COMP 354
Design Document for myMoney

Team PA-PK

March 17, 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 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) architectural pattern at its core.

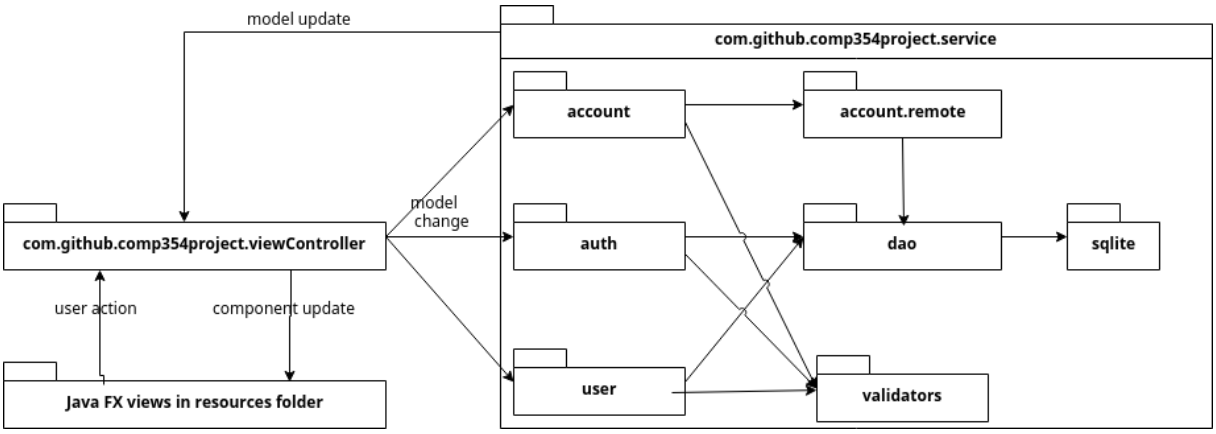
The *view* is implemented through a JAVAFX front-end interface, which consists mostly of hard-coded tables structures or menus, which are then populated with table entries by the model, as requested by the view controllers. The view is interactive, and reports any events triggered by the user to the view controllers, which then pass the requests to the model controllers (services), which handle the modification of the model. Once updated, the model then passes its changes back to the view controllers, 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 the most complex part of the system. They are organised in a layered structure of services, which each handle a different sector of the application (session management, account services, user services, etc). 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 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 composed of data access objects (DAOs), which are used to apply edits to the database, after the upper layers of the controllers 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 servers. The only time the second database is actually accessed is when the user first adds a bank account. In this case, 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, from the DAOs, 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



We herewith present the architectural diagram of the design presented above, and provide an explanation of each package (but not for the subsystems within each package). A succint and precise description of each subsystem is available in the next section.

Notice first that MVC's style forces a clear separation of concerns, and thus emphasises a great amount of intercommunication between each section. To embrace this, using interfaces for each component presented above will clearly separate the implementation

from the structure of the module, which eases parallel addition, modification, or testing of the system.

The com.github.comp354project.service package is the main subsystem of our project. As mentioned above, it is organised in a layered manner, wherein each layer handles its own services, and use the services of the layer below it within worrying about that layer's implementation. Data validation and processing is offered by each service: The account service, for example, validate calls to add or delete bank accounts, edit a transaction's category, or query for specific accounts. It does this by querying the database using an account DAO, and ensuring data integrity and validity (with regards to the business rules). It does not, however, worry about user authorisation, and simply assumes the layer above it (The user service) will have handled it. The com.github.comp354project.viewController calls services within this package to update the view and the model.

The com.github.comp354project.viewController package contains a number of controllers for each different view. They are the ones handling the requests from the users, which they pass to the services in the package described above. They are also the ones who pass any errors from the services to the views, mostly to be used for testing and alerting the user of any problems that might have occured.

The com.github.comp354project.service package.account.remote package is a subsystem to our services which mocks an API call to remote servers of banks or credit card companies. In our case however, we don't actually have access to such systems. For this reason, the remote data exists in an SQLite database like our local one.

3.2 Subsystem Interface Specifications

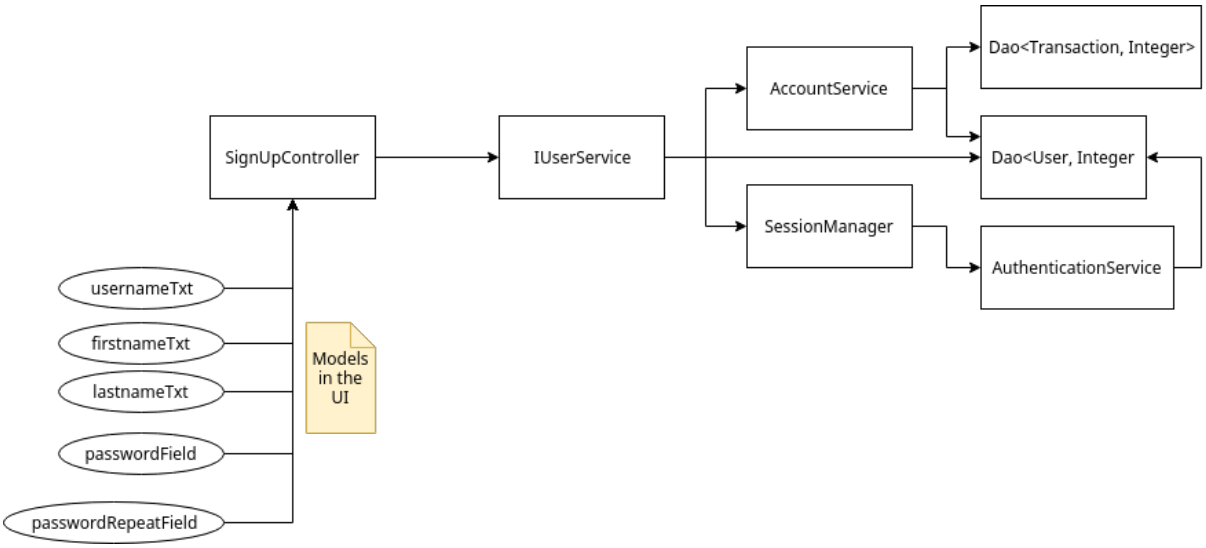
Specification of the software interfaces between the subsystems, i.e. specific messages (or function calls) that are exchanged by the subsystems. These are also often called "Module Interface Specifications". Description of the parameters to be passed into these function calls in order to have a service 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

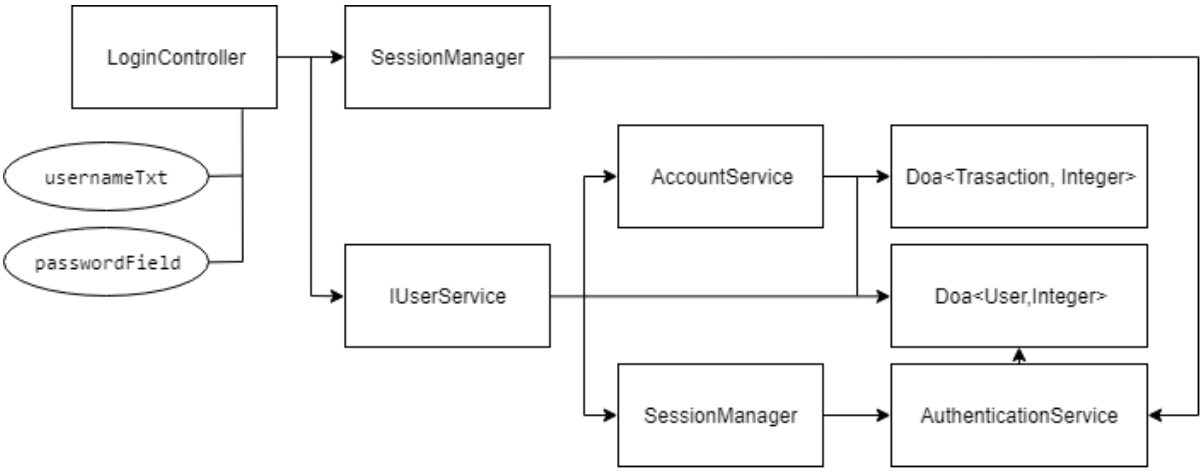
View Controllers

We herewith present the view controllers, and the services they access.

