

# Rags to Riches

## Software Engineering

### Team 5:

Alejandro Aguilar  
Arjun Ohri  
Deep Patel  
Kartik Patel  
Elisa-Michelle Rodriguez  
William He  
Bryan Benalcazar

March 5, 2017

[Github Page](#)

# TABLE OF CONTENTS

<b>1 INTERACTION DIAGRAMS</b>	<b>6</b>
<b>2 CLASS DIAGRAMS AND INTERFACE SPECIFICATION</b>	<b>11</b>
2.1 Class Diagram	11
2.2 Data types and Operational Signatures	13
2.3 Traceability Matrix	22
<b>3 SYSTEM ARCHITECTURE AND SYSTEM DESIGN</b>	<b>23</b>
3.1 Architectural Styles	23
3.2 Identifying Subsystems	24
3.3 Mapping Subsystems To Hardware	26
3.4 Persistent Data Storage	26
3.5 Network Protocol	26
3.6 Global Control Flow	27
3.7 Hardware Requirements	27
<b>4 ALGORITHMS AND DATA STRUCTURES</b>	<b>28</b>
4.1 Algorithms	28
4.2 Data Structures	28
<b>5 USER INTERFACE DESIGN AND IMPLEMENTATION</b>	<b>29</b>
<b>6 DESIGN OF TEST</b>	<b>38</b>
6.1 Use Cases and Unit Testing	38
6.2 Test Coverage	47
6.3 Integration Testing Strategy	48
<b>7 PROJECT MANAGEMENT</b>	<b>49</b>
7.1 Merging Contributions for Individual Team Members	49
7.2 Project Coordination and Progress Report	49
7.3 Plan of Work	50
7.4 Breakdown of Responsibilities	51
<b>8 References</b>	<b>52</b>

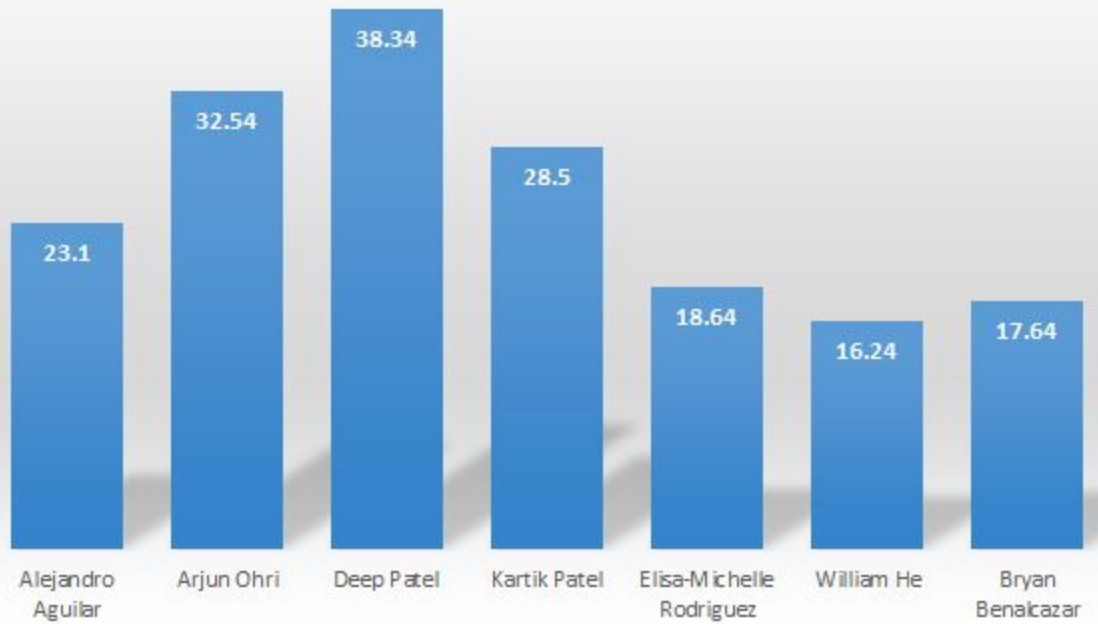
## 0 INDIVIDUAL CONTRIBUTIONS BREAKDOWN

---

### 0.1 Responsibility Matrix

	Team Members						
	Alejandro Aguilar	Arjun Ohri	Deep Patel	Kartik Patel	Elisa-Michelle Rodriguez	William He	Bryan Benalcazar
Sec.1: Interaction Diagrams (30 points)	2%	15%	15%	15%	22%	15%	16 %
Sec.2: Class Diagrams and Interface interactions (10 points)	0%	25%	26%	0%	8%	26%	15%
Sec.3: System Architecture and System Diagram (15 points)	35%	3%	27%	35%	0%	0%	0%
Sec.4: Algorithms and Data Structures (4 points)	0%	30%	40%	0%	0%	15%	15%
Sec.5: User Interface and Implementations (11 points)	0%	55%	5%	40%	0%	0%	0%
Sec.6: Design of Tests (12 points)	10%	26%	16%	0%	16%	16%	16%
Sec.7: Project Management (18 points)	25%	25%	25%	25%	0%	0%	0 %

## Report 2 Point Allocation



# 1 INTERACTION DIAGRAMS

---

The following interaction diagrams will showcase the system interactions in our software, which demonstrates where the software is most prominent. For each particular use case, we will outline the interactions among the controllers and databases. Also, we will analyze multiple cases in which the systems will handle different scenarios, such as success and failure scenarios. Overall, using the database and controllers are vital to the functionality of the application and success of each use case.

## UC-1 Register

The Register use case begins with creating a new account for a first time player or logging into an existing account for a returning player. Every user needs an account in order to play the game. Once the first time player registers a new account, his or hers information is stored in the Account Database and can be edited through the Account View. The Account Database is used to store account information. This is to allow users to login and ensure there isn't duplicate account usernames. If there is duplicate information then the system will display an error message.

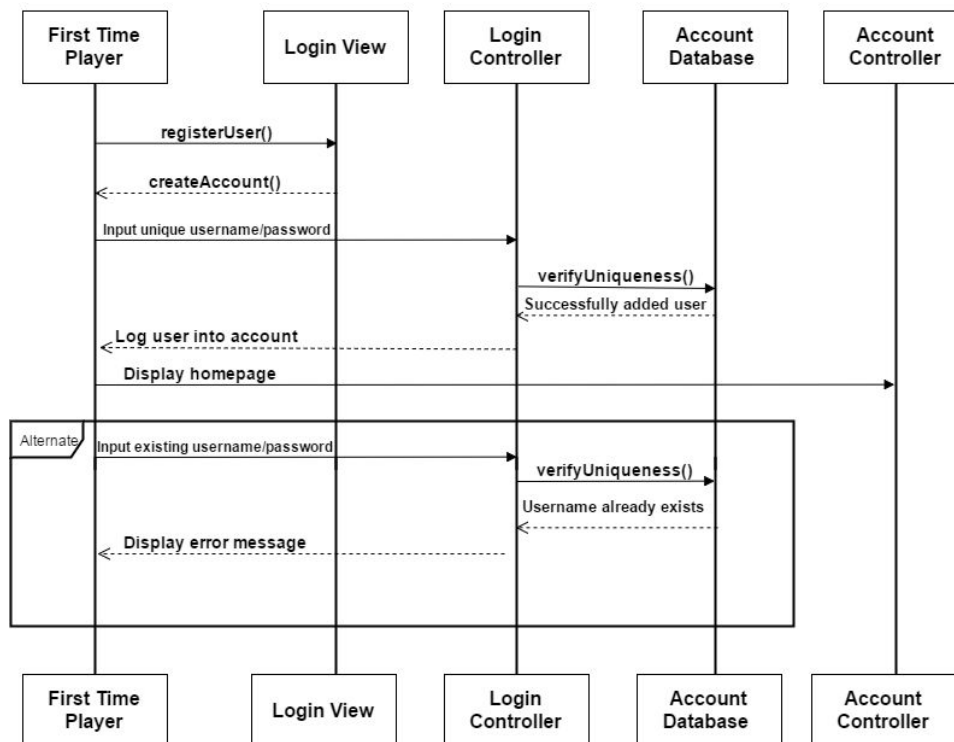


Figure 1.1: UC-1 Register

## UC-5 FindPublicGame

The FindPublicGame use case begins with trying to find a public game. Once the player is on the Home Page, the user chooses to find a public game by clicking a button shown on the Account View. The user is then put into the Game Queue where they need to wait until the queue is filled up with 3 additional players, at which point the Public Game Controller creates the game. The user is then shown the view associated with a game being found. If at any point the user wants to stop finding a game, they can press on the “Close” button on their screen. The game will stop finding a game and leave the queue. The application will go back to main homepage.

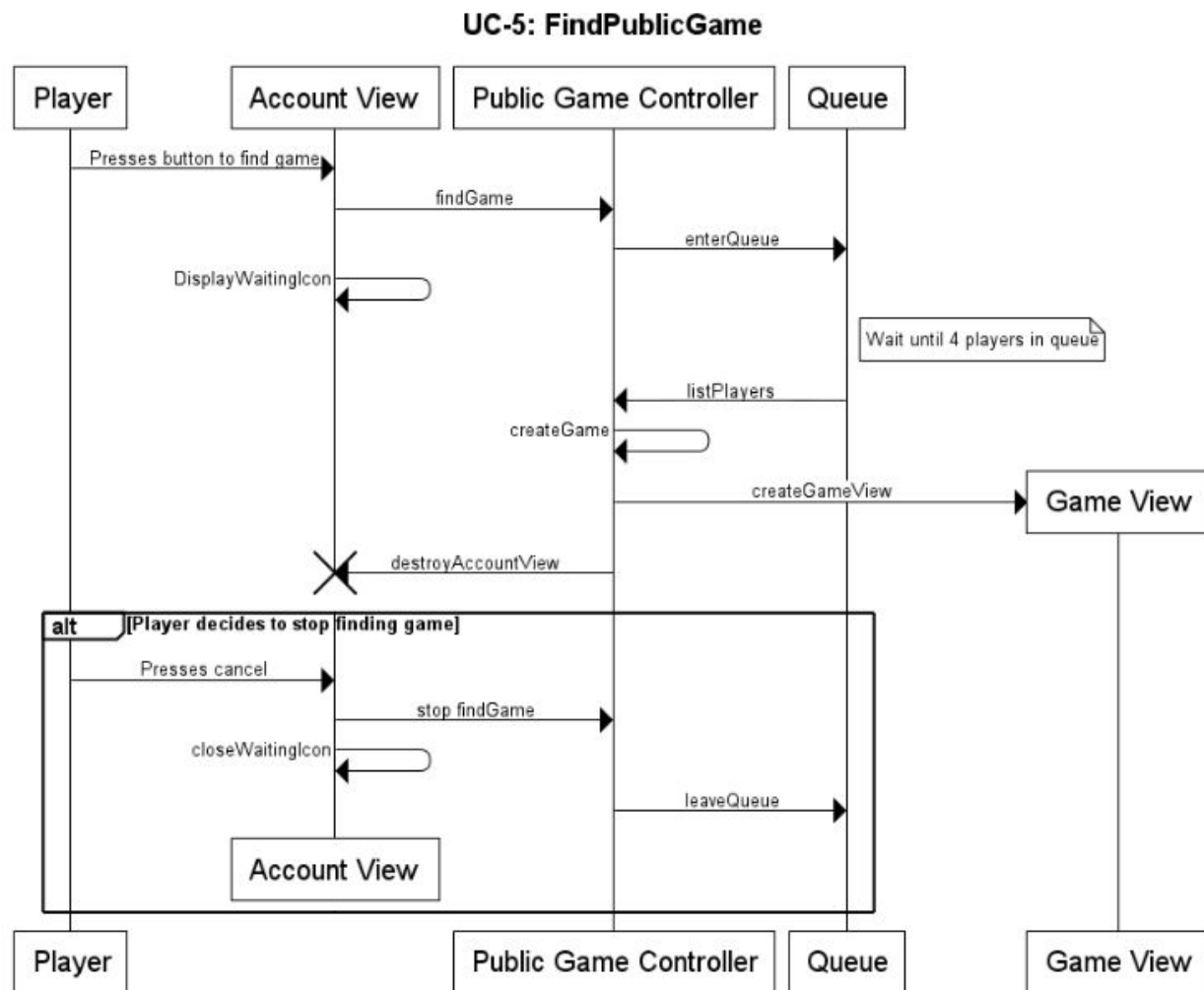


Figure 1.2: UC-5 FindPublicGame

Account view and public game controller are loosely coupled. This is great for our project because it allows us to implement the “Cancel” option very easily. For example, because findGame is a method which only does 1 task, we can choose to end that method and not have it negatively affect other parts of the project.

