FINANCEFX FINAL REPORT

By Gavin Wells & Nicholas Rinaldi

List of Figures

List of Tables

1. Introduction

1.1. Purpose and Scope

The purpose of this finance application is to provide a means for a single user to look at their balance, income, and expenses in an organized manner (all in one place) and to provide a way to calculate their future balance and savings after inputting specific changes to their income.

1.2. Product Overview (including capabilities, scenarios for using the product, etc.)

The application can be used on a single Windows computer. It is ideal for situations in which the user needs a simple, condensed view of their balance, income, and expenses. When the necessary information is provided, there are graphs in the Calculations scene of the program to give the user a visual display of their finances, and an investment calculation feature. While providing a convenient overview of one's finances and simple prediction analysis, the app is not meant for advanced money management. The application will not automatically update the user's balance, income, and expenses whenever changes happen in external banking apps, so the user must input those manually. There is no login functionality either, so the user must keep information stored by the application protected by their own means (i.e. password lock on their computer, keep the computer in a safe place, etc.).

1.3. Terms, Acronyms, and Abbreviations

IDE (Integrated Development Environment) – Software platform used to write, edit, compile, and debug code.

2. Project Management Plan

2.1. Project Organization

To keep track of our project and the many changes made to it, we used Github.com, an online repository site, to upload our project and note specific changes made with each upload. This was an ideal setup for managing this project, because all the communication in this team was done online. Physical transferring of project data would have been far more difficult, given these circumstances.

2.2. Lifecycle Model Used

The lifecycle model used is the waterfall model. All the requirements for the project were made before development, and documentation of changes made to the project was written down at each stage.

2.3. Risk Analysis

Missed meetings (Risk 1)	**Emergency causing suspended app development for extended time (Risk 2)**	**Project behind schedule (Risk 3)**	**Features costing more development time than intended (Risk 4)**
Risk chance: Moderate	Risk chance: Low	Risk chance: Moderate	Risk chance: Moderate
Impact: Moderate	Impact: High	Impact: Low/Moderate	Impact: Low/Moderate

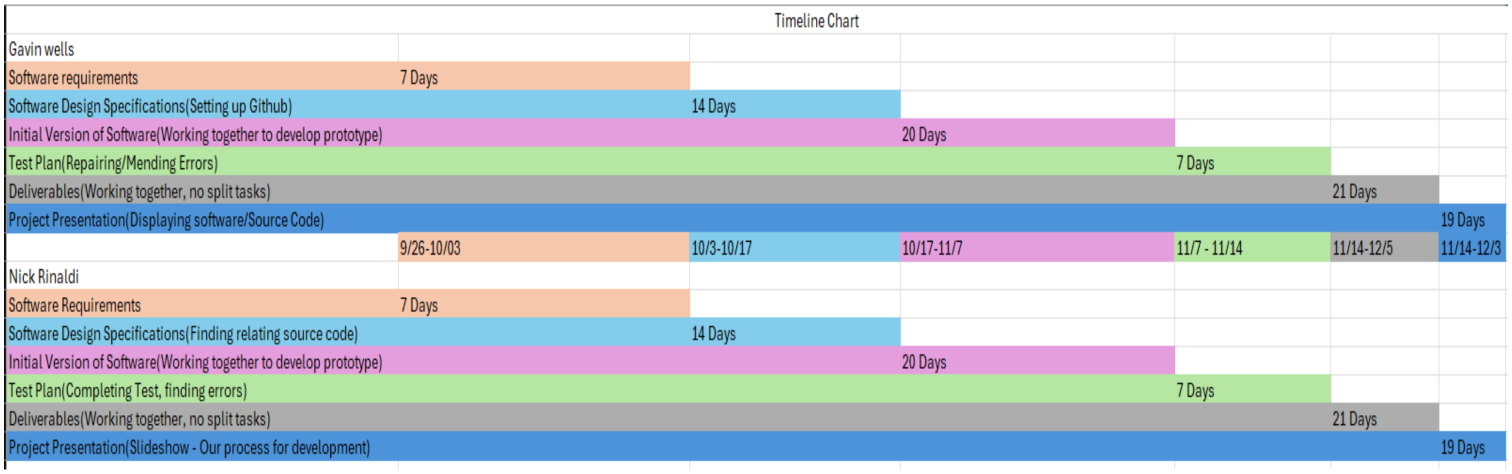
(Fig. 1) Risk Analysis Table

2.4. Hardware and Software Resource Requirements

The hardware required to use this software is a Windows computer, but no other software other than this application is required to run it. For multiple users, separate Windows computers will need to be used between users, as there is no login functionality.

