Rags to Riches

Software Engineering

Team 5:

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

April 23, 2017
Github Page

TABLE OF CONTENTS

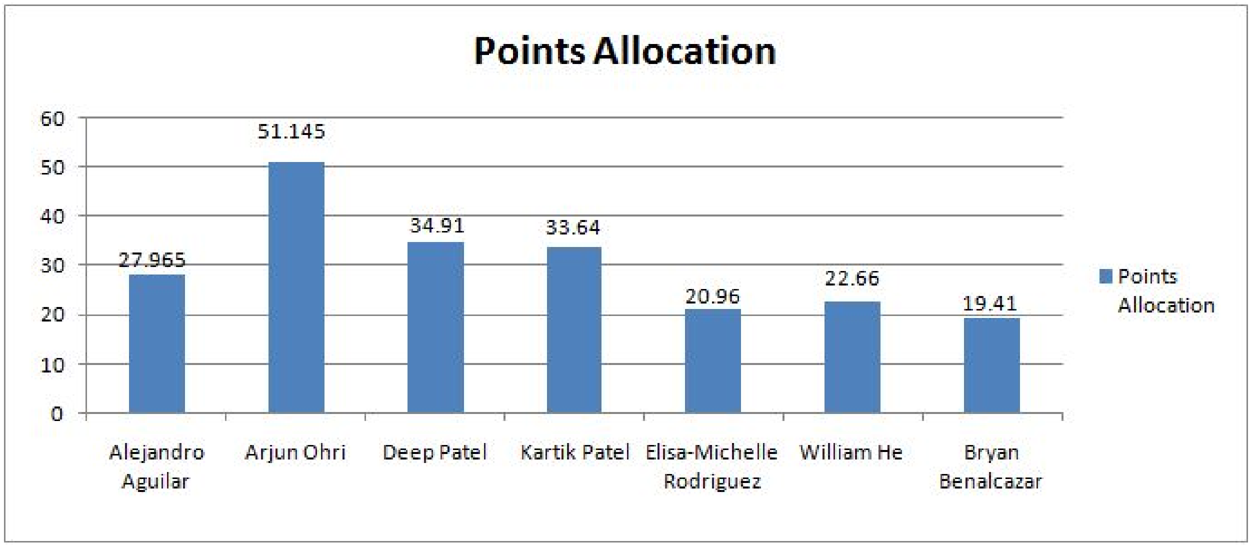
0 INDIVIDUAL CONTRIBUTIONS BREAKDOWN

0.1 Responsibility Matrix

	Team Members						
	Alejandro Aguilar	Arjun Ohri	Deep Patel	Kartik Patel	Elisa-Michelle Rodriguez	William He	Bryan Benalcazar
Summary of Changes (5 pts)	0%	100%	0%	0%	0%	0%	0%
Sec.1: CSR (6 pts)	5 %	30 %	5 %	40 %	5 %	5 %	10 %
Sec.2: Glossary of Terms (4 pts)	5 %	30 %	5 %	40 %	5 %	5 %	10 %
Sec.3: System Requirements(6 pts)	30%	15 %	35 %	0%	0%	20 %	0%
Sec.4: Functional Requirements Specifications (30 pts)	7%	20%	13%	17%	16%	13%	14%
Sec.5: Effort Estimation (4 pts)	50%	0%	50%	0%	0%	0%	0%
Sec.6: Domain Analysis (25 pts)	18%	17.3%	14.3%	18.3%	11.7%	11.7%	8.7 %
Sec.7: Interaction Diagrams (40 pts)	2%	15%	15%	15%	22%	15%	16 %
Sec.8: Class Diagram and Interface Specification (20 pts)	25%	24%	24%	0%	8%	16%	12%
Sec.9: System Architecture and System Design (15 pts)	35%	3%	27%	35%	0%	0%	0%

Sec.10: Algorithms and Data Structures (4 pts)	0%	30%	40%	0%	0%	15%	15%
Sec.11: User Interface Design and Implementation (11 pts)	0%	55%	5%	40%	0%	0%	0%
Sec.12: Design of Tests (12 pts)	10%	26%	16%	0%	16%	16%	16%
Sec.13: History of Work, Current Status, and Future Work (5 pts)	0%	100%	0%	0%	0%	0%	0%
Sec.14: References (5 pts)	8.3%	50%	8.3%	8.3%	8.3%	8.3%	8.3%
Sec.15: Project Management (13 pts)	30%	20%	20%	30%	0%	0%	0 %

0.2 Responsibility Allocation Chart



Summary of Changes

1. Edited domain analysis and included reflective domain diagrams
2. Provided clearer explanation of game functionality in UI diagrams
3. Further clarified particular system requirements
4. Added additional use cases for fully-dressed descriptions
5. Added further sequence diagrams for use cases
6. Expanded set of UI diagrams
7. Included further references used throughout development process
8. Polished use case diagrams and distinguish between certain terms
9. Edited interaction diagrams to reflect class diagrams
10. Clarified algorithm and data structures descriptions
11. Included further comments on UI diagrams
12. Added features as suggested by first demo feedback
13. Included OCL contracts in Class Diagram and Interface specification sections
14. Added history of work, current status, and future work section

1 Customer Statement of Requirements

1.1 Problem Statement

Tabletop games have been a staple in entertainment even before the rise in technology. Nowadays, many favorites have been ported over to virtual systems and are still being enjoyed in their new form. The game should be modelled after these tabletop games, one such in mind is, *Monopoly*. Games like *Monopoly* are competitive by nature and allow for a user to be hooked into a game with other users. The turn-based system employed by these games naturally provides a moment of downtime for each user allowing the user to take in all the information they can and make the best decisions based on this information. The user should be able to join a queue and wait to be matched into a game against other users or be able to join a game with a group of friends through an invitation.

At the start of a game of *Rags to Riches*, each user should start with the same amount of capital so as to not provide any unfair advantages and be given background information on each of the companies relevant to the game. The game should be set in a detailed fantasy world with fantasy companies and products rather than our real world. This will build on the entertainment factor of the game as users will be dropped in an unknown world to explore.

During the game, the users should be able to perform market orders on the various companies available. The user should be able to make informed decisions based on their opponent's decisions as well as the market trend information provided to the user. Once a user has enough influence over a certain company, they should be able to influence the company into making decisions that have a chance of benefitting the user or hurting other users. There should also be a chat system within the game to allow for users to communicate and interact with one another.

If a user goes bankrupt and is completely out of money, they automatically lose and are removed from the game. If a user reaches the amount of money set as the threshold for winning, they are awarded the win for the game. Each user should be shown a quick overview of the game and any achievements they may have unlocked before the game exits.

Achievements should be stored on user accounts and can be utilized as a way of tracking the user's progress in learning the fundamentals. Once a user hits a certain milestone of progress, they should receive a reward that will unlock new abilities or help them in future games. The user should be able to see their progress towards their next milestone/reward and the rewards should be valuable enough that the user would want to reach for those milestones.

Along with being able to see their own achievements, the user should be able to see where they stack up against other players of the game as well as see their lifetime stats. Additionally, the user should be provided with basic account management options relevant to most applications. Administrative tools, such as banning users, should be available for those user's with the correct privileges. The user should have the option of integrating their social media accounts with the game so that they can quickly make posts showing off achievements, bragging about a win, etc. By being able to share to social media the user will be able to get validation of their successes from their peers and utilize that to further their progress through future games.

The user should be able to access *Rags to Riches* quickly and easily. With the prevalence of smartphones these days, it makes sense to design an app for the game so that the user is able to play the game with only the minimal number of strokes. As a result of the game being on an app, the user will be able to play the game on-the-go rather than being tied down to a computer. There is an estimated total of 107.7 million android users, by including the ability of sharing achievements via social media the app can become well known and easier to discover. With this increased ease-of-accessibility, *Rags to Riches* hopes to be an entertaining multiplayer turn-based game for the general public, those who do not have extensive knowledge of stock exchanges and investment management.

2 Glossary of Terms

Rags – A colloquial term for not having a lot of money. The user starts with "rags" and the objective is to make more money than their opponents.

Riches – A colloquial term for having a lot of money. A user turning their starting capital into "riches" is the objective of the game.

User Groups:

- **Player** – A basic user who participates in a game and has control over their account settings.
- **Opponent** – The opposing user who participates in a game. An "opponent" is also a "player".
- **Administrator** – A user with additional privileges allowing for the management of basic users, including banning users from the game.

Game – A multiplayer turn-based game set in a fantasy world where the players manage their investments with the end goal of turning their "rags" into "riches".

Turn – The time where a player can perform actions. Players are able to manage their investments during this time. Opponents can not perform actions at this time.

Bankrupt – A player has no more money remaining. When bankruptcy occurs, the player loses and is removed from the game.

Win – A player successfully turned their "rags" into "riches". The player reached the threshold set for winning the game.

Loss – A player went into bankruptcy or another player won the game.

Capital – The amount of money each player starts with at the beginning of a game.

Stock – A type of security that signifies ownership in a corporation and represents a claim on part of the corporation's earnings and assets.

- **Ask Price** – Price at which trader will sell a stock
- **Bid Price** – Price at which a trader will buy a stock

Portfolio – An account of all the assets associated with a player in the game. Each player will have their own portfolio.
Investment – The purchase of stocks that are not to be consumed immediately but rather to be used in the future to create wealth.

Order – An investor must place an order to buy or sell an asset.

- **Buy** – An order to purchase an amount of stock.

- **Sell** – An order to sell an amount of stock.

- **Short** – A sell order performed using borrowed stocks.

- **Cover** – A buy order performed to return previously loaned stocks.

- **Limit** – An order that sets the maximum or minimum at which you can buy or sell stocks.

- **Stop** – An order that will only happen at a defined price called the stop price.

Influence – The greater the amount of stocks and percentage a user has in a company, the greater influence they have. Having a large influence in a company provides the user with a greater variety of decisions.

Decision – When it is the user's turn, they can make a decision. Decisions include but are not limited to the buying of stocks, selling of stocks, etc.

Milestone – A set goal that shows the player is making progress

Achievement – A milestone reached by the player resulting in a reward.

Reward – An item unlocked through an achievement.

3 System Requirements

3.1 Enumerated Functional Requirements

Identifier	Priority	Requirement
REQ-1	5	The system will allow new users to register accounts.
REQ-2	5	The system will allow returning users to login to their accounts.
REQ-3	5	The system will keep track of user account information.
REQ-4	3	The system will allow users to manage their account: ● Change password ● Change contact info ● Notification settings
REQ-5	1	The system will allow users to check their achievements. There will be a button on the homescreen of a user that users can click to view achievements they've been able to complete.
REQ-6	3	The system will allow administrators to manage user accounts. The admin should be able to edit account information online. Preferably by a database that holds the data.
REQ-7	4	The system will provide an initial tutorial interface for newly registered users.
REQ-8	4	The system will create a game based on a variety of settings. The game's initial currency can be changed.
REQ-9	4	The system will allow players to host a private game with custom settings: ● Time allowed per turn ● Starting capital ● Threshold for winning
REQ-10	4	The system will allow players to invite friends to their private games using a 4-character passcode.
REQ-11	4	The system will allow players to join a private game with their friends.
REQ-12	2	The system will allow players to join a queue for a public game with other players using default settings.

REQ-13	4	The system will allow players to view various game-related information, such as standings, player portfolios, and ownership percentage.
REQ-14	5	The system will automatically determine market data for the game. Data will be determined from stock information of previous years.
REQ-15	5	The system will allow users to initiate market orders: ● Buy ● Sell ● Short ● Cover ● Limit ● Stop
REQ-16	3	The system will allow players to influence companies once they own a certain percentage of the company. With influence in a company, a player can deny other players from purchasing stock in that company.
REQ-17	1	The system will allow players to communicate with each other inside of a game. There should be a button on the game view that allows players to talk to other players in the game.
REQ-18	4	The system will recognize win and loss scenarios for the game.The player with the highest capital at the end of the game will be declared the winner.
REQ-19	1	The system will allow users to integrate their social media accounts to post messages and codes for private game invites.
REQ-20	1	The system will provide a leaderboard for users to check how they stack up against other players.

3.2 Enumerated Nonfunctional Requirements

Identifier	**Priority**	**Requirement**
REQ-21	5	The system will be able to run on Android devices.
REQ-22	4	The system will be lightweight to provide fast performance even on low end devices.
REQ-23	3	The system will have a similar theme across the stock information page, the settings page, and the game page.
REQ-24	4	The system will securely store personal user information.

REQ-25	5	The system will store all data and information in a database with no storage being done on the user's device.
REQ-26	3	The system will allow the user to navigate the app in the fewest number of strokes possible.
REQ-27	3	The system will be active 24/7.

3.3 On Screen Appearance Requirements

Identifier	Priority	Requirement
REQ-28	3	**Initial Landing page** – This is shown on first boot. The game and its terms are explained here. Its the tutorial page.
REQ-29	5	**Registration page** – Users will be able to create a new account.
REQ-30	5	**Login page** – Users will be able to login to an existing account
REQ-31	4	**Home Page** – Users will be able to create a game, join a game through an invite, or find an online game. They will also be able to manage their settings and achievements.
REQ-32	4	**Settings page** – Users will be able to change their password, their contact info, their notification settings, and volume.
REQ-33	4	**Create A Private Game page** – User can create a private game with custom settings: time allowed per turn, starting capital, threshold for winning, etc.
REQ-34	3	**Join A Private Game page** – User can enter a code given to them by a friend to join an existing private game.
REQ-35	5	**Game Page** – Users will be able to see their current assets, company information, current game standings, current market trends, each player's wealth, and a button to move on to the next player's turn.

4 Functional Requirements Specification

4.1 Stakeholders

A stakeholder is defined as a party that has an interest in a company, and can either affect or be affected by the business. For the majority of the time, the primary stakeholders in a typical corporation are its investors, employees, and customers. Stakeholders can be internal or external. Internal stakeholders are those parties whose interest in a company is tied to a direct relationship, such as employment, ownership, or investment. On the other hand, an external stakeholder are those who are not directly tied to the company yet are affected by the company's actions and business outcomes.

The target audience for Rags to Riches is students, novice investors, and anyone who wishes to gain a working knowledge of economics and the stock market. Even those with a greater understanding of the stock market who wish to sharpen and hone their skills and review some of the conceptual background can benefit from using this app. Users who wish to eventually dive in the real stock market will have benefitted from the utilization of the app and be armed with an experiential advantage over those who did not. Rags to Riches can be expanded to educate larger swaths of aspiring investors in the form of being used as an educational resource in high school and college level economic courses. The various features than the app offers will likely garner word of mouth awareness from those who use the app, who can encourage like minded individuals to also experiment around with it.

Rags to Riches is a non-profit, free application with the sole purpose of education. At no point are there plans to incorporate advertisements to generate revenue. Rags to Riches believes that advertisements are distracting and cumbersome, which detract from the user's experience. Because the app's sole purpose is that of education, cluttering the user's interface with ads will distract them from the best experience that can be provided, and perhaps hamper their overall education. Another strategy Rags to Riches strongly opposes is the inclusion of in app purchases or microtransactions. These "pay to win" strategies are potentially game breaking, unfair, and lessen the overall experience. Rags to Riches strives to provide a equal, balanced, enjoyable, and educational experience to all users.

4.2 Actors and Goals

- **Player –** Initiating Actor
  Goals:
  1. To create an account
  2. To login into their account
  3. To create a private game
  4. To find and join a public game
  5. To check their achievements
  6. To perform market transactions in their game
  7. To view the portfolios and balances of players in their game
  8. To view the standings of their game
  9. To communicate with other players in their game
  10. To post achievements or progress updates to their social media account

- **Game Administrator –** Initiating Actor
  Goals:
  1. To set the initial rules for the private game they create
  2. To manage the private game they create

- **Game** – Participating Actor

- **Stock Simulator –** Participating Actor

- **Account Database –** Participating Actor

4.3 Use Cases

4.3.1 Casual Description

UC-1: Register

Allows a new user to register an account for the application. The user will be required to login using this information in the future.
Requirements: REQ-1, REQ-3, REQ-7

UC-2: Login

Allows a registered user to login to their account for the application and link to social media. The system will retrieve all of the user's information and personal settings. The user will be automatically logged in to their account unless they have manually logged out.
Requirements: REQ-2, REQ-3

UC-3: ManageAccount

Allows a registered user to edit their account information, such as account password and email address, as well as manage their application settings and view their achievements.
Requirements: REQ-4

UC-4: FindPublicGame

Allows a registered user to join a queue for a public game. Once four players are found in the queue, they players are removed from the queue and placed in a game.
Requirements: REQ-8, REQ-12

UC-5: CreatePrivateGame

Allows a registered user to create a private game to be played with friends. A 4 character passcode is generated to invite other users to the game.
Requirements: REQ-8, REQ-9, REQ-10

UC-6: JoinPrivateGame

Allows a registered user to join a private game hosted by a friend by entering the 4-character code sent to them.
Requirements: REQ-10, REQ-11

UC-7: ViewMarketData

Allows a player to view information regarding the various companies and their market history to aid them in their decision-making process.
Requirements: REQ-13, REQ-14

UC-8: PlaceMarketOrder

Allows a player to place a market order to buy or sell stocks in the game. The order is placed during the user's turn and changes are reflected immediately.
Requirements: REQ-14, REQ-15

UC-9: ViewPortfolio

Allows a player to view the current assets of any player inhe game. The user is able to see information regarding other user's influence in companies to aid them in their decision-making process. User is able to chat with other players if they wish to communicate with those players.
Requirements: REQ-13, REQ-14

