Detailed Design

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 2158204741

2 Preface

This document outlines the detailed architectural and detailed design of the android application “Sanity.” This document is intended for the software engineers constructing “Sanity” and serves as a guideline of how to implement the application.

3 Introduction

The android application “Sanity” will be split into two design aspects: architectural design and detailed design. This document enumerates the architectural components of the system through an architectural diagram, a class diagram, a package diagram, and several sequence diagrams. Detailed explanations of our diagrams including component/use case functionality and our rationale for our pattern/design choices are discussed in the section immediately following each diagram.

4 Architectural Design

We will be using the three tiered state-logic-display pattern comprised of a UI display interface, a logic controller (server) and a database (state).

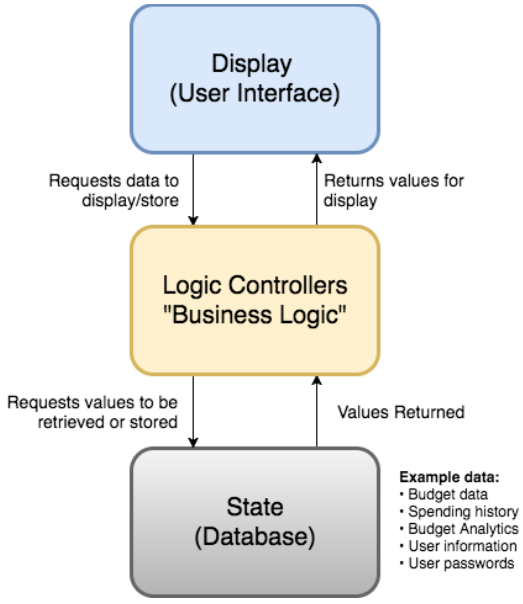

Figure 1: Three tier architecture diagram

The first component, the Display, will be for the user to interact with, create a profile, and add or remove budgets using the user interface. The Logic Controllers will contain classes such as the budget manager which can modify, add or remove budgets as well as the Budget classes which can check if the user has gone over the maximum amount. The first two components are connected by methods that request to store data such as addTransaction, and the Logic Controllers can also send lists of budgets back to the Display to show to the user. The last component, the Database, holds user and budget information and is connected to the other components by storing extra user and budget information on request and by returning stored the information on request.

The three tiered architectural pattern will allow the database, user display, and logic controllers to work separately since calls are made relatively rarely between the three, only when new information is added or displayed. Compared to the Model View Controller (MVC) architecture we believe our three tiered architecture to be better suited for our application design and system goals. We believe our Logic Controllers should act as a conduit between the UI and the database. MVC, however, couples the server functionality with the database in the Model, limiting the separation between two critical system components.

Sanity will implement the layered architectural style. Building off of the three tiered architectural pattern, the layered architectural style naturally comes to mind. As opposed to the pipe and filter style where system components do not know of other components and do not conform to a component hierarchy, the layered style has components that take precedence over other components, with a clear flow of control. By implementing the layered architectural style we can correctly model the system's components and connectors and the constraints that define their relations to one another. For instance, the state should have little to no interaction with the view, whereas the logic should act as a go-between with both the state and the view.

