Bears & Bulls

332:452 Software Engineering
Report 3

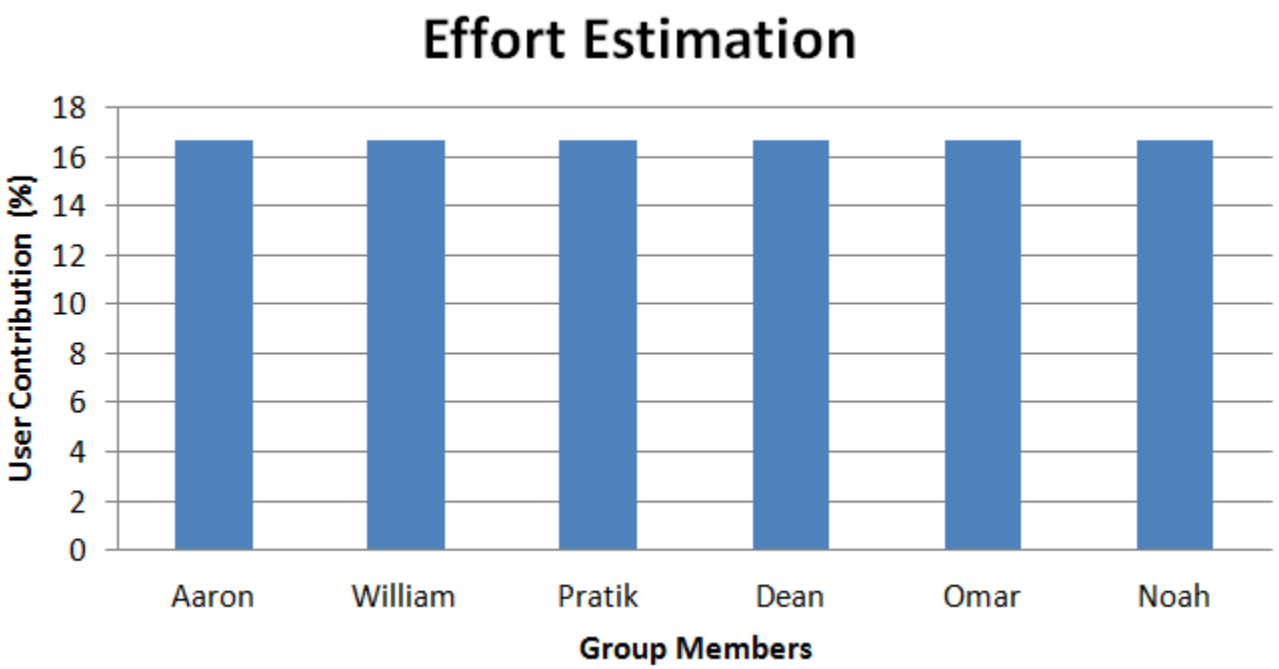
Group 6:
William Pan, Aaron Sun, Pratik Ringshia
Dean Douvikas, Omar Raja, Noah Silow-Carroll

URL: https://apps.facebook.com/bearsandbulls/

May 3, 2012

Contributions Breakdown

Task	William	Aaron	Pratik	Dean	Omar	Noah
Statement of Requirements	x	x	x	x	x	x
Glossary of Terms	x	x	x	x	x	x
Functional Requirements	x	x	x	x	x	x
Effort Estimation	x	x	x	x	x	x
Domain Analysis	x	x	x	x	x	x
Interaction Diagrams	x	x	x	x	x	x
Class Diagrams	x	x	x	x	x	x
System Architecture and Design	x	x	x	x	x	x
Algorithms and Data Structures	x	x	x	x	x	x
UI Design and Implementation	x	x	x	x	x	x
Design of Tests	x	x	x	x	x	x
History of Work	x	x	x	x	x	x
Project Management	x	x	x	x	x	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 using 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

Summary of Changes

System Requirements - Functional
- REQ-5: Removed League fees and prizes
- REQ-6: Removed technical and fundamental indicators
- REQ-8: Removed use of Facebook credits

System Requirements - Nonfunctional
- REQ-15: Removed performance constraint

System Requirements - On-screen
- Requirements enumerated

Stakeholders
- Sponsors no longer listed

Actors and Goals
- Investor - removed adding stocks to watchlist
- League Coordinator - removed adding coordinators and deleting members
- Fund Manager - added actor
- System Administrator - removed payments

Use Cases
- UC-4: Removed player ranking history
- UC-6: Removed Watchlist use case
- UC-9: Removed Pay League use case
- UC-10: Change to comments on leaderboard
- UC-15: Removed Add Coordinator use case
- UC-16: Removed Remove User use case
- UC-17: Removed Update Models use case
- UC-19: Removed Manage Money use case

Use Case Diagram
- Diagram and description updated
- Stock info provider no participates in use cases

Alternatives omitted

Fully Dressed Descriptions
Updated descriptions
Removed deprecated use cases

Use Case Points
Section completely redone

Domain Analysis
Section updated to reflect changes in requirements

Class Diagram and Interface Specifications
Design patterns included
Object constraint language included

User Interface Design and Implementation
New section with updated interface

Design of Tests
Tests for deprecated features removed

1 Customer Statement of Requirements

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 users 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.

2 Glossary of Terms

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* 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*.
- **Private** – A *private league* can only be joined through invitation.
- **Public** – A *league* that can be joined by an *user*.

League Coordinator – A *player* who acts as an administrator of a *league*. Responsibilities include inviting users and managing details of the *league*.

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

Player – A *user* of Bears & Bulls. This member joins *leagues* and competes with existing members. Synonomous with *investor* and *user*.

Portfolio – Detailed account of *stocks* associated with each of a *league's players*. A *player* will have a unique portfolio per *league* and per *fund*. 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.[7]

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.

3 System Requirements

3.1 Functional 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 duration, capital limits, allowed sectores 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 price data, bid/ask sizes, volume and a news feed of relevant articles.
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 simulate market liquidity when trading high volumes of stocks

REQ-9	2	The system shall support trading on the margin. 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-10	1	The system shall allow users to submit comments to the system administrators.

3.2 Nonfunctional Requirements

ID	PW	Requirement
REQ-11	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-12	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-13	4	The system shall have a common aesthetic theme and any two pages shall be separated by no more than 4 links.
REQ-14	3	The sytem shall be platform independent and should run equally well on Windows, Mac and *nix systems. The system shall have consistent appearance between browsers.
REQ-15	2	The system shall maintain function in the event of any changes to Facebook's API.

3.3 On-Screen Appearance Requirements

ID	PW	Requirement
REQ-16	5	The system must fit within a Facebook iframe with no clipping. The maximum width of any page must be 760 pixels.
REQ-17	3	The system must have a consistent look across different browsers and screen resolutions.
REQ-18	3	Advertisements must adhere to Facebook's Advertising Guidelines.

4 Functional Requirements

4.1 Stakeholder

- *Facebook Users* who wish to use the system for entertainment.
- *Novice Investors* who wish to use the system to practice investing.
- *System Administrators* who will maintain the system as well as manage the global *league*.

4.2 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

League Coordinator – Initiating Actor

1. Invite other users to the investment *league*
2. Manage *league* details

Fund Manager – Initiating Actor

1. Invite other users to the fund *league*
2. Manage *fund* details

System Administrator – Initiating Actor, Participating Actor

1. To maintain the database and website
2. To view messages from users

Stock Info Provider – Participating Actor
Database Server – Participating Actor
Web Server – Participating Actor
Facebook – Participating Actor

4.3 Use Cases

4.3.1 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. Transaction history is a compilation of previous trades 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: 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-7: 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-8: Submit Comment

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

Goal: To submit comments to system administrators or on leaderboards. This allows *investor* to provide feedback to system admins, as well as allowing for discussion amongst traders.

Use Case UC-9: 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-10: Join League

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

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

Use Case UC-11: Manage League

Actor: League Coordinator (*Initiating*), Database (*Participating*)
Goal: To manage *league* details such as adding users and setting *league* rules.

Use Case UC-12: 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-13: Update Models

Actor: Sys Admin (*Initiating*)
Goal: To update liquidity model that simulates price slippage during high volatility and block trades.

Use Case UC-14: 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-15: Manage Fund

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

4.3.2 Use Case Diagram

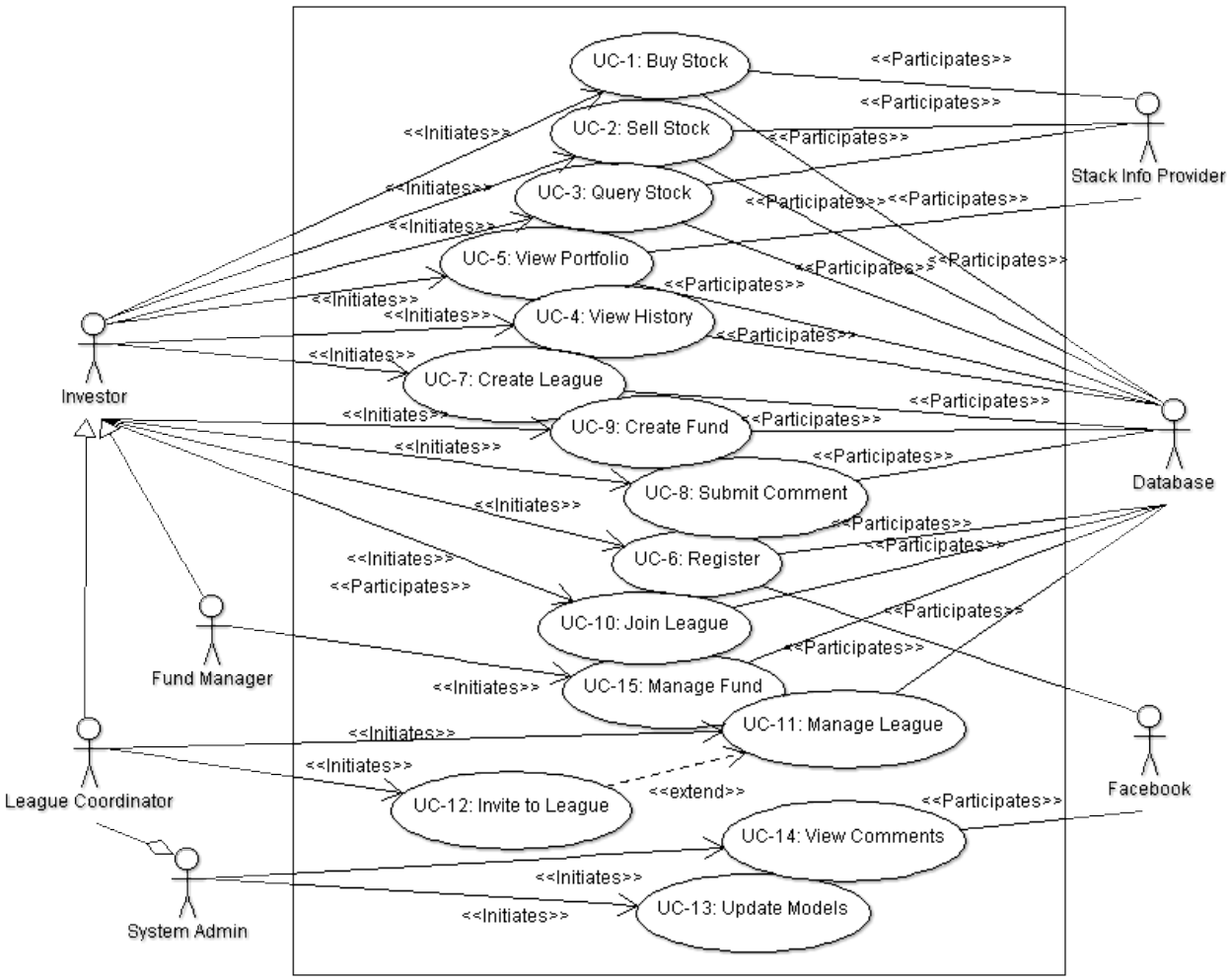


Figure 1: Use Case Diagram [12]

Figure 1 shows the relations between actors and the use cases. The *investor*, system admin, *league coordinator*, and *fund manager* are the only initiating 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 Manage League). Facebook participates in login and registration, pulling account information such as names and profile pictures, and also it provides the comment functionality. The stock info provider

only participates in 4 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.

As 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* duties seemed too similar to the *investor's* duties to warrant a new set of use cases.

4.3.3 Traceability Matrix

R# = REQ-#

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10		
PW	5	5	5	5	4	4	4	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	x		x	x		x		x	x	5	26
UC07					x						4	4
UC08										x	1	1
UC09				x			x				5	9
UC10					x						4	4
UC11					x						4	4
UC12					x						4	4
UC13								x			3	3
UC14										x	1	1
UC15							x				4	4

4.3.4 Fully-Dressed Use Cases

Use Case UC-1: Buy Stock

Related Requirements: REQ-2, REQ-3, REQ-8, REQ-9
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-8, REQ-9

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-6: 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-7: 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-9: 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-11: 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-12: 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-15: 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.

4.4 System Sequence Diagrams

NOTE: Diagrams of alternative implementations are in the appendix.

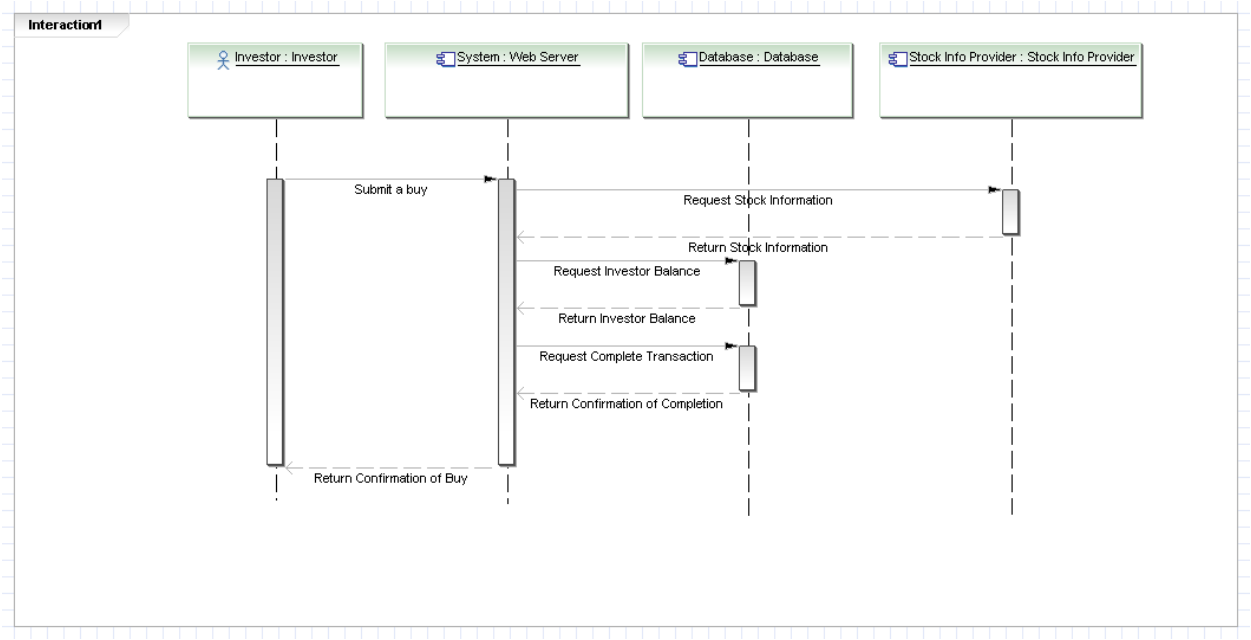


Figure 2: UC-1: Buy Stock

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.

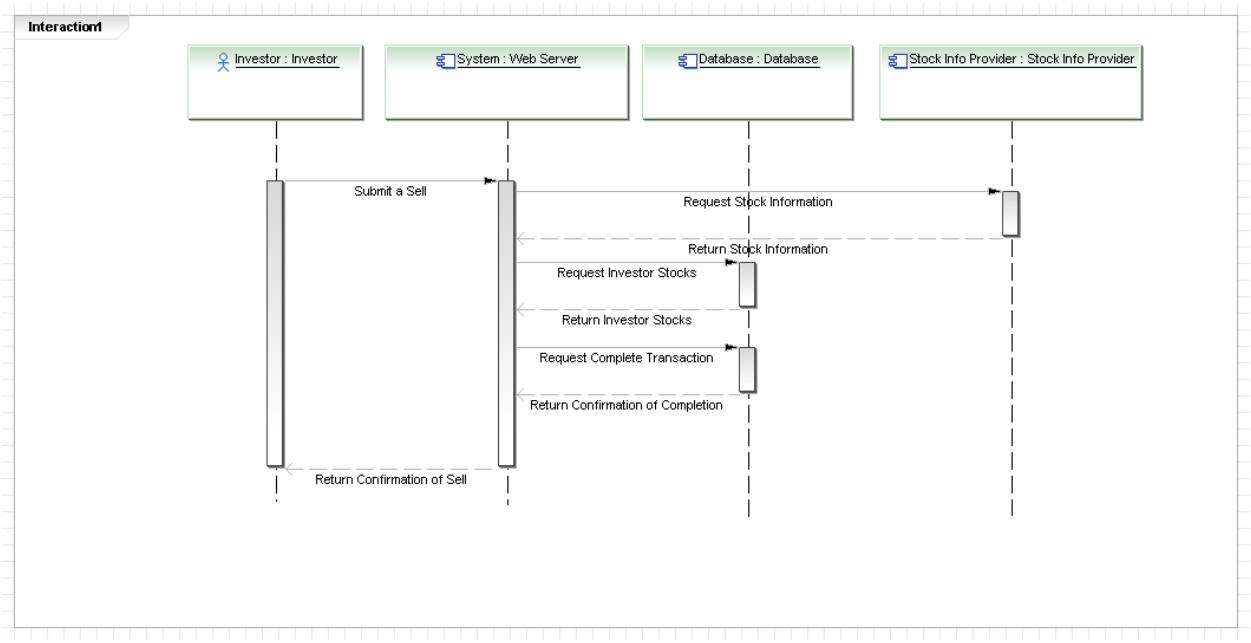


Figure 3: UC-2: Sell Stock

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.

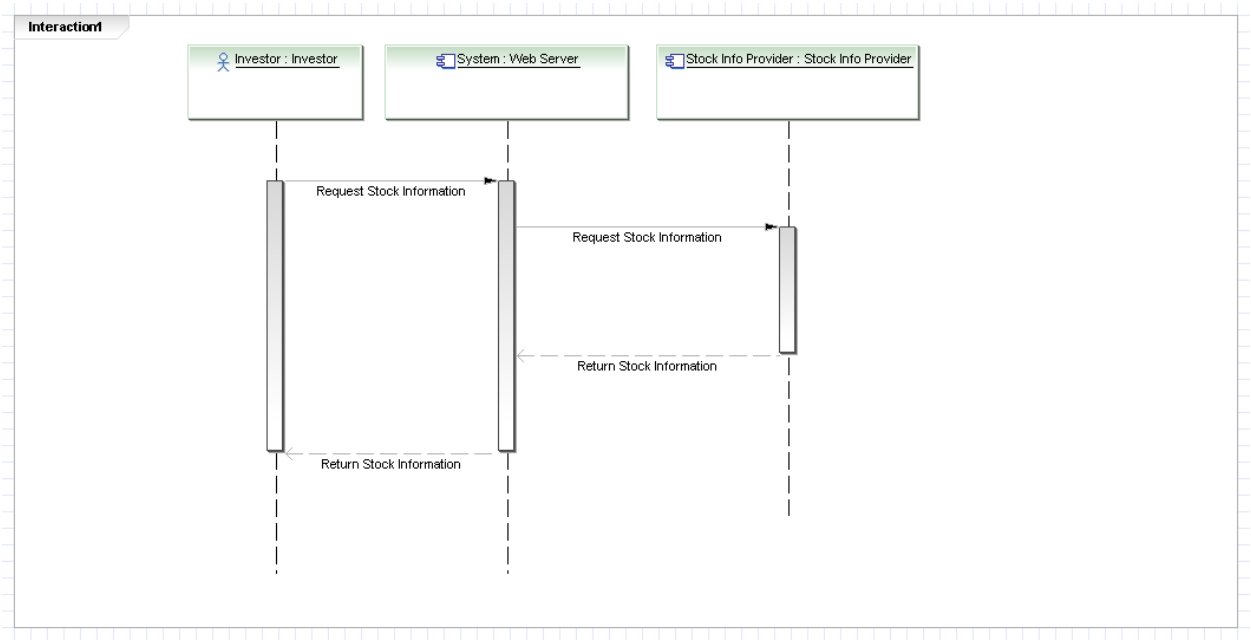


Figure 4: UC-3 Query Stock

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).

