COMP 354
Requirements for the project myMoney

Team PA-PK

February 12, 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

Table 2: Revision history

Version	Date
1.0	10th February 2018

Contents

List of Figures

List of Tables

1 Document Purpose

The purpose of this document is to define requirements for the desktop application myMoney. This document may thus be to orient the development of the application. It seeks to understand the requirements of the problem, formulate the necessary functions and properties needed to answer this problem and its requirements, and then test these functions against these requirements. Hence, it may be used by our users to specify the problem and its requirements, by the developers to understand what functions their system must implement, and what to test their system against. The primary audience is the development team of the system, and the project testers for fine-tuning their testing strategy. It may also be read by users of the end product to find out the functionality of the system.

2 Project description

2.1 Introduction

At the present time, users who have more than one bank account can quickly get overwhelmed with the differing methods of access and interfaces for each account. It becomes arduous to access every account and difficult to visualise how much money one has and how one's budget changes on a day-to-day basis. There exists a plethora of software for money management, each greatly varying in design due to the complex and multifarious clientele. Existing applications of quality also tend to have extensive budgeting capabilities, and setting them up then learning to use them requires investing some number of hours that often scares off casual users who aren't interested in complex accounting functions. This lack of appropriate software for centralised, simple money management costs time and frustration for the common user, and discourages them from fully taking advantage of services that are provided by their banks, for fear of the complexity that make come with such services. Even simply keeping track of their finances becomes a daunting task, which negatively impacts both the users and the banks. We seek to design a desktop application to solve the most common issues in a simple and lightweight manner. a more in-depth discussion of the user profile is available in section 2.6.

2.2 System Context

Our system will mainly involve the user interacting with a desktop application interface. The user shall provide information about his bank accounts to the interface, which the system will store it in a local database. The system then establish a connection to the banks, passing along the user information. It will then receive the data associated with the bank account(s) of the user, which it will then display to the user. Section 4 gives a more detailed view of all of the functions that the system will be able to perform.

2.3 Scope

myMoney aims to be a small, simple tool which provides an easy method of viewing accounts across multiple banks, their transactions, and categorisation tools for grouping transactions across multiple accounts. One may create a user account and link their bank accounts to easily access them through

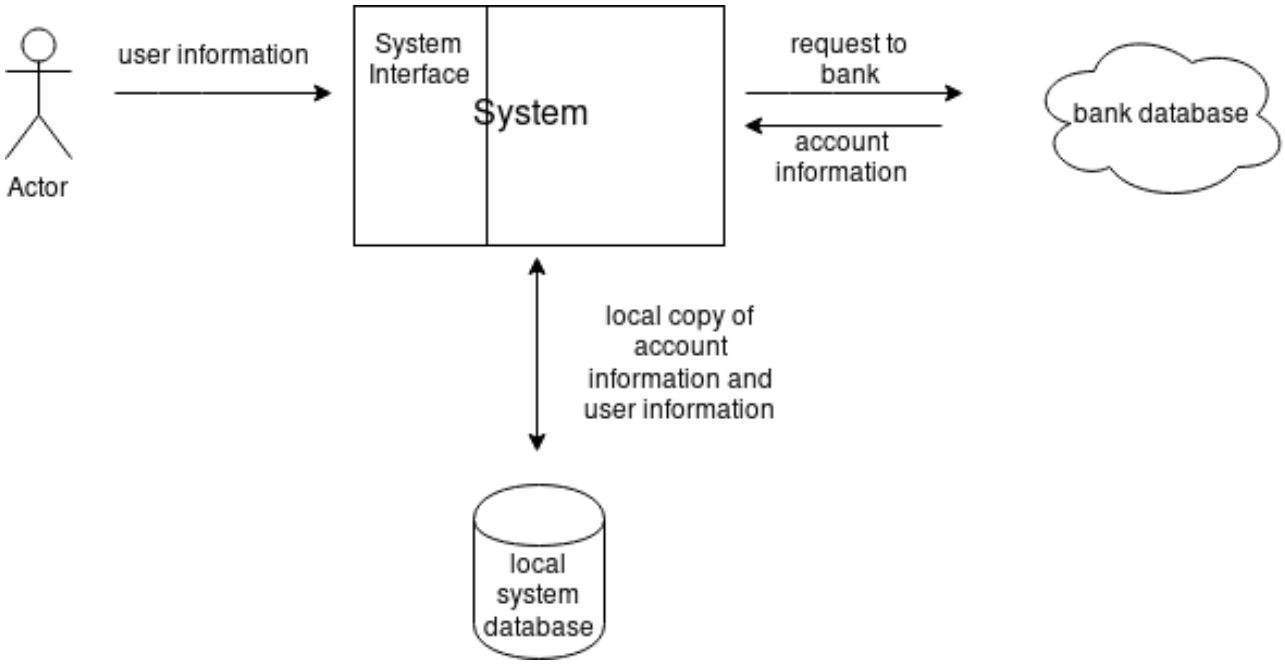


Figure 1: System Context

myMoney. They may check the transactions of a specific account or all of accounts at once, sort the transactions by date, type, or any other property of the transaction, and categorise certain transactions for better organisation and minimal budgeting. myMoney is intended to work alongside banking institutions: it relies on the data provided by banks to create a transaction history for the users. In short, it seeks to be an application for managing expenses so as to quickly make judgment calls on financial decisions, in a way that is accessible to even the most financially inexperienced user.

2.4 Business Goals

- **Compete with existing solutions**

  We are not the sole developers of budgeting applications by far. Not just that, but users already have a procedure for accessing their bank accounts directly through interfaces provided by banks. As such, if we want to attract users to our product, we must be able to compete on the same playing field as these current applications/procedures. We must, at the very least, meet their level of performance, ease of use, security, and other qualities described in 3 for our system.

