TRACE

THE FINANCE TRACKER

2020

MAY 1

COMP 129

Authored by:

Brandon Reno (Scrum Master)

Michael Lanners (Product Owner)

Mason Mendoza (Programmer)

Brandon Raboy (Programmer)

Trace

Summary of Changes

- Abandoned photo of bill feature
- Changed recurring bill feature to push to next month after due date passed
- Abandoned account history feature
- Abandoned icon/colors for different bills
- Abandoned android build (due to technical issues)

Customer Statement of Requirements

Trace is a mobile application that will help the user track and maintain a consistent bill schedule using a calendar and notification system. By using this application and its notifications feature, the user will never forget about an upcoming bill which is commonly monitored by a planner, a practice that is tedious. By forgetting a bill, a person will undergo a late fee, a burden that could have been prevented if the person had Trace. Unique in the way that it makes keeping bills organized and neat due to the calendar feature which has all your payment dates laid out on one month by month calendar. Push notifications will also remind the user, on a specified date, that they must pay a bill soon. Most apps of this genre are forgettable and make the user check them to see when they must pay. Trace is keen on being user friendly, clean and straightforward. Reminding you of when bills are due and having each proceeding due date laid out. Trace is also set apart by the fact that you can keep track of multiple cards and have multiple payments laid out for different cards. You do not need to create five different profiles for five different cards but can rather house five different cards under one profile. Being able to do this and having an organized laid out calendar as well as the feature of notifications allows Trace to soar above its competitors. No more is the billing application world full of foreign language and confusing buttons. Trace is here to make paying bills simple, easy and organized.

Glossary of Terms

Profile – Refers to the unique name associated with the portfolio for the user.

Account – Individual financial accounts within the portfolio.

Balance – Total value in USD due on the account.

Interest – An amount of money charged over a set interval when a loan is active on an account. The exact amount is calculated differently for each loan.

Portfolio – Contains the user's full list of accounts.

Kivy – Python library for creating UI

Buildozer – Python library for creating Android application

Functional Requirements Specification

Stakeholders

College students needing help keeping track of bills

Actor

User (Student tracking bills)

Goals

Budget expenses

Track due dates

Prioritize high interest rates

Use Cases

Initializing profile

ID:	**UC-1**
Title:	**Initializing Profile**
Description:	**Account holder inputs all account information after setting up new profile.**
Primary Actor:	**Account holder**
Preconditions:	**Account holder has set up profile in app**
Postconditions:	**Account holder has all accounts saved in profile**

Main Success Scenario:	**1. Account holder selects "Add Account" once for each separate account he wants to record in profile.** **2. Account holder enters all information for each account.** **3. Account holder saves each account into profile.** **4. Account holder opens settings tab.** **5. Account holder enters total budget amount.** **6. Account holder selects preferred date format.**
Extensions:	**2a. Some of the information entered is not valid.** **— 2a1. System fails to initialize account.** **— 2a2. Account holder enters information a second time, ensuring all data is valid.** **5a. Total budget is not high enough to cover minimum payments for all accounts.** **— 5a1. System alerts account holder that they have not budgeted enough money to pay all minimum balances.** **— 5a2. Account holder either enters a higher budget amount or confirms they want to continue with the current budget amount.**
Frequency of Use:	**Once per profile**