## UC-7 JoinPrivateGame

The JoinPrivateGame use case begins with trying to join a private game. Once the player is on the Home Page, the user chooses to join a private game by clicking a button shown on the Account View. The user is then asked to enter the corresponding passcode to a private game. If the user enters the correct passcode and the game isn't full, the Private Game Controller adds them to the game and updates their view. If either the user enters the wrong passcode or the game is full, the Private Game Controller displays an error message to the user.

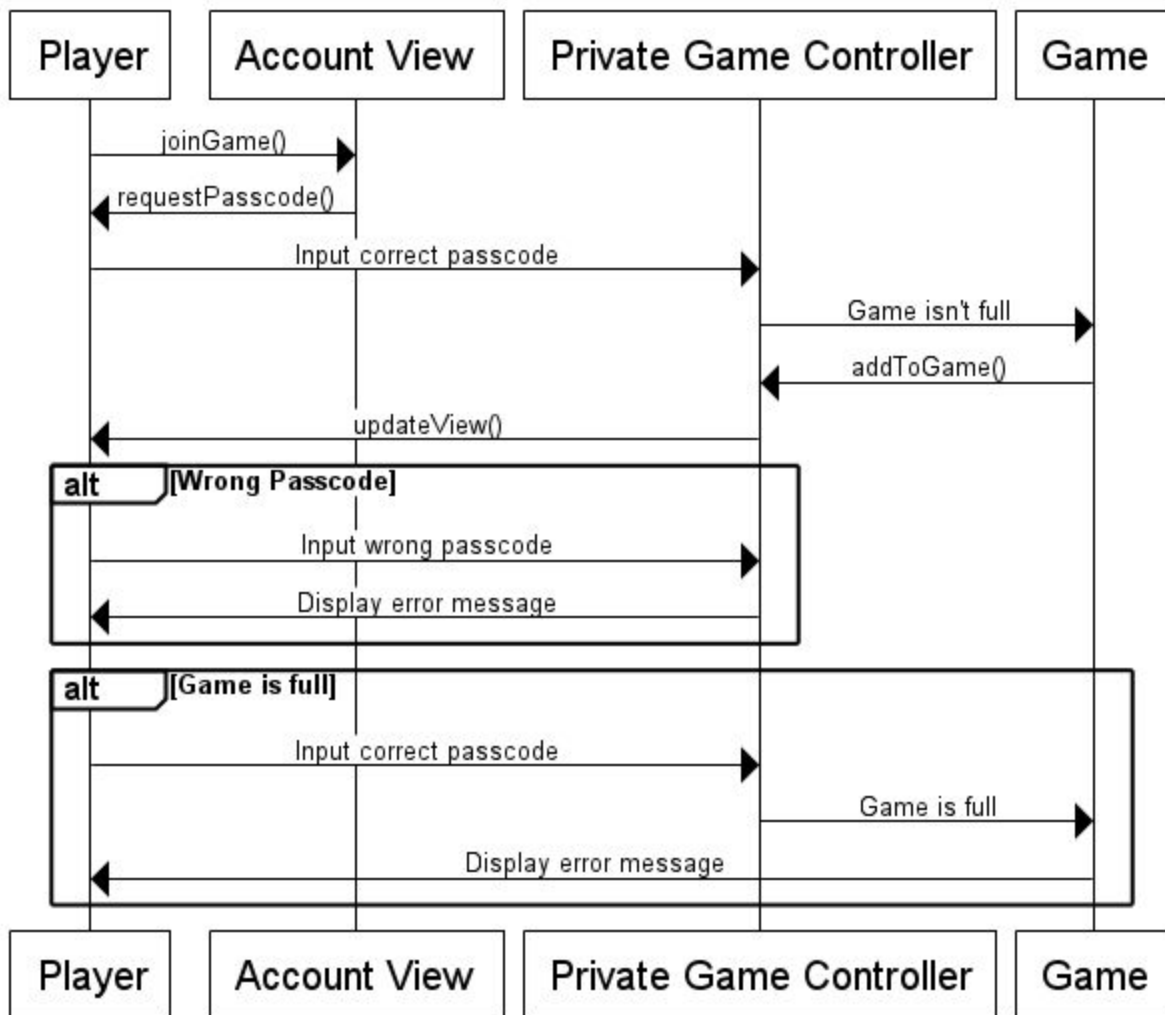


Figure 1.3: UC-7 JoinPrivateGame

The private game controller here is very loosely coupled. Its separate from the public game controller and so we can manipulate the special properties of a private game very clearly and easily. This would have been more confusing if all the games were combined into a single controller.

## UC-8 PlaceMarketOrder

The PlaceMarketOrder use case begins with the player placing a market order. Once this market order is placed, the Game Controller collects information from the Player's Portfolio and the Company they are placing the order on to determine if this is a valid order. If the order is valid, the Game Controller updates the necessary stock information and updates the player's portfolio and then displays a message to the player indicating that the order was a success. If the order is invalid, the Game Controller displays a message to the player indicating that the order is not valid.

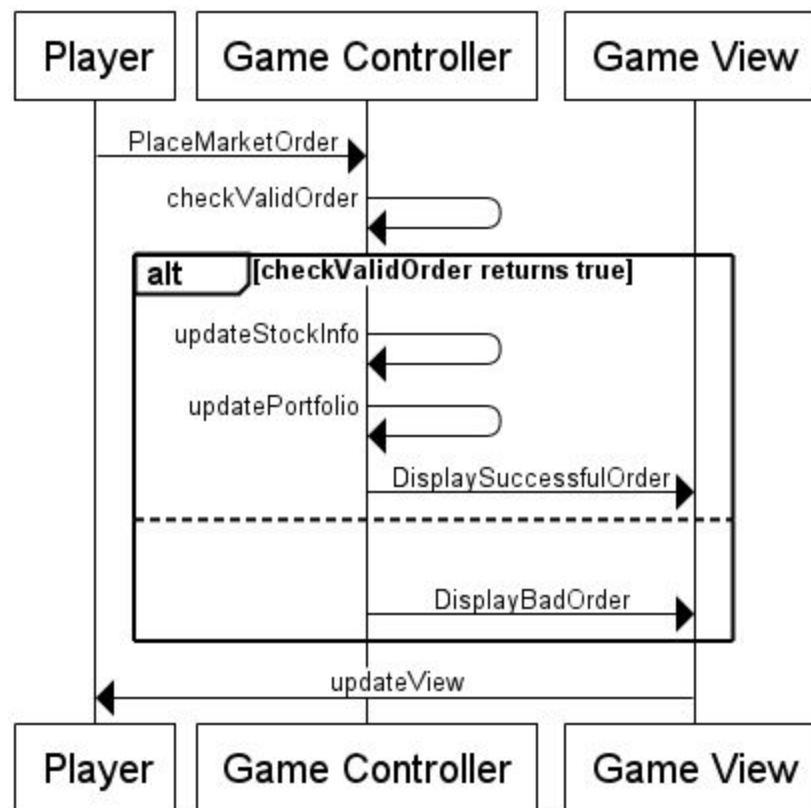


Figure 1.4: UC-8 PlaceMarketOrder

Game Controller is the expert doer since it contains all the information regarding the game. They are also loosely coupled since the classes don't directly affect each other outside of functions.



## UC-9 ViewPortfolio

The ViewPortfolio use case begins with a user selecting to view their portfolio through the button on the Game View. The Game Controller holds all users' portfolio data for a particular session and is able to display it to the user's Game View on command without any external steps.

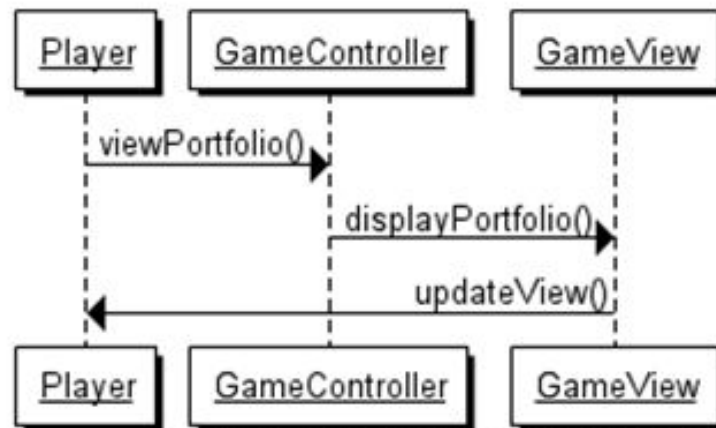
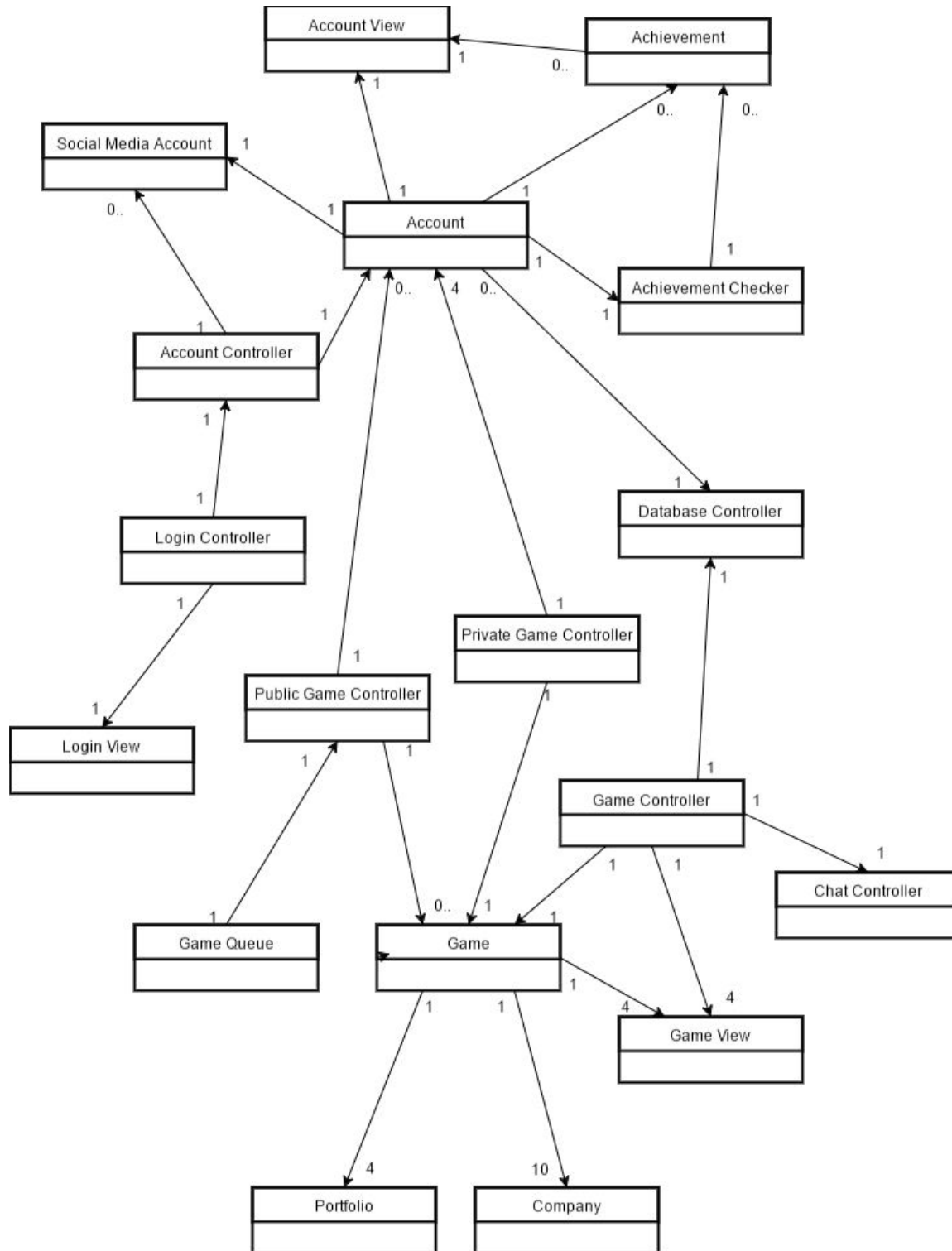


