Fishback Stock and Options Trading Game

Senior Design Project Fall 2017

Team Members:
Caleb Cornett
Chad Stephenson
Melissa Shankle
Mitchell McClure

Customer:
Fishback Management & Research

1. High-Level Design

Our Options Trading game was designed to simulate the trading of Stock Options in an effort to educate its users. With a simple, yet intuitive design, users will be able to begin playing in a virtual stock market by searching for companies, buying options, and viewing their portfolio and past transactions, all with real-time stock data for each company. To accomplish all of these feats, the game will have three layers working in unison: User Interface, Data, External. Visually this will be shown in Figure 1.1 as a High Level Architecture Diagram to display the structure of the back end. Later on in the document more focused and detailed diagrams will be used to demonstrate individual components and how they interact.

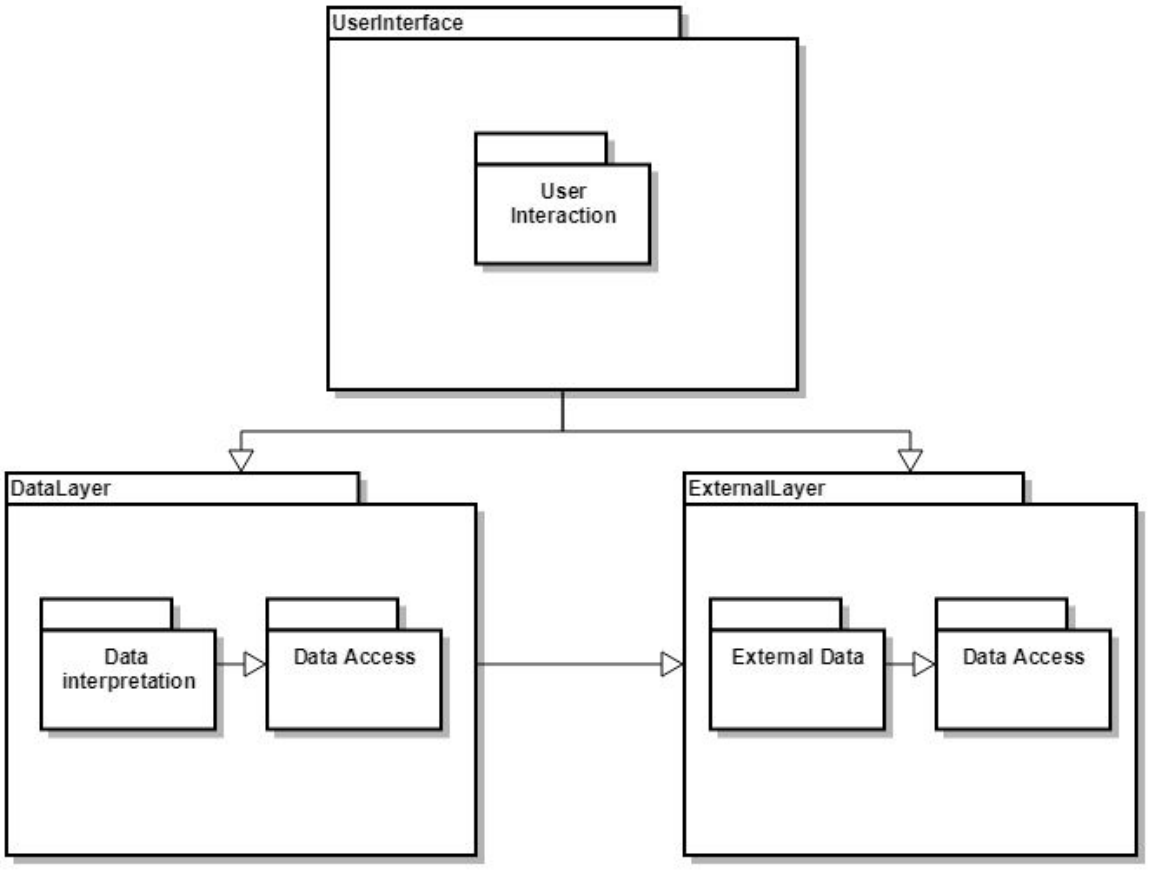


Figure 1.1: High Level Architecture Diagram of Options Trading App

The High Level structure of our game consists of three layers: User Interface, Data, and External. Each layer interweaves with each other to provide the user with an experience that educates them on options trading. The user will pick among screens and depending on the screen, the User Interface layer will take the data structures extracted from the Data or External layers and display them in a readable and appealing way. User Input will be broken up into the

four main gameplay loops: searching for a company, buying options, viewing portfolio and transaction history. Depending on the screen and actions the user has taken, the presentation of information and the information itself will change. However, for any useful information to be displayed, it must be retrieved from either the Data or External Layers.

