Personal Financial Manager Design Specification

D. Weissenberger, J. Doan, M. Maibach, Z. Smith

University of Maryland Global College – CMSC 495 – 24416

Sunday, April 12, 2020

**PREFACE**

This design specification outlines the plan of attack for future development of the Personal Financial Manager application throughout CMSC 495, class number 24416. PMP development and planning will be carried out by members of Group C.

**Personal Financial Manager Design Specification**

**TABLE OF CONTENTS**

[SECTION 1. DESIGN CONSIDERATIONS](#_heading=h.1fob9te) **5**

[1.1 ASSUMPTIONS AND DEPENDENCIES](#_heading=h.3znysh7) 5

[The following assumptions are made for this project:](#_heading=h.v0ggnl45nnnj) 5

[1.2 GENERAL CONSTRAINTS](#_heading=h.t44uwaas0l2e) 5

[1.3 GOALS AND GUIDELINES](#_heading=h.tyjcwt) 5

[1.4 DEVELOPMENT METHODS](#_heading=h.3dy6vkm) 6

[SECTION 2. SYSTEM ARCHITECTURE](#_heading=h.4d34og8) **7**

[2.1 UML DIAGRAM](#_heading=h.cw7ul41n9y0e) 7

[SECTION 3. POLICIES AND TACTICS](#_heading=h.2s8eyo1) **8**

[3.1 GUI COMPONENT](#_heading=h.bnivt4jevd46) 8

[3.1.2 MySQL](#_heading=h.oyghwc5cyw07) 9

[3.1.3 Sqlite](#_heading=h.eaq2s75a169h) 9

[SECTION 4. DETAILED SYSTEM DESIGN](#_heading=h.17dp8vu) **10**

[4.1 GUI COMPONENT](#_heading=h.3rdcrjn) 10

[*4.1.1 Classification*](#_heading=h.26in1rg) *10*

[*4.1.2 Definition*](#_heading=h.lnxbz9) *10*

[*Provides the user with a means to access the application, record transactions, and display reports*](#_heading=h.35nkun2) *10*

[4.1.3 Responsibilities](#_heading=h.nru525yx2c9c) 10

[*4.1.4 Constraints*](#_heading=h.1ksv4uv) *10*

[*4.1.5 Composition*](#_heading=h.44sinio) *10*

[*4.1.5 Uses and Interactions*](#_heading=h.2jxsxqh) *10*

[*4.1.7 Resources*](#_heading=h.z337ya) *10*

[*4.1.8 Processing*](#_heading=h.3j2qqm3) *11*

[*4.1.9 Interface and Exports*](#_heading=h.1y810tw) *11*

[*4.1.10 Detailed Subsystem Design*](#_heading=h.4i7ojhp) *11*

[4.2 DATABASE COMPONENT](#_heading=h.2xcytpi) 12

[*4.2.1 Classification*](#_heading=h.1ci93xb) *12*

[*4.2.2 Definition*](#_heading=h.3whwml4) *12*

[*4.2.3 Responsibilities*](#_heading=h.2bn6wsx) *12*

[*4.2.4 Constraints*](#_heading=h.qsh70q) *12*

[*4.2.5 Composition*](#_heading=h.3as4poj) *13*

[*4.2.5 Uses and Interactions*](#_heading=h.1pxezwc) *13*

[*4.2.7 Resources*](#_heading=h.49x2ik5) *14*

[*4.2.8 Processing*](#_heading=h.2p2csry) *14*

[*4.2.9 Interface and Exports*](#_heading=h.147n2zr) *14*

[*4.2.10 Detailed Subsystem Design*](#_heading=h.3o7alnk) *15*

[SECTION 5. REFERENCES](#_heading=h.23ckvvd) **16**

# SECTION 1. DESIGN CONSIDERATIONS

## 1.1 ASSUMPTIONS AND DEPENDENCIES

## The following assumptions are made for this project:

* All team members are available for their required tasks
* All team members have the required skill set to complete their required tasks
* The timeline created is realistic and achievable

The project will follow a Finish to Start approach, where the next task will not start until the previous task has completed. This follows the guidelines laid out for the project and the general structure of the course. Each team member is dependent upon the other members to complete their tasks in the given time.