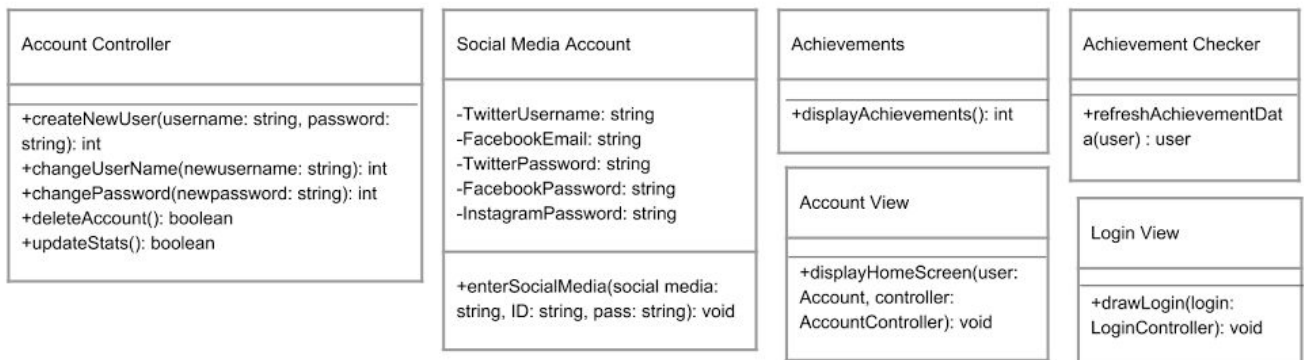
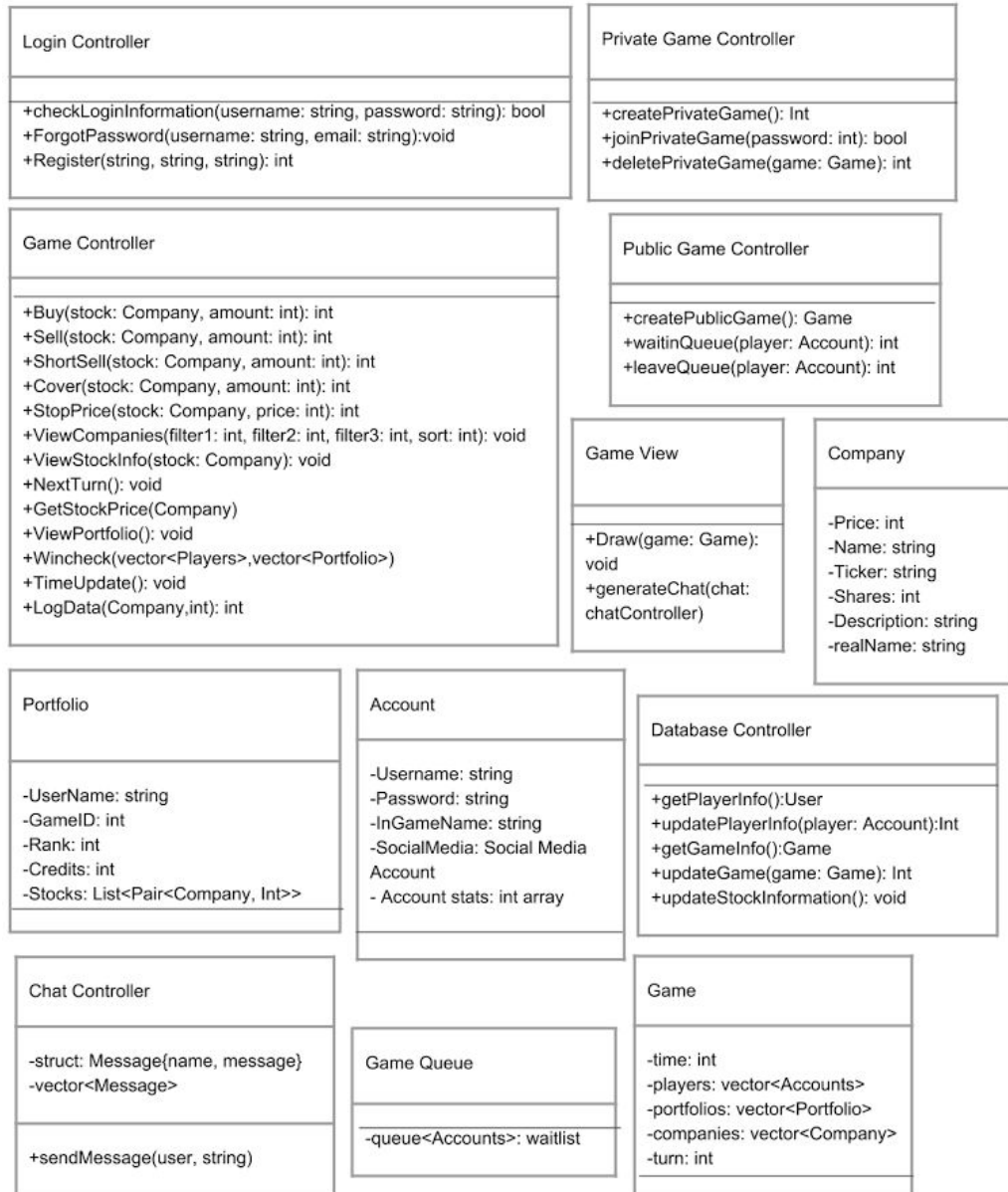
Figure 1.5: UC-9 ViewPortfolio

Game Controller is the expert doer since it contains all information regarding the game and we only use it. They are also loosely coupled.

## 2 CLASS DIAGRAMS AND INTERFACE SPECIFICATION

### 2.1 Class Diagram





## 2.2 Data types and Operational Signatures

### Login View

Used to draw the login view.

#### Methods

- **+drawLogin(login: LoginController): void:** Draws the login screen. Draws the user ID field, the password field, the forget password button and the register field.

### Login Controller

The login controller possesses the responsibility of managing the login of the users in the game. This entails checking if the login information that is inputted by the user is correct. Also, there is an option to forget password and create an account.

#### Methods:

- **+checkLoginInformation(username: string, password: string): bool:** This is the authorization of inputting the username and password. The function takes two parameters, the username and password, in order to validate if there is an account associate with the information. This returns a boolean with true indicating successful login or else there is no account associated with the information provided.
- **+forgotPassword(username: string, email: string):void:** This method is called when the user does not know their password associated with their account. The function takes two parameters, the username and email, in order to be sent an email with their password.
- **+register(string, string, string): int:** Calls Account Controller's createUser to make an account with this username and password. Returns 0 upon completing successfully, and other numbers if the username is taken or the password is invalid, or the two passwords do not match.

### Account

This class contains the information regarding the user's account information.

#### Attributes:

- **-Username: string:** The account's username.
- **-Password: string:** The account's password.
- **-InGameName: string:** The player's in game name.
- **-SocialMedia: Social Media Account:** The information regarding the social media accounts this game is linked to.

## Account View

Responsible for displaying the home screen.

### Methods:

- **+displayHomeScreen(user: Account, controller: AccountController): void:**  
Displays the home screen of the user. The create game button, the find game button, check achievements button, and the settings button.

## Account Controller

The account manager possesses the critical responsibility of managing all user accounts.

This entails the addition of new users creating their accounts and subsequently new and old users to change their account credentials, their username and passwords. The account manager is also accountable for the deletion of user accounts, as well as housing the entirety of the hods

achievement information, from the milestones required to unlock achievements in addition to the achievements unlocked by the individual users themselves.

### Methods

- **+createNewUser(username: string, password: string): int :** This method is executed by the account manager when creating a new user. The user enters their own preferred username as well as password. An integer is returned where if the value is 0 then the username was successfully created otherwise the value corresponds with various error codes.
- **+changeUserName(newusername: string): int :** This method is for the purpose to offer users the option of changing their username. The account manager inputs their new username (a string) This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+changePassword(newpassword: string): int:** This method is for the purpose to offer the user the option to change their password. The account manager inputs a specific user and inputs their new password. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+deleteAccount(): boolean:** This method is for the purpose to offer the option for the user to delete their account. The account manager gathers the information of the user and deletes the account from the database. This returns a boolean to indicate if the deletion of the account was successful.

## Game Queue

The class is responsible for adding players into the game queue.

### Attributes

- **-queue<Accounts>: waitlist** : A first in first out queue that holds players waiting for a public game session, transitioning them into valid public sessions.

## Public Game Controller

The game manager is responsible for storing and managing all information regarding both public games. That is, it is accountable for managing all public games, creating public sessions as per current users entering the queuing system. It is responsible for managing the public queue system, transitioning users through the queue and placing them in valid public sessions bases on a first in, first out principle. After a public game is completed, the game manager is responsible for deleting that session and returning users back to the main menu. Particularly for private sessions, the game manager is dependable for storing, updating, and fetching their unique passcodes.

### Methods

- **+createPublicGame(): Game** : This method exists to allow the game manager to create a public game. It does not require an input, however it performs its operations by creating a valid game session that can be joined by public online players that transition through the queueing system.
- **+waitinQueue(player: Account): int** : This method primarily serves to operate the queueing system that is implemented to allow public online users to proceed to enter valid public sessions. It takes an individual user and places them through the queue, transitioning players into public games on first in, first on basis. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+leaveQueue(player: Account): int**: This method is executed when a user is in queue and wants to leave. The user is basically removed from the queue that would have entered them into a game. This returns an integer with 0 indicating successful completion and other numbers for different error codes.

## Private Game Controller

Deals with the creation of private games, and is responsible for deleting a private game. Also a passcode is returned when a public game is created. This passcode allows for other players to enter a private game.

### Methods

- **+createPrivateGame(): int** : This method allows the game manager to create a private game. Private games are similar to public games; however, users can only enter ongoing valid private game sessions by entering a valid passcode. Once the game is created, a 4 digit passcode is returned to the user which needs to be shared with others who want to join that game.
- **+joinPrivateGame(password: int): bool** : This method allows other users to use a 4 digit passcode to join the private game. If the password is correct, it returns true and adds user to the game, otherwise it returns false and tells the user to try again.
- **+deletePrivateGame(game: Game): int** : This method allows the game manager to delete private games. After a private game has been completed, this method inputs that game and deletes it, thus freeing up system memory. This returns an integer with 0 indicating successful completion and other numbers for different error codes.

## Game

This class contains the data members that compose a game and relevant information.

### Attributes:

- **-time: int**: Goes from 0-60, representing how much time is left in the turn.
- **-players: vector<Accounts>**: The list of players in game.
- **-portfolios: vector<Portfolio>**: The players portfolios, in the same order.
- **-companies: vector<Company>**: The list of companies in game.
- **-turn: int** : An integer describing what turn it is.
- **-year: int**: A random int created upon creating the game that corresponds to a time from the historical stock database.

## Game View

This displays the app during an ongoing game.

### Methods

- **+draw(game: Game): void** : Draws the current state of the game.
- **+generateChat(chat: chatController)** : This method reads the messages stored in a vector in the chatController class and displays them in the game.

## Game Controller

The game controller manages everything encompassing the actual gameplay.

### Methods

- **+buy(stock: Company, amount: int): int** : This method allows for users to purchase stock in a particular company. The user chooses what company they are interested in and the quantity of stock they wish to purchase of that company. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+sell(stock: Company, amount: int): int** : This method is the opposite of buying, allowing users to sell a particular amount of stock in a particular company.. The user selects what company they wish to sell the stock of as well as the desired quantity. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+shortSell(stock: Company, amount: int): int**: This method is not the same as selling a stock, but rather a short selling involves a person trying to sell a stock that they do not own given by the company. The stock will come from the company or from another player that bought the stock. The user selects what company or player in order to place short selling as well as the desired quantity. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+cover(stock: Company, amount: int): int**: This method is only for a user that has placed a short selling on a stock. The user must close the short selling by buying the same shares and returning them to the company or other player. The user selects what company or player in order to place the covering from the short selling as well as the exact quantity that was sold. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+stopPrice(stock: Company, price: int): int**: This method is being able to set a stop price in selling a stock from a particular company. The user selects what company's stock to set a stop price and also the specific price quantity to have set per user. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **--viewCompanies(filter1: int, filter2: int, filter3: int, sort: int): void**: This method lists all the companies and their basic information including price, amount, change, Name, and Ticker with filters and different things to sort by.