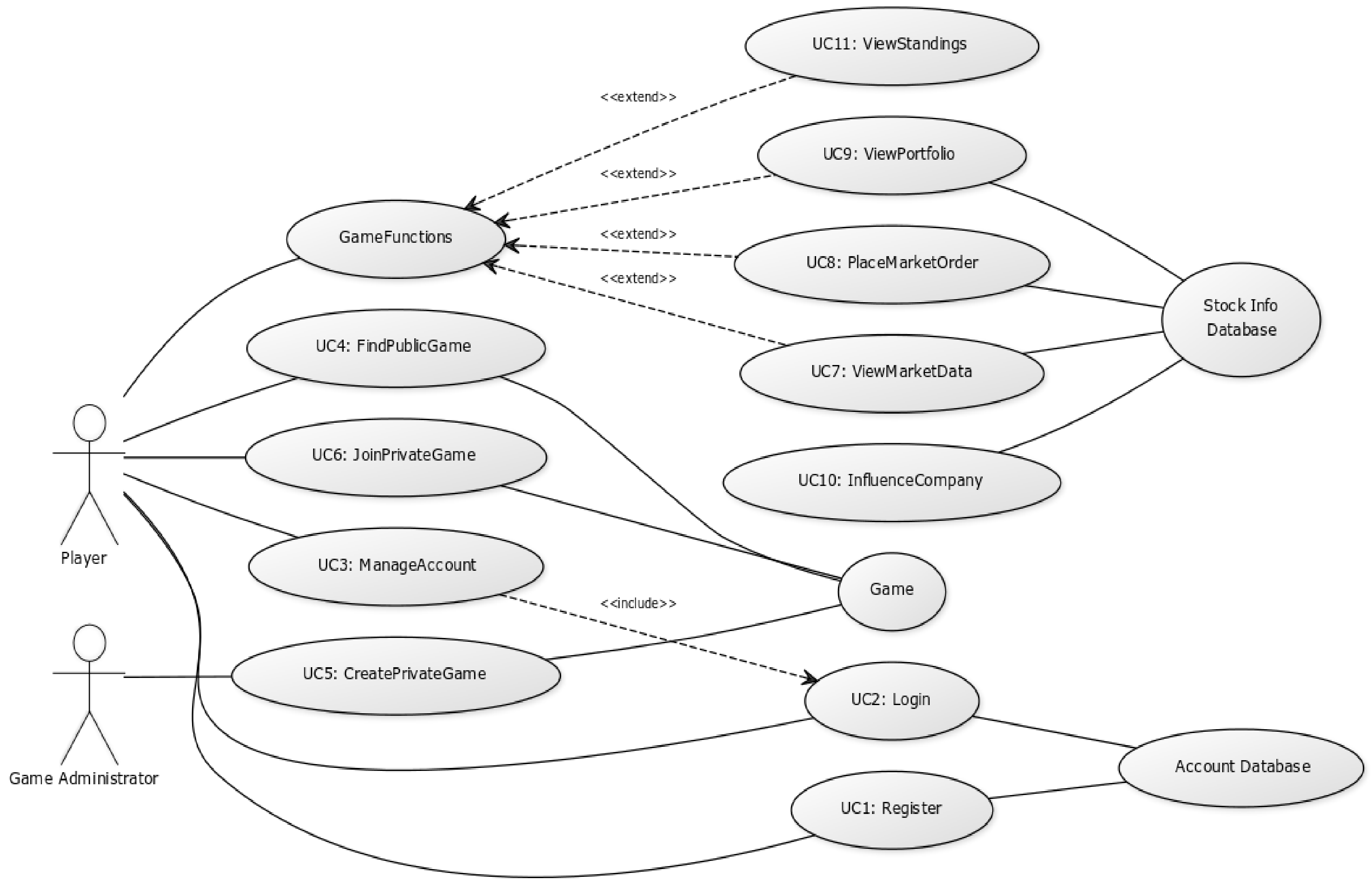
UC-10: InfluenceCompany

Allows a player to influence a company once they own a certain percentage of the company. The user is able to perform additional actions based on this.
Requirements: REQ-13, REQ-14, REQ-16

UC-11: ViewStandings

Allows a player to view the current standings of the players in the game based on net worth. The user is also able to see a brief overview of another player's assets.
Requirements: REQ-13

4.3.2 Use Case Diagram



4.3.3 Traceability Matrix

REQ	**PW**	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11
REQ-1	**5**	**X**										
REQ-2	**5**		**X**									
REQ-3	**5**	**X**	**X**									
REQ-4	**3**			**X**								
REQ-5	**1**											
REQ-6	**3**											
REQ-7	**4**	**X**										
REQ-8	**4**				**X**	**X**						
REQ-9	**4**					**X**						
REQ-10	**4**					**X**	**X**					
REQ-11	**4**						**X**					
REQ-12	**2**				**X**							
REQ-13	**5**							**X**		**X**	**X**	**X**
REQ-14	**4**							**X**	**X**	**X**		**X**
REQ-15	**5**								**X**			
REQ-16	**3**											**X**
REQ-17	**1**											
REQ-18	**4**											
REQ-19	**1**											
REQ-20	**1**											
Max PW		**5**	**5**	**3**	**4**	**4**	**4**	**5**	**5**	**5**	**4**	**5**
Total PW		**14**	**10**	**3**	**6**	**12**	**8**	**9**	**10**	**9**	**4**	**12**

4.3.2 Fully-Dressed Description

Use Case UC-1: Register	
Related Requirements:	REQ-1, REQ-3, REQ-7
Initiating Actor:	Player
Actor's Goal:	To create an account.
Participating Actors:	Account Database
Preconditions:	The user is not logged into an account.
Postconditions:	The user successfully registers an account and can use it to log in to the app from any compatible device. The user account database is updated with the new player's account information.
Flow of Events for Main Success Scenario:	
1.	-> The user is on the initial landing page.
2.	-> On the initial landing page, the player clicks the register option
3.	-> The player enters their desired account name, their password, and their password again to confirm it.
4.	<- If the user confirmed their password correctly, the user account database is updated with the new information and the system welcomes the new user.
Flow of Events for Account Name is taken or Passwords do not match:	
1a.	-> The user enters an account name that is already taken.
1b.	-> The user's password does not match the confirmation password.
2.	<- The user is provided with an error message explaining the situation and is encouraged to try again.

Use Case UC-2: Login	
Related Requirements:	REQ-2, REQ-3

Initiating Actor:	Player
Actor’s Goal:	To log into an account.
Participating Actors:	Account Database
Preconditions:	The user is not logged into an account.
Postconditions:	The user enters a login ID and password and is successful in logging into the app. The app then is successful in receiving account data data from the database.
Flow of Events for Main Success Scenario:	
1.	-> The user is on the login page.
2.	-> The user enters a login ID and password and clicks login.
3.	<- If the user confirmed their account info correctly, the app downloads the account information if not already on the app to the users device, and the user is presented with the home screen that allows them to join a public or private game.
Flow of Events for Account info not matching.	
1a.	-> The user enters incorrect login credentials.
1b.	-> The user fails to enter the login credentials correctly 4 more times.
2.	<- The user is provided with an error message that they’ve entered the wrong credentials 5 times and will have to wait 1 minute before attempting to log in again.

Use Case UC-3: Manage Account	
Related Requirements:	REQ-4

Initiating Actor:	Player
Actor’s Goal:	To edit account information, such as account password, email address.
Participating Actors:	Account Database
Preconditions:	The user is logged into an account.
Postconditions:	The user successfully changes their email address, their password, or their social media information.
Flow of Events for Main Success Scenario:	
1.	-> The user is on their homescreen.
2.	-> The user clicks the settings icon and chooses to change either the saved email address, the password, or social media.
3.	-> The player enters their desired new account email address, their password, or social media. User enters the information again in another field to confirm.
4.	<- If the desired change was successful, the app presents the user with a success dialog.
Flow of Events for Failure of Account Information Change.	
1a.	-> The user enters a new email address, a new password, or a new social media account.
2.	<- The user is provided with information that either their account email address is already taken, the password is not at least 8 characters, or the social media credentials are incorrect.

Use Case UC-4: FindPublicGame	
Related Requirements:	REQ-8, REQ-12
Initiating Actor:	Player
Actor's Goal:	To find and join a public game
Participating Actors:	Player, additional players, Queue system
Preconditions:	Players have valid accounts. Queue system functioning.
Postconditions:	Player is successfully processed through the queue and is placed into a public game against other online players.
Flow of Events for Main Success Scenario:	
1.	-> Player clicks the find public game option on the main screen.
2.	<- Player enters the queue that sorts players into online games on "first-come, first-served" basis.
3.	<- After being successfully processed through the queue, the player is placed into a public game where they play against other online players.
Flow of Events for Extensions (Alternate Scenarios): Queue System Down	
1.	-> Player clicks the find public game option on the main screen.
2.	<- The queue system is down or overloaded at the moment and is unable to place the player into a public game. The player receives an error message apologizing and explaining the situation.
The arrows on the left indicate the direction of interaction: -> Actor's action; <- System's reaction	

Use Case UC-5: CreatePrivateGame	
Related Requirements:	REQ-8, REQ-9, REQ-10
Initiating Actor:	Game Administrator
Actor’s Goal:	To create a private game
Participating Actors:	Game Administrator and players
Preconditions:	The game administrator has a valid account.
Postconditions:	The game presents a 4 digit code that the game administrator can share with other players who then use the code to join the private game.
Flow of Events for Main Success Scenario:	
1.	-> Game administrator creates a game from the home screen.
2.	-> Game administrator changes the settings of the game, such as number of players and starting currency amount and chooses the private game option.
3.	<- Game presents the 4 digit code that the game administrator can share.
Flow of Events for Different Sharing Method:	
1.	-> Game administrator creates a game from the home screen.
2.	-> Game administrator changes the settings of the game, such as number of players and starting currency amount and chooses the private game option.
3a.	<- Game administrator chooses the share the 4 digit code by using a social media such as Twitter or facebook and system shares it through this medium.

Use Case UC-6: JoinPrivateGame	
Related Requirements:	REQ-10, REQ-11
Initiating Actor:	Player
Actor’s Goal:	Join a public game
Participating Actors:	Player, additional players
Preconditions:	A user is on the “Join private game” screen.

Postconditions:	The user has entered the waiting queue for the private game.
Flow of Events for Main Success Scenario:	
1.	-> A player is on the join private game screen and enters the correct private code.
2.	-> Player enters the private game queue. The private game will begin once 2 other players enter the private game by entering the passcode.
Flow of Events for Failure to Enter Private Game:	
1.	-> Player enters an incorrect private game code.
2.	<- Dialog comes up telling the user the code is incorrect and prompts the user to try again.

Use Case UC-7: ViewMarketData	
Related Requirements:	REQ-13, REQ-14
Initiating Actor:	Player
Actor's Goal:	To view stock information.
Participating Actors:	Stock Simulator
Preconditions:	A user is playing the game.
Postconditions:	The user views past stock info.
Flow of Events for Main Success Scenario:	
1.	-> A player is on the game page.
2.	-> Player wants to view past stock trends about a specific stocks and clicks a stock
3.	<- Dialog comes up with the stock information.
Flow of Events for Inability to view Stock Info.	
1a.	-> Player wants to view past stock trends about a specific stocks and clicks a stock
2.	<- Error dialog comes up, asking user to try again

Use Case UC-8: PlaceMarketOrder	
Related Requirements:	REQ-14, REQ-15
Initiating Actor:	Player
Actor's Goal:	To perform market transactions in their game
Participating Actors:	Players, Stock Information Database
Preconditions:	Each player that has a valid account has its own capital to buy stocks.
Postconditions:	Each player is presented with the option of buying more stocks if capital is available. Also, each player is able to sell their current stocks in order to buy more. Also, given the other market orders, each player can perform a short, cover, limit, and stop during each turn in the game.
Flow of Events for Main Success Scenario:	
1.	-> Player with valid account enters new game, given a capital, and time limit for each turn.
2.	-> Each player in the game at their respective turn with their given capital chooses to place a market order within their given time limit for their turn.
3.	<- The system performs the market transaction and updates the information on the player's screen.
Flow of Events for Market Transaction Error:	
1.	-> Player tries to place a market order that exceeds the amount of money they have.
2.	<- The system can not perform this transaction and returns an error.

Use Case UC-9: ViewPortfolio	
Related Requirements:	REQ-13, REQ-14
Initiating Actor:	Player
Actor's Goal:	To view the portfolios of players in their game.
Participating Actors:	Player, Desired Player in current game, Account Database
Preconditions:	The player and desired players both have valid accounts and are both present in the same current game session
Postconditions:	The player is presented with the portfolio of the player they desired to view in their current game, which displays their information about their current stocks, achievements, and statistics.
Flow of Events for Main Success Scenario:	
1.	-> Player enters a valid game session
2.	-> Player clicks on their desired player in a session, and clicks on "View Portfolio".
3.	<- The player is presented with the portfolio of their chosen player. When they are finished viewing it, they press the x icon in the top right.
Flow of Events for Account Database Malfunction:	
1.	-> Player enters a valid game session
2.	-> User clicks on desired player and then upon "View Portfolio"
3.	<- The Account Database is down and cannot bring up that player's portfolio information. The player receives a message of the error

Use Case UC-10: InfluenceCompany	
Related Requirements:	REQ-13, REQ-14, REQ-16

Initiating Actor:	Player
Actor's Goal:	To gain influence over a company by buying a majority of stake in the company.
Participating Actors:	Player
Preconditions:	A user is on the game screen
Postconditions:	A user gains influence over a company and the ability to deny other players from buying stake in a company.
Flow of Events for Main Success Scenario:	
1.	-> A player buys enough stock in a company to get a stake.
2.	-> Player then gets the option to deny other players from buying that company's stock.
Flow of Events for Failure to gain influence.	
1.	-> Player attempts to buy a majority stake in a company but fails to do because they don't have enough money to buy a lot of stock.

Use Case UC-11: ViewStandings	
Related Requirements:	REQ-13
Initiating Actor:	Player
Actor's Goal:	To view current standings of an ongoing game.
Participating Actors:	Player
Preconditions:	A user is on the game screen
Postconditions:	A user is able to view the standings of each player in the game.
Flow of Events for Main Success Scenario:	
1.	-> User clicks the standings button in game.
2.	-> Dialog popups with the standings of the 4 player in the game and their current capital value.
Flow of Events for Failure to view standings.	

1.	-> The standings data may not load properly, and an error will show on the dialog indicating so, prompting the user to reclick the standings button on the game page.

4.4 System Sequence Diagrams

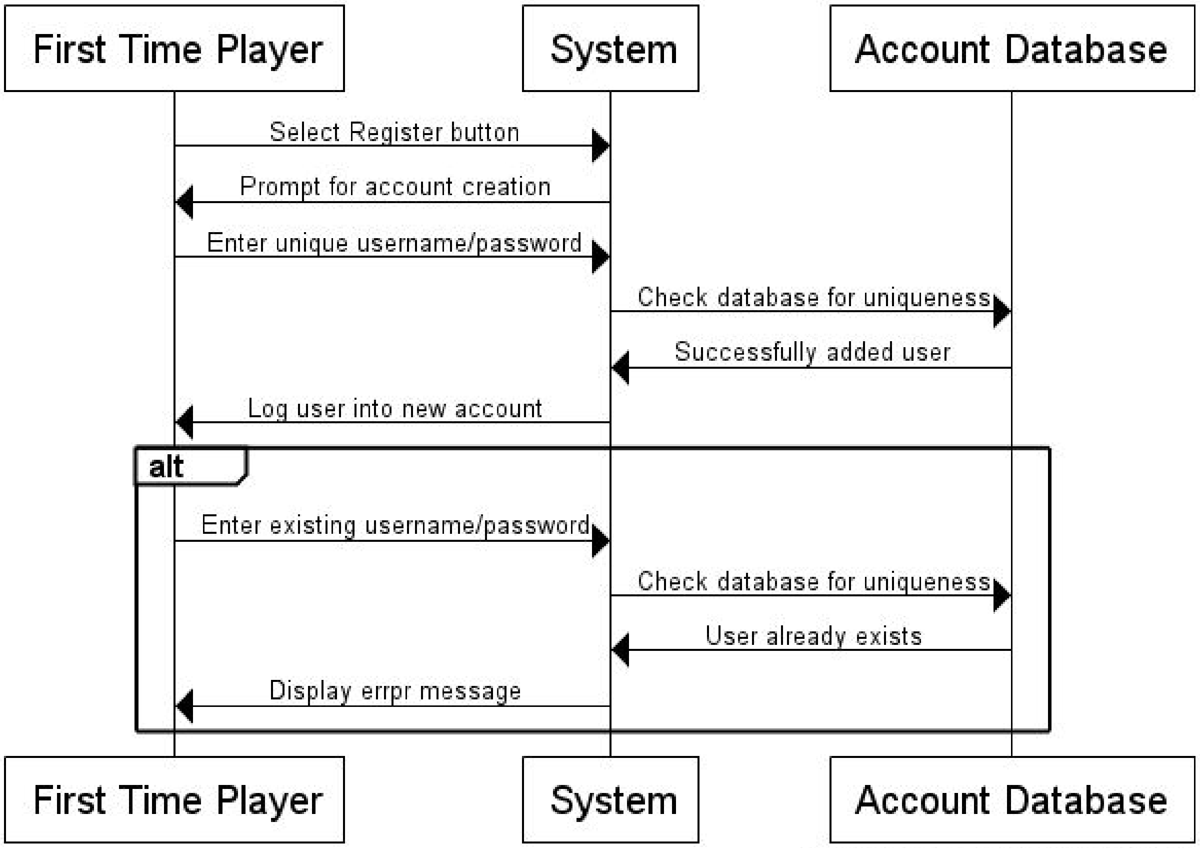

First Time Player
System
Account Database
Select Register button
Prompt for account creation
Enter unique username/password
Check database for uniqueness
Successfully added user
Log user into new account
alt
Enter existing username/password
Check database for uniqueness
User already exists
Display errpr message
First Time Player
System
Account Database

Figure 3.1: UC-1 Register

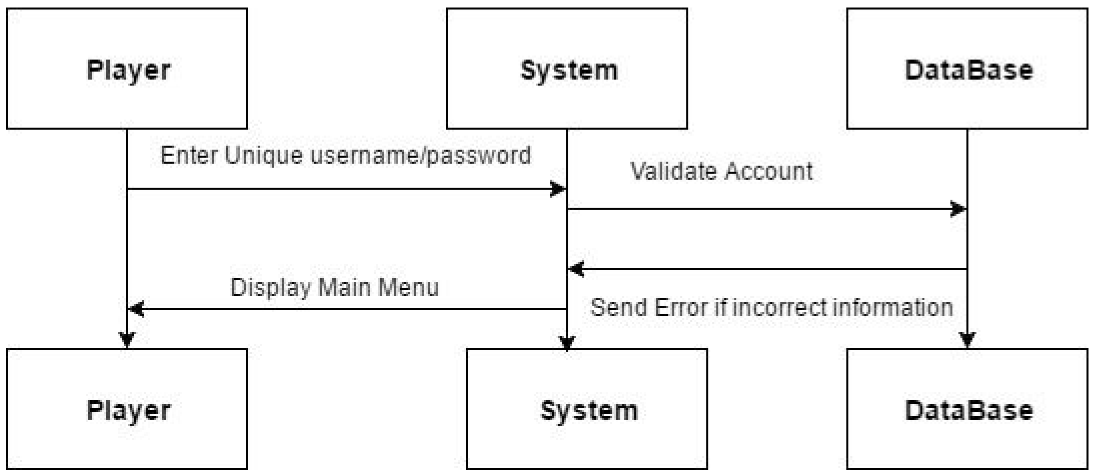


Figure 3.2: UC2 Login

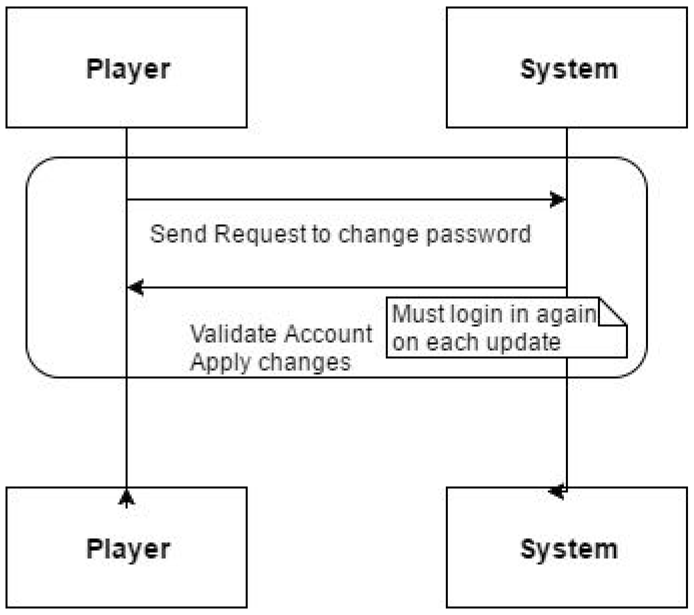

Player
System
Send Request to change password
Must login in again on each update
Validate Account
Apply changes
Player
System