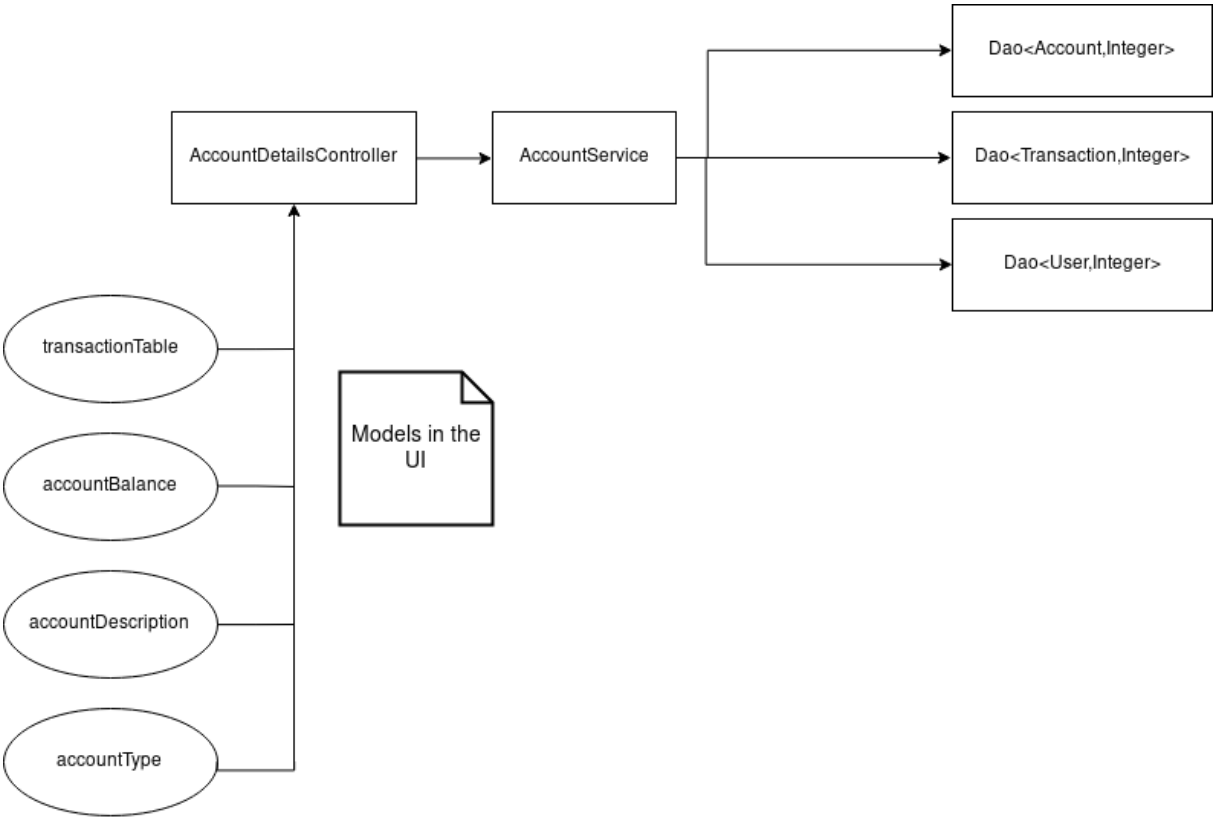
SignUpController Interfaces Below are the different models and services used in the SignUpController view.


Dao<Transaction, Integer>
AccountService
SignUpController
IUserService
Dao<User, Integer
SessionManager
AuthenticationService
usernameTxt
firstnameTxt
lastnameTxt
passwordField
passwordRepeatField
Models in the UI

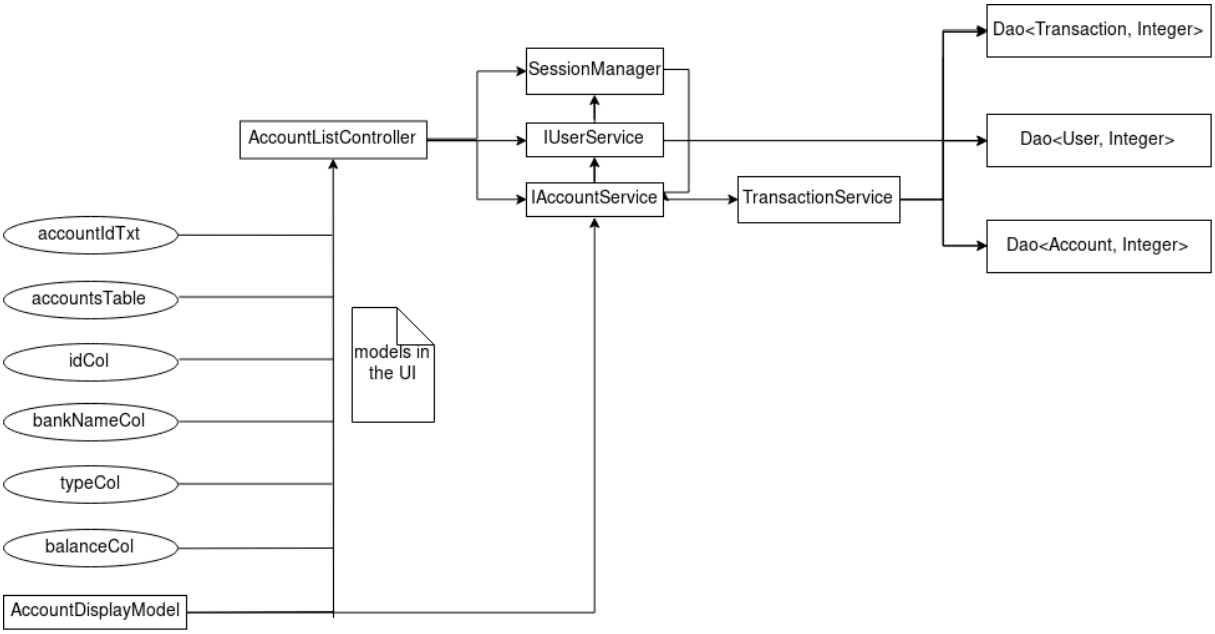
LoginController Interfaces Below are the different models and services used in the LoginController view.



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



AccountListController Interfaces Below are the different models and services used in the AccountListController view.