The data layer consists of Data Interpretation, which feeds off of Data Access. As long as the user has been authenticated with Game Center ID, the Data Access layer will allow the system to interact with the database and the External Layer. In the Data Interpretation layer, it will interpret information gathered from the external layer that is relevant when a user completes a transaction and stores it in a data structure. The Data Interpretation layer also displays the data structures and External data in a way that the user can then interact with. Depending on the user's action and what screen it is on, different data will be retrieved. For example, on the portfolio page, Available Funds and owned options will be displayed. After the user interacts and performs further transactions with the data, such as using an option to buy more stock, the data will be updated in the Database. The transaction data and other data stored in the database will be based off initial data pulled from the External Layer for a company's page.

All of the data and information displayed in the app stems from financial data extracted in the External Layer. When a user views a company page, financial data is retrieved to display relevant information to the user. If the user decides with this information to purchase an option, the necessary data is stored or updated within the database. The external data is accessed through third-party API's such as Google Finance. To summarize, the User Interface layer shows information gathered from the External Layer on Company Pages, other pages show information retrieved from the Database which was also stored from data retrieved from the External Layer.

2. Detailed Design

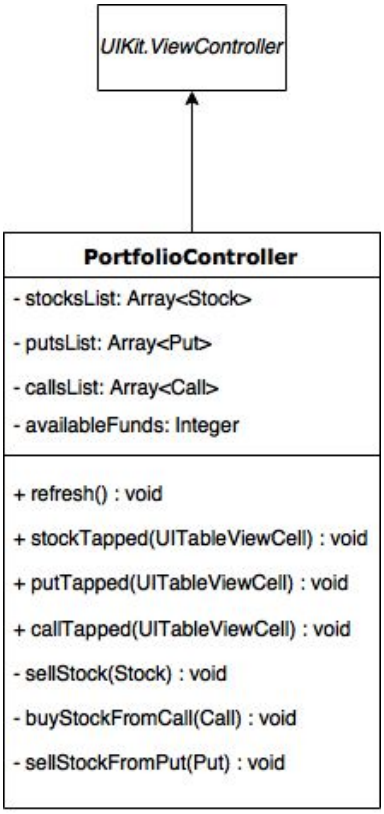
2.1 Detailed Class Diagram (Caleb)

The app implements the Model-View-Controller (MVC) design pattern, so most classes in the program will fall into one of those three categories. "View" classes will provide the app with user interface elements. "Model" classes will contain information and pull from/push to the database and financial service. Finally, "Controller" classes act as liaisons between the View and the Model, updating the UI and notifying the Model of any changed information. The separation of these components will ensure the program remains reasonably decoupled while providing clear responsibilities for each class. Thankfully, Apple has made it fairly straightforward to use MVC in app development, as evidenced by the built-in and recommended framework of Views and Controllers.

In fact, the iOS developer tools have been developed such that there may be no need for proprietary classes to create View elements. By using Apple's Xcode Interface Builder tool, each screen (or "View") in the app is created in a WYSIWYG environment. This ensures the simple implementation of iOS standards, such as tabbed menu bars, list views, and responsive

design. Given the variety and versatility of the visual editor and built-in UI elements, we do not anticipate the need to create new classes for View elements. Most of the object-oriented design will come with the Controller and Model classes.