- **+viewStockInfo(stock: Company): void:** This method causes more information about a company to show up.
- **+nextTurn(): void:** This method increments the turn, the year, updates all company prices and information, updates user portfolios, and updates ranking.
- **+getStockPrice(Company) :** Asks the database for the next value of the company's price using its real name.
- **+viewPortfolio(): void:** Calls game view to draw the current portfolios and standings.
- **+wincheck(vector<Players>,vector<Portfolio>):** Checks who won on turn 25.
- **+timeUpdate(): void:** Checks time. Upon reaching 0, calls NextTurn and sets time to 0.
- **+logData(Company,int): int :** This method is allocated for logging the data within the stock manager, including the transactions, the remaining stocks, as well as the simulation within the database. This returns an integer with 0 indicating successful completion and other numbers for different error codes.

## Database Controller

The database manager is responsible for the main function of managing the entire system database. This crucial accountability ranges all the data that encompasses the account information, the status regarding the ongoing public as well as private games, updating and fetching information to and from the database, and ensuring that the application is functioning as expected. The database manager grandfathers all information and databases.

### Methods

- **+getPlayerInfo():User :** This method allows the database manager to access an individual user's credentials. By using an instance of the User/Player class in this method, it is able to retrieve that information and transfer it to the subsequent manager that requires that information.
- **+updatePlayerInfo(player: Account):int :** This method exists for the purpose of the database manager to update an individual's player information. The database manager can then delegate the information that requires update to one of the other managers, signaling a successful delegation with a boolean. This returns an integer with 0 indicating successful completion and other numbers for different error codes.

- **+getGameInfo():Game** : The method of getting an ongoing game's information is important to the database manager. By using this method on an instance of the Game class, it can receive and transmit this information to pertinent managers, particularly the game manager, and also perform routine analysis ensuring that ongoing games are performing as expected and the application is functioning properly.
- **+updateGame(game: Game): int** : This method allows the database manager to input a valid ongoing game session and perform routine as well as required updates. Both as a precaution and a remedy, this method also ensures that ongoing games are running smoothly and verifies with a boolean to signal successful update completion. This returns an integer with 0 indicating successful completion and other numbers for different error codes.
- **+updateStockInformation(): void**: This method is called by the game controller to update the stock information of each company given in the game based on the market orders that are created by each player.

## Chat Controller

Deals with the ingame chat system.

### Attributes:

- **-struct: Message{name, message}**
- **-vector<Message>**: Holds all the messages sent in the game. Each message is stored under a struct, in the form of who wrote the message and the message entered as string.

### Methods:

- **+sendMessage(user, string)**: Takes in the message entered by the user as a string, as well as the user who entered the message and saves the message in a vector

## Company

The class representing a company and holds data relevant to the said company.

### Attributes

- **Price: int** : A number corresponding to the current price of a particular stock.
- **-Name: string** : The name of the Company.
- **-Ticker: string** : The Company's ticker.
- **-Shares: int** : The total number of shares in this company.
- **-Description: string** : Description of the company.
- **-realName: string** : The real name of the company this represents (hidden).

## Portfolio

The class representing a user's portfolio, which holds the stocks a user owns in that current game session. It also holds the user's current game statistics such as their profits, rank, as well their username and the data that corresponds to the game session, defined as gameId.

### Attributes

- **-Username: string** : A self selected string representing a user's name in the game.
- **-GameID: int** : A unique integer that that corresponds to and identifies to an ongoing valid game session. Both public and private games have a unique GameID.
- **-Rank: int** : An integer representing the current place a user is in that ongoing session. Being in first place means the user has the highest net worth in comparison to all other users in that session, and is poised to win the game.
- **-Credits: int** : An integer representing the amount of currency a player has in liquid form capable of buying stocks and gained from selling stocks.
- **-Stocks: List<Pair<Company, Int>>** : A list of all Stocks a player owns along with how much of each one they have.
- **-Net Worth : int**: An integer representing the total value of a player's assets, equal to credits +  $\sum \text{Stocks.Company.Price} * \text{Stocks.Int}$ .

## Social Media Account

The class represents the social media accounts linked to existing player accounts.

### Attributes

- **-TwitterUsername: string** : A string that consists of the Twitter account username of the associated player
- **-FacebookEmail: string**: A string that consists of the Facebook account email of the associated player.
- **-InstagramUsername: string**: A string that consists of Instagram account username of the associated player
- **-TwitterPassword: string**: A string that consists of the Twitter account password of the associated player
- **-FacebookPassword: string**: A string that consists of the Facebook account password of the associated player
- **-InstagramPassword: string**: A string that consists of the Instagram account password of the associated player

### Methods

- **+enterSocialMedia(social media: string, ID: string, pass: string): void** : Once a user chooses to implement their twitter, facebook, or instagram into their account, this method will check the login information in that social media account.

## Achievements

Holds information about which achievements a user has completed and is responsible for displaying achievements on a user's screen.

### Methods

- **+displayAchievements(): int:** This method runs when the user wants to view their achievements. Data that tells this function whether or not an achievements requirements have been fulfilled is stored under a variable. This variable is updated when the user is passed onto the achievements manager class.

## Achievement Checker

The class responsible for the management of all information and milestones regarding achievements. That is, it houses the data and descriptions of all in game achievements and the requirements that need to be fulfilled in order for all individual players to unlock them. The achievement system is uniform throughout, with all users being able to unlock the same achievements. This is to ensure complete fairness and cohesion. The achievement manager is accountable for the access of user portfolio data that is utilized in the determination if the particular user has sufficiently completed the requirements to be awarded an achievement.

### Methods

- **+refreshAchievementData(player: Account) : Account:** Every time a user refreshes their achievements pages or completes a game, this method runs and information about a user is passed on to the server. The information (such as the number of games they have won) is processed by this method and another field in the user class is updated which lets the front end system know which achievements the user has completed.

## 2.3 Traceability Matrix

Class\Domain	Login View	Login Controller	Account View	Account Controller	Game Queue	Public Game Controller	Private Game Controller	Game	Game View	Game Controller	Database Controller	Chat Controller	Company	Portfolio	Social Media Account	Achievement	Achievement Checker
Login View	√	√															
Login Controller	√	√		√													
Account			√	√	√	√	√				√				√	√	√
Account View			√	√												√	
Account Controller		√	√	√											√		
Game Queue						√	√										
Public Game Controller			√			√	√	√									
Private Game Controller			√				√	√									
Game						√	√	√	√	√			√	√			
Game View								√	√	√							
Game Controller								√	√	√	√	√					
Database Controller			√							√	√						
Chat Controller										√		√					
Company								√					√				
Portfolio								√						√			
Social Media Account			√		√										√		
Achievement			√	√												√	√
Achievement Checker			√													√	√

The table above illustrates the evolution of our domain concepts to the conception of our classes. Previously, the design of our domain concepts was specific and essentially written as clear descriptions of classes. As a result, the domain concepts seamlessly transitioned into this phase and become our classes, which are refined and functional.

## 3 SYSTEM ARCHITECTURE AND SYSTEM DESIGN

---

### 3.1 Architectural Styles

Rags to Riches utilizes several known software tools and principles into it's design. Using these well established design principles will allow us to develop the application in an efficient, proven way. Some of the architectural systems we'll be including are: model view controller, event driven architecture, and object oriented design.

#### Model-View-Controller

The Model-View-Controller (MVC) pattern separates the software into three subsections: the model, the view, and the controller. Through the separation of the software into these subsections, the software proves to be easier to maintain and can be modularized to readily update individual parts without affecting others. [2]

With development frequently being focused on a particular area, whether it be the front-end or back-end, this allows a developer to easily access the parts they need to. It also makes it much easier for multiple developers to work on the software at once. The MVC pattern is additionally useful in reducing any unnecessary overhead at runtime as resources will only be called upon when they are actually needed.[3]

- The **model** encapsulates the data and any logic or computations needed to process the data.
- The **view** displays the data contained in the model to the user.
- The **controller** facilitates changes to the model as a result of interaction with the view and changes to the view as a result of changes in the model data.

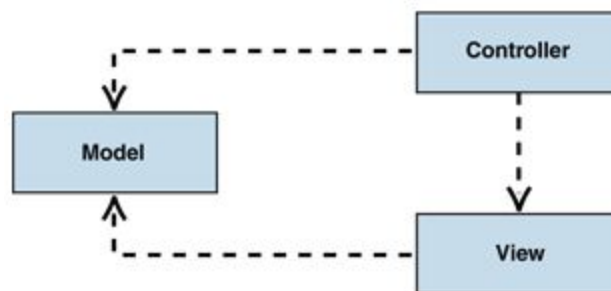


Figure 3.1: MVC Diagram

## Event Driven Architecture

Events are defined as any meaningful change in state. New information or a change in information can change the state of the software. The event driven architecture allows for the propagation of information in a near real time case. There is no official method to implement an event driven architecture, as it can vary depending on the project, but we'll be scattering single purpose event processing components throughout our project that receive and process events. [4]

## Object-Oriented Design

The responsibilities of the software are divided between different objects, containing the relevant information and behavior related to a specific portion of the software. By utilizing the object oriented approach, it makes software easier to understand as well as makes development more efficient.

## Front-End and Back-End

The front-end and back-end are colloquial terms referring to a certain aspect of the software. Both will be developed using their own set of technologies. The front-end component refers to the view of the application on the device that our users will see. The back-end component refers to all the logic operations that will be performed for the application that will be running on an external server.[5]

### 3.2 Identifying Subsystems

Our software will be divided into three main subsystems: the front-end, the back-end, and the external data.

#### Front-End

The front-end system will consist of the user interface which displays the views of our application. The system will be developed and customized specifically for Android phones. The front-end will maintain communication with the back-end by sending the relevant data from a user's interaction to the back-end system. The front-end system will run entirely on the user's device.

## Back-End

The back-end system will be responsible for processing the data received from the front-end as well as returning the relevant data back to the front-end system. The back-end will handle all logic functions and calculations needed for the application and will interact with the external subsystem for certain resources to be used in the application. It will handle tasks such as account registration and account login, public and private games, and in game data such as buying and selling stock.

## External

The external system consists of the outside resources we will be integrating within our application. The stock API is a necessity that will be used to get the data to allow our game to function. As an additional feature, social media accounts will be binded to user's accounts to allow them to share progress to their friends.

