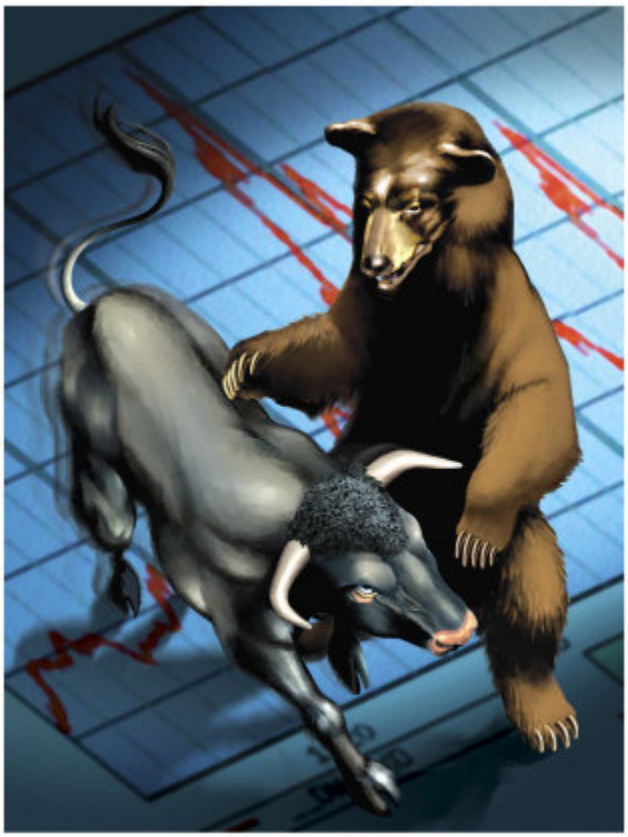
Bears & Bulls

332:452 Software Engineering



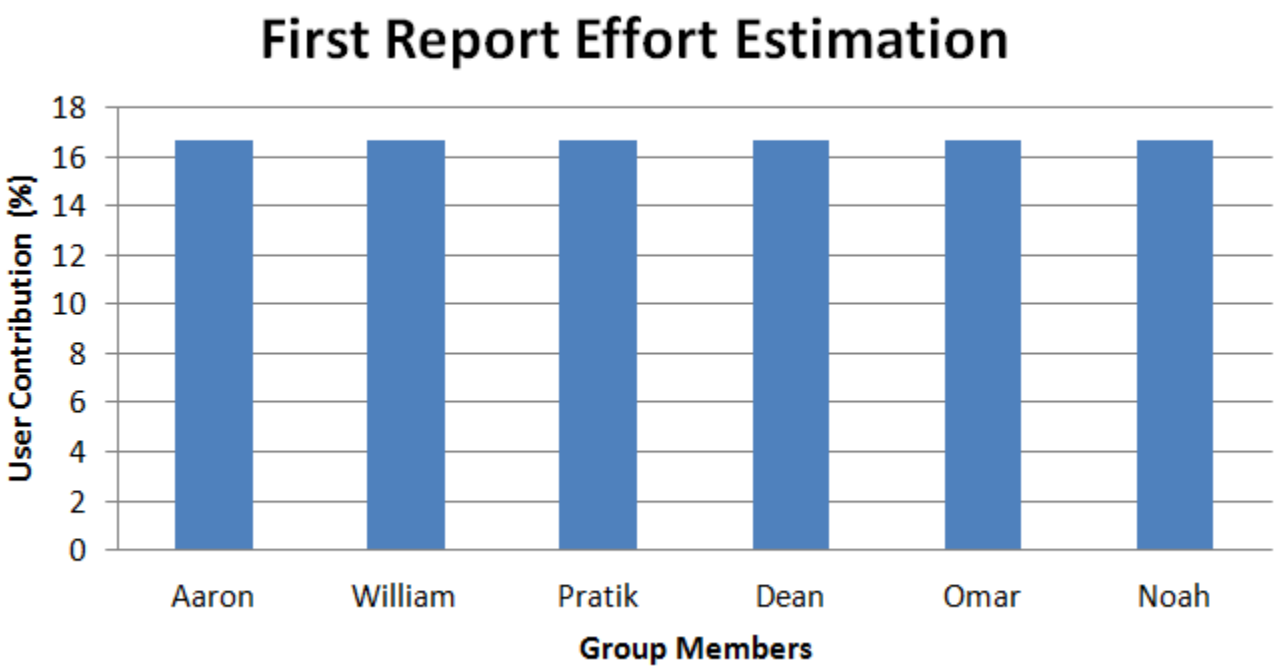
Group 6:
William Pan, Aaron Sun, Pratik Ringshia
Dean Douvikas, Omar Raja, Noah Silow-Carroll

URL: Not yet provided

February 17, 2012

Contributions Breakdown

Task	William	Aaron	Pratik	Dean	Omar	Noah
Statement of Reqs.	x	x				
a. Problem Statement	x	x				
b. Glossary of Terms	x		x		x	
System Reqs.	x	x	x		x	
a. Functional Reqs.	x	x				x
b. Nonfunctional Reqs.	x				x	x
c. Appearance Reqs.			x			
Functional Specs.	x	x				
a. Stakeholders	x	x				
b. Actors and Goals		x				
c. Casual Use Cases		x				
d. Use Case Diagram		x				
e. Fully Dressed		x				
f. Sequence Diagrams		x				
UI Specifications			x			
a. Preliminary Design			x			
b. User Effort Estimation			x			
Domain Analysis				x	x	
a. Domain Model				x	x	x
b. Operation Contracts				x	x	
c. Mathematical Model	x					
Plan of Work		x				
References	x					
Project Management	x					



The above chart summarizes the contributions from various team members in terms of effort. Based on the course website, our grades would normally be calculated by used a point breakdown for each section. However, we, the group, would much appreciate it if you could distribute the total points for this report as the chart dictates, where all team members have contributed equally. Many of the contributions from the team members cannot be quantified by the grading scheme, and we all worked equally.
Thank you.

Contents

1 Statement of Requirements

1.1 Problem Statement

Investing has long been the activity of the wealthy. The advent of the discount broker has lowered the barriers of entry so that almost anyone can become an active participant in the stock market. Nevertheless commission costs, the risk of losing money, and a lack of capital can still drive off would-be investors. Bears & Bulls strives to remove these remaining deterrents by simulating a discount broker and allowing users to practice investing in a risk free environment. Most importantly, in keeping in line with what the investor wants, Bears & Bulls will simulate the real-life stock market.

To fulfill the investor's requirements, Bears & Bulls provides many of the services of a real-life broker. It allows investors to create and manage portfolios through its user friendly interface. The investor has the ability to buy and sell stocks through market, limit, buy stop and stop loss orders. Bears & Bulls also supports margin accounts, and allows investors to buy on margin, providing capabilities that an investor might not ordinarily have the means to afford. Bears & Bulls will use real world data by retrieving actual stock information and executing the orders based on these prices. Since no real assets are being exchanged, Bears & Bulls will determine price slippage for large trades or volatile markets to better simulate a real transaction.

An investor's portfolio will contain information about the stocks that he currently owns, such as quantity, current market price, total gain and ticker symbol. This will give the investor a clear overview of his holdings, and allow him to evaluate his current standings. Bears & Bulls will keep a history of the investor's transactions so that he can refer back to them to reevaluate his strategies.

As with all major brokers today, Bears & Bulls will give the investor access to a wide range of market data. Investors can use Bears & Bulls to access critical market information, such as charts, fundamental indicators and technical indicators. Bears & Bulls will also support watchlists, which give investors a quick summary of stocks they are interested in. Overall, Bears & Bulls' goal is to strike a balance between ease of use and depth in order to appeal to beginners and veteran traders alike.

Unlike other market simulators, Bears & Bulls will be introduced as a Facebook application to take advantage of Facebook's large user base and the growing trend of social networking. Integrating Bears & Bulls into Facebook will streamline the login progress and allow users to access the application directly from their Facebook account. This eliminates the need for a lengthy registration process and will also

allow users to keep tabs on their friends and exchange trading ideas.

To create a more compelling user experience, Bears & Bulls introduces the ability to create, join and compete in leagues. Leagues provide users a way to test their investing mettle against friends or other players within Bears & Bulls. Leagues can be public or private, and the creator can decide the rules of the league, as well as who can and cannot participate in it. The ability to place entrance fees and payouts to winners adds an additional dimension of competitiveness.

In order to include everyone in the social aspect of the game, Bears & Bulls offers it own public leagues. Every portfolio an investor manages will be associated with a league. Bears & Bulls' Public leagues are open ended and provide investors an environment to invest in without the pressure of competition. The best performing portfolios will still be ranked so skilled investors can demonstrate their investing acumen.

Perhaps the most exciting feature that Bears & Bulls introduces is the concept of *Funds*. Bears & Bulls allows investors to create their own funds, either a hedge fund or a mutual fund, and manage other investors' money. This feature has not been found in any existing stock market simulator and is completely unique to Bears & Bulls. Investors confident in their abilities can set up a fund and try to entice other investors to invest in it. The fund managers will be able to set the rules of the fund, including who they accept money from, what their management fees are, and what strategies they will employ.

Communication is central to the design of Bears & Bulls. By encapsulating it within Facebook, users are provided a suite of tools to share their thoughts on various trades. As the only application of its kind in Facebook, it is unlikely that user's will be perfectly satisfied with Bears & Bulls. As such, Bears & Bulls also facilitates communication between users and system developers by including a convenient comment submission system. This will help Bears & Bulls' developers make improvements as the program grows.

1.2 Glossary of Terms

League Coordinator – A player who acts as an administrator of a private *league*. Responsibilities include inviting/deleting users and managing details of the *league*, such as entry fee.

Fund – A pooled investment vehicle. *Funds* are run by managers who recieve either a maintenance fee, performance fee or both. *Investors* may invest in a *Fund* a if they believe the *Fund's* manager can help them realize greater gains.

Investor – A person who commits capital expecting to see his/her capital grow in value. *Players* in our system are *investors*.

League – A *league* is a registered group with a particular set of rules. *Leagues* are comprised of *players*. There are multiple types of *leagues*.

- **Global** – A *league* comprised of all *players* of the game. Upon joining the Bears & Bulls, players are automatically added to this *league*. There is only one global *league*. *Coordinators* of the *league* are the creators of this program.
- **Private** – A *league* which is headed by a *coordinator* and requires approval before active membership.
- **Public** – A *league* without a *coordinator*. It can be joined by users without needing approval.

Order Ticket – Form *players* must complete to place an order for the sale or purchase of an *asset*.

Player – A user who registers with our service and creates a *portfolio*. This member joins *leagues* and competes with existing members. Synonomous with *investor* and *user*.

Portfolio – Detailed account of *assets* associated with each of a *league's players*. A *player* will have a unique portfolio per *league*. The player's goal is to maximize the value of his portfolio in comparison with the rest of the *league's* members.

Slippage – Price difference between what a trade executes at and the price of the previously executed trade.[10]

Stock – A type of *asset* that represents ownership of a corporation. *Players* will be able to purchase and sell stocks for their *portfolios*.

Stop Order – A type of order used to protect gains or limit losses. Stop loss orders are activated if a stock drops below the stop price and buy stop orders are activated if a stock rises above the stop price.

Ticker Symbol – A unique series of letters assigned to a *stock* for the purpose of trading.

User – A person who would use the system. Synonomous with *Investor*.

Volatility – The tendency for a stock's price to make drastic moves.

Watchlist – A list of *stocks* that displays relevent information regarding each *stock*. Each *investor* has a watchlist and may add and remove *stocks* from the list.

2 System Requirements

2.1 Function Requirements

PW = Priority Weight

ID	PW	Requirement
REQ-1	5	The system shall allow new users to register an account with their Facebook profile.
REQ-2	5	The system shall support order placement by filling out an order ticket. The order ticket shall include order type, quantity, symbol, price type and term. The order ticket shall be placed in an order queue to be processed.
REQ-3	5	The system shall review the order queue periodically and: • Immediately execute market orders. • Convert order to market order if order conditions are met. • Remove canceled or expired orders • If none of the above, leave order untouched.
REQ-4	5	The system shall maintain a database of user portfolios and transactions. The database will also include *league* rankings for each player
REQ-5	4	The system shall support investing *leagues*. Users shall be allowed to create *leagues* and specify prizes, duration, capital limits and entrance fees. The system shall also support official *leagues* and rankings based on return on investment.
REQ-6	4	The system shall provide market data, including: • Charts • Fundamental Indicators P/E, Range, Beta • Techinical Indicators Moving Averages, Bollinger Bands, . . . [8]

REQ-7	4	The system shall allow users to create and manage *Funds*. The rules of a *Fund* are specified when the *Fund* is created. These rules include the types of trades they are allowed to do and the types of assets they are allowed to hold. *Investors* can choose to invest money in *Funds* and *Fund* managers can choose to accept or decline *investors*.
REQ-8	3	The system shall use Facebook Credits to accept entrance fees from *league* participants and make payments to *league* winners
REQ-9	3	The system shall simulate market liquidity when trading high volumes of stocks
REQ-10	2	The system shall support margin accounts. The system shall require an initial and mainentance margin for assets purchased on margin. The system shall automatically exit positions that fall below maintenance margin. The user shall be notified that his position has been exited.
REQ-11	1	The system shall allow users to submit comments to the system administrators.

2.2 Nonfunctional Requirements

ID	PW	Requirement
REQ-12	5	The system shall be simple to use and have a minimal learning curve. Data shall be presented in such a way that the user's focus is automatically drawn to it when the user views the page. Whenever a user navigates to a page the main content of the page shall be placed at the center of the screen and the user shall not have to scroll to view the data or access the majority of the options on the page.
REQ-13	5	All user data shall be stored in the system's database. No user information shall be stored on the user's device. User's shall not be able to directly modify any data. There must be at least two copies of every record in case of system failure.
REQ-14	4	The system shall have a common aesthetic theme and any two pages shall be separated by no more than 4 links.

REQ-15	3	The system performance shall be consistent. Users shall not experience any notable latency from the system. Users with a broadband connection shall not experience more than 0.1 s between executing commands and seeing the result.
REQ-16	3	The sytem shall be platform independent and should run equally well on Windows, Mac and *nix systems. The system shall have consisten appearance between browsers.
REQ-17	2	The system shall require minimal maintenance. The system shall require maintenance at most once a week.
REQ-18	2	The system shall maintain function in the event of any changes to Facebook's API.

2.3 On-Screen Appearance Requirements

Since Bears & Bulls is tightly integrated with Facebook, some guidelines must be observed when designing the website. Since pages will be embedded in an iframe within Facebook, a max-width of 760px must be respected for our app. Additionally, to keep consistent with Facebook's UI experience, Bears & Bulls will avoid using any pop-ups. We will therefore be using modal dialogs in place of Javascript alert boxes where needed. Additionally, although the iframe sandboxes much of the app's functionality from the Facebook webpage, care must be taken to avoid interference between any front-end scripts that Bears & Bulls might use. Another limitation occurs in the domain of advertisements, as we must adhere to Facebook's Advertising Guidelines (http://www.facebook.com/ad_guidelines.php) to curate the content and positioning of our advertisements.

3 Functional Requirements

3.1 Stakeholder

- *Facebook Users* who wish to use the system for entertainment.
- *Novice Investors* who wish to use the system to practice investing.
- *Sponsors* who wish to advertise by creating and sponsoring *leagues*.
- *System Administrators* who will maintain the system as well as manage the global *league*.

3.2 Actors and Goals

- Investor – Initiating Actor, Participating Actor
  1. To create an account
  2. To make trades
  3. To research stocks
  4. To view transaction and player ranking history
  5. To view and edit account information
  6. To view porfolios and balances
  7. To create and/or join investment *leagues*
  8. To create and/or join a *Fund*
  9. To submit comments to system administrators
  10. To add stocks to his/her *watchlist*
- League Coordinator – Initiating Actor
  1. Invite other users to the investment *league*
  2. Add coordinators
  3. Manage *league* details
  4. Delete members
- System Administrator – Initiating Actor, Participating Actor
  1. To maintain the database and website
  2. To view messages from users
  3. To receive payments form advertisers and make payments to winners
- Stock Info Provider – Participating Actor
- Database Server – Participating Actor
- Web Server – Participating Actor
- Facebook – Participating Actor

3.3 Use Cases

3.3.1 Casual Description

Use Case UC-1: Buy Stock

Actor: Investor (*Initiating*), Stock Info Provider (*Participating*), Database (*Participating*)

Goal: To buy a stock. This involves filling out and submitting an order ticket and includes market, limit, buy to cover and buy stop orders. Buy orders may use margin if the *investor's* account is a margin account. Market prices will be queried from Stock Info Provider.

Use Case UC-2: Sell Stock

Actor: Investor (*Initiating*), Stock Info Provider (*Participating*), Database (*Participating*)

Goal: To sell a stock. This involves filling out and submitting an order ticket and includes, limit short sell and stop loss orders. Market prices will be queried from Stock Info Provider.

Use Case UC-3: Query Stock

Actor: Investor (*Initiating*), Stock Info Provider (*Participating*), Database (*Participating*)

Goal: To search ticker symbols and view market information for specified stock. Information will include prices, charts, fundamentals, news articles, etc. Information will be queried from Stock Info Provider.

Use Case UC-4: View History

Actor: Investor (*Initiating*), Database (*Participating*)

Goal: To view transaction history and player ranking history. Transaction history is a compilation of previous trades; player ranking history is a compilation of daily player rank within a *league*.

Use Case UC-5: View Portfolio

Actor: Investor (*Initiating*), Stock Info Provider (*Participating*), Database (*Participating*)

Goal: To view portfolio and balances. This includes all currently owned stocks as well as monetary balances.

Use Case UC-6: Watch Stock

Actor: Investor (*Initiating*), Database (*Participating*)

Goal: To add stocks to *watchlist*. The *watchlist* will display the stocks and their prices on the user's homescreen. The *watchlist* will email the user when user-set pricepoints have been met.

Use Case UC-7: Register

Actor: Investor (*Initiating*), Database (*Participating*), Facebook (*Participating*)

Goal: To register for an account. This creates a game account that will retrieve user information from Facebook.

Use Case UC-8: Create League

Actor: Investor (*Initiating*), Database (*Participating*)

Goal: Create an investment *league*. Upon creating a *league*, the *investor* is given the position of coordinator within the *league*.

Use Case UC-9: Pay League

Actor: Investor (*Initiating*), Database (*Participating*)

Goal: To pay investment *league* entrance fee for *leagues* with entrances fees.

Use Case UC-10: Submit Comment

Actor: Investor (*Initiating*), Database (*Participating*), Facebook (*Participating*)

Goal: To submit comments to system administrators. This allows *investor* to provide feedback to system admins.

Use Case UC-11: Create Fund

Actor: Investor (*Initiating*), Database (*Participating*)

Goal: To create a *Fund* (hedge/mutual fund). Upon creating the *Fund*, the *investor* becomes the *Fund's* manager.

Use Case UC-12: Join League

Actor: Investor (*Initiating*), Investor (*Participating*), Database (*Participating*)

Goal: To join a *league* and participate in it.

Use Case UC-13: Manage League

Actor: League Coordinator (*Initiating*), Database (*Participating*)

Goal: To manage *league* details such as setting entrance fee, demoting coordinators, and setting *league* rules.

Use Case UC-14: Invite to League

Actor: League Coordinator (*Initiating*), Investor (*Participating*)

Goal: To invite other *investors* to joing the *league*. Invitations are the only way to joing private *leagues*.

Use Case UC-15: Add Coordinator

Actor: League Coordinator (*Initiating*), Investor (*Participating*), Database (*Participating*)

Goal: To designate another *league* member as a *League Coordinator*.

Use Case UC-16: Remove User

Actor: League Coordinator (*Initiating*), Database (*Participating*)

Goal: To eliminate an existing user account and disable access from a *league*. This function is used to remove users when deemed necessary.

Use Case UC-17: Update Models

Actor: Sys Admin (*Initiating*)

Goal: To update liquidity model that simulates price slippage during high volatility and block trades.

Use Case UC-18: View Comment

Actor: Sys Admin (*Initiating*), Facebook (*Participating*)

Goal: View user comments. Comments will be logged and taken into consideration for future patches to the system.

Use Case UC-19: Manage Money

Actor: Sys Admin (*Initiating*), Facebook (*Participating*)

Goal: To receive Facebook credits from *league* participants for *leagues* with entrance fees and pay *league* winners.

Use Case UC-20: Manage Fund

Actor: Fund Manager (*Initiating*), Database (*Participating*)
Goal: To accept and decline *investors* who wish to invest in the *Fund*.

