Food Budgeter

4002-542 Native Mobile App Development

Akia Vongdara, Tim Wong

Contents

[Definition Statement 3](#_Toc356394485)

[Market Research 4](#_Toc356394486)

[Sketches 5](#_Toc356394487)

[Implementations 6](#_Toc356394488)

[Current State of Implementation 7](#_Toc356394489)

[Future Releases 7](#_Toc356394490)

[Self Evaluation 8](#_Toc356394491)

[Definition Statement and Features 8](#_Toc356394492)

[iOS Software Design 9](#_Toc356394493)

[Database Modeling 10](#_Toc356394494)

[User Interface Design and Experience 10](#_Toc356394495)

[Above and Beyond 11](#_Toc356394496)

# Definition Statement

Food Budgeter allows user to keep track of the cost of what they eat. By using this app, users can set a spending limit for themselves on food, and even discover new ways to spend their money. This financing app may be popular for those who want to know more about their spending habits and change it. It also differs from other competing app in the sense that it gives more detail on the ingredients bought and used.

**Brainstorm Features**

* Add details to items used for their recipes such as the price and store location.
* Create and store recipes.
* Create a monthly and weekly budget.
* Create a weekly diet and spending plan.
* Recommend cheap restaurants.
* Recipes have an approximate cost estimated by the prices of the items used.
* View list of recipes.
* Sort recipes by cost, name, or time length.
* Search up other user’s recipes.
* Current items/ingredients database.
* View Monthly/Weekly Budget.

**Filtered Features**

* Add details to items used for their recipes such as the price and store location.
* Create and store recipes.
* Create a monthly and weekly budget.
* Recipes have an approximate cost estimated by the prices of the items used.
* View list of recipes.
* View Monthly/Weekly Budget.
* Sort recipes by cost, name, or time length.
* Current items/ingredients database.

## Market Research

**MoneyBook (3.5 stars)**

Features: Handles expenses and income. A handy overview of your previous transactions. Recording a transaction takes seconds. Rollover into following month. Number pad with calculation. Edit all your data at any time. Customizable categories. Set when your month starts. Switch between budget and salary mode. Progress over the previous months. Recurring monthly transactions.

Weaknesses: You can only set a per month spending limit (or income) and watch your spendings go into different categories. Updating incomes changes past incomes as well.

Opinions: The data in this app should be mutable.

**Toshl (4 stars)**

Features: Compare the rate of your spending with the time of the month. One-time budgets, daily budgets, weekly budgets, bi-weekly budgets, monthly budgets, yearly budgets. Move remaining funds to the next budget. Learn from your budget history. Plan family bills for upcoming months and know how much you can spend. Works with any currency. Enter custom exchange rates.

Weaknesses: Free version only allows one budget at a time.

Opinions: A very good app, but everything is too generalized. You don’t really know what items you bought.

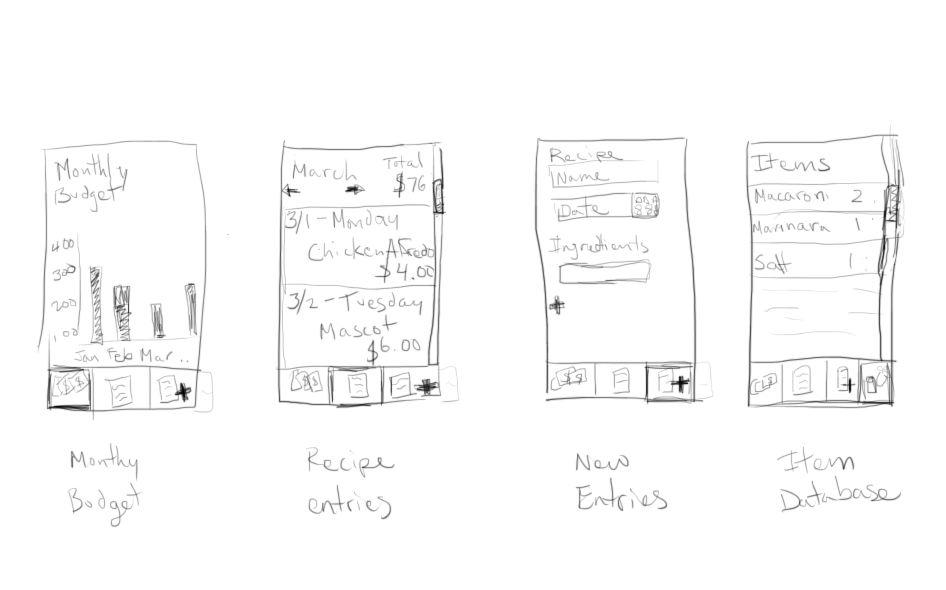
**Mint (4.5 stars)**

Features: View all of your accounts including: checking, savings, credit cards, investments, and retirement. Mint automatically pulls in and categorizes your transactions. No need to enter your own transactions. Know how much you have left to spend. Mint creates a customized budget based on your actual spending. Add manual transactions at the time of purchase so you can track cash spending.