5 Detailed Design

5.1 Package Diagram

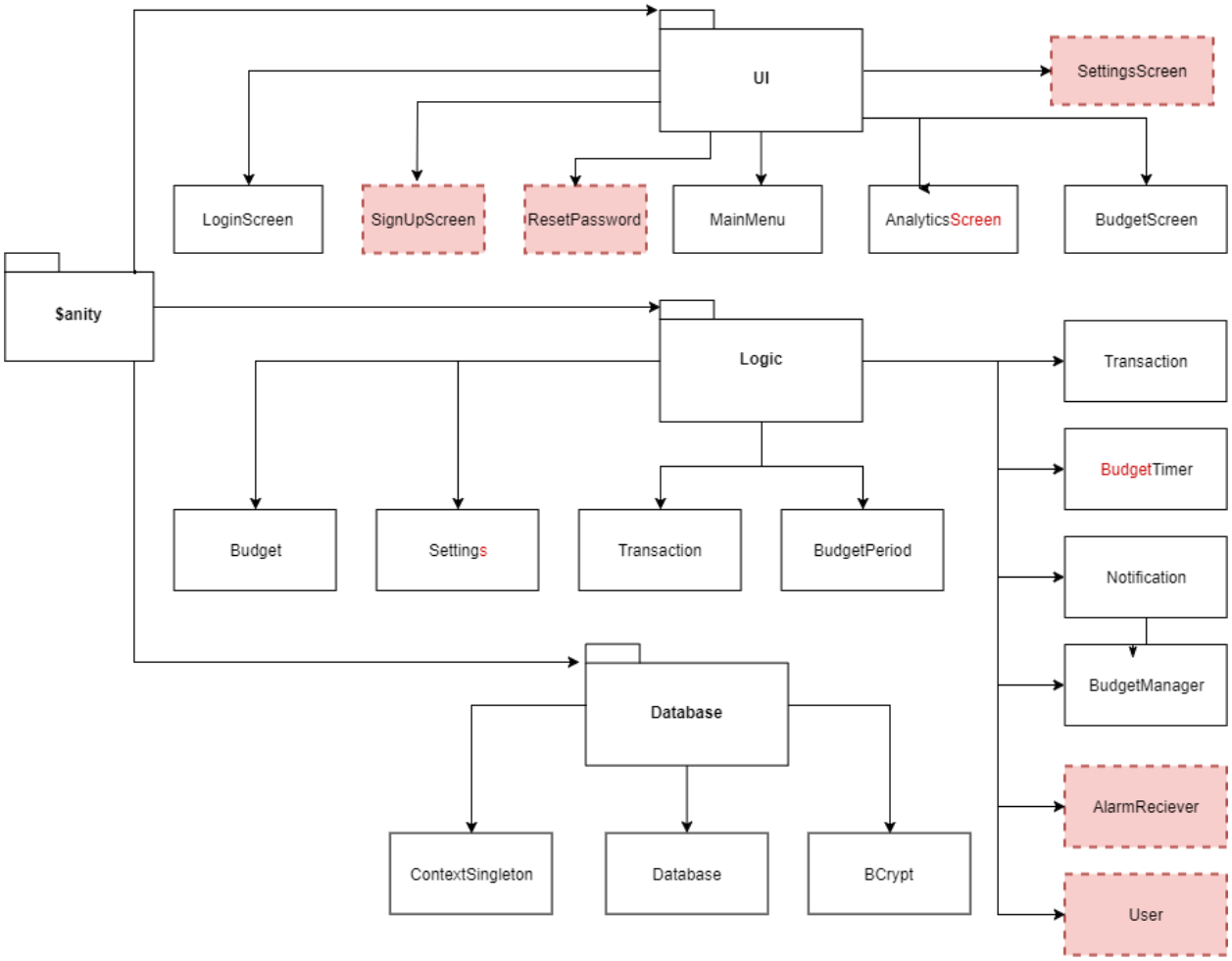

Figure 2: Package diagram

Above is the package diagram for Sanity. The diagram enumerates the different packages and illustrates the dependencies between the packages. The three main packages: UI, Logic, and Database, correspond to each of the three tiers in our three tiered architecture. Stemming from each of the three main packages are classes that provide services and functionality. For instance, the UI package is comprised of the LoginScreen class, the MainMenu class, the Analytics class, and the BudgetScreen class. Each of these classes defines a view in our UI tier, so naturally there is a

separation of control and an encapsulation of UI display elements and data such as buttons and table view text.

Note, however, that even though the UI, Logic, and Database packages are separated entities that have their own classes, the classes within each package will often interact classes in other packages. For example, a user event handled by the BudgetScreen class in the UI package will trigger a call to the BudgetManager class in in the Logic package which, in turn, may make a call to the Database class in the Database package.

