COMP 354
Requirements for the project myMoney

Team PA-PK

April 5, 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

Table 2: Revision history

Version	Date	Changes
1.2	5th April 2018	added more detail to use cases
1.1	12th March 2018	fixed business rules
1.0	11th February 2018	completed requirements

Contents

1 Document Purpose **4**

2 Project description **4**

2.1 Problem Introduction 4

2.2 System Context 4

2.3 Project Scope 5

2.4 Business Goals 6

2.5 Domain Concepts 6

Domain Model 6

2.6 Actors 8

User 8

Bank 8

3 Requirements and Business Rules **8**

3.1 Non-Functional Requirements 8

3.2 Business Rules 14

3.3 Functional Requirements 14

4 User-Stories **15**

4.1 Description of File Format: Input 16

4.2 Description of File Format: Output 16

5 Glossary of Domain Concepts **17**

6 Use Cases **18**

6.1 Overview 18

7 Reference **32**

List of Figures

1 System Context 5

2 Domain Model 7

3 Use Case Diagram 18

List of Tables

1 Team . . . 1
2 Revision history . . . 1
3 Non-Functional Requirement 1 - Reliability and usability . . . 8
4 Non-Functional Requirement 2 - Simplicity and ease-of-use . . . 9
5 Non-Functional Requirement 3 - Performance . . . 9
6 Non-Functional Requirement 4 - Maintainability and testability . . . 10
7 Non-Functional Requirement 5 - Security . . . 10
8 Non-Functional Requirement 6 - Portability . . . 11
9 Non-Functional Requirement 7 - Scalability . . . 11
10 Non-Functional Requirement 8 - Data integrity . . . 12
11 Non-Functional Requirement 9 - Java . . . 12
12 Non-Functional Requirement 10 - Object-oriented design . . . 12
13 Non-Functional Requirement 11 - Model-view-controller architecture . . . 12
14 Non-Functional Requirement 12 - SQLite database . . . 13
15 Glossary of Domain Concepts . . . 17
16 Use Case 1 - Create User Account . . . 19
17 Use Case 2 - Delete User Account . . . 20
18 Use Case 3 - Add Bank Account to a User Account . . . 21
19 Use Case 4 - Remove Bank Account from a User Account . . . 22
20 Use Case 5 - View Transactions for Specific Bank Account . . . 23
21 Use Case 6 - View All Transactions from all Bank Accounts . . . 24
22 Use Case 7 - Update User Account . . . 25
23 Use Case 8 - Sort Transactions by Any Attribute . . . 26
24 Use Case 9 - Categorize Transaction . . . 27
25 Use Case 10 - Filter Transactions by Date Range . . . 28
26 Use Case 11 - Filter Transactions by Category . . . 29
27 Use Case 12 - Export Statement as a CSV file . . . 30
28 Use Case 13 - Email Statement . . . 31

1 Document Purpose

The purpose of this document is to define requirements for the desktop application myMoney. This document may thus be used 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 the 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 for this document are the stakeholders of the project and the development team of the system, as well the project testers for fine-tuning their testing strategy.

2 Project description

2.1 Problem 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. Unfortunately, the high quality programs in this domain also tend to have complex and highly extensive accounting capabilities, and setting them up as well as learning how to use them requires investing some number of hours that often scare off casual users. This lack of appropriate software for 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 may 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 establishes a connection to the banks, passing along the user information. It will receive the data associated with the bank account(s) of the user, which it will then display to the user. Section 3.3 gives a more detailed view of all of the functions that the system will be able to perform.