3.3.2 Use Case Diagram

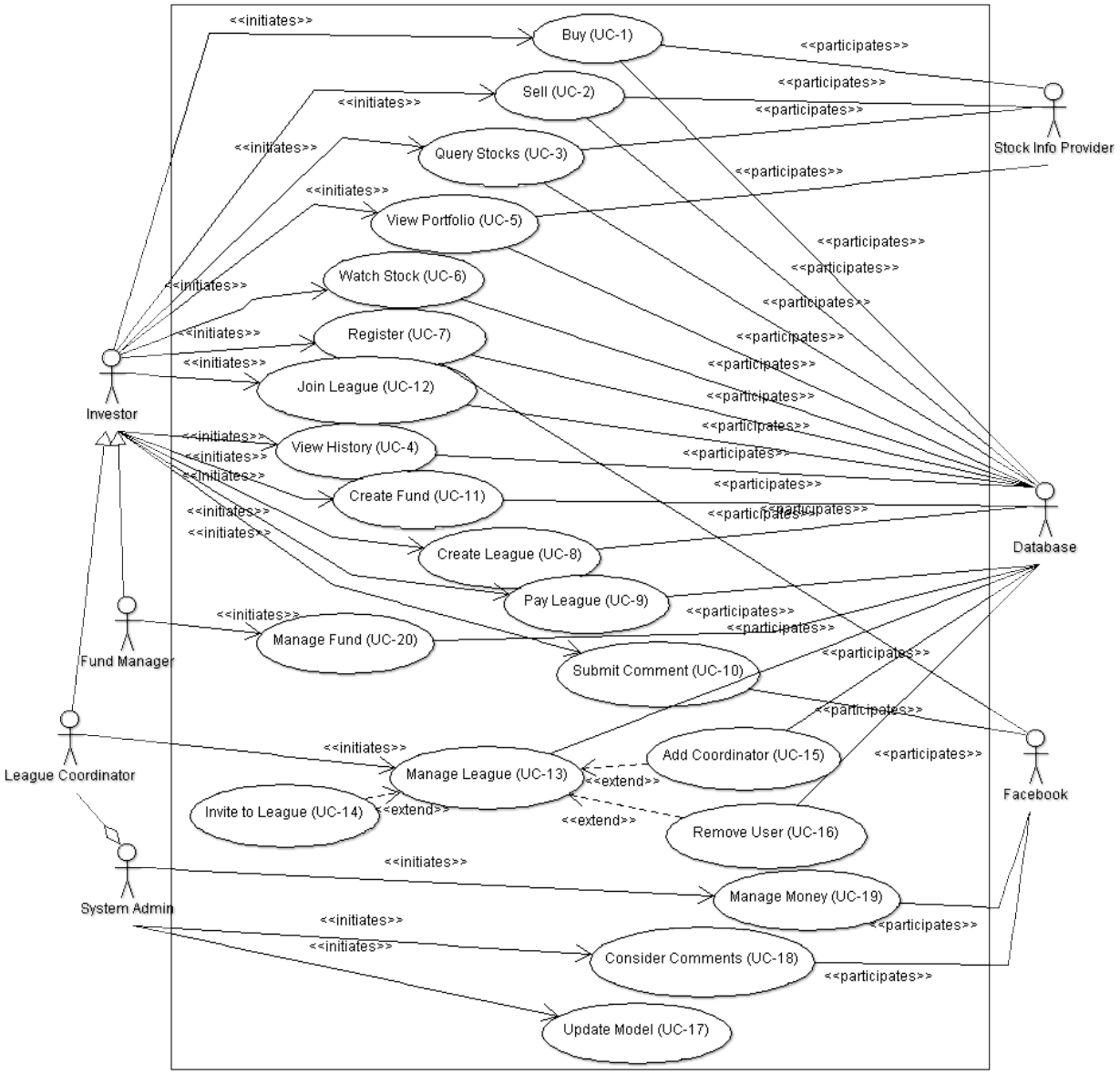


Figure 1: Use Case Diagram [2]

In this diagram, the actors involved with a specific use case are shown. The *investor*, system admin, *League Coordinator*, and *Fund* manager are the only ini-

tiating actors in this diagram, and there are no non-human initiating actors. The database is seen to participate in almost every use case except some of those that are initiated by the system admin. The majority of the use cases are also initiated by the *investor*, with some use cases being initiated by the more specialized *League Coordinator* and *Fund* manager. In this design, we have that the *Fund* manager only has one other use case than the regular *investor* because his duties reflect very closely to those of the *investor*. The *League Coordinator* assumes many more responsibilities than a normal *investor* (like Invite to League and Add Coordinator). Facebook participates in everything dealing with money, and also it provides the comment functionality. The stock info provider only participates in four use cases, all of which query it at some point in their action. The use case diagram here shows that there is low coupling within the system because almost all use cases have only a total of two actors either initiating or participating.

In an alternate scheme (not depicted), the stock info provider is regularly queried by the system, and the system sends the data to the database where it is stored. This makes the stock info provider assume much less responsibilities, and it would only be a participating actor in potentially one use case (something along the lines of System Query). Also, the *Fund* Manager has his own set of use cases in managing a *Fund* (like Fund Buy and Fund Sell) in order to reflect that it is a *Fund* that is carrying out these duties. This alternate scheme was not chosen because we did not wish for the database to hold that much information, and also the *Fund* Manager's duties seemed too similar to the *Investor's* duties to warrant a new set of use cases.

3.3.3 Fully-Dressed Description

Use Case UC-1: Buy Stock

Related Requirements: REQ-2, REQ-3, REQ-9, REQ-10
Initiating Actor: Investor
Actor's Goal: To buy a stock and add it to his portfolio
Participating Actors: Database, Stock Info Provider
Precondition: The user must have an account and have enough money for the purchase.
Postcondition: The user's portfolio must be debited the amount of the purchase and the stock must be added to the user's portfolio. Information about the stock must be updated for the lifetime of the stock in the portfolio.
Flow of Events for Main Success Scenario:

1 → The *investor* searches for a ticker symbol and fills out and submits an

order ticket with the amount he wishes to buy.

2 → The system adds the order ticket to the order queue, and when the system reaches the ticket, the system queries the stock info provider for the price of the stock.

3 ← Stock info provider sends the price of the stock to the system.

4 → The system determines the price of the buy and if order conditions are met it queries the database for the *investor's* balance.

5 ← The database sends the *investor's* balance to the system.

6 → The system determines that the balance is enough to satisfy the buy and the system signals the database to perform the transaction.

7 ← The database adds the stock to the *investor's* portfolio, his balance is decreased by the buy amount, and the transaction is recroded in the transaction history. The database signals to system that the transaction is complete.

8 ← The system signals to the player: "Transaction Completed".

Flow of Events for Not Enough Money:

1 → The *investor* searches for a ticker symbol and fills out and submits an order ticket with the amount he wishes to buy.

2 → The system adds the order ticket to the order queue, and when the system reaches the ticket, the system queries the stock info provider for the price of the stock.

3 ← Stock info provider sends the price of the stock to the system.

4 → The system determines the price of the buy and if order conditions are met it queries the database for the *investor's* balance.

5 ← The database sends the *investor's* balance to the system.

6 ← The system signals to the player: "Error in transaction: Balance too low". The order ticket is removed from the list.

Use Case UC-2: Sell Stock

Related Requirements: REQ-2, REQ-3, REQ-9, REQ-10

Initiating Actor: Investor

Actor's Goal: To sell a stock from his portfolio and receive cash from it.

Participating Actors: Database, Stock Info Provider

Precondition: The user must have an account and have enough of the particular stock for the sell.

Postcondition: The user's credited the amount of the sale and the stock must be removed from the user's portfolio.

Flow of Events for Main Success Scenario:

1 → The *investor* searches for a ticker symbol and submits an order ticket with the amount he wishes to sell. (The order ticket will display the stocks the *investor* currently has when he chooses the option "sell").

2 → The system adds the ticket to the order queue, and after the system reaches the ticket in the queue the system queries the stock info provider for the price of the stock.

3 ← Stock info provider sends the price of the stock to the system.

5 → The system determines the price of the sell and when order conditions are met it queries the database for the number of shares of the stock in the *investor's* portfolio.

6 ← The database sends the number of stocks to the system.

7 → The system determines that the number of shares is greater than or equal to the amount he wishes to sell. The system signals the database to perform the transaction.

9 ← The database subtracts the number of shares from the *investor's* portfolio, his balance is credited by the sell amount and the transaction is recorded in the transaction history. The database signals to the system that the transaction is complete.

10 ← The system signals to the player: "Transaction Complete".

Flow of Events for Not Enough Stock:

1 → The *investor* searches for a ticker symbol and fills out and submits an order ticket with the amount he wishes to sell. (The order ticket will display the stocks the *investor* currently has when he chooses the option "sell");

2 → The system adds the ticket to the queue, and after the system reaches the ticket in the queue the system queries the stock info provider for the price of the stock.

3 ← The stock info provider sends the price of the stock to the system.

4 → The system determines the price of the sell and when order conditions are met it queries the database for the number of shares of the stock in the *investor's* portfolio.

5 ← The database sends the number of stocks to the system.

6 ← The system determines that the number of stocks is less than the number he wishes to sell.

7 ← The system signals to the player: "Error in Transaction: Not enough shares held". The order ticket is removed from the queue.

Use Case UC-3: Query Stocks

Related Requirements: REQ-2, REQ-3, REQ-6
Initiating Actor: Investor
Actor's Goal: To search ticker symbols and view market information for the stock
Participating Actors: Stock Info Provider, Database
Precondition: The user must have an account
Postcondition: The market information for the stock must be displayed on the screen.
Flow of Events for Main Success Scenario:
1 → The *investor* clicks the "Markets" link and searches a ticker symbol and queries a stock.
2 → The system queries market data from the stock info provider
3 ← The stock info provider sends the data to the system.
4 ← The system displays the market data on the page.
Flow of Events for Error in Retrieving Data:
1 → The *investor* browses to the "Markets" section and searches a ticker symbol and queries a stock.
2 → The system queries the market data from the stock info provider.
3 ← The stock info provider fails to send data to the system, and notifies the system that there was an error.
4 ← The system displays "Error retrieving data".

Use Case UC-5: View Portfolio

Related Requirements: REQ-4, REQ-5, REQ-6, REQ-7
Initiating Actor: Investor
Actor's Goal: To view his current portfolio and cash balances
Participating Actors: Database, Stock Info Provider
Precondition: The investor must have an account.
Postcondition: The investor's portfolio and balances must be displayed on the screen.
Flow of Events for Main Success Scenario:
1 → The *investor* navigates to "Portfolio & Balances".
2 → The system queries the database for the *investor's* portfolio and balances.
3 ← The database sends the *investor* portfolio and balances to the system.
4 → The system queries the stock info provider for the price information for the stocks held in the portfolio.
5 ← The stock info provider sends the requested data to the sytem.

6 ← The system displays the portfolio and balances with the stock information provided by the stock info provider.

Flow of Events for Error in Retrieving Data:

1 → The *investor* navigates to "Portfolio & Balances"

2 → The system quesries the database for the *investor's* portfolio and balances

3 ← The database sends the *investor's* portfolio and balances to the system.

4 → The system queries for stock information for the stocks held in the *investor's* portfolio.

5 ← The stock info provider fails to send the requested data to the system, and notifies the system that there was an error.

6 ← The system displays the portfolio and balances without the data provided by the stock info provider and displays without the data provided by the stock info provider and displays "Error Retrieving Data".

Use Case UC-7: Register

Related Requirements: REQ-1

Initiating Actor: Investor

Actor's Goal: To create an account

Participating Actors: Database, Facebook

Precondition: The system must support account creation.

Postcondition: A new account is in place for the user. This account will hold information such as name, portfolio holdings, balances, etc.

Flow of Events for Main Success Scenario:

1 → The *investor* navigates to the application Facebook's website and clicks "Get App"

2 ← Facebook displays page asking if the *investor* will allow the app to access information.

3 → The *investor* clicks "Allow".

4 ← Facebook authenticates the user.

5 ← The system signals to the database to create a new account with the above information.

6 ← The database creates the user account and signals to the system that the account was created.

7 ← The system signals to the user that an account has been created.

Use Case UC-8: Create League

Related Requirements: REQ-5
Initiating Actor: Investor
Actor's Goal: To create an investment *league*
Participating Actors: Database, Webmail Server
Precondition: The *investor* must have an account
Postcondition: The new *league* must be created, with the initiating *investor* as the *League Coordinator*.
Flow of Events for Main Success Scenario:
1 → The *investor* navigates to "Investment Leagues" and fills out a form with *league* name, entrance fee, starting funds, etc.
2 → The system determines that the *league* name is unique, and signals to the database to create a new *league* with the above information.
3 ← The database creates the stated *league*, and signals to the system that it has suceeded.
4 ← The system signals to the user "League Creation Successful!"
Flow of Events for Duplicate League Name:
1 → The *investor* navigates to "Investment Leagues" and fills out a form with *league* name, entrance fee, starting funds, etc.
2 ← The system determines that the *league* name is not unique, and the system signals to the user "League Name is Already Taken".
3 → Loop back to step 2 and continue to either main success scenario or alternate scenario.

Use Case UC-11: Create Fund

Related Requirements: REQ-4, REQ-7
Initiating Actor: Investor
Actor's Goal: To create a *Fund*
Participating Actors: Database
Precondition: The *investor* must have an account
Postcondition: The new *Fund* must be created, with the initiating *investor* as the *Fund* manager.
Flow of Events for Main Success Scenario:
1 → The *investor* navigates to "Funds" and fills out a form with *Fund* name, rules, restrictions, and description.
2 → The system determines that the *Fund* name is unique, and signals to the database to create a new *league* with the above information.
3 ← The database creates the stated *Fund*, and signals to the system that it has suceeded.

4 ← The system signals to the user "Fund Creation Successful!"

Flow of Events for Duplicate Fund Name:

1 → The *investor* navigates to "Funds" and fills out a form with *Fund* name, rules, restrictions, and description.

2 ← The system determines that the *Fund* name is not unique, and the system signals to the user "*Fund* Name is Already Taken".

3 → Loop back to step 2, and continue to either main success scenario or alternative scenario.

Use Case UC-13: Manage League

Related Requirements: REQ-5

Initiating Actor: League Coordinator

Actor's Goal: To manage *league* details such as starting balance, entry fee, duration, limiting capital, etc.

Participating Actors: Database

Precondition: The user changing *league* details must be the *League Coordinator*

Postcondition: The *league* details are successfully modified.

Flow of Events for Main Success Scenario:

1 → The *league coordinator* navigates to "Manage League".He then fills out the *league* information form and submits it to the system.

2 → The system determines that all changes are valid, and the system signals to the database to implement the changes.

3 ← The database implements the changes in *league* settings, and signals to the system that the changes were made successfully.

4 ← The system signals to the *league coordinator* that the settings were successfully saved.

Flow of Events for Invalid Changes:

1 → The *league coordinator* navigates to "Manage League".He then fills out the *league* information form and submits it to the system.

2 → The system determines that one or more changes are invalid, and the system signals to the user "Invalid Changes".

3 → Loop back to step 2, and continue to either the main success scenario or alternate scenario.

Use Case UC-14: Invite to League

Related Requirements: REQ-5

Initiating Actor: League Coordinator
Actor's Goal: To invite other *investors* to join the *league*.
Participating Actors: Database, Investor
Precondition: The inviter and invitee must have an account, and the inviter must be a *league coordinator*.
Precondition: The *investor* becomes a member of the *league*.
Flow of Events for Main Success Scenario:

1 → The coordinator chooses "Invite to League" and selects the appropriate *investor*.
2 ← The system notifies the invitee that he has been invited to join the league.
3 ← The system notifies the inviter that the invitation was sent.

Use Case UC-20: Manage Fund

Related Requirements: REQ-7
Initiating Actor: Fund Manager
Actor's Goal: To manage *Fund* details such as rules, restrictions, and descriptions.
Participating Actors:
Precondition:
Flow of Events for Main Success Scenario:

1 → The *league coordinator* navigates to "Manage Fund".He then fills out the *Fund* information form and submits it to the system.
2 → The system determines that all changes are valid, and the system signals to the database to implement the changes.
3 ← The database implements the changes in *Fund* settings, and signals to the system that the changes were made successfully.
4 ← The system signals to the *Fund* manager that the settings were successfully saved.

Flow of Events for Invalid Changes:

1 → The *league coordinator* navigates to "Manage Fund".He then fills out the *Fund* information form and submits it to the system.
2 → The system determines that one or more changes are invalid, and the system signals to the user "Invalid Changes".
3 → Loop back to step 2, and continue to either the main success scenario or alternate scenario.

3.3.4 Traceability Matrix

R# = REQ-#

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11		
PW	5	5	5	5	4	4	4	3	3	2	1	Max	Total
UC01		x	x						x	x		5	15
UC02		x	x						x	x		5	15
UC03		x	x			x						5	14
UC04				x								5	5
UC05				x	x	x	x					5	17
UC06						x						4	4
UC07	x	x		x	x		x			x	x	5	26
UC08					x							4	4
UC09								x				3	3
UC10											x	1	1
UC11				x			x					5	9
UC12					x							4	4
UC13					x							4	4
UC14					x							4	4
UC15					x							4	4
UC16					x							4	4
UC17									x			3	3
UC18											x	1	1
UC19					x			x				4	7
UC20							x					4	4

3.4 System Sequence Diagrams

NOTE: Diagrams of alternative implementations are in the appendix.

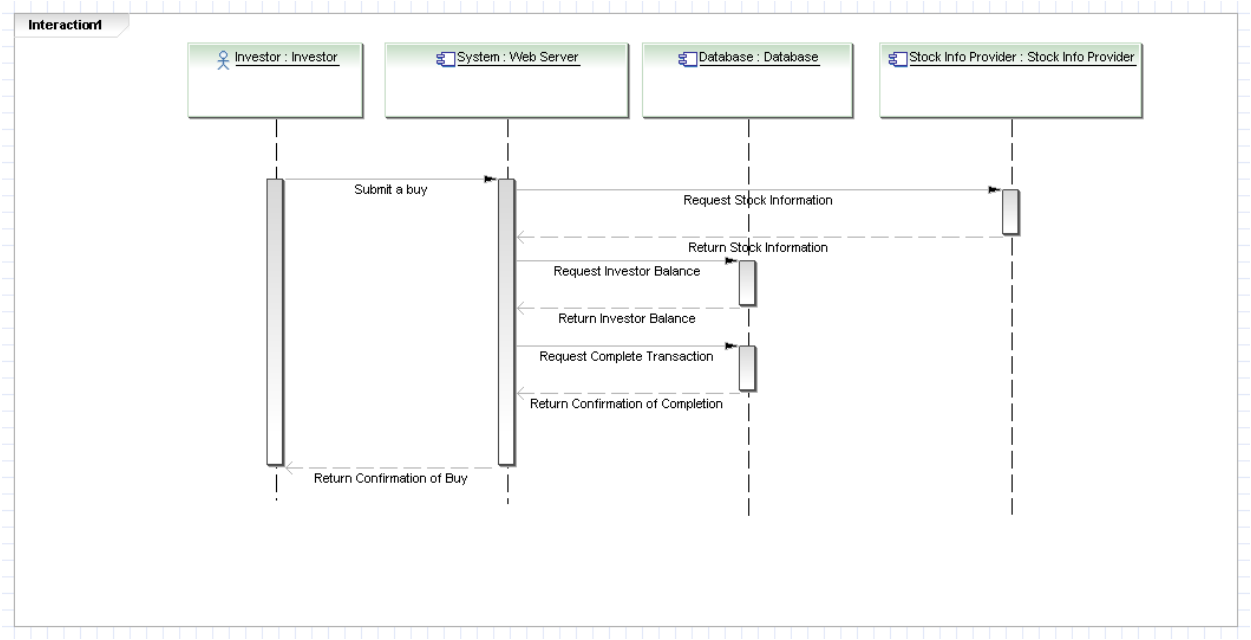


Figure 2: UC-1: Buy Stock

In this sequence diagram, the *investor* starts by submitting a buy to the web server. The system then requests the stock information from the stock info provider. Following this, the system requests the *investor's* balance from the database, and if the *investor* has enough money for the transaction, the system requests that the database complete the transaction. Lastly, the system returns a confirmation of the buy to the *investor*.

For the alternate scenario where the investor does not have enough money, the system does not request the database to carry out the transaction and instead sends an error back to the investor.

An alternative implementation for this use case was discussed, where instead of the system querying the stock info provider for the stock prices, it would query the database for the stock prices. In this model the database would have up to date information on the stock price info (from periodic queries from the system), and thus the stock info provider is left out of the transaction. However, this idea was not implemented because we decided it would be too much information to cache and would not be too practical.