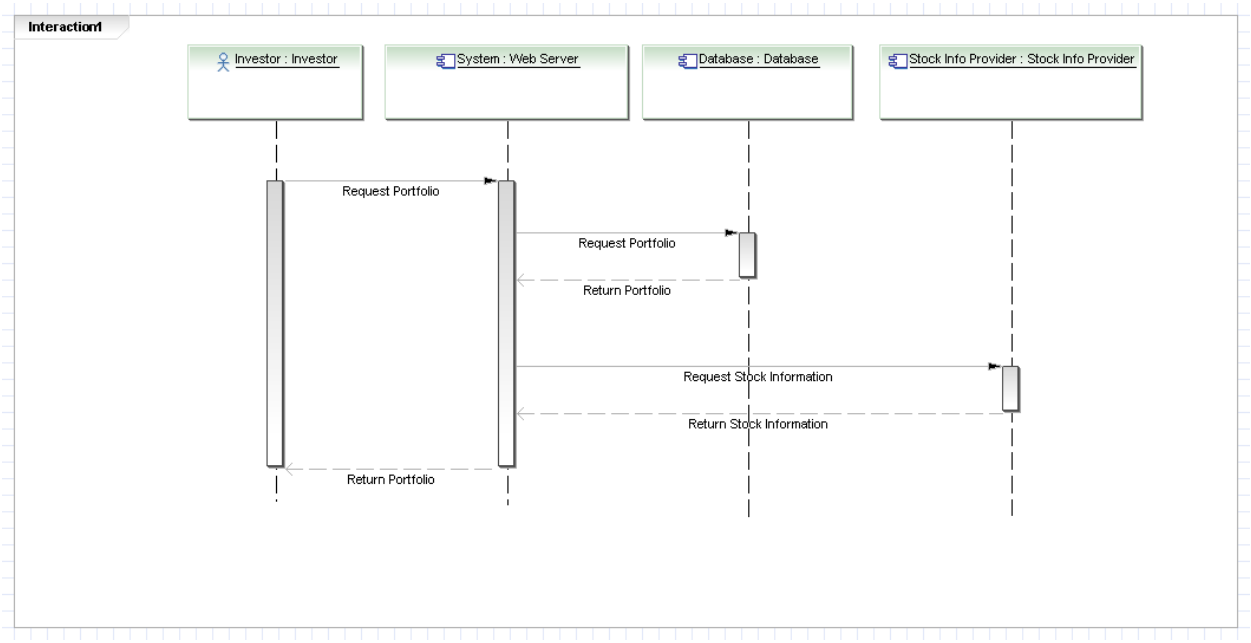


Figure 5: UC-5 View Portfolio

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).

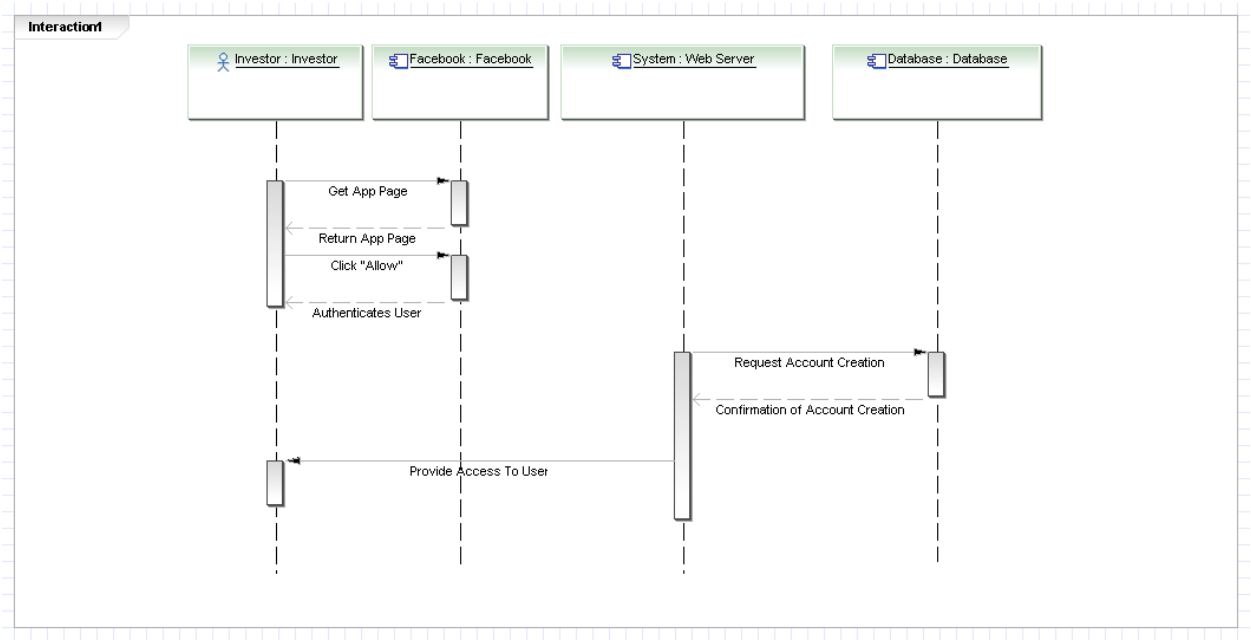


Figure 6: UC-6 Register

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.

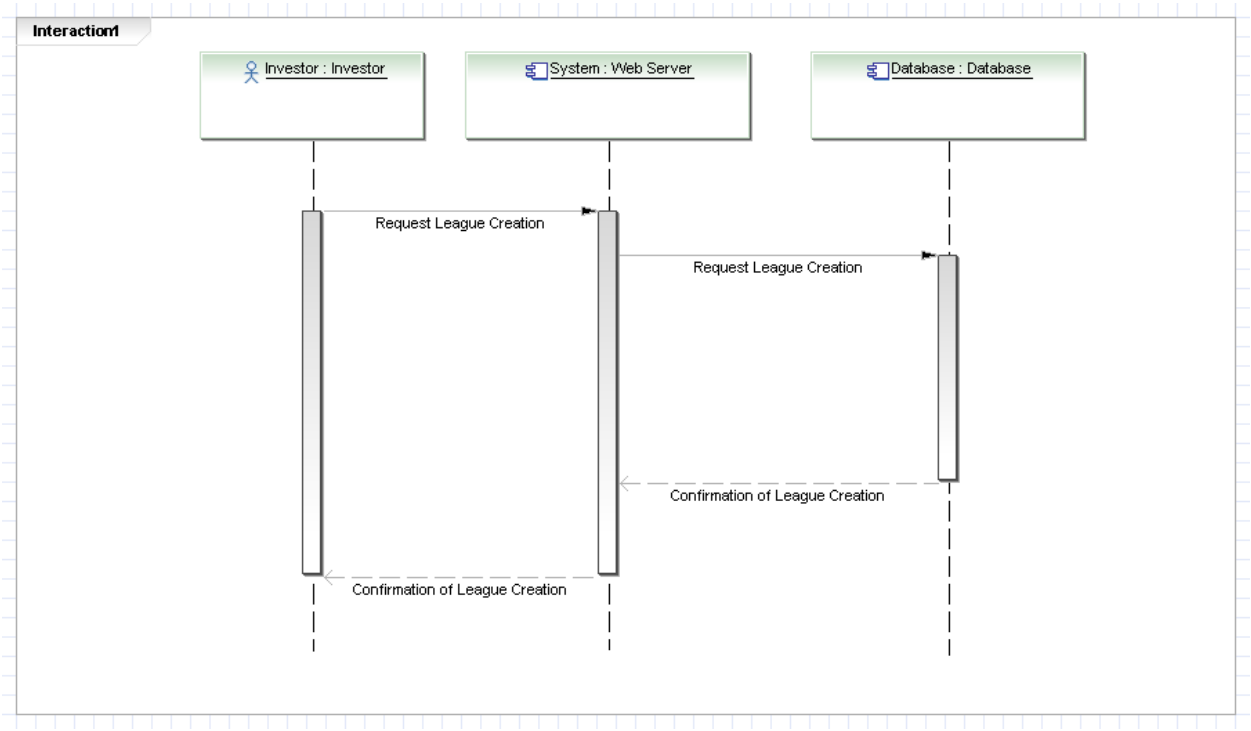


Figure 7: UC-7 Create League

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.

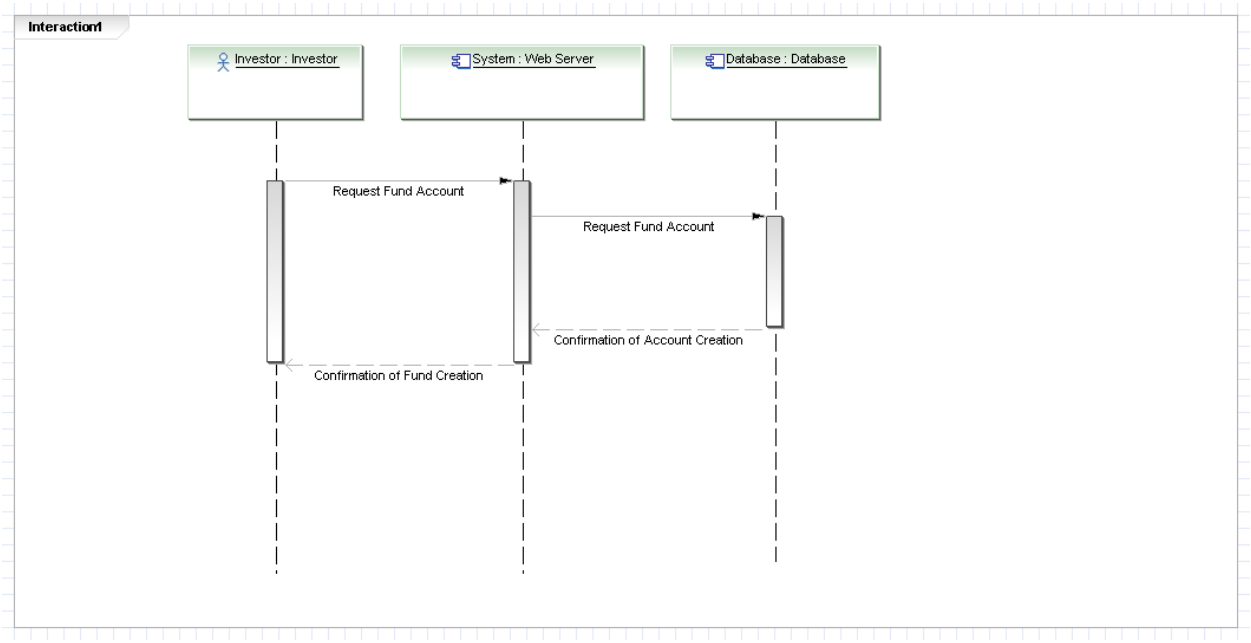


Figure 8: UC-9: Create Fund

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.

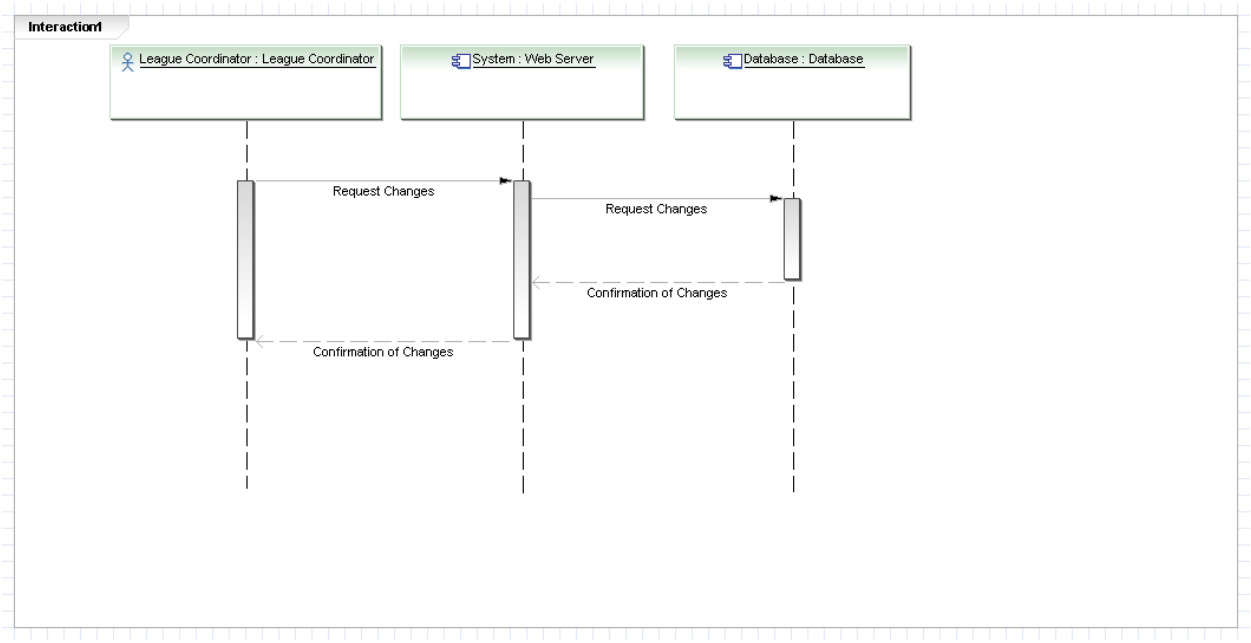


Figure 9: UC-11 Manage League

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.

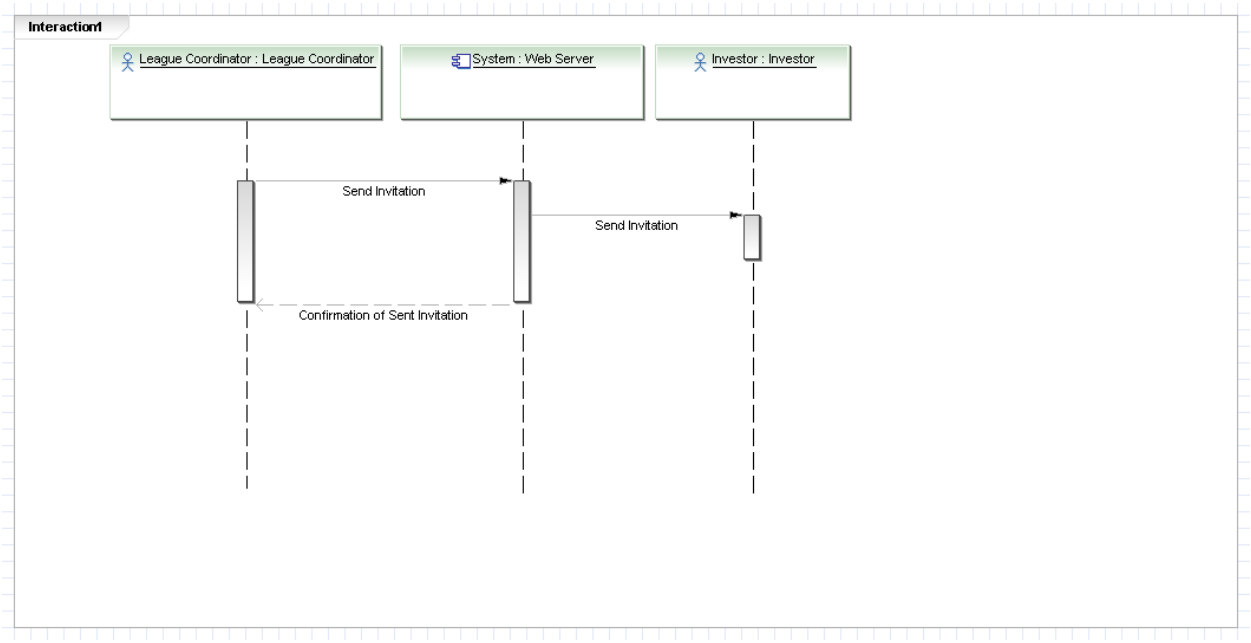


Figure 10: UC-12 Invite to League

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*.

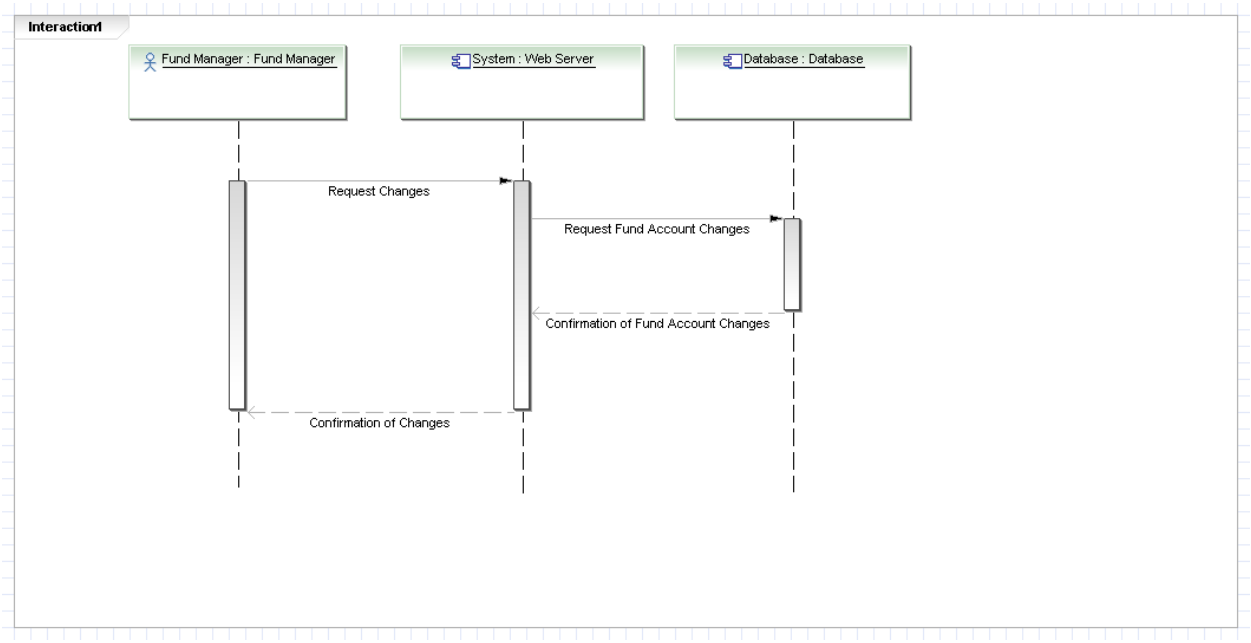


Figure 11: UC-15 Manage Fund

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.

5 Effort Estimation using Use Case Points

Use Case Points (UCP) method provides the ability to estimate the person-hours a software project requires based on its use cases. UCP method analyzes the use case actors, scenarios, nonfunctional requirements, and environmental factors and joins them in a simple equation: UCP = UUCP * TCF * ECF.

- Unadjusted Use Case Points (UUCP) – Measures complexity of functional requirements
- Technical Complexity Factor (TCF) – Measures complexity of nonfunctional requirements
- Environmental Complexity Factor (ECF) – Asseses development teams experience and their development environment

5.1 Unadjusted Use Case Points

UUCP are computer as a sum of the following two components:

- Unadjusted Actor Weight (UAW) – Combined complexity of all the actors in all the use cases
- Unadjusted Use Case Weight (UUCW) – Total number of activities contained in all the use case scenarios

UUCP = UAW + UUCW

5.1.1 Unadjusted Actor Weight

The weights for Actor classification are computed via the following table: Actor classification and associated weights

Actor	Description of Actor Type	Weight
Simple	The actor is another system which interacts with our system through a defined application programming interface (API)	1
Average	The actor is a person interacting through a text-based user interface, or another system interacting through a protocol, such as a network communication protocol	2
Complex	The actor is a person interacting via a graphical user interface	3

Actor Classification for Bears & Bulls

Actor	Description of Characteristics	Complexity	Weight
Investor	Investor is interacting with the system through a graphical user interface.	Complex	3
League Coordinator	League Coordinator is interacting with the system through a graphical user interface.	Complex	3
Fund Manager	Fund Manager is interacting with the system through a graphical user interface.	Complex	3
System Admin.	System Administrator is interacting with the system via a graphical user interface. (Creators of Bears & Bulls)	Complex	3
Stock Info. Provider	Stock Information Provider (Yahoo Finance) is interacting with the system through a network protocol.	Average	2
Database Server	Database is another system interacting through a protocol.	Average	2
Web Server	Web Server is another system interacting through HTTP	Average	2
Facebook	Facebook is the system interacting through a protocol which is the website used to host our entire application.	Average	2

UAW (Bears & Bulls) = 4 * Average + 4 * Complex = 20

5.1.2 Unadjusted Use Case Weight

The weights for use cases are computer via the following table:

Use Case Category	Description of Category	Weight
Simple	Simple user interface. Up to one participating actor (plus initiating actor). Number of steps for the success scenario: no more than 3. If presently available, its domain model includes no more than 3 concepts.	5
Average	Moderate interface design. Two or more participating actors. Number of steps for the success scenario: 4 to 7. If presently available, its domain model includes between 5 and 10 concepts.	10
Complex	Complex user interface or processing. Three or more participating actors. Number of steps for the success scenario: at least 7. If available, its domain model includes at least 10 concepts.	15

Use case classification for Bears & Bulls

Use Case	Description	Category	Weight
Buy Stock (UC-1)	Simple user interface. 8 steps for main success scenario. Three participating actors (Investor, Database, Stock Info Provider).	Complex	15
Sell Stock (UC-2)	Simple user interface. 10 steps for main success scenario. Three participating actors (Investor, Database, Stock Info Provider).	Complex	15
Query Stock (UC-3)	Simple user interface. 4 steps for main success scenario. Three participating actors (Investor, Database, Stock Info Provider).	Average	10
View History (UC-4)	Simple user interface. 6 steps for main success scenario. Two participating actors (Investor, Database).	Average	10
View Portfolio (UC-5)	Simple user interface. 6 steps for main success scenario. Three participating actors (Investor, Database, Stock Info Provider).	Average	10

Register (UC-6)	Simple user interface. 7 steps for main success scenario. Three participating actors (Investor, Database, Facebook).	Average	10
Create League (UC-7)	Simple user interface. 4 steps for main success scenario. Three participating actors (Investor, Database, Web Server).	Average	10
Submit Comment (UC-8)	Simple user interface. 4 steps for main success scenario. Three participating actors (Investor, Database, Web Server).	Simple	5
Create Fund (UC-9)	Simple user interface. 3 steps for main success scenario. Two participating actors (Investor, Database).	Simple	5
Join League (UC-10)	Simple user interface. 5 steps for main success scenario. Three participating actors (Investor, Database, Web Server).	Average	10
Manage League (UC-11)	Moderate user interface. 4 steps for main success scenario. Two participating actors (League Coordinator, Database).	Average	10
Invite to League (UC-12)	Simple user interface. 3 steps for main success scenario. Three participating actors (League Coordinator, Database, Investor).	Average	10
Update Models (UC-13)	Moderate user interface. 7 steps for main success scenario. One participating actor (System Administrator).	Average	10
View Comment (UC-14)	Simple user interface. 3 steps for main success scenario. Two participating actors (Investor, Database).	Simple	5
Manage Fund (UC-15)	Moderate user interface. 4 steps for main success scenario. Two participating actors (Fund Manager, Database).	Average	10

5.1.3 Computing Unadjusted Use Case Points

UUCW(Bears & Bulls) = 3 * Simple + 10 * Complex = 145
UUCP(Bears & Bulls) = UAW + UUCW = 20 + 145 = 165

5.2 Technical Complexity Factor

Technical Complexity Factor (TCF) is computed using thirteen standard technical factors to estimate the impact of productivity of the nonfunctional requirements for the application. We then need to assess the perceived complexity of each technical factor in the context of the project. A perceived complexity value is between 0 and 5, with 0 meaning trivial effort, 3 meaning average effort and 5 meaning major effort. Each factors weight is then multiplied by perceived complexity factor to produce calculated factor. Two constants are used with TCF. The constants utilized are C1 = 0.6 and C2 = 0.01.

$$TCF = Constant1 + Constant2 \times TechnicalFactorTotal = C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i$$

Technical complexity factors and their weights:

Technical Factor	Description	Weight
T1	Distributed system	2
T2	Performance objectives	1
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Reusable design or code	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent use	1
T11	Special security features	1
T12	Provides direct access for third parties	1
T13	Special user training facilities are required	1

Technical complexity factors for Bears & Bulls:
PC = Perceived Complexity, CF = Calculated Factor

Technical Factor	Description	Weight	PC	CF
T1	Distributed, web-based system	2	3	6
T2	User expects good performance, but will tolerate network latency	1	3	3
T3	End-user expects efficiency, which is achieved through caching	1	4	4
T4	Internal processing required multiple interactions with other subsystems	1	4	4
T5	No requriement for reusability	1	0	0
T6	No user installation required	0.5	2	1
T7	Ease of use was very important	0.5	5	2.5
T8	Portable since it runs in a browser	2	2	4
T9	Relatively simple to add new features	1	2	2
T10	Concurrent use is required	1	4	4
T11	Security handled by Facebook	1	0	0
T12	No direct access for third parties	1	0	0
T13	No training required	1	0	0

$\text{TCF} = 0.6 + (0.01 \times 32.5) = 0.925$.
This results in a decrease of the UCP by 7.5 %.