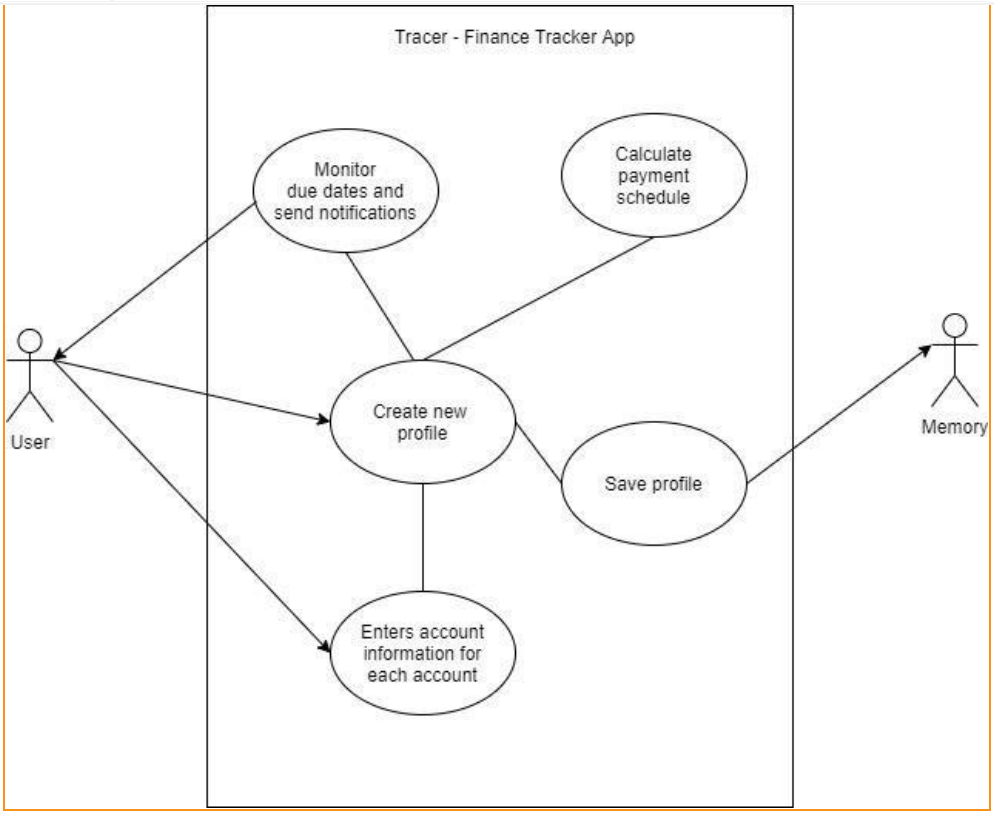
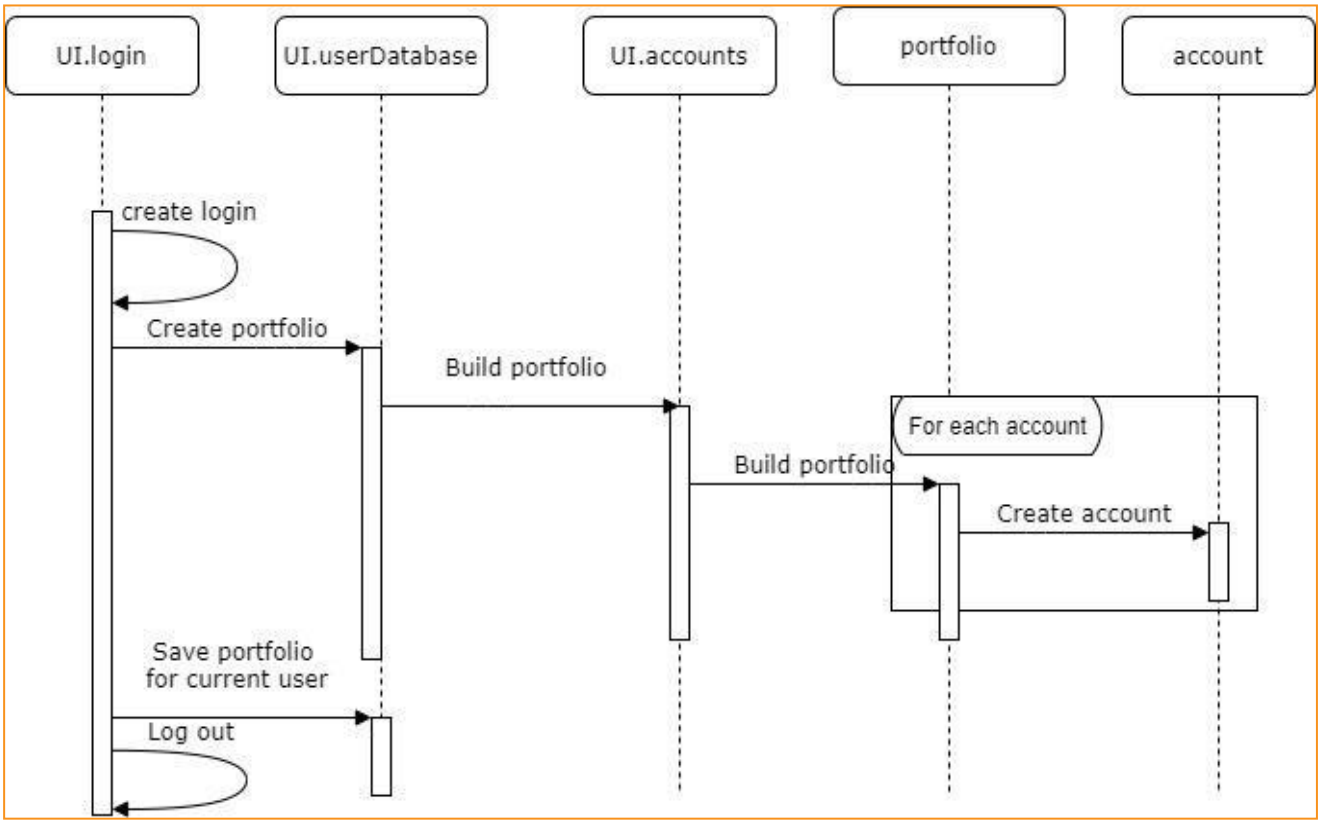


Figure 1: UC-1 Interaction Diagram

In the following sequence diagram, you can see how we use the portfolio class and account class to compartmentalize the storage of data. This allows each class to handle the specific tasks related to it, while allowing the UI classes to easily request the data it needs to display the desired output.