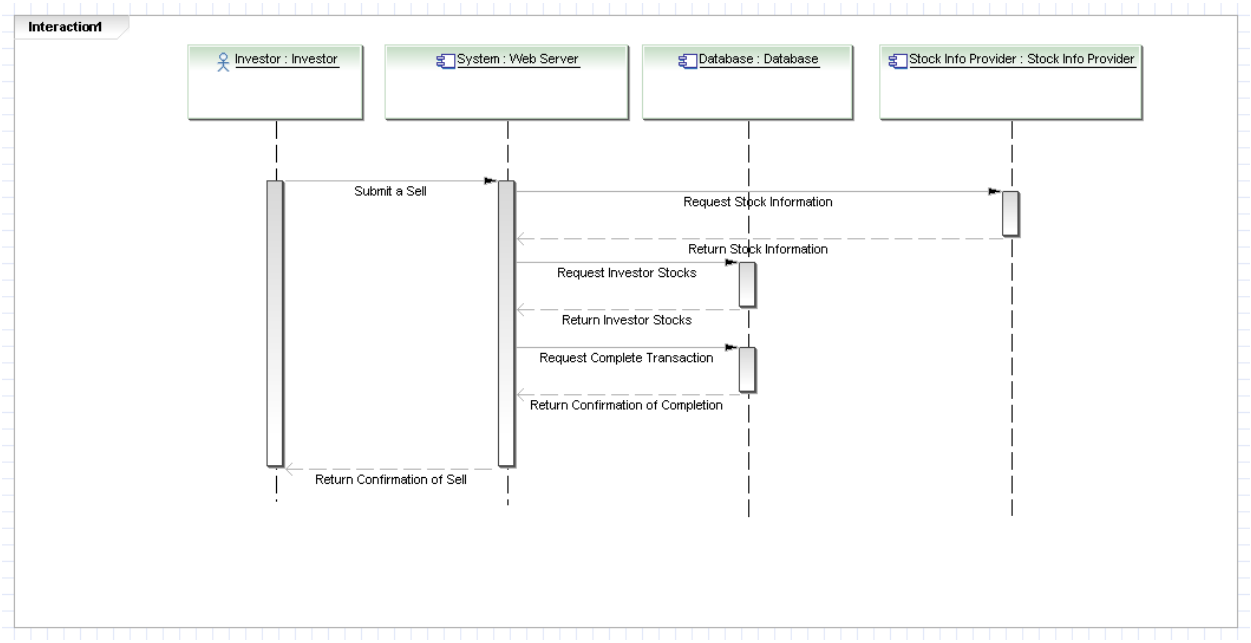


Figure 3: UC-1: Sell Stock

In this sequence diagram, the *investor* starts by submitting a sell to the web server. The system then requests the stock information from the stock info provider. Following this, the system requests the amount the stock that the *investor* holds from the database, and if the *investor* has enough stocks for the transaction, the system requests that the database complete the transaction. Lastly, the system returns a confirmation of the sell to the *investor*.

For the alternate scenario where the investor does not have enough stocks, the system does not request the database to carry out the transaction and instead sends an error back to the investor.

An alternative implementation for this use case was discussed, where instead of the system querying the stock info provider for the stock prices, it would query the database for the stock prices. In this model the database would have up to date information on the stock price info, and thus the stock info provider is left out of the transaction. However, this idea was not implemented because we decided it would be too much information to cache and would not be too practical.

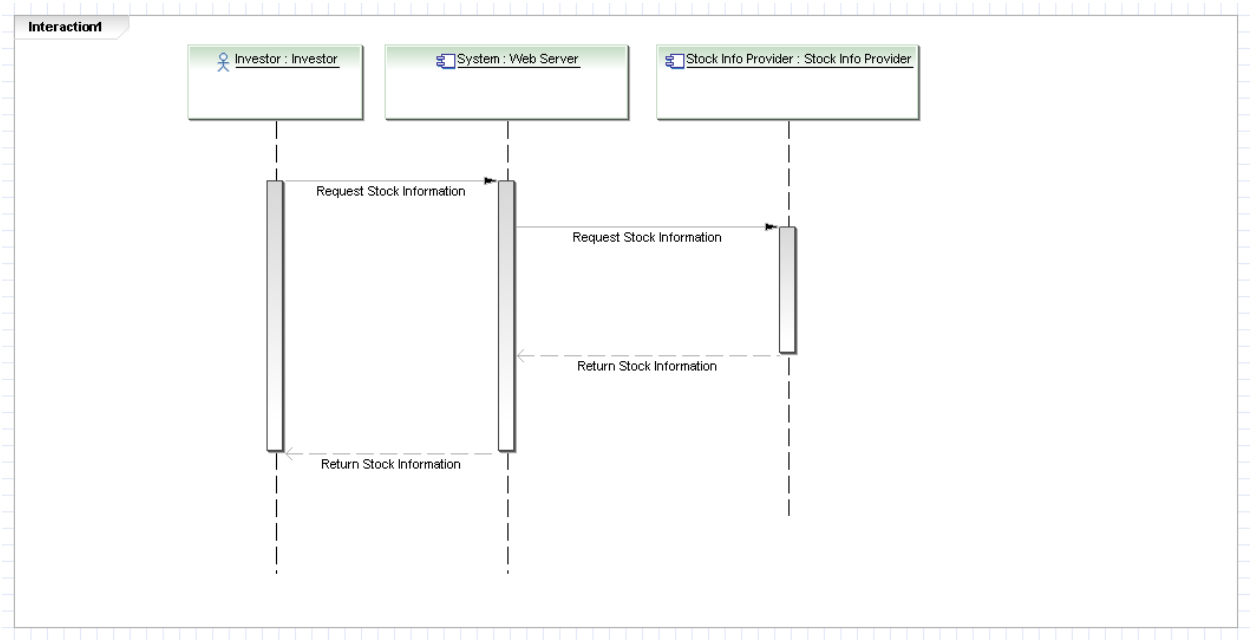


Figure 4: UC-3 Query Stock

In this sequence diagram, the *Investor* first requests stock information from the system. The system then queries the stock information from the stock info provider, and then feeds it back to the *Investor*.

For the alternate scenario where the request from the stock info provider fails, the system sends an error back to the user that it was not able to retrieve the data.

As discussed in the buy and sell use cases, an alternative implementation would have been that the system queries the database instead for the information. (Please see Sequence Diagram for Buy Stock for full discussion on it).

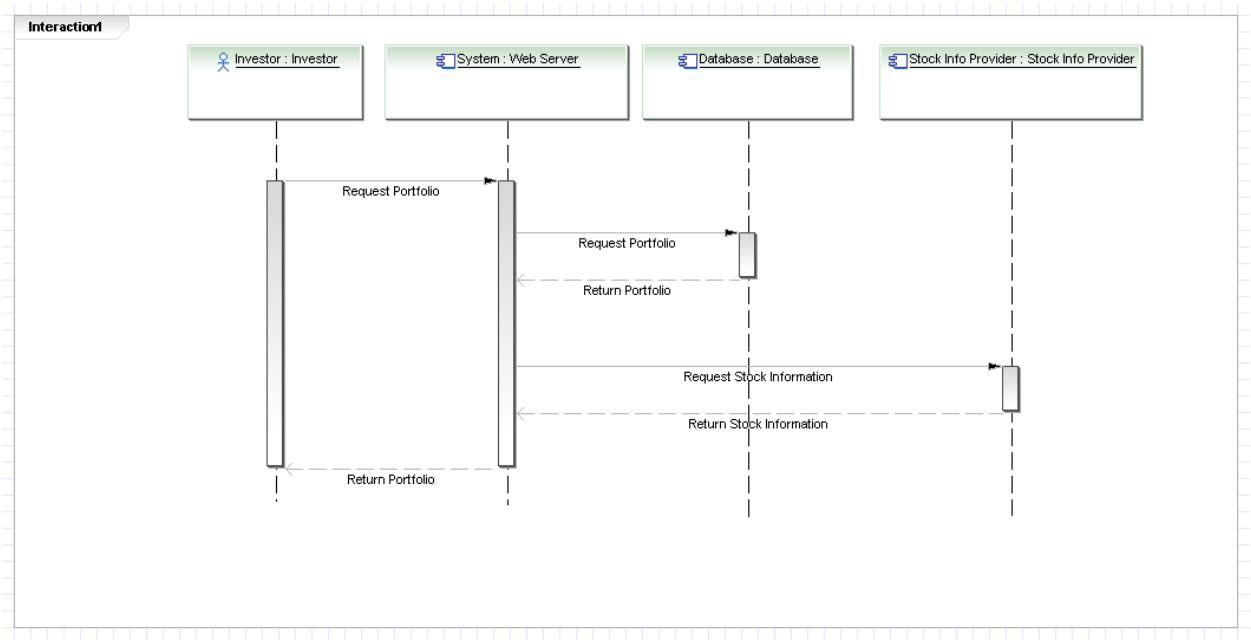


Figure 5: UC-5 View Portfolio

In this sequence diagram, the *investor* starts off by navigating to his portfolio and in this way requests the portfolio from the system. The system then requests the portfolio from the database, and when the portolio is returned to the system, the system then queries the stock info provider for the current stock prices for the stocks in the portfolio. The system then sends back to the user the portfolio with the stock prices.

For the alternate scenario where the system is unable to retrieve data from the stock info provider, it will instead return an incomplete portfolio to the investor where the stock names and quantities are displayed, but no information on the stock is given.

As discussed in the buy and sell use cases, an alternative implementation would have been that the database stores the stock prices. When the system makes the call to the database, the database compiles both the portfolio as well as the prices before sending it off to the system. This implementation cuts out the step of the system querying the stock info provider, but again it was determined that this would not be the most efficient strategy because the system would have to query the stock info provider periodically for the price updates and store it in the database cache. (Please see the Sequence Diagram Buy Stock above for full discussion on it).

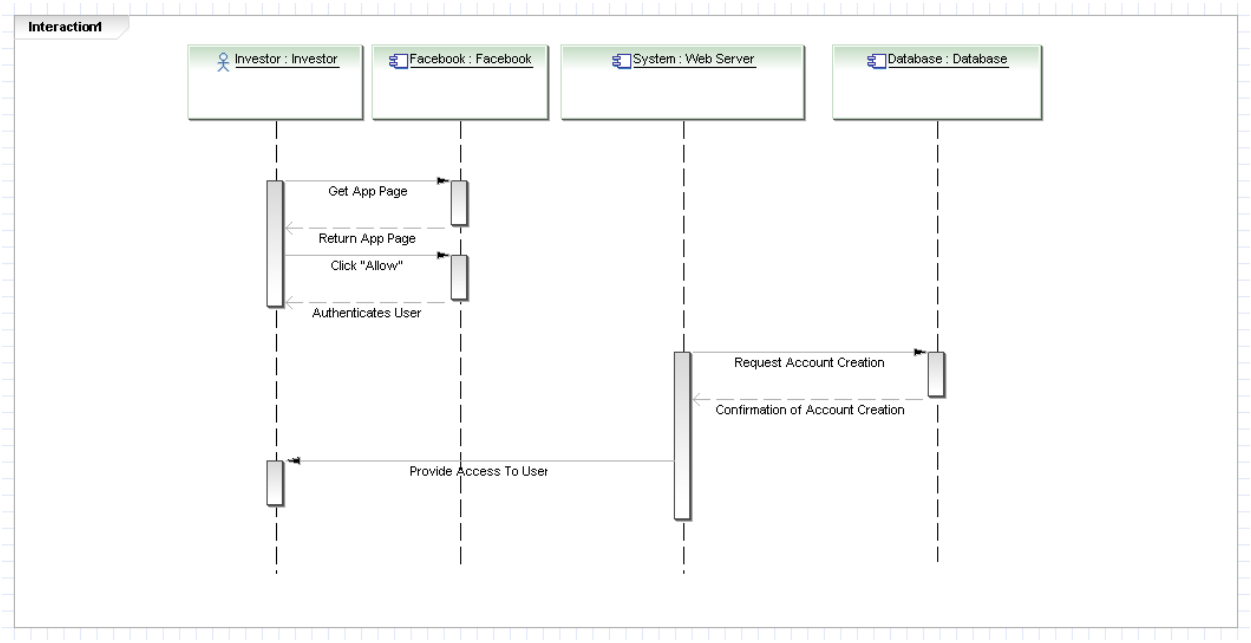


Figure 6: UC-7 Register

In this sequence diagram, the *investor* first navigates to the app and clicks "Get App" on the Facebook website. Folowing this, Facbook returns the App page to the *investor*, prompting the *investor* to make a choice of whether to allow the app to access information or not allow it. The *investor* clicks "Allow" and Facebook authenticates the *investor*. The system then requests from the database that an account be created, and when this is done the system provides access the *investor* access to the site.

An alternate implementation for this use case was discussed where instead of creating an account directly for the *investor*, the *investor* would have to first navigate within the system and click "Register" within the system after being authenticated by Facebook. However, this idea was discarded because it seemed unnecessary for the *investor* to go through that route.

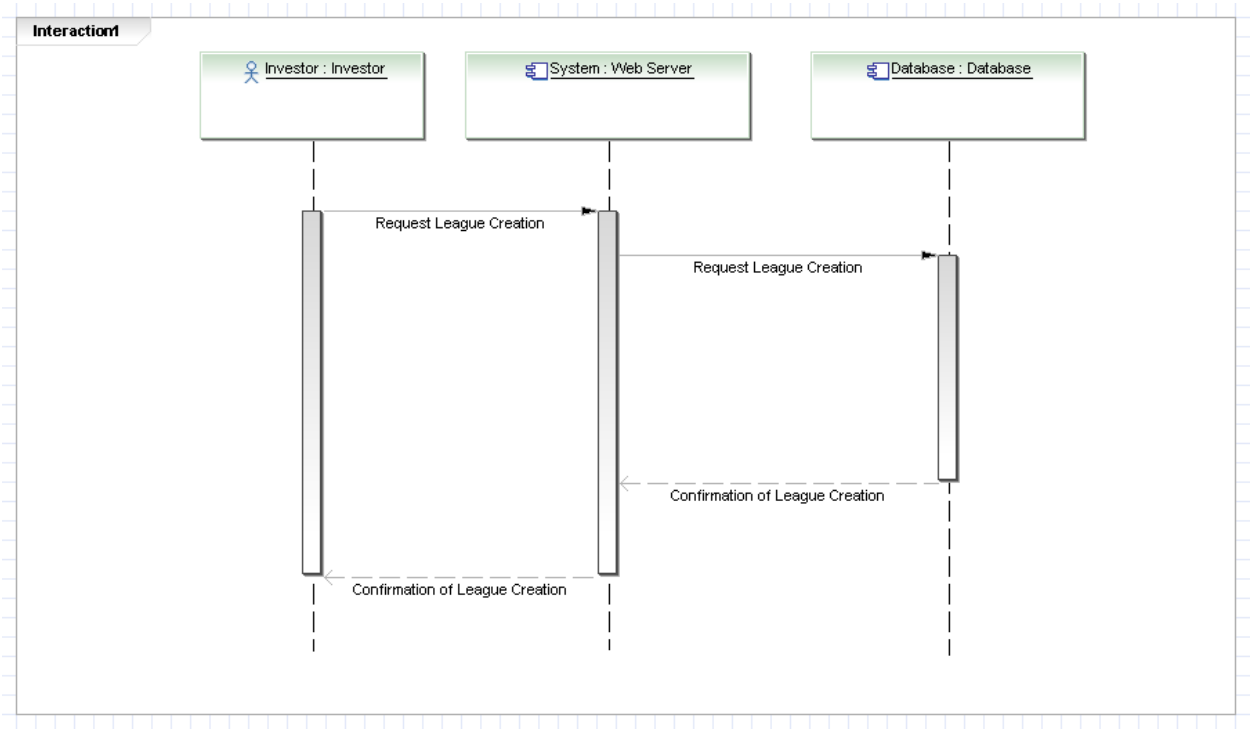


Figure 7: UC-8 Create League

In this sequence diagram, the *investor* requests a new *league* from the system, which in turn requests a *league* from the Database to be created. After the account is created, the Database signals to the system that is has been created, and the system signals back to the *investor* that the *league* has been created.

For the alternate scenario, if the database detects that there is a duplicate name, it will return an error to the system, which in turn will give an error back to the investor.

There were no real alternative implementations that we discussed, since this seemed the only logical procession of events.

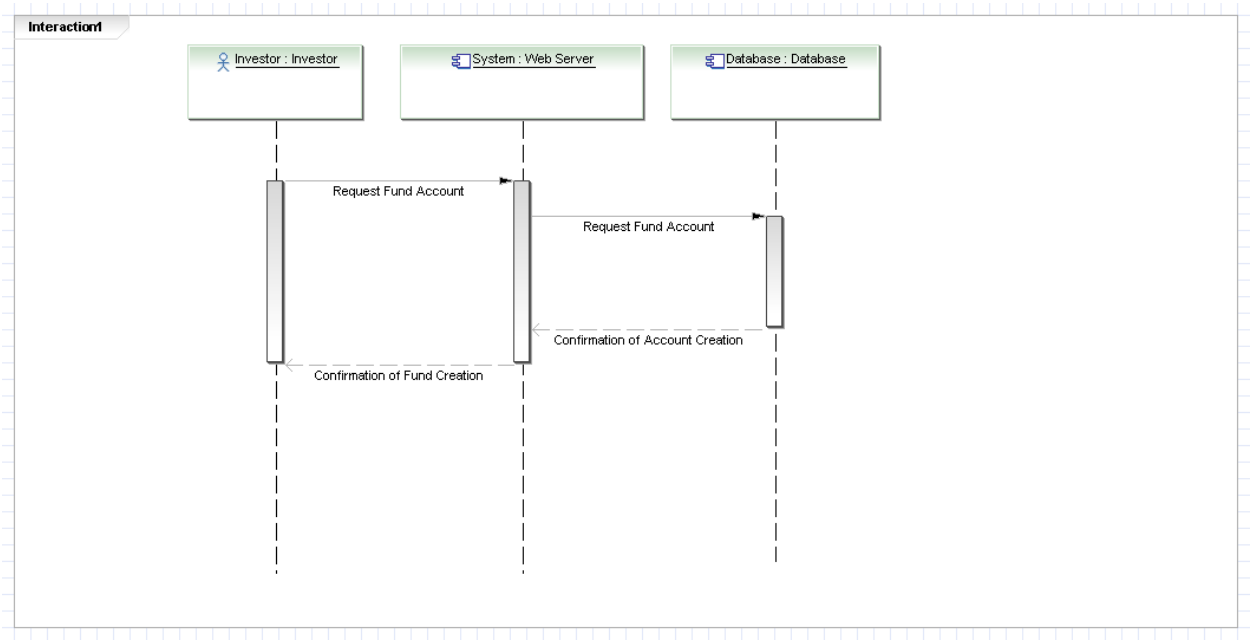


Figure 8: UC-11: Create Fund

In this sequence diagram, the *investor* requests a new Fund Account from the system, which in turn requests a Fund Account from the Database to be created. After the account is created, the Database signals to the system that is has been created, and the system signals back to the *investor* that the account has been created.

For the alternate scenario, if the database detects that there is a duplicate name, it will return an error to the system, which in turn will give an error back to the investor.

There were no real alternate implementations that we discussed, since this seemed the only logical procession of events.

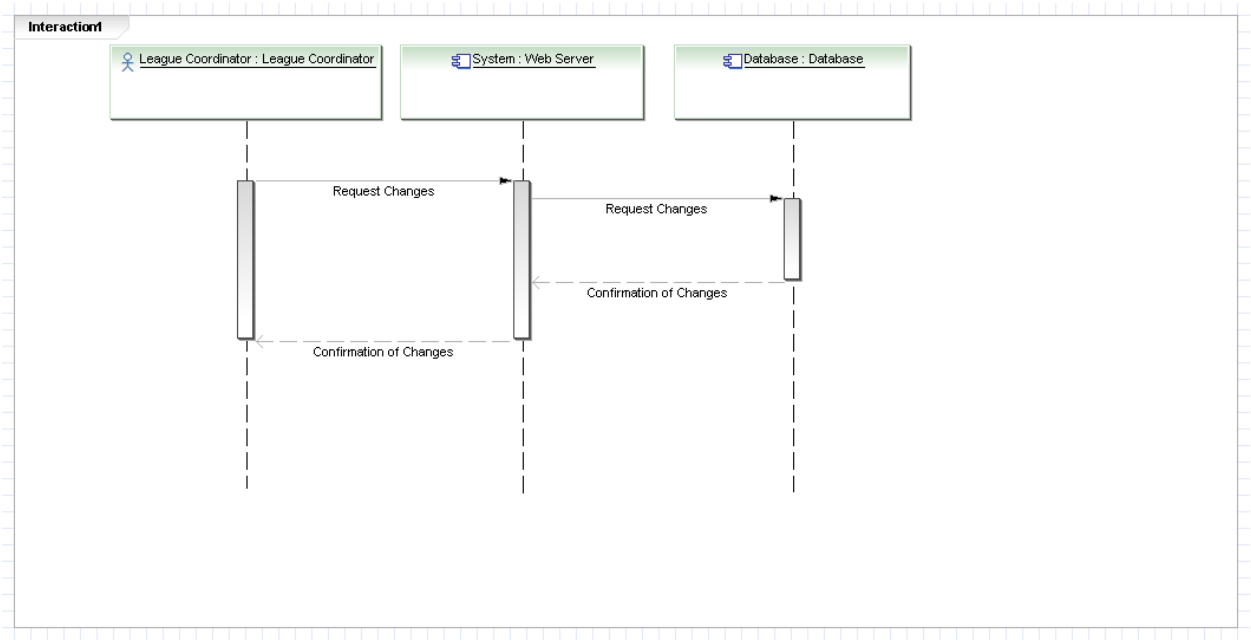


Figure 9: UC-13 Manage League

In this sequence diagram, the *League Coordinator* requests that changes be made to the *league* settings, and after the system has verified the changes are valid, it sends a request to the database to implement these changes. After these changes have been made, the database signals to the system that the changes have been made, and the system then signals to the *League Coordinator* that the changes have been made.

For the alternate scenario, if the database detects that there is an error in the changes (for example an invalid value entered in a field), it will return an error to the system, which in turn will give an error back to the investor.

There was no alternative implementation of this use case that was discussed since this one seemed to be the only logical way to do it given the actors that we had.