5.3 Environment Complexity Factor

The Environment Complexity Factor (ECF) is computed utilizing eight standard environmental factors to measure the experience level of the people on the project and the stability of the project. We then need to assess the perceived impact based on perception that factor has on projects success. 1 means strong negative impact, 3 is average and 5 means strong positive impact.

TCF is computed utilizing thirteen standard technical factors to estimate the impact of productivity of the nonfunctional requirements for the application. We then need to assess the perceived complexity of each technical factor in the context of the project. A perceived complexity value is between 0 and 5 with 0 meaning that it is irrelevant, 3 means average effort and 5 means major effort. Each factors weight is then multiplied by perceived complexity factor to produce calculated factor. Two constants are used with ECF. The constants utilized are C1 = 1.4 and C2 = -0.03.

$$ECF = Constant1 + Constant2 \times EnvironmentalFactorTotal = C_1 + C_2 \cdot \sum_{i=1}^{8} W_i \cdot F_i$$

Environmental complexity factors and their weights:

Environmental Factor	Description	Weight
E1	Familiar with the development process	1.5
E2	Application problem experience	0.5
E3	Paradigm experience	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	1
E7	Part-time staff	-1
E8	Difficult programming language	-1

Environmental Complexity Factors for Bears & Bulls:
PI = Perceived Impact, CF = Calculated Factor

Environment Factor	Description	Weight	PI	CF
E1	Beginner familiarity with UML-based development	1.5	1	1.5
E2	Half of team has familiarity	0.5	3	1.7
E3	Some knowledge of object-oriented approach	1	3	3
E4	Average lead analyst	0.5	2	1
E5	Highly motivated overall	1	4	4
E6	Requirements were stable	2	5	10
E7	Student staff (part-time)	-1	4	-4
E8	Used new programming languages but resources were readily available	-1	5	-5

ECF = 1.4 - (0.03 × 12) = 1.04
This results in an increase of UDP by 4%.

5.4 Calculating the Use Case Points

As mentioned earlier, UCP = UUCP × TCF × ECF.
From above calculations, UCP variables have the following values:
UUCP = 165
TCF = 0.925
ECF = 1.04
UCP = 165 × 0.925 × 1.04 = 158.73 or 159 use case points.

5.5 Deriving Project Duration from Use-Case Points

UCP is a measure of software size. We need to know the teams rate of progress through the use cases. We need to utilize the UCP and Productivity Factor (PF) to determine duration. The equation for computing Duration is:

$$Duration = UCP \times PF$$

Productivity Factor is the ratio of development person-hours needed per use case point. Assuming a PF = 28, the duration of our system is computed as follows:

$$Duration = UCPXPF = 159 * 28 = 4452$$

4452 person-hours for the development of the system.

This does not imply that the project will be completed in 4452/24 = 185 days 12 hours. A reasonable assumption is that each developer on average spent 20 hours per week on project tasks. With a team of six developers, this means the team makes 6 *20 = 120 hours per week. Dividing 4452 person-hours by 120 hours per week, we obtain the total of approximately 37 weeks to complete this project. Weve spent 15 weeks approximately on the project so far which gives us 21 weeks left to complete this project according to our estimation. The reason for the large estimate is due to the highly over-estimated productivity factor.

6 Domain Analysis

The old domain model is in the appendix.

6.1 Domain Model

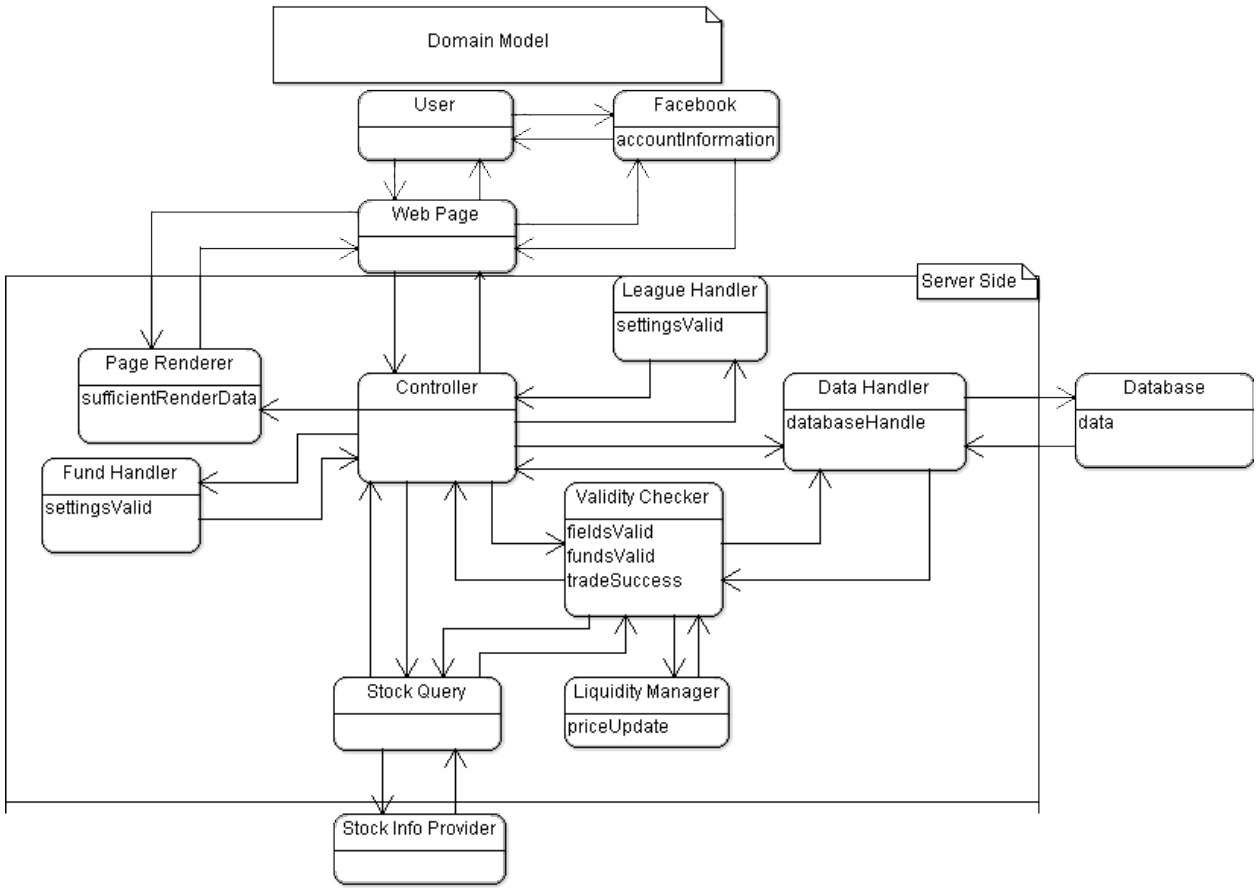


Figure 12: Domain Model

Figure 12 shows Bears & Bulls' new, updated Domain Model. The subsequent diagrams give insight into how the concepts work to satisfy the key use cases of our updated website. We will not show any alternate models, as we are no longer considering options on how to implement the design; the implementation is already done. The only major change from our old model is the addition of the Controller concept, which is now central to the flow of our processes, taking a lot of responsibility off of other concepts. Additionally, minor changes include the way some of our concepts interact, and how we have replaced web server, web browser, and web framework with just web page. We've added the Facebook concept as well, to show its involvment in the Registration use case.

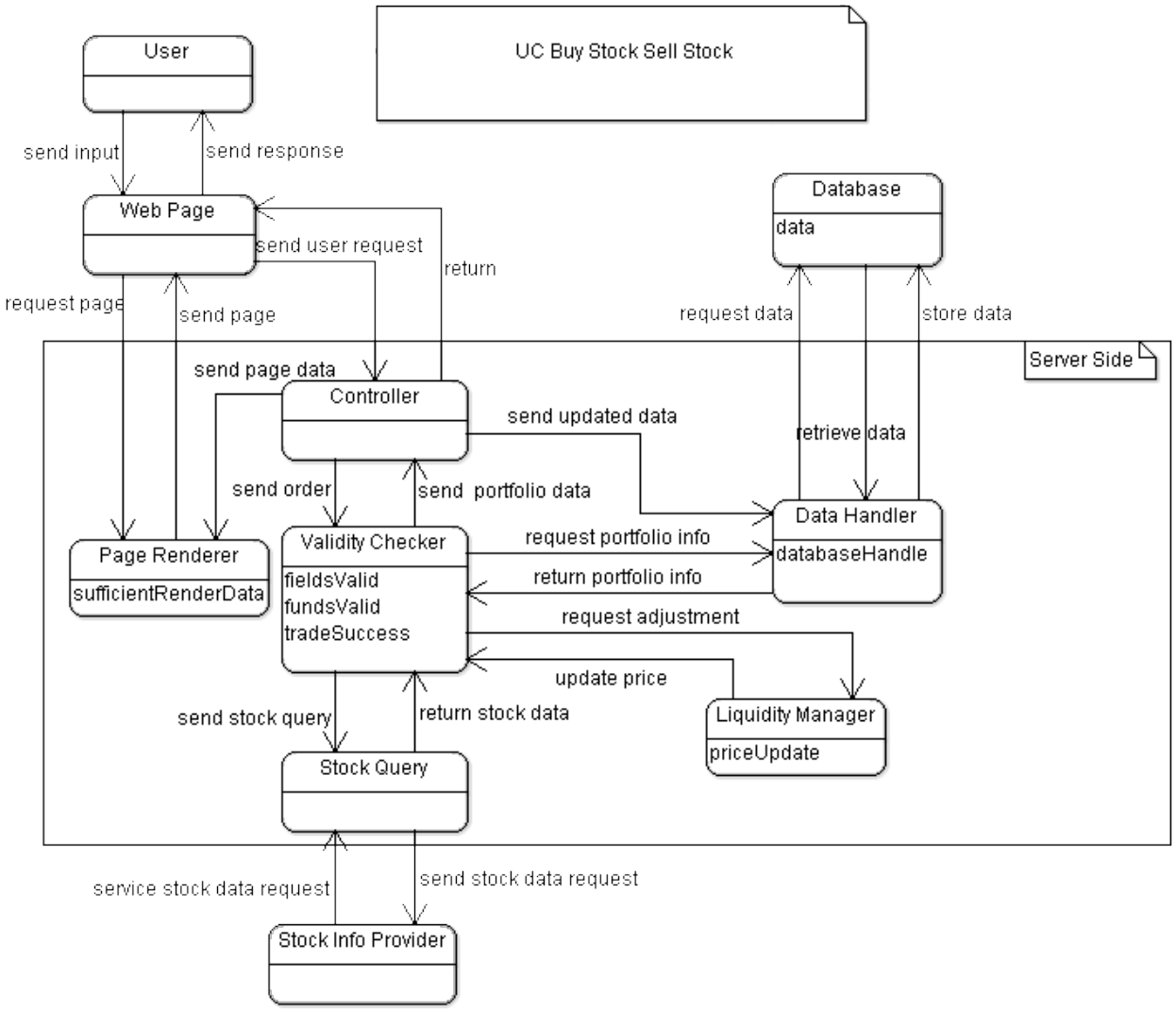


Figure 13: Place Order

Figure 13 represent both our buy (UC-1) and sell (UC-2) use cases since they behave in the same way. The User fills out order information on the web page, and sends to request to order to the Controller. The controller relays the order to the Validity Checker so that it can send the corresponding stock query to the Stock Query concept, which fetches the necessary information from the remote Stock Info Provider. The Validity Checker then sends a request to the Liquidity Manager to adjust the stock price based on our liquidity model. Now the Validity Checker must retrieve the User's balance in order to verify the transaction is valid; it requests for the Data Handler to get this information from the Database. If the transaction is successful, the Controller tells the Data Handler to update the User's portfolio. Then the Controller will let the Page Renderer know what page to generate and pass necessary data. The Web Page is informed of the completion of the order and knows to request the page to be viewed from the Page Renderer.

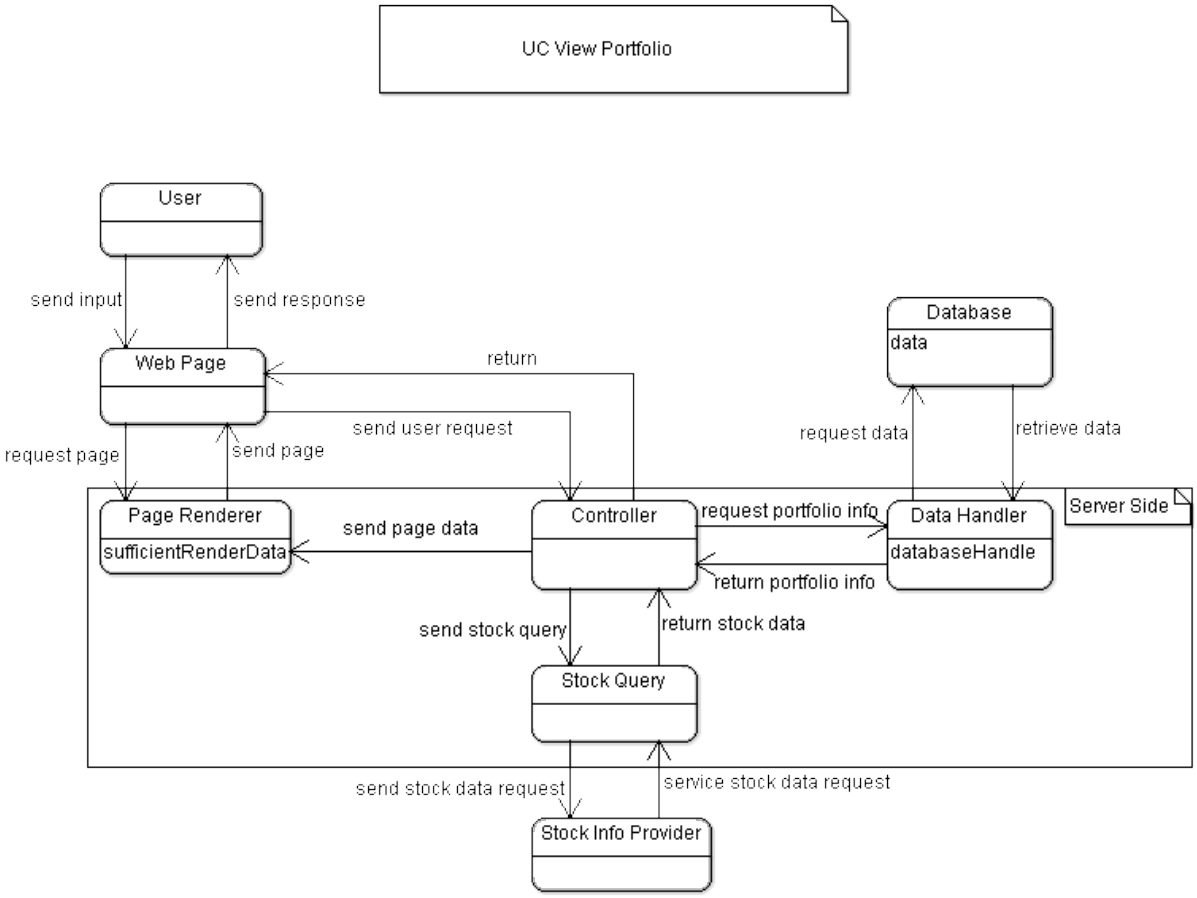


Figure 14: View Portfolio

Figure 14 shows the UC-5 View Portfolio. The User queries about a portfolio from the Web Page, and this request gets sent to the Controller. To get the necessary data, the Controller will send a request for the portfolio info to the Data Handler, which obtains this data from the Database. The Controller will then query Stock Query for each stock held by the User, which will obtain the necessary information from Stock Info Provider. The portfolio is now ready to be viewed, so Controller gives the Page Renderer all necessary data and then lets the Web Page know the process has been finished. The Web Page requests the Page Renderer to create the required page to be viewed.

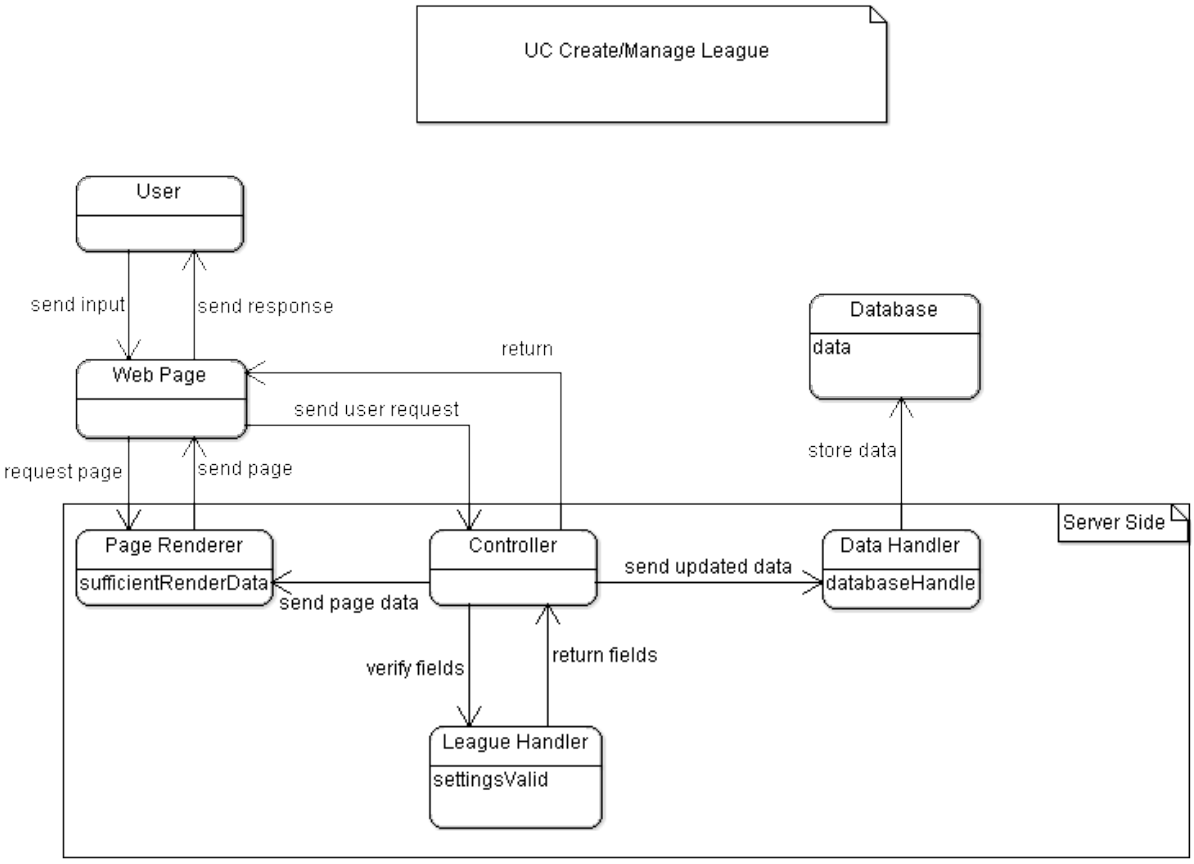


Figure 15: Create League

The creation and management of *leagues* are represented in Figure 15, which correspond to UC-7 and UC-11. The User fills in the necessary fields in order to create or change a *league* on the Web Page, then submits this info. The Controller will receive the request and call on the League Handler to verify the validity of the fields. If there are no errors, the Controller will inform the Data Handler to store the new *league* or its new settings. Then (regardless of the validity of the fields), the Controller provides the necessary page data to the Page Renderer and informs the Web Page of the completion of the process. The Web Page calls for the Page Render to create the necessary page to be viewed.