5.2 Class Diagram

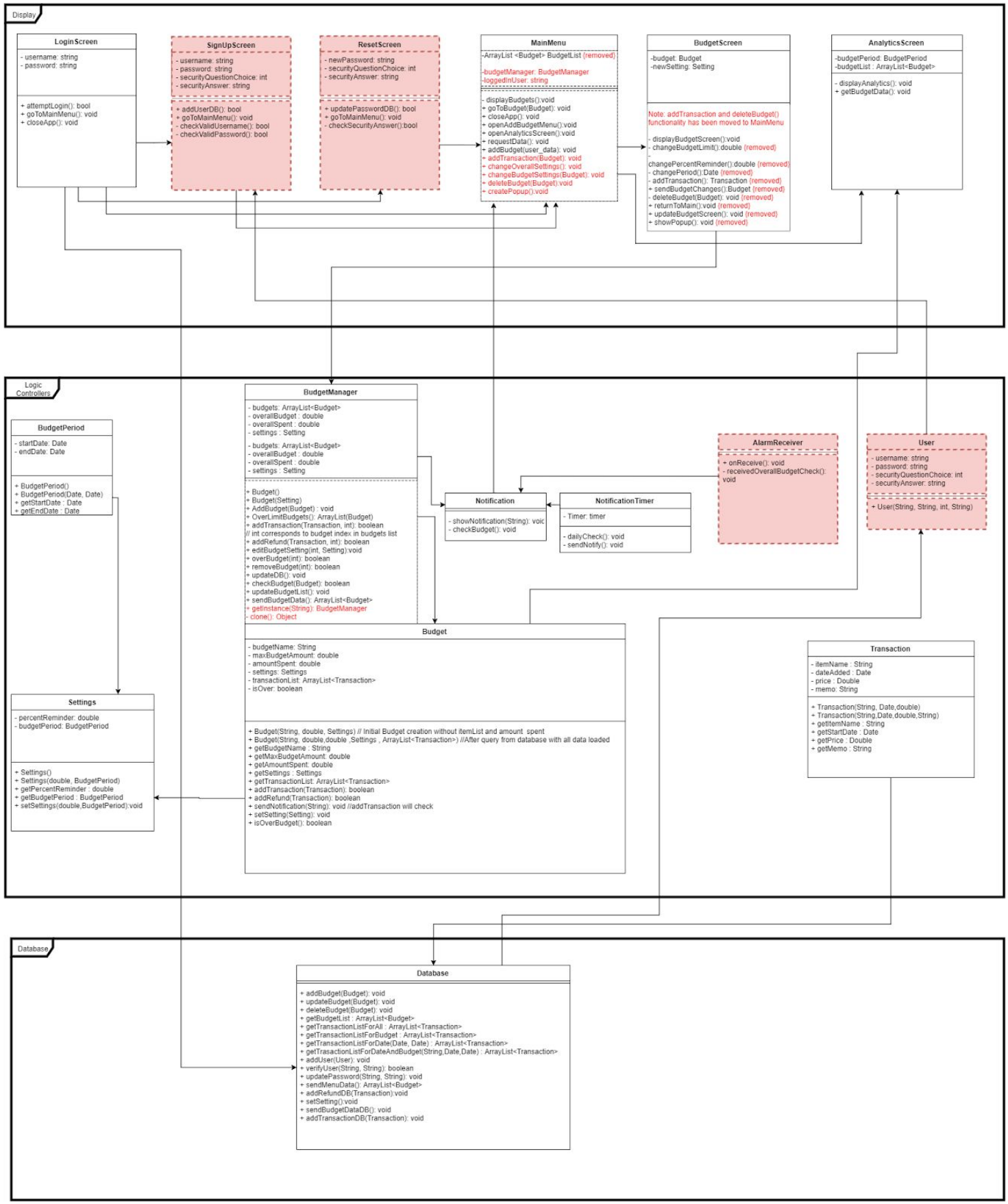

Figure 3: Class Diagram

The class diagram outlined above shows the interactions between classes and packages within Sanity. The class diagram is partitioned according to package where the three main packages: Display, Logic Controllers, and Database correspond to the three tiers of our architectural design pattern (three tier architecture). One unique design decision we made was for a the LoginScreen class in the Display package class to interact directly with the Database class rather than go through one of the Logic Controllers classes. Our rationale behind this was because the login functionality was not substantial enough to warrant having its own controller class, say “LoginController”.

As can be seen, the diagram is formatted in standard UML, where each class contains member variable declarations as well as function declarations. On further inspection, the reader can see whether a member/function is declared public (+) or private (-).

5.3 Sequence Diagrams

5.3.1 Notification Timer

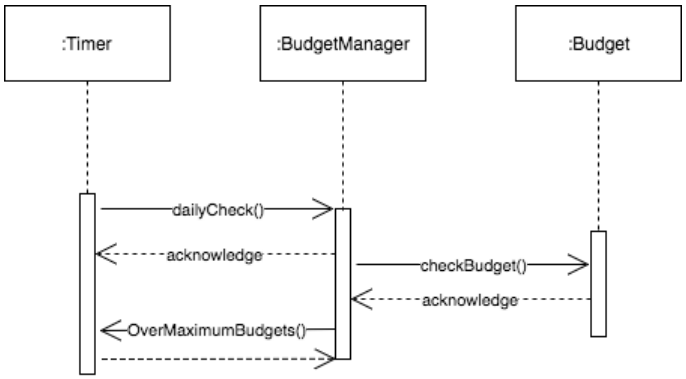

Figure 4: Notification Timer Sequence Diagram

The notification timer sequence diagram models the interaction between the Timer class and the BudgetManager class. In this specific scenario, the Timer is checking with the BudgetManager to see if any budgets have gone over their threshold

limits. If any budgets meet that criteria, they are returned to the Timer by the BudgetManager.