4 Detailed Design

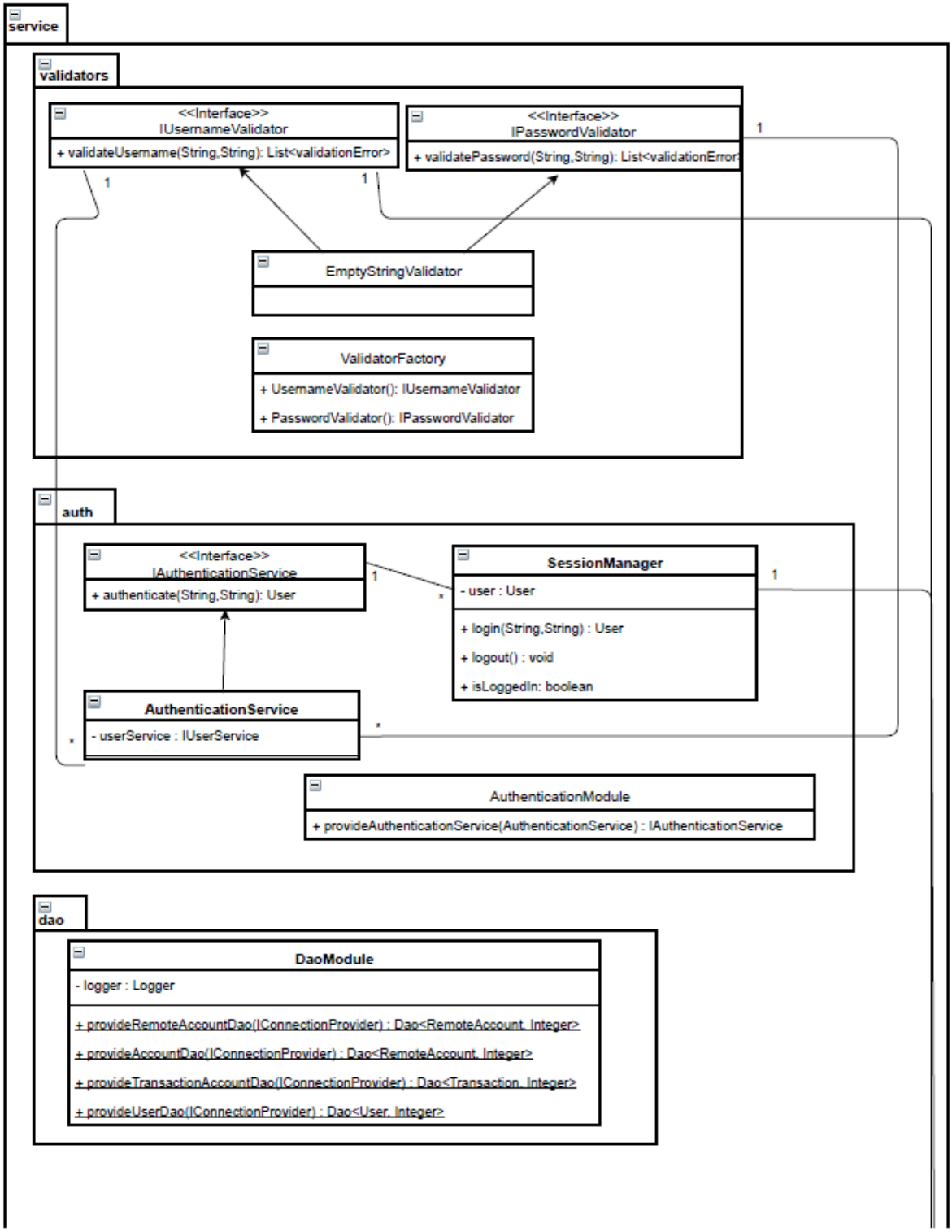
The myMoney system architecture is designed to be easily modified because of the low coupling between the modules. This was done with interfaces and auto injection of dependencies in classes. Each service package has a Module class designed to bind and provide an implementation to an interface. This 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.

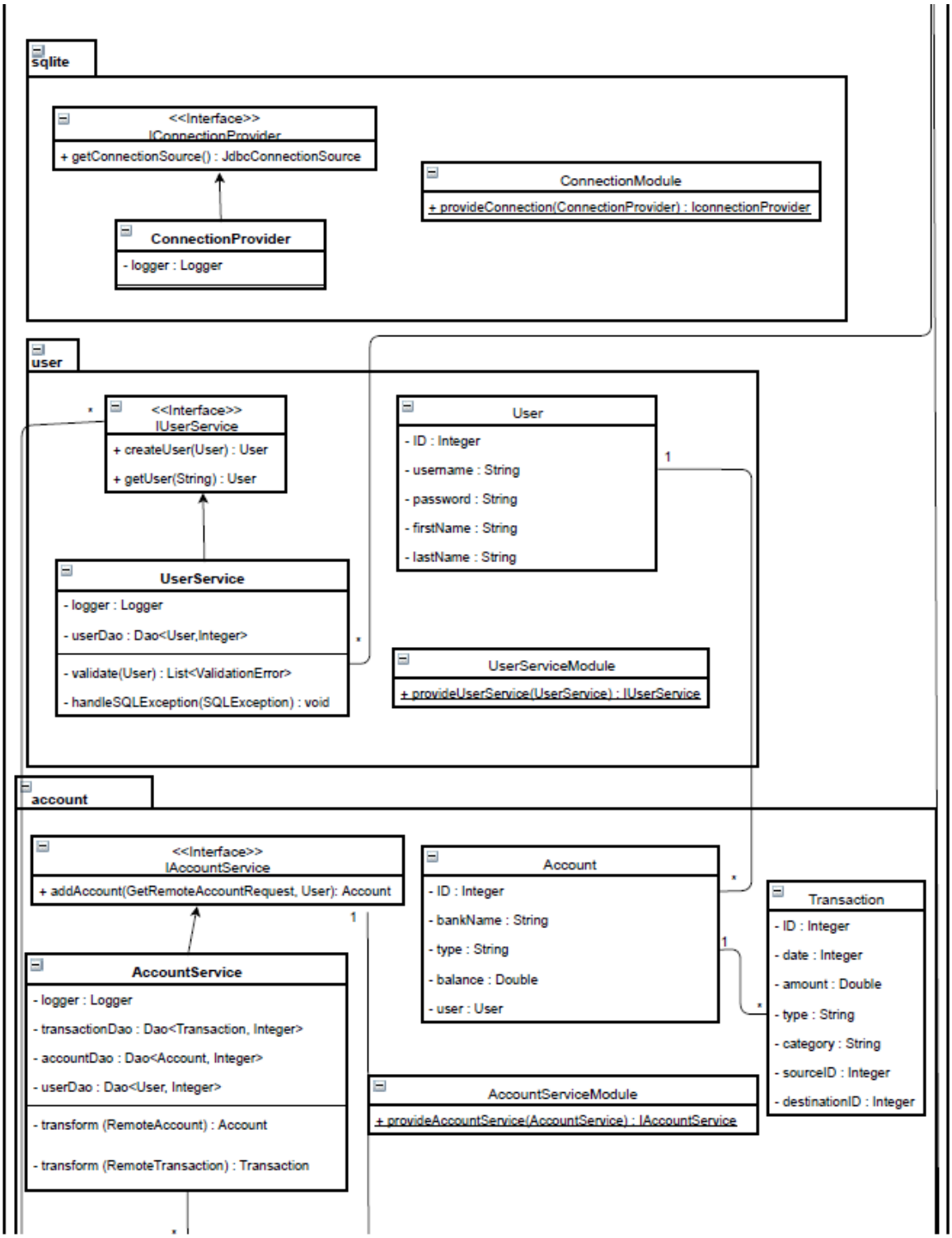
As a side note, we noticed that merge conflicts using git were much less likely to happen because we can each work on different parts of the system without modifying another module.