Figure 3.3: ManageAccount

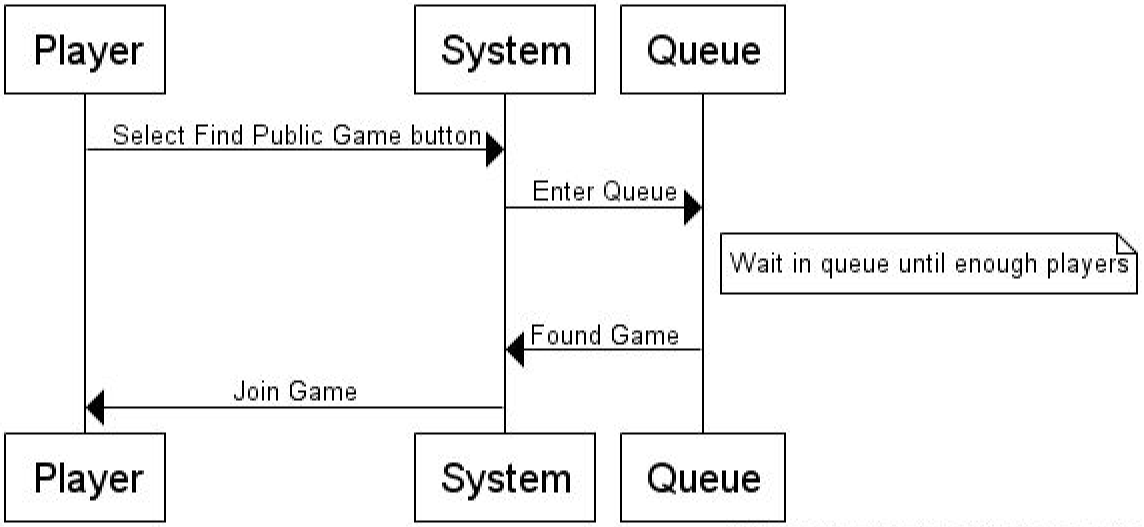

Player
System
Queue
Select Find Public Game button
Enter Queue
Wait in queue until enough players
Found Game
Join Game
Player
System
Queue

Figure 3.4: UC-4 FindPublicGame

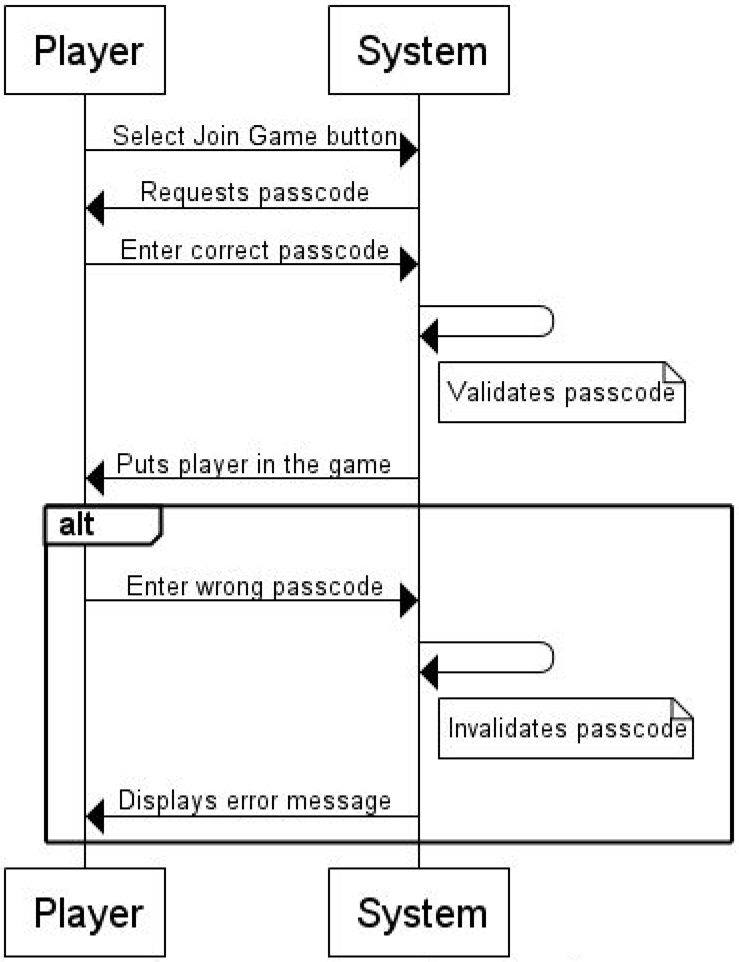


Figure 3.6: UC-6 CreatePrivateGame

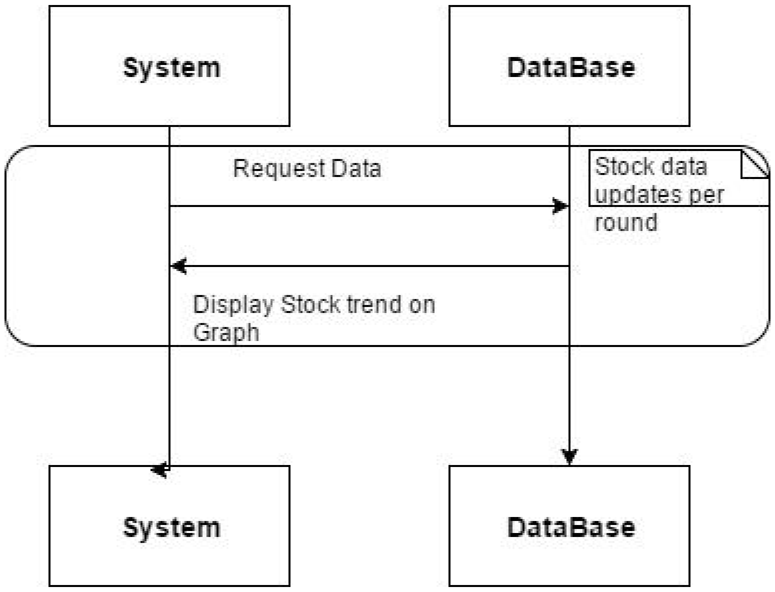


Figure 3.7: UC-7 ViewMarketData

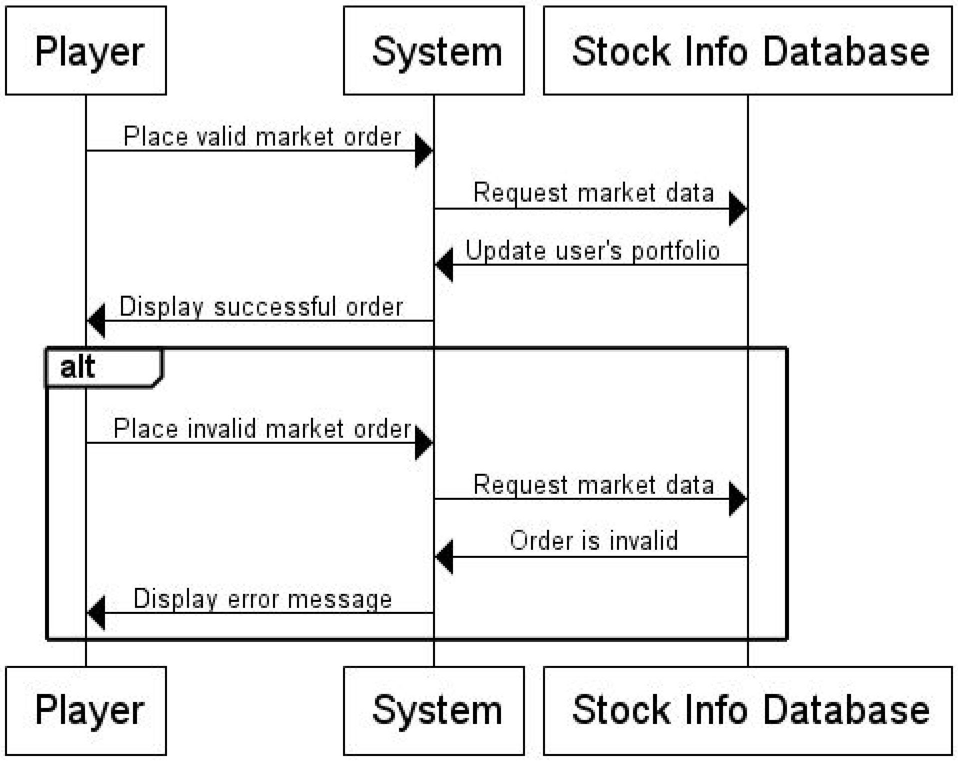


Figure 3.8: UC-8 PlaceMarketOrder

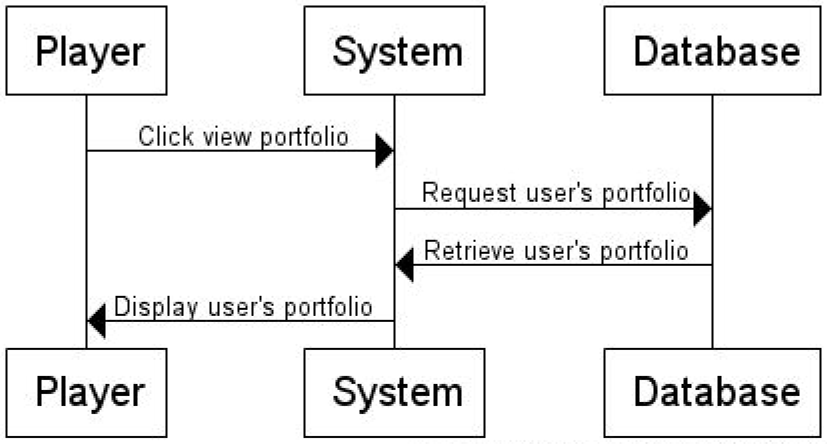


Figure 3.9: UC-9 ViewPortfolio

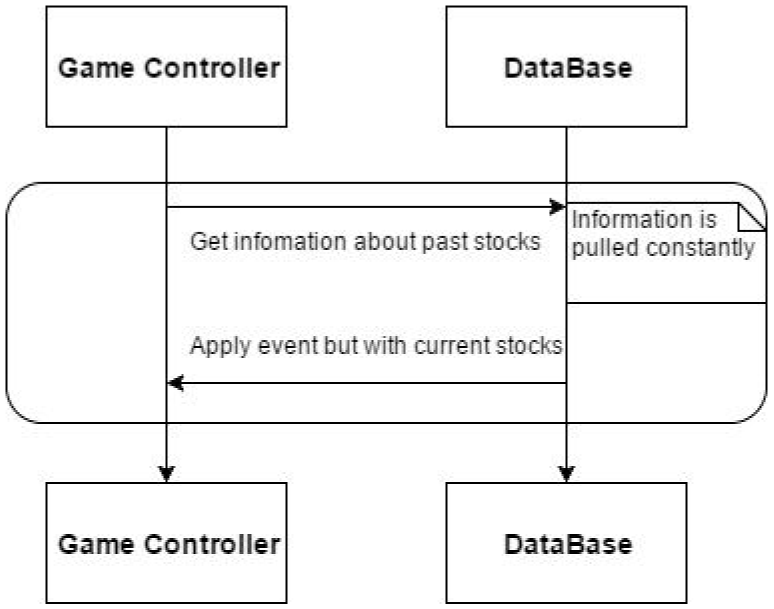

Game Controller
DataBase
Get infomation about past stocks
Information is pulled constantly
Apply event but with current stocks
Game Controller
DataBase

Figure 3.10: UC-10 InfluenceCompany

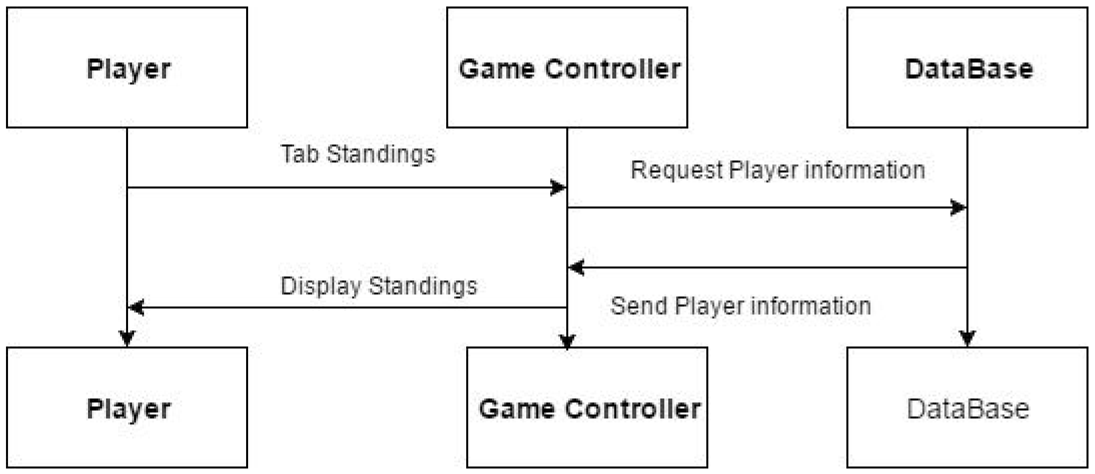


Figure 3.11: UC-11 ViewStandings

5 Effort Estimation

UCP = UUCP (Unadjusted Use Case Points) x TCF (Technical Complexity Factor) x ECF (Environmental Complexity Factor)

- Unadjusted Use Case Points: The measurement of functional complexity requirements
- Technical Complexity Factor: The measurement of nonfunction complexity requirements

5.1 Unadjusted Use Case Points

- UUCP = UAW + UUCW
    - UAW: Combined complexity of all actors in use cases
    - UUCW: Total # of activities contained in all use cases

5.1.1 Unadjusted Actor Weight

Actor	Description of Characteristics	Complexity	Weight
Player	The player interacts with the application and plays the game.	Complex	3
Game Administrator	The Game administrator creates a private game for others to join.	simple	1
System Administrator	The System Administrator makes sure players are removed from game queue as games become playable, it also makes sure profiles and accounts are kept updated.	Complex	3
Player/Stocks Database	The stock information and accounts information is stored in this database.	Average	2
Web Server	The Web server contains database where accounts, stock, and trends	Average	2

UAW = (2 * complex) + (2 * average) + (1 * simple) = 2*3 + 2*2 + 1*1 = 11

5.1.2 Unadjusted Use Case Weight

Use Case	Action	Description	Clicks or Steps	Comple xity	Weight
UC-1	Register	Press “Register” button Fill out text boxes with correct information Press “Register” button to submit (will bring you to main menu)	3	Simple	5
UC-2	Login	Press on “Username” text field Press any keys for username (6-8 characters) Press the enter key to move to the next text field Press any keys for password (6-8 characters)	4	Average	10
UC-3	Create Private Game	Press “Create Game” In Game mode A screen with a four digit number will be prompted Select sharing method (i.e Facebook. Contacts, private message) Select Start game	4	Complex	15
UC-4	Join Public Game	Click “Join Public Game”, in Game Mode Player is placed into queue that will process them and place them into an online game	1	Simple	5
UC-5	Place market orders	Press desired stock A page will be prompted showing stock trends	2	Simple	5

		select buy will be on right side Select amount of shares, and press the confirm button			
UC-6	Manage Account	Click on Settings “gear” at top right of home screen	1	simple	5
UC-7	View Portfolio	Press on avatar, or an opponent's’ avatar New screen will be prompted, select View Portfolio”. This brings up information about the clicked upon user. (Current stocks, achievements, statistics)	2	complex	15
UC-8	View Achievements	Click on trophy at top right of home screen	1	average	10

UUCW(Stock Game) = 4 x Simple + 2 x Average + 2 x Complex = 4 * 5 + 2 * 10 + 2 * 15 = 70

5.1.3 Computing Unadjusted Use Case Points

UUCP = UAW + UUCW
UUCP = 11 + 70 = 81

5.2 Technical Complexity Factor

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight * Perceived Complexity)
T1	Android based App	2	5	2 * 5 = 10
T2	Light and speedy	1	2	1 * 2 = 2
T3	Efficient for the user, and non technical	1	2	1 * 2 = 2
T4	Simple internally, for other developers to be able to add on to the system	1	2	1 * 2 = 2
T5	Reusable code	1	4	1 * 4 = 4

T6	Simple to install, from the android app store	0.5	0	0.5 * 0 = 0
T7	Easy to use	0.5	2	0.5 * 2 = 1
T8	Portable because of the smartphone android app	2	0	2 * 0 = 0
T9	Code Easy to change	1	3	1 * 3 = 3
T10	Concurrent use	1	5	1 * 5 = 5
T11	User data security	1	3	1 * 3 = 3
T12	No direct access from a third party	1	0	1 * 0 = 0
T13	No user training	1	0	1 * 0 = 0
Technical Factor Total				32

TCF = Constant1 + Constat2*Technical Factor Total = 0.6 + (0.01*32) = 0.92

5.3 Environment Complexity Factor

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor (Weight * Perceived Complexity)
E1	Android based project familiarity	1.5	1	1 * 1.5 = 1.5
E2	Application Problem Experience	0.5	1	1 * 0.5 = 0.5
E3	Paradigm Experience	1	3	1 * 3 = 3
E4	Beginner level analyst	0.5	1	0.5 * 1 = 0.5
E5	Motivation	1	4	1 * 4 = 4
E6	Stable requirements	2	5	2 * 5 = 10
E7	Part Time staff	-1	0	-1 * 0 = 0

E8	Difficult programming language	-1	2	-1 * 2 = -2
Total				15.5

ECF = Constant1 + Constant2 *Enviormental Factor Total = 1.4 + (0.03*15.5) = 1.865

5.4 Calculating Use Case Points

UCP = UUCP x TCF x ECF =	**137.86**
UUCP =	**81**
TCF =	**0.92**
ECF =	**1.85**

5.5 Deriving Project Duration from Use Case Points

We need to utilize the UCP and Productivity Factor (PF) to determine duration of successfully implementing our project.The equation for Duration is the following:

$$\textit{Duration} = \textit{UCP} \times \textit{PF}$$

$$\textit{Duration} = 137.86 * 28$$

Assuming a the factor is 28, then our is duration is 3860 person-hours for the development of our project.If our team of seven developers spent a minimum of 18 hours per week on project tasks, then that would mean 126 hours per week are spent on the project. If the duration is divided by the total of hours a week our team should be spending then we would be allotted approximately 31 weeks to have well polished product. A semester is usually about 15 weeks which would allow for 16 more weeks to test, debug and added or remove concepts in order to fully successfully implement our project. This is only true however, assuming the productivity factor is 28 and 18 hours is spent a day by 7 people on project tasks.