Adjusting budget amount

ID:	**UC-2**
Title:	**Adjusting budget amount**
Description:	**Account holder determines their current budget does not meet their actual ability. Account holder adjusts saved budget amount so their payment schedule will reflect their actual financial abilities.**
Primary Actor:	**Account holder**
Preconditions:	**Account holder is logged into profile**
Postconditions:	**Account holder's budget is adjusted to a new amount**
Main Success Scenario:	**1. Account holder clicks on settings tab. 2. System displays current settings. 3. Account holder clicks on budget setting. 4. Account holder enters new desired budget amount. 5. System saves new budget amount to profile.**
Extensions:	**4a. Budget is not enough to cover the remaining minimum payments. — 4a1. System alerts account holder. — 4a2. Account holder either enters a higher budget or confirms that he wants a budget that will not cover remaining payments. 4b. Budget is higher than total balance of all remaining accounts combined. — 4b1. System alerts account holder. — 4b2. Account holder either enters lower amount for budget or confirms they want to budget more than necessary to pay off all accounts.**
Frequency of Use:	**Whenever account holder's available budget changes.**

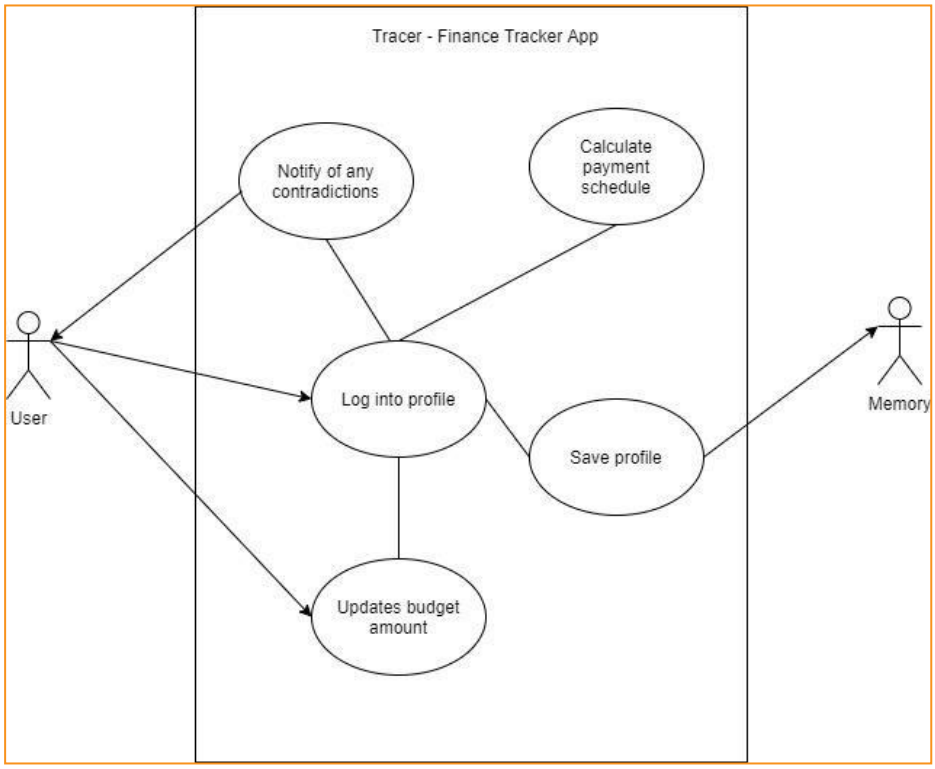
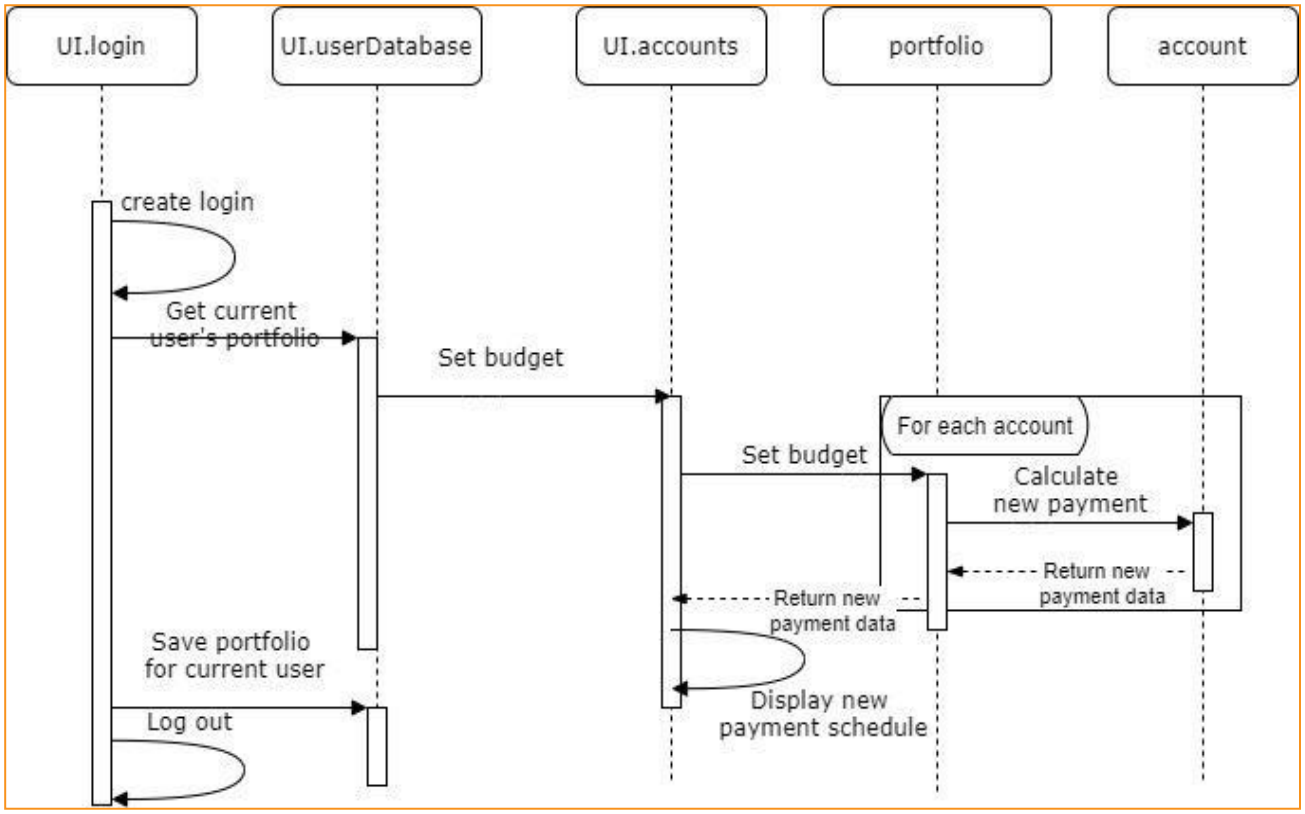