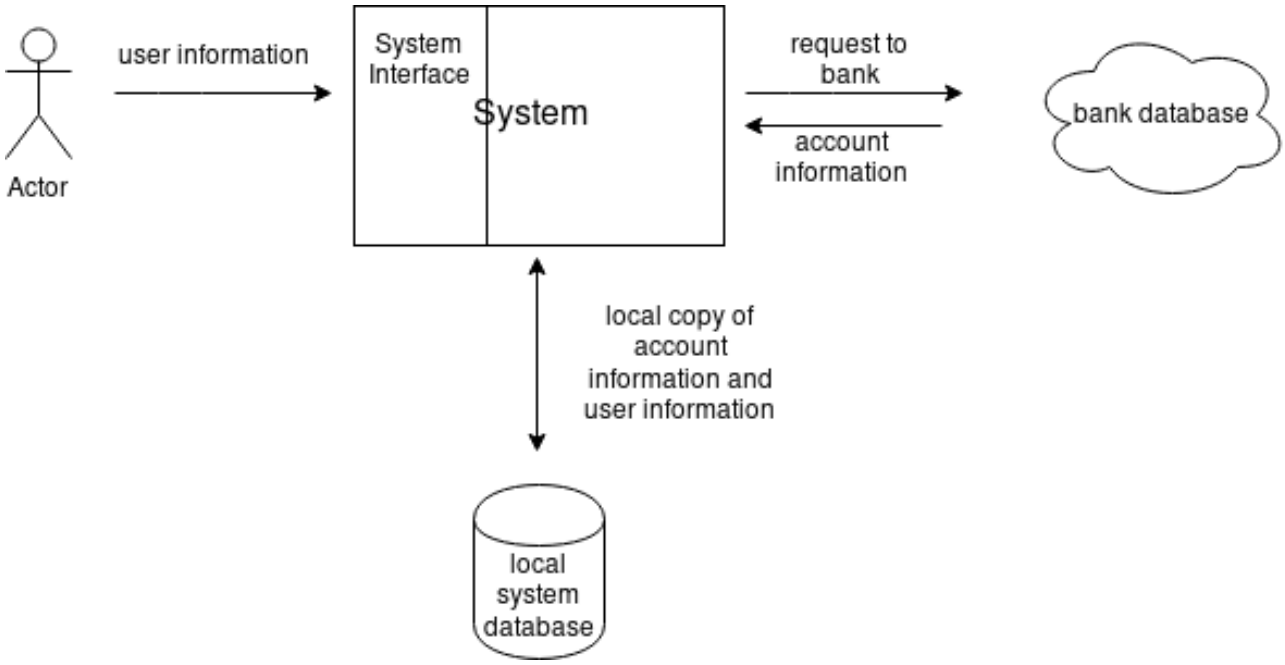

Figure 1: System Context

2.3 Project 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 user may create a personal account and link their bank accounts to easily access them through myMoney. They may check the transactions of a specific bank account or all of their bank accounts at once. They may also sort their 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 or technologically 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 lightning-fast applications for frequent yet momentary usage. They want to monitor their cash spending and make long-term financial decisions based on clear, understandable data that they can access and modify rapidly.

- **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 accommodated 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

In this section we 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.