6 Domain Analysis

6.1 Domain Model

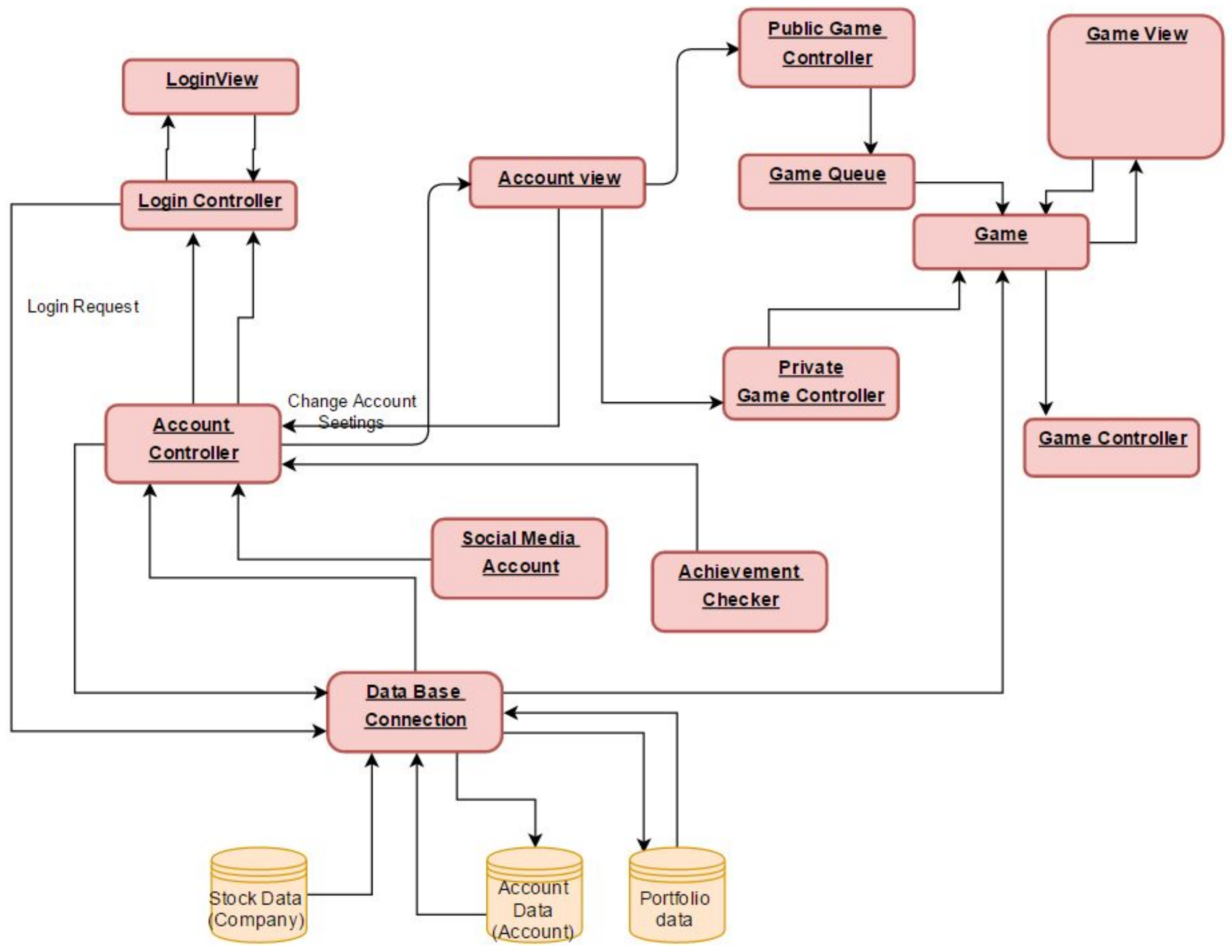


Figure 6.0: Domain Model

Figure 6 shows the Rags to Riches Domain Model. The subsequent chart lists what every part of this diagram is responsible for and the diagrams describe specific user cases that utilize the different parts of this diagram.

Responsibilities Description	Concept Name
Allows a new user to register for an account. The user will be required to login using this information in the future.	Login Controller
Allows a registered user to login to their account. The system will retrieve all of the user's information and personal settings. The user will be automatically be logged in with their account unless they have manually logged out.	Login Controller
Displays login screen to the user and allows them to register or access their account.	Login View
Container for user's account data, such as their login information, match settings, and account settings.	Account
Allows a registered user to edit their account information, such as account password and email address, as well as manage their application settings.	Account Controller
Allows a user to check their achievements and rewards.	Account Controller
Displays option to the user when logged-in to their account but not in a game yet.	Account View
Container to hold a queue of users searching for games	Game Queue
Allows a registered user to join a queue for a public game. Once four players are found in the queue, they players are removed from the queue and placed in a game.	Public Game Controller
Allows a registered user to create a private game to be played with friends. A 4 character passcode is generated to invite other users to the game.	Private Game Controller
Allows a registered user to join a private game hosted by a friend by entering the 4-character passcode sent to them via other users.	Private Game Controller
Container to hold game-related information, such as game settings, game players, and game stats.	Game
Display relevant game information to the user in an organized and easy manner.	Game View

Allows a player to view information regarding the various companies and their market history to aid them in their decision-making process to place a market order..	Game Controller
Prepare a database query to perform the given stock transaction and update the game values as the game moves on.	Database Connection
Allows a player to view the current assets of another player in the game. The user is able to see information regarding other user's influence in companies to aid them in their decision-making process to place a market order.	Game Controller
Allows a player to view the current standings of the players in the game based on net worth. The user is also able to see a brief overview of another player's assets.	Game Controller
Allows a player to influence a percentage of the company. The user is able to perform additional actions based on the influence.	Game Controller
Container to hold information related to a particular company in the game, such as the company description, history, current market value, market trends, etc.	Company
Container to hold information related to the user's portfolio, such as their assets.	Portfolio
Allows players to communicate with each other within the game.	Chat Controller
Container to hold information related to a user's social media account.	Social Media Account
Allows user to post messages onto their social media account(s).	Account Controller
Container to hold information related to a particular achievement.	Achievement
Allows a player to obtain achievements based on milestones in the game.	Achievement Checker

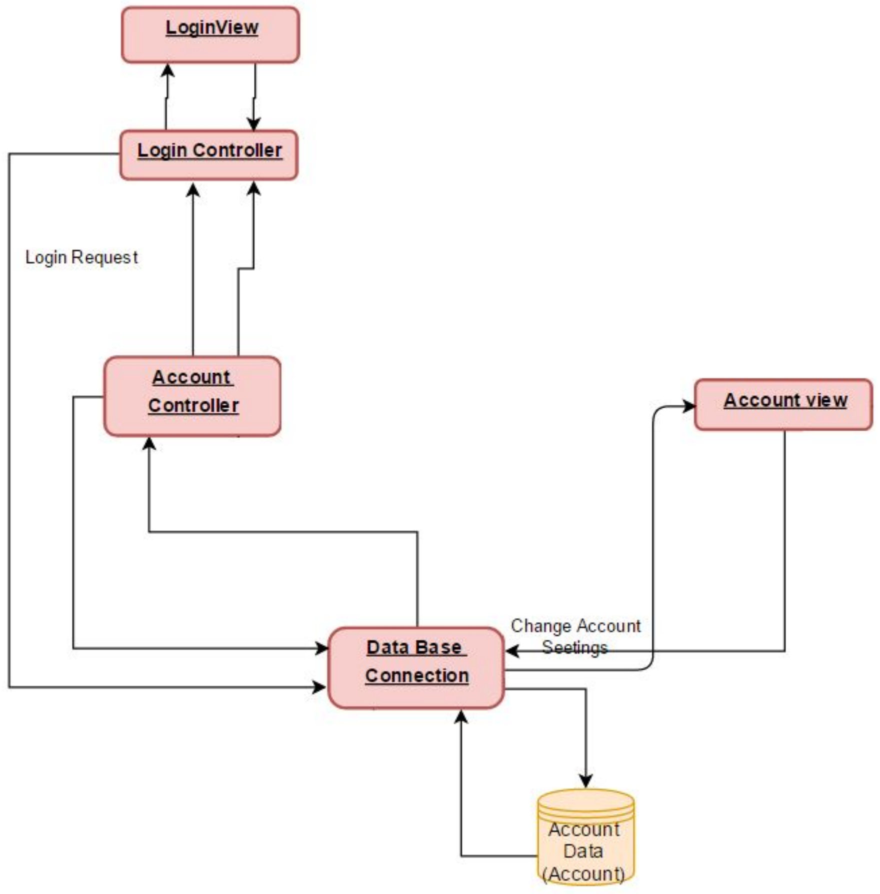


Figure 6.1: Register UC-1/Login UC-2/Manage Account UC-3

Figure 6.1 shows the registration and login user cases of our domain model. The user either registers a new account or logs in using the login view where the user ID and password are entered. The login controller confirms the data or enters new login data to the account database. If the login is successful, the app downloads the account data from the database.

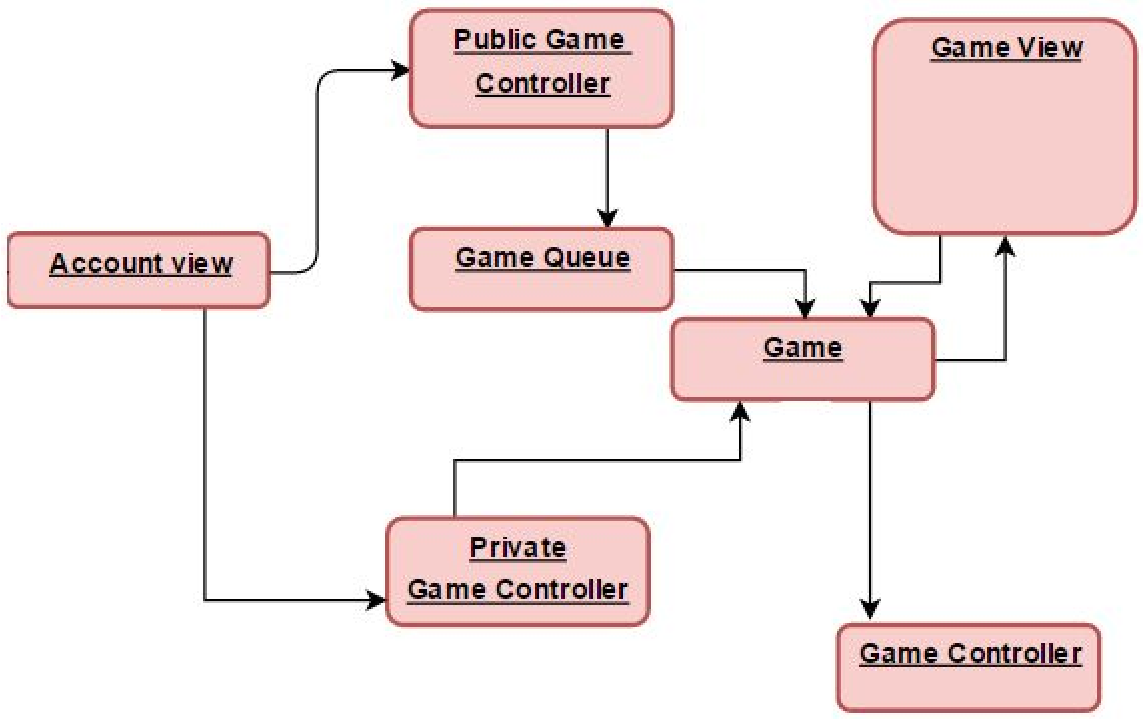


Figure 6.2: Find Public Game UC-4 / Create Private Game UC-5 / Join Private Game UC-6

In figure 6.2, the user case find public game is shown as the user enters a public game from their main homepage view, is entered into a queue and then begins the game and enters game view. The join private game is shown as well. Once a user enters the private game code they are entered into the game. The create private game user case is also shown. The private game controller generates a private game code that other users enter to join the private game.

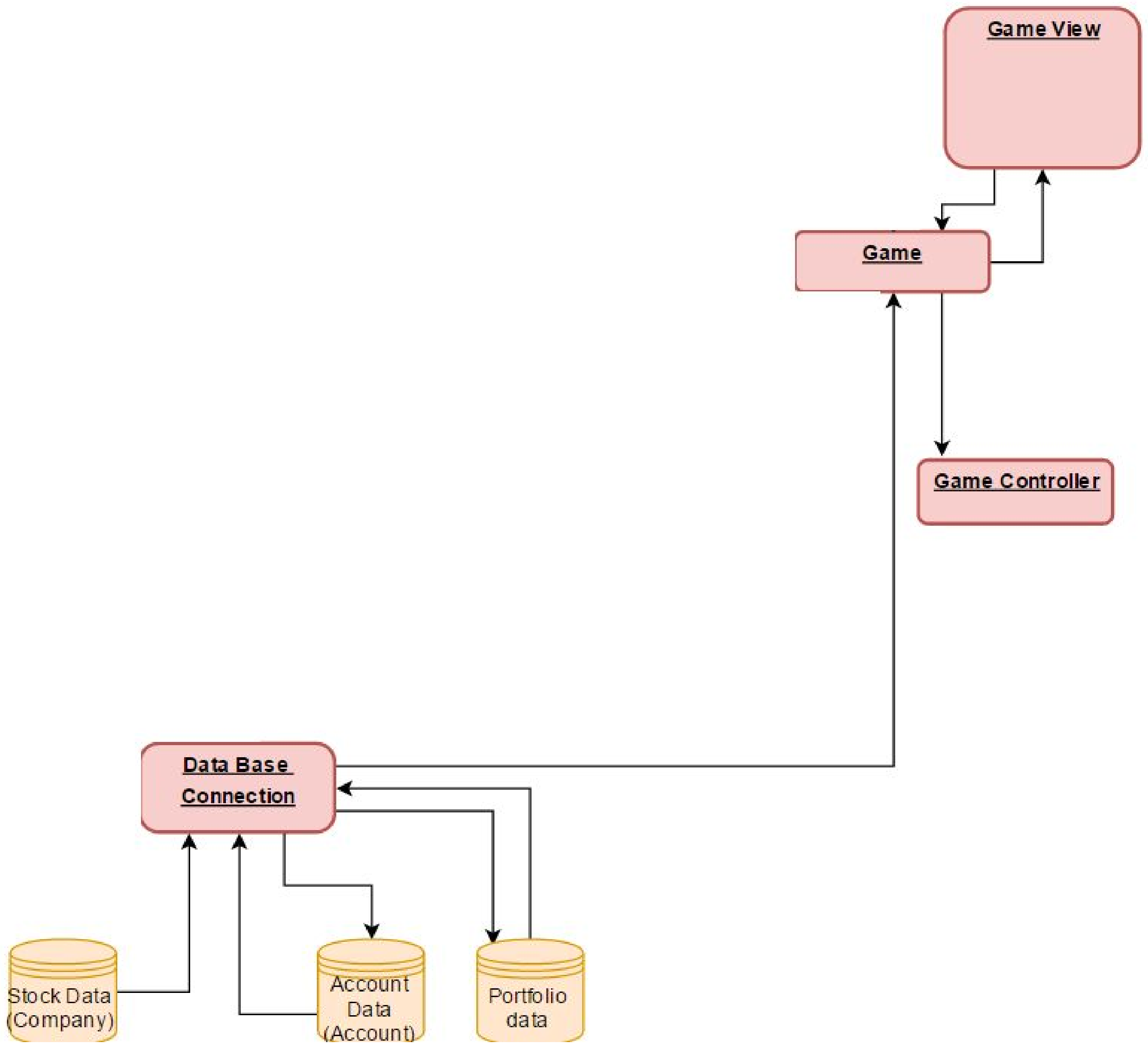


Figure 6.3: View Market data UC-7 / View Portfolio UC-9 / View Standings UC-11

The user case view market data is shown in this diagram. The game connects to the database which holds the stock data (company) information. The portfolio data is gathered in the same way, as the game connects to the database. The current standings are also also taken once connected to the database. Every turn the balance of the players in a game updates in the database, and this is the information the game acquires to display in the game view.

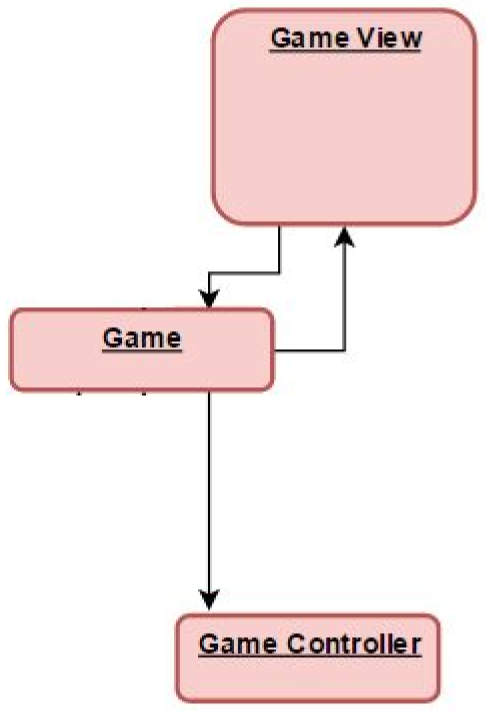


Figure 6.4: Place Market Order UC-8 / Influence Company UC-10

Figure 6.4 shows the user cases place market order and influence company. Place market order (such as buy and sell) is done entirely in the game. Influence company is as well.

6.1.1 Concept Definitions

Login Controller

For anyone to access the application, they must have a registered account. When a user is prompted to input their email address, the login controller will check the database to see if there is an account that matches the email address. If so, the user will be asked if they forgot their password and direct them to a link to reset their password through email and another link to proceed to the login page. On the contrary, the user will proceed to create an account and the information will be stored in the database.

Login View

The Login View is the first page that will be seen by users of the application. The user will enter their information and a request will be sent to the Login controller for verification. If the data is verified then the home page will load. If not, the Login view will display an error message. Users who have not manually logged out from account settings will be directed to the home page when they restart or open the application.

Account
The Account is a container that has information regarding to the particular user's account.

Account View
The Account View will display information to the user while they are logged-in to their account, but not currently in a game. They will be able to find games, join or create games, see their achievements, view their settings, etc.

Account Controller
A logged-in user can manage their account settings or see any achievements they may have gotten. The user can also post messages on their social media.

Game Queue
The Game Queue is a container that has information regarding all players currently searching for a game.

Public Game Controller
Once a player has successfully logged in, one of the game options that will be displayed on the home page is Find Game. When the user chooses the Find Game option then they will be put into a queue. Once there are four players in the queue, the system creates a game.