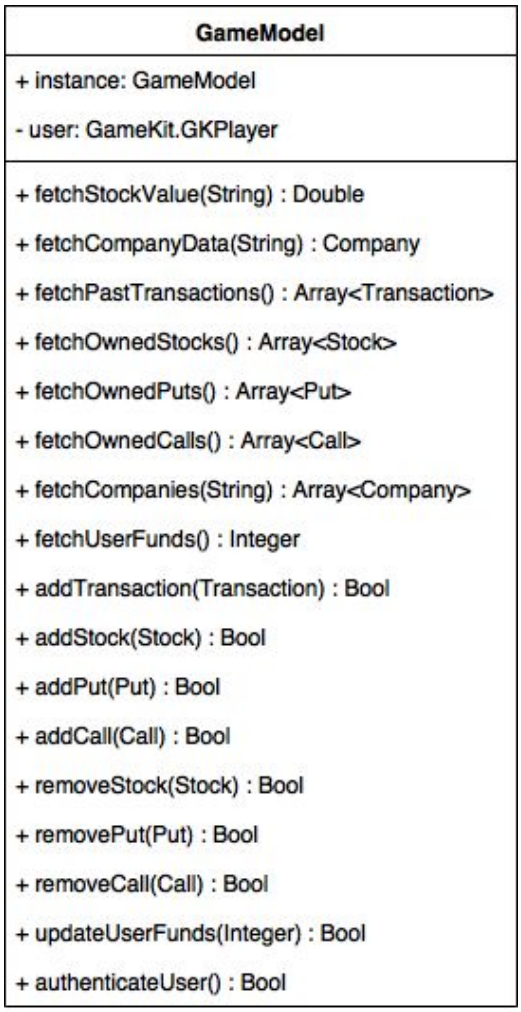
Apple's standard UIKit library includes a basic "ViewController" class, which updates the view, handles user interaction, and performs state changes. Every screen of an iOS app will have a dedicated ViewController subclass that performs the necessary work for that screen. There will be 5 distinct screens in the app (Welcome, Portfolio, Search, Company, History) so each will be associated with a unique class inheriting from UIKit.ViewController.



On the left is a diagram of one such class, PortfolioController. It manages the Portfolio View that displays the user's available funds and all of their owned stocks and options. Accordingly, it has three expandable arrays to hold each type of stock or option: stocksList, putsList, and callsList. (Stock, Put, and Call are simple classes or structs that contain all necessary data about the individual stock item.) The available funds are stored in an integer. Since every purchase or sale involves 100 shares of stock and every stock value is at least $0.01, there can never be a non-integer value for user funds or transaction amounts. These variables are directly controlled by the class's methods.

The PortfolioController methods serve to update the view and Model and respond to user input. The *refresh()* method updates the class's variables and the view with the latest information from the Model. The *stockTapped, putTapped,* and *callTapped* methods are event handlers for user interaction with the list elements for stocks, puts, and calls. The final three methods are called within the event handlers to perform actions on the stocks, calls, and puts. If a stock is sold, its value is added to the amount of available funds, it is removed from the stocksList, the Model is updated to remove the stock and change the available funds, and finally the View is refreshed. Buying stock from a call works similarly -- it adds stock, subtracts from funds, and removes the call from the local list and the Model. Selling stock from a put removes stock, increases funds, and removes the stock and call. Together, these methods provide a convenient and comprehensive interface for updating the game state and Model.

While each screen requires a unique View and ViewController, the entire app will share one Model class, called GameModel. This will be a Singleton class that any ViewController subclass can access through a static reference. GameModel is responsible for all communication with the database server, saving information locally, and requesting data from the financial information service.