Figure 2: UC-2 Interaction Diagram

Portfolio gets new budget and has accounts update, then returns new schedule. Allowing each class to manage itself as it knows best.



Advancing schedule

ID:	**UC-3**
Title:	**Advancing schedule**
Description:	**Account holder wants to update account information and calculate payment schedule for the new month.**
Primary Actor:	**Account holder**
Preconditions:	**Account holder has an existing profile that has previously been initialized.**
Postconditions:	**Account holder has new payment schedule.**
Main Success Scenario:	**1. Account holder selects "Update Accounts" from menu. 2. System iterates through accounts verifying all information is accurate. 3. Account holder selects "Calculate Payments". 4. System displays current payment schedule.**
Extensions:	**2a. Account holder needs to update some information on an account. — 2a1. Account holder selects "Update Account". — 2a2. Account holder enters updated information.**
Frequency of Use:	**Once per billing cycle.**

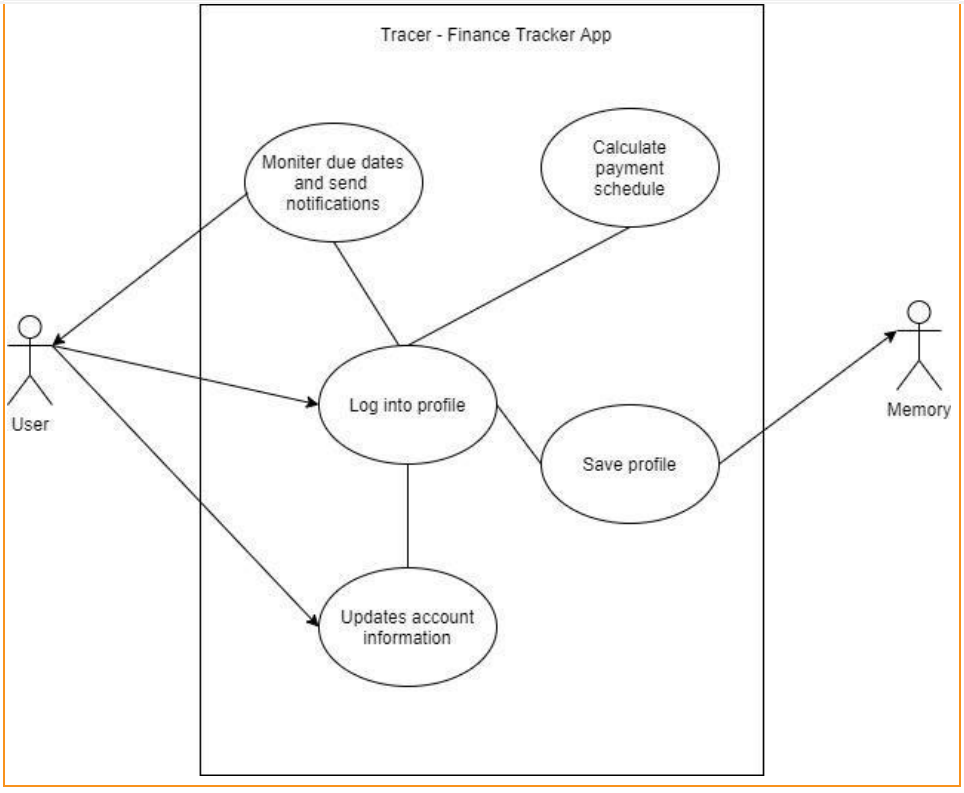
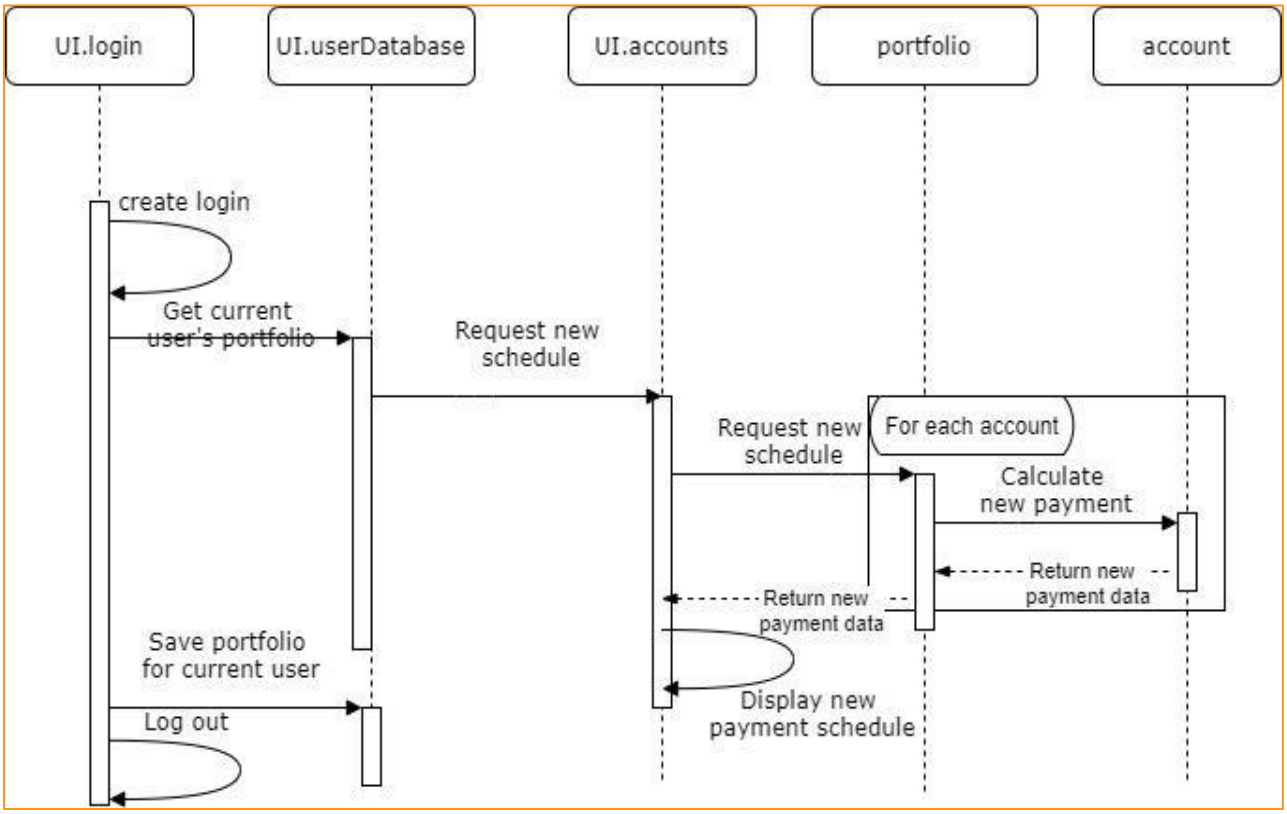


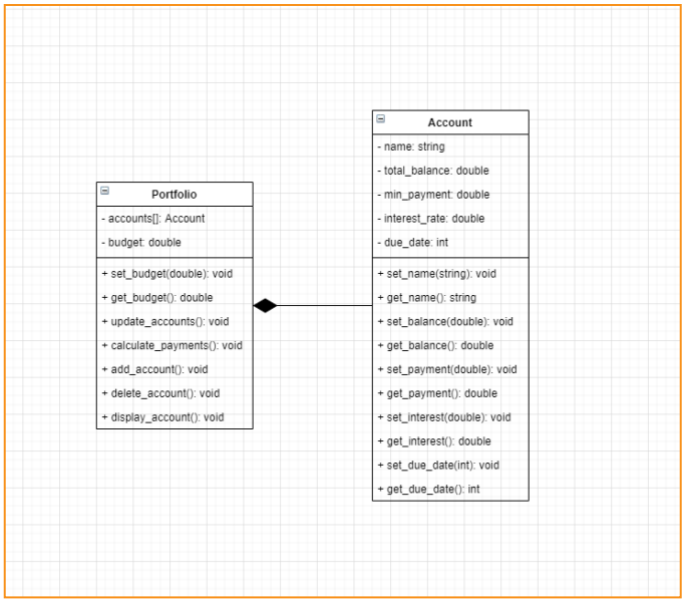
Figure 3: UC-3 Interaction Diagram

With the compartmentalization of the classes, this process is very similar to the setting budget one, with the main difference being instead of updating budget amount, the portfolio is just updating the due dates of the accounts to the following month.