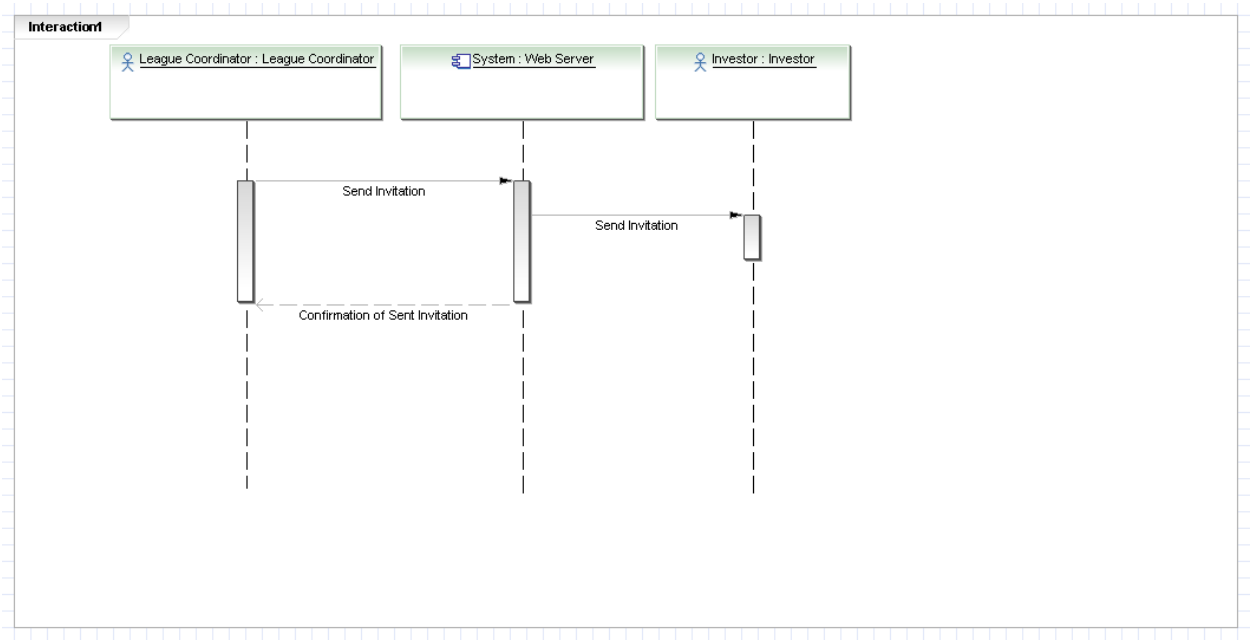


Figure 10: UC-14 Invite to League

In this sequence diagram, the *League Coordinator* requests that the system sends an invitation to the *investor*, and after the system has sent it to the *investor*, the system returns a confirmation to the *League Coordinator* that an invitation was sent.

There was an alternative implementation of this use case where we discussed the possibility that *league* members could also invite other *investors*. However, this seemed to be an unwise choice since it could cause a mass of unwanted invitations, thus this functionality was restricted to the *League Coordinator*.

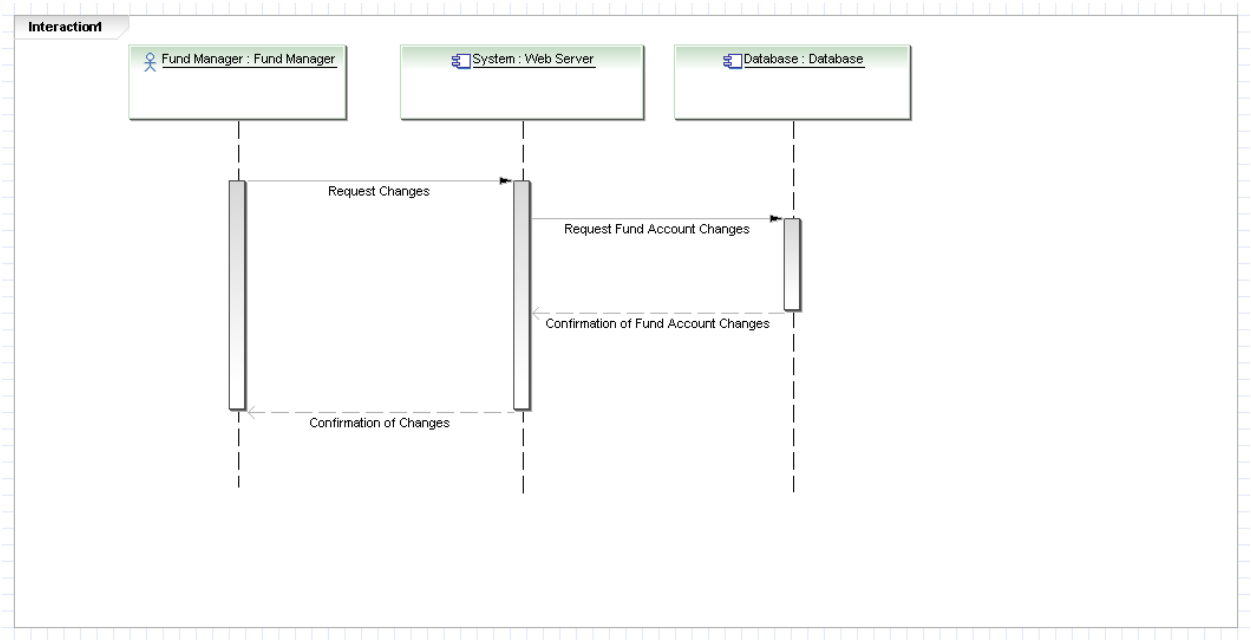


Figure 11: UC-20 Manage Fund

In this sequence diagram, the Fund Manager requests that changes be made to the *Fund* settings, and after the system has verified the changes are valid, it sends a request to the database to implement these changes. After these changes have been made, the database signals to the system that the changes have been made, and the system then signals to the *Fund* Manager that the changes have been made.

For the alternate scenario, if the database detects that there is an error in the changes (for example an invalid value entered in a field), it will return an error to the system, which in turn will give an error back to the investor.

There was no alternative implementation of this use case that was discussed since this one seemed to be the only logical way to do it given the actors that we had.

4 User Interface Specifications

Bears & Bulls benefits greatly from being an app on Facebook since this eliminates the tedious sign up process and buying your first stock is as easy as just installing the app. The entire app workflow was designed with simplicity in mind, and so it was stressed that a user should be able to get from any one screen to another with a minimum of four clicks.

4.1 Preliminary Design

4.1.1 Dashboard

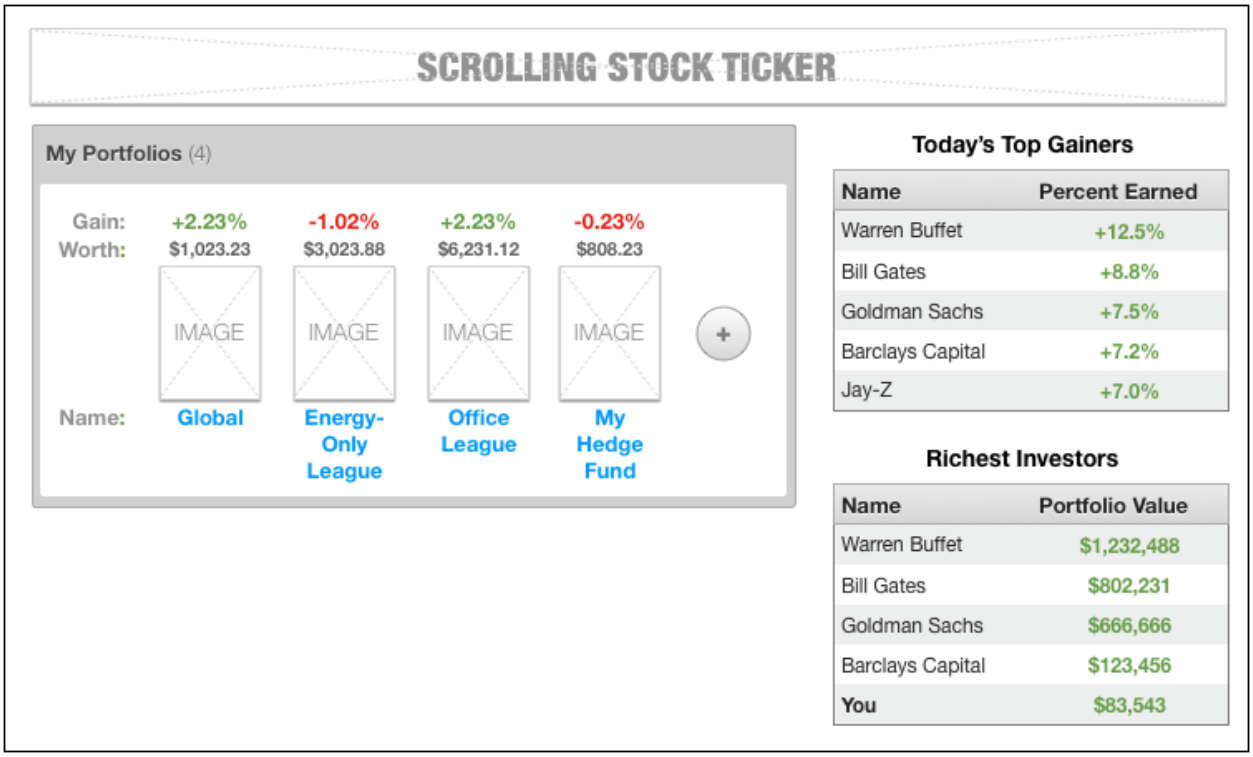


Figure 12: Dashboard

Upon visiting the app, the user is first presented with the dashboard page. If the user is playing for the first time, they will have a default portfolio already set up that is associated with the global *league*. This page provides them with a summary of the performance of their portfolios, a scrolling stock ticker, and the high scores list for the global *league*. From here the user could either click on one of their portfolios and manage it (Figure 15) or add a new portfolio (Figure 13) by clicking on the + button.

4.1.2 Add Portfolio

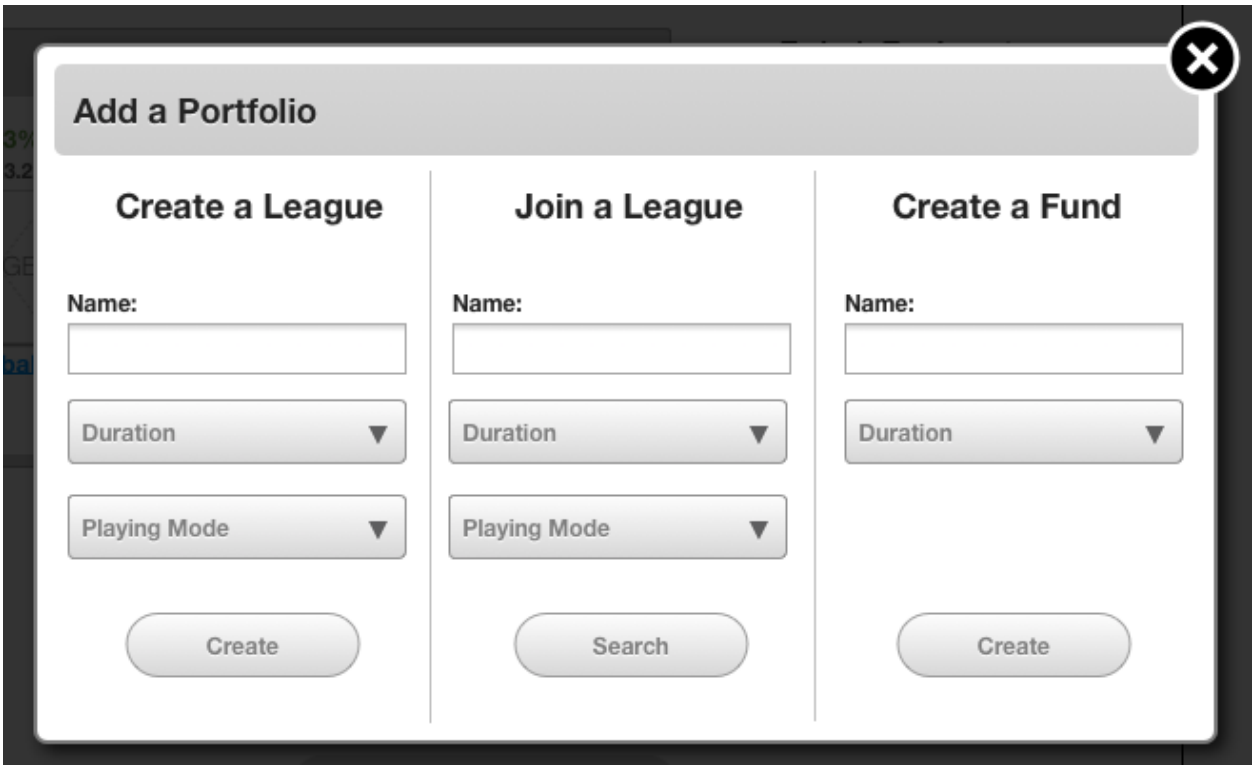


Figure 13: Add Portfolio

When the user clicks on the + button to add a portfolio, a modal dialog box pops up presenting them with one of three ways to add a portfolio. If the user wishes to create a *league*, they are required to input the name of the *league*, the duration of the *league*, and the rules for *league* operation as enumerated by the playing mode dropdown. After entering the parameters and clicking Create, the user is taken to a screen to invite users to that *league* (Figure 15). If the user wishes to join a *league*, they can type in the name of the *league* if they know it or search for open *leagues* accepting players by providing their preferences in terms of duration and playing mode. After clicking Search, the user will randomly be put in a *league* that best suits their preferences.

4.1.3 Invite Users to League

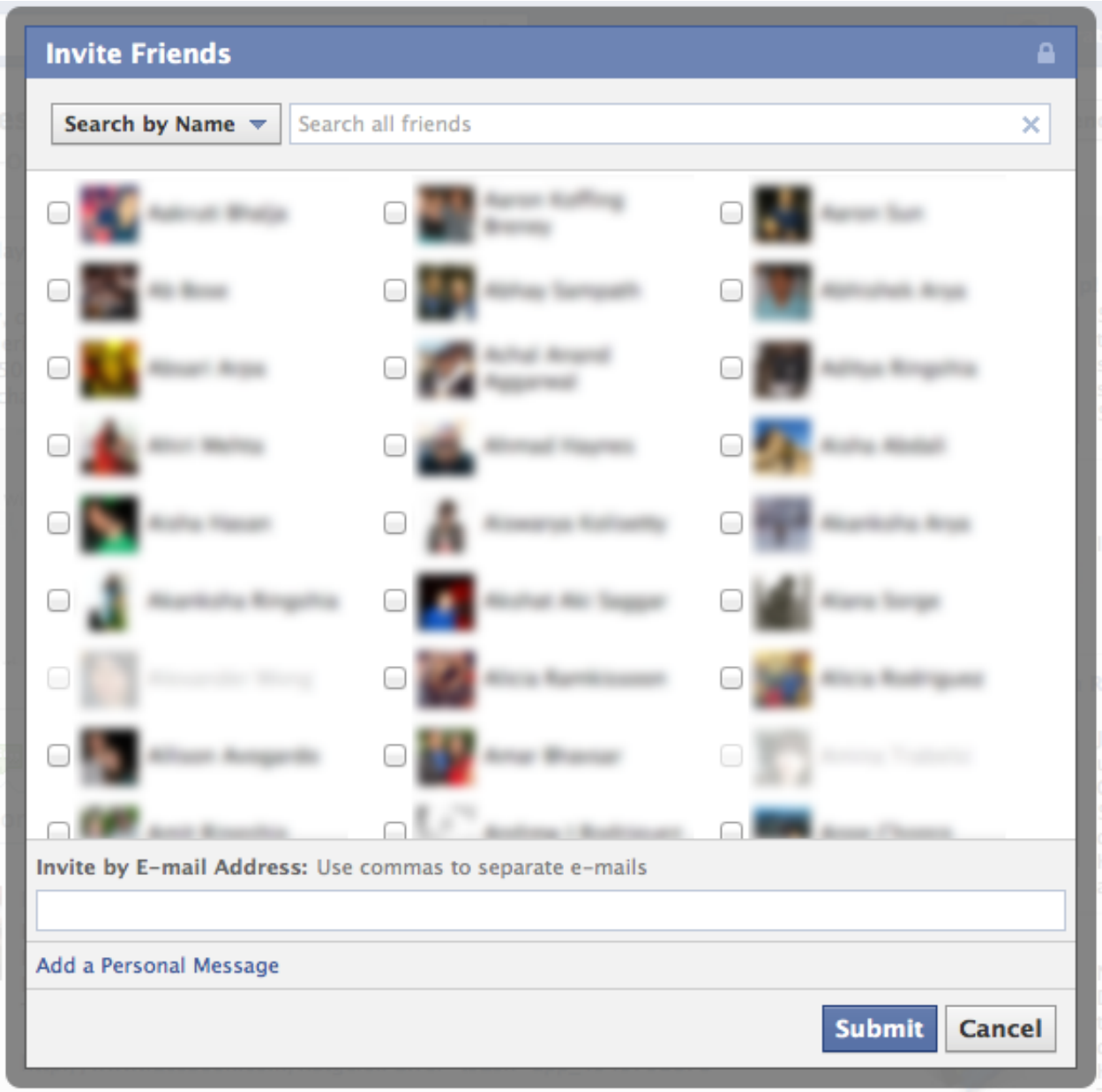


Figure 14: Invite Users to League

Here we benefit from being an app on Facebook by being able to provide the user with a list of all their Facebook friends. The user simply selects the friends that they would like to send an invite to, along with an optional personal message if they wish.

4.1.4 Manage Portfolio

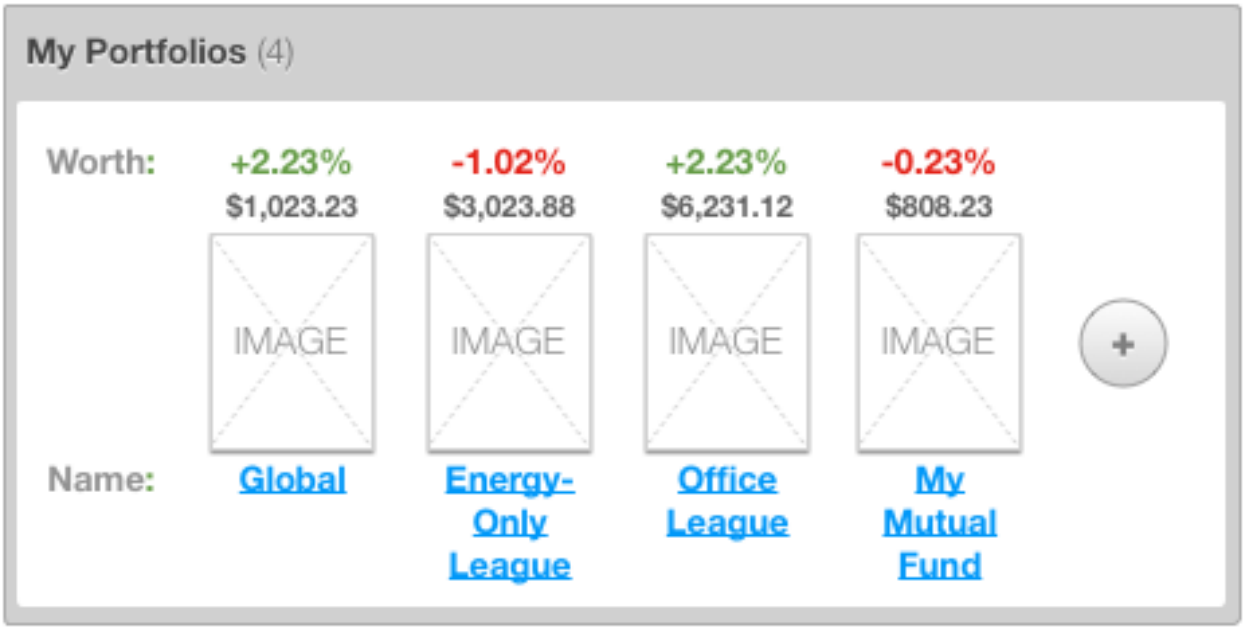


Figure 15: Manage Portfolio

Once a portfolio has been created, the user can populate it with the stocks they wish to purchase. Clicking Buy More brings up the Research Stocks page (Figure 15). They can also view the performance of their existing stocks, the amount of cash they have on hand, and sell any stocks that they wish to by specifying the quantity and clicking on the Sell button. Additionally, to see how they are faring in the *league*, they can click on the *league* name at the top to be taken to the *league* page (Figure 18).

