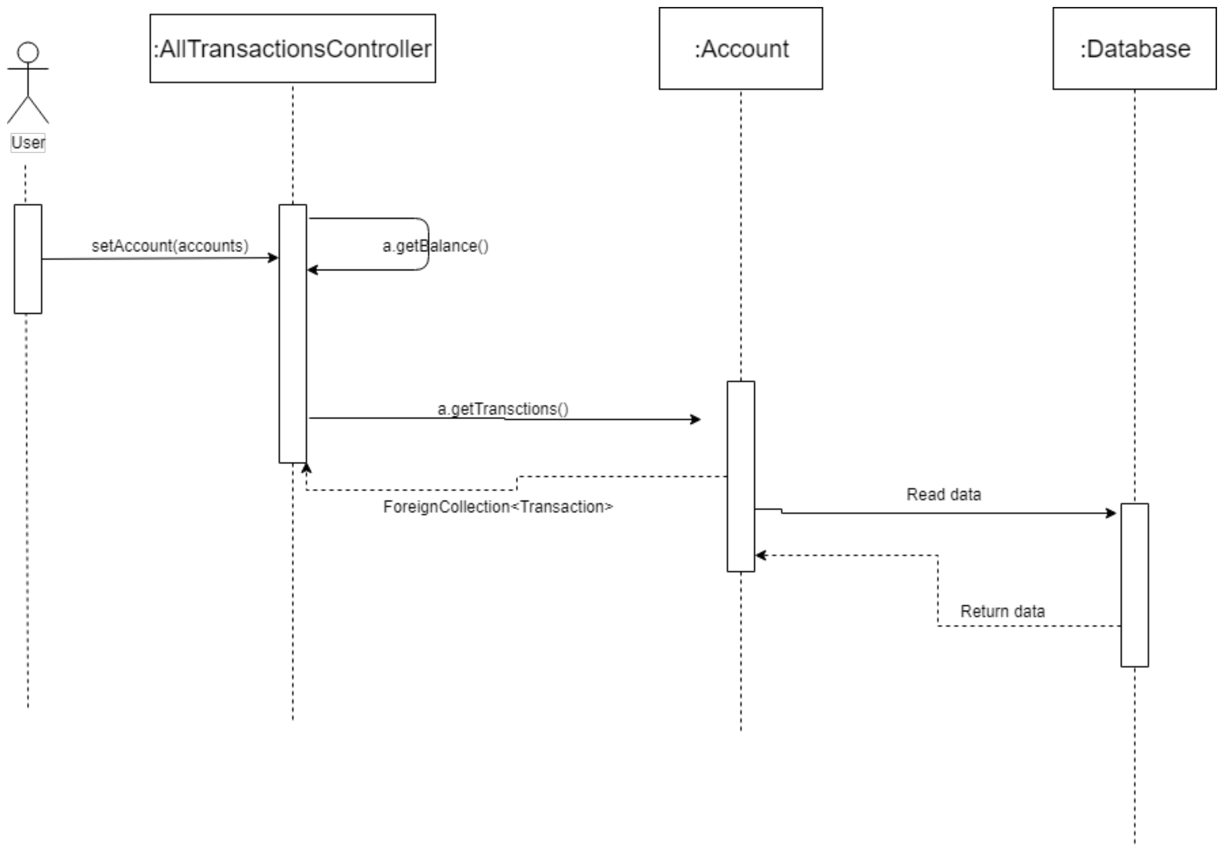


Figure 4: Use case 5 Sequence Diagram

Use Case 6: View All Transactions from all Bank Accounts

The follwoing scenario describes the actions occur when user click the button¡view all transactions¿ for viewsing all transactions from all bank accounts.

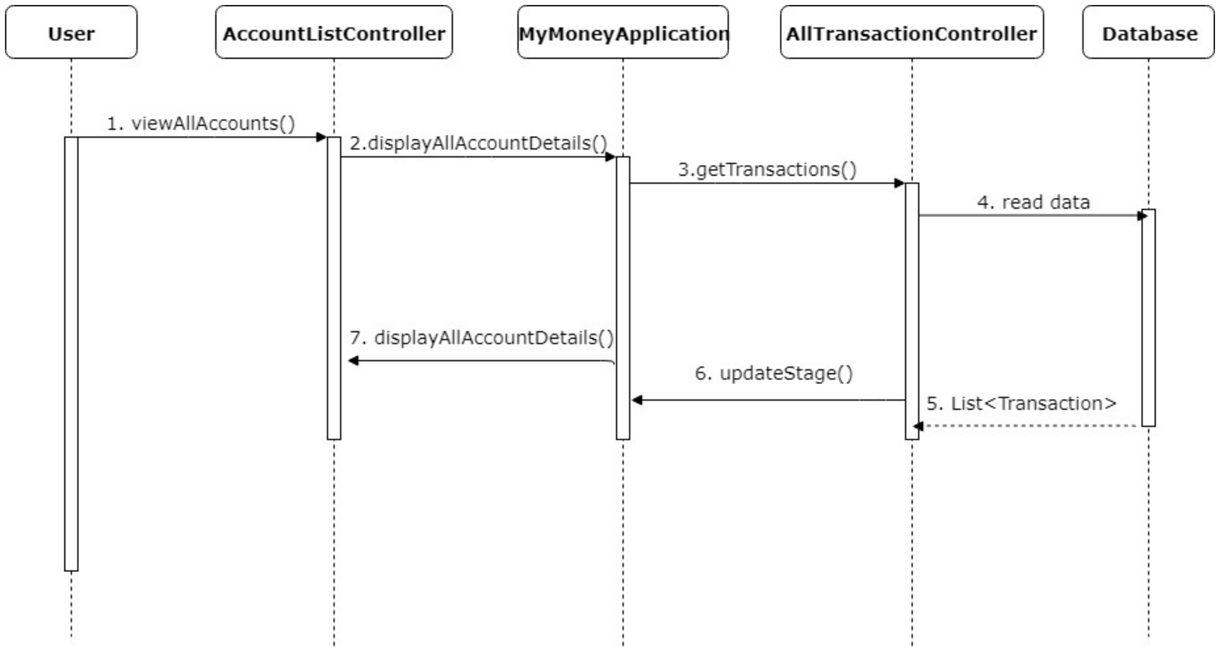


Figure 5: Use case 6 Sequence Diagram

6 Reference

- User information: As our user and use-cases was based on feedback provided by our developers, our references lie mainly within our own team.
- Craig Larman - Applying UML and Patterns
- Greg Butler's course COMP 354 content
- MIT Curricular Information System Software Requirements Document
- Carnegie Mellon Business Goals
- Use-Case: Oracle
- Google Dagger Github