- **Target market: wide user-base composed mainly of millennials, students, new professionals**

  Our customer group user group, and development team happen to involve the same people: young professionals and students with little to no expected background in financing. As customers, this group has no interest in complex budgeting functions (if they do, they are not the target audience aimed by this system), but instead seeks some way of organising tracking their total assets. This group favours quickly accessed applications for frequent yet momentary usage. They

want to quickly monitor their cash spending, and make long-term financial decisions based on clear, understandable data that they can access nearly immediately.

- **Future-proof, long-term robustness of the system** We wish to be able to support this system on a long-term basis. We also want to support users who might have a rather long transactional history, or might, over the years of using our application, gain such a transactional history, and these should be accomodated by the system to guarantee long-term success.
- **Reduce total cost of ownership**

  For the above business goal, it is imperative for our application to be relatively cheap to maintain and support after development, as it would lengthen the lifetime of our system.

2.5 Domain Concepts

Domain Model

We herewith provide the domain model of our system, useful for users and documenters seeking to understand the general setup of our system. Useful to understanding this model is the the context of this system, provided in figure 1, to see the connections between the system and the actors. For a more in-depth diagram of the system, see figure 3 below.

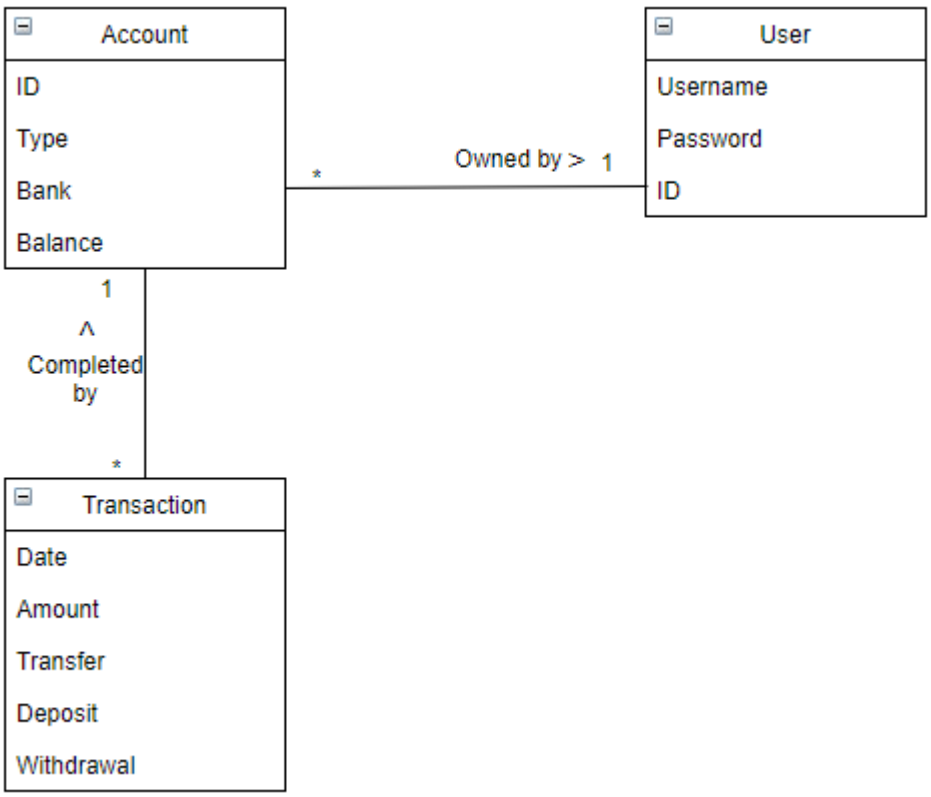
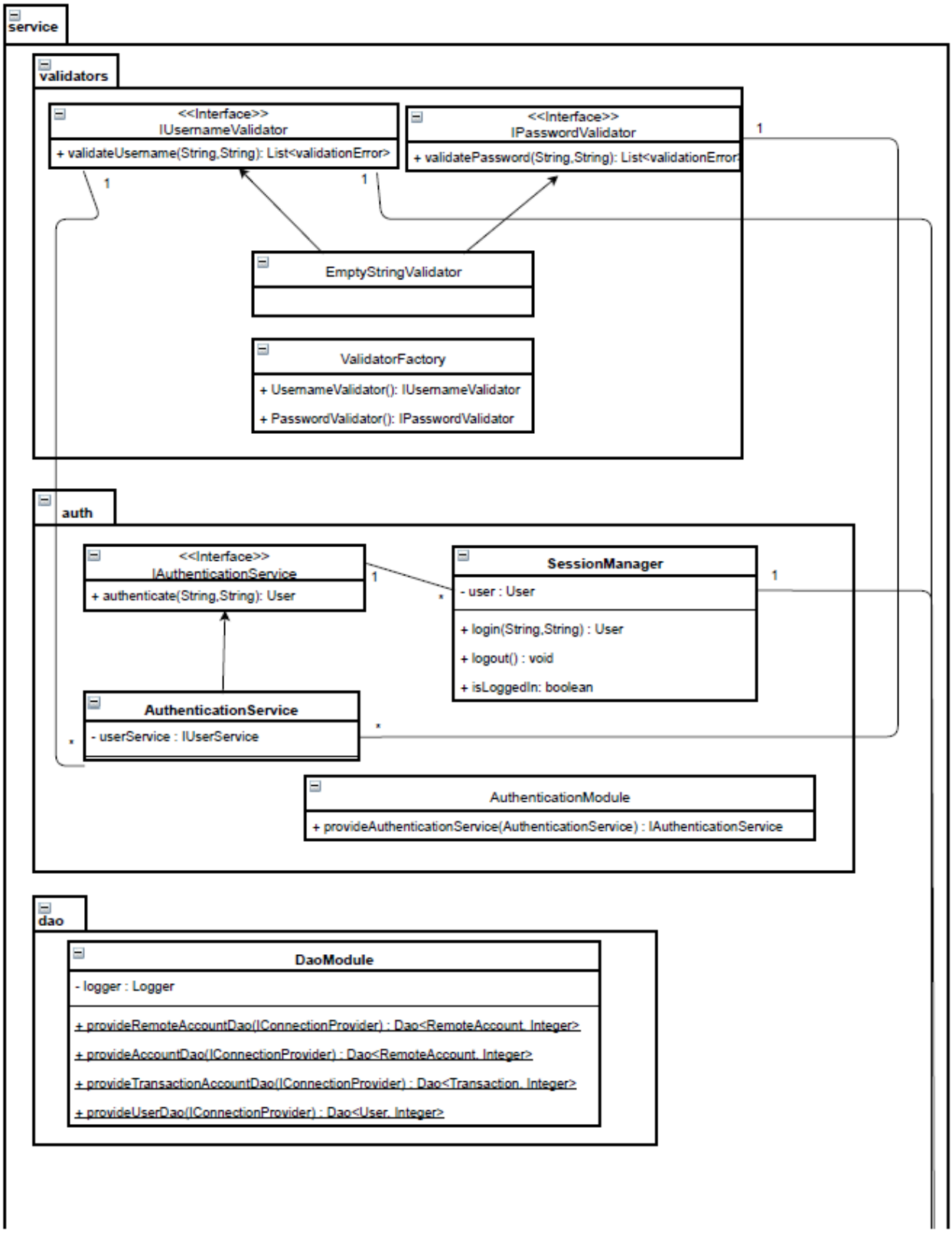
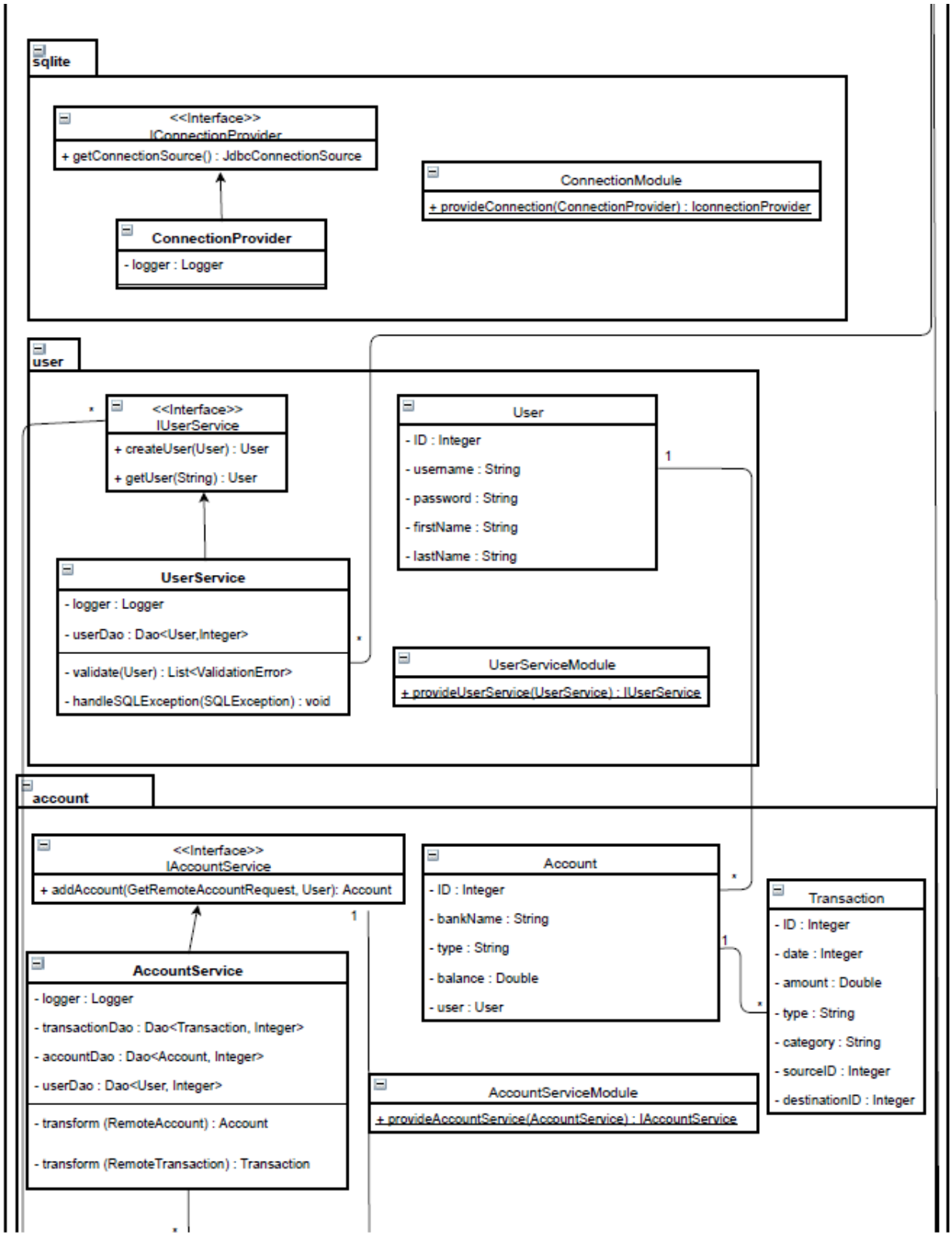