4.1.5 Research Stocks

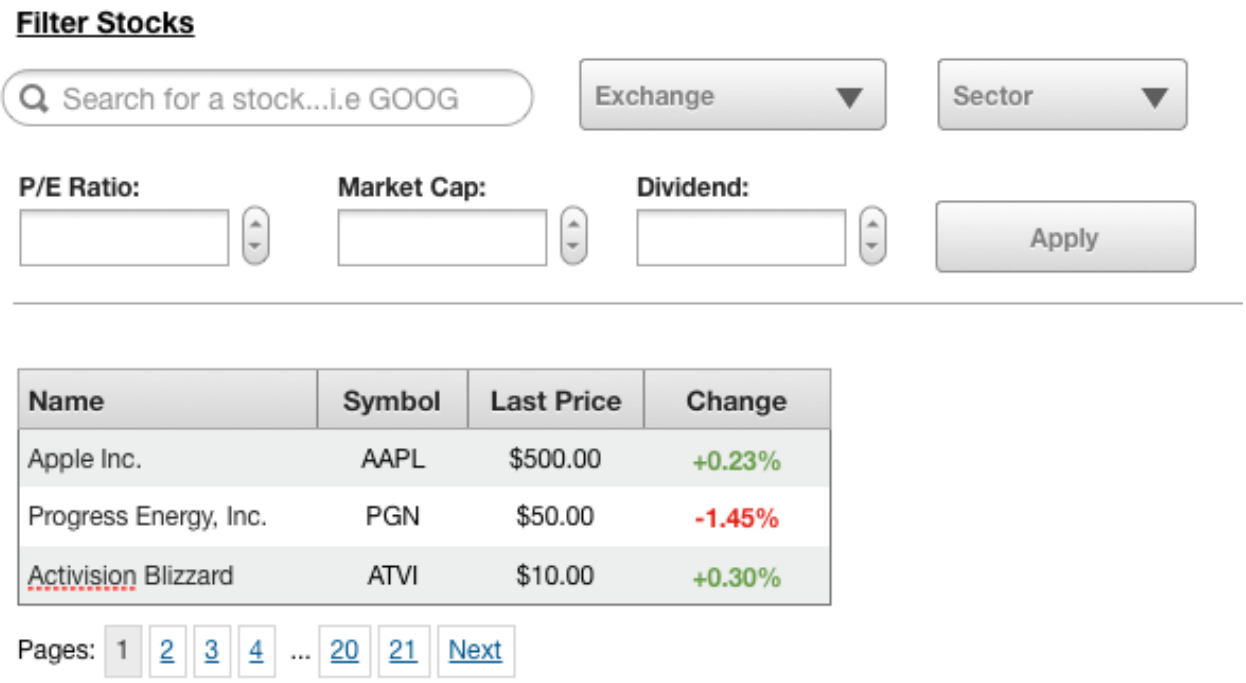


Figure 16: Research Stocks

This page allows the user to search for stocks based on simple parameters, namely the exchange its traded on, which sector it belongs in, the P/E ratio, the market cap, and the percent dividend offered. From here the user can click on the name of the stock they are interested in, which will bring them to the stocks page (Figure 15). If there are too many stocks to be listed at once, a simple pagination mechanism will allow the user to traverse through the list in portions.

4.1.6 Purchase Stocks

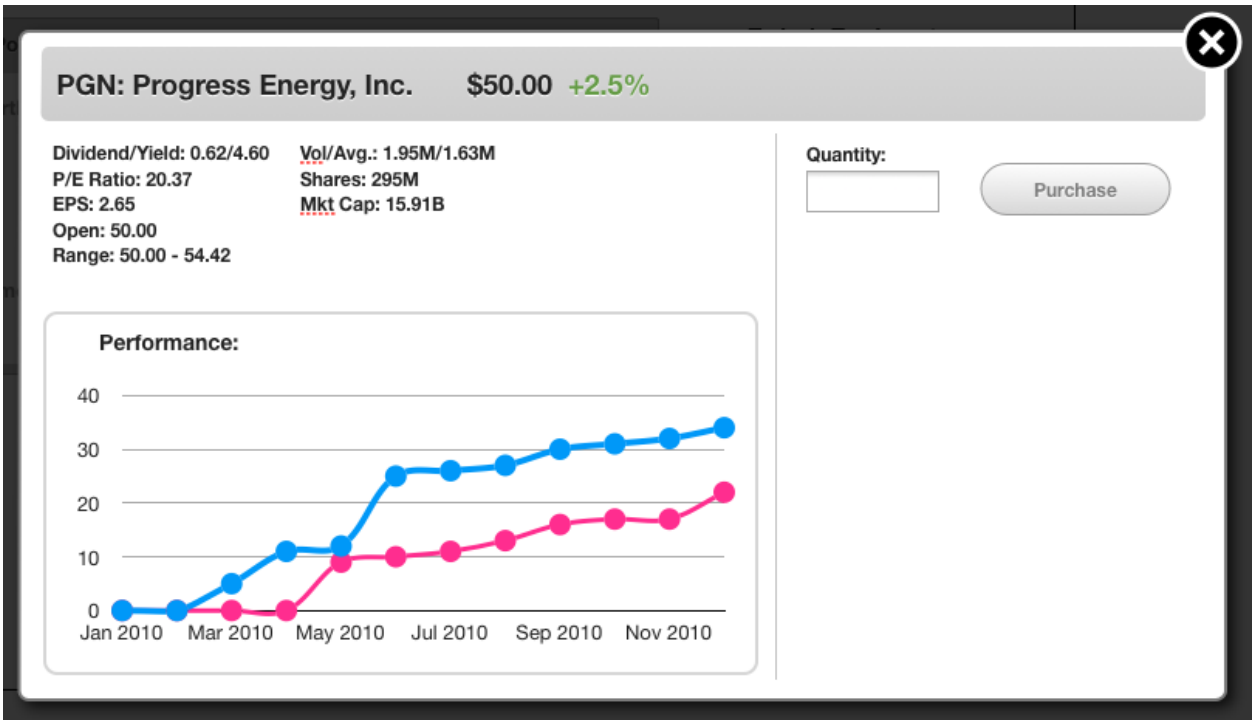


Figure 17: Purchase Stocks

When a stocks name is clicked, the user is presented with this screen that summarizes the stocks performance and lists key stock information. From this popup, the user can place an order to purchase stock.

4.1.7 League Page

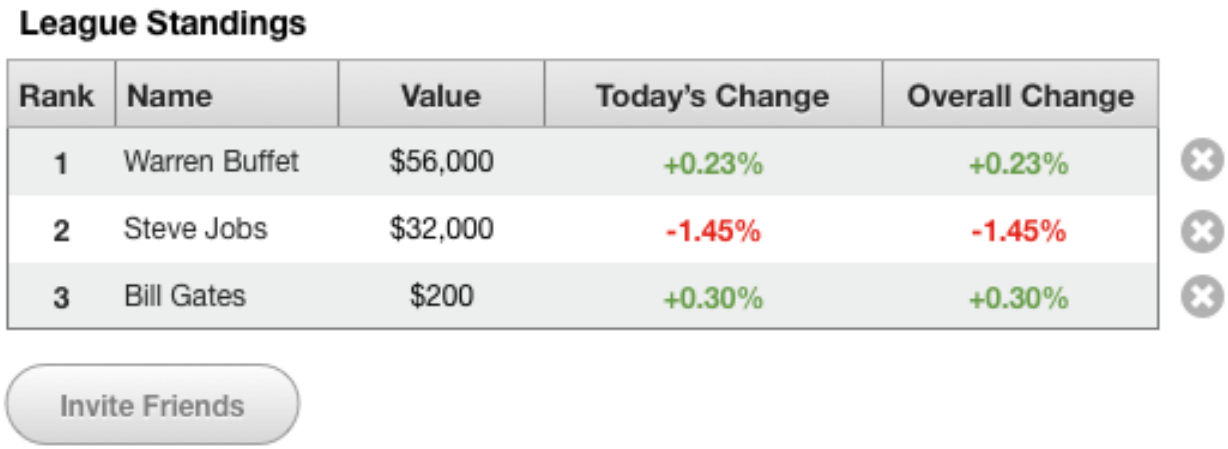


Figure 18: League Page

On this page, a brief summary of the portfolios of all the *league* members is provided. The stocks in competitors portfolio are kept confidential, however the total value is made public. The players are ranked in order of their portfolio value. If the *league* manager is viewing the page, he also has the ability to remove players from the *league* by clicking on the x next to their name. He also has the ability to invite more players to the *league* by clicking on the Invite Friends button, which brings up the Facebook provided popup (Figure 14).

4.2 User Effort Estimation

Usage Scenario	Clicks	Keystrokes
Buy a stock	7	5
Sell a stock	5	1
Create new league	6	14
Create new fund	5	13
Invite user to league	4	10
Remove user from league	4	0

Buy Stock

1. Click on App on Facebook.
2. Click on League.
3. Click on Portfolio.
4. Click on Buy More.
5. Click on the "Search for a stock" text field.
6. Press keys to enter Stock ticker to purchase (ex. AAPL).
7. Press Apply
8. Click on the Stock
9. Press keys to enter number of shares (ex. 15).
10. Click on Purchase

Sell Stock

1. Click on App on Facebook.
2. Click on League.
3. Click on Portfolio.
4. Press keys of "Qty." field next to stock you want to sell (ex. 5 of AAPL)
5. Click on Sell
6. Click on Confirm

Create League

1. Click on App on Facebook.
2. Click on Add Portfolio.
3. Enter name in either Create a League or Join League. (ex. B A R R H A L L L E A G U E)
4. Click in information in the two drop down menus named Duration and Playing Mode.
5. Click on Create.

Create Fund

1. Click on App on Facebook.
2. Click on Add Fund.
3. Enter name in in Create Fund. (ex. Q U A N T U M)
4. Click in information in the one drop down menu.
5. Click on Create.

Invite User to League

1. Click on App on Facebook.
2. Click on League Name
3. Click on Invite Friends.
4. Type in "Search all friends" name of new members (ex. P A R I A S O K A N)
5. Click on Submit

Remove User from League

1. Click on App on Facebook.
2. Click on League Name
3. Click on x next to the player's name.
4. Click on Confirm.

.

5 Domain Analysis

5.1 Domain Model

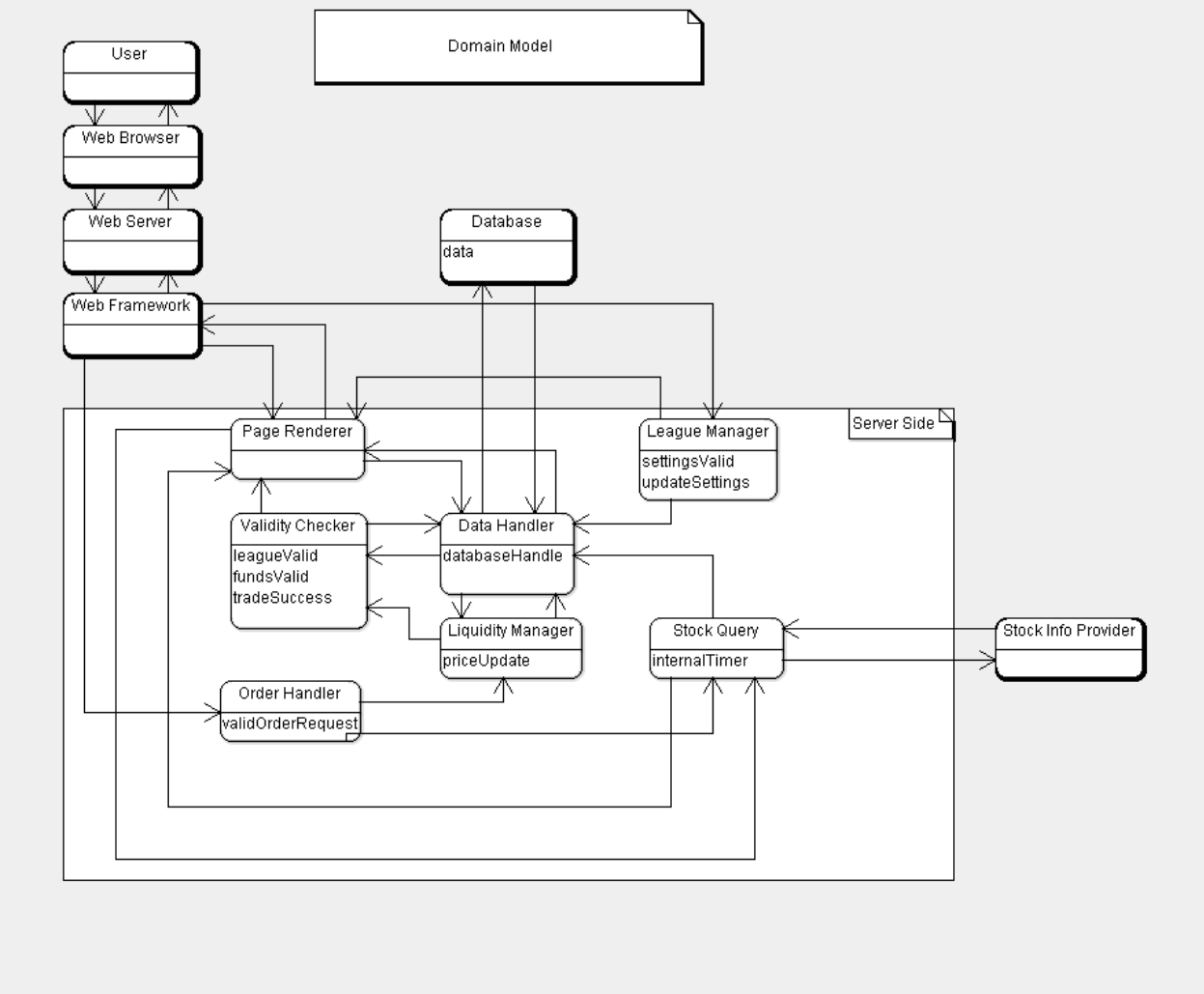


Figure 19: Domain Model

Figure 19 shows Bears & Bulls' general Domain Model as a whole. The subsequent diagrams will give insight into how the concepts work to satisfy the key user cases of the website. Alternate models for the use cases will also be shown. The key difference between the alternate model and the accepted model is the alternate models use of caching to store market data instead of retrieving it when needed. The database would get updated periodically by requesting new information every time interval, for example every minute.

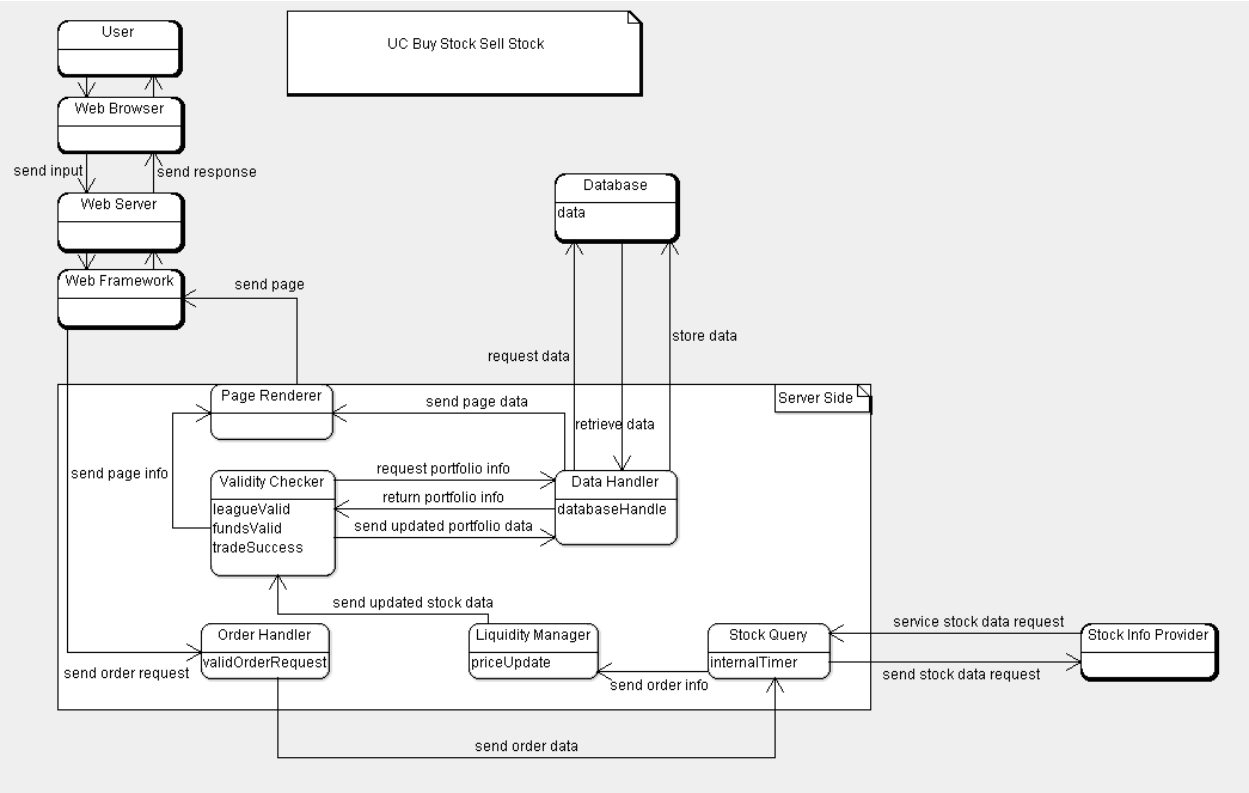


Figure 20: Place Order

Figure 20 will represent UC-1 and UC-2. It represents both our buy and sell use cases since they behave in the same way. The User's order information eventually makes its way to the Web Framework, which passes the data to the Order Handler. It then relays the data to the Stock Query, which will fetch a price from Stock Info Provider based on what stock is ordered. Stock Query will send this price and the rest of the order data to the Liquidity Manager to adjust the price based on an algorithm. This updated order data then travels to the Validity Checker so the trade can be deemed valid. It requirs the user's portfolio data for this, so it sends the User and *league* ID to the data handler along with a request for portfolio data about funds and *league* settings. Once the trade is judged as valid or not, it will send updated portfolio info to be stored to the data handler if needed and pass the necessary info to the Page Renderer to display a page showing the success and result of the trade. This rendered page is sent to the Web Framework, to be shown to the User.

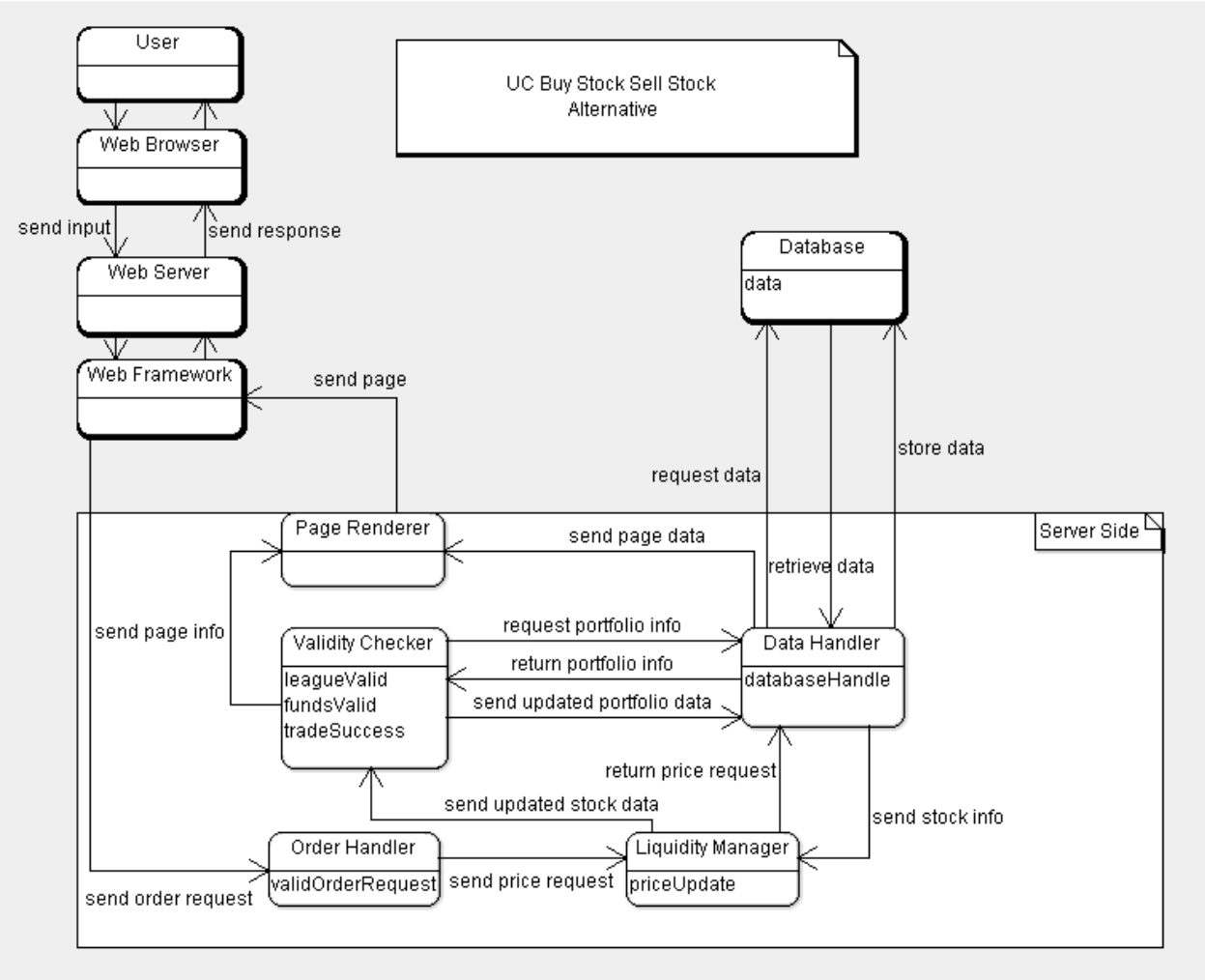


Figure 21: Place Order Alternate

The alternate case for placing an order, differing in the ways described at the start of the section is shown in Figure 21.