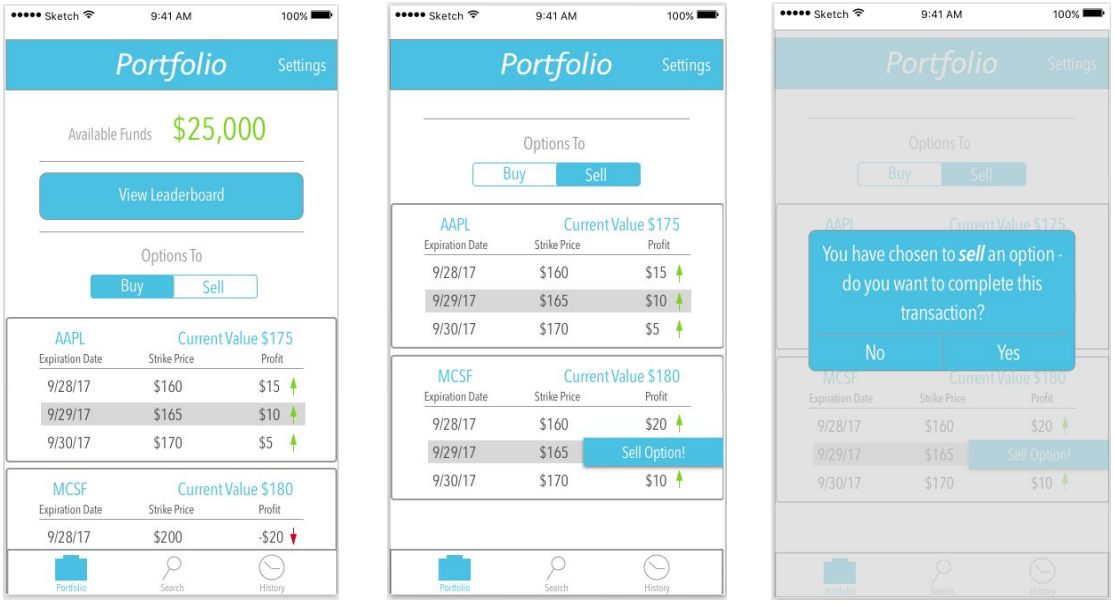
The GameModel class has only two variables. The first is a static reference to its one instance (as part of the Singleton pattern). In the class constructor, if the static "instance" variable has not been set, it will refer to this particular instance of GameModel. The second variable is a private reference to the current user, stored as a GKPlayer object. Since the app uses Apple's GameKit framework, user authentication can be performed easily and the resulting user profile is stored inside the GKPlayer object. This user variable is used to access the correct database information through the class methods.

There are many methods in this class, but they serve primarily two purposes: getting and sending information to/from the database and financial service. The methods with the prefix "fetch" retrieve requested information and return it in a convenient format. The String parameter of *fetchStockValue* and *fetchCompanyData* should be the stock symbol of the company / stock value requested. The parameter of *fetchCompanies* is a query string -- the method will search the financial data for any company names or symbols that match the query.

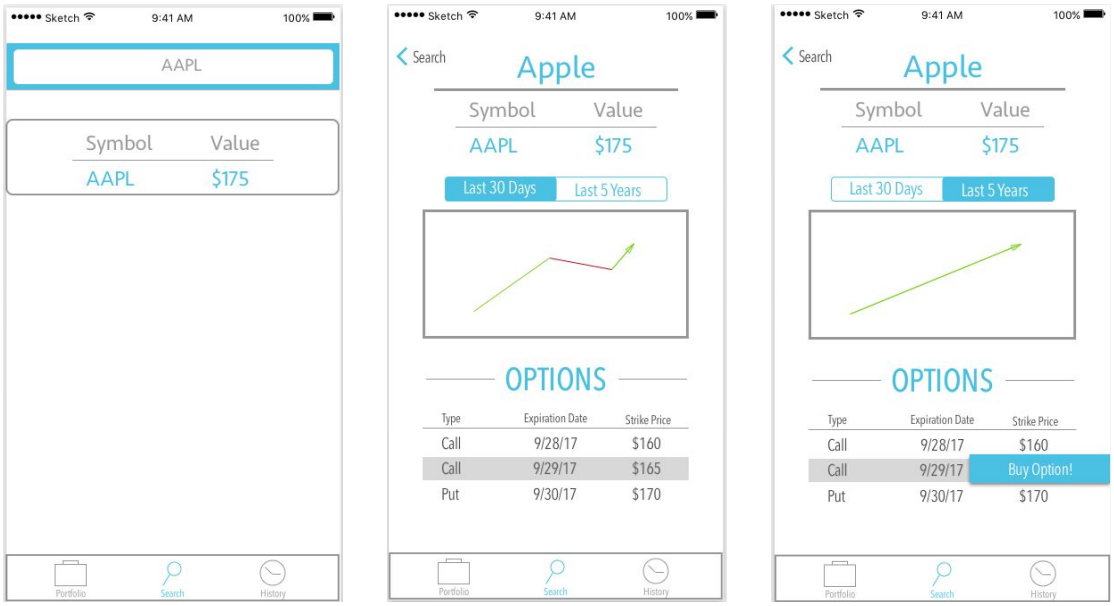
The "add", "remove", and "update" methods update information in the database. The parameters will determine the object to add, the object to remove, or the updated value (only for *updateUserFunds*). For instance, *removeStock(Stock)* will remove the given stock from the database and return whether the operation was successful.

The final method in the class is *authenticateUser()*. This is performed once at the beginning of each game session to ensure the user is connected to GameCenter and the database server. This is where the private "user" variable is assigned.