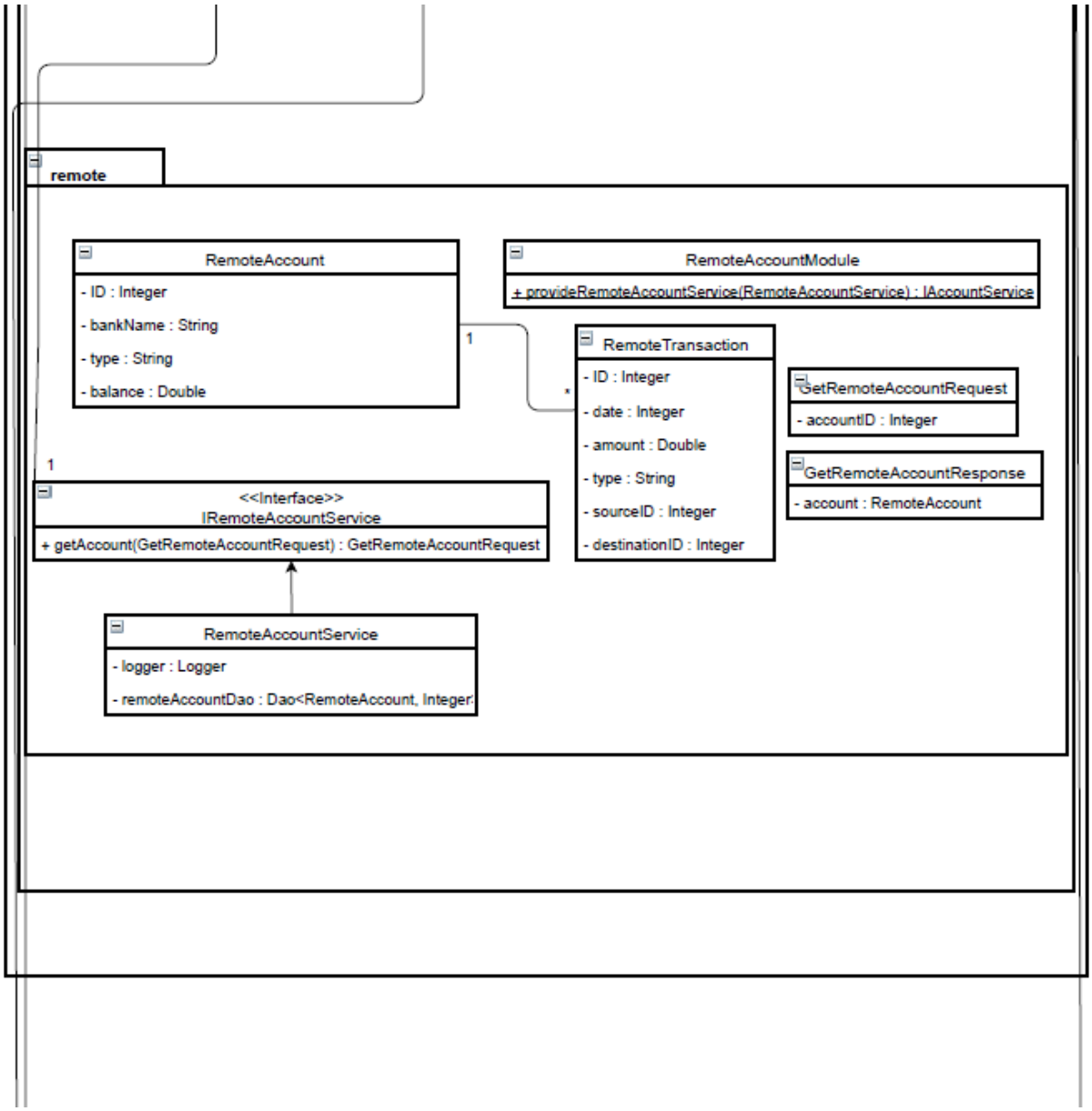
Figure 2: Domain Model

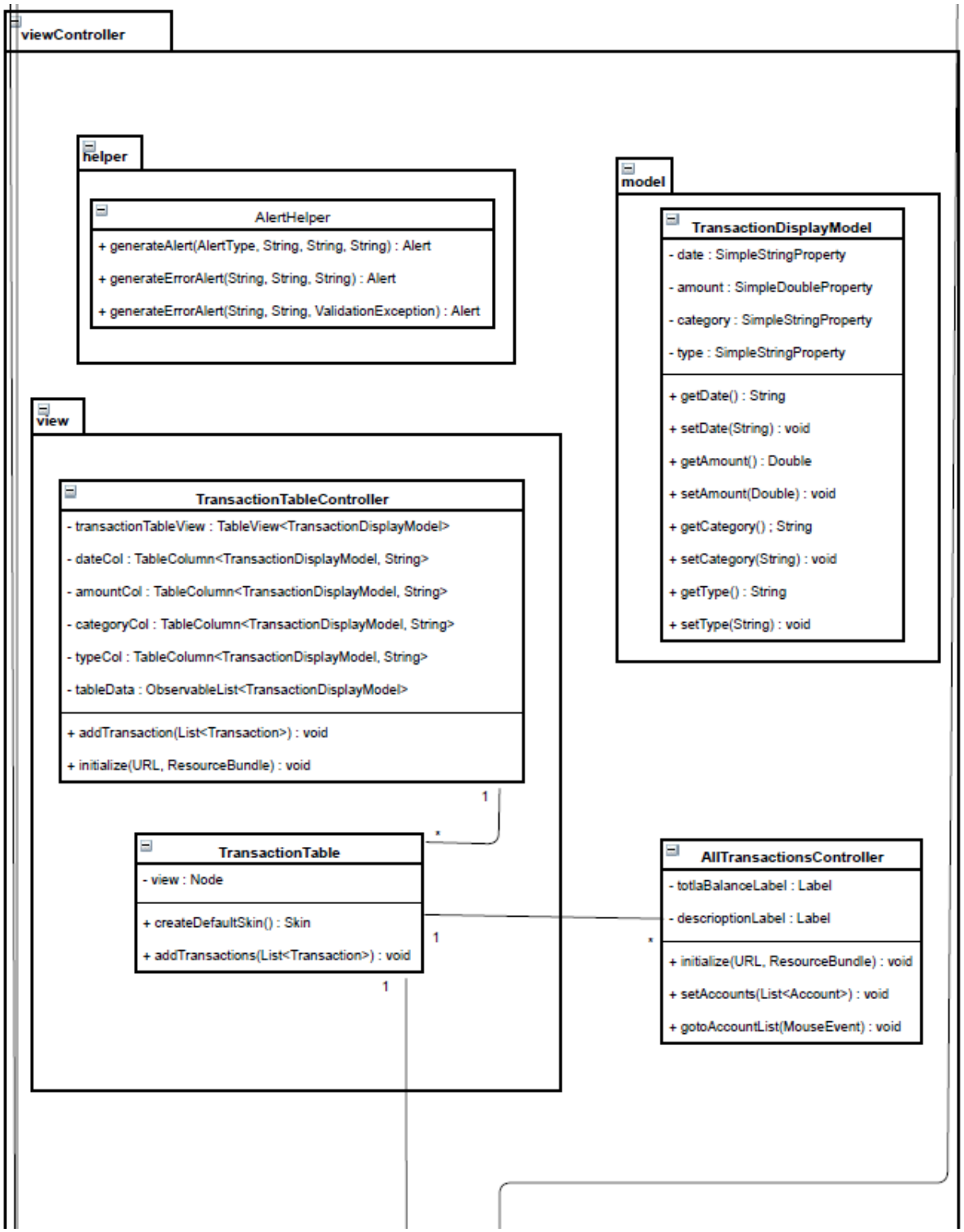
Class Diagram

We herewith provide the class diagram of our system, useful for the system developers and testers. If a term is unclear, view