5.3.2 Add a Transaction

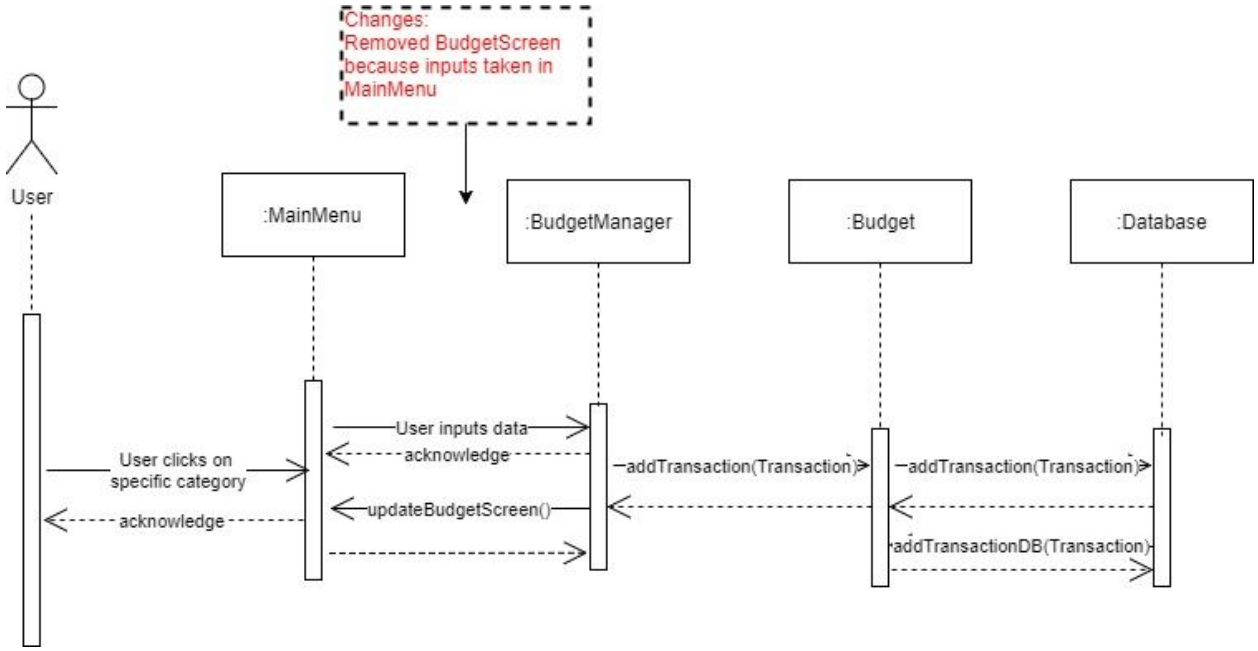

Figure 5: Add a Transaction Sequence Diagram

The add a transaction sequence diagram represents the action of a user manually adding a transaction to a specific budget. First, the user selects a specific budget category by clicking, which segues the user from the main menu to the budget screen. Here, the user inputs the data pertaining to the transaction (e.g. cost). Then, the budget manager will add the transaction to the database and upon receiving acknowledgement of a successful write, the budget screen will update the budget with the new transaction.

5.3.3 User Login

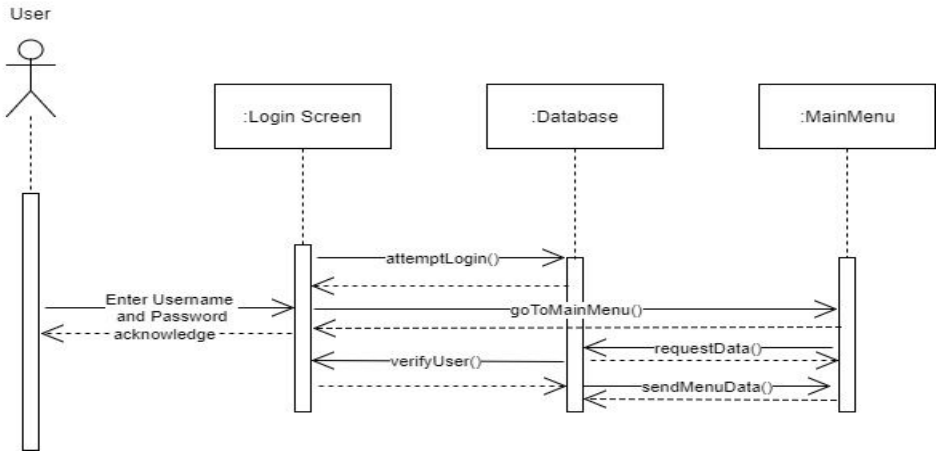

Figure 6: User Login Sequence Diagram

The user login sequence diagram represents the action of a user manually adding a transaction to a specific budget. First, the user enters the username and password in the login screen. Then, the login screen will make a call to the database and upon receiving acknowledgement of a successful login that matches the database, the login screen will take the user to the main menu.