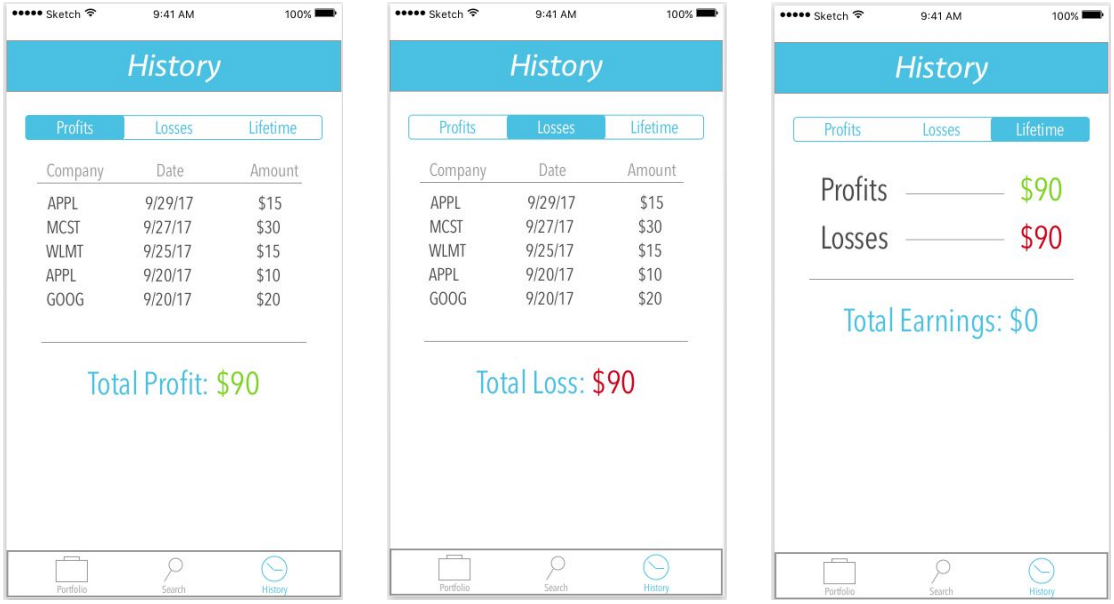
2.2 User Interface Design (Melissa)



This first row of images reflect the main functions for the Portfolio tab. The first one simply shows the layout of information for a user: their available game fund, a button to view the leaderboard, and the options they have bought. The Options To tab switches between options the user can buy and sell which helps the user know that one tab will add to their funds and the other will subtract from it. Each company the user buys options from will have their own cards, shown by the gray box outlines. The information for each company includes the symbol, current stock price, and the options the user has obtained. The option information includes the expiration date, strike price, and profit/loss margin to be made with an accompanying arrow reflecting whether it's a profit or loss. The difference shown in the second image reflects when a user either taps or swipes on an option. A 'Sell Option!' button will appear and become tappable. The third image shows the pop up message a user will receive after tapping the button. It confirms with the user their action before finishing the transaction.



These three images show the Search functionality. The first one shows a typical search page where a search has been started. The proposed search only yielded one result. After tapping a result, a user will see the company page. This page contains necessary company information including the name, symbol, current stock value, historical chart, and options list. The historical chart can be switched between last thirty days and last five years using the switch tab above it. The options list contains the type of option, expiration date, and strike price for each option of the company. The third image shows the same behavior as the portfolio page when a user chooses to buy an option.



The last section illustrates the history tab. Here a user can see the previous twenty transactions for profits and losses. The profits tab will show each option that has produced a profit, which included the company, date of transaction, and profit amount. The second image is

the same except for losses. The third image is the lifetime history which simply calculates whether the user has a positive or negative (or in this case neutral) lifetime earnings.

2.3 Design Patterns

User Interface Patterns (Melissa)

As our app is for iOS, our design is meant to imitate the normal navigation and feel of typical iOS apps. Sketch was used for the design process because it has the ability to create iOS projects that include normal components which are simple to drag and drop. For our overall design, we were going for clean and simple that would appeal to the large majority of users. We went with easy to read fonts and tried to size them as appropriately as possible for each page. Our goal was to create our app as intuitive as possible since our target segment are elderly people with a general lack of tech knowledge.