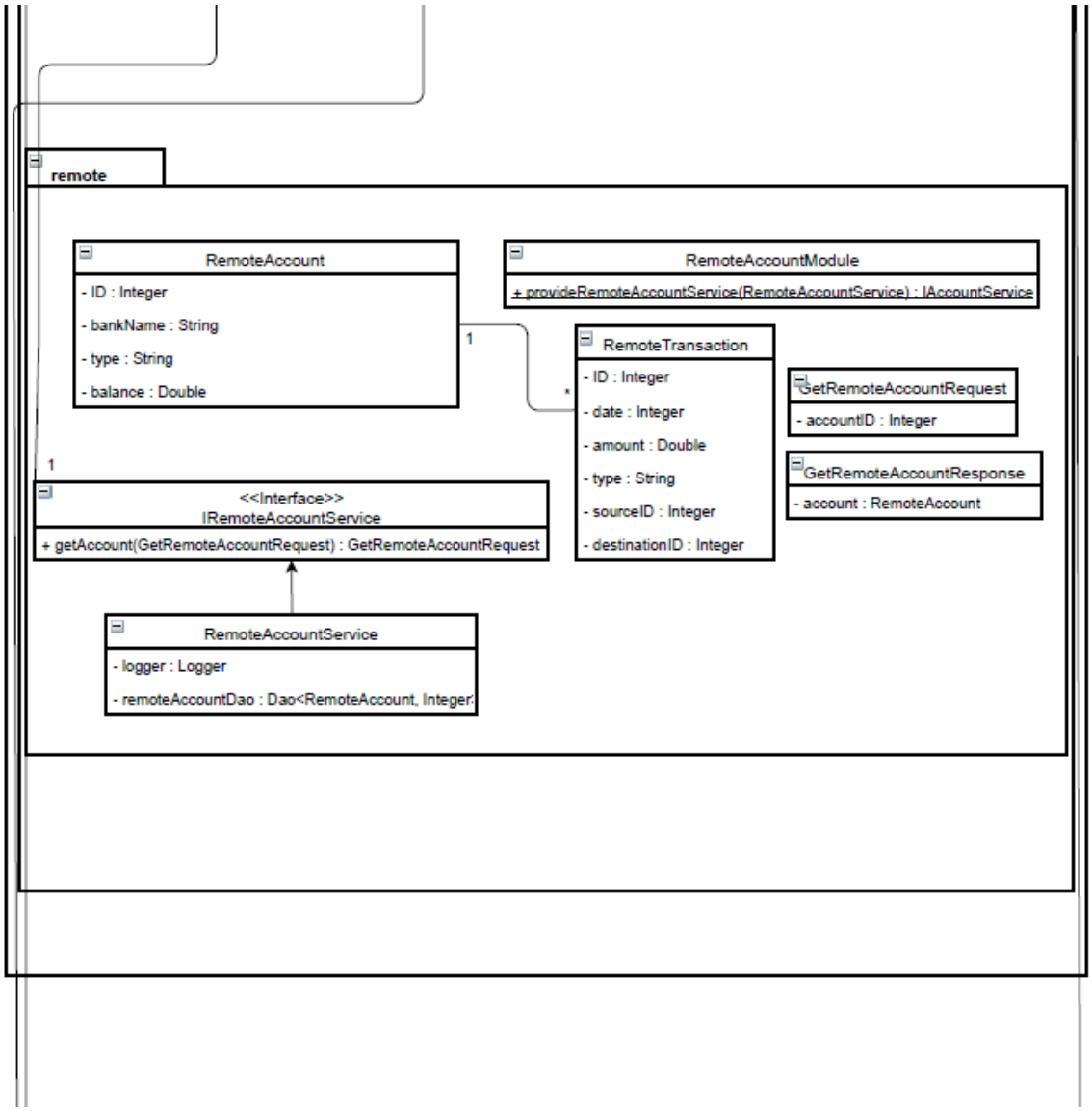
The tool used for this purpose is Dagger version 2.

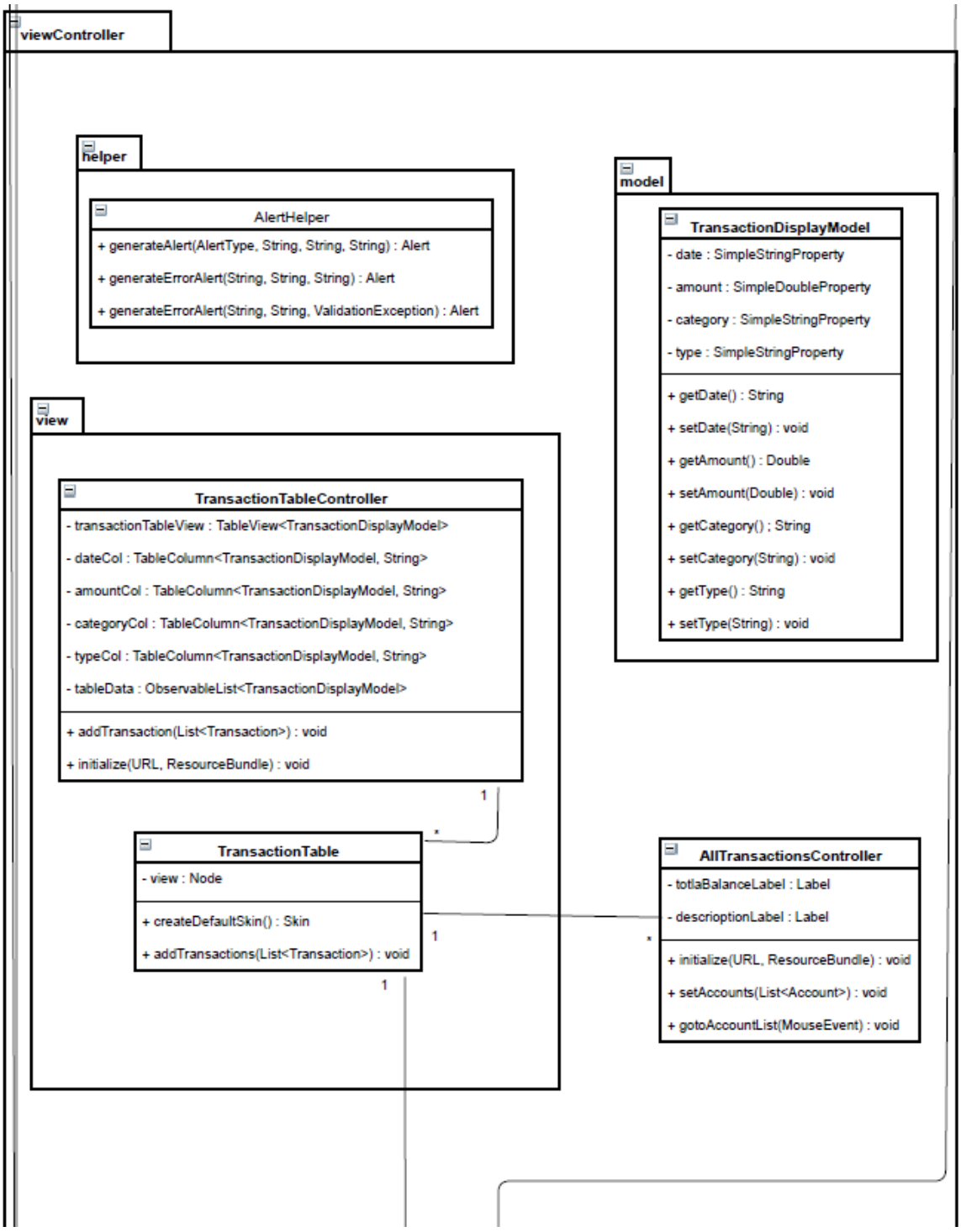
4.1 Class Diagram

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 4.3 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
Transaction
- ID : Integer
- date : Integer
- amount : Double
- type : String
- category : String
- sourceID : Integer
- destinationID : Integer
AccountServiceModule
+ provideAccountService(AccountService) : IAccountService