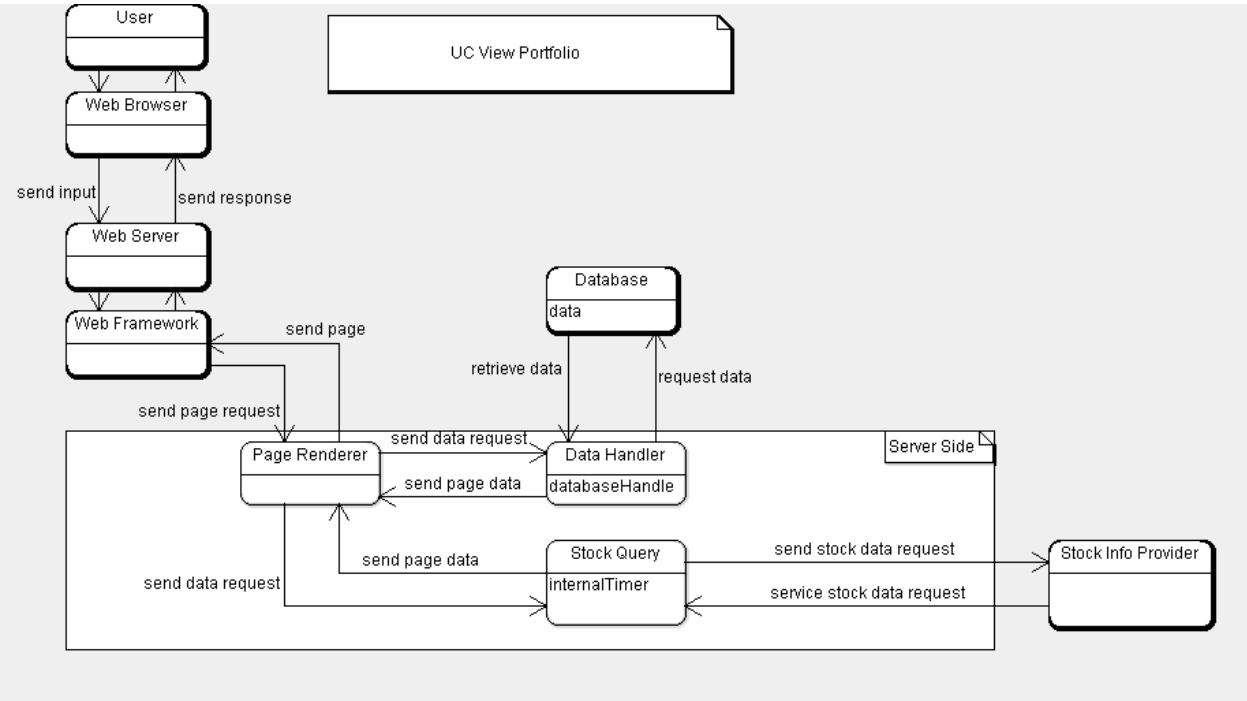


Figure 22: View Portfolio

Figure 22 shows the UC-5 View Portfolio. The User's query about a portfolio gets sent down to the Web Framework which in turn will request a page to be rendered by the Page Renderer. To get its necessary data, the Page Renderer will send a request for updated stock prices to the Stock Query, and a request for the portfolio info to the Data Handler. The Stock Query will retrieve the data from the Stock Info Provider, and the Data Handler will get its data from the Database. One they have collected the data, they both return it to the Page Renderer, which will generate the page for the Web Framework to send to the User for viewing.

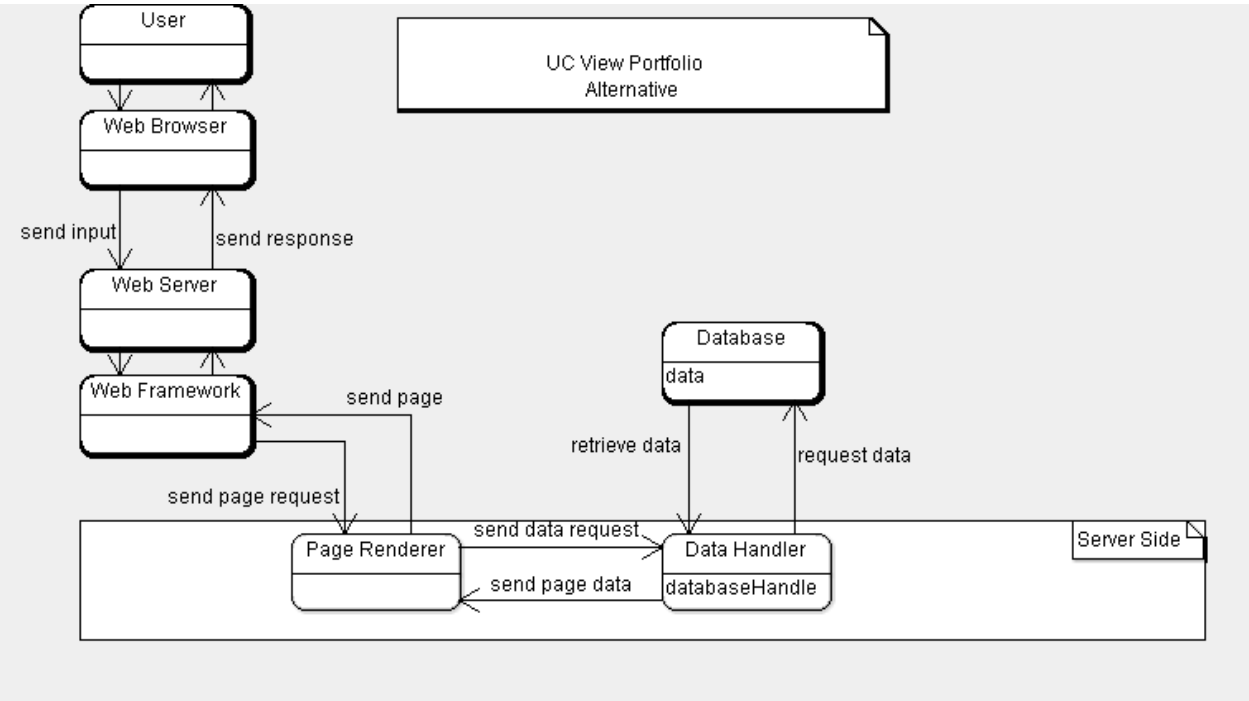


Figure 23: View Portfolio Alternate

The alternate case is shown in Figure 23, using the Database instead of the Stock Info Provider.

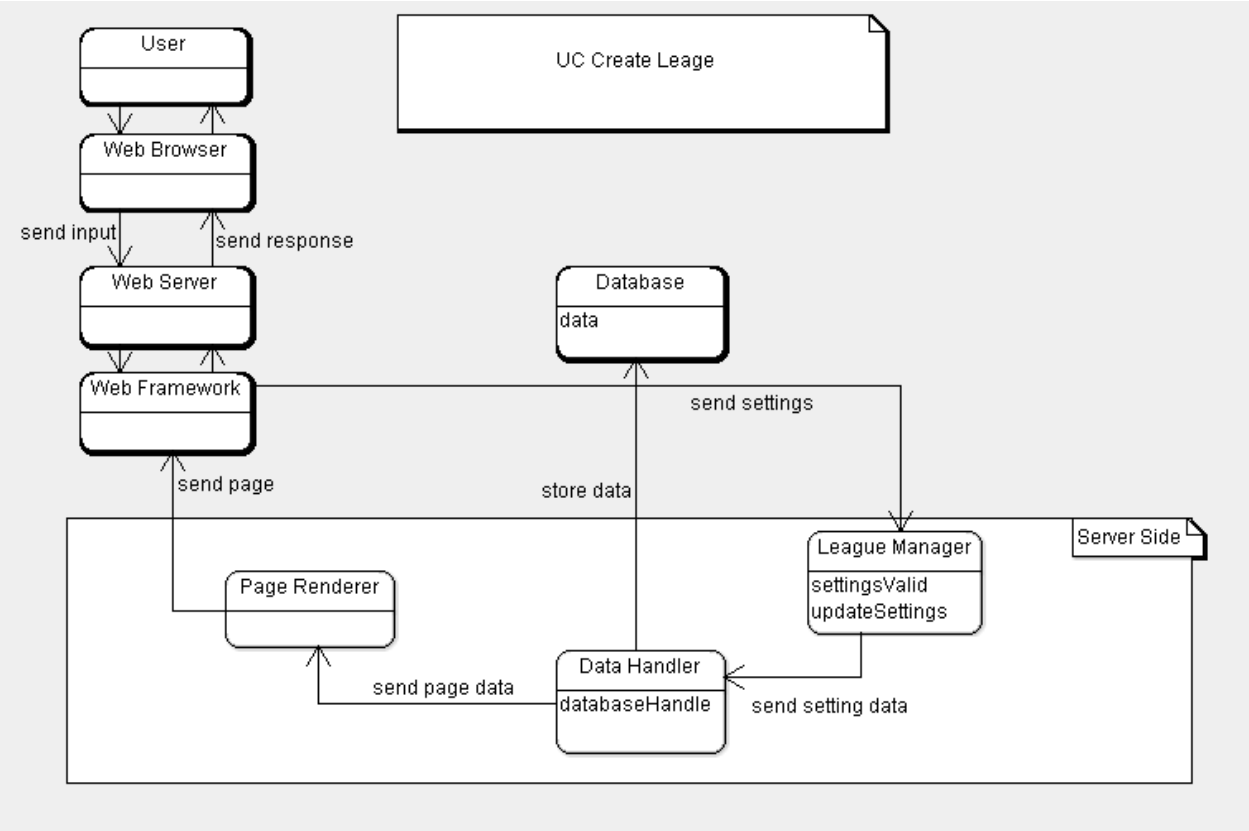


Figure 24: Create League

UC-8 Create League is represented shown in Figure 24. Note that this model will also essentially cover creating *Funds*, as well as maintaining both *leagues* and *Funds*. The only thing difference is what the User would have to input for settings. The Web Framework eventually receives the User's desired initial or modified settings and sends it to the League Manager. The League Manager will generate new data for the *league* or *Fund*, and send this data along with the settings info to the Data Handler to be stored within the Database. The Data Handler will then pass on necessary data for the Page Renderer to create a page. Once rendered, it is passed to the Web Framework to be shown to the User.

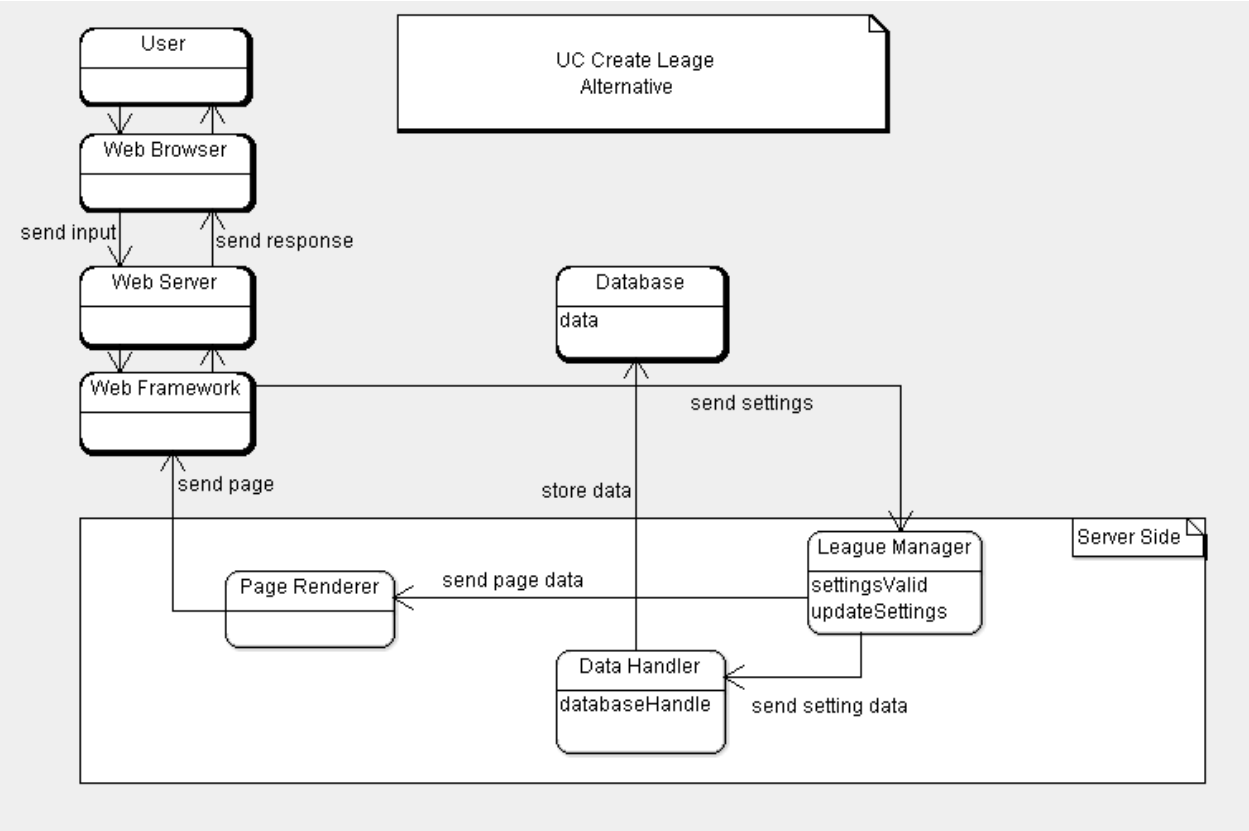


Figure 25: Create League Alternate

The alternate case involves the League Manager sending necessary page data to the Page Renderer rather than the Data Handler, as shown in 25.

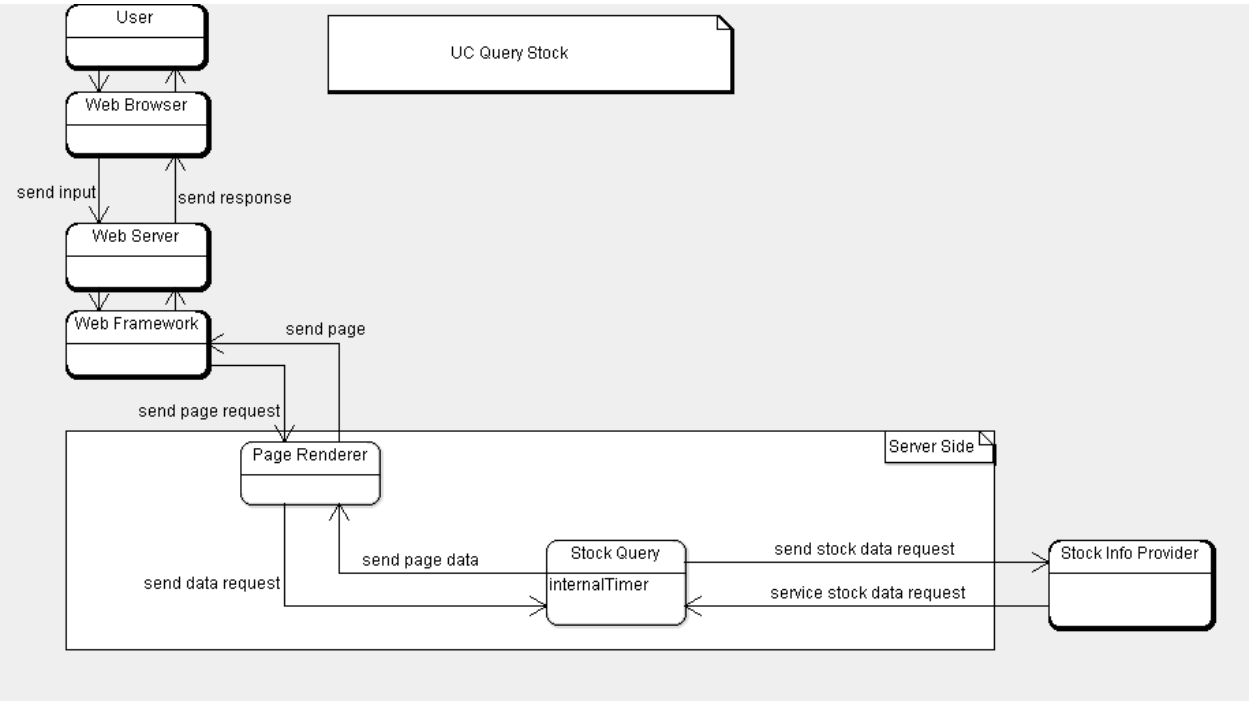


Figure 26: Query Stocks

Figure 26 shows the UC-3 Query Stocks. The requested stock is sent through to the Web Framework and handed off to the Page Renderer. This concept then requests the data for said stock from the Stock Query, which fetches the information from the Stock Info Provider. The Stock Query will return the required data to the Page Renderer which will create its page for the Web Framework to send to the User for viewing.

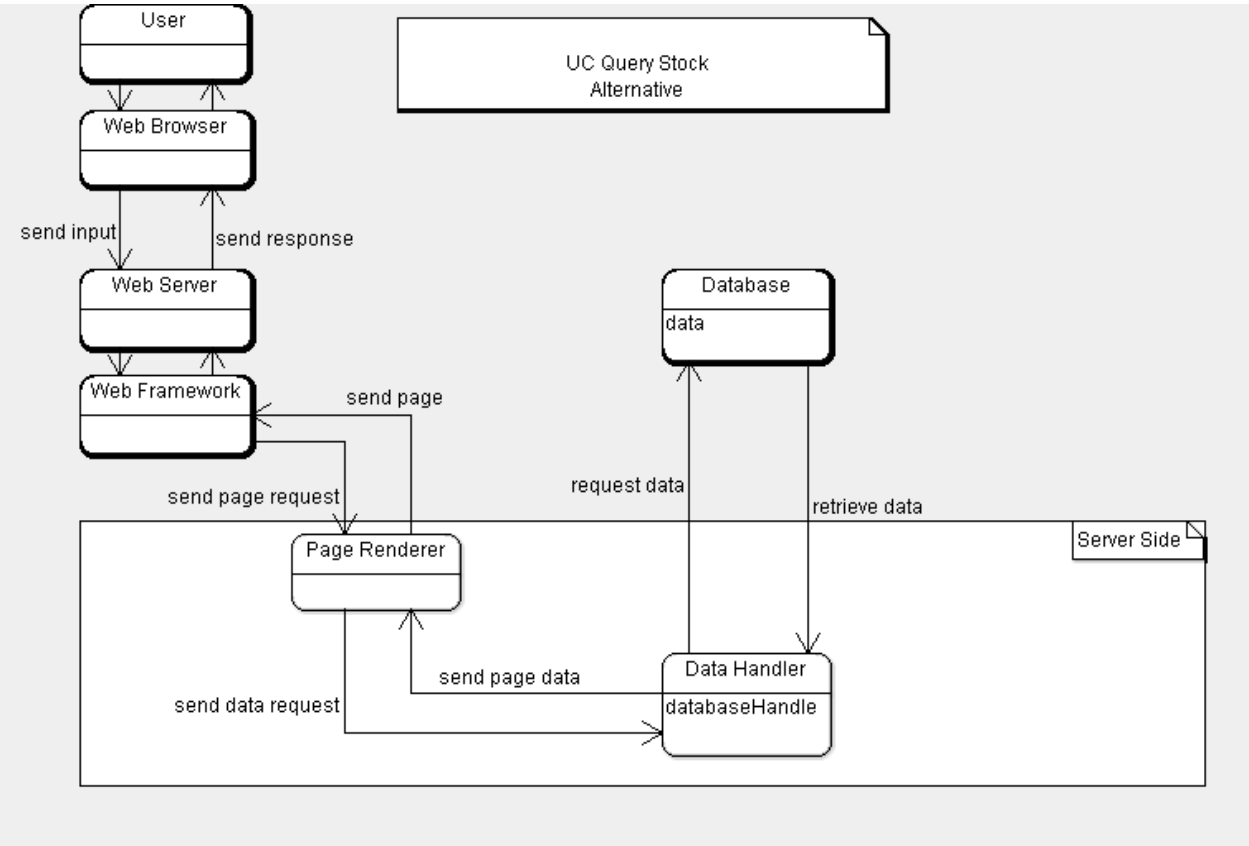


Figure 27: Query Stocks Alternate

The alternate case, using the Database instead of the Stock Info Provider, is shown in 27.