5.3.4 Adding a Budget

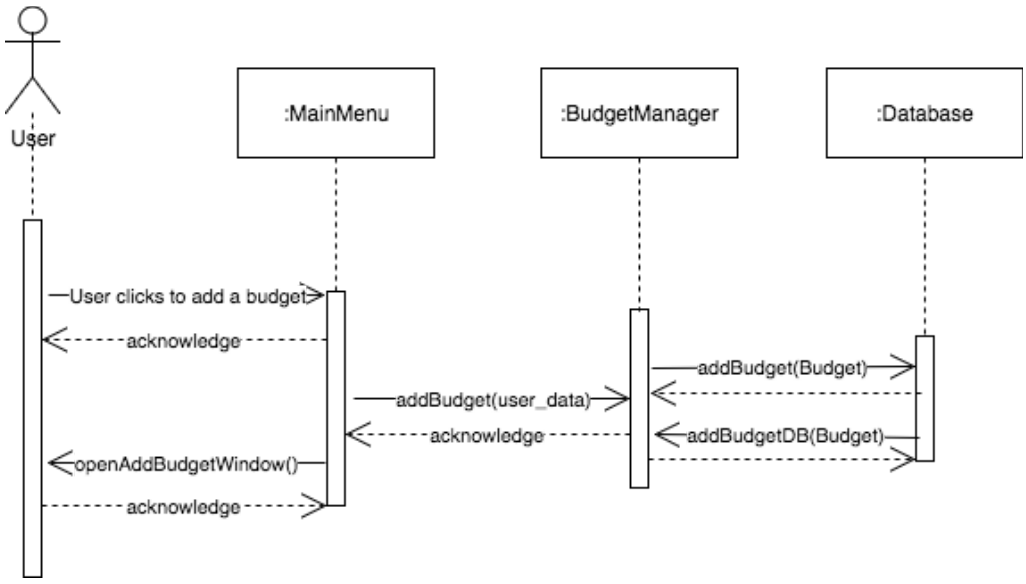

Figure 7: Adding a Budget Sequence Diagram

The adding a budget sequence diagram models the situation where a user decides to add a new budget to his/her list of budgets. In the main menu, the user clicks add a budget which calls the addBudget function with user_data as a parameter. The contents of user_data include budget time period, budget limit, among other metrics.

5.3.5 Adding Refund

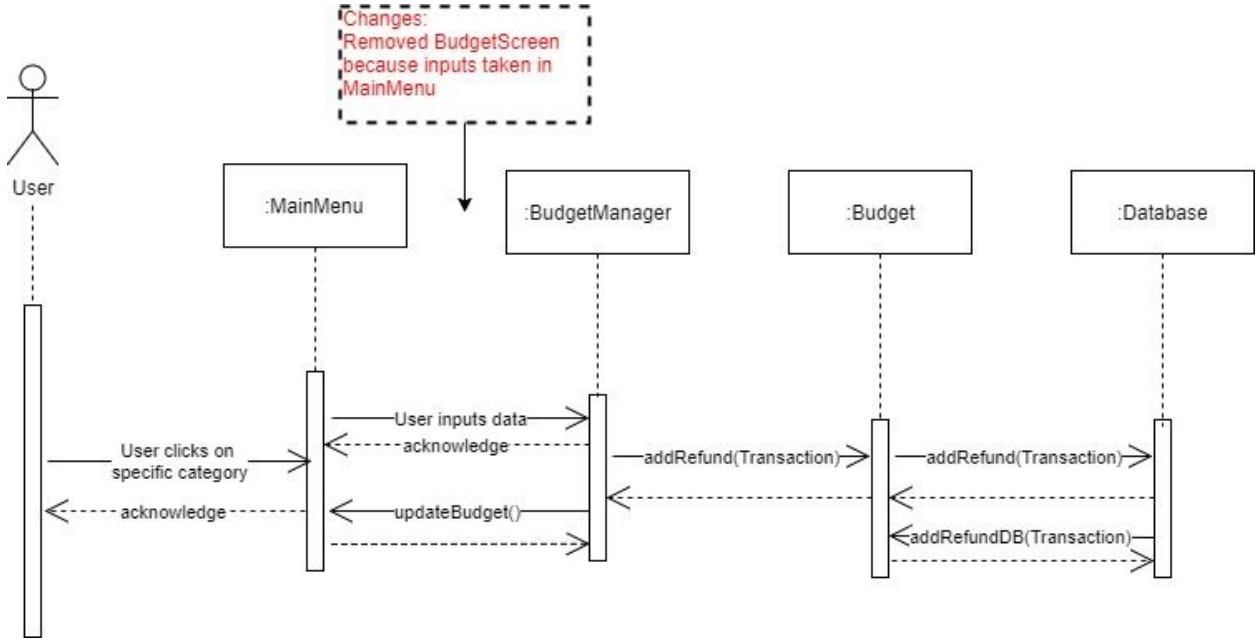

Figure 8: Adding a Refund Diagram

The adding a refund sequence diagram represents the action of a user manually adding a refund to a specific budget. First, the user clicks on the specific budget category that they wish to add a refund to. Then, the budget screen will allow the user to add the refund data and sends the data to the budget database which updates the appropriate data to the database. The budget manager then acknowledges that the data has been updated successfully and a confirmation will be sent to the user.