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


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
*

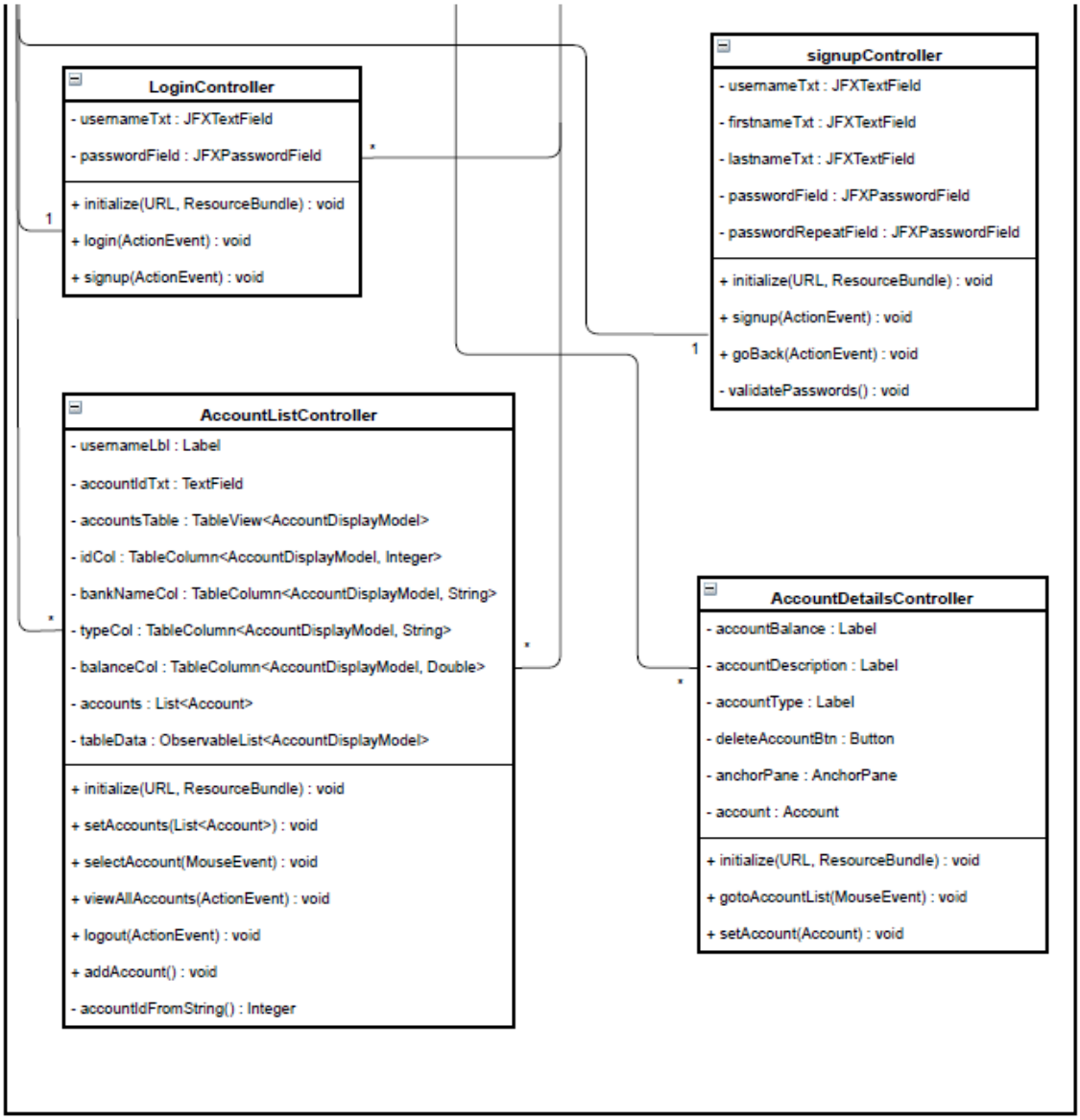


Figure 1: Class Diagram

4.2 Classes

Table 2: Interface ApplicationComponent

Class Name	com.github.comp354project.ApplicationComponent			
Type	Interface			
Inherits	N/A			
Implements	N/A			
Description	Class dependencies can be injected into the classes defined in the inject methods. This class is used for the Dagger2 injection framework			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	inject(MyMoneyApplication myMoneyApplication)	void	Injector for MyMoneyApplication class
	public	inject(LoginController loginController)	void	Injector for the LoginController class
	public	inject(AccountListController accountListController)	void	Injector for the AccountListController class
	public	inject(SignUpController signUpController)	void	Injector for the SignUpController
	public	inject(TransactionTableController tableController)	void	Injector for the TransactionTableController
	public	inject(UpdateUserAccountController updateUserAccountController)	void	Injector for the UpdateUserAccountController

Table 3: Class BusinessRulesConstants

Class Name	com.github.comp354project.BusinessRulesConstants			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	Contains business rules configuration for validators			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	public	Integer	USERNAME_MIN_LENGTH	The minimum length of a username
	public	Integer	USERNAME_MAX_LENGTH	The maximum length of a username
	public	Integer	PASSWORD_MIN_LENGTH	The minimum length of a password
	public	Integer	PASSWORD_MAX_LENGTH	The maximum length of a password
	public	Integer	CATEGORY_MIN_LENGTH	The minimum length of a category
	public	Integer	CATEGORY_MAX_LENGTH	The maximum length of a category
Methods	**Visibility**	**Name**	**Returns**	**Description**
None				

Table 4: Class Main

Class Name	com.github.comp354project.Main			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	Launches the application			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	main(String[] args)	void	The entry point of the application

Table 5: Class MyMoneyApplication

Class Name	com.github.comp354project.MyMoneyApplication			
Type	Class			
Inherits	Application			
Implements	N/A			
Description	Entry point for the GUI of the application			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Logger	logger	Logs event information
	public	MyMoneyApplication	application	The GUI entry point variable
	protected	SessionManager	sessionManager	Manages user sessions
	private	ApplicationComponent	component	Used to instantiate and inject classes
	private	Stage	primaryStage	Used to display the GUI
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	MyMoneyApplication	MyMoneyApplication	Constructs the class. Initializes an ApplicationComponent for depedency injection
	public	getScene()	Scene	Returns the current scene
	public	start(Stage primaryStage)	void	Displays the first GUI when the application launches
	private	updateStage(String fxml, String title, int width, int height)	T	Updates the current view
	private	setStageTitle(String title)	void	Sets the view's title
	public	displayLogin()	void	Displays the login view
	public	displaySignUp()	void	Displays the sign up view
	public	displayAccounts()	void	Displays the user accounts view
	public	displayUpdateUser()	void	Displays the update user view
	public	displayAccountDetails(Account account)	void	Displays the account details view
	public	displayAllAccountDetails(List accounts)	void	Displays all accounts details view