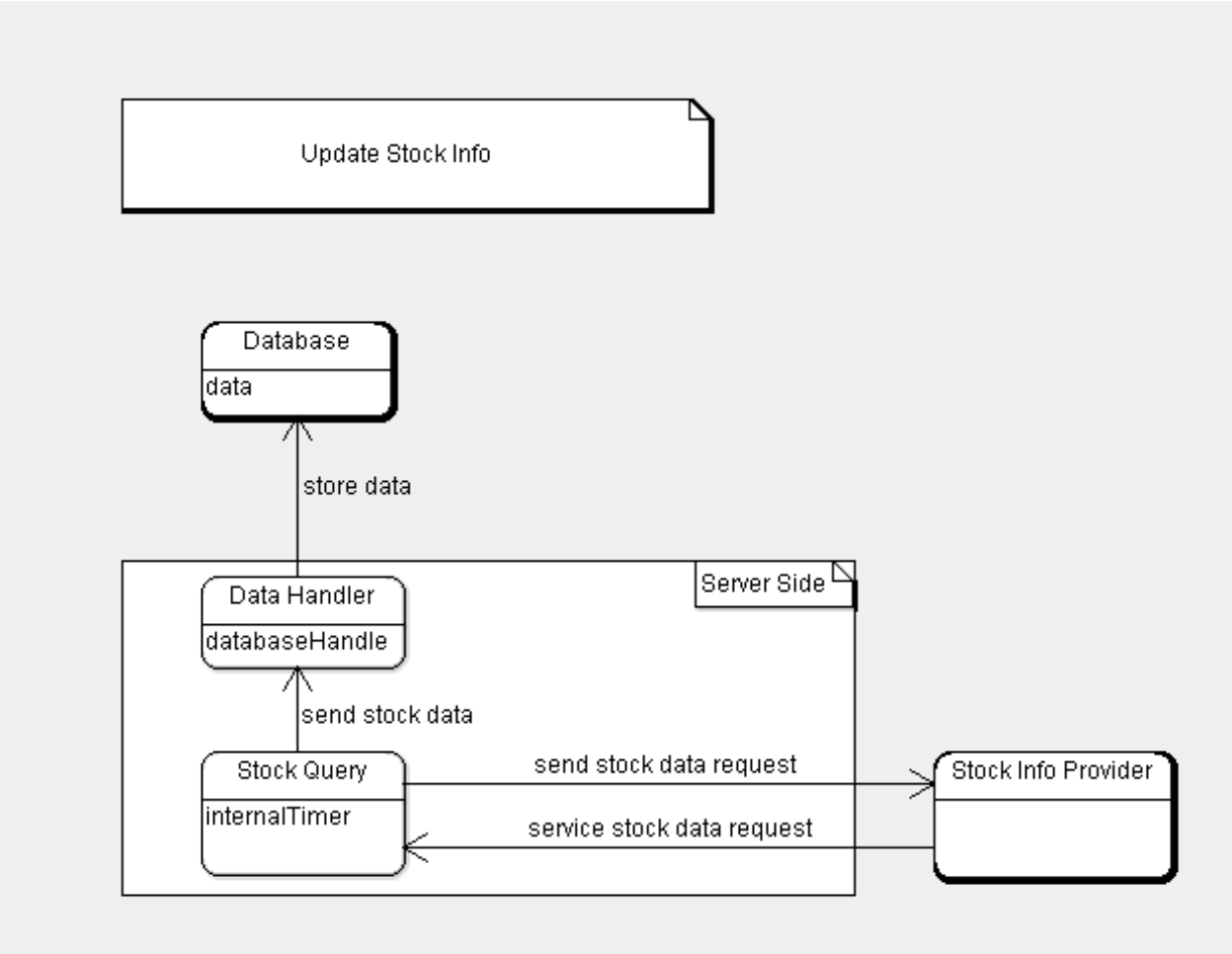


Figure 28: Updating Stock Info

The last diagram (Figure 28) shows Updating Stock Info, a required process if the alternative domain models are to be used. It involves the Stock Query being signaled by an internal timer to request all stock info from the Stock Info Provider, which it will in turn send to the Data Handler to store in the Databse.

5.1.1 Concept Definitions

The definition of our concepts are as follows:

User

Definition: A player playing Bears & Bulls.
Responsibilities:

- Manage portfolio
- Make requests for trades
- Manage *leagues*
- Navigate through website

Web Browser

Definition: The user's browser which runs from the user's device.
Responsibilities:

- Take requests from the user
- Send requests to the Web Server
- Get responses from the Web Server
- Display the response from the Web Server

Web Server

Definition: HTTP web server, running on a web host
Responsibilities:

- Receive requests from Web Browser
- Send requests to Web Framework
- Get responses from Web Framework
- Send responses to the Web Browser

Web Framework

Definition: APIs to help display user-friendly output
Responsibilities:

- Receive requests from Web Server
- Sends request to appropriate handler: application or database
- Receive rendered pages in the form of structured data
- Send responses to the Web Server

Page Renderer

Definition: Takes user requests and creates a page which is user-friendly
Responsibilities:

- Determine the information required to be rendered and request it
- Receive the required information
- Convert the information into user-friendly format
- Send rendered pages to the Web Framework

Order Handler

Definition: Application conducting transactions of stocks
Responsibilities:

- Receive requests from Web Framework
- Determine what the request is and readies for manipulation
- Request updated price info
- Transmit necessary information to other concepts

Stock Query

Definition: Fetch real-time stock prices
Responsibilities:

- Receive requests for stock price
- Request information from Stock Info Provider
- Retrieve information from Stock Info Provider
- Send real-time stock prices to be stored for application's use

Validity Checker

Definition: Checks if a trade is valid
Responsibilities:

- Receive updated order information
- Request and receive portfolio data
- Determine if sufficient funds are available for the transaction
- Determine if trade is allowed for given user and portfolio based on *league* or *Fund* settings
- Send updated portfolio information if necessary
- Send data reflecting successful/unsuccessful trade to be redered

Liquidity Manager

Definition: Manipulates price to realistic real world prices for slippage
Responsibilities:

- Receive stock and order data
- Utilize algorithm to reflect realistic trades in the market
- Determine new price
- Send out updated stock information

Data Handler

Definition: Communicates with Database to service data requests
Responsibilities:

- Receive and send every kind of data used in system
- Request data from Database
- Send data to be stored in Database

League Manager

Definition: Can create and upkeep *leagues* and *Funds*
Responsibilities:

- Receive initial or modified settings input for desired *league* or *Fund*
- Pass *league* or *Fund* data to be stored
- Pass *league* or *Fund* data for rendering of a page

5.1.2 Association Definitions

The following association definitions are provided for the domain models that model not only for the important use cases, but also any alternative models for said use cases:

Concept Pair	**Association Description**	**Association Name**
Web Browser ↔ Web Server	User interacts with browser	send input, send response
Web Framework ↔ Order Handler	Passes volume, trade type, User ID and *league* ID	send order request
Web Framework ↔ Page Renderer	Request to visit page, sends rendered page in form of data	send page request, send page
Web Framework ↔ League Manager	Passes the user's desired settings	send settings
Page Renderer ↔ Data Handler	Requests data to correctly render page, passes necessary data	send data request, send page data
Page Renderer ↔ Stock Query	Asks for data on specific stocks, send data on specific stocks	request stock data, send stock data
Page Renderer ↔ League Manager	Sends the required data to render page	send page data

Page Renderer ↔ Validity Checker	Passes necessary data for the page to be rendered	send page info
Order Handler ↔ Liquidity Manager	Sends order information to reflect real-life prices	send price request
Order Handler ↔ Stock Query	Passes necessary order data	send order data
Stock Query ↔ Stock Info Provider	Asks for stock data, return stock data	send stock data request, service stock data request
Stock Query ↔ Data Handler	Sends stock data to be stored	send stock data
Stock Query ↔ Liquidity Manager	Passes updated order information	send order info
Validity Checker ↔ Data Handler	Asks for user's portfolio information for validity purposes, passes user's portfolio information, passes updated portfolio information following trade	request portfolio data, return portfolio data, send new portfolio data
Validity Checker ↔ Liquidity Manager	Sends updated stock data to be checked	send updated stock data
Liquidity Manager ↔ Data Handler	Sends order information, returns new price	send stock info, return price request
Data Handler ↔ Database	Stores incoming data, request certain data, retrieve needed data	store data, request data, retrieve data
Data Handler ↔ League Manager	Sends the settings data to be stored	send settings data

5.1.3 Attribute Definitions

Most of our concepts do not need to hold their own data, as our website is dynamic and web-based. We also have not yet made the decision to cache data. Thus, nearly all data is stored in a single database. The sparse attributes that must be accounted for are as follows:

Concept	**Attribute**	**Meaning**
Data Handler	databaseHandle	Interacts with the database.
Database	data	Stores data for future use. Includes all data used in the system, including League ID, User ID, stock volume and price data, league settings, fund settings, and portfolio data such as transaction history.
Facebook	accountInformation	We don't need to keep detailed account of user information as Facebook has already done it for us. Also we don't need to create new login information as that is handled by Facebook.
Stock Query	internalTimer	Necessary for an alternate domain model where the Database is refreshed with all stock information periodically.
Page Renderer	sufficientRenderData	Determines if the required data to render the page is there.
Order Hander	validOrderRequest	Checks to see if there is all the required data for an order.
Liquidity Manager	priceUpdate	Generates a new price value of the ordered stock.
League Manager	settingsValid, updateSettings	Determines if the User's settings input are valid. Will also upkeep settings based on User modification, signaling changes to be made to Database.

Validity Checker	leagueValid, fundsValid, tradeSuccess	Compares funds and price and checks league settings to make sure a transaction is valid. Determines if trade is a success.

5.1.4 Traceability Matrix

Use Case	PW	User	Web Browser	Web Server	Web Framework	Page Renderer	Order Handler	Stock Query	Avilability Find	Liquidity Manager	Data Handler
UC01-02	15	x	x	x	x	x	x	x	x	x	x
UC03	14	x	x	x	x	x		x			
UC05	17	x	x	x	x	x		x			x
UC08	4	x	x	x	x	x					x

5.2 System Operation Contracts

Register User

Preconditions:

- None

Postconditions:

- User has a portfolio associated with the league.
- User's name, portfolio holdings and other information are stored in the database.

Buy Stocks

Preconditions:

- Investor's intended transaction is less than available cash balances.
- Stock provider has the stocks available for purchase.
- Transaction is valid under league settings.
- Transaction data is valid.

Postconditions:

- Update database with user's new stock holdings.

Sell Stocks

Preconditions:

- Investor has the assets he is attempting to sell.
- Transaction is valid under league settings.
- Transaction data is valid.

Postconditions:

- Investor's portfolio is adjusted in database to reflect transaction.
- Stock inventory is updated in database.

Query Stocks

Preconditions:

- Stock exists

Postconditions:

- None

View Portfolio

Preconditions:

- Initiating investor owns the portfolio.

Postconditions:

- None.

Create Portfolio

Preconditions:

- User has permission to create a portfolio, as dictated by league coordinator
- Input data is valid

Postconditions:

- User's portfolio reflects membership in the league
- Portfolio information stored in the database

Create Fund

Preconditions:

- Input settings are valid

Postconditions:

- New fund's information stored in the database

Create League

Preconditions:

- Input settings are valid

Postconditions:

- League information stored in the database

Invite to League

Preconditions:

- User has permission to join a league
- Valid invitee User ID and League ID

Postconditions:

- Invite information sent to Manage League for further interaction

Manage League

Preconditions:

- User has league coordinator privileges
- Input settings are valid

Postconditions:

- League information is up to date and is reflected in the database

Manage Fund

Preconditions:

- Fund manager initiates a erquest
- Input settings are valid

Postconditions:

- Any updated information is updated in the database

View Account Information

Preconditions:

- User is logged in

Postconditions:

- None

View League Standings

Preconditions:

- User has access privileges to the league
- League exists

Postconditions:

- None

5.3 Mathematical Model

Bears & Bulls' sole mathematical model is the model used to calculate price slippage when executing block trades. Slippage is usually associated with large equity *Funds* and institutional *investors* [12] since their actions tend to flood the exchange with an abundance of buy or sell orders. This puts pressure on the price of the security to move up in the case of a buy or to move down in the case of a sell. Slippage is usually not an issue for highly liquid markets with low volatility. Traders buying and selling blue chip stocks would therefore experience very little slippage, even when the volume is very high. On the other hand, a trader buying huge volumes of penny stocks can easily cause price movements through his actions alone. Thus Bears & Bulls provides a mathematical model for estimating price slippage.

The two factors that determine slippage are volatility and liquidity. High volatility by definition implies high price swings and so more slippage. Highly liquid markets have many buyers and sellers and so a large trade can be made without affecting price to the same extent as in an illiquid market. Let v represent the volatility of a security, l represent its liquidity, and p the average price. It is clear the that average price of a trade should be directly related to v and inversely related to l. Let us examine what would happen to the average price an *investor* pays if he were to buy a large block of shares.

In a completely involatile market ($v = 0$), the trader would experience no slippage based on his trading action since no volatility implies no price movement. Likewise, in a perfectly liquid market ($l = \infty$), there is always a willing counterparty for the trade at the given price and quantity. Let s be the current ask price for security and z be the current ask size. From the above relation, we can see that

$$p = (1 + \frac{v}{l}) \times s$$

satisfies the conditions that there is no price movement from the current ask price. The equation also preserves the relationship between p, v and l. Let $N = 1 + \frac{v}{l}$ and assume that it is greater than 1. If the block size the buyer wishes to buy is less than the ask size, then the buyer only needs to buy from that seller to fill his order. Thus the price would not deviate from the seller's ask price. If the buyer wants to buy more than the current ask size, he must buy from additional seller to complete his order. A simplifying assumption will be made that the next seller sells the same block size as the previous seller. The price per share for the next seller is $N \times s$. Continuing this pattern, the n^{th} block will sell at $N^{n-1} \times s$. The last block may not be filled as the buyer may not want to buy in multiples of the current ask size. Thus if we let b be the total number of shares the buyer wishes to buy and $n = \lfloor \frac{b}{z} \rfloor$ be the number of whole blocks bought, then the total price of paid by the *investor* is:

$$p_{total} = \Big[\sum_{i=0}^{n-1} (1 + \frac{v}{l})^i \times s \times z \Big] + (1 + \frac{v}{l})^n (b - (n-1) \times z)$$

Consider the example where an *investor* wishes to purchase 1000 shares of XYZ and the current ask is \$110.00 $\times$ 300 shares and $N = 1 + \frac{v}{l} = 1.01$. In this case, $s = 110.00, z = 300, b = 1000, n = 3$. The first block of 300 is sold at \$110.00. The second block of 300 is sold at \$111.10, the third block is sold at \$112.21 and the final 100 shares is sold at \$113.33. The total price paid is \$100106.33, or an average price per share of \$111.33. This is a 1.2% change from the ask price.

Now it is necessary to determine v and l. Many mathematical models have been dedicated to predicting volatility and liquidity and there is still nothing that can accurately predict either of them. That said, there are ways of qualitatively measuring volatility and liquidity that will suit our needs.

Volatility can be measured by the stock β. β is the correlation between a stock's movement relative to the movement of the market as a whole. Consider plotting the percentage moves of the market versus the changes in price of a stock. A β of 1 would mean that every percentage move in the market should result the same percentage move in the stock price. Higher β generally implies higher volatility. Liquidity can be measured by the bid-ask spread of a security. Highly liquid assets, such as currencies, usually have bid-ask spreads of a few hundreths of a percent. Less liquid assets such as mid-cap stocks, have bid-ask spreads of one or two percent of their price. For our model, we will use $v = \frac{\beta}{100}$ to represent the volatility of a stock. Let r be the bid-ask spread and s the last price of the stock. Liquidity l will be defined as $l = \frac{s}{r}$. Thus a spread of 0 would mean infinite liquidity. N would then be defined as $N = \Big(1 + \frac{\beta r}{100s}\Big)$.

6 Plan of Work

Contribution Breakdown

Responsibilities Matrix	William	Aaron	Pratik	Dean	Omar	Noah
Proposal	x					
First Report						
Requirements	x	x	x	x	x	x
Use Cases	x	x		x		
Domain Model				x	x	x
User Interface			x		x	
Plan of Action	x					
Second Report						
Class Diagram and Interface Specification	x		x			
System Architecture and Design	x	x	x	x	x	x
Algorithms and Data Structures	x	x				
User Interface Design an Implementation			x	x	x	x
Progress Report and Plan of Work	x					
Report Editing	x	x	x	x	x	x
First Demo						
Database	x	x				
Integration to Facebook			x	x	x	x
Basic User Interface	x	x	x	x	x	x
Connection to Database	x	x	x			
Testing	x	x	x	x	x	x
Third Report						
Collation of Reports	x	x	x			
Update Report	x	x	x	x	x	x
Second Demo						
Implement Rest of Use Cases	x	x	x	x	x	x
Testing/Enhancing	x	x	x	x	x	x

GANTT project
January 2012
February 2012
March 2012
April 2012
May 2012
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
10
11
12
13
14
15
16
17
18
19
20
21
22
Proposal
First Report
Second Report
Class Diagram and Interface Specification
System Architecture and System Design
Algorithms and Data Structures
User Interface Design and Implementation
Progress Report and Plan of Work
Report Editing
First Demo
Database
Integration to Facebook
Basic User Interface
Connection to Database
Testing
Third Report
Coallate First and Second Report
Update Report
Second Demo
Implement Rest of Use Cases
Testing/Enhancing
[4 Day(s)]
[1/23/12 - 1/27/12]
[15 Day(s)]
[1/27/12 - 2/17/12]
[15 Day(s)]
[2/17/12 - 3/9/12]
[4 Day(s)]
[2/17/12 - 2/23/12]
[7 Day(s)]
[2/23/12 - 3/3/12]
[3 Day(s)]
[3/1/12 - 3/6/12]
[5 Day(s)]
[2/28/12 - 3/6/12]
[1 Day(s)]
[3/6/12 - 3/7/12]
[2 Day(s)]
[3/7/12 - 3/9/12]
[27 Day(s)]
[2/17/12 - 3/27/12]
[24 Day(s)]
[2/17/12 - 3/22/12]
[6 Day(s)]
[2/17/12 - 2/25/12]
[18 Day(s)]
[2/24/12 - 3/21/12]
[2 Day(s)]
[3/22/12 - 3/24/12]
[2 Day(s)]
[3/23/12 - 3/27/12]
[23 Day(s)]
[3/27/12 - 4/27/12]
[6 Day(s)]
[3/27/12 - 4/4/12]
[17 Day(s)]
[4/4/12 - 4/27/12]
[28 Day(s)]
[3/27/12 - 5/4/12]
[18 Day(s)]
[3/27/12 - 4/20/12]
[10 Day(s)]
[4/20/12 - 5/4/12]

[5]

The preceding table is the responsibility matrix for this group. In general, Aaron and William will do most of the compiling of the reports, while everyone also works on writing the reports. Aaron and Pratik will lay down the class diagrams and interface specifications as a groundwork for the rest of the system. The system architecture and design will be a collaborative effort between all members in the group, while algorithms and data structures will mostly be determined by Aaron and William. Pratik is the most experienced in implementing user interfaces, thus he will take the lead on user interface design and will work closely with Noah, Omar, and Dean. Finally, the editing of the reports it will be a collaborative effort too.

The general scheme for the rest of the project will be that Aaron and William will implement the database, while the rest of the group will work on the interface. From there, William, Aaron, and Pratik will make the connections between the various parts. If any group member has finished his delegated tasks, he will jump around to wherever he is needed.