2.5. Deliverables and schedule

The deliverables and the schedule for them are outlined in the following timeline chart (Fig. 1):



Timeline Chart

Gavin wells						
Software requirements	7 Days					
Software Design Specifications(Setting up Github)		14 Days				
Initial Version of Software(Working together to develop prototype)			20 Days			
Test Plan(Repairing/Mending Errors)				7 Days		
Deliverables(Working together, no split tasks)					21 Days	
Project Presentation(Displaying software/Source Code)						19 Days
	9/26-10/03	10/3-10/17	10/17-11/7	11/7 - 11/14	11/14-12/5	11/14-12/3
Nick Rinaldi						
Software Requirements	7 Days					
Software Design Specifications(Finding relating source code)		14 Days				
Initial Version of Software(Working together to develop prototype)			20 Days			
Test Plan(Completing Test, finding errors)				7 Days		
Deliverables(Working together, no split tasks)					21 Days	
Project Presentation(Slideshow - Our process for development)						19 Days

(Fig. 2) Deliverables Timeline Chart

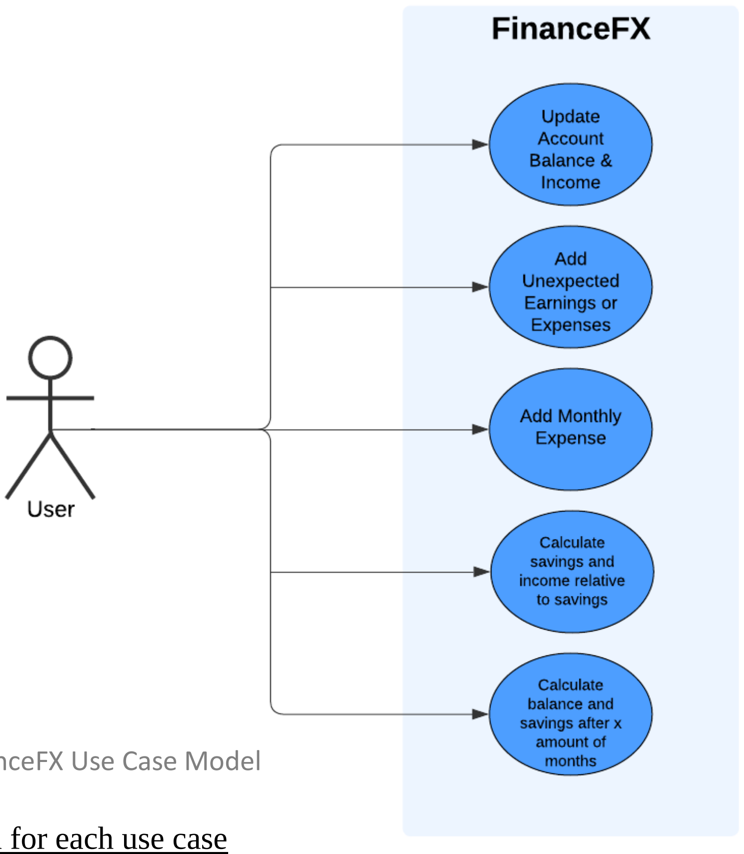
3. Requirement Specifications

3.1. Stakeholders for the system

The stakeholders are Professor Fu, Gavin Wells, and Nicholas Rinaldi.

3.2. Use cases

3.2.1. Graphic use case model


FinanceFX
Update Account Balance & Income
Add Unexpected Earnings or Expenses
Add Monthly Expense
Calculate savings and income relative to savings
Calculate balance and savings after x amount of months
User

(Fig. 3) FinanceFX Use Case Model

3.2.2. Textual Description for each use case

1. Update Account Balance & Income: User inputs an integer or decimal value into the text fields for both the Account Balance and Income update text fields in the FinanceFX application Account Overview scene and presses the corresponding buttons next to them to set the user's account balance and monthly income (after taxes).

2. Add Unexpected Earnings or Expenses: User inputs an integer or decimal value into the text fields for both the Unexpected Earnings or Unexpected Expenses text fields and presses the corresponding buttons next to them to either add or subtract the value entered from the user's balance. Unexpected earnings or expenses are meant to represent any money the user gets or loses separately from their income or monthly expenses. These functions can be found in the Account Overview and Expenses scene, respectively.