Table 6: Class Account

Class Name	com.github.comp354project.service.account.Account			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	Used to hold the account information of the user			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Integer	ID	bank account identification number
	private	String	type	type of bank account (chequing, savings, ect)
	private	Double	balance	Amount inside the account
	private	User	user	name of the user
	private	ForeignCollection<Transaction>	transactions	transaction object
Methods	**Visibility**	**Name**	**Returns**	**Description**
	none	none	none	none

Table 7: Class AccountService

Class Name	com.github.comp354project.service.account.AccountService			
Type	Class			
Inherits	N/A			
Implements	IAccountService			
Description	Class used to request information from the bank database in order to add or delete an account to myMoney application			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Logger	logger	logger object attribute used to keep track of errors
	private	Dao<Transaction,Integer>	transactionDao	Dao object used to query the database
	private	Dao<User,Integer>	userDao	Dao object used for quering the database
	private	IRemoteAccountService	remoteAccountService	attribute used to access database
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	addAccount	Account	method to request bank information from the database
	public	deleteAccount	void	method to delete a particular account from myMoney application
	public	transform	Account	method to create the appropriate banking info to display for the myMoney app based on the retrieved banking info
	public	Transaction	transform	method to create the appropriate transaction info to display for the myMoney app based on the retrieved banking info

Table 8: Class AccountServiceModule

Class Name	com.github.comp354project.service.account.AccountServiceModule			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	used to return need objects for account and transaction needs			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	None	none	none	none
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	provideTransactionService	transactionService	return transactionService Object
	public	provideAccountService	accountService	returns accountService Object

Table 9: Interface IAccountService

Class Name	com.github.comp354project.service.account.IAccountService			
Type	Interface			
Inherits	N/A			
Implements	N/A			
Description	interface class for adding and deleting an account			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None	None	None	none	none
Methods	**Visibility**	**Name**	**Returns**	**Description**
	N/A	addAccount	N/A	none
	N/A	deleteAccount	N/A	none

Table 10: Interface ITransactionService

Class Name	com.github.comp354project.service.account.ITransactionService			
Type	Interface			
Inherits	N/A			
Implements	N/A			
Description	interface class to updating transactions based on categories			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None	None	None	None	None
Methods	**Visibility**	**Name**	**Returns**	**Description**
	N/A	updateTransactionCategory	Transaction	N/A

Table 11: Class Transaction

Class Name	com.github.comp354project.service.account.Transaction			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	Class used to contain the attributes needed to hold a transaction's details			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Integer	date	date of a transaction
	private	Double	amount	dollar amount of a transaction
	private	String	type	the type of a transaction
	private	String	category	the category of a transaction
	private	Integer	sourceID	ID number
	private	Integer	destinationID	ID number
	private	Account	account	name of the account
Methods	**Visibility**	**Name**	**Returns**	**Description**
	None	None	None	None

Table 12: Class TransactionService

Class Name	com.github.comp354project.service.account.TransactionService			
Type	Class			
Inherits	N/A			
Implements	ITransactionService			
Description	class used to help with transaction changes			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Logger	logger	object used to interact with TransactionService class
	private	Dao<Transaction,Integer>	transactionDao	object used to perform methods related to transactions
	private	ICategoryNameValidator	categoryValidator	object used to validate if a category is correct
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	TransactionService	N/A	constructor
	public	updateTransactionCategory	Transaction	used to update a specific transation

Table 13: Class DaoModule

Class Name	com.github.comp354project.service.dao.DaoModule			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	DAO module to bind interfaces to their interfaces and provide them to the classes that require them			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Logger	logger	Logs event information
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	provideRemoteAccountDao(IConnectionProvider connectionProvider)	Dao<RemoteAccount, Integer>	Returns the implementation of a RemoteAccountDao
	public	provideAccountDao(IConnectionProvider connectionProvider)	Dao<Account, Integer>	Returns the implementation of an AccountDao
	public	provideTransactionDao(IConnectionProvider connectionProvider)	Dao<Transaction, Integer>	Returns the implementation of a TransactionDao
	public	provideUserDao(IConnectionProvider connectionProvider)	Dao<User, Integer>	Returns the implementation of a UserDao

Table 14: Class ConnectionModule

Class Name	com.github.comp354project.service.sqlite.ConnectionModule			
Type	Class			
Inherits	N/A			
Implements	N/A			
Description	Module that creates a connection to the database			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Returns**	**Description**
	protected	provideConnection(ConnectionProvider connectionProvider)	IConnectionProvider	Returns the implementation of a ConnectionProvider

Table 15: Class ConnectionProvider

Class Name	com.github.comp354project.service.sqlite.ConnectionProvider			
Type	Class			
Inherits	N/A			
Implements	IConnectionProvider			
Description	Instatiates a connection to an SQLite database			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	private	Logger	logger	Logs events
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	ConnectionProvider()	ConnectionProvider	Constructs the class
	public	getConnectionSource()	JdbcConnectionSource	Returns a database connection source

Table 16: Interface IConnectionProvider

Class Name	com.github.comp354project.service.sqlite.IConnectionProvider			
Type	Interface			
Inherits	N/A			
Implements	N/A			
Description	Instatiates a connection to a database			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Returns**	**Description**
	public	getConnectionSource()	JdbcConnectionSource	Returns a database connection source

Table 17: Class GetRemoteAccountRequest

Class Name	com.github.comp354project.service.account.remote.GetRemoteAccountRequest				
Type	Public				
Inherits	N/A				
Implements	N/A				
Description	Retrieve the remote account request				
Attributes	**Visibility**	**Data Type**	**Name**	**Description**	
	Private	Integer	accountID	Identification of an account	
Methods	**Visibility**	**Name**	**Returns**	**Description**	**Throws**
None					

Table 18: Class GetRemoteAccountResponse

Class Name	com.github.comp354project.service.account.remote.GetRemoteAccountResponse				
Type	Public				
Inherits	N/A				
Implements	N/A				
Description	Retrieve the response for the remote account request				
Attributes	**Visibility**	**Data Type**	**Name**	**Description**	
	Private	RemoteAccount	account	The remote account	
Methods	**Visibility**	**Name**	**Returns**	**Description**	**Throws**
None					

Table 19: Interface IRemoteAccountService

Name	com.github.comp354project.service.account.remote.IRemoteAccountService			
Type	Public			
Inherits	N/A			
Implements	N/A			
Description	The interface for the remote account service (request and response)			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Throws**	**Description**
	Public	IRemoteAccountService	ValidationException	Remote account service

Table 20: Class RemoteAccount

Class Name	com.github.comp354project.service.account.remote.RemoteAccount				
Type	Public				
Inherits	N/A				
Implements	N/A				
Description	The remote account with details				
Attributes	**Visibility**	**Data Type**	**Name**	**Description**	
	Private	Integer	ID	ID of the account	
	Private	String	bankName	Name of the bank	
	Private	String	Type	Type of the account	
	Private	Double	balance	Balance of the account	
	Private	ForeignCollection	transactions	Transactions of the account	
Methods	**Visibility**	**Name**	**Returns**	**Description**	**Throws**
None					