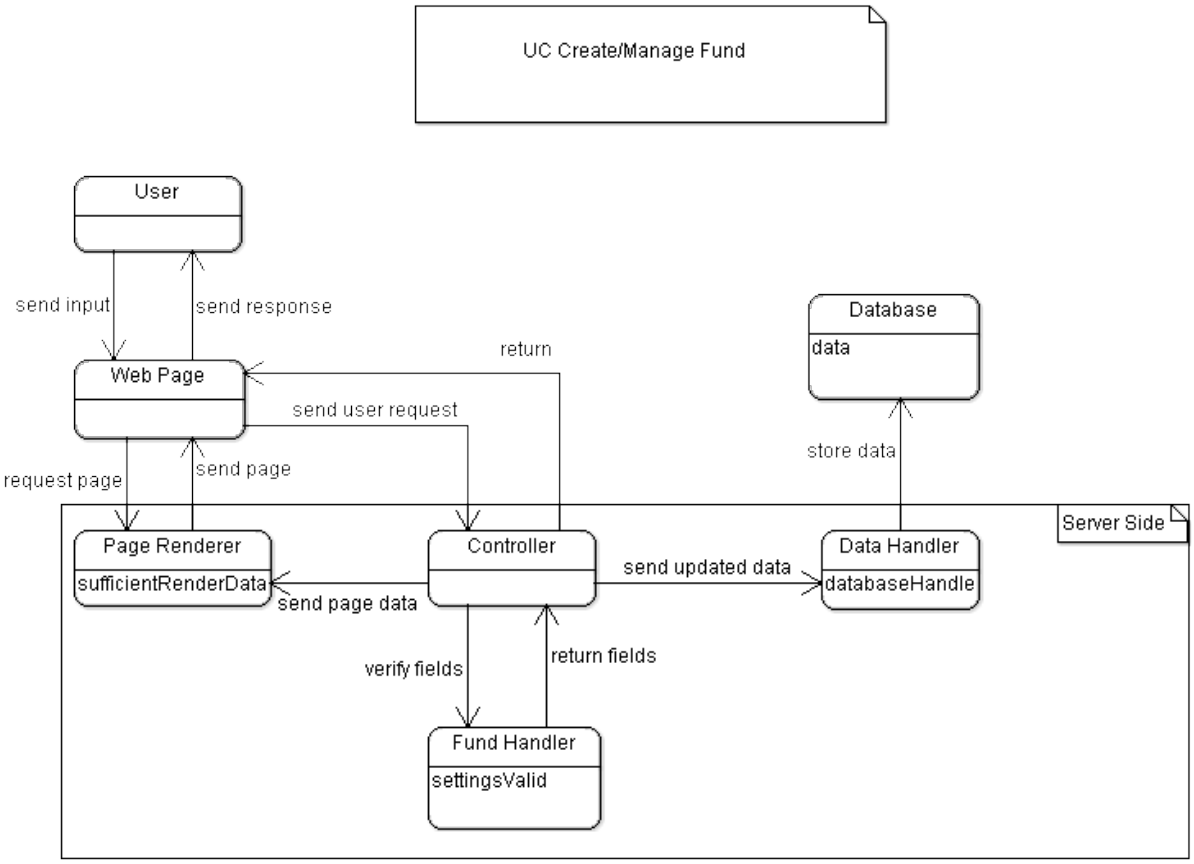


Figure 16: Create Fund

Figure 16 is identical to figure 15, replacing the League Handler with a Fund Handler. It corresponds to UC-9 and UC-15, the creation and maintenance of *Funds*. For insight on the process flow, please refer to figure 15.

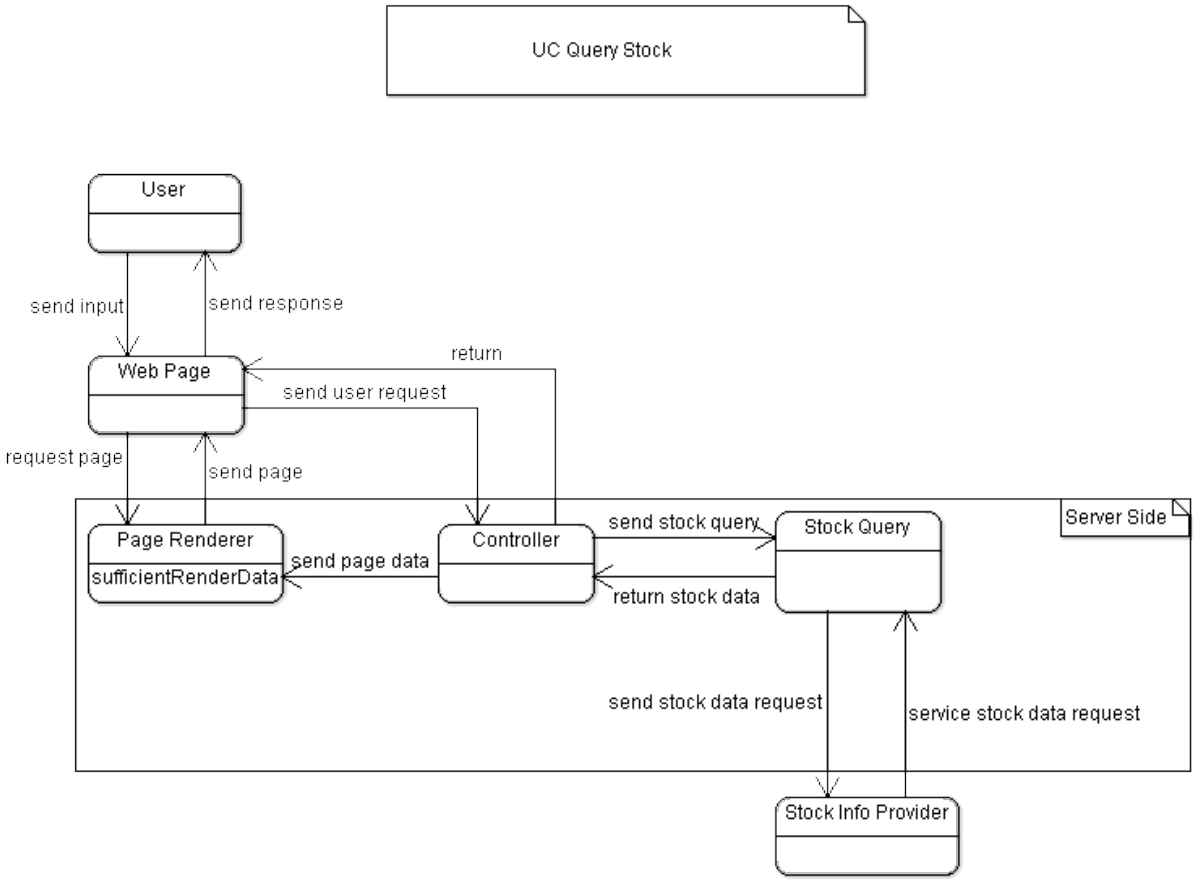


Figure 17: Query Stocks

Figure 17 shows the UC-3 Query Stocks. The stock is requested through the Web Page by the User, which tells Controller to inform the Stock Query to fetch the correct stock data from Stock Info Provider. Note that even an invalid ticker symbol will go through the same steps, the Stock Query will just return N/A or 0 for all the fields. The Controller now sends the data to be rendered to the Page Renderer and then notifies the Web Page that the process is complete. The Web Page knows to request the page from the Page Renderer, which then services the request and generates the correct page to be viewed by the User.

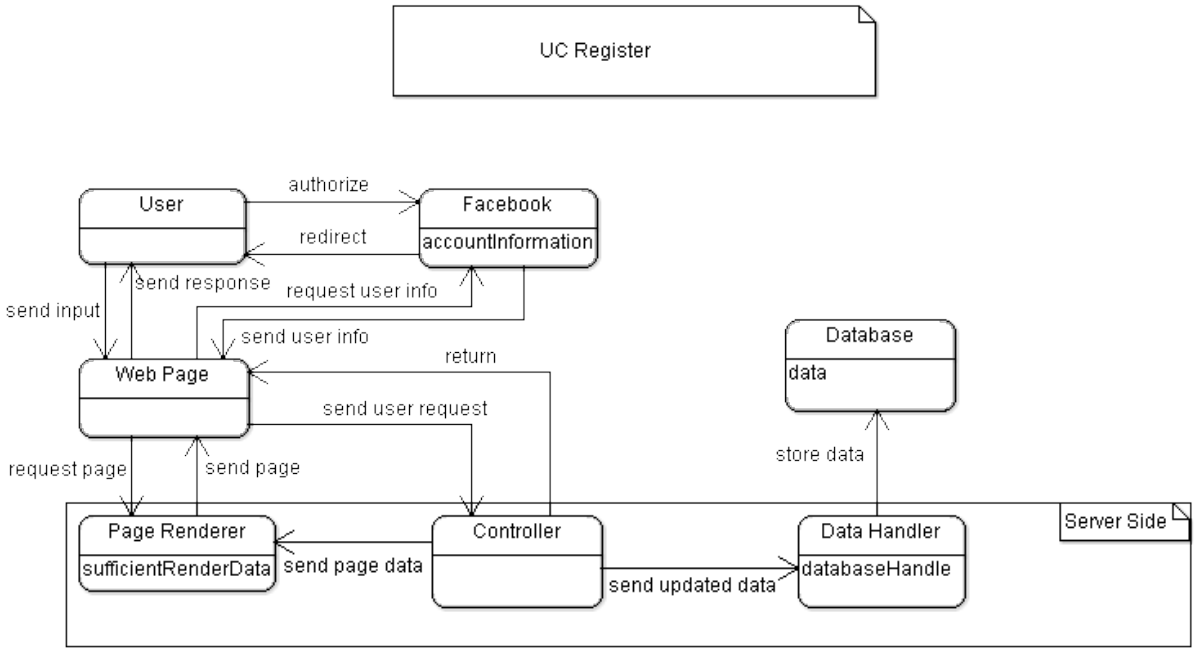


Figure 18: Register

The UC-6 Register is represented in Figure 18. This is a new use case the was not covered in the old Domain Model, as we now have more insight into how registration works with Facebook. First, the User tries to access the Home Page(Web Page), but since he is not yet registered, after an authorize check by Facebook, he is redirected to a page generated by Facebook's server. Once he chooses to add our application, our Web Page will request user info from Facebook, and upon obtaining this information, pass the data to our Controller. The Controller will then send instructions to the Data Handler to a new account in our Database. The Controller then passes necessary data to the Page Renderer and informs Web Page that the process is complete. The Web Page will call for the Page Renderer to generate the home page to be viewed by the User.

6.1.1 Concept Definitions

Our concept definitions have been updated to reflect a better understanding of how our concepts work together. Some concepts have been added, some removed, and the responsibilities of each have slightly changed since our original Domain Model.

User

Definition: A player playing Bears & Bulls.
Responsibilities:

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

Web Page

Definition: The page that the User is currently viewing, which runs from the User's device.
Responsibilities:

- Take requests from the User
- Send requests to the Controller
- Send page requests to the Page Renderer
- Update page to be displayed when new page is rendered

Page Renderer

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

- Receive the required data to generate new page
- Convert the data into user-friendly format
- Send rendered pages to the Web Page

Controller

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

- Receive user requests from the Web Page
- Inform Web Page when process is complete
- Send page data to be rendered

- Send *League* and *Fund* settings to be validated
- Send updated portfolio info
- Request stock queries
- Request account creation
- Request an order

Stock Query

Definition: Fetch real-time stock prices
Responsibilities:

- Receive requests for stock data
- Request information from Stock Info Provider
- Retrieve information from Stock Info Provider
- Send updated stock data

Validity Checker

Definition: Checks if a trade is valid
Responsibilities:

- Request updated stock price based on liquidity model
- Request and receive portfolio data
- Determine if sufficient funds are available for the transaction
- Send updated portfolio information if necessary
- Send queries for stock data

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 Handler

Definition: Can create and upkeep *Leagues*
Responsibilities:

- Receive initial or modified settings input for desired *League*
- Determine if settings are valid

Fund Handler

Definition: Can create and upkeep *Funds*
Responsibilities:

- Receive initial or modified settings input for desired *Funds*
- Determine if settings are valid

6.1.2 Association Definitions

The following association definitions have been updated to reflect the revamped Domain Model.

Concept Pair	**Association Description**	**Association Name**
Web Page ↔ Page Renderer	Request to visit page, sends rendered page	request page, send page
Web Page ↔ Controller	Passes the user's desired action, informs of process completion	send user request, return
Controller ↔ Page Renderer	Passes necessary data for page rendering	send page data
Controller ↔ Stock Query	Asks for data on specific stock, send data on specific stock	send stock query, return stock data
Controller ↔ Validity Checker	Requests order to be carried out, passes new portfolio data	send order, send portfolio data
Controller ↔ League Handler	Passes updated league settings, validates updated settings	verify fields, return fields
Controller ↔ Fund Handler	Passes updated fund settings, validates updated settings	verify fields, return fields
Controller ↔ Data Handler	Passes updated data, ask for portfolio data to perform process, return altered portfolio data	send updated data, request portfolio info, return portfolio info

Stock Query ↔ Stock Info Provider	Asks for stock data, return stock data	send stock data request, service stock data request
Stock Query ↔ Validity Checker	Asks for to query specific stock, return retrieved stock data	send stock query, return stock data
Validity Checker ↔ Data Handler	Asks for user's portfolio information for validity purposes, passes user's portfolio information	request portfolio data, return portfolio data
Validity Checker ↔ Liquidity Manager	Sends order information to determine adjusted price, return updated price	request adjustment, update price
Data Handler ↔ Database	Stores incoming data, request certain data, retrieve needed data	store data, request data, retrieve data

6.1.3 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 decided to not cache data, and thus we do not require an internal timer for our Stock Query concept. Thus, nearly all data is stored in a single database. In our Class Diagram, the types of data stored are shown separately. 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.
Page Renderer	sufficientRenderData	Determines if the required data to render the page is there.
Liquidity Manager	priceUpdate	Generates a new price value of the ordered stock.
League Handler	settingsValid	Determines if the User's settings input are valid for the given *League*.
Fund Handler	settingsValid	Determines if the User's settings input are valid for the given *Fund*.
Validity Checker	fieldsValid, fundsValid, tradeSuccess	Compares funds and price and checks all required fields to make sure a transaction is valid. Determines if trade is a success.

6.1.4 Traceability Matrix

Use Case	PW	User	Web Page	Page Renderer	Controller	Validity Checker	Stock Query	League Handler	Fund Handler	Liquidity Manager	Data Handler
UC01-02	17	x	x	x	x	x	x			x	x
UC03	15	x	x	x	x		x				
UC05	14	x	x	x	x		x				x
UC06	9	x	x	x	x						x
UC07-11	13	x	x	x	x			x			x
UC09-15	16	x	x	x	x				x		x

6.2 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:

- None

Postconditions:

- None

View Portfolio

Preconditions:

- Initiating investor owns the portfolio

Postconditions:

- None

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:

- Valid invitee User ID and League ID

Postconditions:

- None

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:

- User is a fund manager
- Input settings are valid

Postconditions:

- Any updated information is updated in the database

View League Standings

Preconditions:

- User has access privileges to the league
- League exists

Postconditions:

- None

6.3 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* [11] 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 $N = \left(1 + \frac{\beta r}{100s}\right)$.

7 Interaction Diagrams

7.1 Use Case 1/2: Buy/Sell Stocks

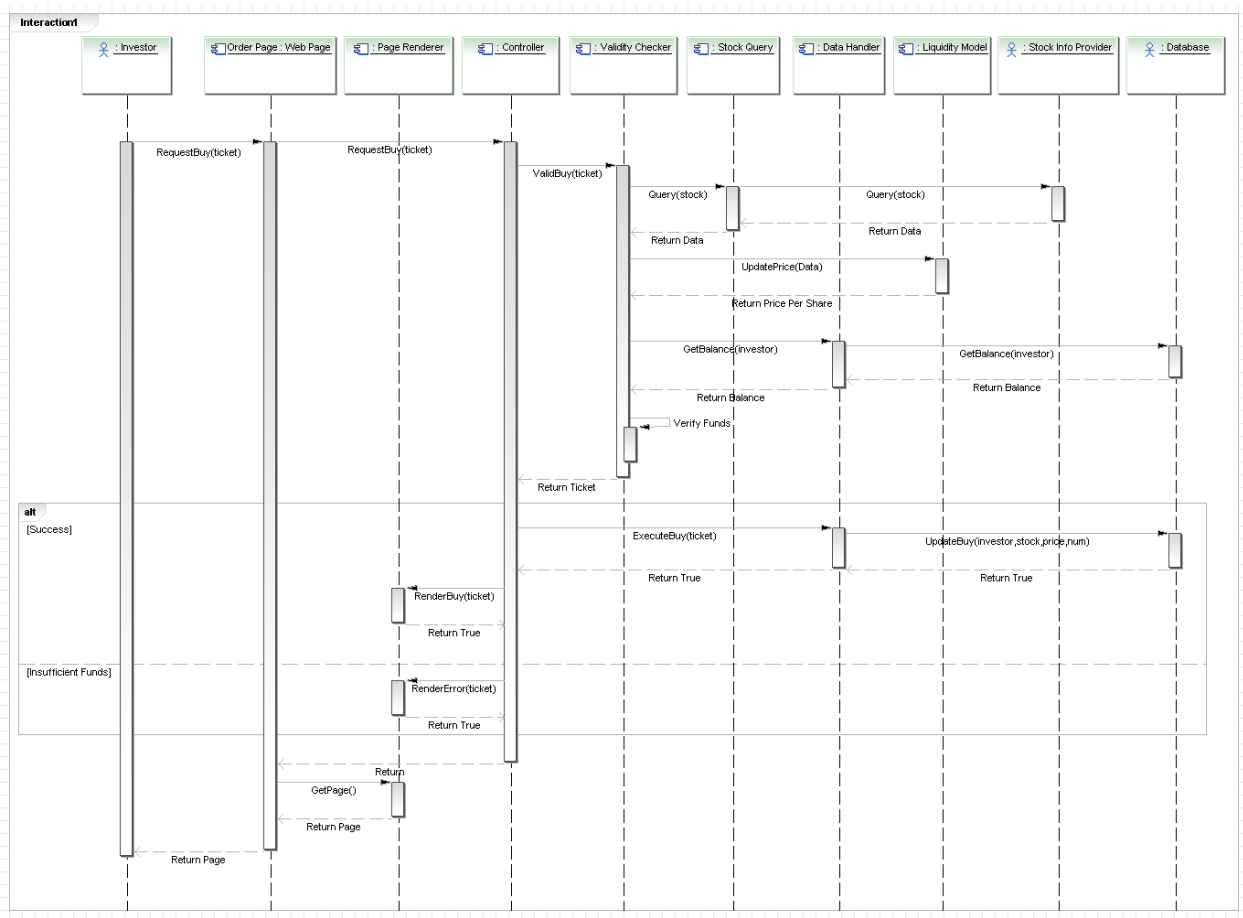


Figure 19: Interaction Diagram for Buy Stocks (From Report 1 Fig. 2)

When a buy or sell event occurs, the process begins with the Investor Actor initiating a RequestBuy to the Web Page through the web interface. The Investor must have provide a valid ticket, which includes a stock symbol and the amount of that stock that they wish to trade. The ticket also contains the user ID and the transaction type. Tickets have a price and validity field as well, but these will be populated by the Stock Query and Validity Checker respectively. The Web Page relays this information to the Controller, who's duty it is to execute the trade if possible. First, to find out if the trade is possible, the Controller sends the ticket to the Validity Checker. For buys, the Validity Checker must first determine the market price of the stock after being adjusted by the liquidity model, and second it must get the Investor's account balance and determine if there are sufficient funds to execute the transaction. If so, it returns back a ticket that is