Private Game Controller
Once a player has successfully logged in, one game option that will be displayed on the home page is Create Game. When the user chooses the Create Game options then another page loads prompting the game specifications such as number of players, captial and turn duration. The user will input the requested information and the game will be created. A four digit passcode entry is prompted when Join Game is chosen. Upon successful entry, the player will be added into the game.

Game
The Game is a container that has information regarding a particular game being played.

Game View
The game view will be seen by any player in a game. It is where information regarding the game will be displayed and will allow for a multitude of functions that will be relevant to the game, such as placing market orders.

Game Controller
Handles all logic behind the game functions and will handle any queries made by the user on the Game View.

Database Connection

Necessary to access data on user accounts as well as stock simulation data used in the game. When data needs to be verified a request will be sent to the database to look for it. The database will also store information such as a company's market history and other primary information.

Chat Controller
Handles the chat system between players in a game.

Company Stock Controller
Information regarding a particular company in the game will be stored in this container, such as the company description, history, current market value, market trends, etc. Other information such as their stock value, or how much stock can be bought will also be here.

Portfolio
The Portfolios is a container that has information regarding a particular player in the game, such as assets.

Social Media Account
The Social Media Account is a container that has Information regarding a user's social media account.

Achievement
The Achievement is a container that has Information regarding a particular achievement.

Achievement Checker
Periodically checks if a user has hit a certain milestone and is to be awarded an achievement.

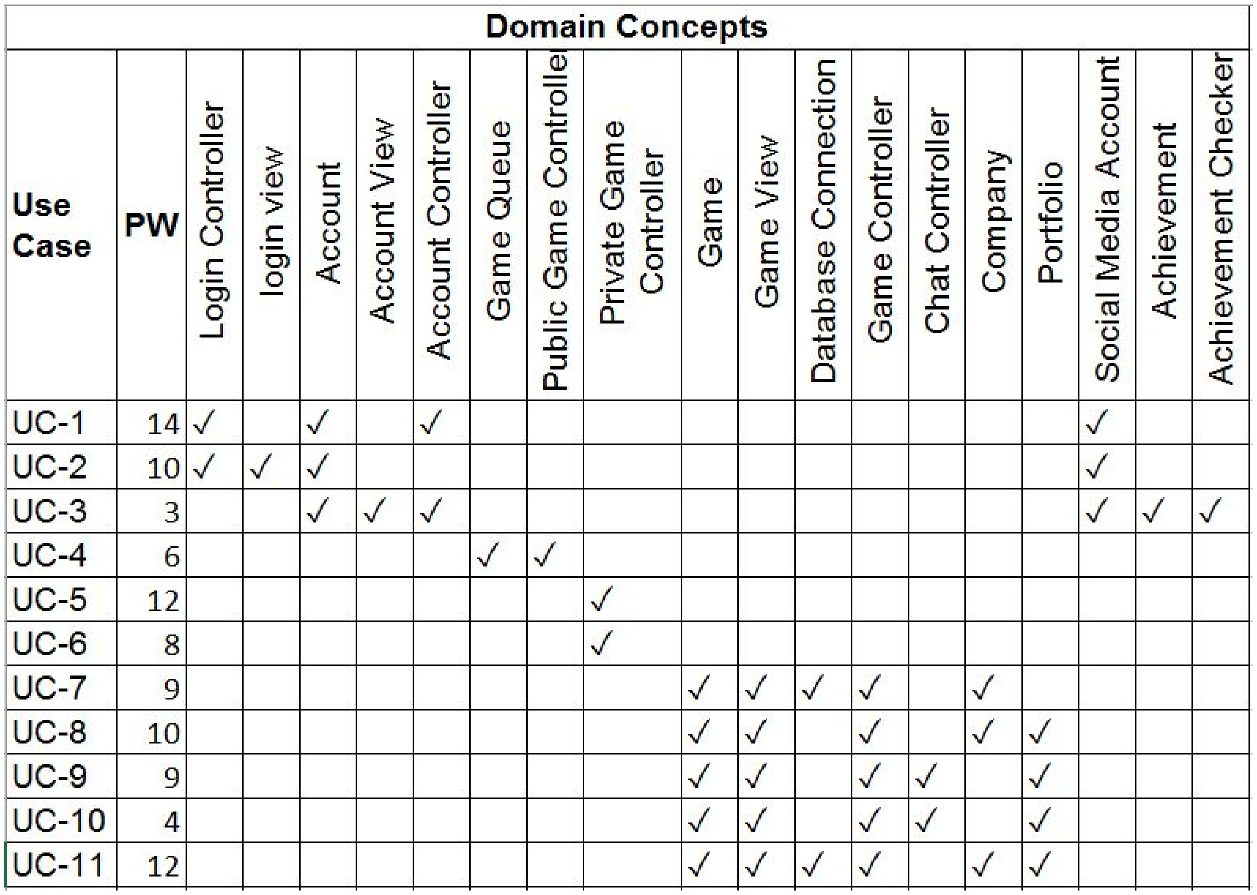
6.1.2 Association Definitions

Concept Pair	Association Description	Association name
Login View ↔ Login Controller	Login View passes on login information that needs to be verified by the Login Controller.	Send login data
Account View ↔ Account	The accounts view page requests information about a user which it retrieves from the Account containers.	Send User account information
Public Game Controller ↔ Game Queue	The game queue handles multiple players trying to enter a public game. This information is passed onto the public game controller.	Find Match
Game View ↔ Game	Information regarding a game is passed onto the game view.	Send game information
Game View ↔ Chat Controller	The chat controller sends chat data to the game view.	Send Chat Information
Game View ↔ Game Controller	Information about a game is sent to the game view.	View Game
Game ↔ Database controller	The database controller passes on primary information about a company to the game.	Send Company Info
Game View ↔ Portfolio	Information about players and the stocks they own is passed onto the game view.	Send Player Info
Achievement ↔ Achievement Checker	Information that an achievement has been accomplished is passed onto the achievement container.	Send achievement information
Account ↔ Social Media Account	Social media account passes on social media information about a user's to the accounts container.	Send account information
Game ↔ Game Controller	Information about stocks that are bought or sold is passed onto the game.	Send stock information. Update stock information. Send other players game information.

6.1.3 Attribution Definitions

Concept	Attribution	Attribution Definitions
login controller	login	If username and password input match, logs in, else return wrong password.
account controller	changePass	After confirming current password and making sure new password and confirm new password are the same, change password to new password.
public game controller	pblcontrolGame	Handles entering users into a public game
private game controller	createPvtGame, createPvtCode joinPvtGame	Controls the local game information and creates a 4 digit code that will be shared to other players that will be joining the private game. If entered code is correct, allows a user to join the private game,
chat controller	sendChatInformation	Users, chat statements. Sends chat information to the game
achievement checker	checkAchievement	Checks if an achievement is accomplished.
game controller	sellStock buyStock borrowStock updatepricedata viewRankings viewPlayerInfo viewCompanyInfo influenceCompany	Decreases the stock in user assets and user gains in game currency equal to the value of the stock. Decreases user currency and gain stock amount equal to the price. Gain currency equal the the value of the stock borrowed, but lose currency equal the the value of the stock borrowed after a period of time. Request updated stock price data from database. Show the players ranked by how much currency they would have if they liquidated all their assets. Show how much money and what stocks a player has. View the current information, profile, price, ownership, background, history and more of a company. User Influences a company causing repercussions in future prices.

6.1.4 Traceability Matrix



		Domain Concepts																	
Use Case	**PW**	Login Controller	login view	Account	Account View	Account Controller	Game Queue	Public Game Controlle	Private Game Controller	Game	Game View	Database Connection	Game Controller	Chat Controller	Company	Portfolio	Social Media Account	Achievement	Achievement Checker
UC-1	14	✓		✓		✓											✓		
UC-2	10	✓	✓	✓													✓		
UC-3	3			✓	✓	✓											✓	✓	✓
UC-4	6						✓	✓											
UC-5	12								✓										
UC-6	8								✓										
UC-7	9									✓	✓	✓	✓		✓				
UC-8	10									✓	✓		✓		✓	✓			
UC-9	9									✓	✓		✓	✓		✓			
UC-10	4									✓	✓		✓	✓		✓			
UC-11	12									✓	✓	✓	✓		✓	✓			

6.2 System Operational contacts

UC-1 Register/Create an Account

• **Pre-conditions**
– If a new user is opening the application for the first time, in order to jump right into the game, the user must create a login and register within the system
• **Post-conditions**
– After registration, the database is updated with the basic information of the new player.

UC-5 FindPublicGame

• **Pre-conditions**
– Investor must be logged in to their account
– User must not been in a current game
• **Post-conditions**
– player has joined a game
– Queue holding games is updated

UC-2 Create/Join League

• **Pre-conditions**
– User must be logged in
– User must fill in all required text boxes
• **Post-conditions**
– A four digit pin is generated and given to creator
– Other user must input pin to join game

UC-8 Place a Market Order

• **Pre-conditions**
– User is logged into their account.
– User must be in a public or private game
– User must have enough funds in their account to place a market order
• **Post-conditions**
– User profile is reflected with any change to funds and stocks.
– In Game data has been updated with these changes.

UC-9 ViewPortfolio

• **Pre-conditions**
– User is logged into their account.
– User is in a public or private game
• **Post-conditions**
– Any adjustments made to the player's portfolio has been updated in the database.

6.3 Mathematical Model

Rather than use a statistical model for stock price prediction, our application will utilize data of stocks from previous years.

Our application involves players can either buy stocks, sell stocks, request a short, buy a cover, limit a specific stock, and create a stop price of a specific stock from various companies. However, the companies that exist within the app are fictional. Each of these corporations will possess its own characteristics, traits, as well as a backstory. We will develop these qualities for each of the companies, providing users with the information and background that they can apply when making the decision of investment. The aim of this background knowledge about the companies is to provide the user with both incentive and deterrents about the individual and unique companies. This will allow the user to proceed and make well informed and educated decisions.

While the backstories and information about each company will be unique, each company's data will be mapped to a company that exists in the real world. The term mapped is used in order to illustrate how the company performs economically according to the stock market. Since the data will be from previous years, the year 2013 applies well to Rags to Riches because it was a strong year for the stock market in which there were no market crashes like the one in 2010. Since 2013 is considered a positive year for the stock market, it allows users to have a more pleasant experience, which simplifies the game for all users regardless of age or skill level. This year is also selected to ensure that users do not possess any recent economic knowledge about major corporations, and thus make the experience more balanced and fair for all users. Selecting the year 2015 or 2016 is less appropriate, because while most users might not house any recollection of the stock market performance of that year, a small percentage of users might hold that knowledge, which could result in player disadvantage and less overall game balance.

In essence, the “model” that our app will utilize will be a set of data from several real world company’s stock performance in the year 2013. Each of the fictional companies that exist within the app will be mapped to one of the real world companies, and thus possess that company’s stock performance of 2013. The user will be provided with the background information about the individual companies, but not what real world company’s stock data it will be mapped to nor the fact that the data is from 2013. The remaining aspects of the fictional companies will be entirely unique and specialized to heighten the experience of the app and the user’s enjoyment. This method of utilizing old stock data rather than develop a statistical model for stock price prediction simplifies this portion and allows for further effort and creative software development to be poured into more novel and interesting parts of the Rags to Riches app.

7 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.

We chose to make sequence diagrams for the most important use case scenarios.

Difference between game view and game:
The game view is responsible for updating the player's game screen constantly with new information. There are many fragile aspects to the game view that have dynamic information that we thought it was best to separate the game and the game view. The game controller holds the information and feeds it to the game view which then displays the information to the user.

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.

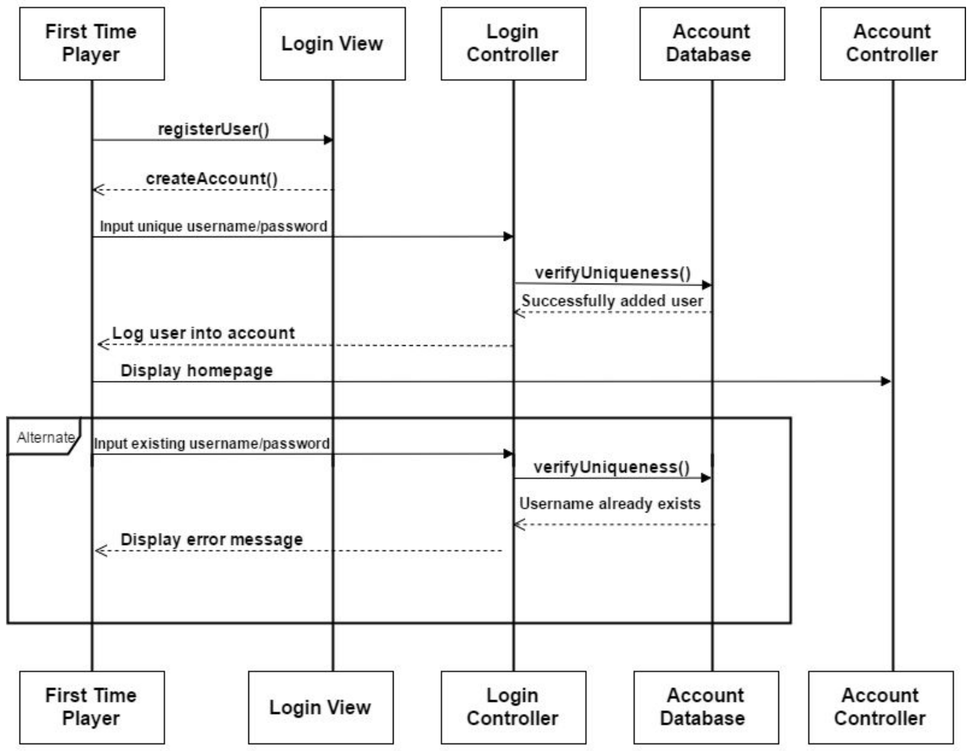


Figure 7.1: UC-1 Register

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.

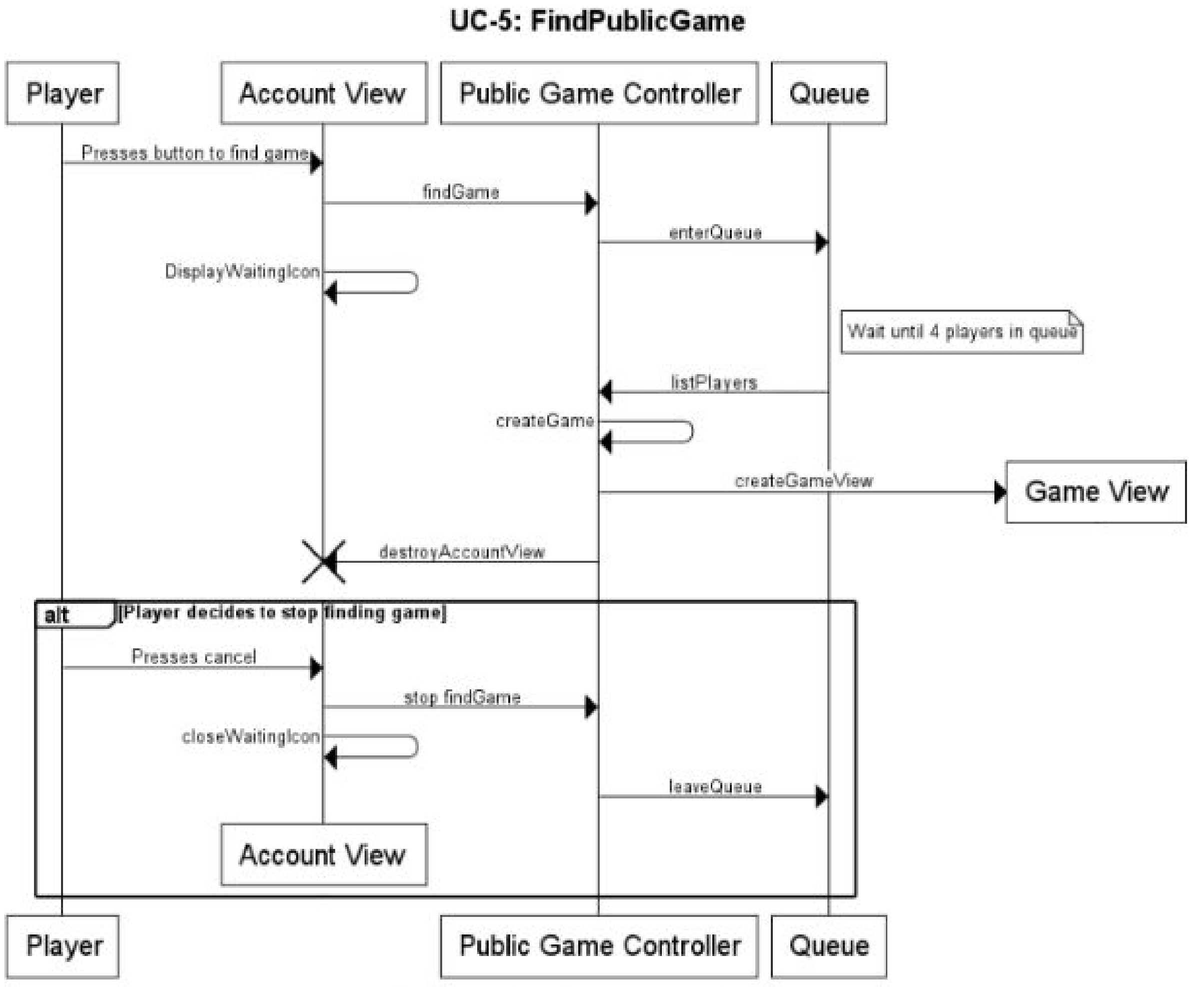


Figure 1.2: UC-5 FindPublicGame

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.