## 1.2 GENERAL CONSTRAINTS

The follow general constraints apply to this project:

* The application must perform within the specifications given
* No member’s tasks should consume more time than is available to complete the task
* The final project must be completed in accordance with the course deadlines
* The project must provide all necessary documentation and results as prescribed by the course outline

## 1.3 GOALS AND GUIDELINES

We seek to achieve the following goals:

* Complete out project on time with all required or specified functionality
* Produce a complete project, including all steps prescribed in a timely manner
* Foster teamwork and involvement

## 1.4 DEVELOPMENT METHODS

The PFM project team has followed the steps of the Software Development Lifecycle, and this document was prepared during the design phase (Stackify, 2020). The group leader must approve all commits to our PFM project repository, but live discussions are the preference.

We will follow the Google Java Style Guide and check to enforce compliance before testing (Google, n.d.).

Database operations will be atomic as enforced and the database itself will be encoded in utf-8. The structure has not yet been checked for normal form, but optimization may be needed after testing.

# SECTION 2. SYSTEM ARCHITECTURE

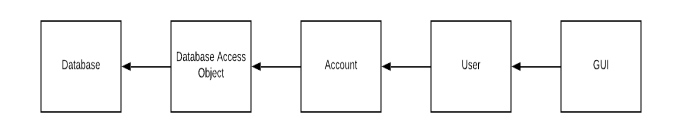
The PFM design process is subdivided by the Graphical User Interface [GUI] component, and the database component. The database component is included to address the requirement that a user must be able to track transactions even after restarting the program. The gui component is needed so the user can interact with a simple interface that has intuitive behavior.

The project team started by examining the outlines of some example receipts, and exploring different transaction scenarios. Payment methods, for example, include both card and cash, but this implies a further relationship between the user and their card.

Program requirements also include that the user can enter receipt information, or account transactions such as deposits and withdrawals. This creates a distinction between account- and non-account-transactions.

## 2.1 UML DIAGRAM

Here is the simplified UML class diagram. Note that the Database is not a class, but rather the remote storage database that is included in the Database Component.



# SECTION 3. POLICIES AND TACTICS

The PFM GUI and database will be packaged together with their dependencies and User Guide, so that once the user has a running copy of Java, they should be able to execute the GUI, which will provide the interface for all interactions with the database. Before viewing or inserting any data, the user will need to log in with a known username and password, which is also a database interaction. The ability to manage login credentials is not included but it is assumed that that information will be given to the user. Sqlite includes database authentication, which can be invoked before launching the GUI.

## 3.1 GUI COMPONENT

The GUI for the application will be written in JavaFX and will include components that allow the user to login to the application, record transactions, and produce reports based on recorded data. The project team decided to use an application called Scene Builder to design and create the GUI (Oracle, n.d.).

**3.2 DATABASE COMPONENT**

The design process for the database will go as follows:

1. Create table and row definitions in prepared \_.sql files, using properly formed Sqlite commands, before modifying the database itself, and run this by the project team.
2. Load the database contents and schema. This is a good time to insert some test data and perform some queries.
3. Based on agreed-upon GUI operations, refine the query statements, using the agreed-upon Sqlite Java library. Choice of library will depend upon both the user requirements and the design of the GUI component.

The project team discussed several possibilities for database solutions that would fit the use case, meet program requirements, foster collaboration, and avoid extending the timeline.

*3.1.1 MongoDB*

Was explored initially due to its availability via the MongoDb Atlas cloud solution. This was before the team explored the requirements of such a remote connection.

### *3.1.2 MySQL*

Is a simple option with which the team has had experience throughout their educational careers at UMGC. This would require that the user install a separate program that has a limited feature set.

### *3.1.3 Sqlite*

We ultimately decided on this option because it is described as a single \_.db file that is minimized for client usage. Further research is ongoing about the security implications, which so far include file confidentiality and integrity.

# SECTION 4. DETAILED SYSTEM DESIGN

## 4.1 GUI COMPONENT

### *4.1.1 Classification*

This component is a graphical user interface.

### *4.1.2 Definition*

### Provides the user with a means to access the application, record transactions, and display reports

### *4.1.3 Responsibilities*

The GUI is the main portal into the application. Without it, the application cannot be used

### *4.1.4 Constraints*

The user must have installed a recent version of Java on their computer before running the PFM.

### *4.1.5 Composition*

The GUI will be composed of standard components found in the JavaFX library

### *4.1.5 Uses and Interactions*

The GUI is used as the sole user interface. It will be used to display all information to the user as well as allow the user to create and modify existing records of transactions. The GUI will also need to interact with the database component of the application by sending and receiving records to and from the database component

### *4.1.7 Resources*

### *4.1.8 Processing*

The PFM is written in Java, which runs in its own virtual machine, and includes several dependencies which may affect performance and thus the user experience. Testing can be done in different environments to draft a list of final hardware and software requirements.