Weaknesses: Generalizes purchases. You can’t know what you bought.

Opinions: A very good app, but everything is too generalized. You don’t really know what items you bought.

## Sketches



## Implementations

**Must-haves**

* View Monthly Budgets
* Create Recipe Entries
* View Entries
* Item storing
* Recipes uses cost of items to estimate the cost of the meal
* Sort entries
* View Weekly Budgets
* Set budgets

**If time allows**

* Fuzzy Searching for items

**Will not implement**

* Create a weekly diet and spending plan
* Recommend cheap restaurants
* Search up other user’s recipes
* Accessible database by any user

# Current State of Implementation

The features that have been implemented in Release 1 are as follows:

* Addition (logging) of food items
* Viewing all stored food items
* Removal of food items
* Saving food items in database
* View Total Spent (budget)

# Future Releases

Features that could be implemented in future releases:

* View budgets based on specified timeframes
* Support for automatic recipe cost calculation based on the grocery items used to make them
* Sorting of items
* Ability to set a budget
* Share recipes and budgets
* Undo additions, deletions, and any other modification of items and database
* Local and iCloud backups of database
* Export data in Excel format
* Process budget data in graph
* Recommend cheap restaurants based on user location

# Self Evaluation

We implemented the features that were the most important for the program to work as desired. The current state of implementation lists the features that were fully implemented in this release.

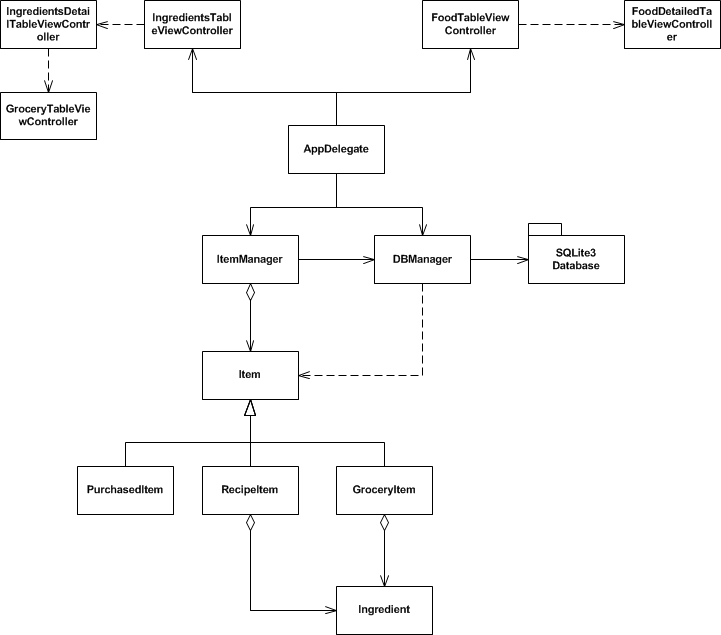
Unfortunately, we encountered many issues related to lack of time and availability; the schedules of the team members did not match up at all. Developer resources and knowledge were also low; we were still unsure of some concepts such as delegation and one of the team members did not own a Mac, greatly affecting his ability to contribute to the project. Due to these issues, we were forced to remove some features out of scope for this release.

Despite those issues, we utilized many features and followed processes that were not required, which resulted in a high-quality, well-tested, and easily maintainable program. Adding the features missed in this release should be fairly easy with our design. These features and processes are discussed more in the following sections. Due to this, we believe our project is of A-level work.

## Definition Statement and Features

Release 1 of our product meets the most important functionality listed in the product definition statement that enables all the other functionality we had planned to even be possible in the first place. Such functionality includes the addition and removal of different kinds of items and the ability to store and recall them from the database.

## iOS Software Design



The software architecture was based upon Model-View-Controller (MVC); Control and ViewController modules perform operations on Model (item) modules, then View modules (the xibs) are updated as needed. An example of this is when items are added into the database – the ViewController invokes the operations, which then performs operations on the model and the database, then updates the view with a message based upon the result of the operations.

While Objects that should not aggregate each other should utilize protocols and delegation to communicate with each other, due to the lack of developer knowledge and time constraints, we have resorted to simple references between objects. Only a few of the ViewControllers use the Delegation pattern. We plan on refactoring this in a future release.