3. Add Monthly Expense: User inputs values into 4 separate text fields: Type, Name, Cost, and Day Due. The values entered are of types String, Double, and Integer, respectively (Type and Name are both String). These fields are all meant to represent a single monthly expected cost, and they can be found in the Expenses scene. After pressing the corresponding "Add Monthly Expense" button, the cost will be entered into the expenses table (found in the same scene) with each of its values.

4. Calculate savings and income relative to savings: User inputs a percentage of their income to save each month into the Percent of income saved (monthly) text field and presses the corresponding Calculate button next to it to generate and see the amount of their earnings that will be saved each month and what their monthly earnings will be after saving that much. This function can be found in the Calculations scene.

5. Calculate balance and savings after x amount of months: User inputs an amount of months that they would be saving their money for in the Months text field and presses the corresponding Calculate button next to it to see what their future balance will be in that amount of time and how much they will have saved. This feature can be found in the Calculations scene.

3.3. Rationale for your use case model

Each one of the use cases listed in the model represents the core features of the application. There are a couple of non-essential functions of the application not listed in the model, which are the Savings vs. Months graph and the "Project savings if invested in S&P 500" button. The graph was not included because the user might not use the graph if they only wanted to see a specific number of months, whereas the graph shows the amount saved at each point up to that point. The "Project savings if invested in S&P 500" button is something that is a convenient feature but is also something that might be too specific of a use-case to concern the general userbase of the application. One other reason both were not included is because both do not have user input fields specific to their use and they are secondary outputs to the input fields they rely on.

3.4. Non-functional requirements

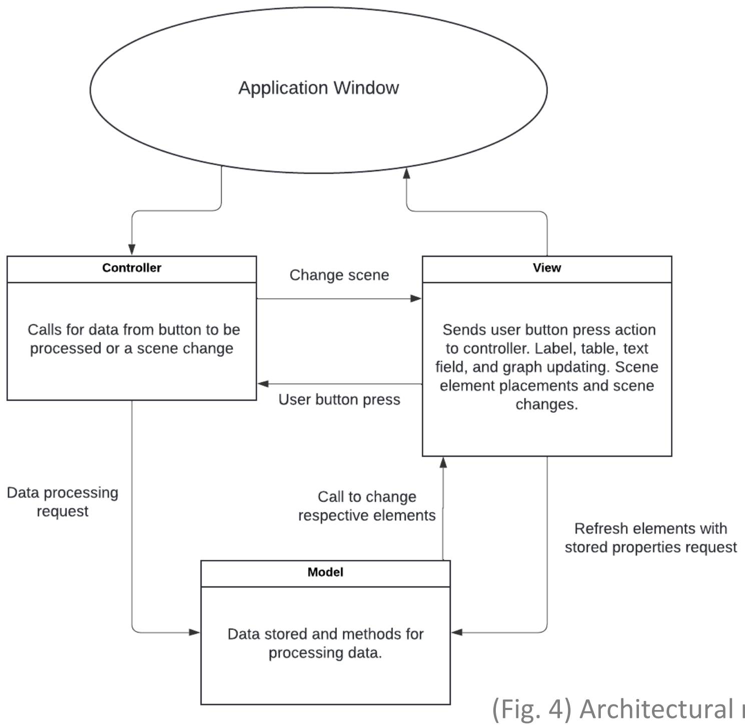
Non-functional requirements of the application include appropriate proportionality of UI elements, use of non-conflicting colors in design, and a max limit of 2 to 3 images per scene to keep the scenes from becoming too cluttered.

4. Architecture

4.1. Architectural style(s) used

The architectural style used in this application is the Model-View-Controller style.

4.2. Architectural model (includes components and their interactions)


Application Window
Controller
Calls for data from button to be processed or a scene change
Change scene
User button press
View
Sends user button press action to controller. Label, table, text field, and graph updating. Scene element placements and scene changes.
Data processing request
Call to change respective elements
Refresh elements with stored properties request
Model
Data stored and methods for processing data.

(Fig. 4) Architectural model of FinanceFX

4.3. Technology, software, and hardware used

JavaFX was used as the language for programming this application. NetBeans and Eclipse were the IDEs. Two separate Windows laptop computers were used to develop and test the application.