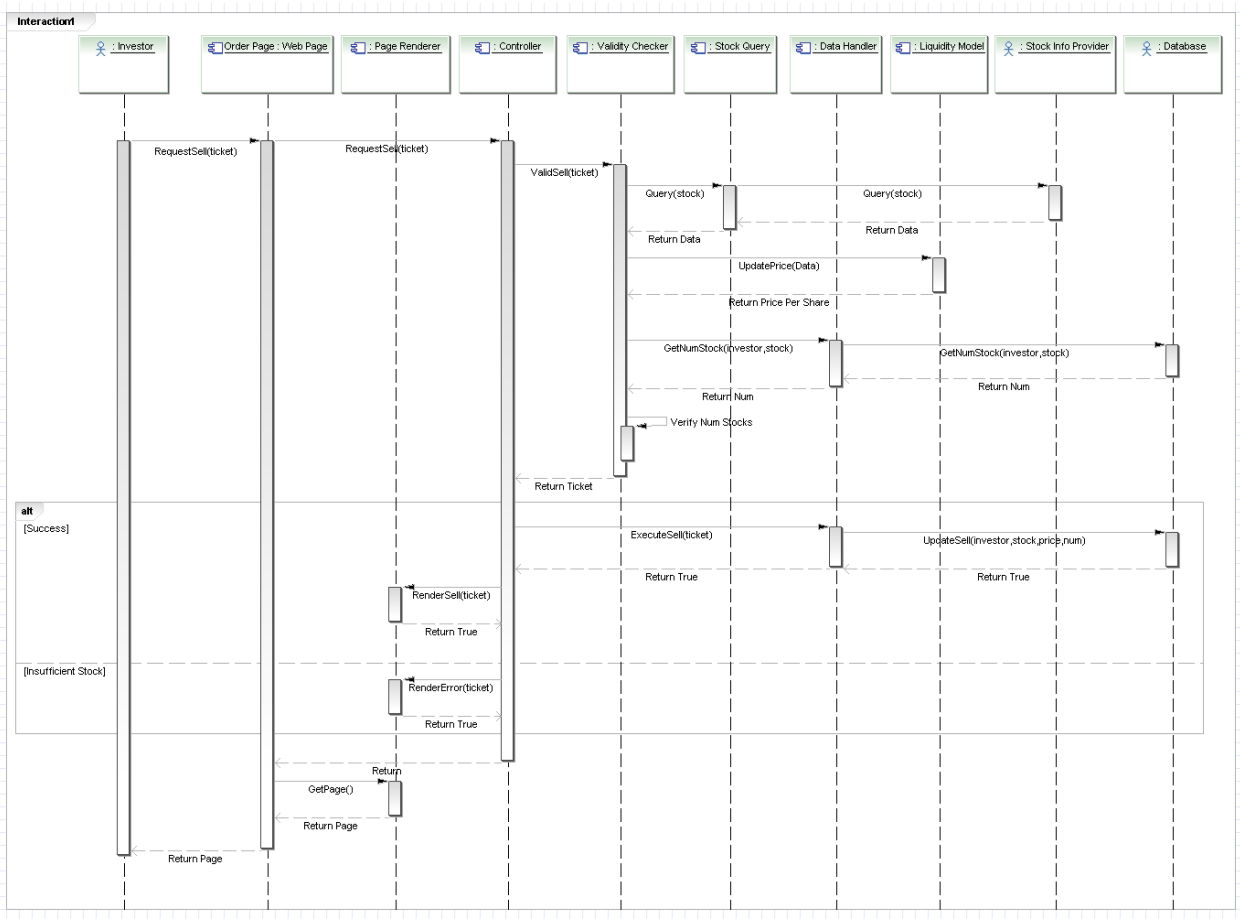


Figure 20: Interaction Diagram for Sell Stocks (From Report 1 Fig. 3)

now stamped as being valid. For sells, it must make sure that the Investor has enough of the stocks required to make the sell and enough balance to pay the commission. Validity Checker then calls Stock Query instead of querying Stock Info Provider directly. This follows the Expert Doer principle since Stock Query already has the ability to interface with the Stock Provider. Once the controller has a valid ticket, it calls the Data Handler to update the Database to reflect the new state after the transaction has been conducted. While creating the Data Handler introduces another component which is against the principle of Loose Coupling, we decided it was paramount to keep it since anytime an object needs to modify the database, it can do so through the Data Handler rather than implementing its own programming logic to communicate with the database. This is yet another example of the Expert Doer principle (and also High Cohesion). Once the Data Handler is done, the Controller notifies Page Renderer of the resulting status of the entire procedure, so that a page can be displayed accordingly to notify the user of the success or failure of their action. The Page Renderer then returns the page back to the Web Page which the Investor will see.

7.2 Use Case 3: Query Stocks

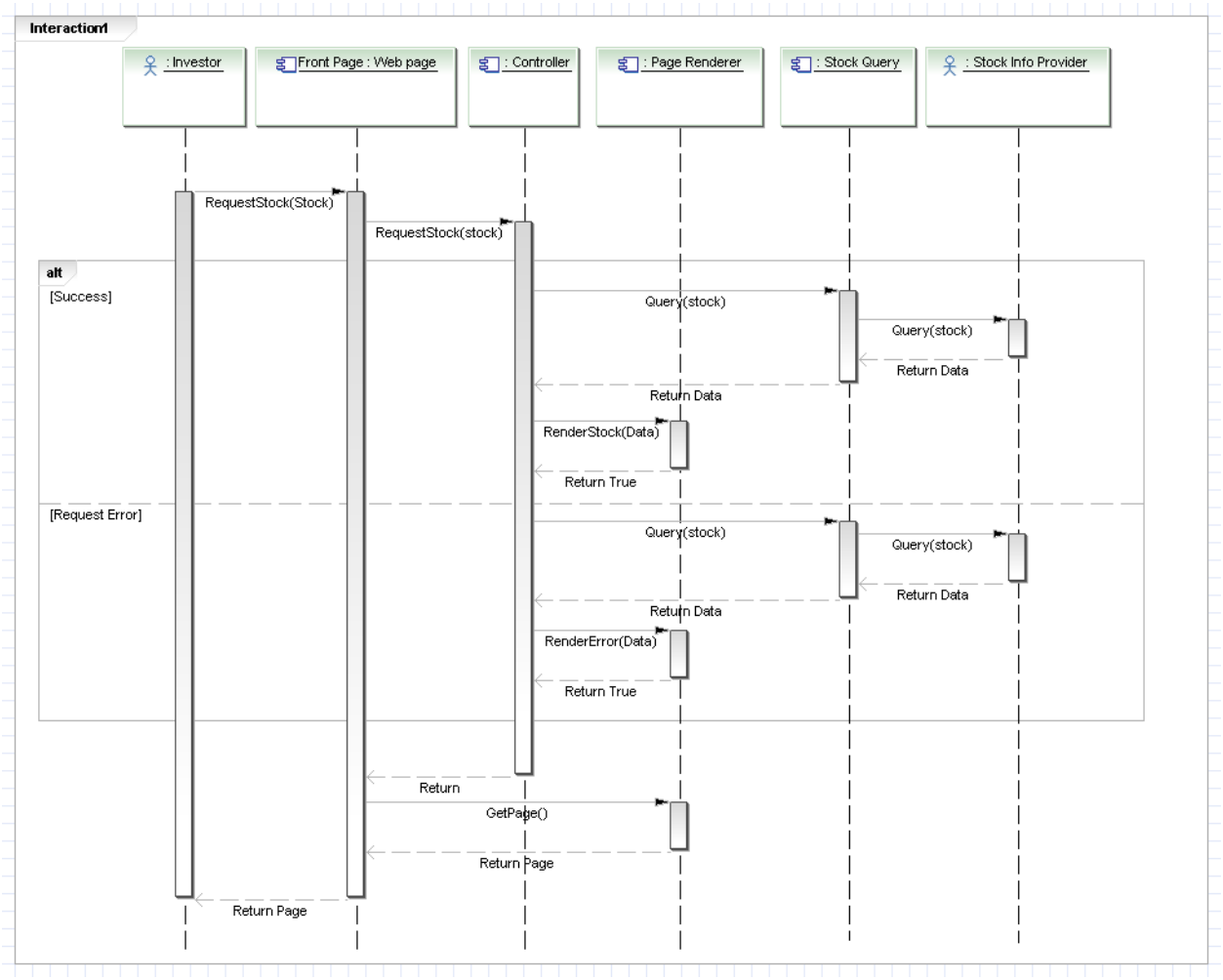


Figure 21: Interaction Diagram for Query Stocks (From Report 1 Fig. 4)

To receive information about a single stock, the Investor first chooses the stock through the Web page. The Web page then tells the Controller to fetch the stock and it's relevant information. The Controller messages Stock Query to get the state of the stock currently as provided by the Stock Info Provider. Once the Controller has this information, it sends it to the Page Renderer which formats it into HTML, and returns it to the Web page. This diagram displays the properties discussed above, mainly Expert Doer and High Cohesion.

7.3 Use Case 5: View Portfolio

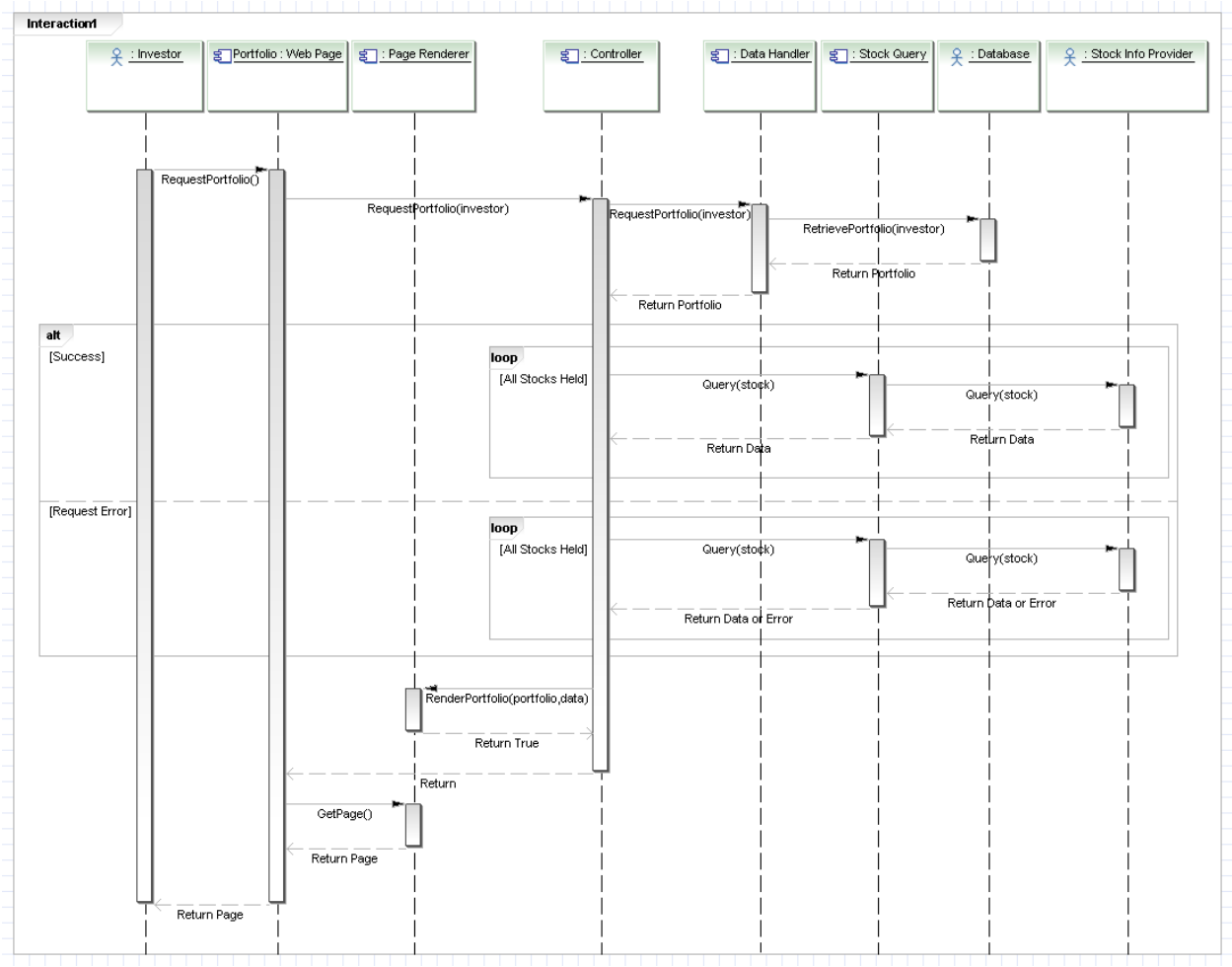


Figure 22: Interaction Diagram for View Portfolio (From Report 1 Fig. 5)

When the Investor wants to view their Portfolio, they notify the Web Page, which communicates with a Controller. The Controller queries the Data Handler to retrieve the investor's stocks. Once the controller has the list of stocks, it iterates through each of them and uses Stock Query to get their respective prices. These prices will be used to populate a data object containing the portfolio's stocks and net worth which will then be returned to the Page Render where it is embedded into HTML before being served back to the Web Page. An alternative failure case that is worth mentioning is the Stock Info Provider returning an error in response to a request for a stock's information. This error will be noted in the data object by the controller, and the Page Renderer will make note and display whatever it can without the data.

7.4 Use Case 7: Register

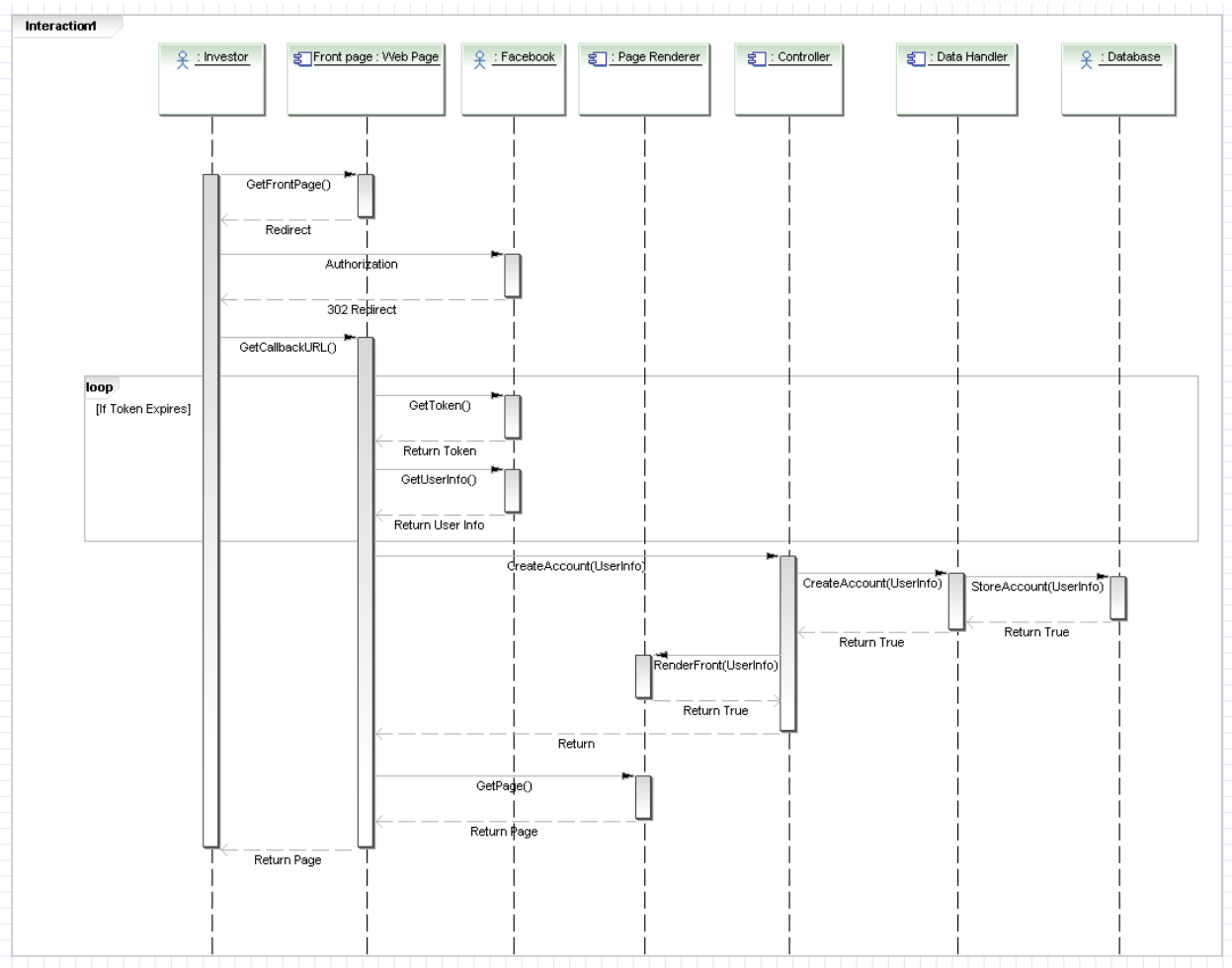


Figure 23: Interaction Diagram for Register (From Report 1 Fig. 6)

On a user's first visit to the Bears & Bulls URL, they are redirected to a page where they are given the option to authorize the app and add it. After allowing Facebook to authorize the app and access their data, the user is sent another redirect to a page that will set up their account for them. This page then creates a session by messaging Facebook and requesting a token. Once successful, it get's the user's info, namely their unique Facebook id, which it will use to create the user's account. In order to create the account, the Web Page calls a Controller that interfaces with the Data Handler (the standard method for interfacing with the database) to create a new row in the Investors table for the user. When successful, the controller is notified and the Page Renderer is told to serve a web page accordingly to get the user started.

7.5 Use Case 7/11: Create League/Fund
Use Case 9/15: Manage League/Fund

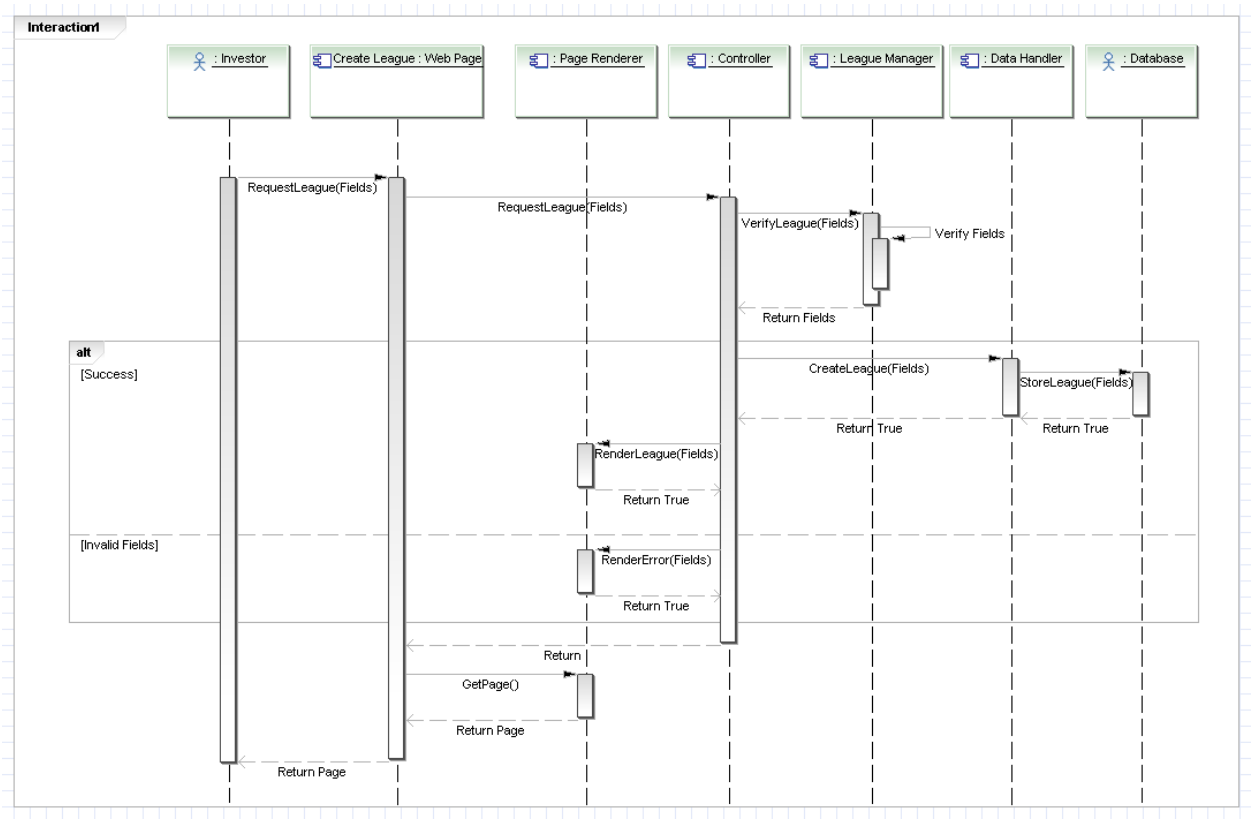


Figure 24: Interaction Diagram for Create League (From Report 1 Fig. 7)

The creation and modification of a league and a fund work essentially in the same way. The Investor initiates the action through the Web page, which hands off the task to the Controller. The Controller then communicates with the Fund Handler or the League Manager depending on what is being created or modified to see if the action is valid (an example of High Cohesion and Expert Doer). If so, the controller informs the Data Handler of what fields to update in the database to reflect the actions being carried out. The Page Renderer is told to return a page back to the Web page accordingly so the Investor can be notified. The failure case occurs when a setting is invalid, and the page renderer will display an error accordingly.