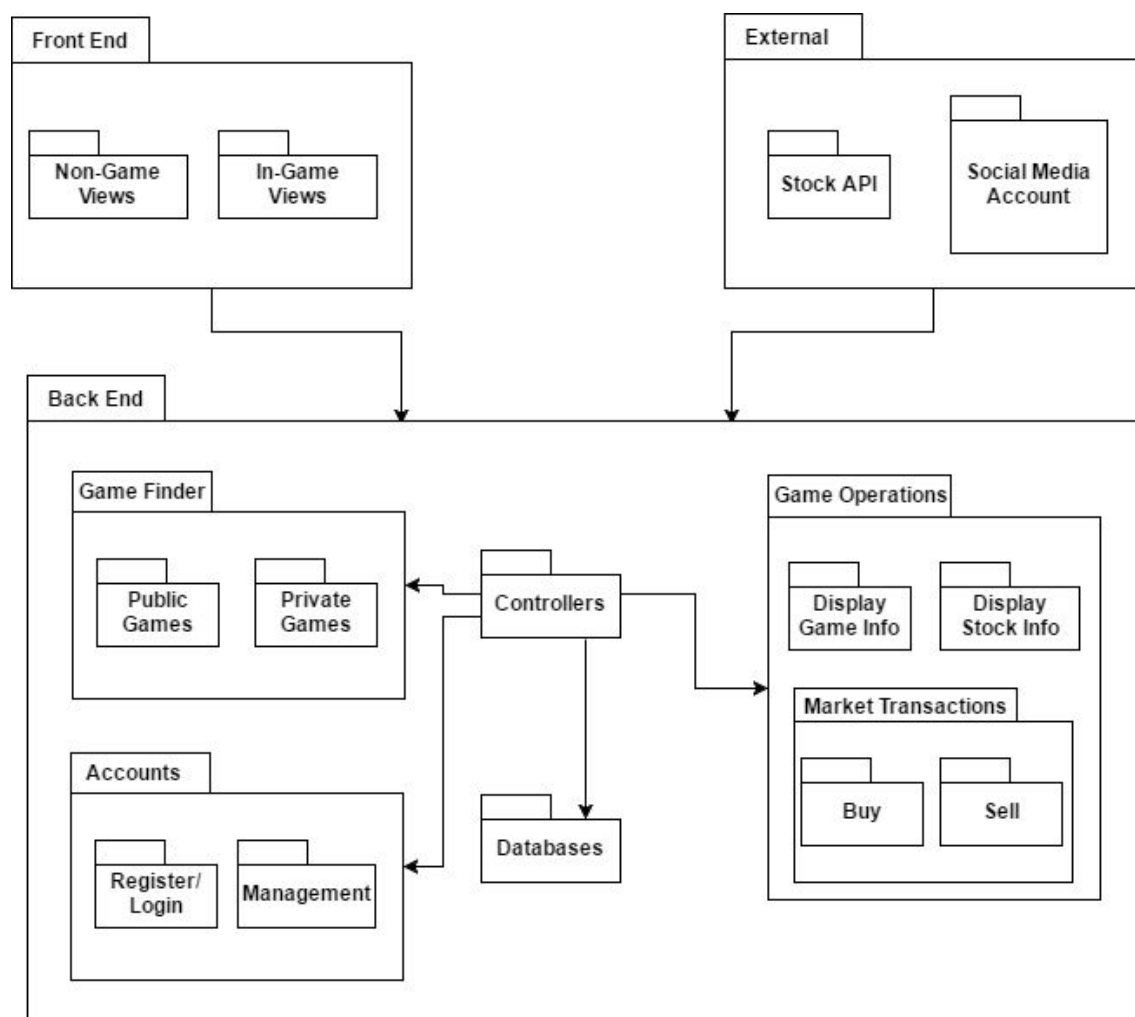


Figure 3.2: UML Package Diagram



### 3.3 Mapping Subsystems To Hardware

When designing an application, it is important to consider the processing power of the device running the application. In order to provide security and efficiency, the back-end will run on an external server separate from the user's device. The user's device will handle the front-end operations, specifically communicating user events on the graphical user interface to correct back-end operations.

### 3.4 Persistent Data Storage

Our application requires a decent amount of persistent data storage. The application will need to store records of user accounts and the data to be used in simulating stocks. We will be using relational databases to hold our data implemented through the use of MySQL. Primarily, there will be two main databases: one for user accounts and one for stock information. These databases won't interact with one another and will be used in separate aspects of the application.

### 3.5 Network Protocol

In order for our application to access webpages such as Facebook, knowledge of the Hypertext Transfer Protocol (HTTP) is needed. HTTP allows for clients (say a browser) and servers (say a device or computer machine) to communicate with one another. In this case the client will be an Android device with a web browser and the server will be the host of a website. Java sockets will be needed to implement this type of communication, as sockets serve like communication tunnels.[6]

A standard communication message to view a page on Facebook would be as follows:

**Desired page:** <http://www.facebook.com/userpage332.html>

*Client sends*

**Client:** GET <http://www.facebook.com/userpage332.html> HTTP/1.0

*Server reads*

**Server:** GET </userpage332.html> HTTP/1.0

The server will then send the information on that page to back to the client and the information will be displayed as a regular webpage.

## 3.6 Global Control Flow

### Execution Order

Some functions will need to occur in a linear fashion, but in general our app will be event driven. All the features of the app will have to be triggered by a certain entity, whether it be the user or the system itself. Most of the event-driven functionalities of the software will arise as a result of user interaction. For example:

- **Logging In:** any user must enter their information before being logged in.
- **Create a Game:** any user must choose their custom settings before creating a game.
- **Achievements:** required criteria must be completed before they can be awarded

There are still many other functionalities (stock trading, portfolio viewing, joining game, etc.) that can only be triggered by the user's interaction. However, some of the event-driven functionalities of the software will arise as a result of the system, such as data retrieval, error checking, and market transactions based on stored data.

### Time Dependency

Our app is not tied by real time stock markets or current stock markets and therefore are not time dependent like previous projects. The app will have simulated stock prices based on data from a subset of pre-selected and hyper-analyzed corporations of a certain time period. This allows for a more robust and educational environment in which the game can be experienced within a smaller timeframe in hopes of retaining interest. All events will be depended on user input rather than time.

### Concurrency

Threads will be needed in order to allow multiple processes to occur. For example, if there are many players online, then there should be more than one queue to contain players. Threads will also play a major role in getting the relevant information from social media so players can share their progress. If the ambitious goal of implementing group messaging is executed, then threads will allow for multiple device communication.

## 3.7 Hardware Requirements

The hardware requirements for this project is an Android smartphone. The device must have a color display with a minimum of resolution of 640 x 480 pixels. In order to play online with other players the device must have reliable internet access and thus a minimum network bandwidth of 56kps is needed. Storage will be required both internally on the device and externally on servers. The device will need to be able to store the application's packages and required data repositories. The server will need to be able to store user accounts, profiles, game data, and stock data.

## 4 ALGORITHMS AND DATA STRUCTURES

---

### 4.1 Algorithms

Our application utilizes weekly stock data from various years rather than implementing a mathematical model for stock data calculation. Therefore, the algorithm that will be designed and implemented will store all of this stock data information within a hashmap. As the project development progresses, additional major working algorithms may require implementation and will be detailed here. While we will not be using algorithms for stock simulation, we will use insertion sort in order to apply the sort function to the stocks. Although insertion sort has a Big O time complexity of  $O(n^2)$  time, because there will only be 10 stocks per game, it will work fairly quickly and sufficiently.

#### Insertion Sort

Insertion sort is where you create a new array by taking the smallest or biggest number in the remaining array. Although it has Big O of  $(n^2)$ , due to the low number of stocks, it is as effective as the other algorithms and less complex to implement.

### 4.2 Data Structures

#### ArrayList

An arraylist will be one of the major data structures that is used in *Rags to Riches*. An arraylist will be used when players create a private game, which it will act as the private game waiting room. The users will enter the correct passcode in order to join the private game. Once the arraylist is filled with 4 players then the private game will commence. Each game will use an arraylist that will store each player's portfolios, which includes stocks and players' assets. Also, the game will use the arraylist to store each Company's stock price, achievements, and messages in the Chat Controller. The arraylist is a default data structure in Java.

#### Queue

When players wish to enter a public game, they'll be inserted into and processed through a public game queue. Because a Queue follows the first in first out (FIFO) principle, players who began searching for the game first will be given priority in finding a game. The Queue data structure will be implemented using the Java Queue util. Once at least four members have been enqueued, the four players will be "dequeued" (popped off) from the queue and be entered into a public game session.

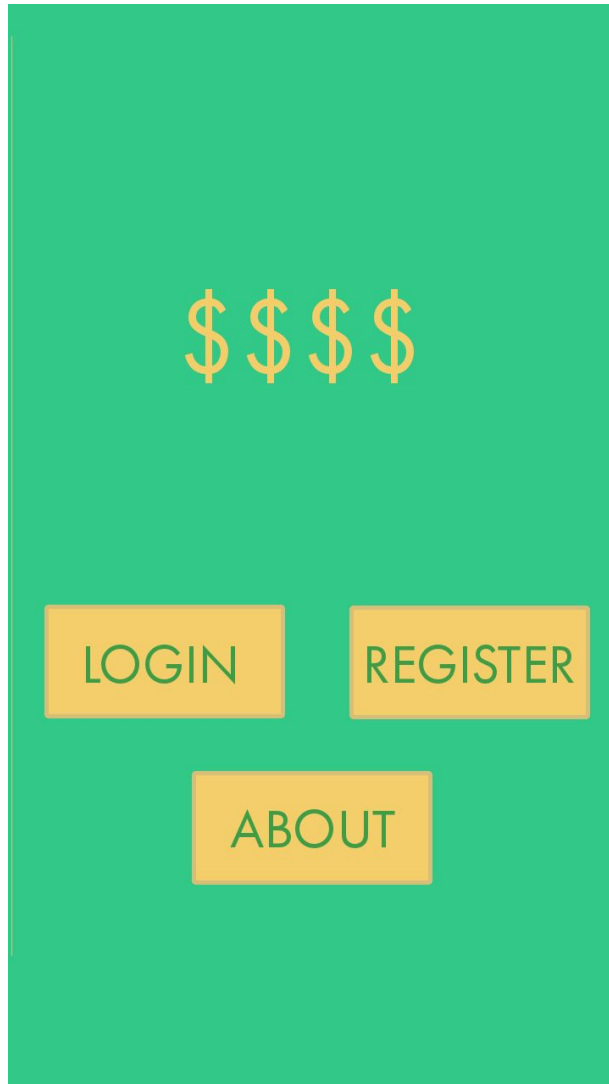
## 5 USER INTERFACE DESIGN AND IMPLEMENTATION

---

Currently, at the time of this partial report submission, our team has not yet implemented the initial screen mock-ups that were developed for Report 1. Nevertheless, our team has organized to meet during the spring recess in order to accomplish the bulk of the development for the application and prepare for the demonstrations. Our plan from report 1 has remained intact, with the same level of user effort required as detailed in section 4.2.

One of the core principles in our design procedure is to make the application extremely user friendly. The aim of our user interface is to be intuitive, simple, and concise, thus allowing for people regardless of age to access and enjoy the application. As follows, the user effort will be consistent with the previous descriptions, aiming to be as intrinsic and minimalistic as possible. Cosmetic attributes, such as changes in colors and styles, are negligible in terms of functionality, only significant in attractiveness and other superficial characteristics.

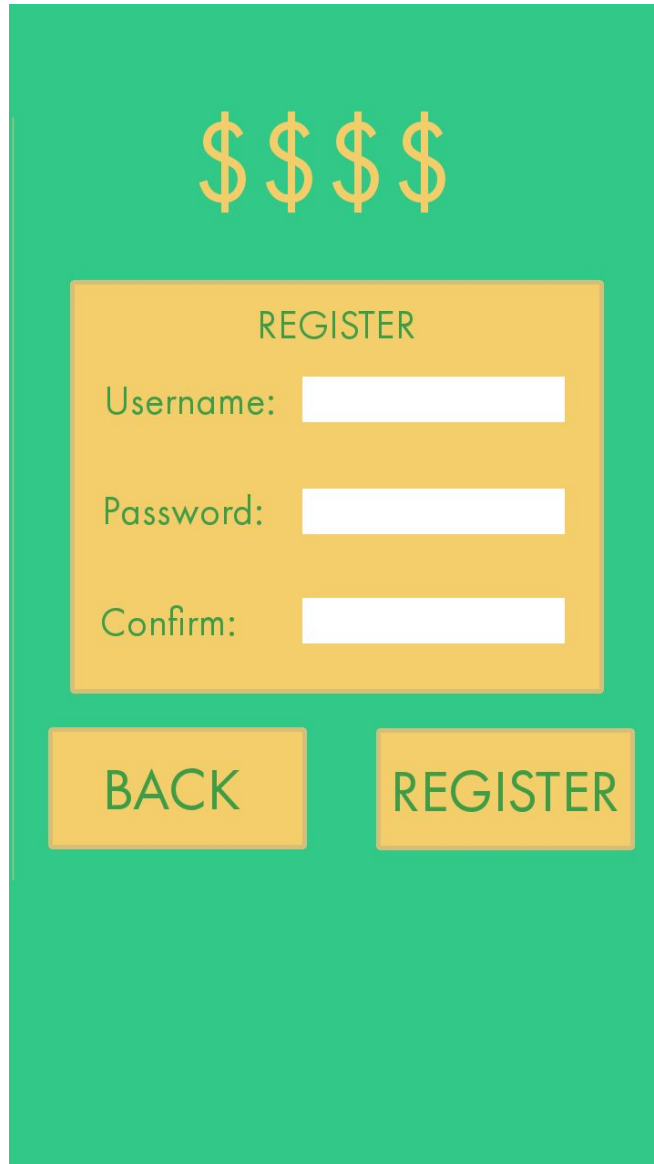
The main focus of our user interface is the principle of ease-of-use. In our application, ease-of-use relates to our design decision on the user interface to be easily understandable, operable, and intuitive. In essence, it should be operable without any question or clarification, or without the requirement of reading excessive instruction or documentation. The user interface is to be clean, simple, and well organized, without any flashiness or excessive colors, pictures, or graphics, which could contribute to greater user effort. Minimizing user effort will allow the maximization of ease of use, which will allow the application to provide the user with an overall positive, educational, and enjoyable experience. Follows are initial screen mock-ups from report 1, which will commence implementation soon.