Table 21: Class RemoteAccountModule

Class Name	com.github.comp354project.service.account.remote.RemoteAccountModule			
Type	Public			
Inherits	N/A			
Implements	N/A			
Description	The module for remote account class			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
None				
Methods	**Visibility**	**Name**	**Returns**	**Description**
	Default	provideRemoteAccountService	remoteAccountService	Module provide the remote account service

Table 22: Class RemoteAccountService

Class Name	com.github.comp354project.service.account.remote.RemoteAccountService			
Type	Public			
Inherits	N/A			
Implements	IRemoteAccountService			
Description	The services that the remote account can provide			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	Private	Logger	logger	Gets the log of the Remote AccountService.class
	Private	Dao <Remote Account, Integer >	Remote AccountDao	RemoteAccountDoa
Methods	**Visibility**	**Name**	**Throws**	**Description**
	Public	RemoteAccountService(Dao <RemoteAccount,Integer >remoteAccountDao)	N/A	The constructor class for Remote AccountService
	Public	getAccount(GetRemoteAccountRequest request)	Validation Exception	Return the account information if there is a request for it and if it exists

Table 23: Class RemoteTransaction

Class Name	com.github.comp354project.service.account.remote.RemoteTransaction			
Type	Public			
Inherits	N/A			
Implements	N/A			
Description	The remote transaction class			
Attributes	**Visibility**	**Data Type**	**Name**	**Description**
	Private	Integer	ID	Identification of the remote transaction
	Private	Integer	date	Date of the transaction
	Private	Double	amount	Amount of money transitioned
	Private	String	type	Type of transaction
	Private	Integer	SourceID	Identification of the source where the money was originally resided
	Private	Integer	destinationID	Identification of the destination where the money will be transitioned
	Private	Remote	account	The main account of the user
Methods	**Visibility**	**Name**	**Returns**	**Description**
None				

4.3 Glossary of Domain Concepts

Table 24: 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 organized, 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.

4.4 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 describes the actions that occur when the user clicks on the sign up button

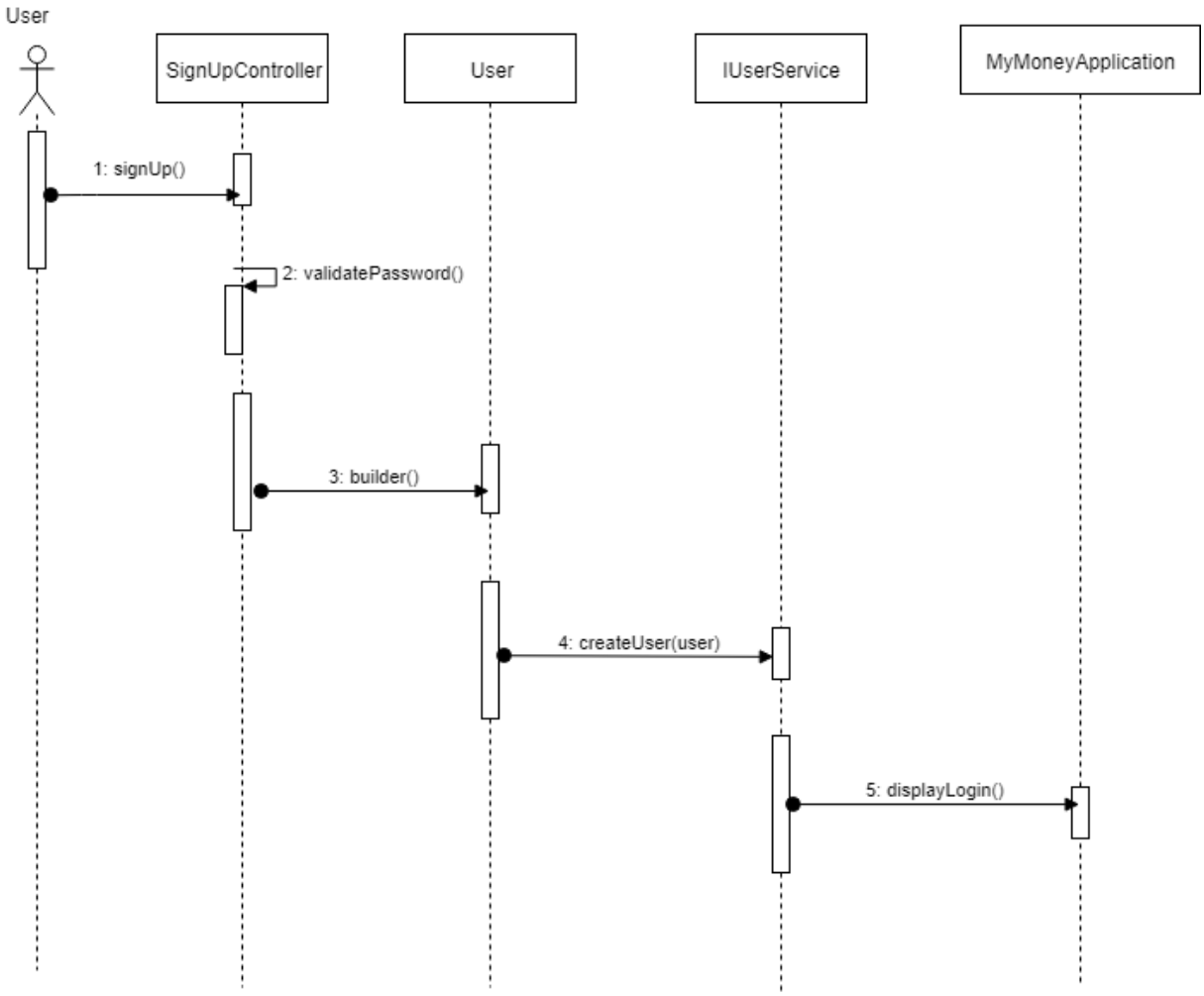


Figure 2: Use case 1 Sequence Diagram

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.