section 4.1 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

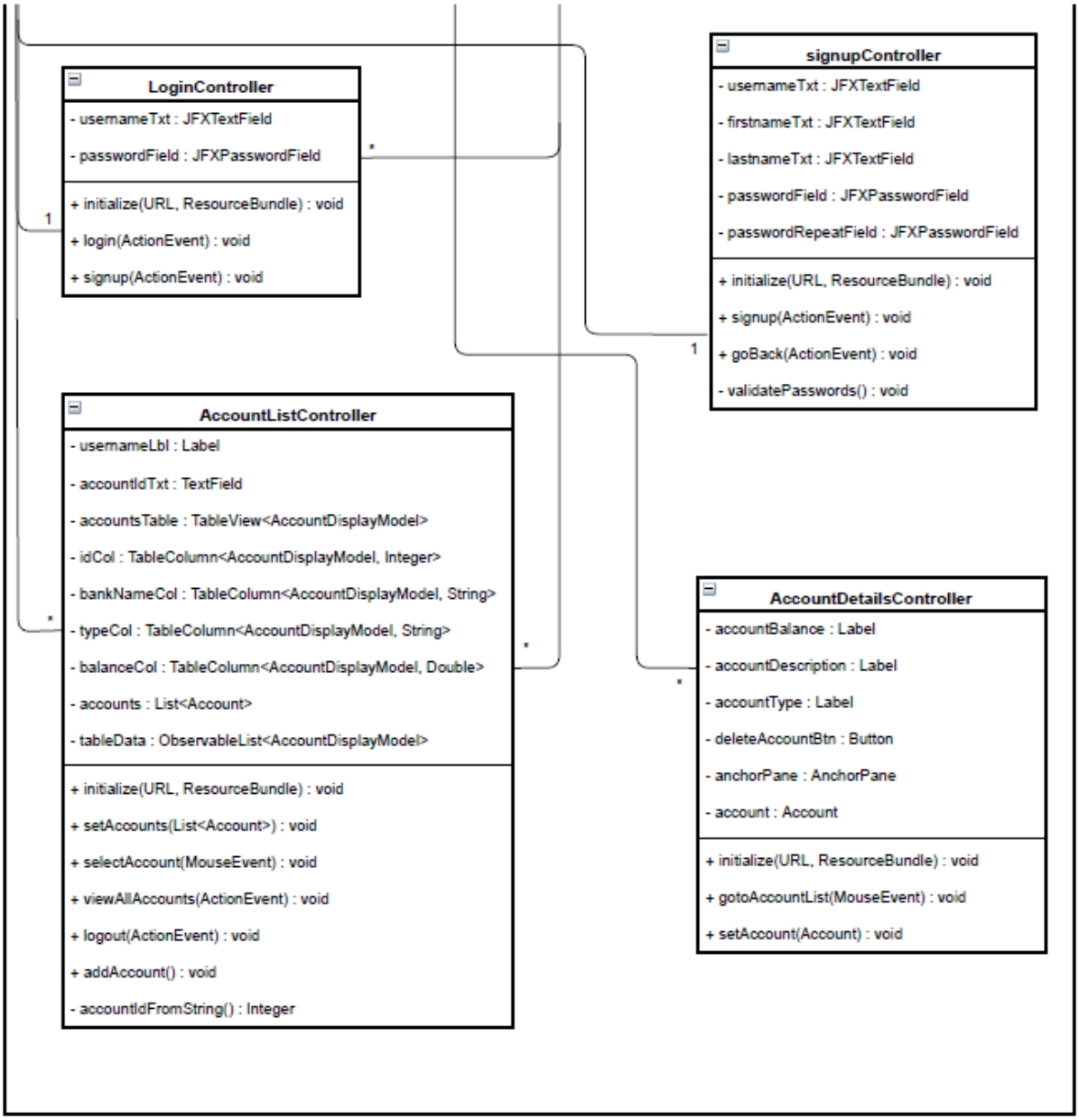


Figure 3: Class Diagram

Glossary of Domain Concepts

labelglossary

Table 3: 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.