Class Diagram

The class diagram shows how the individual object classes manage their own internal variables. Portfolio's functions of update_accounts() and calculate_payments() are where the algorithms come into play to update each individual account's information to reflect the portfolio as a whole.



Interface requirements

Username - unique non-null string

Password - string of characters between 8 and 14

Account name - non-null string

Total balance - numerical value

Interest rate - numerical value between 0 and 1 (representing percentage of annual interest charged on account)

Minimum payment - numerical value greater than or equal to 0

Budget - numerical value greater than or equal to total minimum payments

Due date - valid future date

Nonfunctional Requirements

Functionality:

The system shall work on current Android and iOS operating systems.

The system may not have full capability while no internet connection is available.

The system shall request a password from the user upon opening the application.

Usability:

The system shall use a color palette aimed for easy readability.

Appropriate UI buttons will respond to touch from the touchscreen.

Reliability:

The system shall store information into dependable memory.

The system shall test datatypes before using them.

The system shall be capable of receiving updates with an internet connection.

Performance:

The system shall store all data locally to improve speed.

The system shall not use RAM due to the nature of smartphone applications.

The system will use very little memory for full functionality.

Supportability:

The system shall receive updates when necessary.

Code shall be documented allowing for new programmers to maintain the application easily.

Tests shall be conducted with each operating system update to ensure compatibility.

System Architecture and System Design

Architectural Styles Abstract-data-type style

Problem: Allow users to manage and manipulate data, which is to be stored and protected locally via encryption. Be able to notify the user at appropriate times.

Context: Pythonic methods using mobile friendly libraries

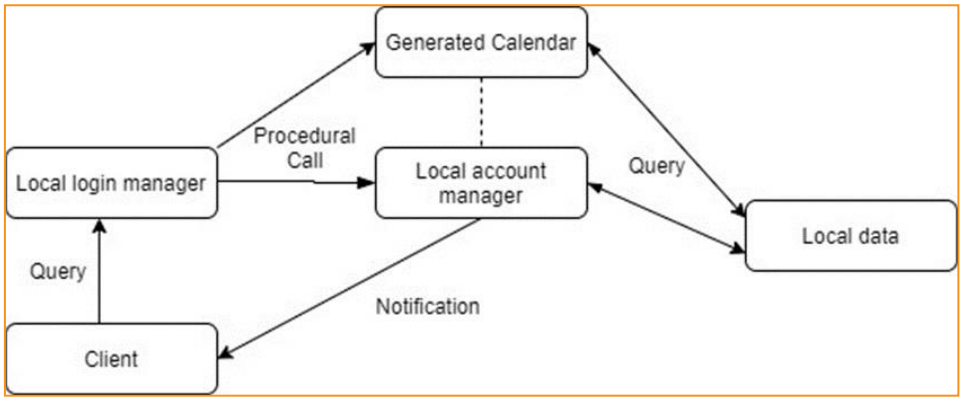
Solution:

System Model: Component stores data locally

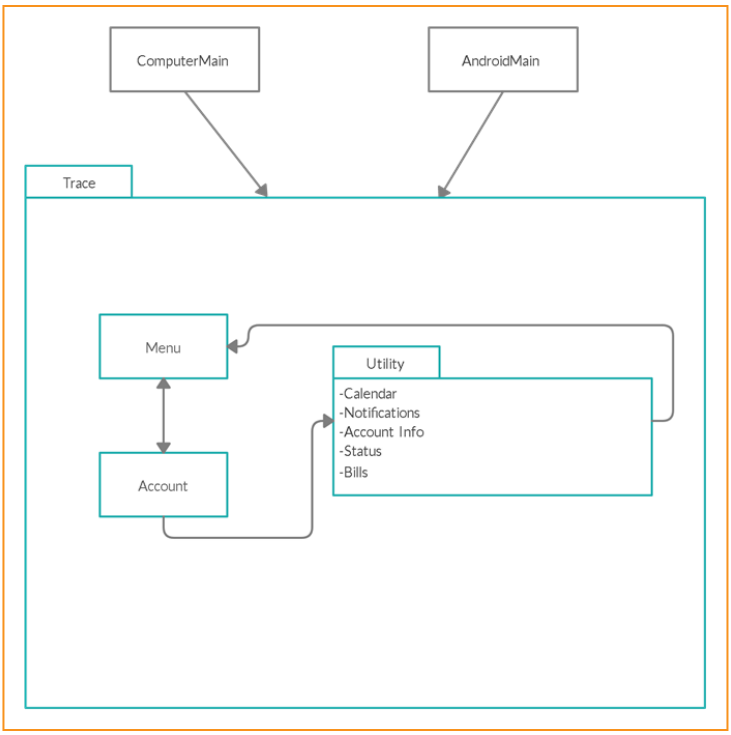
Components: Modules using local data, managers

Connectors: Procedure calls, automatic invocations for notifications

Control Structure: Single thread, control is centralized



Subsystems



Persistent Data Storage: Local storage is required. Username and password are both stored locally, under specific encryption for security. All information the user chooses to save, including portfolios, and account data will also be saved locally. All files will be saved as .txt files.