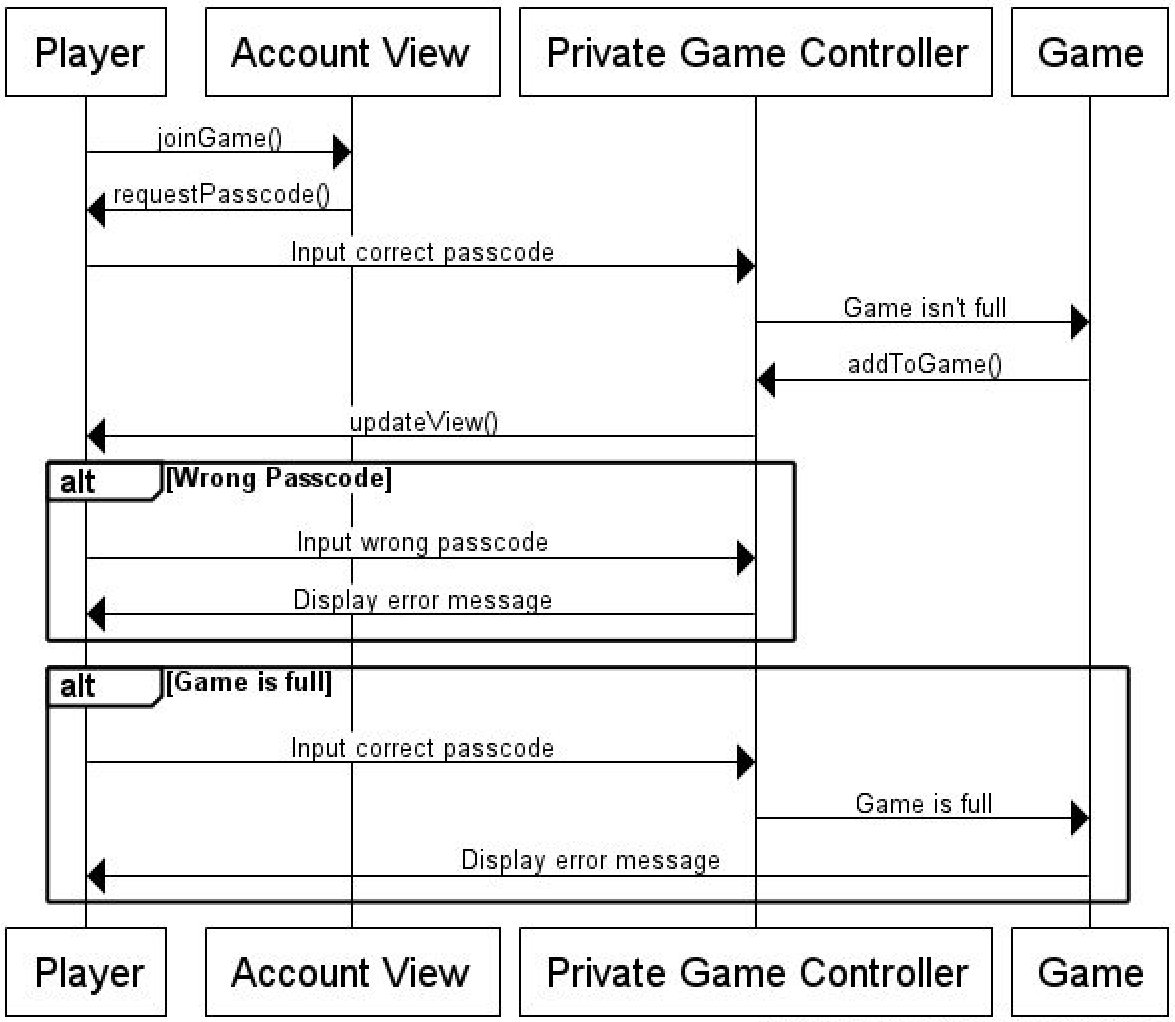


Figure 7.3: UC-7 JoinPrivateGame

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.

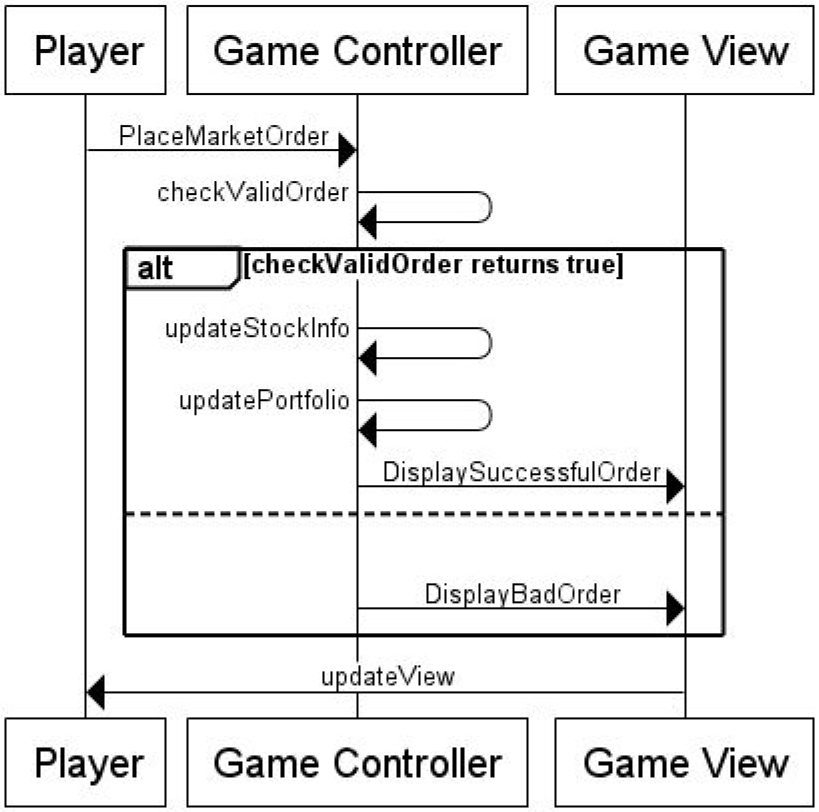


Figure 7.4: UC-8 PlaceMarketOrder

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.

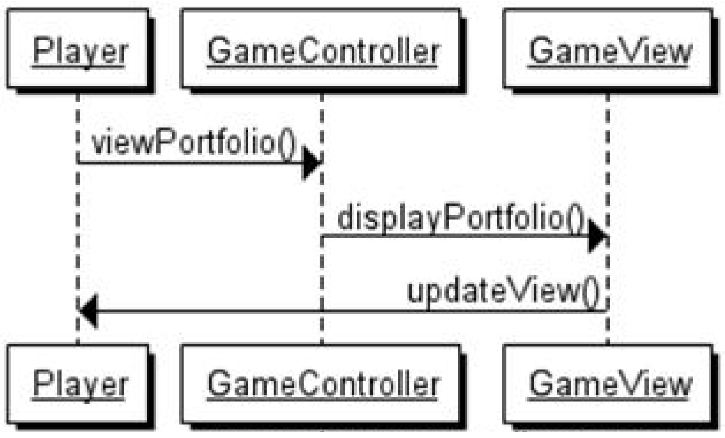
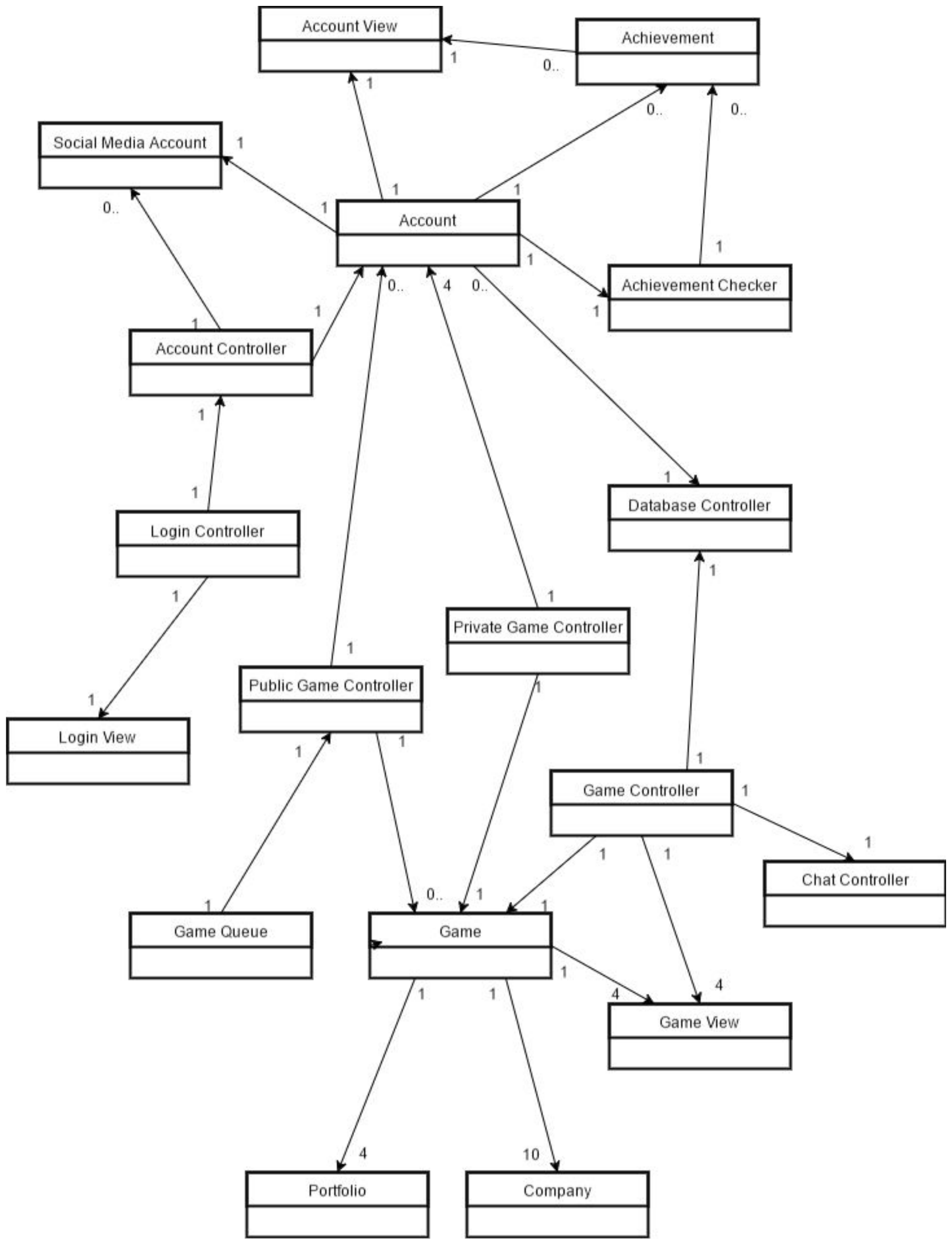


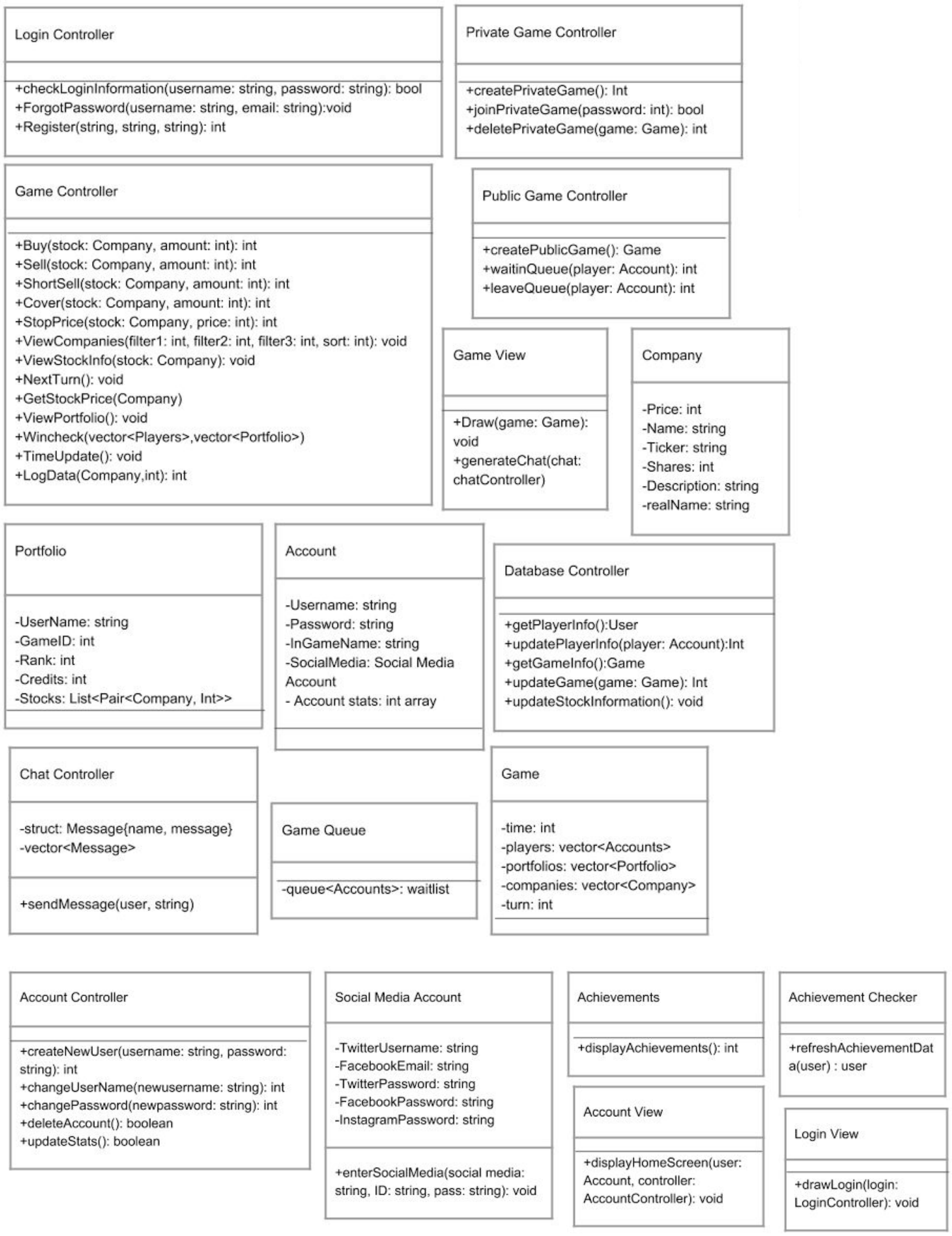
Figure 7.5: UC-9 ViewPortfolio

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

8 Class Diagram and Interface Specification

8.1 Class Diagram





```
Login Controller
----------------
----------------
+checkLoginInformation(username: string, password: string): bool
+ForgotPassword(username: string, email: string):void
+Register(string, string, string): int
```

```
Private Game Controller
----------------
----------------
+createPrivateGame(): Int
+joinPrivateGame(password: int): bool
+deletePrivateGame(game: Game): int
```

```
Game Controller
----------------
----------------
+Buy(stock: Company, amount: int): int
+Sell(stock: Company, amount: int): int
+ShortSell(stock: Company, amount: int): int
+Cover(stock: Company, amount: int): int
+StopPrice(stock: Company, price: int): int
+ViewCompanies(filter1: int, filter2: int, filter3: int, sort: int): void
+ViewStockInfo(stock: Company): void
+NextTurn(): void
+GetStockPrice(Company)
+ViewPortfolio(): void
+Wincheck(vector<Players>,vector<Portfolio>)
+TimeUpdate(): void
+LogData(Company,int): int
```

```
Public Game Controller
----------------
----------------
+createPublicGame(): Game
+waitinQueue(player: Account): int
+leaveQueue(player: Account): int
```

```
Game View
----------------
----------------
+Draw(game: Game):
void
+generateChat(chat:
chatController)
```

```
Company
----------------
-Price: int
-Name: string
-Ticker: string
-Shares: int
-Description: string
-realName: string
```

```
Portfolio
----------------
-UserName: string
-GameID: int
-Rank: int
-Credits: int
-Stocks: List<Pair<Company, Int>>
----------------
```

```
Account
----------------
-Username: string
-Password: string
-InGameName: string
-SocialMedia: Social Media
Account
- Account stats: int array
----------------
```

```
Database Controller
----------------
----------------
+getPlayerInfo():User
+updatePlayerInfo(player: Account):Int
+getGameInfo():Game
+updateGame(game: Game): Int
+updateStockInformation(): void
```

```
Chat Controller
----------------
-struct: Message{name, message}
-vector<Message>
----------------
+sendMessage(user, string)
```

```
Game Queue
----------------
----------------
-queue<Accounts>: waitlist
```

```
Game
----------------
-time: int
-players: vector<Accounts>
-portfolios: vector<Portfolio>
-companies: vector<Company>
-turn: int
----------------
```

```
Account Controller
----------------
----------------
+createNewUser(username: string, password:
string): int
+changeUserName(newusername: string): int
+changePassword(newpassword: string): int
+deleteAccount(): boolean
+updateStats(): boolean
```

```
Social Media Account
----------------
-TwitterUsername: string
-FacebookEmail: string
-TwitterPassword: string
-FacebookPassword: string
-InstagramPassword: string
----------------
+enterSocialMedia(social media:
string, ID: string, pass: string): void
```

```
Achievements
----------------
----------------
+displayAchievements(): int
```

```
Achievement Checker
----------------
----------------
+refreshAchievementDat
a(user) : user
```

```
Account View
----------------
----------------
+displayHomeScreen(user:
Account, controller:
AccountController): void
```

```
Login View
----------------
----------------
+drawLogin(login:
LoginController): void
```

8.2 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 mananger 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 + Σ Stocks.Company.Price*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.

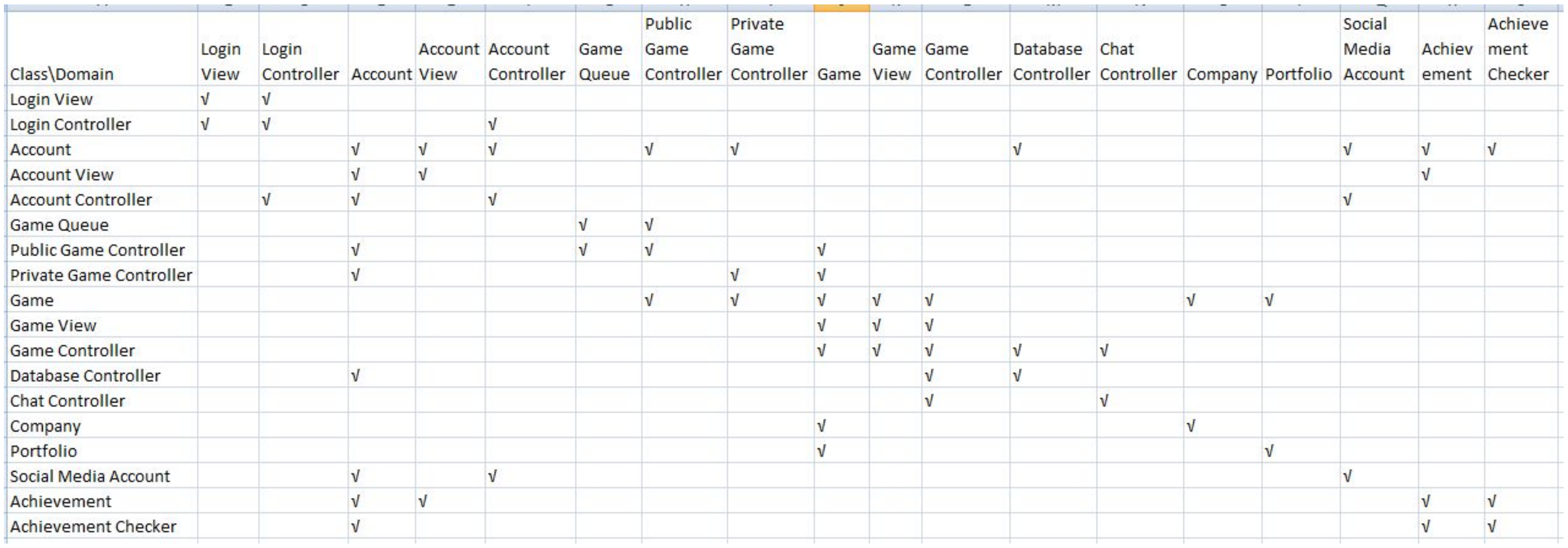
Achievement Checker

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.