**Figure 5.1: Initial Landing Page**

The initial landing page, which is the first page the user sees when they enter the app.

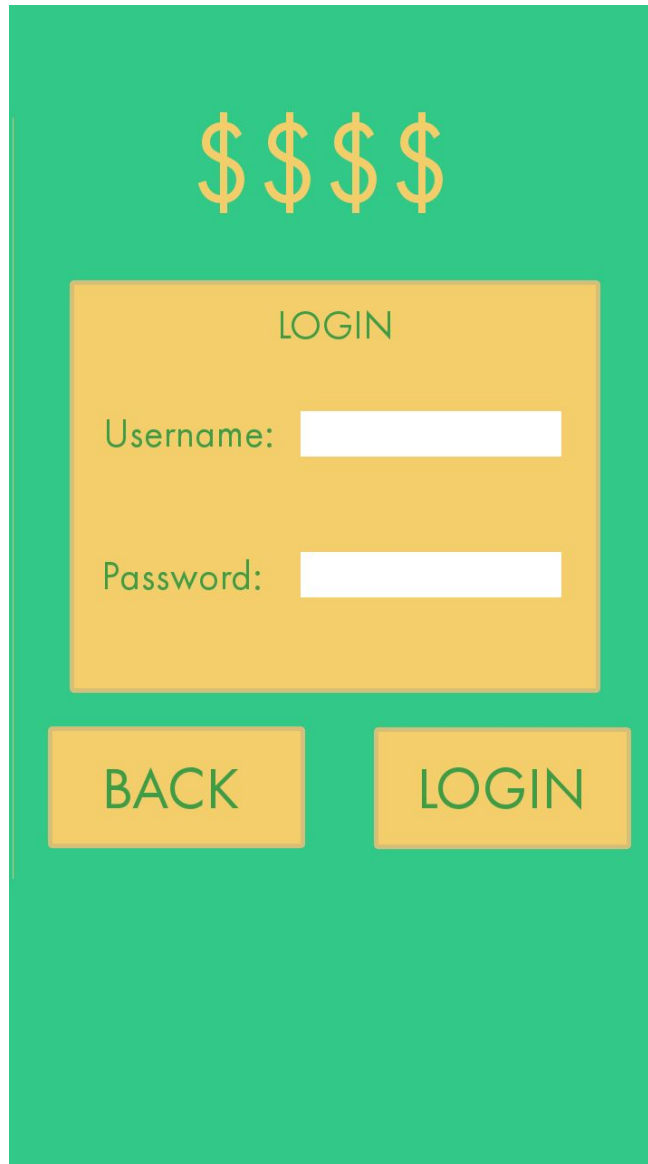
Selecting login brings the user to the login page, while register brings the user to the register page. Selecting about displays the privacy policy as well as the application terms and conditions.



The image shows a mobile application interface for a registration page. At the top, there are four yellow dollar signs (\$\$\$\$) on a green background. Below this, there is a yellow rectangular box containing the word "REGISTER" in green capital letters. Underneath the title, there are three input fields: "Username:", "Password:", and "Confirm:", each followed by a white rectangular text box. At the bottom of the green background, there are two yellow buttons: "BACK" on the left and "REGISTER" on the right, both in green capital letters.

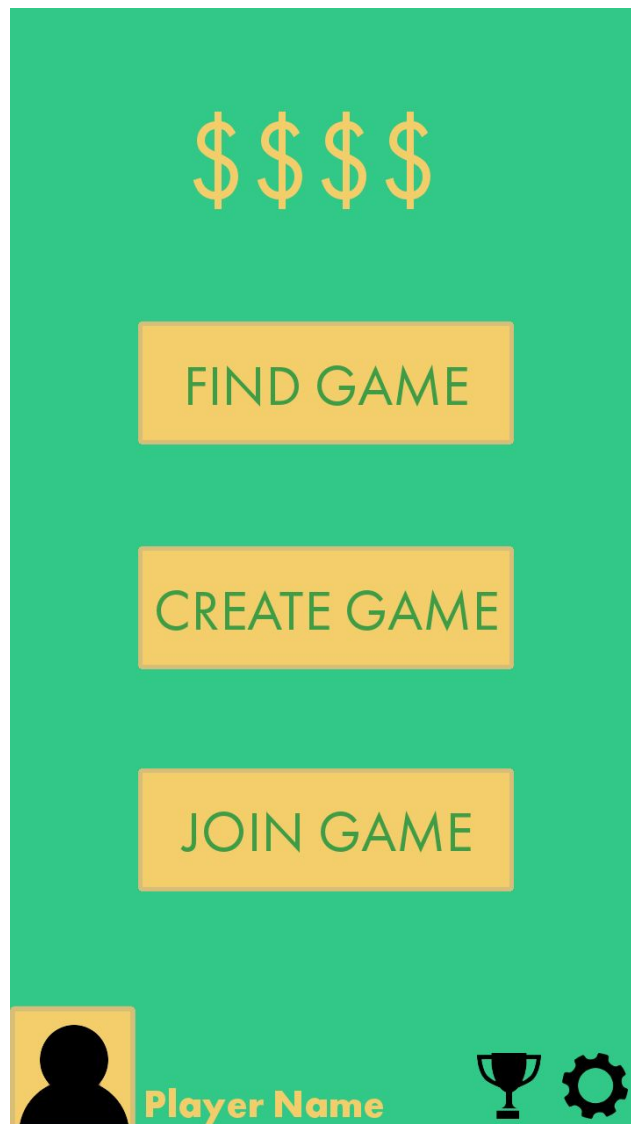
**Figure 5.2: Register Page**

The register page, where users can enter a username, email address, and password in the fields. If password and confirm (password) match, when the user presses register, the user is registered and brought to the home screen. If the passwords are not the same, an error message is displayed. Selecting back brings the user back to the initial landing screen.



**Figure 5.3: Login Page**

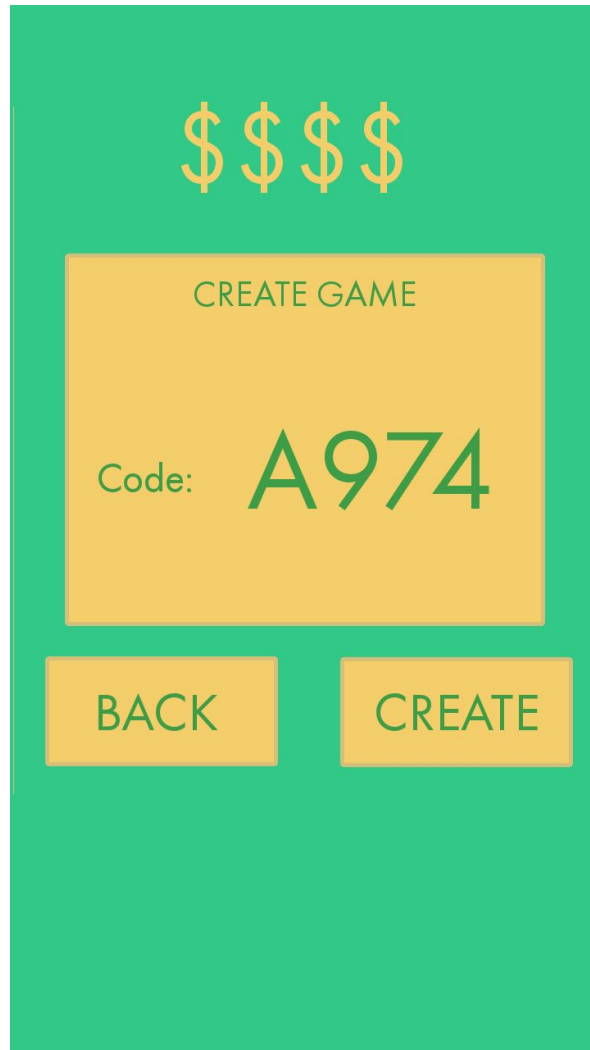
The login screen, where users can enter a username and password in the fields. Pressing login with correct credentials brings user to home screen. Incorrect credentials displays error message.



**Figure 5.4: Application Home Page**

The home screen. Gives users the option to find a public game, create a private game, or join a private game. Selecting the icons in the bottom right allows the user to go proceed to the settings or achievements pages.

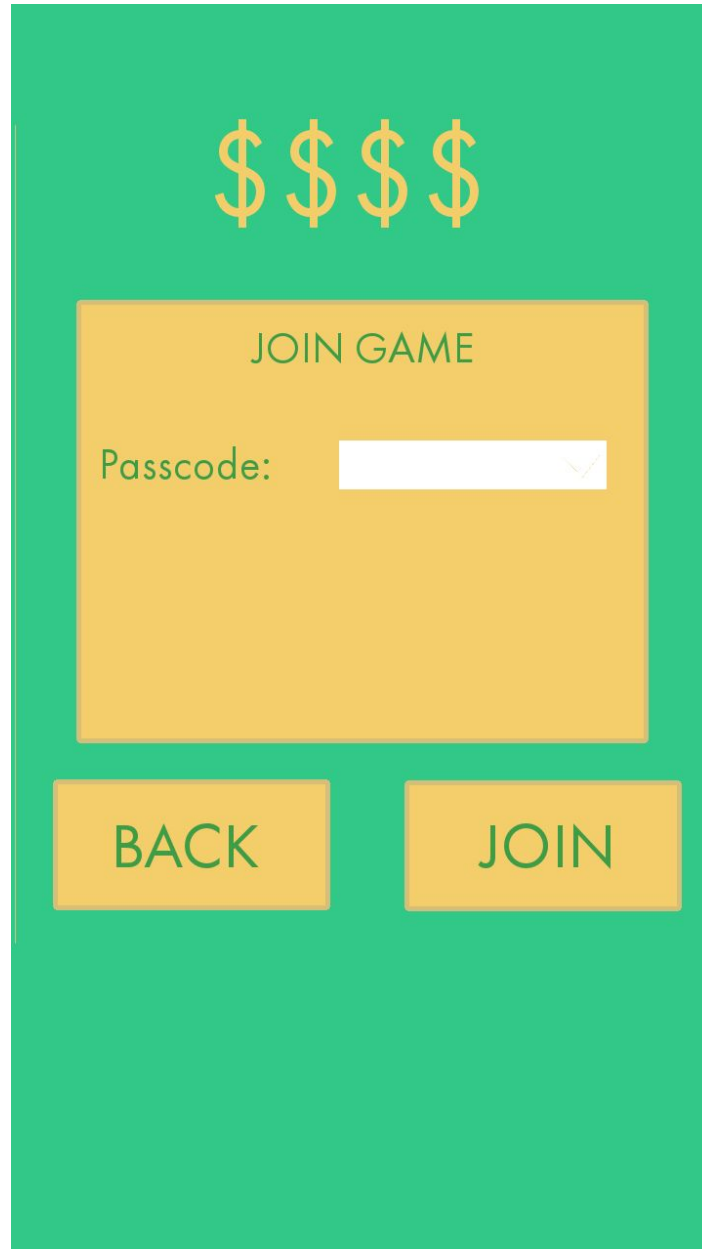




**Figure 5.5: Create Private Game Page**

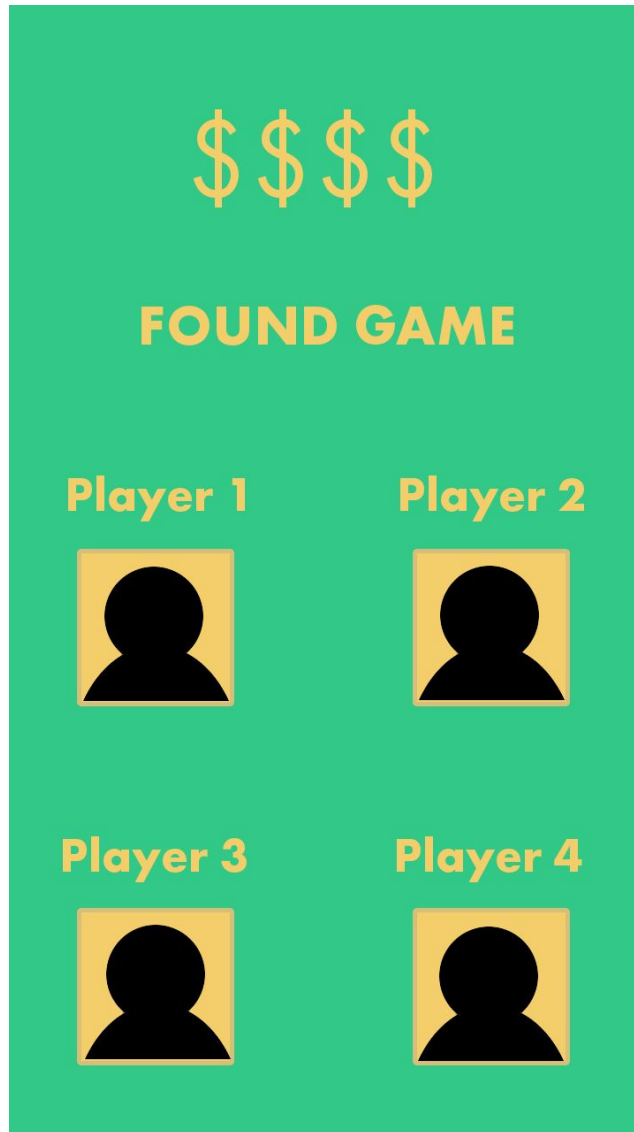
This page is shown to the user when they create a private game. A random 4 digit passcode is shown which the user can share to other players who want to play in the private game.