Network Protocol: This is N/A for our system because we run on one machine.

Global Control:

Execution Order: Our system is designed in a linear fashion meaning that each user must go through the exact same steps in order to setup their account. Once the account is set up, then we are nonlinear, and the user can do as they wish whenever they wish. In the beginning all users must create an account, and login to their created account. Upon completion they are free to do whatever they want in a loop format.

Time Dependency: There will be timers in our system in the field of DateTime objects in which track which due dates are due when and send notifications to the user on those dates. We do not have any times outside of this for example something that is constantly updating. Our time constraints are real time and can be classified as periodic since they are used every month to notify the user.

Concurrency: Our system does not use multiple threads.

Hardware Requirements

Display minimum resolution of 370 x 600 pixels

Disk storage minimum of 20 MB

Algorithms and Data Structures

Algorithms

The only algorithm we use is one to calculate the ideal payment schedule based off the various interest rates of the accounts and total budget available to be distributed. At the time of execution, the program will arrange the accounts by interest rate (descending) and then starting from the highest interest rate, it will determine the difference between the remaining budget and the amount required to pay the minimum balance of the remaining accounts. Then it applies as much of the difference to the current account as possible, without exceeding the total amount owed. It will then proceed to the next account and continue until suggested payment has been calculated for all accounts.

Data Structures

Array - The portfolio holds an array of accounts. We felt this was the most efficient means to store them, as it gives us the flexibility to sort in various ways at various times. For instance, when calculating payments, we sort the array by descending interest rates, but when viewing payment schedule, we sort by ascending due dates.

UI Design and Implementation

Preliminary Design: Users will be greeted with a login screen asking for their username and password credentials. If the user has an account, they will enter these credentials to login. If the user does not have an account yet they are given the option to create an account using a button at the bottom of the page “Create an account”. This screen walks the user through creating an account and then they are prompted back to the login page to log in. Upon entrance into Trace there are three options or boxes to tap.

Portfolio, which will let you access your accounts, Calendar, which will let you see your payment dates in a calendar format, and about which will contain information on Trace. Parameters the user will have to enter range from their username and password on the login page to their account information when adding or editing an account. Account information will include an account name, total balance, minimum payment, interest rate and due date.

User Effort Estimation: Creating an Account is the first scenario we will walk through. Upon launching Trace, you will take two taps or clicks into each text box on the login screen to enter your username and password. Your third click will be pressing login, this is assuming you have an account. The user will tap portfolios and then tap add accounts and this will prompt the user with the various parameters they need to enter to create an account. With five different fields that need user entry the user will have to tap five times and enter five different values about their account. Upon clicking submit the account has been created. For this scenario, it weighs more on the clerical data entry side due to the user entering more data than clicking (excluding clicks on different data boxes).

Checking the calendar is the second scenario we will walk through. Entering the app is the same as the beginning scenario, three taps and two inputs, upon entrance we will click calendar and a calendar viewing all your payment dates from all the accounts in your portfolio appears. With this scenario it is more weighted on user interface navigation since the number of taps outweighs the number of inputs the user needs to enter.

Viewing accounts is the final scenario we will walk through. The beginning is the same as the previous two, entering credentials is three taps and two inputs, clicking the portfolio button is another tap and clicking view accounts is the final tap. Upon clicking this button all accounts are shown that the user possesses. This again weighs on the user navigation side as the taps outweighs the amount of inputs entered again.

Screen Mockup Revisions:

In report one the initial screen mockups are very black and white in color and in functionality. Nothing is very stylized, and it is all very blocked out and plain. We decided to go the route of simplicity and use simple icons on a tinted yellow background to style our page. We decided to do this for ease of use. We are creating a bank app, and with banking the first thought that comes to someone's head is stress. So with our UI design we wanted to make the user feel a sense of ease for each action they are performing. This is also why we have only 1-5 icons per page, so the user does not feel overwhelmed while figuring out what they need to do. We also changed the font sizes on screen and popups to match the new font style which is overall just more fun that our original. We made this change because we want this app to be inviting and welcoming, not too professional and uniform.

Use Case Final Demo:

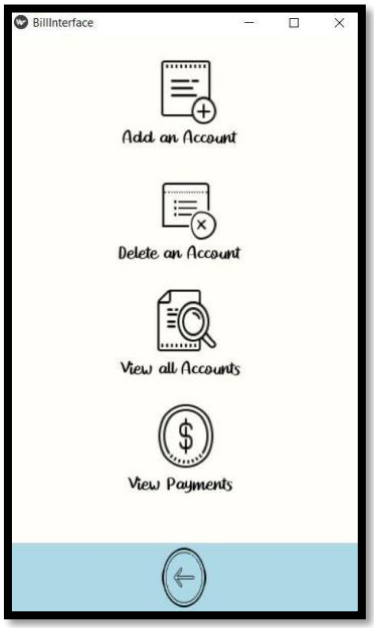
We will show one use case in our final demo and that will be going through the process of creating a profile and adding an account to your portfolio. To do this the user clicks create a profile. The user then fills out their information and clicks create profile. They are then prompted to login to their new account and upon doing so land on the main menu screen. From here they go to the settings tab and set their portfolio budget. Upon setting this number they can click on the Email notifications tab to sign up for email notifications. Next the user will go back and click on portfolios which will lead them to a new page with multiple options to do with your portfolio. The user will click on add an account and then enter the information that is prompted. Upon entering the information, a popup alerts the user that the account has been made. The user can make sure the account has been made by clicking on view accounts where all accounts that have been added to their portfolio stay. At this point the user can confirm that they have added an account and the demo is over.