4.4. Rationale for your architectural style and model

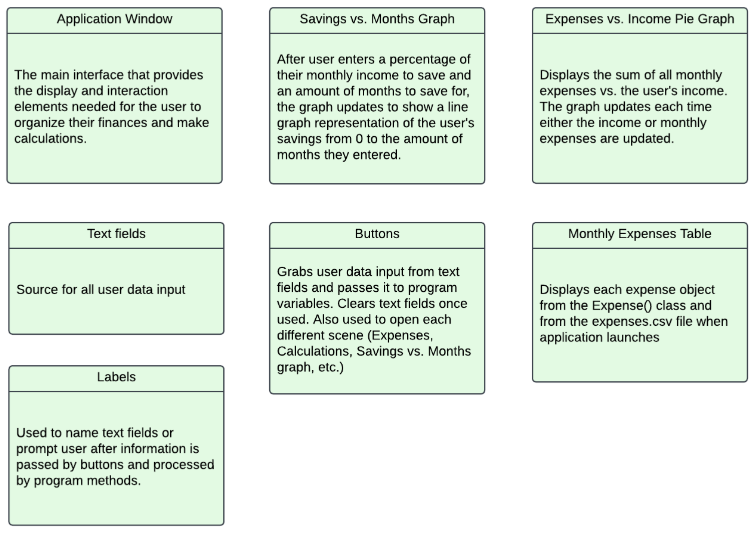
The Model-View-Controller style was the best choice for FinanceFX, because it is an application that primarily functions to receive user input through text fields and buttons (Application Window to Controller, then View to Controller) and to display changes made to the application model (user balance, income, etc.) (Controller to Model) with that input using the application displays (labels, graphs, etc.) (Model to View, then View to Application Window).

5. Design

5.1. User Interface design

There are 3 scenes for this JavaFX application: The Account Overview, Expenses, and Calculations scenes. These 3 scenes are essentially the different ‘pages’ that the user can go between to access relevant features. The Account Overview scene contains labels, text fields, and buttons that allow the user to see and manipulate the user’s starting balance and income. The Expenses scene contains the same element types that allow the user to add or remove monthly costs as well as unexpected costs, and there is a table for viewing each monthly cost added. The Calculations scene contains labels, text fields, and buttons that allow the user to see the user’s income and balance (future and present) after setting a percentage of their monthly income to save. There are also two graphs, “Savings vs. Months” and “Expenses vs. Income,” that allow the user to view their savings, income, and expenses from a visually proportional point of view. The “Savings vs. Months” graph has a button that must be clicked after setting a percentage of income and an amount of months in order to be viewed.

5.2. Components design (static and dynamic models of each component)



Application Window

The main interface that provides the display and interaction elements needed for the user to organize their finances and make calculations.

Savings vs. Months Graph

After user enters a percentage of their monthly income to save and an amount of months to save for, the graph updates to show a line graph representation of the user's savings from 0 to the amount of months they entered.

Expenses vs. Income Pie Graph

Displays the sum of all monthly expenses vs. the user's income. The graph updates each time either the income or monthly expenses are updated.

Text fields

Source for all user data input

Buttons

Grabs user data input from text fields and passes it to program variables. Clears text fields once used. Also used to open each different scene (Expenses, Calculations, Savings vs. Months graph, etc.)

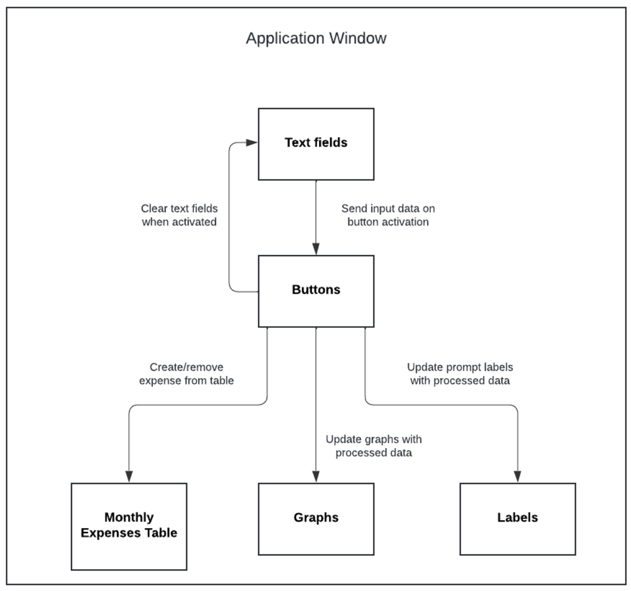
Monthly Expenses Table

Displays each expense object from the Expense() class and from the expenses.csv file when application launches

Labels

Used to name text fields or prompt user after information is passed by buttons and processed by program methods.