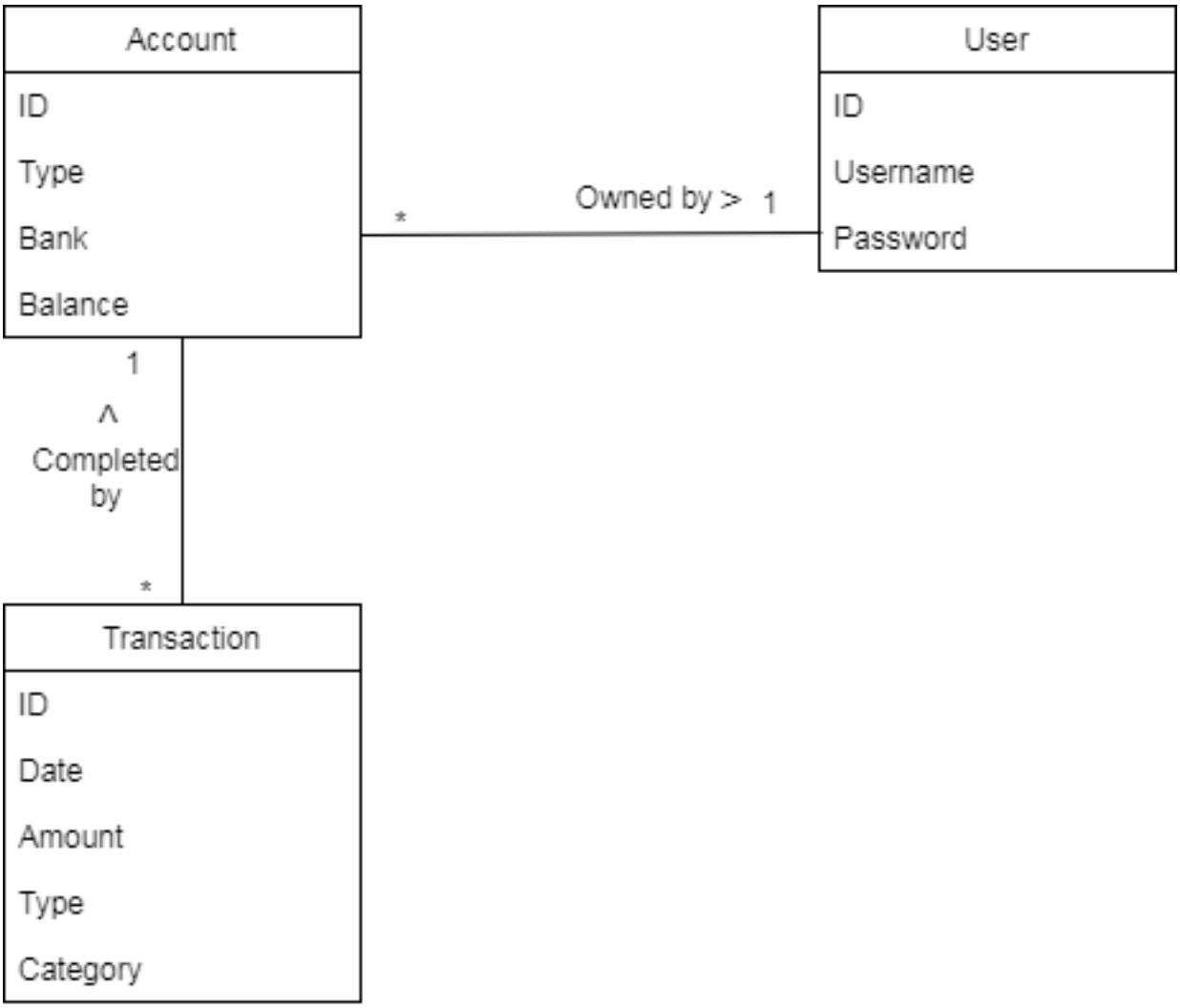

Figure 2: Domain Model

2.6 Actors

User

Our main actor is the user. All use cases are triggered by the user, as 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 middleman between the user demanding information in a specific format, and the banks holding that information in an inconvenient format.

3 Requirements and Business Rules

3.1 Non-Functional Requirements

Table 3: 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 bank 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 4: 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 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 5: 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 6: 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 reduces 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 7: 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 8: 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 9: 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 10: 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 11: 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 12: 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 13: 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 14: 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.

3.2 Business Rules

These business rules are the limits placed on the functions described in 3.3.

- **BR1: Length on account parameters set by the user**
  - The following attributes on the user account must be between 4 and 16 characters long: username, password, first name, and last name.
  - *Reasoning:* We do not want usernames and passwords to be too short for security concerns, nor do we want them to be too long, for data size concerns. We also require a first name and last name for security reasons, as our app accesses sensitive bank data.

- **BR2:** User Account authentication
  - A user must be correctly authenticated before he may view any data related to a user account, such as user account username and other paramaters, bank accounts associated with said user account, or bank account transactions. To be properly authenticated (also referred to as 'login in'), a user must provide the correct username and password.
  - *Reasoning:* Privacy is a huge concern for users when dealing with their financial information. Hence, we want to ensure that none of their data may be accessed unless the user has authenticated access to the account in question.

- **BR3:** User Account authorisation
  - A user may only view or edit information related to the user account he has logged into. He may not view or edit bank accounts or user account information of other users.
  - *Reasoning:* For privacy concerns once again, we want to ensure that a user may only modify that which he can access through authentication. We clarify the separation between authentication (actually logging in) and authorisation (what level of authority comes with being logged in?).

- **BR4:** Bank authorisation
  - A user may only add bank accounts once he has logged in, and he may only add account to which he has authorised access, given to him by his bank. He verifies this by providing the bank account ID of the bank account he wishes to access.
  - *Reasoning:* We want to ensure we follow the authentication requirements of the banks, once again for privacy reasons, but also to ensure a good-faith relationship with the banks is kept.