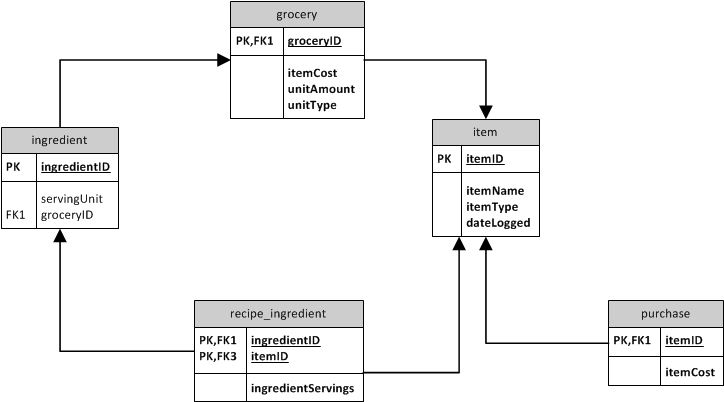
Lazy loading is used for all Table views; data for a specific table cell is not loaded until the cell is within view of the user. However, in the interest of time, the entire table is reloaded when items are added or removed.

Throughout project development, patterns of repeating code were refactored into single methods and classes, therefore increasing code maintainability and reusability. Some Examples

*Gang of Four* design patterns were also utilized; the application of these patterns is explained more in depth in the Above and Beyond section.

Each operation involving items is reflected within the database. For example, when an item is added to the program, that item is also written to the database. This allows for short, frequent usage of the application without losing the state should the application be closed or halted.

## Database Modeling



The SQLite database was designed upon the relational model and mirrors the class representation of the model. Data is normalized into different entities to reduce modification anomalies and duplicated data, minimize redesign of the entire database when expanding upon it, increase design readability by other users, and allow for generalized queries. Furthermore, the database design allows for subtypes to be easily created instead of solving insertion anomalies with NULL values and allows for easy detection duplicate data and prevention of it from being added.

## User Interface Design and Experience

The appropriate ViewController objects were used in appropriate places; views that listed out items utilize UITableViewController and the different user modes (i.e. view items, log items) are easily accessible via the UITabViewController.

The sections of the TableViewControllers are separated if appropriate; the list of all items contains only one section, but the detailed view controllers have a section for each piece of data.

Pushing and popping views in a Navigation Controller are all animated for aesthetic purposes.

Each action notifies the user of its success – for example, if an item is successfully added, the view displays a small but noticeable success message to the user, which aids in keeping the user’s focus on the application. When an item is not added due to an error, the error is handled in a graceful manner; a notification pops up to get the user’s attention and turn it to that error and a message is also displayed where the success message is also displayed.

## Above and Beyond

The project contains extra work not required by the Final Project Deliverables document:

We followed a formal approach to developing this program, based upon a Sashimi model-like software development lifecycle model:

* Time was dedicated to brainstorming and analyzing business requirements, then translating them into technical requirements.
* Afterwards, time was dedicated towards the design of the software and the model of the database. In our software design, we aimed to reduce tight coupling between classes by following the Law of Demeter, increase the cohesion of each class, and adhere to the MVC architecture. In our database model, we aimed to normalize data to eliminate duplicate data and anomalies that may occur with the many operations that would be performed on the database. We adjusted our design and requirements as necessary.
* Implementation followed design. One person worked on the backend model while the other worked on the view, then the two were connected through implementation of the controllers. We adjusted to changes in our design and requirements and applied them to our implementation as needed.
* Automated unit tests were written using the SenTestKit framework, which is available in the iOS Software Development Kit (SDK) and can be run using the Test operation in XCode. These tests contained functions that needed to be tested (such as writing and retrieving from the database) and could be automatically run. Therefore, time spent on testing was reduced compared to the time it would have taken to manually test database and model subsystem functionality via running the app. We continuously tested using these tests as well as manual tests via the GUI during the course of implementation to verify and validate our functionality.

We kept all development under automated source control using Git and Github. The source code and commit log can be seen at <https://github.com/timwong1992/FoodBudgeter>.

All item data is stored in a local SQLite3 database using the SQLite framework, which is included with the iOS SDK. The use of a SQLite database allows for the use of fast, optimized queries that return only the data the user needs, saving valuable time and resources. In contrast, property lists load all of its data into memory upon application launch. Furthermore, the database allows the user to store as much information as the device can hold; a plist will only be able to store as much data as the device can load in memory before the application causes errors.

The *Gang of Four* design patterns, which are well renowned in the field of software engineering, were utilized in this project. When applied in appropriate contexts, these design patterns utilize the principles of object-oriented programming and software design to solve common problems encountered in software. These patterns generally increase code reusability and class cohesion as well as reduce tight coupling when applied properly. Furthermore, since these patterns have already been proven to work for specific problems, time spent on testing could potentially be reduced as developers do not have to verify if the pattern works and instead only validate if it works correctly. Specific patterns utilized in this program include the Command pattern (RemoveItemCommand) and the Façade pattern (ItemManager, DBManager).