In the original mockup, this create game page required the user to enter the number of players in the game, initial capital, and the turn duration. We've simplified the game by having these settings be a value the user cannot change. When the user creates a private game, a 4 digit passcode to enter the private game will instantly be generated. This simplification will allow players to start the private game a lot quicker.



**Figure 5.6: Initial Landing Page**

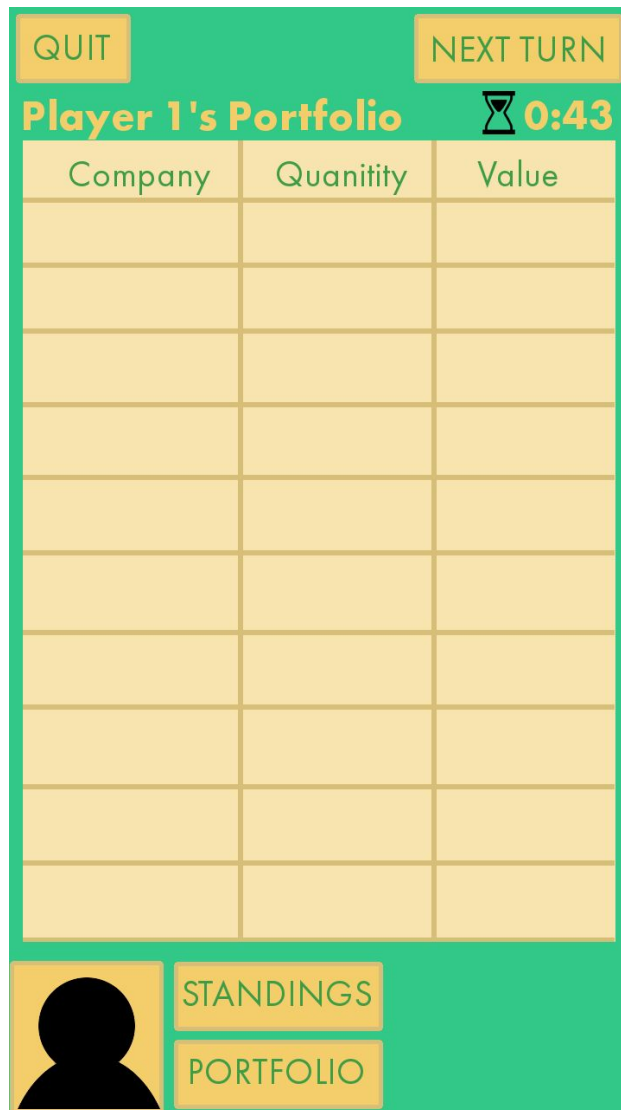
The user is asked to enter a passcode in order to join a private game session that was created by another user. Once passcode is entered the user is processed to the specific private game given the corresponding passcode.



**Figure 5.7: Found Game Page**

After selecting the option, to join a public game, the user will be processed through a queue. After a short time, this screen will appear, indicating that the player has successfully been processed, other online players have been found, and the public will commence shortly.

The player's names and pictures are displayed here as well.



**Figure 5.8: View Portfolio Page**

While in a public or private game, the player can view their own portfolio or an opponent's portfolio. The above screen is shown, displaying all of the player's stocks, quantities, and values that they own. They can also view the player standings. They can return at any point by clicking the top left, a timer is present allowing them to know when the turn is finished

## 6 DESIGN OF TEST

### 6.1 Use Cases and Unit Testing

#### Login and Registration

Test-case Identifier: TC-1 (Login) Function Tested: <b>checkLoginInformation(username: string, password: string): bool</b> Pass/Fail Criteria: A user's login credentials will be checked. The user must be registered with the system.	
Test Procedure: Login with Correct Username and Password. Login 6 times with invalid Username and 6 times with incorrect password.	
Evaluation	Result
Pass	User is able to log into the correct account with the corresponding username and password. If an invalid username is given, outputs Wrong Username. If a wrong password is given, output: Wrong Password: N Attempts Remaining, where N starts at 4 and decrements by 1 with each wrong attempt. If N reaches 0, do not let further attempts.
Fail	Ur Allow login with an invalid password or username. Allows further attempts after the 5th wrong one. Logs in the the wrong account.

Test-case Identifier: TC-2 (Register) Function Tested: <b>register(username: string, password: string): int</b> Pass/Fail Criteria: This test case checks if the user has provided proper field values on the registration page.	
Test Procedure: Input invalid email. Input password of length 7. Input password without special character. Input password of length of 8 and with 1 special character.	
Evaluation	Result
Pass	The ID is at least 9 characters without special characters.
Fail	The ID is not 9 characters or has special characters.

Pass	If the password meets the criteria of at least 8 characters and one special character then the account is created. If the password does not satisfy the criteria then a message outputs: "Password does not match criteria. Enter password with at least 8 characters and 1 special character."
Fail	This does not output an error if the password entered is at least 8 characters and has 1 special character. This does not create the account if the password does not meet the criteria.

Test-case Identifier: TC-3 (Forgot Password) Function Tested: <b>forgotPassword(username: string, email: string):void</b> Pass/Fail Criteria: This test case checks if the user does not remember their password and needs to provide their email address to have their password sent.	
Test Procedure: Input an invalid email. Input a valid email. Check email.	
Evaluation	Result
Pass	If inputted email is valid, (follows format of "string",@,"string",,,"string"), then the password is sent to the email that is inputted. If the user enters an invalid email then it prompts an output of "Invalid E-mail. There is no account associated with this email. Please input correct email."
Fail	Does not output an error if an input is taken without an "string" followed by @ followed by "." followed by "string". Does not allow the user to receive an email with their password.

## Account Controller

Test-case Identifier: TC-4 (Delete Account) Function Tested: <b>deleteAccount(use:user): boolean</b> Pass/Fail Criteria: This test case confirms if all the details of a user account can be changed..	
Test Procedure: Call deleteAccount() function and confirm account is deleted.	
Evaluation	Result
Pass	The account is deleted, it does not exist in the account database.
Fail	The account still exists.

Test-case Identifier: TC-5 (Join Public Game) Functions Tested: <b>changeUserName(newusername: string): int</b> , <b>changePassword(newpassword: string): int</b> Pass/Fail Criteria: This test case confirms if all the details of a user account can be changed..	
Test Procedure: Call each function and confirm that the ID is changed after running the changeUserName() function, the password is changed after running the changePassword() function.	
Evaluation	Result
Pass	A user's ID is changed, and users pass is changed.
Fail	Users ID or pass is not changed. We know which function is incorrect based on if the ID or pass does not change.

## Public Game Controller

Test-case Identifier: TC-6 (Join Public Game) Function Tested: <b>waitinQueue(player: Account): int</b> Pass/Fail Criteria: This test case confirms if players can enter the queue to enter a public game.	
Test Procedure: Calls function.	
Evaluation	Result
Pass	A player enters the public game queue.
Fail	A player does not enter the public game queue. .

Test-case Identifier: TC-7 (Leaving Queue) Function Tested: <b>leaveQueue(player: Account): int</b> Pass/Fail Criteria: This test case confirms if a player no longer wants to be part of the public game, then the player is able to leave the queue.	
Test Procedure: Call function with 3 accounts. Call function with another account. Call function with 5 accounts	
Evaluation:	Result:
Pass	Players are shown a waiting page with 3 accounts. All 4 players enter game view once another account is called. With 5 accounts, the first 4 to call enter a game, and the 5th is shown the waiting page.
Fail	Any other result.



Test-case Identifier: TC-8 (Join Public Game) Function Tested: <b>createPublicGame(): game</b> Pass/Fail Criteria: This test case confirms if players can enter a public game by adding 4 players to the public game queue. The game should automatically begin if the test was successful.	
Test Procedure: Call function with 3 accounts. Call function with another account. Call function with 5 accounts.	
Evaluation	Result
Pass	Players are shown a waiting page with 3 accounts. All 4 players enter game view once another account is called. With 5 accounts, the first 4 to call enter a game, and the 5th is shown the waiting page.
Fail	Any other result.

## Private Game Controller

Test-case Identifier: TC-9 (Create Private Game) Function Tested: <b>createPrivateGame(): int</b> Pass/Fail Criteria: This test case allows a user to create a private game with a passcode of exactly 4 digits. If the passcode is created successfully, a box will display asking to share the game with others via the social media platform. If a passcode is not entered, non-digits are entered or less than 4 digits are entered, an error message will prompt.	
Test Procedure: Input invalid passcode. Input valid passcode.	
Evaluation	Result
Pass	An error message is shown if an invalid password is entered and asks for a new input. When valid input is inputted, offers option to share the game with others via social media and shows an empty game waiting room.

Fail	Any other result.
------	-------------------

<p>Test-case Identifier: TC- 10(Join Private Game)</p> <p>Function Tested: <b>joinPrivateGame(password: int): bool</b></p> <p>Pass/Fail Criteria: This test case confirms if a player enters the correct 4-digit passcode for a specified private game. The game should automatically begin if the test was successful. Once 3 other players join the private game, anyone else who inputs the correct passcode will not be able to join it anymore.</p>	
<p>Test Procedure:</p> <p>Call Function with for room with correct password. Call Function for room with incorrect password. Call function to room with correct password 3 times.</p>	
Evaluation	Result
Pass	The 1st caller enters the game's waiting room. The 2nd caller is outputted Incorrect Password. The 3rd-5th Caller enter the game's waiting room. The 6th caller is outputted: "Room full".
Fail	Any other result.

## Game Controller

<p>Test-case Identifier: TC-11 (Buy Stock)</p> <p>Function Tested: <b>Buy(stock: Company, amount: int): int</b></p> <p>Pass/Fail Criteria: This function should decrease the player's currency by amount multiplied by stock.price, increase the player's portfolio's stock by amount, and decrease the company's available stock by amount.</p>	
<p>Test Procedure:</p> <p>Call Function to buy stock more than the amount of shares left. Call Function to buy stock more than affordable. Call function to buy stock that you can afford and company has shares left.</p>	
Evaluation	Result
Pass	There is enough stock to be bought, there is enough currency, and the player's currency is decreased by the amount multiplied by stock price.

Fail	There is not enough stock to be bought, or not enough currency, or the player's currency is not decreased by amount multiplied by stock price.
------	--

Test-case Identifier: TC-12 (Sell Stock) Function Tested: <b>Sell(stock: Company, amount: int): int</b> Pass/Fail Criteria: The player's currency value should increase once this function is called. The change in currency should be number of stocks sold * price of stocks. The players stock portfolio, and the number of stocks available to be sold by a company should also update.	
Test Procedure: Call function to sell stock, with amount to be sold less than or equal to amount of stocks owned in a particular company. Call function to sell stock that exceeds actual stock owned.	
Evaluation	Result
Pass	There is stock to be sold, and the player's currency is increased by the quantity of stocks sold multiplied by the stock price.
Fail	There is no stock to be sold, or the player's currency is not increased by the amount of stocks sold multiplied by the stock price.