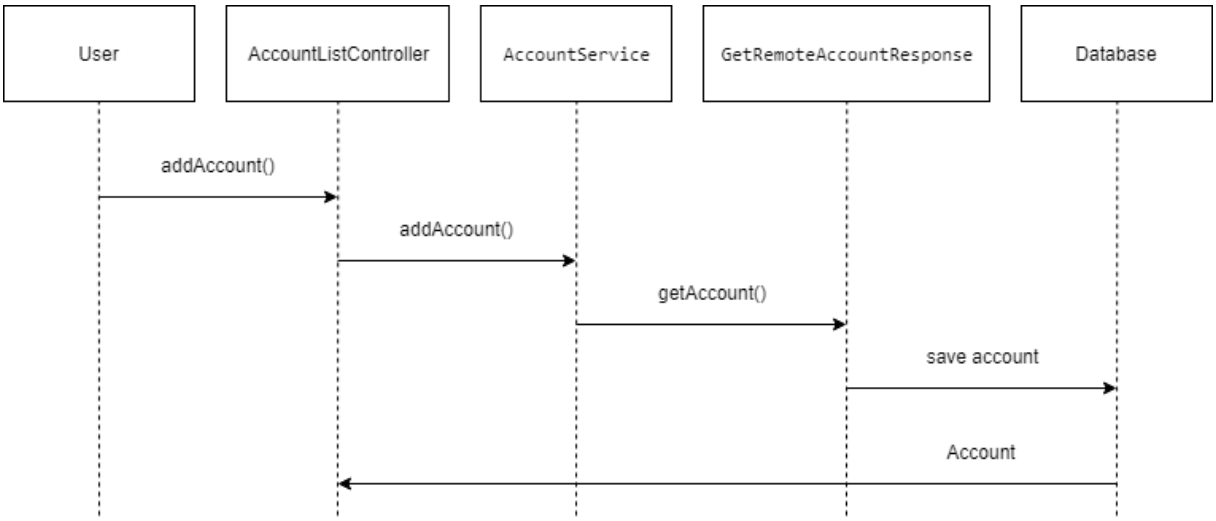


Figure 3: Use case 3 Sequence Diagram

Use Case 5: View Transactions for Specific Bank Account

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

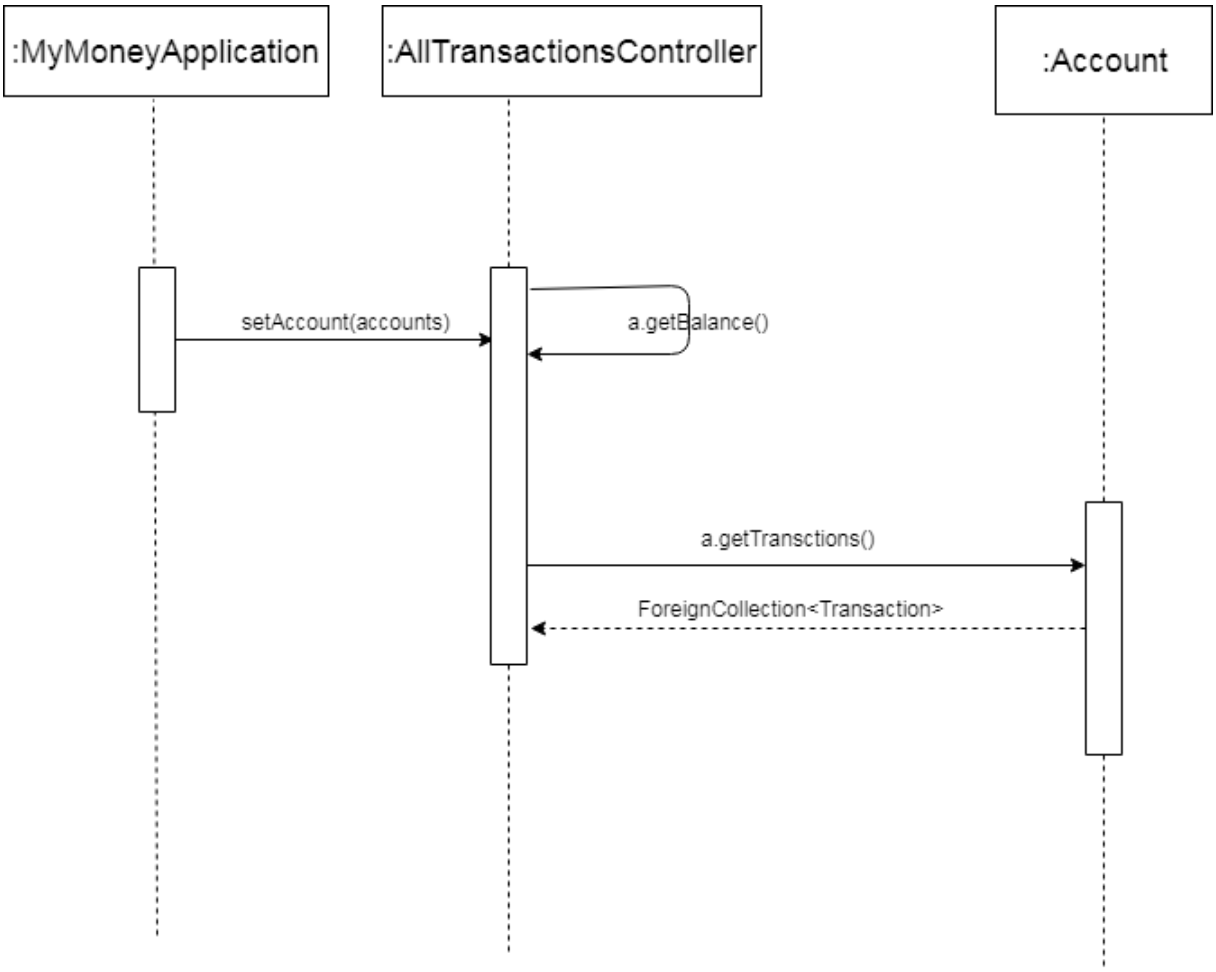


Figure 4: Use case 5 Sequence Diagram

Use Case 6: View All Transactions from all Bank Accounts

The following scenario describes the actions that occur when the user click the button "view all transactions" for viewing all transactions from all bank accounts.

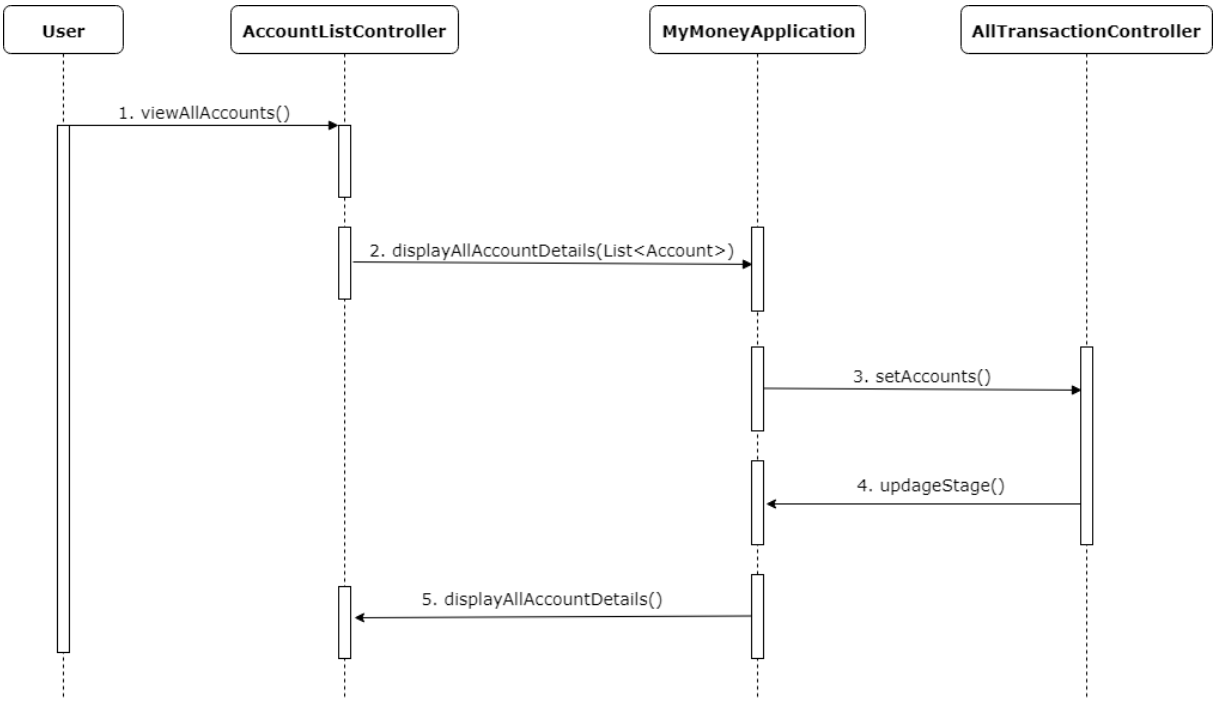


Figure 5: UseCase 6 Sequence Diagram

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