8.3 Traceability Matrix



Class\Domain	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	Achiev ement	Achieve ment 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.

8.4 Design Patterns

Command Pattern

The command pattern is implemented throughout the entire application and manifests itself as the “Account” . Our controller class commands are carried out by classes such as game controller, game, achievements, , and database controller. Everything in our system corresponds to an unique player account, which calls to the database to update. The controller knows the receiver of each action request, and knows the appropriate sequence of calls to make based on a given request. As seen on our class diagram the account class calls upon other classes given a request, and thus any future updates will be easier to implement using this pattern.

Publisher Subscriber Pattern

A publisher subscriber pattern has also be used to implement our game due to the pattern’s flexibility with events. A publisher subscriber pattern is used in implementing our event system in our game. In our game, methods such as buy and sell would be the publishers while subscribers are the achievement checker and the database controllers. As soon as either is "published" the database controller sends information to the database to update/retrieve. The achievement checker also reacts to either event by checking whether a player has successfully completed a requirement to unlock an achievement. The publisher subscriber pattern is crucial in order to implement an event system that would allow stocks to react to "news" within the game that would affect the trends of stocks. By using the publisher subscriber pattern, the game controller class would be able to react to any event much easier.

8.5 Object Constraint Language

context AccountController::CreateAccount(UserInfo : userinfo) : Boolean post: (hasAccount = true) - The new player has an account upon registration

context GameController::RequestPortfolio(string : player) void pre: (player → Account.portfolio = this.portfolio) - View your own portfolios or others

context DatabaseController::ExecuteOrder(Ticket : ticket) : Boolean pre: (ValidateSell()) post: (Account.Update()) - The player's portfolio is updated to accommodate bought/sold stocks

Context PrivateGameController::CreatePrivateGame(Fields : fields) : Boolean post: (game→name = private) - Private game will be created and a code will appear that will grant users access to game

context GameController::JoinGame(String : username, String : game) : Boolean post: (game→this.game) - The User will enter game queue until number of required players is satisfied

9 System Architecture and System Design

9.1 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.

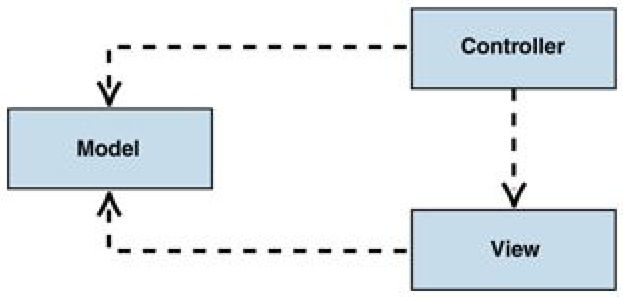


Figure 3.1: MVC Diagram

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]

9.2 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.

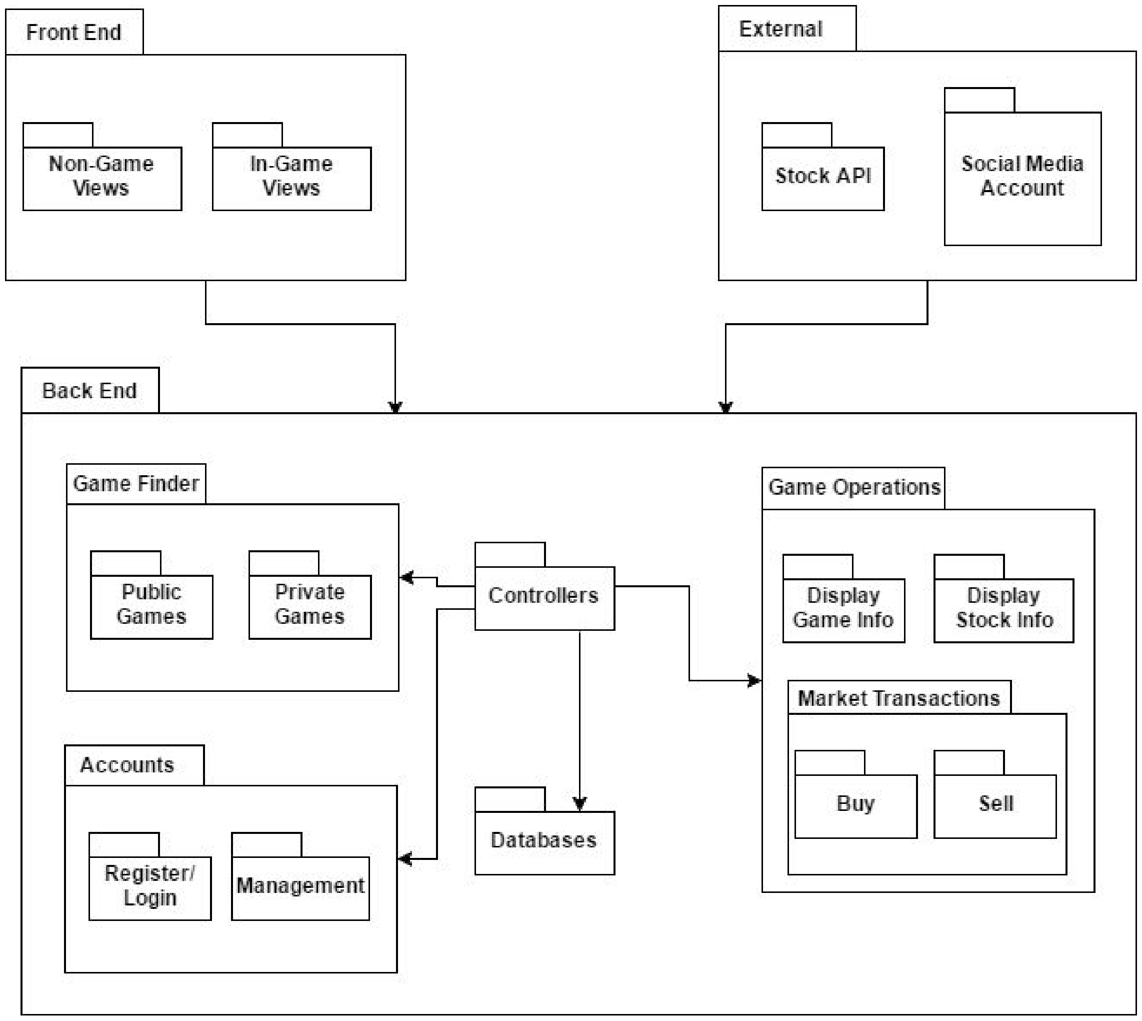


Figure 9.2: UML Package Diagram

9.3 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.

9.4 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.

9.5 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.

9.6 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.

9.7 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.

10 Algorithms and Data Structures

10.1 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. Stock information is sorted initially using insertion sort.

10.2 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.

11 User Interface Design and Implementation

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 have been implemented to be resembled as close as possible.

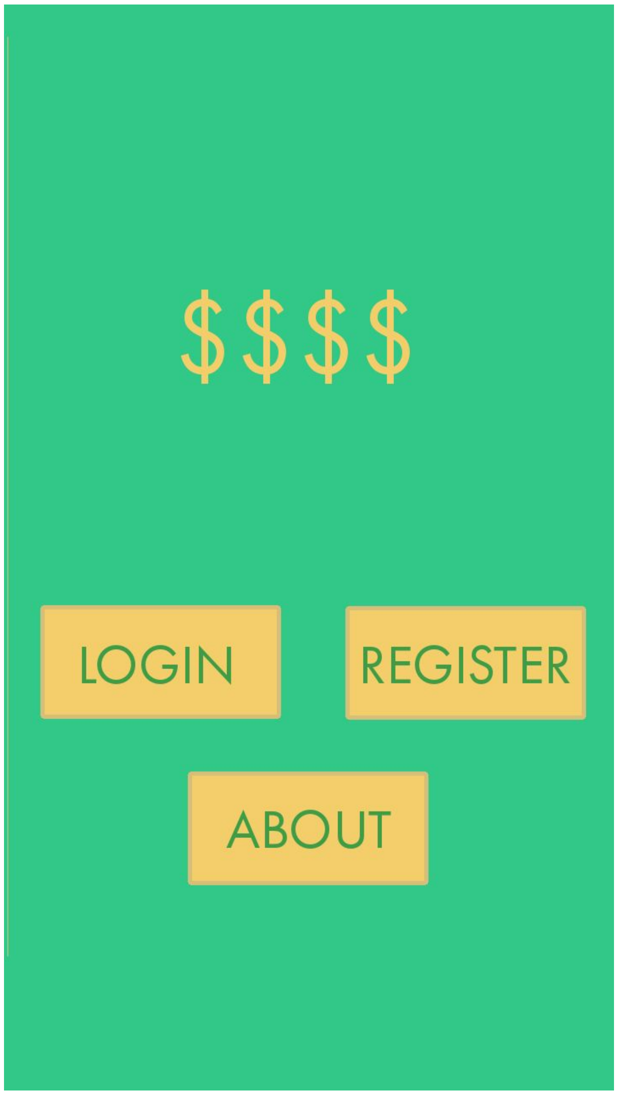


Figure 11.1: Initial Landing Page

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.

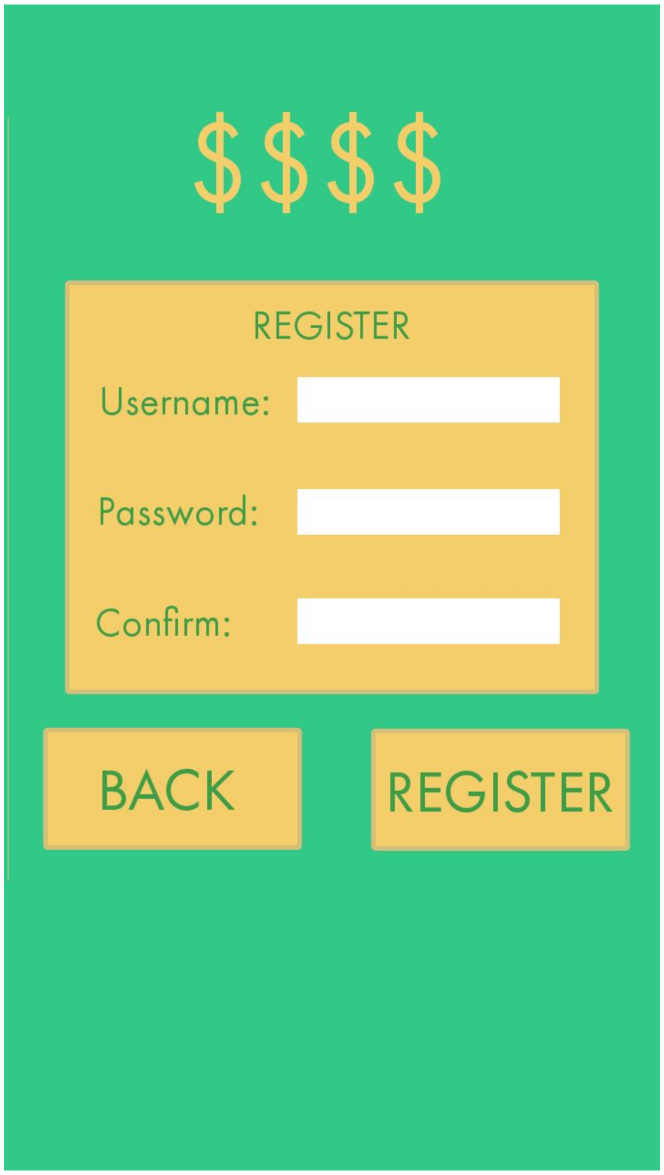


Figure 11.2: Register Page

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.

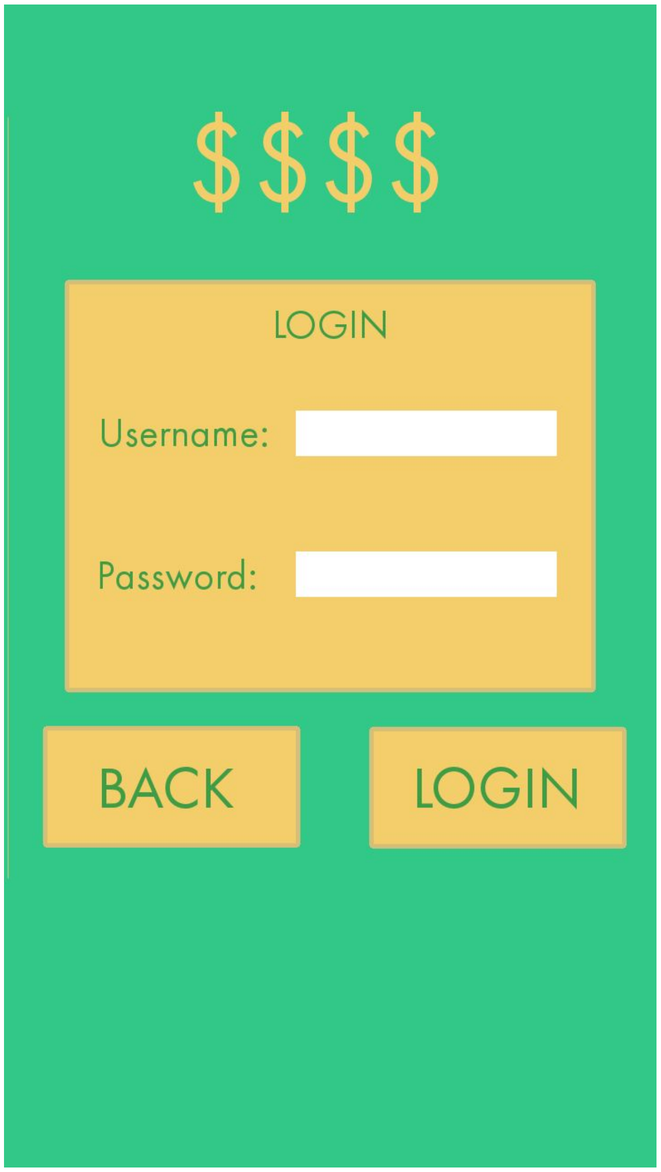


Figure 11.3: Login Page

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.

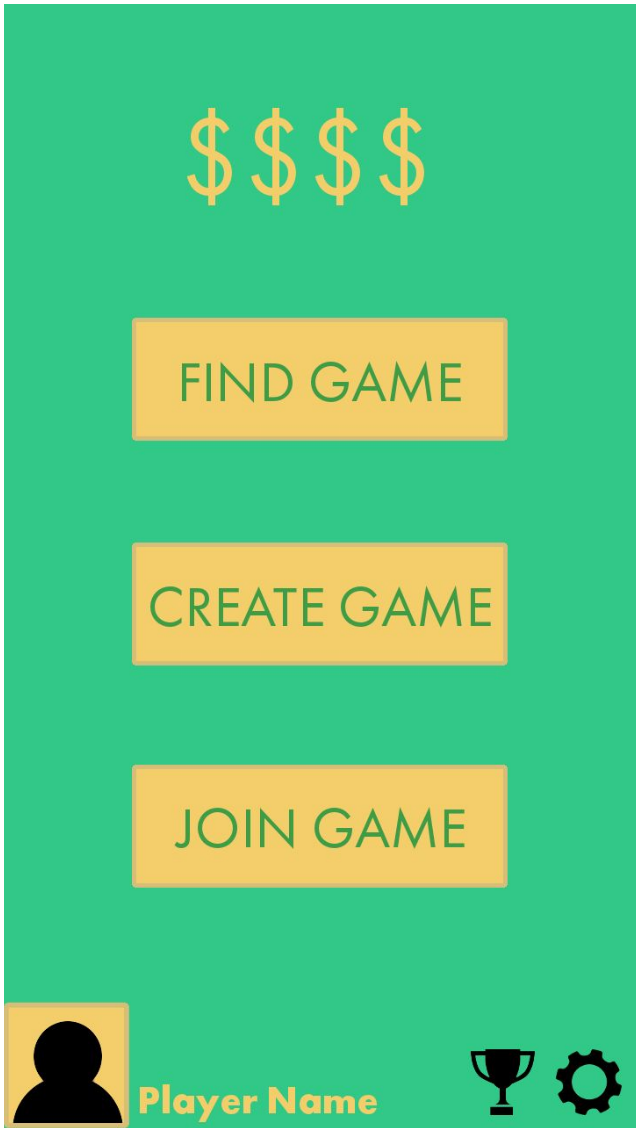


Figure 11.4: Application Home Page

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.

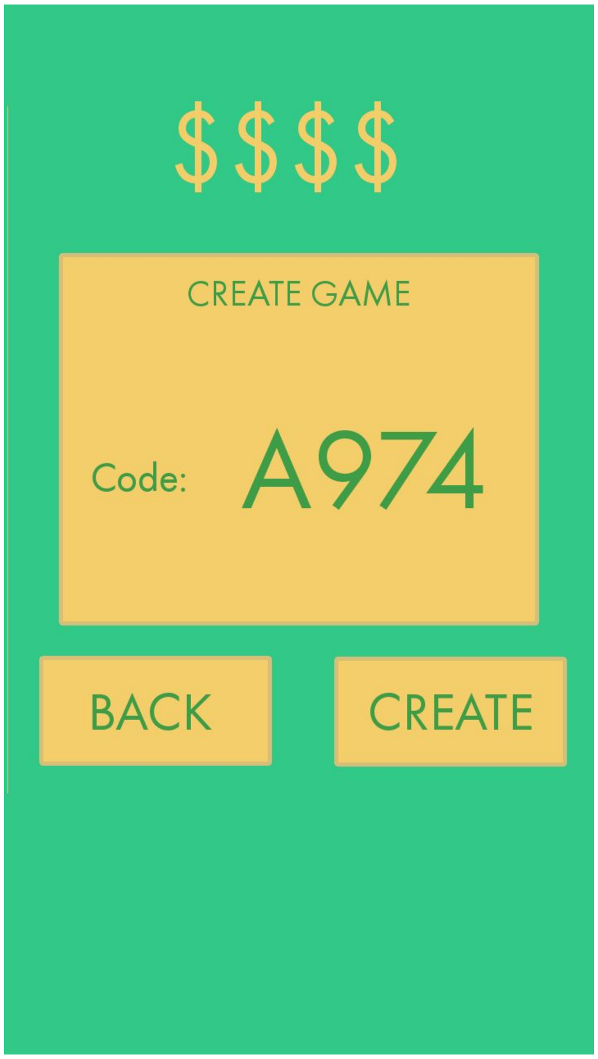


Figure 11.5: Create Private Game Page

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.