Test-case Identifier: TC-13 (View Portfolio Information) Function Tested: <b>viewPortfolio(): void</b> Pass/Fail Criteria: The user's game view changes to a player's portfolio. The function will identify a fake player and the game should change the view to the to wolike players portfolio.	
Test Procedure: Call the function.	
Evaluation	Result:
Pass	Displays the selected player's portfolio with their game information on the user screen.
Fail	Fails to display the selected player's portfolio.

Test Case: TC-14 (View Companies Information) Function Tested: <b>viewCompanies(filter1: int, filter2: int, filter3: int, sort: int): void</b> Pass/Fail criteria: This tests confirms that the database was successfully accessed the company's information and it is displayed correctly.	
Test Procedure: Call Function.	
Results	Actions
Pass	Accesses the vector in the Game Controller class and the company's information are able to be displayed correctly.
Fail	Displays incorrect company information.

## Database Controller

Test Case: TC- 16 (Get Player Info) Function Tested: <b>getPlayerInfo():User</b> Pass/Fail criteria: This test case confirms that all information of associated player is correct, such as username, email, and social media accounts integrated with account.	
Test Procedure: Call Function	
Evaluation	Result
Pass	Displays player's correct account information.
Fail	Displays incorrect player information.

Test Case: TC-15 (Update player info) Function Tested: <b>updatePlayerInfo(player: Account):int</b> Pass/Fail criteria: This test case confirms that a player's account statistics are updated correctly.	
Test Procedure: Complete a game. Pass in a player's profile into the function. Confirm that the function updates a player's profile by checking the players profile values.	
Results	Actions

Pass	Player information is updated correctly.
Fail	Player information was not updated or updated incorrectly.

Test-case Identifier: TC-17 (Update Stock Information) Function Tested: <b>updateStockInformation(): void</b> Pass/Fail Criteria: This test case confirms if the stocks are updated at the end of every turn in a game.	
Test Procedure: Call Updatestockinformation(). Buy(stock, amount). Then call UpdateStockInformation().	
Evaluation	Result
Pass	The stock's price should be changed to the next week's price from the data. The 2nd call of updateStockInformation should have the amount available decreased by amount and the price changed to the next week's price from the data.
Fail	Either the stock price or amount failed to change properly.

## Chat Controller

Test-case Identifier: Test Case: TC-18 (Generate Chat) Function Tested: <b>sendMessage(chat: chatController): void</b> Pass/Fail Criteria: This test case confirms that messages are displayed and the chat feature works successfully.	
Test Procedure: The function is called in order to generate a chat within a game session, either public or private. To test if the chat was successfully generated, inputting text and viewing if that text appears in the chat box with all users able to view it, the test has passed.	
Evaluation	Result
Pass	Accesses the vector in the Chat Controller class and the messages are able to be displayed correctly, with the chat properly generated, and all players in the game session able to view them.
Fail	There is an error in the generation and messages are not able to be displayed, or all users in the game session are not able to view them.

## Achievement Checker

Test-case Identifier: TC-19 (Updates Achievements) Function Tested: <b>refreshAchievementData(useraccount: user): user</b> Pass/Fail Criteria: This test case checks if a user's achievements are properly updated.	
Test Procedure: Input a fake user with manually changed fields for number of games played: 10 games played, 20 games played, 50 games played. Number of games won: 5 games, 10 games, 20 games. Test this function for every changed element.	
Evaluation	Result
Pass	The output user data structure is properly updated corresponding to the input users changed fields.
Fail	If the output user data structure does not have properly updated achievements or if the function fails to return a user.

### 6.2 Test Coverage

Indeed, the quintessential test coverage would be extremely comprehensive, covering and testing every single edge case imaginable for every single method. While the idea of this pristine test coverage is desirable, it is simply not feasible. That is, there are essentially limitless tests as well as edge cases, and as a result, it is simply not humanly possible to know and anticipate the infinite cases. Thus, the plan of our group is to test the core functionality of the application, through extensive test cases, including the edge cases that we are able to anticipate. This centralized, core test system will ensure functionality of the app, and allow remedy of common as well as uncommon bugs or glitches. One idea we processed among us that spawned as the result of the upcoming demos was to provide special end users (such as family and friends) with early alpha and beta builds of the application, allowing users to interact with the app and perhaps root out, identify, analyze, and correct any unforeseen potential issues. Based off those results, our group could then develop additional test coverage to cover any newly thought of edges or use cases, which can improve the overall quality and reliability of the application and thus further reduce the possibility of potential issues in the final version.

### 6.3 Integration Testing Strategy

Integration testing can be done on a local developer machine by emulating the application environment through a virtual machine via VirtualBox and Genymotion, or through a physical Google device. Until the system works in the integration environment it won't be accessible elsewhere. Using GitHub, any history of the application can be stored and filed, this is accomplished by having two branches of source code, a master and dev. Dev is the branch that all new work will be stored in, it will be tested and debugged on a virtual machine or physical Google device. Detailed logs of any system configuration changes will be needed, and once fully tested the source code will be pushed to master. Once the branches are combined the application will be needed to be tested on various developer machines in order to determine if the application can work in any Google Device environment and if a second round of debugging is needed.



## 7 PROJECT MANAGEMENT

---

### 7.1 Merging Contributions for Individual Team Members

Not many issues exist with our overall collaboration and group effort. Communication amongst the team is strong and work is generally evenly divided as all individuals are making efforts to contribute the development process. The group is able to remain in constant communication as a result of the smartphone application called Groupme.

One issue that precipitated the decision of the app functionality, whether it would sway more towards a game or a league. As the use cases were developed the concept for the application as well as the major decision came clear and naturally, which would be the application would primarily be designed as a game.

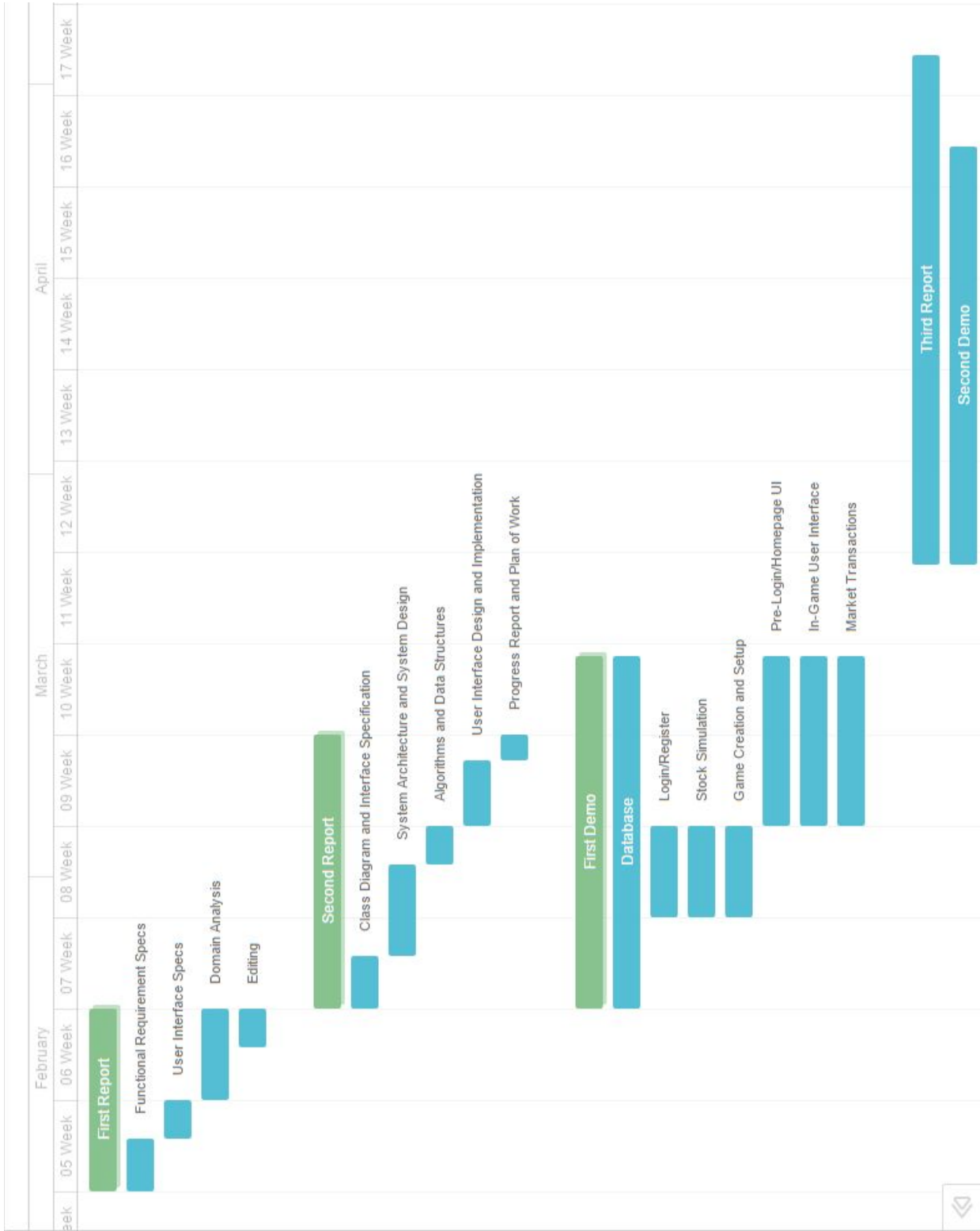
The team has already organized a Github account which will allow for uniform collaboration and development of the project code. The entirety of the project documentation has been developed on Google Docs, which allows for all team members to work cohesively as well as collaboratively, allowing for the work to be merged quite smoothly. Team meetings for important milestones as well as the initial planning of the project have been planned and organized. Alejandro is responsible for reserving conference rooms at a time of the week when every group member is available to meet. A meeting in the reserved conference room is scheduled to develop the base foundation of the project as well as to ensure that all team members possess the proper software, tools, and resources that they will require to efficiently work on their portions of the project.

### 7.2 Project Coordination and Progress Report

The overall progress of the project has been progressing sufficiently. As of this moment, the use cases have not begun implementation. Nevertheless, this will absolutely be remedied in the coming weeks. The Github page will be used to measure any and all evident notions of progress. At the very conception, the group was divided into smaller subgroups, each responsible for completing certain aspects and functionalities of the app. Once the foundation of the project is established, each subgroup will be responsible for the development of their own corresponding portions of the project. Integration of the different parts will occur as the various parts are completed by the three subgroups.



7.3 Plan of Work



## 7.4 Breakdown of Responsibilities

Integration between different pages and across groups will be done throughout the duration of the project as pages that need to be integrated with one another are completed by the different groups and team members.

We will be testing our project using the bottom up strategy. Each team member will be responsible for unit testing their own work and the pages they are responsible for. The final integration of all the parts of the project will then be tested again. All group members will help create the integration test and Kartik will be responsible for executing the test.

### 7.4.1 Group 1: Alejandro and Arjun

The first group in our team will focus on various portions of the project having to do with the user account creation and management, achievements, and social media integration.

**Short Term Goals:** The plan is to make a skeleton of the framework that will be the account page, settings page, and login page. As well as setup a database that will hold account information securely.

### 7.4.2 Group 2: Elisa-Michelle and Bryan

The second group in our team will focus on various portions of the project having to do with creating and joining a private game, joining a public game, leaderboards, and the chat system.

**Short Term Goals:** The plan is to have basic game creation implemented along with an invitation system for other players.

### 7.4.3 Group 3: Kartik, Deep, and William

The third group in our team will focus on various portions of the project having to do with virtual stock simulation, market transactions, and the end-game scenarios .

**Short Term Goals:** The plan is to focus on the storing previous stock information for stock simulations as well as storage and retrieval of user portfolios.

## 8 References

---

- [1] Grant, Peter. "How To Write A Software Proposal | Ehow". *eHow*. N.p., 2017. Web. 5 Feb. 2017.
- [2] "Model-View-Controller." Model-View-Controller. *Microsoft*. N.p., n.d. Web. 04 Mar. 2017.
- [3] "Model-View-Controller". *En.wikipedia.org*. N.p., 2017. Web. 12 Mar. 2017.
- [4] "Event-Driven Architecture". *En.wikipedia.org*. N.p., 2017. Web. 12 Mar. 2017.
- [5] "I Don't Speak Your Language: Frontend Vs. Backend". *Treehouse Blog*. N.p., 2017. Web. 12 Mar. 2017.
- [6] "Explaining HTTP: The Protocol That Makes The Internet Work". *Lifewire*. N.p., 2017. Web. 12 Mar. 2017.