5.3.6 Show Notification

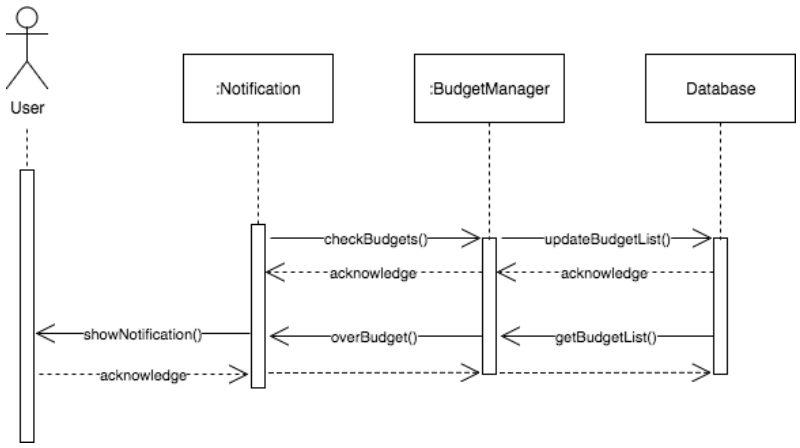

Figure 9: Show Notification Sequence Diagram

The show notification sequence diagram represents the action of the Notification class running its daily check of the user's budgets. The Notification class calls checkBudgets() which the BudgetManager knows to call updateBudgetList() in the Database class. Upon returning the budget list to the BudgetManager via getBudgetList(), the BudgetManager checks to see if any budgets are over their limit with overBudget(). If any of the budgets are over limit, then Notification sends a push notification to the user informing him/her of the budget warning.

5.3.7 View Budget Analytics

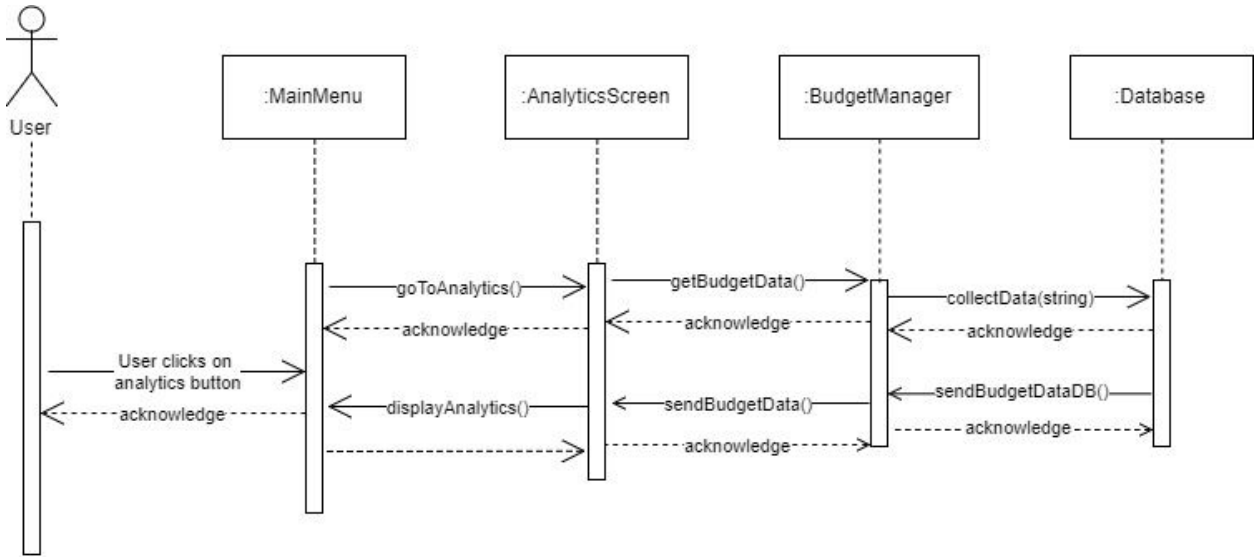

Figure 10: View Budget Analytics Sequence Diagram

The budget analytics sequence diagram represents the action of a user navigating to the analytics screen from the main menu. First, the user clicks on the analytics button on the main menu. Then, the analytics screen will make a call to the budget manager which will collect data from the database. The database then sends the budget data to the budget manager which sends it to the main menu to display to the user.