3.3 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 User-Stories

User-Story 1

As a user, I should be able to create, manage and delete my user account. I should have the options to view and update personal information to facilitate more efficient use of the application and personalise my experience.

Acceptance criteria:

- It should possible to sign up with a username and a password.
- It should be possible to update personal information, including the name, email address, phone number and password.
- The application should provide quality user experience when accessing and updating account information

User-Story 2

As a user, I should have ability to manage my bank account information. The application should allow me to connect my bank accounts, remove bank accounts and view all the information associated with the bank account.

Acceptance criteria:

- It should be possible to connect a bank account to the application.
- It should be possible to remove a bank account from the application.
- The application should display bank account information including:
  - Bank account ID
  - Bank name
  - Account type
  - Account balance
  - History of transactions
- The application should provide a way to see bank account statements for a specific period.
- It should be possible to sort the list of all bank account transactions by any of its attributes, such as the type of the transaction and the amount.

User-Story 3

As a user, I should have the ability to group my bank account transactions into categories, to be able to manage my finances more efficiently.

Acceptance criteria:

- The application should allow to assign a transaction to a category, such as monthly payments, groceries, leisure, etc.
- I should be able to view transactions of a particular category.

4.1 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.

4.2 Description of File Format: Output

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

5 Glossary of Domain Concepts

Table 15: 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.

6 Use Cases

6.1 Overview

Use cases 1 through 4 deal with the user manipulating his user and bank accounts. Use cases 5 through 8 deal with the user viewing the data in different formats. Use case 9 deals with the user manipulating the transactions categories.

NB: Note that the alternate flows are described in the same order as the exception. That is, for example, alternate flow 2 specifies the flow of the application if exception 2 is met.

Below is figure 3 which represents our use case diagram. Following this diagram, we list our uses cases

