Implementation Document

CSCI-310 Software Engineering

Professor: **Nenad Medvidović**

TA: **Sarah Cooney**

Fall 2017

$a₦i₮¥

Ba$iL (TEAM 15)

Tri Nguyen 1410884747

Ang Li 3340862395

Utsav Thapa 9717428024

Andre Takhmazyan 8764629970

Kevin Nguyen 215820474

1. Preface

This document articulates the implementation process of the android application *Sanity*. This includes any architectural changes and any changes to the detailed design and requirements documents as our project moved from paper to a real system. The intended readership of this document is us, the software engineers who developed the system as well as the stakeholders who will be reviewing the system requirements in comparison with the agreed upon requirements. It is also intended for the future software engineers who will be maintaining the system.

2. Introduction

The rest of this document will detail the specific choices that were taken during the implementation phase. Specifically, we will be discussing any intended and unintended divergences from the requirement and design documents and how those changes altered our system architecture. Furthermore, we will discuss what was upheld in the requirement and design documents and how those pre-implementation decisions helped streamline the development of *Sanity*.

The android application *Sanity* was implemented with minimum SDK level 7.1.1 (Nougat). For a database the application state was saved via SQLite. Version control of the project was facilitated through Git and GitHub. System security in the domain of protecting the user's information was accomplished by using a cryptographic library, jBCrypt, for secure salted password hashing of the user's password and security question answers. For visualizing budget analytics, we used MPAndroidChart to display various types of graphs.

3. Architectural Change

For the most part, our implementation conformed to the three tiered state-logic-display pattern that we defined in our detailed design document. We implemented the display, logic, and database as outlined in our previous design. Within each, we changed the following details.

Display

Change

We found that the display, in some rare instances like authentication, actually made calls to the database directly. This action is more in line with the MVC architecture where the view will communicate directly with the model to update certain values.

Rationale

In some instances this design decision was advantageous to us as it allowed us to limit the dependence of several components upon each other. However, having only one component of our system conform to MVC, we felt, did not warrant completely overhauling our entire system to conform to the MVC architecture. For all intensive purposes, our three tier architecture remains the principal design choice because it limits the view's interaction with the database, which is critical for a fintech product.

Logic

Change

We decided to implement the budget manager as a singleton class. This means that at any given time during system execution there is no more than one instance of the budget manager, and our design protects against this. The clonable method was overridden to throw a CloneNotSupportedException() in the instance that a developer tries to clone the budget manager instance. In addition, the budget manager class has a private member for its single instance that can be accessed through the method, getInstance(), which conforms to the singleton design pattern by either initializing the manager if it does not yet exist or returns the existing instance stored in the private member.

Rationale

Though not explicitly elaborated upon during the detailed design phase, this evolutionary change to the budget manager helped us to keep our resource passing efficient and gave us confidence that we were always passing the right resource.

The logic controllers, we found, functioned more as conduits between display and the state (database). Most of the components that made up the logic controllers acted as messengers, transforming data that is collected from the user and passing it to the database as well as providing data from the database to the display for updating the views. The most vital controller was the budget manager, which had access to all of the budget categories for the user.

Database

Change

The addition of more extensive user authentication functionality, like forgot password and reset password functionality, required more methods in the database class [e.g. updatePassword(), getSecurityQuestion(), checkSecurityAnswer()].

Rationale

What we found when developing *Sanity* was that the state (database) played a much larger role in the system than we had previously anticipated. By far, it was the system component that had the most complexity in terms of coupling with other components. Thus, the rationale behind our changes was to support the design changes in the other tiers of our architecture. In the end, we noticed that any measurable change to the view or logic required a parallel change in the database to reflect the change or addition of functionality.

4. Detailed Design Change

Authentication Changes

Change

We added several new displays to our application to increase our functionality. We allow new users to sign up using a SignUpScreen. When the user signs up, they select a security question.

Rationale

This question is needed to allow users to reset their password in the ResetScreen. This lead to changes in the Database class to allow creating new users and changing the passwords. We created new Database methods such as addUser() and updatePassword() as well as methods that service the security question functionality such as getSecurityQuestion() to return the user's security question and checkSecurityAnswer() to check the user's answer to the security question.