5.3.8 Delete a Budget

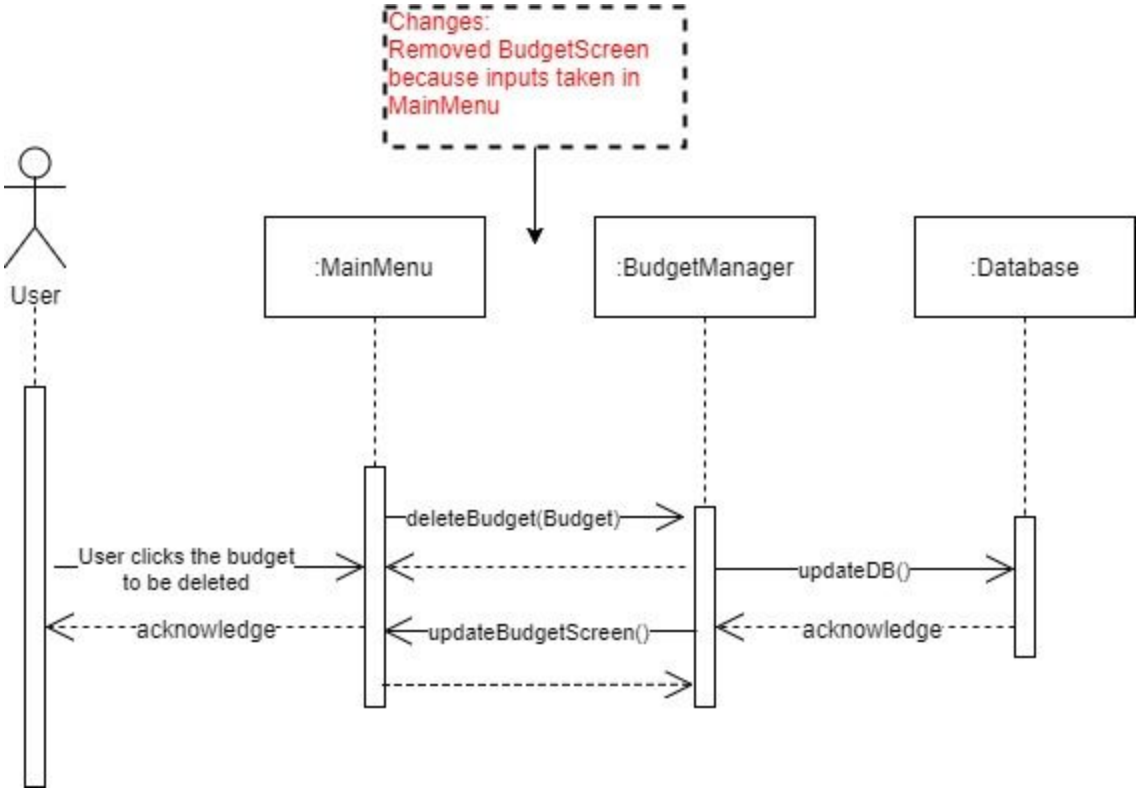

Figure 11: Delete a Budget

The delete a budget sequence diagram models the event where a user elects to delete one of his/her budgets from the main menu. Upon clicking on the budget to be deleted, the main menu will make a call to segue to the budget screen. In turn, the budget screen will make a call to budget manager via deleteBudget() with a budget object as an argument. The budget manager will, in turn, call updateDB() which will delete the budget from the database. On the return stream, the main menu will be updated with the new budget list.