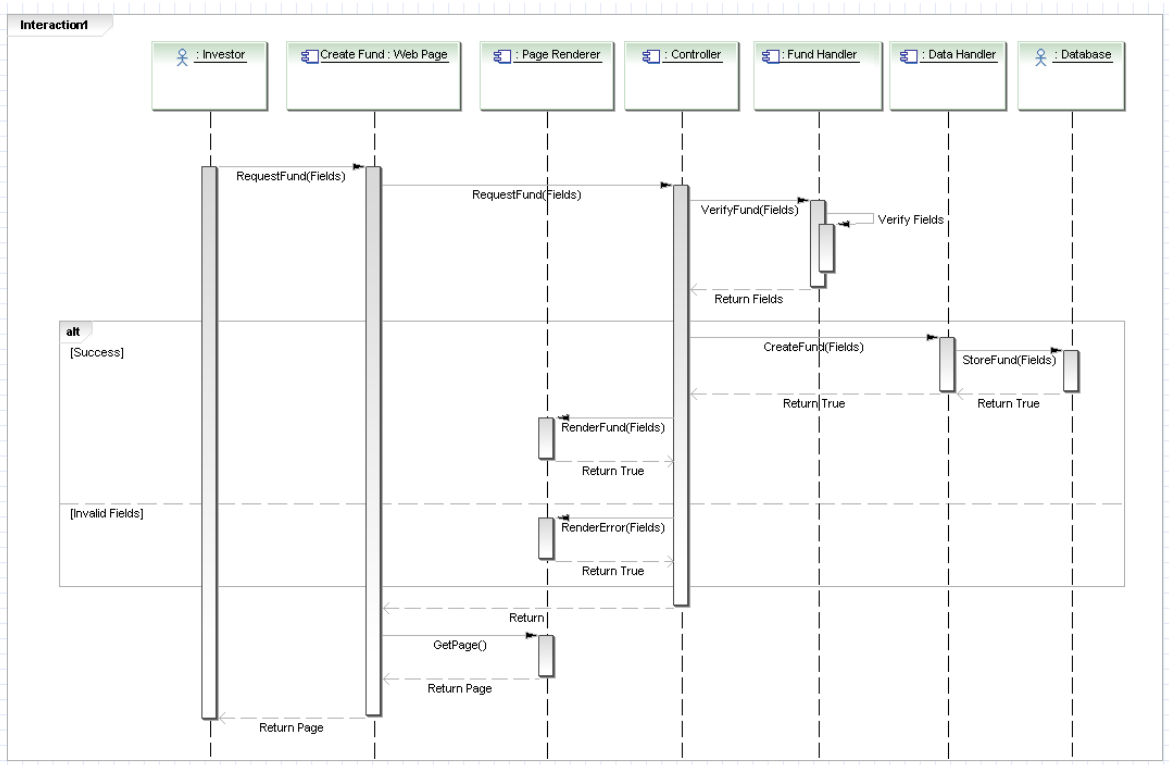


Figure 25: Interaction Diagram for Create Fund (From Report 1 Fig. 8)

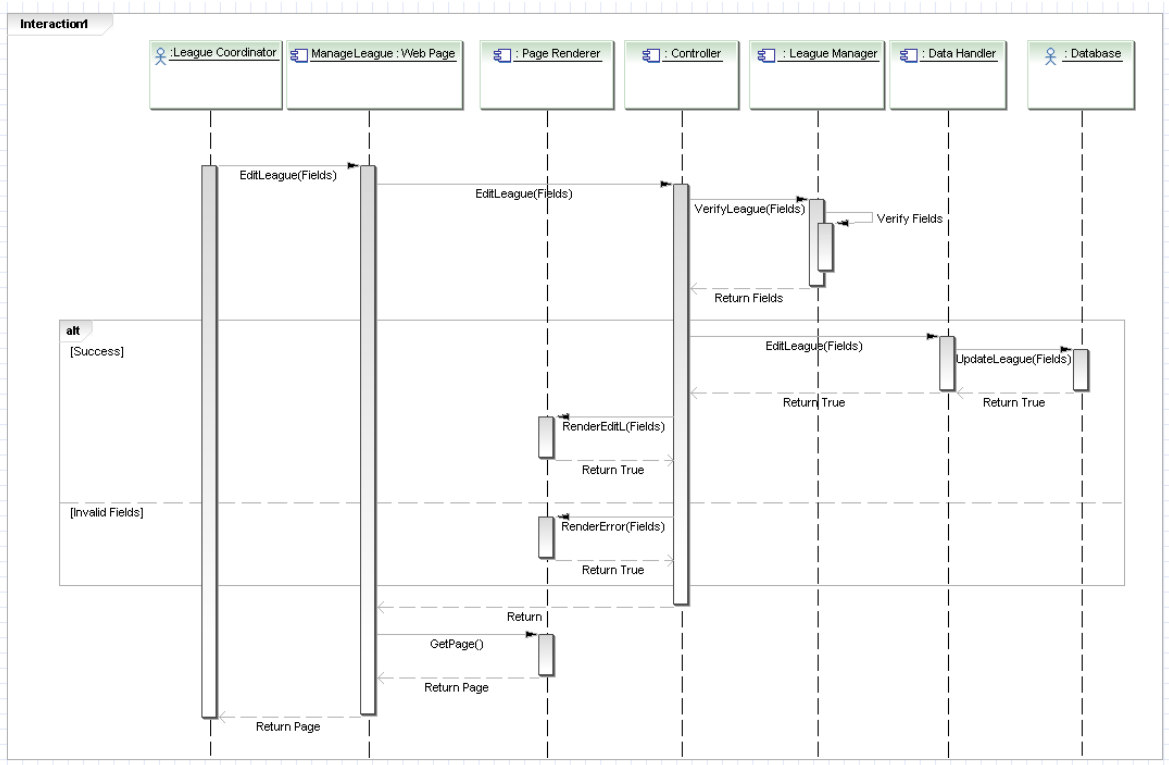


Figure 26: Interaction Diagram for Manage League (From Report 1 Fig. 9)

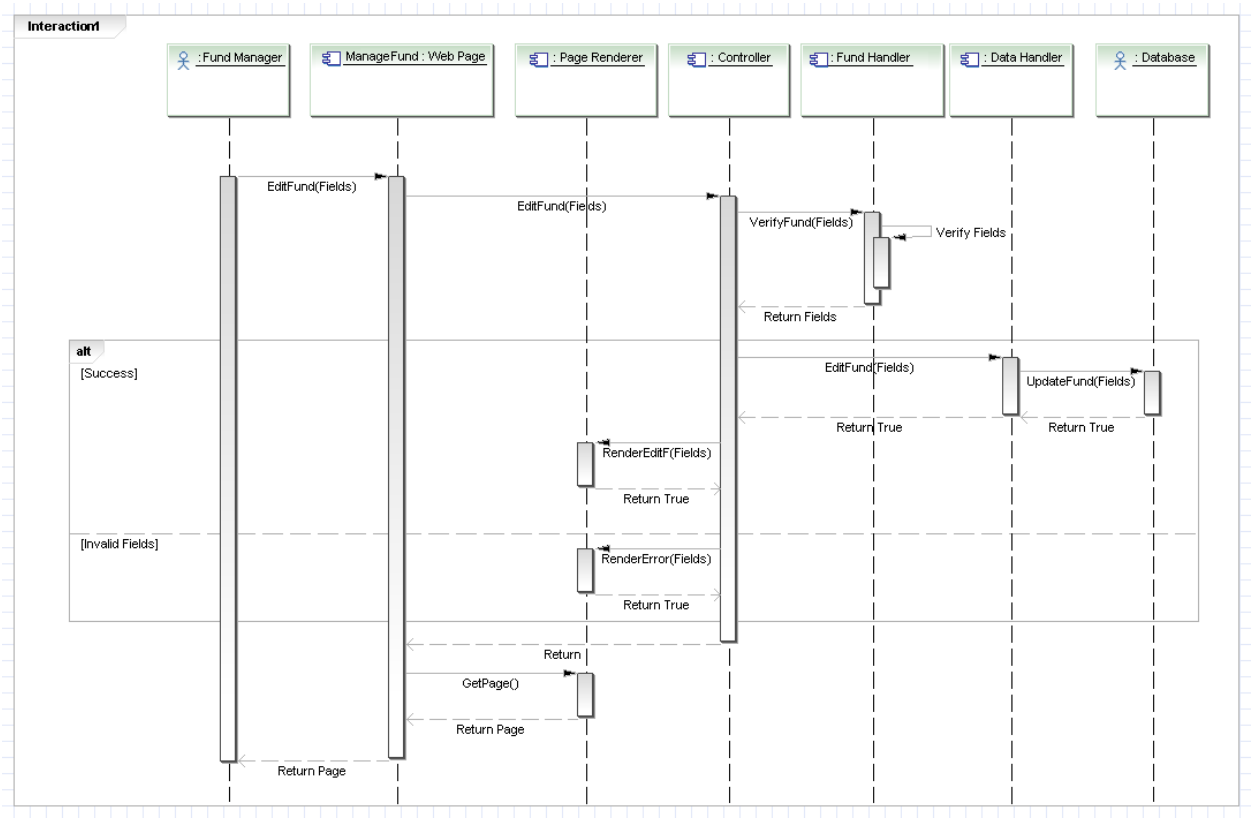
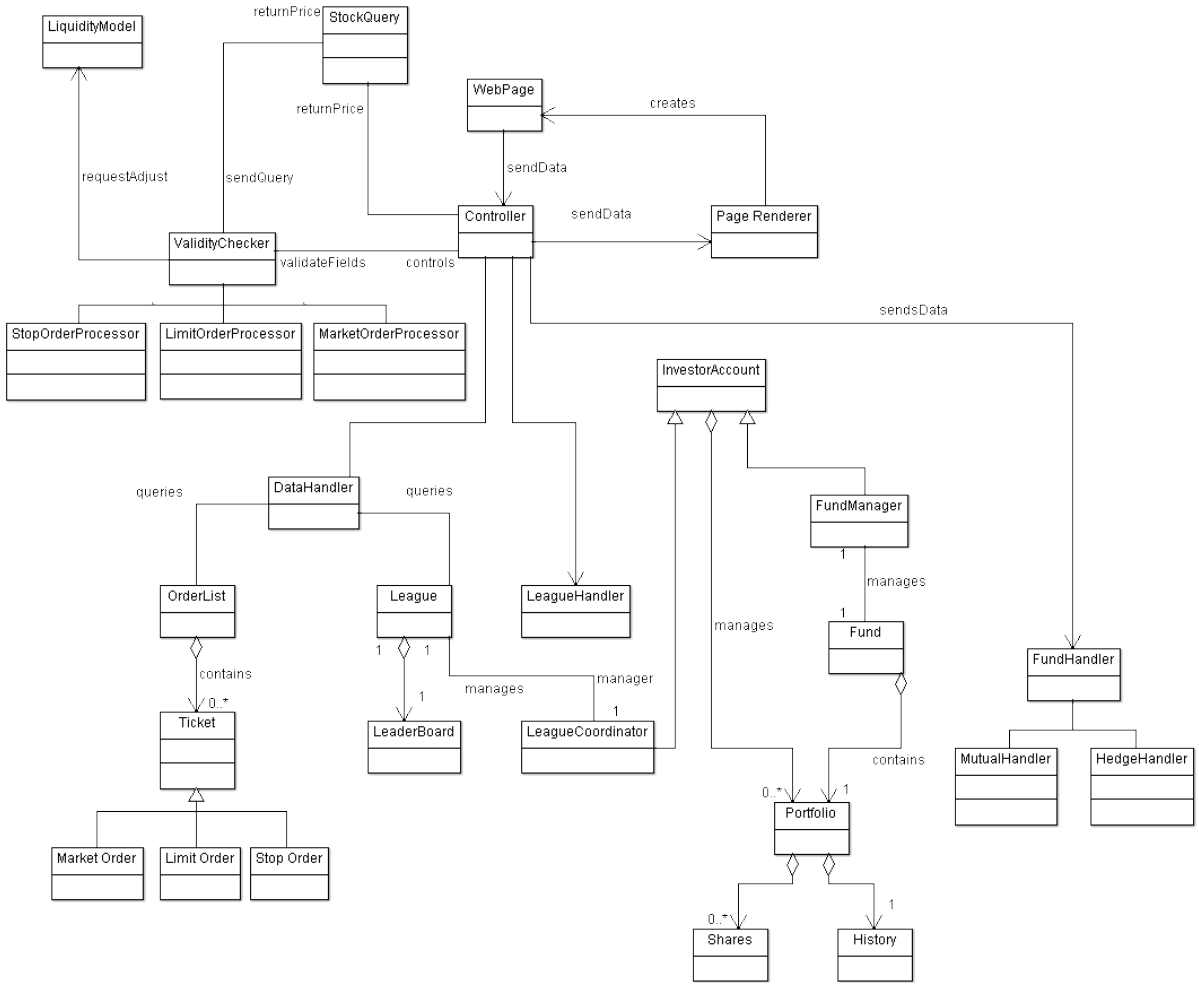


Figure 27: Interaction Diagram for Manage Fund (From Report 1 Fig. 10)

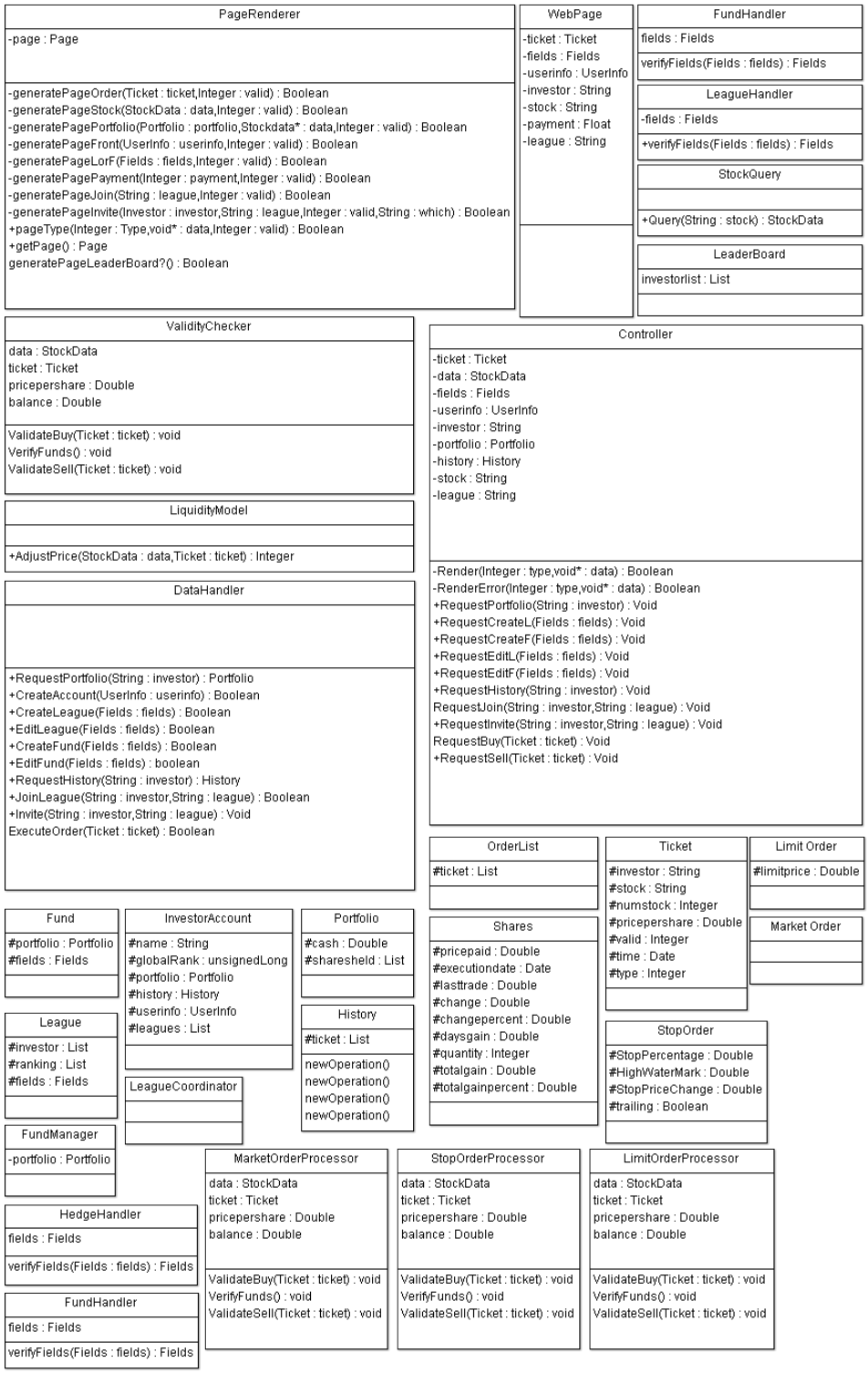
8 Class Diagram and Interface Specification

The following class diagram shows the new relations between classes. The class diagrams are collapsed to show only the names of the classes. The class attributes and methods are listed in the subsequent sections. Five new classes were introduced.

8.1 Class Diagram



8.2 Data Types and Operation Signatures



8.2.1 Controller

Attributes

The controller has the job of conveying messages back and forth between different domain concepts in the domain model. In order to accomplish this, we determined it would be best if the controller had a copy of every data type that it handles as an attribute. This lowers the chance of corrupting data.

- **ticket : Ticket**
  This is a copy of the order ticket that the investor has just submitted.
- **data : StockData**
  This is a copy of the data that the system queries from the Stock Info Provider.
- **fields : Fields**
  This is a copy of the fields that a league or fund fills out during a creation/editing request. Since the various fields are quite similar between the two, one Fields object is used for both.
- **userinfo : UserInfo**
  This is a copy of the user info that the system gets from Facebook. It is only used when an account is created, and the controller sends this to the database.
- **investor: String**
  This is a copy of the investor's username that the controller passes along to the data handler. It is used to find the Investor object from inside the database.
- **portfolio : Portfolio**
  This is a copy of a Portfolio object that the controller passes along.
- **history : History**
  This is a copy of a History object (contains the investor's transaction history) that the controller passes along.
- **stock : String**
  This is a copy of the stock symbol that is passed to the Stock Query for it to get info on the stock. Fund names are also treated as stock names because investors invest in these just like they would a stock
- **league : String**
  This is a copy of the name of a league that the controller passes along

Methods

The controller has many methods which the web page calls in order to let the controller know that it has a request (all except for Render and RenderError). The controller will subsequently convey the message by calling another function.

- **Render(Integer : type,void* : data) : Boolean**
  This method is what the controller calls when it is ready to render a page. The arguments are an Integer for the type of page that is displayed, and a pointer to a data structure containing the data necessary to construct the page. This method calls the appropriate method within page renderer.
- **RenderError(Integer : type,void* : data) : Boolean**
  This method serves the same purpose as the one above, except that it tells the page renderer to render an error version of the page.
+ **RequestBuy(Ticket : ticket)**
  This method is the method that the web page calls in order to request a buy
+ **RequestSell(Ticket : ticket)**
  This method is the method that the web page calls in order to request a sell
+ **RequestPortfolio(String : investor) : Void**
  This method is the method that the web page calls in order to view a portfolio
+ **RequestCreateL(Fields : fields) : Void**
  This method is the method that the web page calls in order to create a league
+ **RequestCreateF(Fields : fields) : Void**
  This method is the method that the web page calls in order to create a fund
+ **RequestEditL(Fields : fields) : Void**
  This method is the method that the web page calls in order to edit league settings.
+ **RequestEditF(Fields : fields) : Void**
  This method is the method that the web page calls in order to edit fund settings.
+ **RequestHistory(String : investor) : Void**
  This method is the method that the web page calls in order to view transaction history.
+ **RequestJoin(String : investor, String : league) : Void**
  This method is the method that the web page calls in order to join a league.
+ **RequestInvite(String : investor, String : league) : Void**
  This method is the method that the web page calls in order to invite an investor to a league.
+ **RequestAddCoord(String : investor) : Void**
  This method is the method that the web page calls in order to add a coordinator to a league.

8.2.2 PageRenderer

Attributes

−**page : Page** This is the current page that the web browser is displaying/will be displayed.

Methods

Every method has an integer parameter called valid. This lets the page renderer know if the page that it should be generating is an error page or a success page.

− **generatePageOrder(Ticket : ticket, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of an order.

− **generatePageStock(StockData : data, Integer : valid) : Boolean**
This method is called in order to render a page displaying the resultsof a stock data query.

− **generatePagePortfolio(Portfolio : portfolio, Stockdata* : data, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of a portfolio viewing.

− **generatePageFront(UserInfo : userinfo, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of an account creation.

− **generatePageLorF(Fields : fields, integer : valid) : Boolean**
This method is called in order to render a page displaying the results of a creation of a fund or league, or an editing of a fund or league.

− **generatePageJoin(String : league, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of joining a league.

− **generatePageInvite(Investor : investor, String : league, Integer : valid, String : which) : Boolean**
This method is called in order to render a page displaying the results of inviting an investor to a league.

+ **pageType(Integer : Type, void*: data, Integer : valid) : Boolean**
This method is called by the controller in order to render a page with the given type, data, and whether it is an error or not.

+ **getPage() : Page**
This method is called by the web page in order to retrieve the page it must display.

8.2.3 DataHandler

Methods

These methods are called by the controller to access the information in the database.

+ **ExecuteOrder(Ticket : ticket) : Boolean**
  This method executes the ticket order by updating the investor's portfolio accordingly.
+ **RequestPortfolio(String : investor) : Portfolio**
  This method is called to request the portfolio data from the database.
+ **CreateAccount(UserInfo : userinfo) : Boolean**
  This method is called to request an account creation.
+ **CreateLeague(Fields : fields) : Boolean**
  This method is called to request a league creation.
+ **EditLeague(Fields : fields) : Boolean**
  This method is called to request the league settings be modified in the database.
+ **CreateFund(Fields : fields) : Boolean**
  This method is called to request a fund creation.
+ **EditFund(Fields : fields) : boolean**
  This method is called to request the fund settings be modified in the database.
+ **RequestHistory(String : investor) : History**
  This method is called to request the transaction history from the database.
+ **JoinLeague(String : investor, String : league) : Boolean**
  This method is called to request that an investor be added to a league in the database.
+ **Invite(String : investor, String : league) : Void**
  This method is called to request that an invite be added to the investor's account.