2.6 Actors

User

Our main actor is the user. All use cases are triggered by the user, as it their requests that directly cause the bank system or our system to take action. All identification needed to access the bank system will be provided by the user. Using this information, the system will be able to answer queries made by the user as described in the use cases.

Bank

Our sole other actor is the bank(s) system. Each bank system provides an API for accessing its bank account data. Our system will pull this data directly from the bank systems (who act as secondary actors), using identification as given by the user for the bank's authentication system. In other words, our system will merely act as a middle man between the user demanding information in a specific format, and the banks holding that information in a inconvenient format.

3 Functional Requirements and Business Rules

3.1 Non-Functional Requirements

Table 4: Non-Functional Requirement 1 - Reliability and usability

Action	**Reliability and usability**
Requirement ID:	01
Type:	Quality of service
Description:	We want to guarantee that our system's reliability is only constrained by the reliability of the banks servers. We also to ensure the application is always usable, except during maintenance or communication with the database.
Reasoning:	The current procedure which users employ benefits from near perfect up-time, as its only constraint is the reliability of the bank servers. In order to be competitive with this current solution, we must match this level of reliability.
Quality attribute scenario:	System operates mostly offline, and can function even without access to the internet using past data. System meets security and connectivity performance of current procedure.

Table 5: Non-Functional Requirement 2 - Simplicity and ease-of-use

Action	**Simplicity and ease-of-use**
Requirement ID:	02
Type:	Quality of service
Description:	The system should be easy to understand and not overburdened by unused features, allowing for clear display of relevant information.
Reasoning:	The current procedure for users to view their bank transactions across different accounts requires them to log in each individual bank account interface, then compare each distinct format for the bank accounts manually. This is tedious for most users, and hence easy to outperform and become competitive with the current procedures. It is also in our interest to pursue this, as our business targets a more casual user-base which tend to favours a more simpler and more lightweight application.
Quality attribute scenario:	Installation of the system should be easy and require very little, if any, decisions from the user beyond choosing to install the system. With no training, a user should be able to intuitively learn and use the system in a short amount of time.

Table 6: Non-Functional Requirement 3 - Performance

Action	**Performance**
Requirement ID:	02
Type:	Quality of service
Description:	The system should ensure minimal load times, notably when connecting to the bank and the local database.
Reasoning:	With our target audience in mind, we want to be able to provide adequate performance when the system is used in frequent, short-time bursts. These sorts of applications tend to be popular with millennial audiences. Focusing on performance also renders our system more competitive, as our target audience highly favours quick interactions with the interface.
Quality attribute scenario:	System files should be small, less than 50MB. When the system connects to the internet (with reasonable bandwidth) to fetch transactions, they are returned to the user in less than 2 seconds.

Table 7: Non-Functional Requirement 4 - Maintainability and testability

Action	**Maintainability and testability**
Requirement ID:	03
Type:	Quality
Description:	The system should be maintainable on a long-term basis, A testing suite is imperative for adaptive development, as it aids in reducing time and cost of debugging, and lets us quickly conduct system diagnostics.
Reasoning:	Due to the dependency of the system upon the banks' API, it is necessary that the system should be easy to alter and maintain, and hence be capable of adapting to third-party changes. This both reduce costs of ownership and favours long-term use of the application.
Quality attribute scenario:	New modifications to the system should be fully covered by unit tests before another modification is implemented.

Table 8: Non-Functional Requirement 5 - Security

Action	**Security**
Requirement ID:	04
Type:	Quality
Description:	The system should guarantee a level of security, requiring valid credentials in order to access any sensitive information and denying access otherwise.
Reasoning:	A user's bank information is incredibly sensitive, and any possible security flaw could cripple the entire system and destroy any sort of trusted relationship that users may have developed with our business. It is thus of utmost importance that any data coming from the banks are accessible solely with proper authentication. Improper security would certainly negatively impact user trust in our business.
Quality attribute scenario:	Database should be encrypted, and testing suite should include verification of security measures on the system.

Table 9: Non-Functional Requirement 6 - Portability

Action	**Portability**
Requirement ID:	05
Type:	Quality
Description:	Since the system is fairly lightweight, it should also focus on compatibility with older hardware (within reason) and support numerous operating systems, to encourage a wide user-base.
Reasoning:	We want to cater to a wide and somewhat casual audience, who may not be technologically adept and still rely on old (but familiar) hardware. The lightweight quality of our application means that it would easily be run on older hardware, provided we also ensure compatibility with said hardware.
Quality attribute scenario:	System should support older versions of popular OS's, such as windows 7/windows vista.

Table 10: Non-Functional Requirement 7 - Scalability

Action	**Scalability**
Requirement ID:	06
Type:	Quality
Description:	The system should function even on a long-term basis, and hence be capable of handling a growing number of transactions. The scaling size of the database should be managed by the system so as not to overwhelm the hardware it is running on.
Reasoning:	As we foresee our system being used on a long period of time, we should also acknowledge the possibility of users acquiring a large history of transactions over time. We should thus plan accordingly and put measures of precautions so as not to overwhelm their machines when loading in a large database. Crippling their machine could result in a negative perception of business and system.
Quality attribute scenario:	System should use SQLite to minimize database size, and take precautions not to load an entire database in memory should the number of transactions exceed a size that may be unmanageable by the hardware.