We chose a light blue shade as the main color because it is a typically gender neutral color, isn't super bright to irritate our older customer segment nor too dull to detract our younger target, and is really just an appealing color (since Facebook, Twitter, and LinkedIn all use a form of blue). Our supporting colors were dark gray for the majority of text, white for titles, green for profit amounts and red for loss amounts. To help our users easily discern between them winning profit and losing money we chose to use the green and red which are patterns seen often to represent good and bad. In addition to the colors, we aided them with up or down arrows for those who can't see the difference between green and red.

The navigation is through the bottom menu, which has three tabs: Portfolio, Search, and History. The three tabs represent the main concerns for a user. Each tab can have multiple pages, but the flow always begins with the tab landing page. There will be a back button on the search results page so the user can go back to the searching page to look up another company. Beyond that instance, the user will switch content within the main navigation tab by switching an on-page tab. For example, on the portfolio page the user can change between Buy and Sell simply by changing the tab position. Lastly, on the Portfolio page we have a settings button space reserved, but we have not decided what will go into there or if it will exist. It seems like for our MVP it will be unnecessary, so it will most likely be a component we visit later if time permits.

Programming Patterns (Caleb)

The app makes use of the Model-View-Controller and Singleton design patterns.

1. Model-View-Controller (MVC) is a structural paradigm that separates concerns into three distinct types of classes: Model (fetching, storing, and sending data), View (displaying UI elements), and Controller (reacting to user input, updating the View and Model as needed). MVC is discussed in more detail in the Detailed Class Diagram section.
2. Singleton is a design pattern in which there is (and can only be) one instance of a particular class. The class has a static reference to its single instance, so any object in the program can easily access it. This will be used for our Model class, since it is imperative the app has only one means of accessing data. If there were multiple

instances of the Model, it would result in inconsistent and incorrect information between each instance and the remote database.

3. Testing (Chad, Mitchell)

Since we are creating a video game for our customer, most of our functionality revolves around what the user can do within the game. With that in mind, we have created test cases in order to check that all functions we implemented work correctly. We will use the test cases to determine if the code that we have written does what it was intended to do. Figure 3.1 contains our test cases, sorted by a case number (and letter to differentiate related cases), the test case, the conditions that need to be satisfied for the test to be considered a pass, and the conditions that constitute a fail.

Case No.	Test Case	Pass Conditions	Fail Conditions
1	User creates in-game account by entering their Apple ID and password to sync to GameCenter when prompted	GameCenter prompt is displayed, in-game account is created when user inputs correct Apple ID and password	GameCenter prompt is not displayed, or in-game account is not created when user inputs correct Apple ID and password
2	Stocks and options owned are displayed when on the portfolio page	All stock and options that are owned are displayed in their respective “Buy” or “Sell” tabs when the portfolio page is accessed	None or some of stocks and options that are owned are displayed in their respective “Buy” or “Sell” tabs when the portfolio page is accessed
3	Leaderboard is displayed to compare user’s score to other users playing the game	Leaderboard is displayed when “Leaderboard” button is pressed, and information in the leaderboard is correct and up to date	Leaderboard is not displayed when “Leaderboard” button is pressed, or the information in the leaderboard is incorrect or outdated
4	User is able to search for company based on company name or symbol	The company that is output from the user search corresponds to the name or symbol that the user	The company that is output from the user search does not correspond to the name

		searched for, or no company is output if user searched for an invalid company name or symbol	or symbol that the user searched for, or a company is output if user searched for an invalid company name or symbol
5a	Current stock price and options that can be bought are displayed on a company's page	Company's page is displayed with its correct current stock price and available options to buy	Company's page is displayed with an incorrect stock price, or with out all available options to buy
5b	A graph showing the company's stock price as far in the past as 5 years is displayed on a company's page	Graph is rendered showing correct stock price research information as far as five years in the past	Graph fails to render, or shows incorrect stock price information, or does not show information as far as five years in the past
6	Stock options are able to be bought from the list of options available on a company's page	After pressing the "Buy" button by an option, the user is prompted to follow through with the transaction or cancel. If canceled, option is not bought. If accepted, the cost of the option is deducted from the user's available funds, and the option is stored in the user's portfolio	After pressing the "Buy" button by an option, the user is not prompted, or if prompted and canceled, option is still bought, or if accepted the cost of the option is not deducted from the user's available funds, or the option is not stored in the user's portfolio
7a	Stock controlled by *put* options can be sold from the portfolio	The amount of stocks that are part of the option are sold successfully at the strike price agreed upon, and that money is added to the user's available funds. The transaction will also be added into the history page	The amount of stocks that are part of the option are not sold successfully, or the stocks are not sold at the strike price agreed upon, or that money is not added to the user's available funds. Or the transaction is not stored within the history page