8.2.4 StockQuery

Methods

+ **Query(String : stock) : StockData**
  This method is called to request stock data from the stock info provider. The data is forwarded straight to the class requesting it, and a copy is not made within the Stock Query.

8.2.5 ValidityChecker

Attributes

The validity checker holds the below attributes that it uses in calculations to determine if an order is valid or not. The validity checker is an interface class that is implemented by the order processors MarketOrderProcessor, StopOrderProcessor and LimitOrderProcessor. These classes are omitted for brevity but implement the methods listed below in similar manners.

- **data : Stockdata**
  This is a copy of the stock data obtained from Stock Query.
- **ticket : Ticket**
  This is a copy of the order ticket that the investor fills out.
- **pricepershare : Double**
  This is a copy of the price per share of the stock, which the liquidity model determines.
- **balance : Double**
  This is a copy of the investor's current account balance.

Methods

+ **ValidateBuy(Ticket : ticket) : void**
  This method is called by the controller to determine if a buy is valid or not.
- **VerifyFunds() : void**
  This method is called by the validity checker in order to determine if the investor has sufficient funds for the transaction.
+ **ValidateSell(Ticket : ticket) : void**
  This method is called by the controller to determine if a sell is valid or not.

8.2.6 LiquidityModel

Methods

+ **AdjustPrice(StockData : data, Ticket : ticket) : Integer**
  This method is called by the validity checker to modify the stock price per share in accordance to how many the investor plans to buy or sell.

8.2.7 WebPage

Attributes

The web page contains a copy of various attributes that it receives from the investor and forwards it on to the controller.

− **ticket : Ticket**
  This is a copy of an order ticket that the investor fills out.
− **fields : Fields**
  This is a copy of the league or fund settings that the investor fills out.
− **userinfo : Userinfo**
  This is a copy of the user info that facebook provides to the system.
− **investor : String**
  This is a copy of the investor's username.
− **stock : String**
  This is a copy of the particular stock that is requested by the investor.
− **league : String**
  This is the name of the league that the investor enters.

8.2.8 FundHandler

The FundHandler is an interface for MutualHandler and HedgeHandler. The description of the individual classes are omitted for brevity but the methods they implement follow the description below. Attributes

− **fields : Fields**
  This is a copy of the fields for the fund.

Methods

+ **verifyFields(Fields : fields) : Fields**
  This method that the controller calls that verifies that the settings for the fund are all valid.

8.2.9 Leaderboard

Attributes

− **investorlist : List**
  This is a list of the top investors ordered by rank.

8.2.10 LeagueHandler

Attributes

− **fields : Fields**
This is a copy of the fields for the league.

Methods

+ **verifyFields(Fields : fields) : Fields**
This method that the controller calls that verifies that the settings for the league are all valid.

8.2.11 Ticket

Attributes

investor : String
This is the investor's username.
stock : String
This is the stock symbol.
numstock : Integer
This is the amount of stock that is being exchanged.
pricepershare : Double
This is the price per share of the stock.
valid : Integer
This is a valid bit: it lets the controller know if the ticket is valid or not.
time : Date
This is the time and date of the ticket submission.
type : Integer
This is the type of transaction (example being stop order).

8.2.12 Shares

This class contains the number of shares of a stock that an investor owns, and information about them. Attributes

pricepaid : Double
This is the price paid for the stock.
executiondate : Date
This is the date of execution of the trade.

lasttrade : Double
This is the price of the lastest trade on the market for the stock.
change : Double
This is the change in the stock from the beginning of the day.
changepercent : Double
This is the percentage change in the stock from the beginning of the day.
daysgain : Double
This is the gain from the stock in the current day.
quantity : Integer
This is the amount of stock that is owned.
totalgain : Double
This is the total gain from the stock from when it was first bought.
totalgainpercent : Double
This is the percentage gain from the stock out of the gains from all stocks the investor holds.

8.2.13 Portfolio

Attributes

cash : Double
This is the investor's balance.
sharesheld : List
This is a list of class shares that the investor owns.

8.2.14 StopOrder

Attributes

StopPercentage : Double
This is the threshold percent change of the stock before the order is executed.
HighWaterMark : Double
This is the highest price reached (or lowest for a buy). This is used for trailing orders.
StopPriceChange : Double
This is the threshold change in price of the stock before the order is executed.
trailing : Boolean
This specifies if the stop order is a trailing stop or not.

8.2.15 LimitOrder

Attributes

limitprice : Double
This is the threshold price for a stock before the order is executed.

8.2.16 MarketOrder

This class is the default order type and has no special requirements. Thus it is represented here only to remind the developer that the market order exists.

8.2.17 OrderList

Attributes

ticket : List
This is a list of tickets that have yet to be executed because conditions for execution have not been met.

8.2.18 History

Attributes

ticket : List
This is a list of class tickets in chronologically backwards order, with the most recent transaction first.

8.2.19 FundManager

Attributes

− **portfolio : Portfolio**
This is the portfolio of the fund, which the fund manager maintains.

8.2.20 LeagueCoordinator

The league coordinator does not have any special attributes or methods that make it different from an investor. This class exists to differentiate an investor from a league coordinator (who is able to call more functions). This class inherits from InvestorAccount

8.2.21 InvestorAccount

Attributes

name : String
This is the username of the investor.
globalRank : unsignedLong
his is the global rank of the investor.
portfolio : Portfolio
This is the investor's portfolio.
history : History
This is the investor's transaction history.
userinfo : UserInfo
This is the investor's personal info that was retrieved from Facebook.
leagues : List
This is the list of leagues that the investor is currently a member of.

8.2.22 Fund

Attributes

portfolio : Portfolio
This is the fund's portfolio.
fields : Fields
This is the various settings of the fund, including fund name.

8.2.23 League

Attributes

investor : List
This is the list of investors that are currently in the league.
ranking : List
This is the list of rankings for each investor (it runs parallel to the investor list).
fields : Fields
This is the various settings of the league, including the league name.

8.3 Traceability Matrix

Class	WebPage	PageRenderer	ValidityChecker	StockQuery	DataHandler	LiquidityManager	LeagueHandler	FundHandler
WebPage	x							
PageRenderer		x						
Controller	x	x						
ValidityChecker			x					
LiquidityModel						x		
StockQuery				x				
DataHandler					x			
FundHandler								x
LeagueHandler							x	
League					x			
LeaderBoard					x			
LeagueCoordinator					x			
InvestorAccount					x			
Portfolio					x			
FundHandler					x			
Fund					x			
Shares					x			
History					x			
OrderList					x			
Ticket					x			
Market Order					x			
Limit Order					x			
Stop Order					x			
MarketOrderProcessor			x					
LimitOrderProcessor			x					
StopOrderProcessor			x					
HedgeManager					x			
MutualManager					x			

Many of the classes map back to the DataHandler concept since they contain data that is queried by the DataHandler. These classes were represented by a single database in the domain model, but shown as separate entities in the class diagram to give more insight on the inner workings and details of our program.

8.4 Design Patterns

Looking back at our previous interaction diagrams, we found that we had unknowingly implemented some design patterns. We will now make it clear what design patterns were implemented in the interaction diagrams.

8.4.1 Command Pattern

The command pattern is implemented in almost every interaction diagram, and manifests itself in the form of our class called "controller". Our controller class is not the same as a traditional controller class, but is actually an interface for many different commands. As you can see through the interaction diagrams, the user calls the controller interface in order to carry out any action. The concrete commands are carried out by classes such as validity checker, stock query, liquidity model, fund handler, and league manager. Many of these actions are logged by the system because the controller makes a call to update the database in most cases. The controller knows the receiver of each action request, and knows the appropriate sequence of calls to make based on a given request. Perhaps a more make appropriate name for the controller would have been "processor" because it is like the central processor within our system. It provides an interface for all the commands in our system, and has the actual commands implemented by various classes derived from it.

8.4.2 Strategy Pattern

The strategy pattern is also implemented in our design. Validity checker has stoporderProcessor, limitorderProcessor, and marketorderProcessor as its three different strategies. During run time, depending on whichever order is chosen, the appropriate one of the three classes is called to handle the request. Similarly, fundHandler has mutualHandler and hedgeHandler as its different strategies. This is not shown in the interaction diagrams in the interest of readability. However, the interfaces (validity checker and fund manager) are both shown in the interaction diagrams, and it is assumed that the appropriate class is called.

8.4.3 Uses of Design Patterns

The use of these design patterns have greatly benefited in the design of the project. In particular, the command pattern greatly simplifies our interaction diagrams and gives us the ability to log transactions within our systems as well as undo certain transactions. Also, it allows us to carry out the actual execution of an order at a different time from when it was submitted. However, perhaps the best part of it is that it easily allows us to add new functionality to our system. As an example of its benefits, consider the interaction diagrams. If the command pattern was not used, then the Web page would have to know where each command had to go, and thus its role as a doing object would be invalid. Also, the web page would have too many roles, and it is much more advantageous to have a dedicated interface for these functions. The strategy pattern is invaluable for the working of this system. It cuts out a lot of conditional logic, and allows for a dedicated class for each type of ticket and fund. In a system that could potentially use a lot of conditional logic, this strategy pattern is very useful. There are no concrete examples to show using the interaction diagrams because it is internal if statements, but one can appreciate the elegance of a strategy method.

8.5 Object Contraint Language

context Controller::RequestPortfolio(string : investor) void
pre: (investor → InvestorAccount.portfolio = this.portfolio)
- You can only view your own portfolios

context Controller::RequestEditL(Fields : fields) void
pre: (InvestorAccount→LeagueCoordinator = true)
- You can only edit a league if you are the league coordinator

ontext Controller::RequestEditF(Fields : fields) void
pre: (InvestorAccount→FundManager = true)
- You can only edit a fund if you are the fund manager

context Controller::RequestInvite(String : investor, String : league) : Void
pre: (InvestorAccount→LeagueCoordinator = true)
- You can only invite people to a league if you are the league coordinator

context DataHandler::ExecuteOrder(Ticket : ticket) : Boolean
pre: (ValidateSell())
post: (InvestorAccount.Update())
- The Investor's portfolio must be updated to accommodate bought/sold stocks

context DataHandler::CreateAccount(UserInfo : userinfo) : Boolean
post: (hasPortfolio = true AND inGlobalLeague = 1)
- The investor will have a portfolio for the Global Public League upon registration

context DataHandler::CreateLeague(Fields : fields) : Boolean
post: (league→name = field:League_Name AND league → this.member AND update())
- The league will be stored in our database (update) , and the league coordinator will have a portfolio for that league.

context DataHandler::CreateFund(Fields : fields) : Boolean
post: (fund→name = field:Fund_Name AND fund → this.member AND update())
- The find will be stored in our database (update), and the fund manager will have a portfolio for that fund

context DataHandler::EditLeague(Fields : fields) : Boolean
post: (league→settings.update(fields))
- League settings will be updated in the database

context DataHandler::EditFund(Fields : fields) : Boolean
post: (fund→settings.update(fields))
- Fund settings will be updated in the database

context DataHandler::JoinLeague(String : investor, String : league) : Boolean
post: (league→this.member AND update())
- The User will now have a portfolio for the league

context ValidityChecker inv:
if(League)
self.balance $\geq$ 0
- The User will not have a negative balance

context ValidityChecker::ValidateBuy(Ticket : ticket)
pre: (ticket$\rightarrow$fields.isValid())
post: ValidityChecker::VerifyFunds is called
- The fields of the order form must be valid for the specified order
- Upon validation, the amount of funds compared to the price must be checked next

context ValidityChecker::ValidateSell(Ticket : ticket)
pre: (ValidateBuy() AND VerifyFunds())
post: DataHandler::ExecuteOrder is called
- The fields of the order form must be confirmed by Validate Buy
- The request to update the database must be called

context ValidityChecker::VerifyFunds inv:
InvestorAccount.portfolio.cash $\geq$ for(I = stock; I $<$ stocknum; i++)
cash += ticket[i].pricepershare*numstock
- The User must have more funds than the cost of the order, or an error is returned

context ValidityChecker::VerifyFunds() : void
pre: (ValidateBuy())
post: DataHandler::ExecuteOrder is called
- The request to update the database must be called

context LiquidityModel inv:
if(ticket.type=buy)
ticket.price $\leq$ updatedPrice
else if(ticket.type=sell)
ticket.price $\geq$ updatedPrice
- If our liquidity model is being applied to a buy, the updated price cannot be lower than the original price. If dealing with a sell, the updated price cannot be higher than the original price.

context FundHandler::verifyFields inv:
self→fields.isValid()
- The fields filled out for fund settings must be valid, or an error is returned

context LeagueHandler::verifyFields inv:
self→fields.isValid()
- The league filled out for fund settings must be valid, or an error is returned

context Ticket inv:
self.numstock > 0
-A ticket can only exist for at least one share of a stock, as orders must include at least one share

context Ticket inv:
pricepershare > 0
- The price of a share is always greater than zero

context Shares inv:
pricepaid > 0
- The price of a share is always greater than zero

context Shares inv:
lasttrade→pricepaid > 0
- The price of a share is always greater than zero

context Shares inv:
changepercent ≥ -100
- Value of a stock can never go below zero, so the percent change will never be less than -100%

context Shares inv:
quantity > 0
- If there were no shares of the stock, it would not be kept track of

context Shares inv:
totalgainpercent ≥ -100
- Value of a stock can never go below zero, so the percent change will never be less than -100%

context Portfolio inv:

if(Portfolio→type!=HedgeFund)

funds ≥ 0

- A portfolio can never have negative funds, unless it is a hedge fund (it can briefly have negative funds, which it would quickly gain back from the sale of stock)

context StopOrder inv:

self.StopPriceChange < self.HighWaterMark

- The stop price change needs to be less than the high water mark

context StopOrder inv:

self.StopPriceChange > 0

- Stop price change cannot be zero or less than zero

context StopOrder inv:

(self.StopPercentage > 0) AND (self.StopPercentage < 100)

- Stop percentage has to be a valid nuber between 0 and 100

context LimitOrder inv:

self.limitprice > 0

- User cannot purchase a stock at a price of zero

9 System Architecture and System Design

9.1 Architectural Styles

Bears & Bulls utilizes several architectural styles with a main focus on the Model/View/Controller approach. Let us take a closer examination into how Bears & Bulls incorporates these various techniques in its implementation.

9.1.1 Model/View/Controller

Bears & Bulls relies heavily on the Model/View/Controller architecture. The main view is the Facebook web interface that the user interacts with. Through this interface the user carries out various tasks as enumerated by our Use Cases. Various controllers will help the user interface with the two main models which are the site database and the stocks model provided by the stock information provider. The view will be represented by HTML, CSS, and Javascript. The controller logic will be implemented using PHP. For the models, the site database will be created using MySQL and the stock model will be made accessible by API calls to an external stock information provider. Most of our concepts fall into the controller category.

9.1.2 Front and Back Ends

The front-end component of our system is our Web UI. This is what the public will see. The back end consists of all the behind the scenes business logic for our app. Even within our controller and model logic, we have representations of front and back ends. For example, for the controller to communicate with the database, it must do so through the DataHandler. Hence, the DataHandler serves as the front end of the database to the controller.

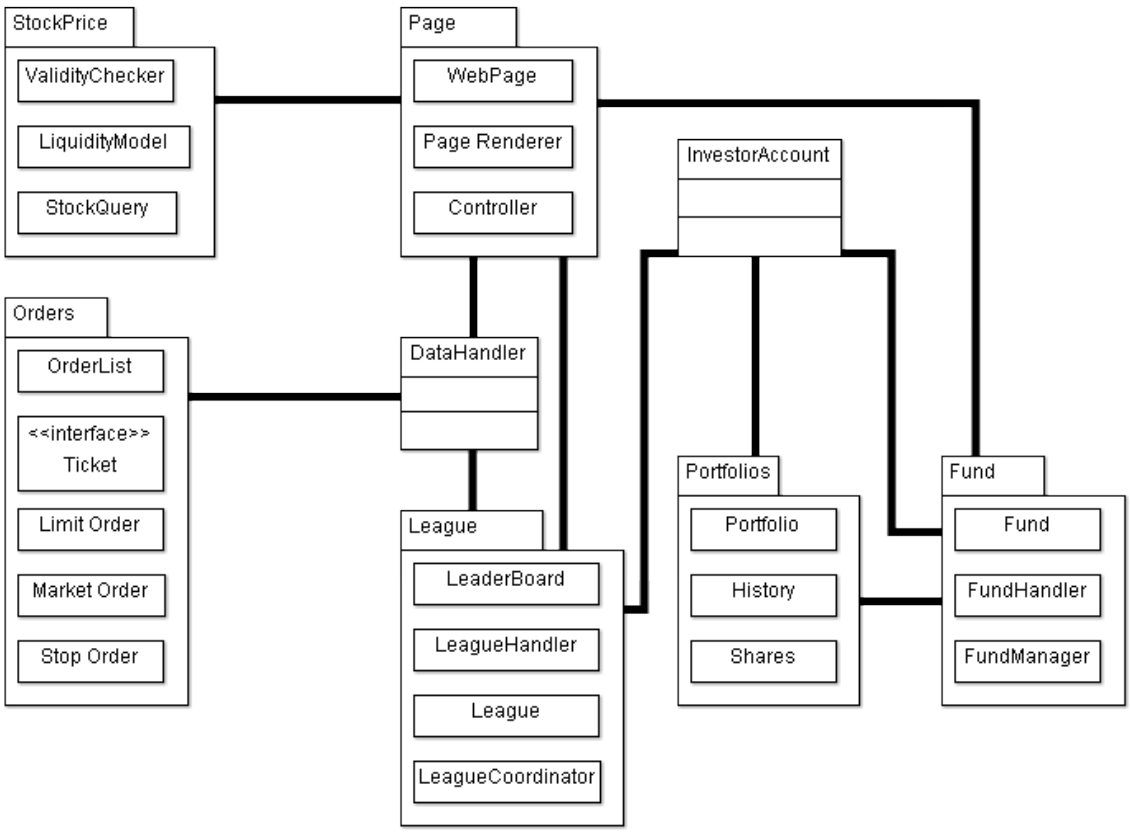
9.1.3 Event-driven Architecture

Any changes to the equilibrium of our system by the user is an event. In this way, the user acts as an event emitter (i.e. initiating buy, sells, creating leagues, etc.). The events are handled by the controller logic, which serves as the event consumer for these events. Another type of event that drives our application are changes in stock price. This is used to execute limit, stop, and stop limit orders.

9.1.4 Object-oriented

Our application uses some object-oriented practices. For example things such as leagues, funds, tickets, transactions, and stocks are all represented as objects. These objects are the lifeblood of our system because all data communication occurs through these objects.

9.2 Identifying Subsystems



Page: (WebPage, PageRenderer, Controller)

Page is the subsystem that directly interacts with the user actor. It is responsible for handling the user's input and relaying to the other subsystems.

League: (League, LeagueCoordinator, LeaderBoard, LeagueHandler)

This subsystem takes care of all things associated with a league, including it's creation and maintenance, as well as displaying information about the league and its players, such as the leader board.

Portfolio: (Portfolio, History, Shares)

This subsystem keeps track of an investor's portfolio, including it's history and content.

Fund: (Fund, FundManager)

This subsystem takes care of all things associated with a fund, including it's creation and maintenance as well as displaying information about the fund and its content.

Orders: (OrderList, Ticket, MarketOrder, LimitOrder, StopOrder)

This subsystem manages all transactions initiated by the investor. Orders is a subsystem that handles all transaction initiated by the investor, such as limit and stop orders, for the system. It keeps track of such orders by creating tickets.

StockPrice: (StockQuery, ValidityChecker, LiquidityManager)

StockPrice's responsibility is to get updated stock prices and alter them based on liquidity, as well as validate transaction based on available cash balance.

9.3 Mapping Subsystems to Hardware

The mapping of subsystems takes place onto two servers. One server, Heroku, was provided by Facebook Developers. Heroku is a cloud application platform which supports any programming language. We manage our app via the Heroku command-line tool and deploy out code via the Git revision control system. All system administrators have access to this account and can submit changes at any time. The capabilities allow PHP and MySQL which will be utilized to display the user interface. Processes are first initiated by the Web Browser when the user requests an action to occur. The DataHandler, Controller, Stock Query, and Page Renderer will all be managed via the Heroku server. For capabilities stored on the Heroku server, the information is stored in the cloud provided by Heroku. Originally we utilized both Heroku and a software engineering computer (sweng-1.engr.rutgers.edu/ group6), but we were able to incorporate all the capabilities via Heroku, so the usage of sweng-1 was eliminated.

9.4 Persistent Data Storage

Bears & Bulls needs to store data that will outlast a single execution of the system in order to keep track of player profiles, stocks and net worth. For each player, the database will store the user's name, cash balances, current stock information and a history of past transactions. A players cash balances is the amount of money not tied up in current stocks. For current stocks, the database will record the stock symbol, quantity of stock, purchase price of the stock, date and time of original transaction, and the price of stock at last update. Updates occur when the portfolio is viewed or when the system updates the current standings of a league. With all this information, the system can calculate a player's net worth, which is his cash balances plus the total value of his current stocks based on the most recently updated prices. The database will also hold a record of past transactions and stocks owned, including the prices that the stocks were bought and sold at, as well as times and dates of all transactions. The database will be implemented using MySQL.