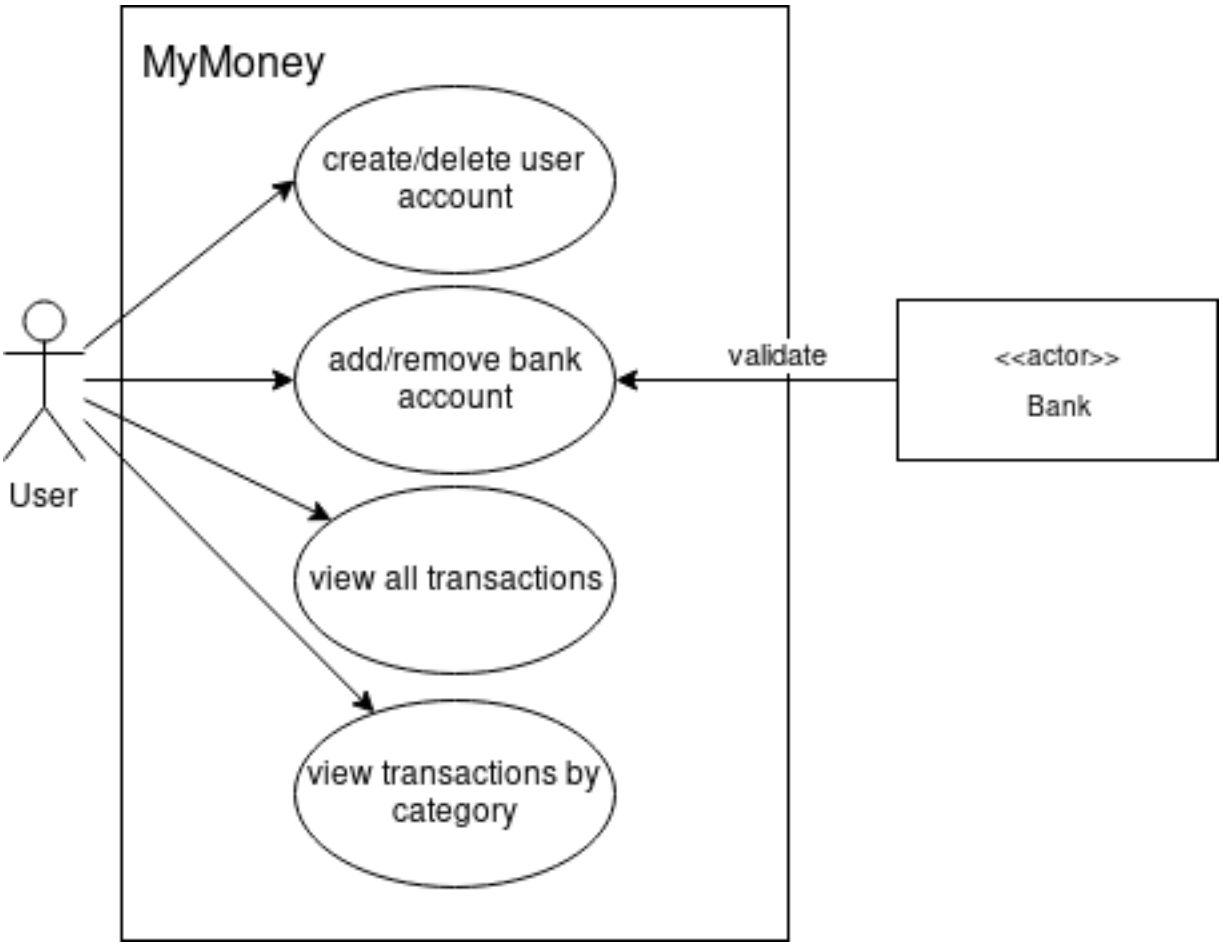

Figure 3: Use Case Diagram

Table 16: Use Case 1 - Create User Account

Action	**Create User Account**
Case ID	01
Summary	User gives information about a new user account to the system, which validates it then 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 and a comprehensible display of the account requirements. 2. Company: Wants user interests to be fulfilled, validation of input, communication with the local account database as well as failure tolerance when dealing with database manipulation.
Pre-Conditions	User has launched the application.
Success Guarantee	Account successfully saved in local account database, with account details as specified by the user.
Main Success Flow	1. User clicks on the sign-up menu, enters a username, first name, last name, email, phone number, and address and password. 2. System validates user account details, then creates new account. 3. System moves to login screen.
Exceptions	1. Another account with the same username already exists. 2. User specified is not created if the business rules for account creation are not met.
Alternate Flows	1. The System will notify the user the his username is already used by an existing user account, and return to the account creation menu. 2. The System will notify the user of unmet account requirement, then return to the account creation menu.
Priority	High
Traces to Test Cases	

Table 17: Use Case 2 - Delete User Account

Action	**Delete User Account**
Case ID	02
Summary	User deletes his user account, removing all bank accounts and information associated with that account.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants easy navigation and secure account deletion, no risk of accidental deletion. 2. Company: Wants user interests to be fulfilled and clean deletion of account and associated data in the database.
Pre-Conditions	User has logged in the user account he wants to delete.
Success Guarantee	user account is successfully deleted, all associated bank information is deleted, and user is returned to the home menu.
Main Success Flow	1. User enters the user account details view, then selects 'delete account'. 2. System brings up a confirmation menu to ensure that this selection was not accidentally entered. 3. User confirms his choice. 4. System successfully deletes the user account as well as all associated bank account and transactions in the local database, then brings the user back to the login/create user menu.
Exceptions	1. User enters incorrect password and is denied ability to delete user account.
Alternate Flows	1. Incorrect password error message will be displayed, the user account will not be deleted, and the user will be returned to the user account details view.
Priority	Medium
Traces to Test Cases	

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 of bank account requirements. 2. Company: Wants user and bank 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 remote database problems. 3. Bank: Wants to satisfy its own 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 account list 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 ID in the text field for bank account addition. 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 the account list is updated with this new account.
Exceptions	1. An invalid account ID was provided. 2. Account has already been added to the user account. 3. Account was not validated by the bank (could not find account, invalid ID on their side of the validation, etc.)
Alternate Flows	1. The account ID is not sent to the bank servers for validation, instead an error will be displayed requesting a valid bank account ID then the system returns to the account list menu. 2. the account ID is validated but found to have been already associated with this user account; this error is displayed by the system, which then returns to the account list menu. 3. the accound ID is validated and queried to the bank, which