7 Appendix

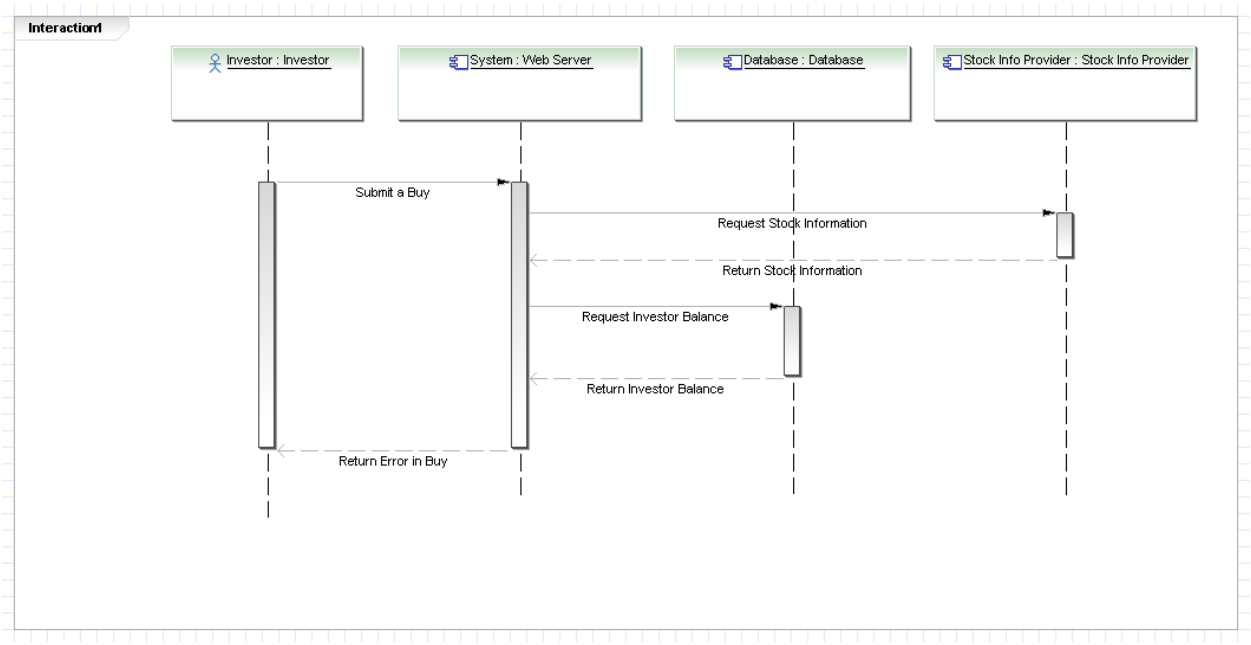


Figure 29: Alternative Sequence Diagram for Buy Stocks

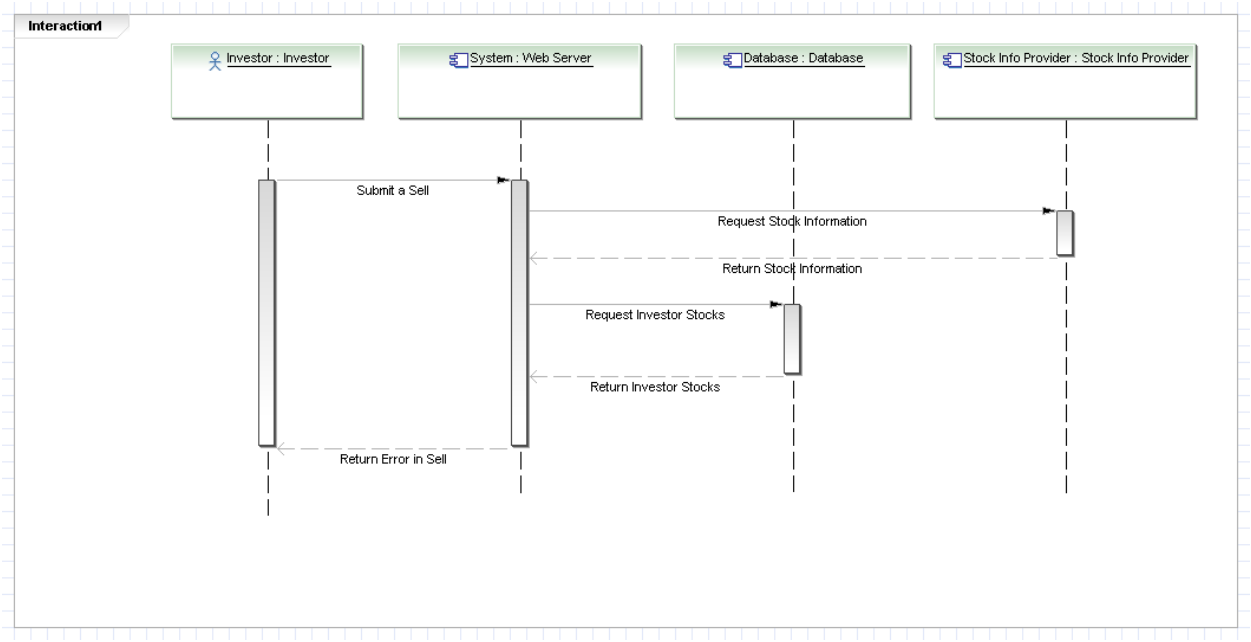


Figure 30: Alternative Sequence Diagram for Sell Stocks

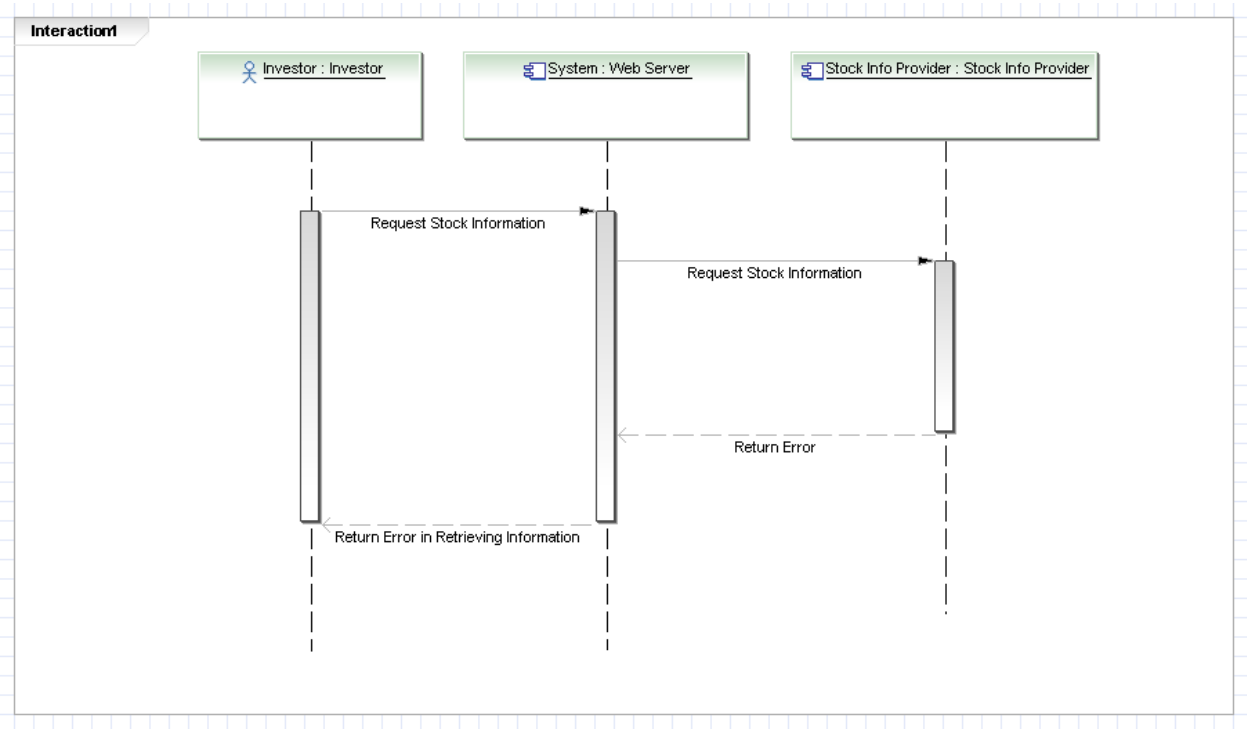


Figure 31: Alternative Sequence Diagram for Query Stocks

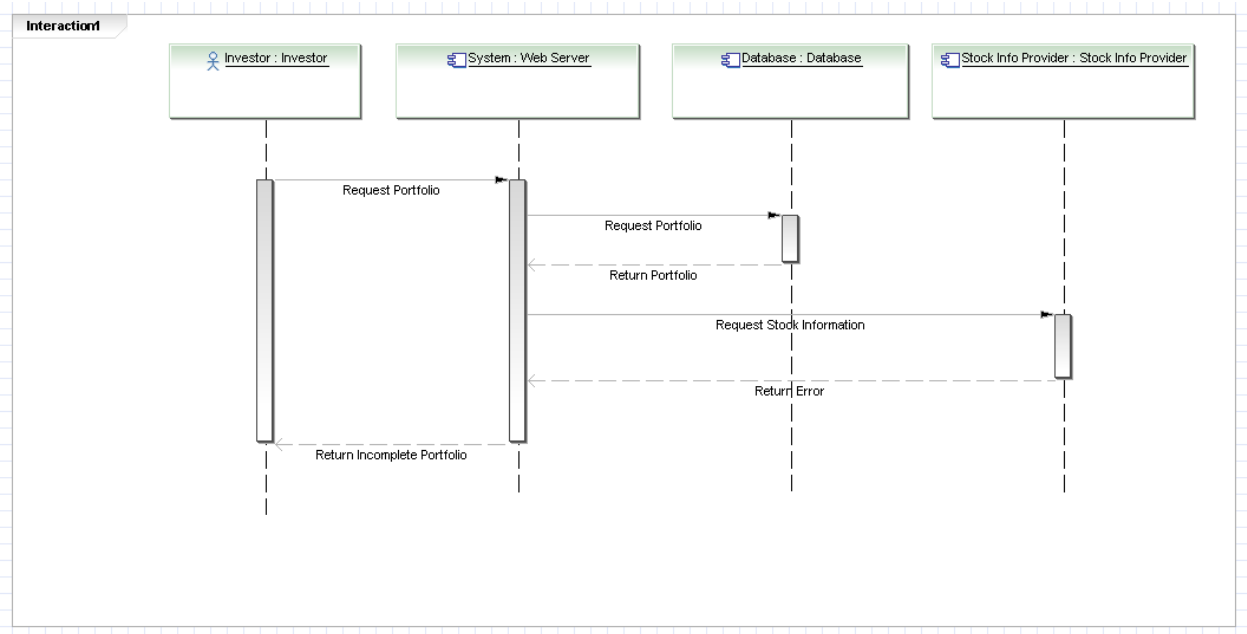


Figure 32: Alternative Sequence Diagram for View Portfolio

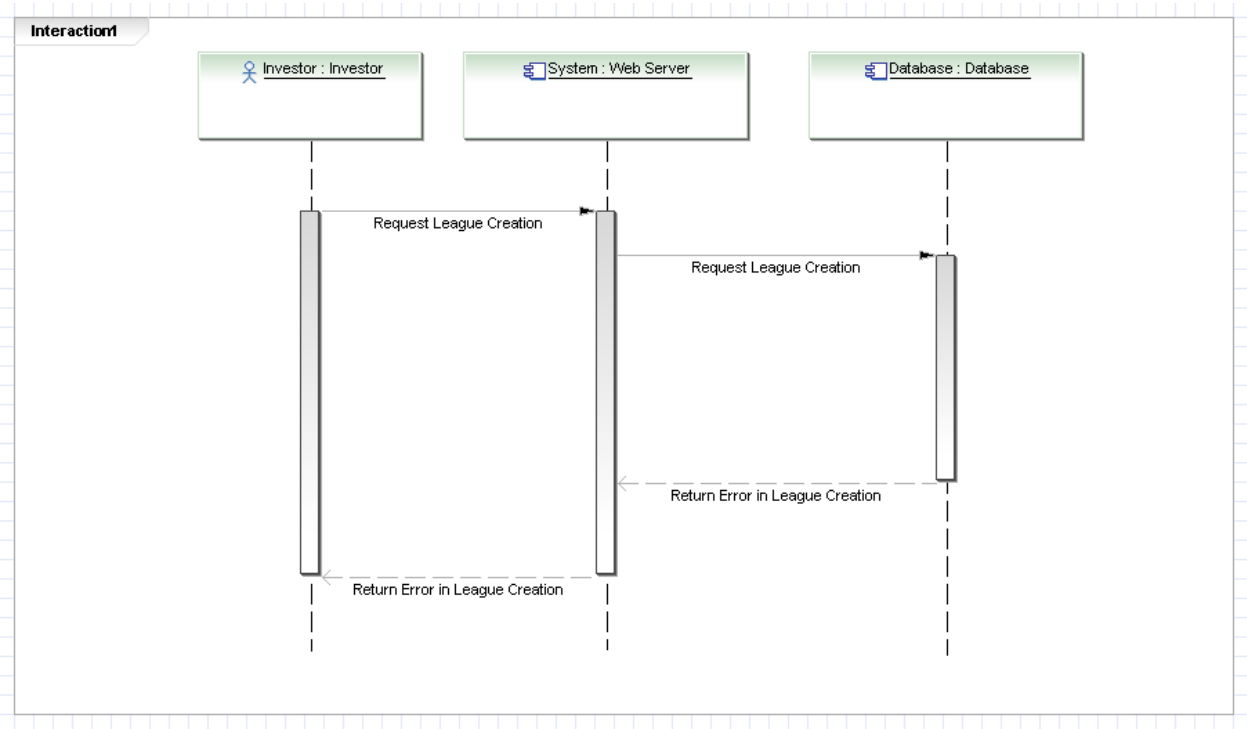


Figure 33: Alternative Sequence Diagram for Creating Leagues

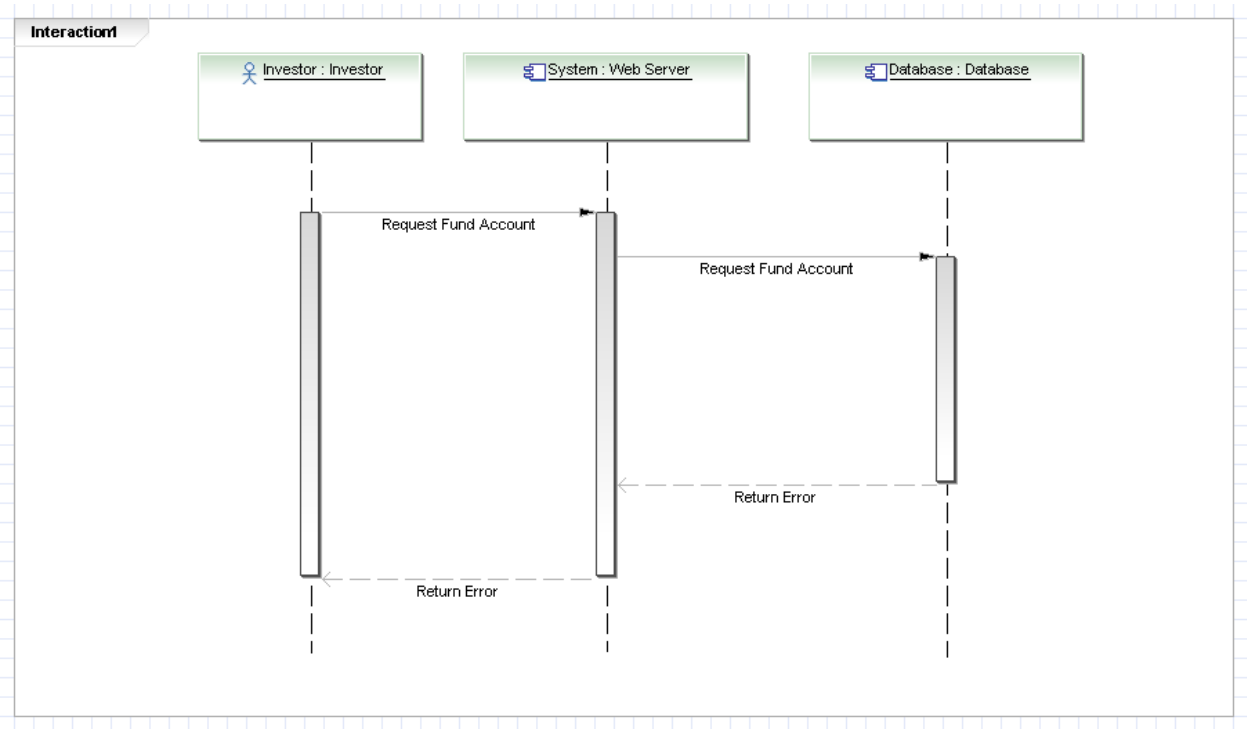


Figure 34: Alternative Sequence Diagram for Creating Funds

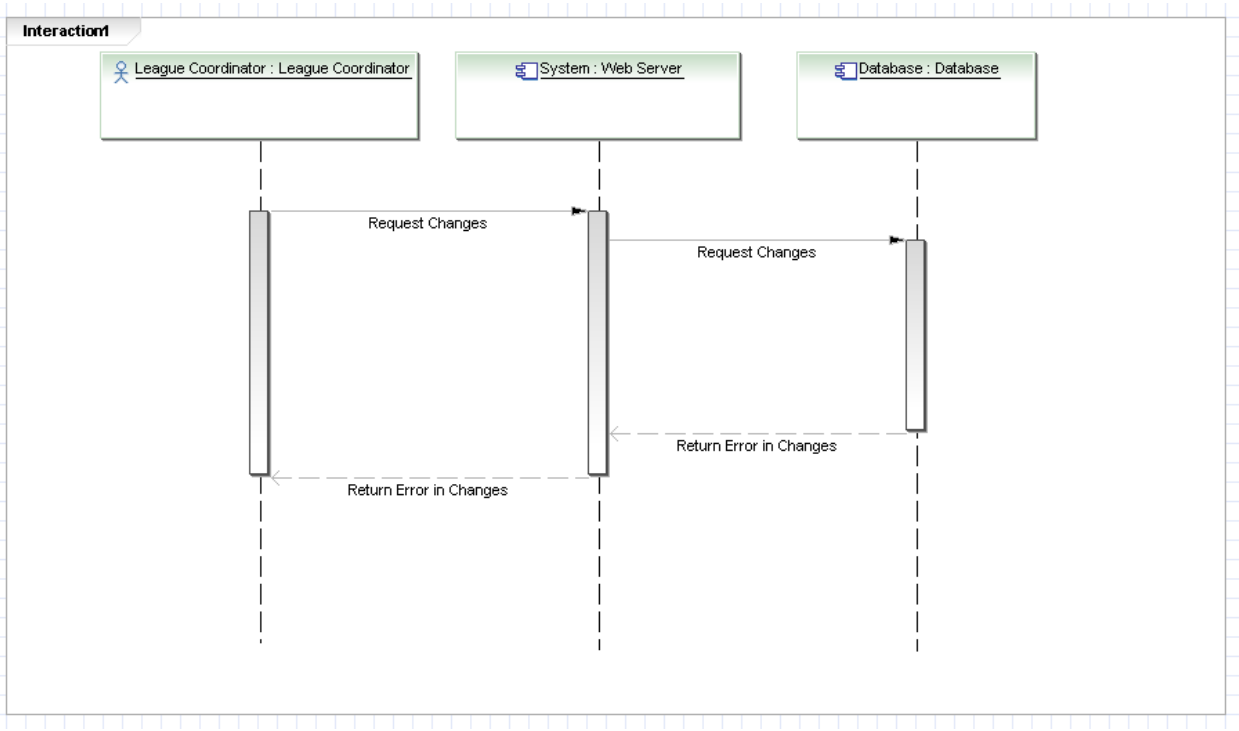


Figure 35: Alternative Sequence Diagram for Manage League

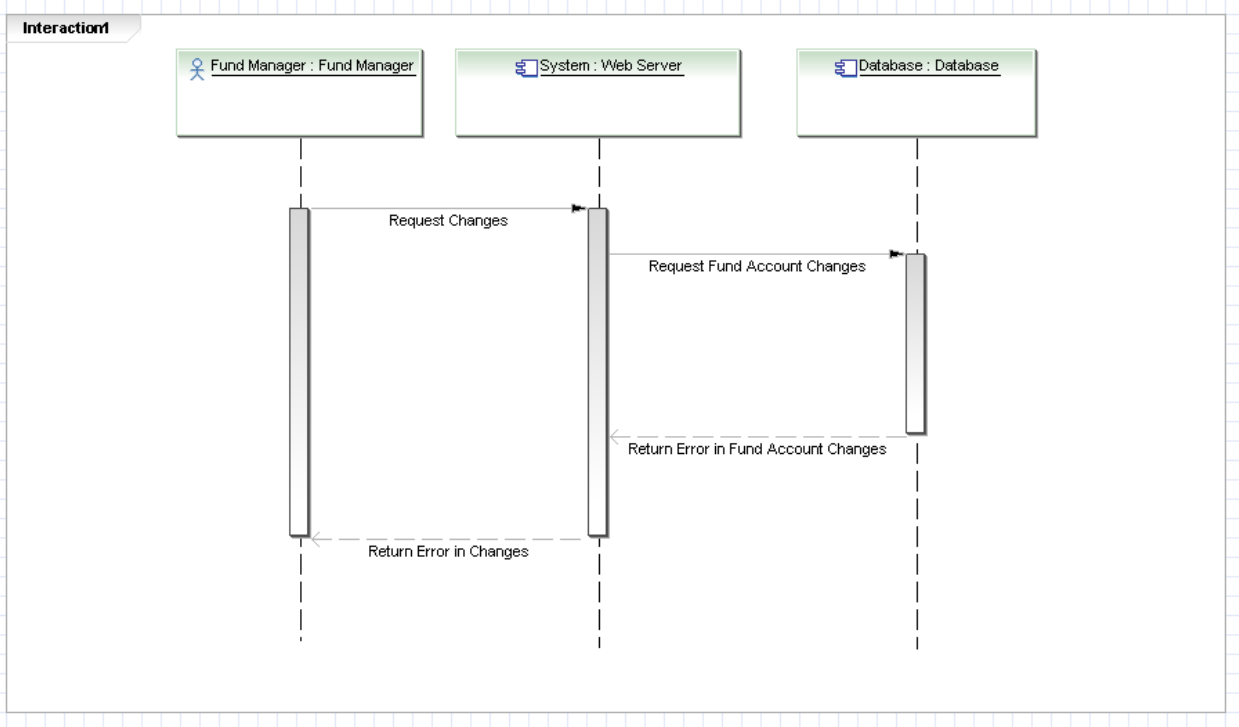


Figure 36: Alternative Sequence Diagram for Manage Fund

8 References

References

[1] http://www.stockmarketnewz.com/2011/10/19/who\%E2\%80\%99s-right-about-commodities-bears-or-bulls/.

[2] Discount Broker. http://www.investopedia.com/terms/d/discountbroker.asp#axzz1mOap5Tqp.

[3] Omondo: The Live UML Company. http://omondo.com.

[4] Facebook Developers.

[5] Benjamin Graham. *The Intelligent Investor*. HarperCollins Publishers, revised edition, 2003.

[6] Technical Indicators and Overlays. http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators.

[7] George Kleinman. *Trading Commodities & Financial Futures*. Prentice Hall, 3rd edition, 2005.

[8] Ivan Marsic. *Software Engineering*. Unpublished, first edition edition, 2012.

[9] Pitfail. http://github.com/pitfail.

[10] Gantt Project. http://www.ganttproject.biz.

[11] Alessandro S. http://www.linkedin.com/answers/financial-markets/equity-markets/MKT_EQU/1231-10230.

[12] Argo UML. http://argouml.tigris.org.