### *4.1.9 Interface and Exports*

The GUI contains classes to represent *Accounts, Users, Receipts*, etc. Objects of these classes are allocated and populated on demand. *Users*, for example, is filled immediately after logging in. These objects are used to display the user’s requested information, allowing better format control.

Buttons on the GUI represent user operations: *new receipt, new withdrawal,* and *new deposit* correspond to event listeners. These listeners invoke a corresponding method for each operation, which will interact with the database access object [DAO] to modify the user’s records.

### *4.1.10 Detailed Subsystem Design*

The classes known so far, and their associated methods, parameters, and attributes:

***Authentication Page***

*OnLoginButtonClick(void)*

*Two input boxes*

***DAO***

*ConnectToDB(user, pass)*

*QueryBuilder(params, query)*

***GetData*** *- There are a number of these methods, many are similar and only vary in the initial parameters it takes.*

*user\_id*

*user acc number*

***Main Page***

*FetchRecentTransac(useraccnum)*

***New Recepit***

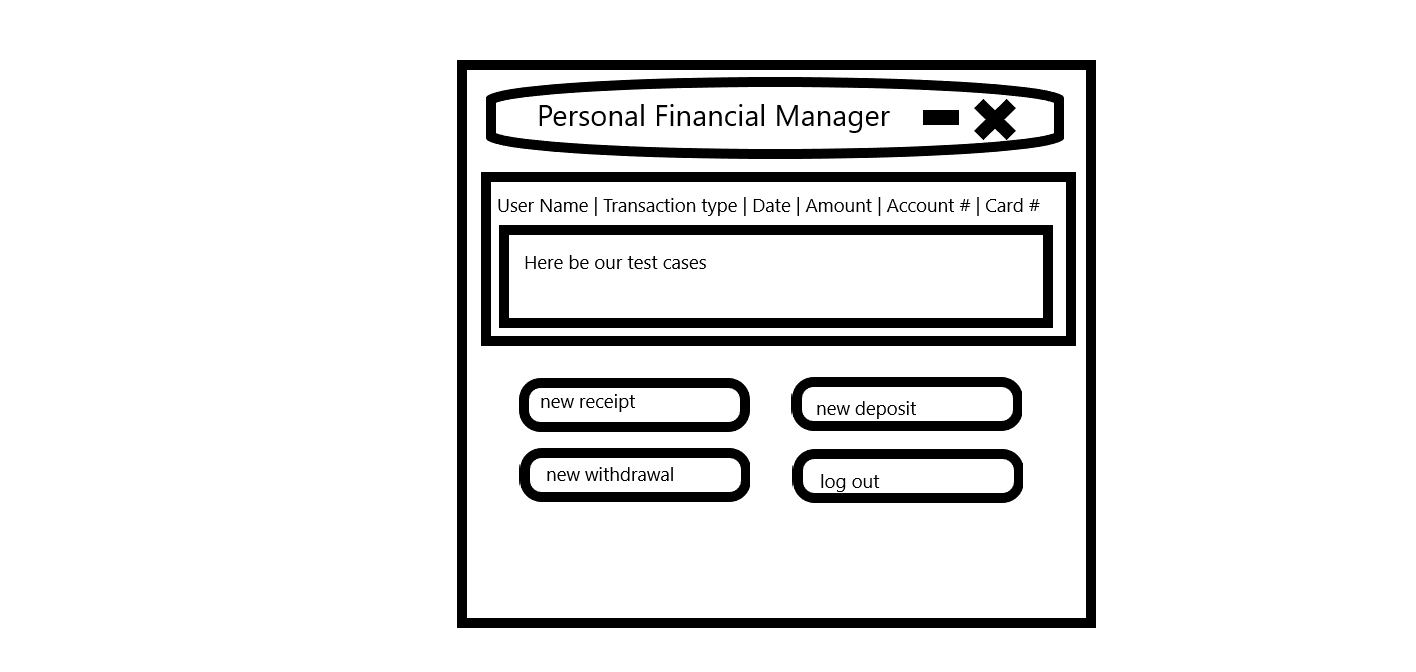
*OnNewRecepitClick(void)*

*OnNewDepositClick(void)*

*OnNewWithdrawClick(void)*

*LogOut(void)*

A new account function may be preferred instead of creating a new one after a deposit or withdrawal. Apart from that, here is an estimate of what the operations the GUI offers:

****

## 4.2 DATABASE COMPONENT

### *4.2.1 Classification*

This component is a relational, client-side database.