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 removes a bank account associated with his user account.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants easy navigation and secure account deletion. 2. Company: Wants user interests to be fulfilled, wants to ensure clean deletion from the database.
Pre-Conditions	User has logged in the user account whose active association with a bank account is the one the user wants to delete.
Success Guarantee	Bank account is successfully removed from that user account, all associated bank information is deleted from the local database, and user is returned to the account home menu.
Main Success Flow	1. User selects the account he wants to remove, then selects *remove selected account*. 2. System successfully deletes the bank account and its associated transactions in the local database, then updates the account list to display this change.
Exceptions	1. The user clicks *remove selected account* without having selected an account.
Alternate Flows	1. No account is deleted and the system ignores the call, staying in the accounts list menu.
Priority	High
Traces to Test Cases	

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 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 transactions associated with a specific bank account. 2. Company: Wants user interests to be fulfilled.
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 transactions of specific bank account.
Main Success Flow	1. User selects specific bank account from list of all accounts. 2. System enters the detailed view for that bank account, displaying all transactions for that account.
Exceptions	1. User has no transactions for selected bank account.
Alternate Flows	System displays the detailed view of the bank account nonetheless, but the list of transaction is empty.
Priority	High
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.
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 in one unified list.
Exceptions	1. User has not added any account. 2. User has no transactions to record.
Alternate Flows	1. System displays an empty list of transactions. 2. System displays an empty list of transactions.
Priority	High
Traces to Test Cases	

Table 22: Use Case 7 - Update User Account

Action	**Update User Account**
Case ID	07
Summary	User can make changes and update personal information
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to user account attributes to keep up-to-date records on information such as address and email as well as changing user account password. 2. Company: Wants user interests to be fulfilled and ensure validation of changed user account attributes.
Pre-Conditions	User has created and logged in a user account.
Success Guarantee	User can easily update changes in personal information to guarantee accurate personal records
Main Success Flow	1. User selects *Your profile*. 2. System shows current user account information. 3. User selects the piece of information that requires updating. 4. User enters new information. 5. System updates and saves the new information.
Exceptions	1. Passwords do not match, user information is not updated 2. First name, last name, or password is left empty, user information is not updated. 3. email is an invalid email address.
Alternate Flows	1. Error message is displayed prompting user that his passwords do not match and user account is not updated. 2. Error message is displayed prompting user that he is missing required information and user account is not updated. 3. Error message is displayed prompting user that his email address is invalid nd user account is not updated.
Priority	Medium
Traces to Test Cases	

Table 23: Use Case 8 - Sort Transactions by Any Attribute

Action	**Sort Transactions by Any Attribute**
Case ID	08
Summary	User can sort transactions by any attribute (date, transaction amount, category, and transaction type.)
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to view transactions sorted by any one attribute. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	User has created and logged in a user account.
Success Guarantee	User can sort transactions by selecting one attribute. The default sorting is ascending order by alphabet or value of number. Double-clicking will sort in descending order.
Main Success Flow	1. User enters a transaction list view by choosing to view all bank account transactions or a detailed bank account view. 2. System shows a transaction list. 3. User selects to view sorted transaction by clicking any one attribute. 4. System sorts transactions by the selected attribute in ascending order.
Exceptions	1. User has no transactions in his account.
Alternate Flows	1. The system will display an empty transaction list.
Priority	High
Traces to Test Cases	

Table 24: Use Case 9 - Categorize Transaction

Action	**Categorize Transaction**
Case ID	09
Summary	User can select a category to represent a transaction.
Description	This use case describes how a user can categorize their transactions with specific labels in order to sort their spending and better manage where most of their money is being spent
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to label each transaction as a specific type of spending, such as rent, bills, leisure, etc., to better manage and track spending habits. 2. Company: Wants user interests to be fulfilled. Wants to facilitate user's ability to budget and manage money through simple categorization of spending.
Pre-Conditions	User has created and logged in a user account, added at least one bank account, and has made at least one type of transaction.
Success Guarantee	User can quickly and efficiently categorize each transaction.
Main Success Flow	1. User enters a transaction list view by choosing to view all bank account transactions or a detailed bank account view. 2. User selects the desired transaction to categorize. 3. User selects the categorize option. 4. System displays the categories in a drop down menu. 5. User selects preferred category to represent the current transaction, or types in his own category. 6. System changes the category attribute on the chosen transaction to the category that the user selected or wrote in.
Exceptions	1. the user writes in a category name of exceeding length (exceeding as defined by the business rules.)
Alternate Flows	1. The system rejects the call and leaves the category unchanged.
Priority	Medium
Traces to Test Cases	