Table 11: Non-Functional Requirement 8 - Data integrity

Action	**Data integrity**
Requirement ID:	07
Type:	Quality
Description:	Bank accounts data (description as well as transactions) must match the data provided by the banks.
Reasoning:	For obvious reasons, we want to guarantee data integrity. Not meeting this requirement would harm the reputation of the business, and render the system rather useless.
Quality attribute scenario:	system should verify data validity, and tests should include verification of matching data between the local database and the banks' database

Table 12: Non-Functional Requirement 9 - Java

Action	**Java**
Requirement ID:	08
Type:	Design Constraint
Description:	The system should be programmed in Java.
Reasoning:	To take advantage of the JVM's portability, we would be able to develop for numerous platforms whilst having to maintain a single version of the system.

Table 13: Non-Functional Requirement 10 - Object-oriented design

Action	**Object-oriented design**
Requirement ID:	09
Type:	Design Constraint
Description:	The design of the system should be object-oriented.
Reasoning:	As we will be using java, we should embrace the style of the language and use object-oriented programming. This will also ease maintainability and portability.

Table 14: Non-Functional Requirement 11 - Model-view-controller architecture

Action	**Model-view-controller architecture**
Requirement ID:	10
Type:	Design Constraint
Description:	The design of the system should use a MVC architecture.
Reasoning:	As our main development time for creating a prototype is fairly short, using MVC will allow us to properly segment each part of our code, and thus give it a strict structure, as well as make it more easily maintainable and testable.

Table 15: Non-Functional Requirement 12 - SQLite database

Action	**SQLite database**
Requirement ID:	11
Type:	Design Constraint
Description:	The system should use SQLite
Reasoning:	As our system is fairly simple and favours a lightweight design, using a more complex database would be a waste of resources.

4 Functional Requirements

This section will cover the functional requirements associated with the myMoney app. These are the software capabilities that must be present in order for the user to carry out the services provided by the app or to execute the use cases.

- **User account creation:** The system allows the creation of user accounts, with information (user-name, password, first name, and last name) provided by the user.
- **User account authentication:** The system shall grant a user access only to a properly authenticated user. If a user does not enter credentials, the system shall notify the user of the failure to authenticate.
- **User account management:** The system shall require proper authentication to modify and/or delete a user account.
- **Bank account management:** The system permits a logged-in user to add, manipulate, or remove bank accounts that are connected to the user account.
- **Transaction management:** The system permits a logged-in user to access the details of transactions associated with all bank accounts that were added to the user account. The system obtains these transactions through the banking system's API. The system allows different display formats, such as 'sorted by date' or 'sorted by type' or 'sorted by bank account.
- **Categorisation:** The system permits the categorisation of transactions, a property by which the transactions may be sorted.

4.1 Glossary

4.2 Description of File Format: Input

The user enters plain text through the interface of the system. The banking systems provide bank account data by passing a copy of their account, in binary form.

5 Description of File Format: Output

The system outputs its database data in plain text form, displayed through its interface.

6 Use Cases

6.1 Overview

Use cases 1 through 4 deal with the user manipulating his accounts. Use cases 5 through 7 and 9 deal with the user viewing the data in different formats. Use case 8 deals with the user manipulating the transactions' formats. For iteration 1, we have only included use case 1, 3, 5, and 6, as those are the ones we have fully implemented.