Main Menu Changes

Change

In the MainMenu, we now allow users to change the overall budget period and notification settings. The Transactions class was changed to implement the Comparable interface so that transactions could be sorted by date.

Rationale

By allowing users to change the overall budget period and notification settings as well as having the transactions sorted by date, we provide the user with a greater level of system configurability, which we found increased system usability and accessibility.

Analytics Changes

Change

In the Analytics, we now display two different charts, one is a pie chart that represents the amount spent in each category compared to the total spending, and the second one is a line chart that represents all of the recent transactions for a given budget. The line chart also displays refunds as negative spending.

Rationale

By displaying two detailed charts, we allow users to see a broad view of all of their spending and the amount remaining in the overall budget, and the line chart below allows the user to see specific details of their exact transactions and the amount. These analytics will give users a clear and detailed picture of their recent spending.

Notifications Changes

Change

We implemented an alarm receiver class, and expanded on the notification class. When the budget is created we call the notification class to set two distinct alarms to check for the budget depending on selected frequency, and on the budget period. The notification classes also have update alarms to update the notification frequency and budget period check for the user.

Rationale

Android has an AlarmManager class that implements the timer to wake up the app on the interval and frequency. We used it and separated our Notification and AlarmReceiver class to take full advantage of the Alarm Manager. This also allowed us to separate our two distinct notification and alarm types, and allowed us to send notification only when the requirements were met.

5. Requirement Change

Sanity now supports the following requirements that were not included in the previous design document:

Functional Requirement

User login

1) Only one user can be online on one phone at a time.

2) Users are required to enter a name, password, security question, and security answer in order to create an account.

3) Users are able to change their password by entering their username and the answer to the security question correctly, then entering a new password to be recorded in the system.

These changes allow for *Sanity* to be a more secure system that require users to set up a security question and answer before logging in and allows them to change their password if necessary but only with answers to the security questions.

Create a Budget

1) There is only one overall budget manager per user, which is a container for individual budget categories.
2) The user will also be able to delete budgets. When a budget is removed, the budget limit is deducted from the overall budget.
3) The user will be able to set a budget period for each budget, the budget period changes will take effect immediately and budgets will reset at the end of the period.
4) The threshold for the budget can be changed by the user, these changes will be reflected immediately in the main menu.

These changes in the requirements specify more clearly the options that users have when it comes to their budgets, for example deleting and editing budgets, and how they are reflected in the system.

Analytics

1) The user will be able to see graphs of their overall transactions for the last month and their transactions per budget for at least the last six budget periods.

The previous requirements did not detail the time length that the analytics would cover, these are now specified in the requirements document.

Non-Functional Requirement

1) Any number of transactions can be made on different phones without crashing.
2) Any number of users can remain online at the same time without crashing.
3) The app will keep user passwords and security answers secure by storing hashes rather than user-entered values.
4) The app will be user friendly and intuitive to operate without the need of a user manual.
5) The app will not cause any excessive battery drain.

6) The app will be fault tolerant and will crash less than once per 20 transactions.

The previous requirements documentation did not detail

6. Use Case Discussions

1) Our implementation of this scenario would not drastically change our existing design since we already have an overall budget manager that stores information regarding all the budget categories. Using this object, we can specify transactions in multiple budget categories from a single view and the budget manager would be able to place each transaction in its respective budget category. This change would only require a new view in the display as well as several methods in the controllers and database to collect and save a group of transactions that are submitted simultaneously. Moreover, since the budget manager has access to individual budget categories, it can increase the budget limits for individual categories using the budget method: setMaxBudgetAmount().

2) While this has not been implemented yet, it will be easy to implement as we already have an alarmreceiver class that sends the notification, and we can allow the user to chose there, and we already have function in the database that allows the user increase amount spent,and overall budget amount so we can both roll over or transfer it to savings.We can also make it only for single period by making it temporary. This change will require few changes, but most of the methods and designs have already been implemented. The changes will be adding a “Savings”, and allowing a temporary roll-over.