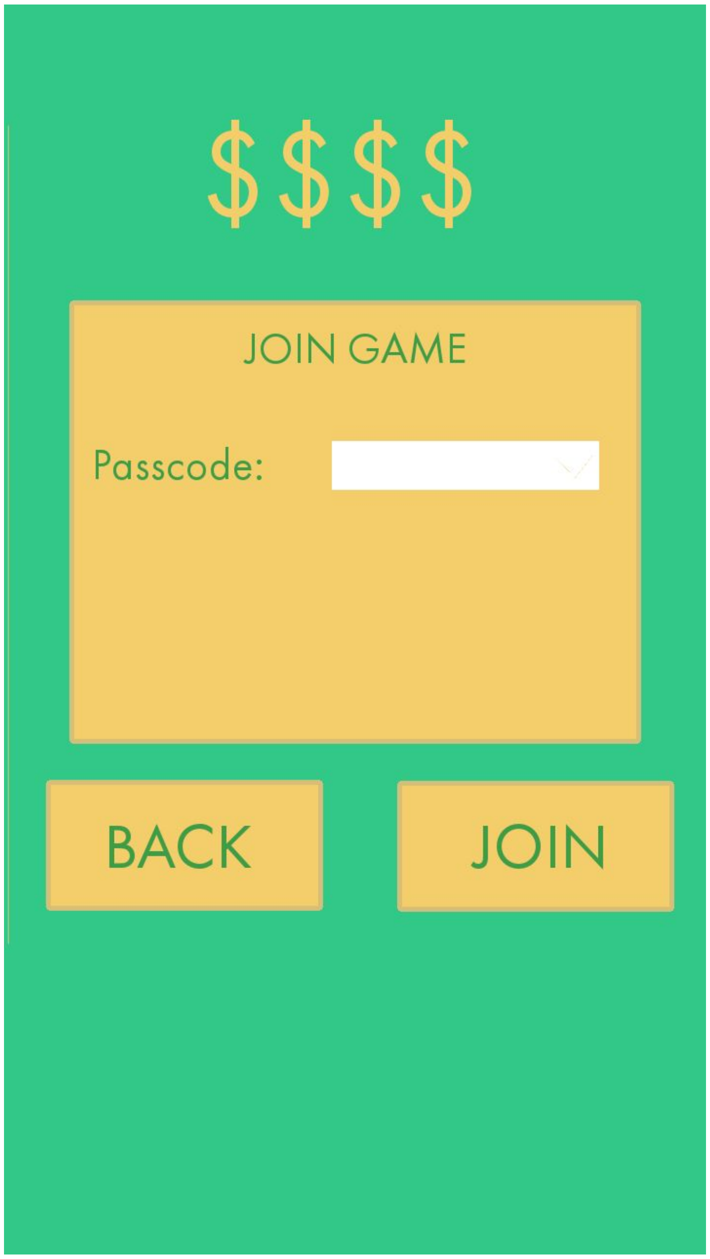


Figure 11.6: Initial Landing Page

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.

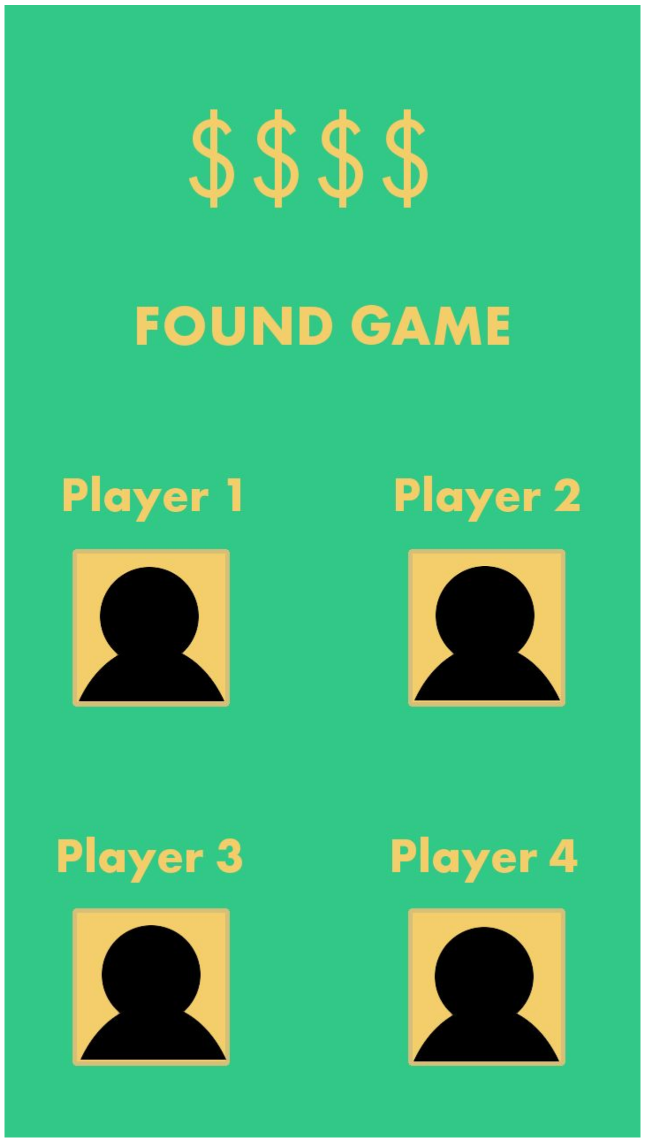


Figure 11.7: Found Game Page

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.

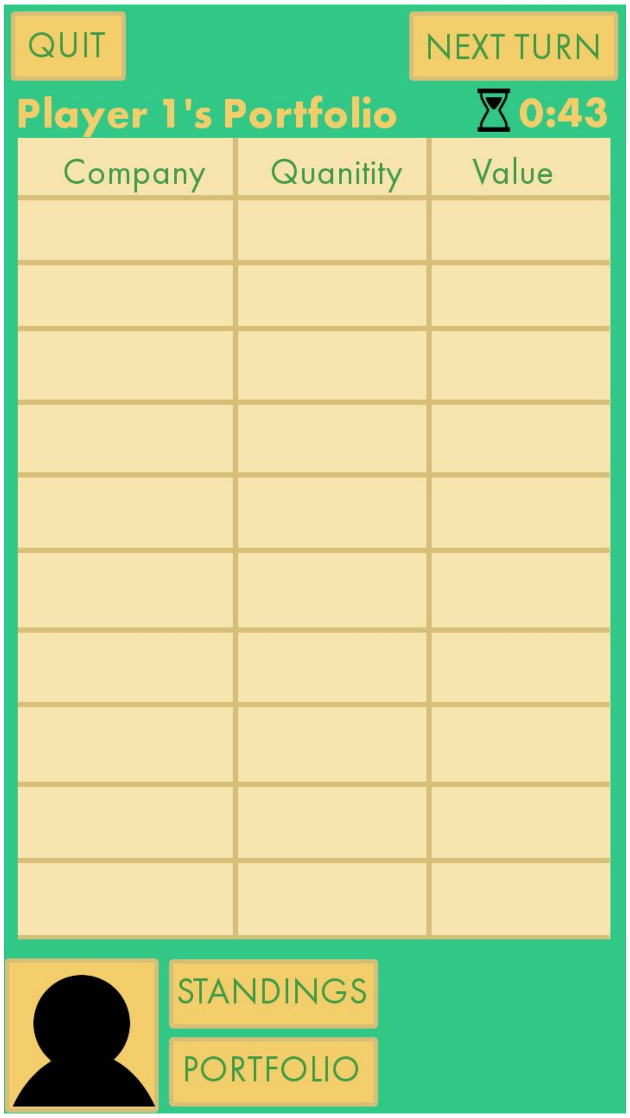


Figure 11.8: View Portfolio Page

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, as well as view stocks available for purchase. 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. A game is finished when all turns have been completed or if all but one remaining player becomes bankrupt.

12 Design of Tests

12.1 Use Cases and Unit Testing

Login and Registration

Test-case Identifier: TC-1 (Login) Function Tested: **checkLoginInformation(username: string, password: string): bool** 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: **register(username: string, password: string): int** 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: **forgotPassword(username: string, email: string):void** 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: **deleteAccount(use:user): boolean** 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: **changeUserName(newusername: string): int, changePassword(newpassword: string): int** 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: **waitinQueue(player: Account): int** 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: **leaveQueue(player: Account): int** 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: **createPublicGame(): game** 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: **createPrivateGame(): int** 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.

Test-case Identifier: TC- 10(Join Private Game) Function Tested: **joinPrivateGame(password: int): bool** 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.	
Test Procedure: Call Function with for room with correct password. Call Function for room with incorrect password. Call function to room with correct password 3 times.	
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

Test-case Identifier: TC-11 (Buy Stock) Function Tested: **Buy(stock: Company, amount: int): int** 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.	
Test Procedure: 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.	
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: **Sell(stock: Company, amount: int): int** 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: **viewPortfolio(): void** 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: **viewCompanies(filter1: int, filter2: int, filter3: int, sort: int): void**

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: **getPlayerInfo():User** 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: **updatePlayerInfo(player: Account):int** 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: **updateStockInformation(): void** 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: **sendMessage(chat: chatController): void** 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: **refreshAchievementData(useraccount: user): user** 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.

12.2 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.

12.3 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.

13 History of Work, Current Status, and Future Work

13.1 History of Work

Through the course of the semester, our team completed all of our planned milestones in an organized, timely, and regular manner. Our first major milestone, the project proposal, was accomplished on time prior to its due date of January 29. After agreeing upon the fantasy stock market project, our team discussed how innovation could be achieved, before ultimately deciding upon developing an Android application. The following few weeks of February, our team continued to discuss and share ideas as we began completing parts of report 1, and then successfully completed the full Report 1 by February 19th.

Furthermore, our team begun scratching the surface of the implementation process, while successfully meeting the deadlines for the partial Report 2 reports. After submitting the full Report 2 on March 12th, our team dove headfirst into implementation, organizing designating coding meetings over Spring Break and the week prior to the first demo. Following the first demo, our team continued on the implementation process, both refining what had already been set while also expanding upon additional features. As we collected and polished our previous reports for the submissions of Report 3 during late April and early May, our team worked vigorously to implement the remaining features to realize the subsequent use cases. After the completion of the second demo, our team began to wind down on the implementation process and proceeded to put the finishing touches on our final report 3 while writing our individual reflective essays that were both due May 1st, and the electronic project archive due May 3rd.

13.2 Current Status

Currently, Rags to Riches is an available and functional application for Android devices. It supports various features, including public and private games, in game achievements, tutorial, working orders, graphs, user portfolios, chat, and user systems with a functioning and concise user interface that can be easily used on Android based phones, tablets, or virtual device. The core functionality of the app has been realized and milestones have been achieved.

13.3 Key Accomplishments

The following key accomplishments have been met during the entire course of the development process of Rags to Riches:

- A clear and responsive user interface available to use on Android based devices
- An implementation for public and private game modes with turn based interactive gameplay
- An implementation of in game achievements that unlock upon the completion of particular user milestones
- An effortless registration and login process
- A concise and user friendly system for users, achievements, portfolios, and games
- An implementation that translates historical stock data that maps it to random fictional companies
- An implementation that displays stocks, data, and other statistics in graphical and numerical methods that enforce education

Additionally, Rags to Riches has implemented the following use cases:

- UC-1
- UC-2
- UC-3
- UC-4
- UC-5
- UC-6
- UC-7
- UC-8
- UC-9
- UC-11

13.4 Future Work

Going forward, our team has pondered a few possibilities of potential expansion and revision. That is, there is always the opportunity for the implementation and realization of additional achievements and fictional companies. These additions would simply be to serve and increase the overall enjoyment of the game, encouraging users to learn about the new companies to be potential investors or chase additional and more difficult achievements.

The other possible consideration to expand upon that would aim to increase user retention and possibly interest potential new users would be the social media integration. That is, our team could attempt to simplify the social media integration process to allow users to easily link their Rags to Riches account to social media websites such as Facebook and Twitter. This could encourage users to continue playing the game, and friends of Rags to Riches users could potentially be interested enough by the user's posts or notifications to possibly download the application and give themselves a try. Indeed, this greater cohesion between the application and users social media could increase replayability and pique fresh interest.

14 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.

[7] "Software Architecture". *En.wikipedia.org*. N.p., 2017. Web. 5 Mar. 2017.

[8] "User Story". *En.wikipedia.org*. N.p., 2017. Web. 3 Feb. 2017.

[9] "Chapter 3: Architectural Patterns and Styles." *Microsoft.* N.p. Web. 1 Mar. 2017.

[10] "Test-Driven Development." *En.wikipedia.org*. N.p., 2017. Web. 13 Mar. 2017.

[11] "Software Development Process." *En.wikipedia.org*. N.p., 2017. Web. 31 Jan. 2017.

[12] "Class Diagram". *En.wikipedia.org*. N.p., 2017. Web. 5 Feb. 2017.

[13] Fontinelle, Amy. "Shareholder". *Investopedia.* N.p, 2017. Web. 28 Jan. 2017.

[14] "Requirements Engineering". *En.wikipedia.org*. N.p., 2017. Web. 31 Jan. 2017.

[15] "Glossary of Stock Market Terms." *NASDAQ.com*. N.p, 2017. Web. 3 Feb. 2017.

[16] "User Interface". *En.wikipedia.org*. N.p., 2017. Web. 12 Feb. 2017.

[17] "Systems Development Life Cycle". *En.wikipedia.org*. N.p., 2017. Web. 8 Feb. 2017.

[18] "Integration Testing". *En.wikipedia.org*. N.p., 2017. Web. 10 March. 2017.

[19] "Domain Analysis". *En.wikipedia.org*. N.p., 2017. Web. 17 Feb. 2017.

[20] “System Sequence Diagram”. *En.wikipedia.org*. N.p., 2017. Web. 20 Feb. 2017

15 Project Management

15.1 Merging Contributions for Individual Team Members

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

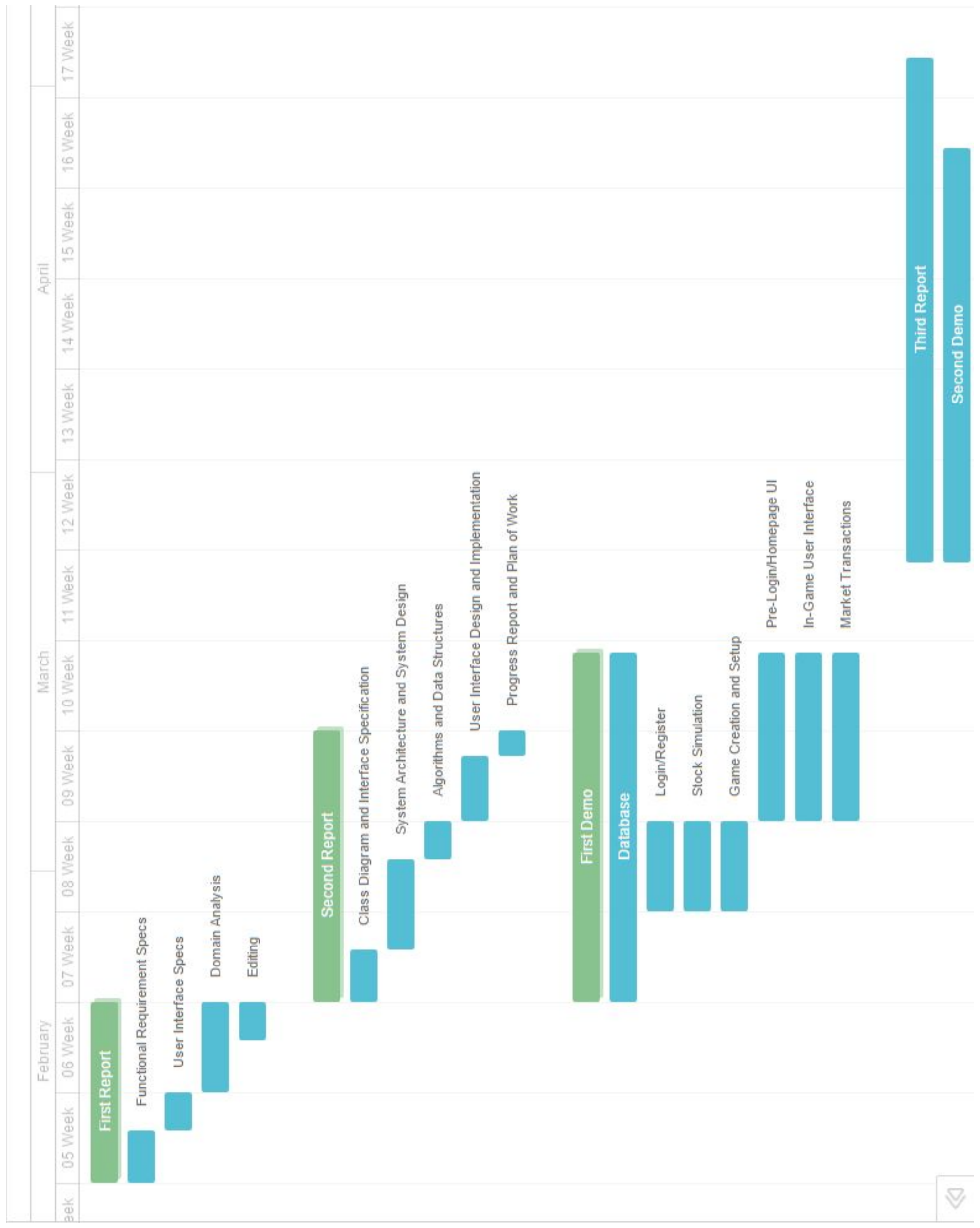
One particular issue that precipitated the decision of the app functionality, whether it would sway more towards a game or a league. As the use cases were developed the concept for the application as well as the major decision came clear and naturally, which would be the application would primarily be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

The team had organized a Github account which allowed for uniform collaboration and development of the project code. The entirety of the project documentation had been developed on Google Docs, which allowed 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 such as demos and discussions as well as the initial planning of the project had been planned and organized. Alejandro was responsible for reserving conference rooms at a time of the week when every group member was available to meet. These meetings consisted of brainstorming and discussion, as well as collaborative implementation and design. The meetings allowed for ensuring team members were accomplishing their designated tasks and to ensure the project was progressing on schedule.

15.2 Project Coordination and Progress Report

The overall progress of the project had sufficed in its progression and development. The Github page has been used to measure changes 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 was established, each subgroup was responsible for the development of their own corresponding portions of the project. Integration of the different components occurred during major milestone dates as well as team meetings.

15.3 Gantt Chart used through development progress



[1]

15.4 Breakdown of Responsibilities

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

We tested our project using the bottom up strategy. Each team member was responsible for unit testing their own work and the pages they were responsible for. The final integration of all the parts of the project was then tested again. All group members helped develop the integration tests and Kartik was responsible for executing the test.

15.4.1 Group 1: Alejandro and Arjun

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

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

15.4.2 Group 2: Elisa-Michelle and Bryan

The second group in our team focused 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.

Goals: The plan was to have basic game creation implemented along with an invitation system for other players.

15.4.3 Group 3: Kartik, Deep, and William

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

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