5.3.9 First Time Over Budget

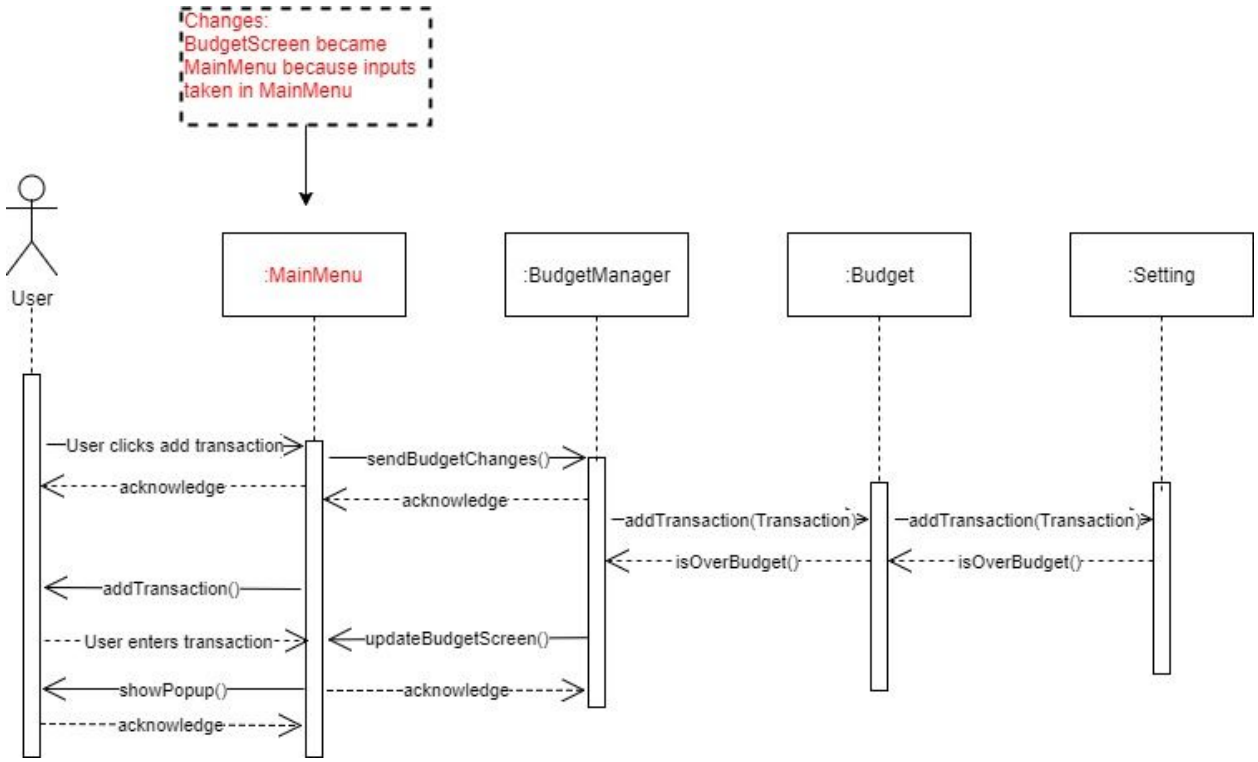

Figure 12: First time over budget

The first time over budget sequence diagram models the event where a user exceeds the budget for the first time. Upon adding a transaction in the BudgetScreen, it calls sendBudgetChanges in the Budget Manager. The BudgetManager then calls addTransaction with Transaction Parameter in Budget which in turn checks with the Setting if it is over budget. Settings finds that the user is over budget, and it calls isOverBudget which updates Budget, which updates BudgetManager. The BudgetManager updates the BudgetScreen, which shows a popup to the user.

5.3.10 Change Budget Limit

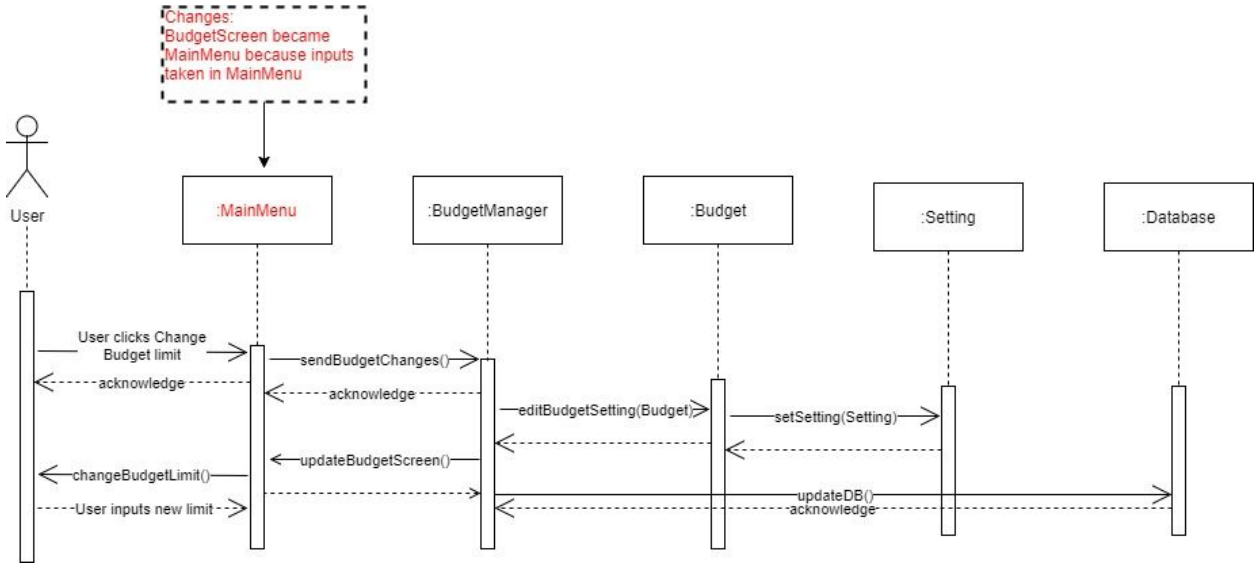

Figure 13: Change Budget Limit

The change budget limit sequence diagram models the event where a user decides to change the maximum allowed spending on one of his/her budgets. The budget screen calls the budget manager which adds a setting to the appropriate budget. The setting holds the maximum amount for the budget. After this, the budget manager calls to update the budget screen and the change is reflected in the budget screen for the user to see.