Table 25: Use Case 10 - Filter Transactions by Date Range

Action	**Filter Transactions by Date Range**
Case ID	10
Summary	User can view all transactions between two selected dates.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to view transactions between two selected dates, to optimize spending habits for select durations. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	User has created and logged in a user account.
Success Guarantee	User can view all transactions from select bank accounts, or all bank accounts between two specified dates.
Main Success Flow	1. User decides to view all bank accounts or selects a specific bank account. 2. System shows transactions. 3. User selects desired dates to view transactions between the two dates. 4. System shows all transactions between the two specified dates.
Exceptions	1. User selects impossible range of dates.
Alternate Flows	1. no transactions are displayed.
Priority	Low
Traces to Test Cases	

Table 26: Use Case 11 - Filter Transactions by Category

Action	**Filter Transactions by Category**
Case ID	11
Summary	User can choose a category to filter by, which will result in all transactions which are not in that category to be removed from the view
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to view only the transactions in a specific category. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	User has created and logged in an account.
Success Guarantee	Only the transactions in the specified category will be displayed.
Main Success Flow	1. User decides to view all bank accounts or selects a specific bank account. 2. System shows transactions. 3. User clicks in the text field for searching by category, and writes in the category he wishes to filer by. 4. System updates the view to show only the transactions with the specified category.
Exceptions	1. User enters a non-existent category (a category by which no transaction is categorised).
Alternate Flows	1. No transactions are displayed.
Priority	High
Traces to Test Cases	

Table 27: Use Case 12 - Export Statement as a CSV file

Action	**Export Statement as a CSV file**
Case ID	12
Summary	Export of a CSV statement consisting of all the transactions currently displayed in the application window.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to save a CSV file of his transactions somewhere on his machine, for easy external use of his transaction history. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	The user has created a user account, logged in, connected at least one bank account and has reached a transactions view either through *view all transactions* view, either through *account details* view.
Success Guarantee	A CSV file has been successfully generated from the transactions currently displayed, and was saved where the user asked for it to be saved.
Main Success Flow	1. User chooses to view all bank account transactions or a detailed bank account view. 2. System shows transactions. 3. User clicks on the button to export to CSV. 4. System displays a menu asking where the user wants this statement to be saved. 5. User selects the directory wherein he wishes to export the statement. 6. System generates the statement and saves it in the desired folder of the user's choosing. It may be blank if the user has no transaction displayed in his view.
Exceptions	1. User enters a non-existent path. 2. User enters an unauthorised path. 3. User enters a valid path, but a file with the same name as the statement to be generated already exists.
Alternate Flows	1. The save button is greyed out, preventing the user from selecting that path and thus requiring a valid path to be entered to move on. 2. The save button is greyed out, preventing the user from selecting that path and thus requiring a valid path to be entered to move on. 3. A confirmation box should ask the user if he wants to overwrite a file if a file already exists in the specified location.
Priority	Low
Traces to Test Cases	

Table 28: Use Case 13 - Email Statement

Action	**Email Statement**
Case ID	13
Summary	User wants to have a CSV statement consisting of all the transactions currently in the application window sent to his email.
Scope	money and budget management application
Level	user-goal
Actors	**User**
Stakeholders and Interests	1. User: Wants to receive a CSV statement of all his transactions, for easy exportation of his transaction history. 2. Company: Wants user interests to be fulfilled.
Pre-Conditions	A user created a user accountl, logged in, connected at least one bank account and has reached a transactions view either through *view all transactions* view, either through *account details* view.
Success Guarantee	A CSV file has been successfully generated from the transactions currently displayed, and was saved where the user asked for it to be saved.
Main Success Flow	1. User chooses to view all bank account transactions or a detailed bank account view. 2. System shows transactions. 3. User clicks on the button to email CSV. 4. System generates the statement and sends it to the user's email.
Exceptions	1. User hasn't set his email in his user account details.
Alternate Flows	1. the System notifies the user that his email address is empty and doesn't send the CSV file.
Priority	Low
Traces to Test Cases	

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