### *4.2.2 Definition*

Long-term and secure storage of user/account information and transactions.

### *4.2.3 Responsibilities*

The PFM depends on this database component to conduct most, if not all operations. An outage will render the whole application unusable.

### *4.2.4 Constraints*

Each table has a unique id, and most have foreign keys referencing other tables. When entering a new transaction, the PFM cannot prompt for a foreign key, meaning some matching must be done with cards.account number, receipts.card number. These fields will need to be non-nullable.

A receipt may be paid with card or cash, and a user may enter a receipt even before populating account data. This means a case exists that card\_num will not match, so the operation must be allowed. A dialog box should warn the user, however, if the card number on the receipt does not match and therefore will not be tracked. Another possibility is for PFM’s internal logic to insert the account row implicitly and hint at the user to make corrections. By contrast, an error page could say “try again or quit” when trying to add a card with a non-matching account number.

Receipts must have a user id, but this is easily enforced using the GUI interface.

In order to perform deposits and withdrawals, an account must exist for the current user, otherwise we can grey out or hide these options. As mentioned above, however, we may decide to have an account be implicitly created **afterwards.**

There is a potential many-to-many relationship between users and accounts, represented as an intermediate table containing foreign keys to both.

Accounts.balance is a non-nullable field because deposits.amount and withdrawals.amount are non-nullable.

### *4.2.5 Composition*

When first starting PFM, the only populated field is Users. Only by inserting new rows in *receipts*, *withdrawals*, or *deposits* can the others be populated. The tables: *accounts*, *deposits* and *withdrawals* all have minimal columns (id’s and balance / amount) laid out so far since they currently exist to provide a link between users and cards. The table *users* contains the data needed to both log in to the program, such as username, password, and last\_update (which could be used for a salt on the password hash). The table *receipts* is the primary way to reflect transactions, and contains such fields as user\_id, card\_num, card\_id (possibly redundant), total, sub\_total, sales\_tax, discount, cash\_paid, and date\_purchase. This allows for adding receipts using different types of transactions.

### *4.2.5 Uses and Interactions*

While the input fields will mirror database columns; behavior and data should be hidden from the user, as well as from outside observers.

### *4.2.7 Resources*

Both the GUI and the database components are crucial for the operation of the PFM, but are designed as standalone. They should need no additional resources besides such dependencies as the Sqlite Java interface.

### *4.2.8 Processing*

The PFM database exists only as a file while the application is not running, and is considered serverless. Database overhead exists within the application, and testing can be done to isolate the effects.

*“****Classic Serverless:*** *The database engine runs within the same process, thread, and address space as the application. There is no message passing or network activity” (Sqlite.org, 2018).*

### *4.2.9 Interface and Exports*

The trusted library sqlite-jdbc.jar is the project team’s top contender to interact with PFM’s GUI component. There is also a forked version called sqlite-jdbc-crypt.jar. Thus far until this phase of design, research has been done independently and more collaboration is required in order to outline the interface. A Database Access Object [DAO] will likely be created on the GUI side, which will be invoked by the methods representing the user’s available operations eg, new receipt, new deposit, new withdrawal.

Currency types are not reflected in the database, but floats are used instead. Users.password is stored as a salted hash using last\_update, which is type *datetime*. IDs are sequential integers. Account numbers, routing numbers, card numbers are all integers. User name is stored as an unformatted varchar or string; however some formatted strings may be used such as email addresses.

### *4.2.10 Detailed Subsystem Design*

*pfm-db-3.dia Database flowchart last modified 04/12/2020*

# SECTION 5. REFERENCES

Google (n.d.). *Google Java Style Guide.* Retrieved from: <https://google.github.io/styleguide/javaguide.html>.

Oracle (n.d.). *JavaFX Scene Builder.* Retrieved from: <https://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html>.

Sqlite.org (02 April, 2018). *Sqlite is serverless.* Retrieved from: <https://www.sqlite.org/serverless.html>.

Stackify (08 April, 2020). *What is SDLC? Understand the Software Development Lifecycle.* Retrieved from: <https://stackify.com/what-is-sdlc/>.