7b	Stock controlled by *call* options can be bought from the portfolio	The amount of stocks that are part of the option are bought successfully at the strike price agreed upon, and that money is deducted from the user's available funds	The amount of stocks that are part of the option are not bought successfully, or the stocks are not bought at the strike price agreed upon, or that money is not deducted from the user's available funds
7c	Owned stock can be sold directly at current value from the portfolio	The amount of stocks to be sold directly are sold successfully at the current stocks' value, and that money is added to the user's available funds. The transaction is stored in the user's History	The amount of stocks to be sold directly are not sold successfully, or the stocks are not sold at the current stocks' value, the money is not added to the user's available funds, or the transaction is not added to the user's history
8a	A list of all past profits can be viewed on the Profits tab on the History page	A list of all stocks sold for a profit is displayed when the Profits tab is pressed, along with the amount of profit each stock made and the total amount of profits made throughout the user's game lifetime	A list of all stocks sold for a profit is not displayed when the Profits tab is pressed, or the amount of profit each stock made is not displayed or is incorrect, or the total amount of profits made throughout the user's game lifetime is not displayed or is incorrect
8b	A list of all past losses can be viewed on the Losses tab on the History page	A list of all stocks sold for a losses is displayed when the Losses tab is pressed, along with the amount of money each stock lost and the total amount of losses had throughout the user's game lifetime	A list of all stocks sold for a loss is not displayed when the Losses tab is pressed, or the amount of money each stock lost is not displayed or is incorrect, or the total amount of losses had throughout the user's

			game lifetime is not displayed or is incorrect
8c	The total earnings of the user can be viewed on the Lifetime tab on the History page	The total profits, total losses and total earnings are displayed when the Lifetime tab is pressed, and the amounts are all correct	The total profits, total losses and total earnings are not displayed when the Lifetime tab is pressed, or the amounts are incorrect
9	The user clicks on one of the three buttons at the bottom of the screen that can take you to portfolio, search, or history	The correct page is displayed after pressing	The button does nothing or the wrong page is displayed

Figure 3.1

We will use the test cases from Figure 3.1 once we have written our code to test the code's functionality. If all of the pass conditions for a test case are satisfied, we will consider the test a pass and the code to be functional. If one or more of the pass conditions for a test are not satisfied, we will consider the test a fail and will rewrite the code. Then we will use the same test case on the rewritten code.

4. Review (Group)

1. The app UI design originally lacked a "Back" button from a Company page to the Search page.
2. The app UI design originally lacked alternating colors for rows in tables, making it hard to determine the physical boundaries between each element in the table.
3. There were multiple fonts used in the document body text, so they were standardized to one font type and size.
4. There was some unclear wording in the High-Level Design section.
5. There were 4 test cases that needed clarification
6. There was one test case that needed to be added.
7. There were 3 minor typos in Section 2.2.

5. Metrics

Complexity of Overall System (Caleb)

The maximum depth of any inheritance tree in our code will be 2.

Product Size (Chad)

We have planned our game based on three user stories. We have a total of 13 test cases, and 12 classes planned.

Product Effort (Mitchell)

	Hours	Word Count
Caleb Cornett	10	1141
Chad Stephenson	8	1180
Melissa Shankle	10	792
Mitchell McClure	6	613

Defects (Caleb)

There were 12 noted defects that have been addressed.

---

6. Web Page and Developer Notebook (Melissa)

Our team chose to use WordPress to keep track of our developer logs. Each team member has their own page in addition to the team page. Member pages are used to record decisions and actions of each member while the team page is for group decisions and meeting notes. As the project progresses, more pages will be created to house documents such as the project plan and this architecture assignment. Our site also includes our contact information and the project overview.

Our site can be found at https://stockexchangegame.wordpress.com/

Word Count as of October 1:

- Caleb - 274
- Chad - 291
- Melissa - 1987
- Mitchell - 788