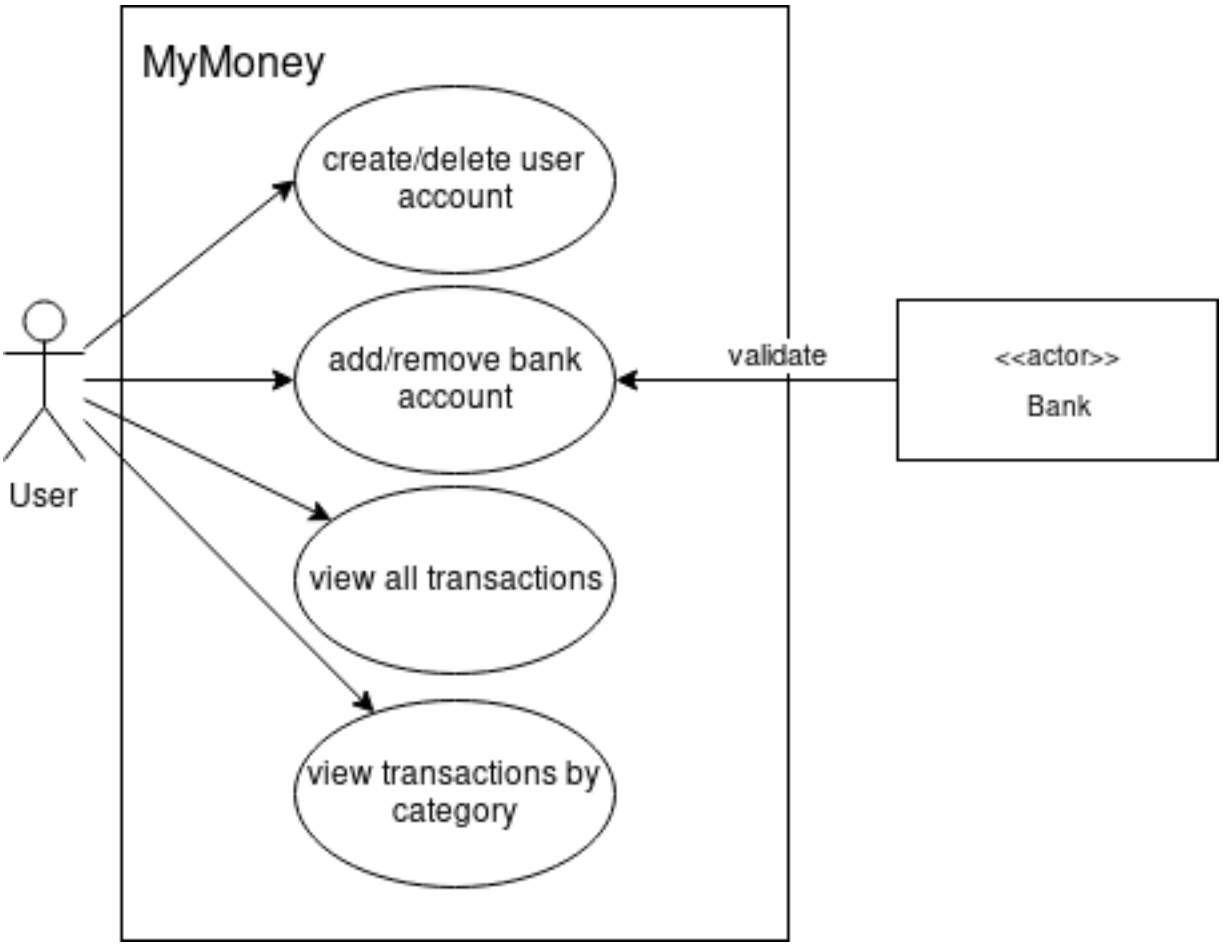


Figure 4: Use Case Diagram

Table 16: Use Case 1 - Create User Account

Action	**Create User Account**
Case ID	01
Summary	User gives information about a new user account, system validates it and creates the account.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants fast and easy account creation, clear and comprehensible display, proof of successful account creation. 2. Company: Wants user interests to be fulfilled, wants to prevent erroneous input, wants fast communication with the local account database as well as fault tolerance in case of database conflicts, issues with editing authorization, or other possible database problems.
Pre-Conditions	User has opened the application and is in the sign-up menu.
Success Guarantee	Account successfully saved in local account database, with name and password as specified by the user.
Main Success Flow	1. User enters a user-name, first name, last name, and password. 2. System validates user-name and password (format, whether user-name is already used, etc). 3. System creates new account. 4. System notifies the user of the successful account creation, then returns to home menu.
Exceptions	
Post-Conditions	
Priority	
Traces to Test Cases	

Table 17: Use Case 2 - Delete User Account

Action	**Delete User Account**
Case ID	02
Summary	User deletes a user account from the local accounts database, removing all bank accounts and information associated with that account.

Table 18: Use Case 3 - Add Bank Account to a User Account

Action	**Add Bank Account to a User Account**
Case ID	03
Summary	User gives information about a new bank account, system sends it to the bank for verification then creates necessary entries in the local database once the bank approves the information.
Scope	money and budget management application
Level	user-goal
Actors	**User**, Bank
Stakeholders and Interests	1. User: Wants fast and easy account creation, clear and comprehensible display, proof of successful account creation. 2. Company: Wants user interests to be fulfilled, wants to prevent erroneous input, wants fast communication with the local account database as well as the bank, wants fault tolerance in case of database conflicts, issues with the bank, or other possible local database problems. 3. Bank: Wants to satisfy its customer base, wants correctly formatted account information given to its API, inexpensive and non-redundant communication of bank account data to third-party applications.
Pre-Conditions	User has logged in a user account and is in the user home menu.
Success Guarantee	Bank account successfully saved in local account database, with information corresponding the data validated by the bank.
Main Success Flow	1. User enters his/her bank account number. 2. System validates input, and sends it to the bank for verification and connection. 3. Bank validates bank account number, and then responds with the bank account information. 4. System records the valid bank account details securely in its database, and notifies the user of the successful addition of the bank account in the database.
Exceptions	
Post-Conditions	
Priority	
Traces to Test Cases	

Table 19: Use Case 4 - Remove Bank Account from a User Account

Action	**Remove Bank Account from a User Account**
Case ID	04
Summary	User deletes a bank account from the local database.

Table 20: Use Case 5 - View Transactions for Specific Bank Account

Action	**View Transactions for Specific Bank Account**
Case ID	05
Summary	User selects specific bank account and views transactions associated with with selected bank account
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants quick and convenient viewing of previous transactions from one specific bank account 2. Company: Wants to give user ability to micromanage every aspect of the application down to each bank account and transaction.
Pre-Conditions	User has created and logged into a user account, and has added at least one bank account to his/her myMoney account.
Success Guarantee	User can view transaction by bank account.
Main Success Flow	1. User selects specific bank account from list of all accounts. 2. System displays all previous transactions under specified bank account. 3. User selects desired transaction. 4. System shows all information about desired transaction, such as date and amount withdrawn, deposited, or transferred.
Exceptions	
Post-Conditions	
Priority	
Traces to Test Cases	

Table 21: Use Case 6 - View All Transactions from all Bank Accounts

Action	**View All Transactions from all Bank Accounts**
Case ID	06
Summary	User can view all transactions that have been made from all bank accounts
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants easy and convenient viewing of all transactions among all bank accounts. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	User has created and logged into a user account, and at least one bank account has been added to his/her myMoney account.
Success Guarantee	User is able to conveniently view all transactions from all institutions in one display.
Main Success Flow	1. User selects View All Transactions option. 2. System shows all transactions across all accounts on one display.
Exceptions	
Post-Conditions	
Priority	
Traces to Test Cases	

Table 22: Use Case 7 - View Transaction by Type

Action	**View Transaction by Type**
Case ID	07
Summary	User can view transactions by the following types: deposits, withdrawals, and transfers.

Table 23: Use Case 8 - Categorize Transaction

Action	**Categorize Transaction**
Case ID	08
Summary	User can select a category to represent a transaction.

Table 24: Use Case 9 - View Transactions by Category

Action	**View Transactions by Category**
Case ID	09
Summary	User can view transaction according to categories of spending.

7 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