UI images

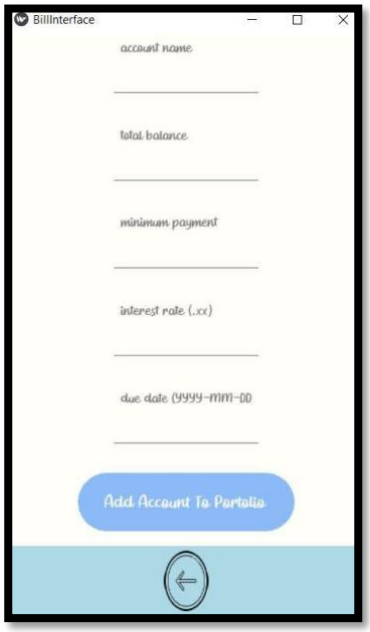
Login screen

Account creation

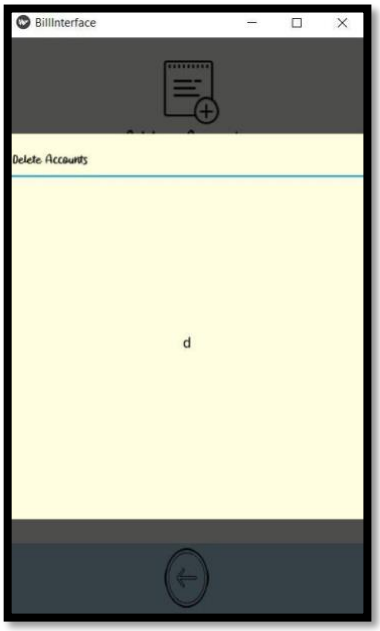
User Home Page



Portfolio screen



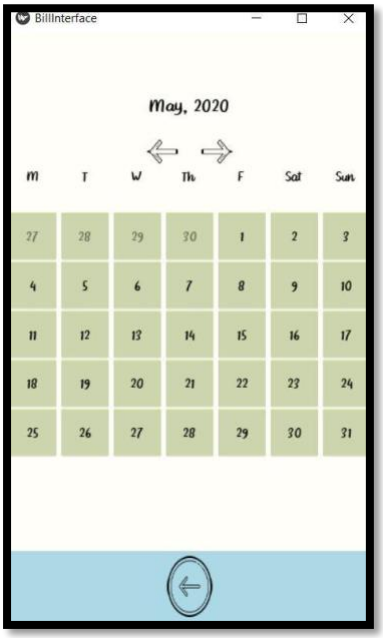
Add Account screen



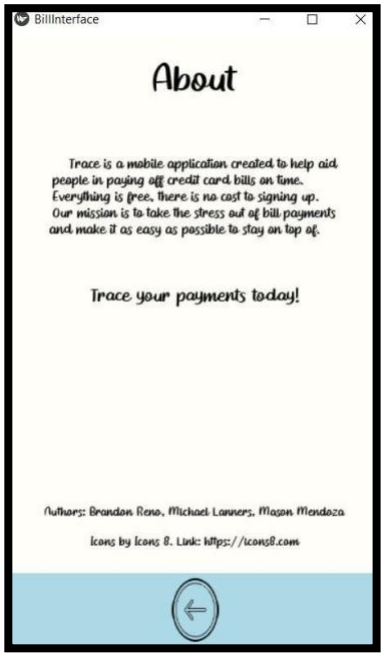
Delete Account screen

Change Password screen

e-mail setup screen



Calendar screen



About page

History of Work

1/20 – Got kivy and python running correctly

2/28 – Monthly bill list ordered by status and date

3/6 – Calendar view

3/13 – Upcoming bills view / encrypted password
3/20 – Recurring bills/push bill to next month after due date
4/3 – Notifications feature
4/15 – Debugging/Refactoring

Current Status of Implementation

All use cases have been implemented and all basic functionality is complete. Meaning a user can create a profile, log in to their profile, add accounts to their portfolio, view these accounts and all of their information, see which accounts need to be paid first, and view a calendar illustrating which accounts need to be paid which days. The functionality that we are missing at this point is a notification system to alert the user that they have a payment that needs to be made. We had talked about linking this application to your actual bank account and if time allows us this would be another feature that we would want to add to our project.

References

"Boost Your Builds!" *Buildozer*, 2012, Buildozer.

Rutgers University. "Software Engineering." Software Engineering, 2012, www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf.

Virbel, Mathieu, and Gabriel Pettier. "Cross-Platform Python Framework for NUI." *Kivy*, Jan. 2011, https://kivy.org/.