(Fig. 5) Static component model


Application Window
Text fields
Clear text fields when activated
Send input data on button activation
Buttons
Create/remove expense from table
Update prompt labels with processed data
Update graphs with processed data
Monthly Expenses Table
Graphs
Labels

(Fig. 6) Dynamic component model

5.3. Rationale for your detailed design models

The static model of FinanceFX’s components (Fig. 5) shows each of the interactive components and their respective output components. Provided with each is a description of their function. The dynamic model (Fig. 6) presents as a mostly hierarchical model, because of the way user input transfers through each component. First, the user must launch the program to see the application window, which holds all the components and the scenes they are in. Text fields are the first point of entry for user data, which are taken once the button associated with them is pressed. From that point, the data is processed by the button, and the methods/class methods in them, and passed onto the display components: Prompt labels, graphs, and the Monthly Expenses Table. The only reason the dynamic model is not a traditional hierarchical one is because the button does influence the component directly above it in one way (it clears them once pressed).

5.4. Traceability from requirements to detailed design models

Each of the components from the design models represent a part of each use case in the requirements section 3.2.2. Each use case has an associated text field, button, prompt label, and/or a graph. The design models showcase how each component involved in those use cases are impacted by one another. As shown in the dynamic model, the order of components involved in each use case generally follow the order of: Text field, button, display.

6. Test Management

6.1. A complete list of system test cases

Input features that were tested are as follows: Update of Account Balance, Update of Monthly Income, Add unexpected earnings, Add Monthly Expense (Type, Name, Cost, Day due), Add unexpected losses, Calculating income after saving a user entered percent of it, calculating the user's balance after a user entered number of months, and all immediate user information (i.e. balance, income, and monthly expenses) are saved to file after entering each respectively.

6.2. Traceability of test cases to use cases

Each of the test cases that were tested correspond to each use case in section 3.2.2., respectively, except for the saving of user information. This is because the saving of user information is not a specific user use case, but rather a feature to be automatically handled by the application. After a user inputs any user pertinent information, the application should save it to a text file and be available when the application is launched again.

6.3. Techniques used for test case generation

Our IDE of choice (NetBeans and Eclipse) was used as a debugging tool and a means to run the project while testing for errors. We tested each of the listed fields from 6.1 with erroneous entry data (i.e. incorrect data types, incorrect characters, etc.) and checked if the safeguards that were put in place to prevent and address them to the user functioned properly. For the requirement of saving the information to file, we made sure that the file holding user information was updated any time we updated information relevant to the file in the application. When the application closed, we checked that the information saved was available when it was launched again.

6.4. Test results and assessments (how good are your test cases? How good is your software?)

The results of our tests revealed a few issues. However, after fixing those issues, our software functioned as desired, and there are no other major problems that prevent the user from using each feature successfully. While there may be issues we are not aware of, we have tested for any we could think of, and, as a result, we are satisfied with the current state of the application.

6.5. Defects reports

Defects that were found and corrected were: Expense table not removing only entry, users being able to enter negative months for savings, users being able to enter a savings percentage equal to or less than zero and equal to or more than 100, and users being able to enter a monthly expense without a name or type. The expense table would not let the user remove the only entry from file because the code removed it as an object before trying to search the file for it, which caused a null error. The negative months and percentages greater or less than the range of 0 to 100 were fixed with conditional statements that prevented those values from being entered and told the user to enter a value respecting those conditions. Conditional code was also used to prevent the user from entering a monthly expense without at least a name, so that there is at least one way to identify the expense against the others.

7. Conclusions

7.1. Outcomes of the project (are all goals achieved?)

The development of FinanceFX has been a success, and all fundamental features have been implemented without any major issues. One non-essential feature was also able to be implemented, which was the Expenses vs. Income graph. The implementation of a login feature would have been a nice addition, however, the time needed to implement that feature did not fit the schedule for this application.

7.2. Lessons learned

Through the development of FinanceFX, our team learned how to manage a single project across multiple computers with an online repository and that effective communication is crucial to maintaining organized code. There were a few moments early on when we implemented code before speaking to each other about it, and the result was that code was not incorporated in a manner that respected both of our original intentions for the features it helped produce. With more effective planning and time management, we might have also had the opportunity to complete implementing login functionality. Ultimately, this project has shown us that project management is just as important, if not more so, as project development.

7.3. Future development

After the completion of the core features of this project and the submission of the results to this class, we will spend some time learning how we can implement a login feature to the application, so that multiple users can access the application on one computer without worrying about the security of their information. We might also look into making an online version, so that users can access the application and their information on it through multiple devices, wherever there is internet access.