Name: Noah Silow-Carroll					
Cash: $6,435.00					
Market Value: $36450.00					
Stocks					
Symbol	**Qty**	**Price Paid**	**Date Bought**	**Last Trade**	**Day's Gain**
GOOG	50	610.31	2/24/12	618.15	+2.37
YHOO	-100	14.48	2/27/12	14.91	-0.12
F	50	12.20	2/27/12	11.97	-0.03
Transaction History					
Symbol	**Transaction Type**		**Price**	**Quantity**	**Date**
YHOO	Sell Short		14.48	100	2/27/12
F	Buy		12.20	50	2/27/12
F	Sell		34.83	100	2/24/12
GOOG	Buy		610.31	50	2/24/12

9.5 Network Protocol

Bears & Bulls utilizes Facebook for its operation. This system uses an IFrame Canvas application which is an IFrame surrounded by the Facebook chrome. Since the application is essentially a website wrapped in Facebook's application environment, the entire system communicates via HTTP.

The Facebook Platform uses OAuth 2.0 protocol for authentication and authorization. If the user is already logged into Facebook, the system will validate the login cookie stored on the users browser which authenticates the user. If the user is not logged in, they are prompted to enter their login information. Since authentication is handled by Facebook, Bears & Bulls will not do additional user authentication.

9.6 Global Control Flow

Bears & Bulls is an event-driven system which waits for certain actions to occur and responds the them. The users profiles will be updated on the dashboard for all leagues everytime the dashboard is loaded. Upon receiving this request the system contacts the StockInfoProvider and updates the users portfolio accordingly. To view league standings, our system needs to update every league member's portfolio so the user can view up-to-date league standings. To view information on a stock, the StockInfoProvider is contacted to provide up-to-date information concerning said stock. The order of execution for order tickets uses a linked-list sorted by the time an order is received. Executed orders are removed from the list as they are executed.

9.7 Hardware Requirements

Bears & Bulls is optimized for use with a color display with a minimum resolution of 760xn pixels because the maximum pixel width allowed by iframe is 760 pixels. The user doesnt require any hard drive space for this application as all the required data is stored on Bears & Bulls' servers. A network connection is required to access Facebook. If Facebook authentication is accessible, then Bears & Bulls is accessible. Users devices need an Internet connection to connect to the system. This is the main hardware requirement our system needs.

10 Algorithms and Data Structures

10.1 Algorithms

Most of the functions of Bears & Bulls take user inputs and return outputs with minimal data manipulation other than page rendering. As such, there are really no noteworthy algorithms to discuss apart from the model used to simulate price slippage for block trades. ADD NEW MODEL A relational database will be used for persistent data storage. Most of the data in the system will be entered into the database and so algorithms for manipulating the data, such as sorting and searching, are handled by the database. Thus search and sorting algorithms are not in the scope of the system and will not be discussed.

10.2 Data Structures

The main structure of concern that is used in Bears & Bulls is a linked-lists. Linked-lists are used to hold the pending order tickets that the system has stored within it.

The linked-list structure was chosen because it supports easy insertion and deletion from the list. The linked list was chosen over the queue because the queue does not support deletion from any point within the queue. This is necessary because the orders are not necesarrily executed in the order they are received. The system iterates over the list of tickets and skips orders whose order condititions have not been met. An order submitted later may be executed first if its order condition is fulfilled first and that ticket should be removed immediately after execution, regardless of the location.

An array was not chosen because the data structure must be able to support an arbitrary number of tickets in at any given time. Thus a fixed-size array would not be appropriate. The overhead of resizing a fixed size array to handle insertions and deletions makes an array-based list implementation a poor candidate for the order list. Both arrays and linked-lists are O(1) in terms of insertion, and for our purposes they are both O(n) for retrieval because each retrieval requires a traversal. However, the list has a O(1) deletion while the array will have O(n) for deletion due to shifting.

Most of the data that is needed by Bears & Bulls is stored in a relational database and so container classes such as investor lists and league membership lists are not in the scope of the system.

11 User Interface Design and Implementation

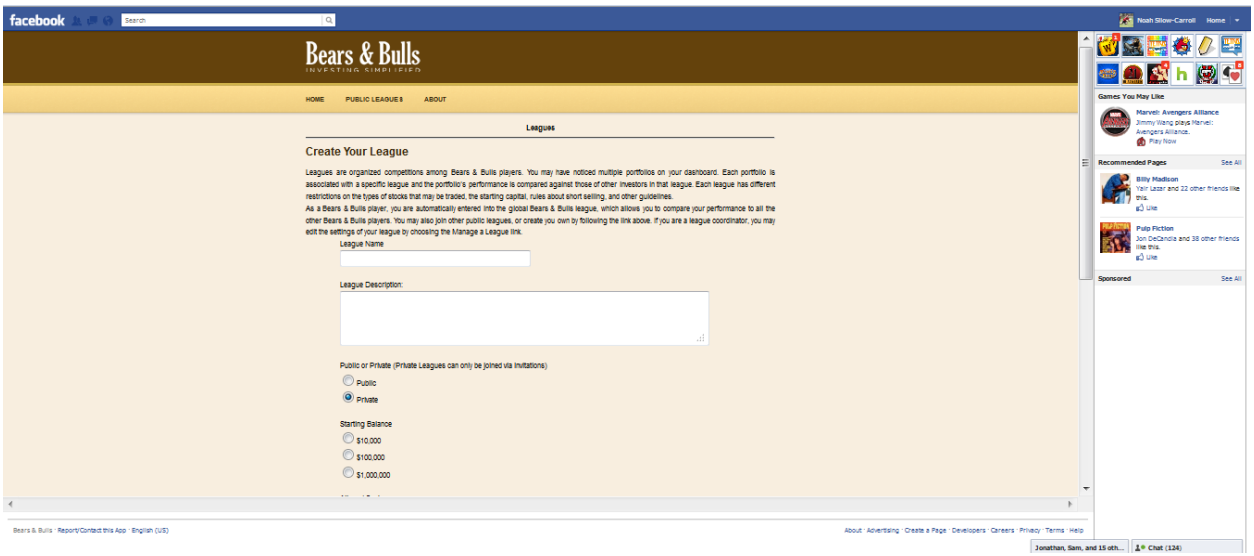
Bears & Bulls benefits greatly from being an app on Facebook since this eliminates the tedious sign up process. Buying your first stock is as easy as just installing the app. The entire app workflow was designed with simplicity in mind, and so it was stressed that a user should be able to get from any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

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 and funds, snapshots of the three major markets for the day and links to create and manage leagues and funds.

Create League

Clicking on Create under Leagues on the dashboard will allow the user to create a league and specify the league name, description, public/private model, starting balance, allowed sectors and whether investing in funds is allowed. Creating funds is very similar to creating leagues.

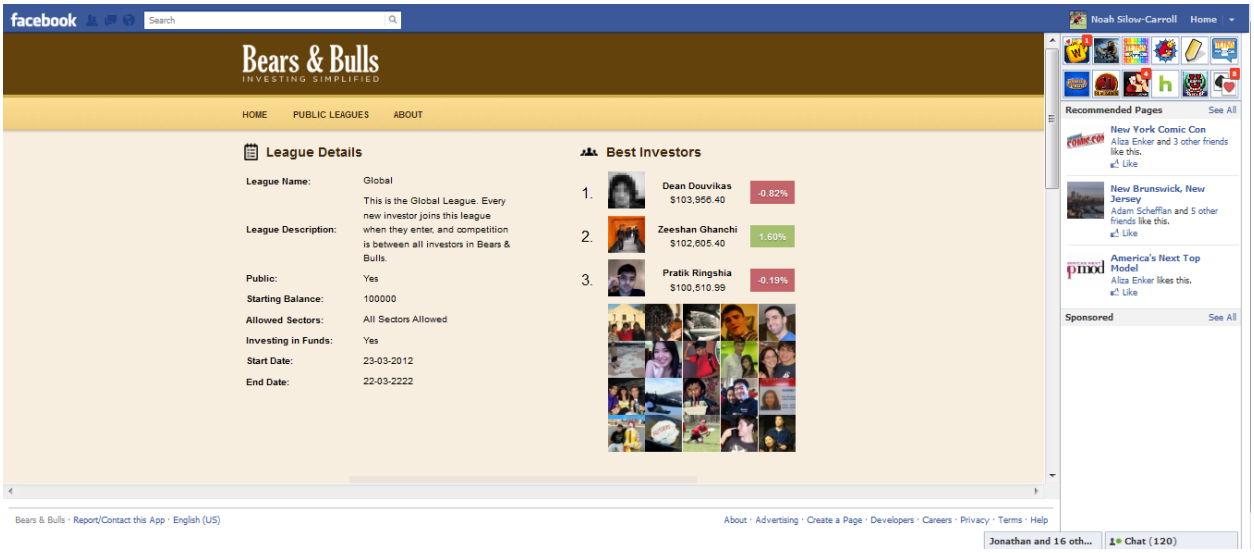


League Invitation

When a league invitation is received, the View Invites link under Leagues on the dashboard will be highlighted. Clicking on this brings the user to the invite page where they can view league details and either accept or decline the invitation.

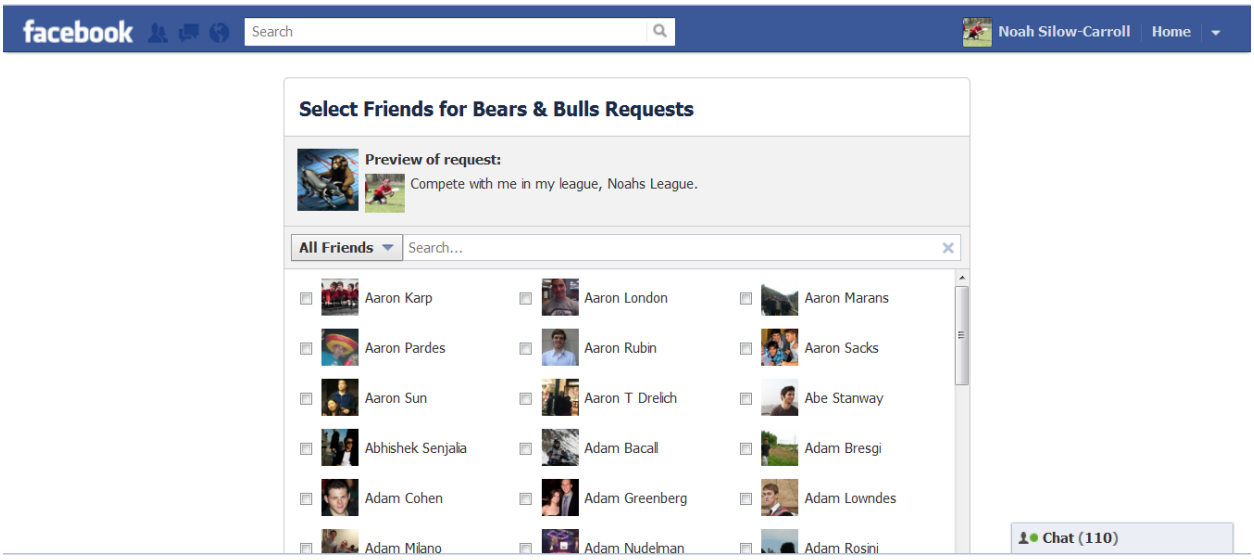
League Details

Clicking on League Details for any league on the dashboard presents the user with rules/settings for the league, league rankings and a comment section where members of the league can discuss stocks and brag about their progress. Changing the settings of a fund will bring the user to a similar page for the fund.



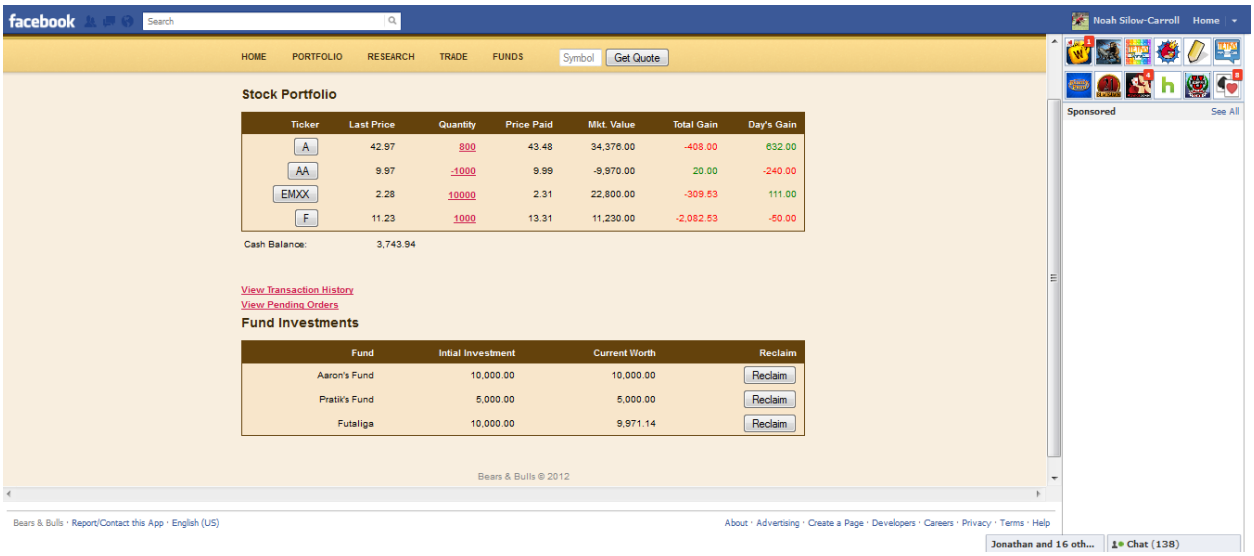
Invite Friends

Clicking Invite Friends for any league run by the user allows the user to send Facebook invitations to any of their friends on facebook.



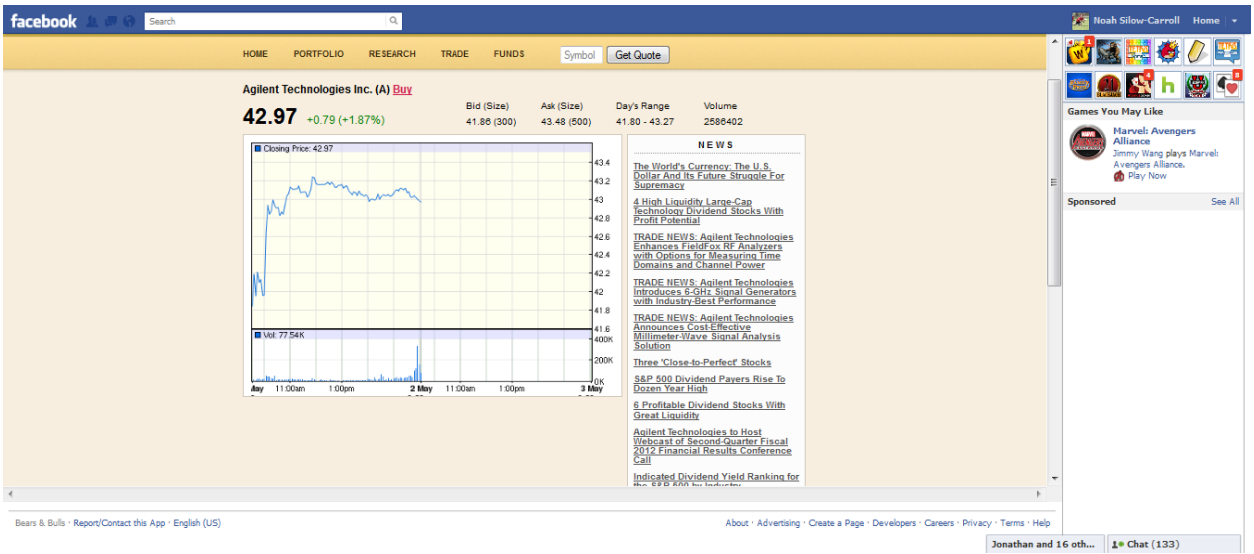
Portfolio

Clicking on any of the leagues or funds on the dashboard will bring the user to their portfolio for that league or fund. Here they can view their stocks, transaction history, pending orders and fund investments.



Research Stock

After finding a specific stock and clicking Get Quote, the user is presented with stock data, graphs and links to recent articles on the stock.



12 Design of Tests

NOTE: There was some confusion with the tests for Report 2. The tests were correct according to Professor Marsic and hence the tests remain unchanged except for those tests that no longer apply.

12.1 State Diagrams

Note on State Diagrams

Depicted below are the state diagrams for leagues and order tickets. Although there are many more classes within the system, these two are the only classes that contain non-trivial states (idle and active). Therefore, only these two will be depicted below. Test cases have been developed for all classes though.

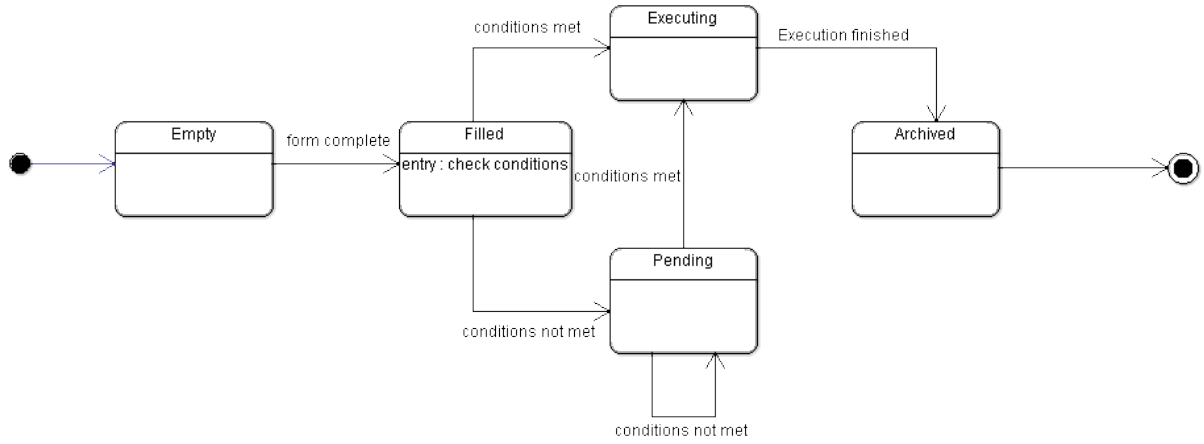


Figure 28: State Diagram of Order Ticket

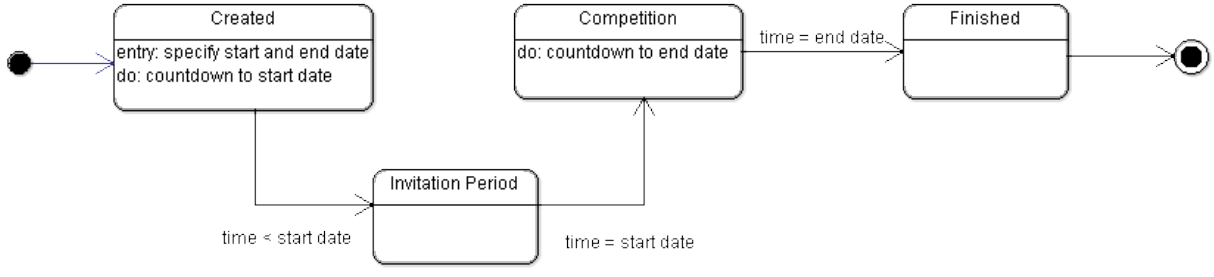


Figure 29: State Diagram of League

12.2 Unit Tests

12.2.1 Controller

Test-case Identifier: TC-1 **Function Tested:** Controller::Render(Integer : type,void* : data) : Boolean **Pass/Fail Criteria:** The test passes if the correct data to be rendered is passed to PageRenderer. The test fails if this data is incorrect or incomplete.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data to be rendered is passed, pageType of PageRenderer gets called
–Call Function (Fail)	–Data to be rendered is incomplete or incorrect, function returns zero

Test-case Identifier: TC-2 **Function Tested:** Controller::RenderError(Integer : type,void* : data) : Boolean **Pass/Fail Criteria:** The test passes if the correct data to be rendered is passed to PageRenderer. The test fails if this data is incorrect or incomplete.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data to be rendered is passed, pageType of PageRenderer gets called
–Call Function (Fail)	–Data to be rendered is incomplete or incorrect, function returns zero

Test-case Identifier: TC-3 **Function Tested:** Controller::RequestPortfolio(String : Investor) : Void **Pass/Fail Criteria:** The test passes if the correctly matching portfolio is signaled to be retrieved. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, RequestPortfolio of DataHandler is called
–Call Function (Fail)	–Data does not get sent, RequestPortfolio of DataHandler is not called

Test-case Identifier: TC-4 **Function Tested:** Controller::RequestCreateL(Field : fields) : Void **Pass/Fail Criteria:** The test passes if the request complying with the correct User's league settings is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, CreateLeague of DataHandler is called
–Call Function (Fail)	–Data does not get sent, CreateLeague of DataHandler is not called

Test-case Identifier: TC-5 **Function Tested:** Controller::RequestCreateF(Field : fields) : Void **Pass/Fail Criteria:** The test passes if the request complying with the correct User's fund settings is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, CreateFund of DataHandler is called
–Call Function (Fail)	–Data does not get sent, CreateFund of DataHandler is not called

Test-case Identifier: TC-6 **Function Tested:** Controller::RequestEditL(Field : fields) : Void **Pass/Fail Criteria:** The test passes if the request complying with the correct User's edits is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, EditLeague of DataHandler is called
–Call Function (Fail)	–Data does not get sent, EditLeague of DataHandler is not called

Test-case Identifier: TC-7 **Function Tested:** Controller::RequestEditF(Field : fields) : Void **Pass/Fail Criteria:** The test passes if the request complying with the correct User's edits is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, EditFund of DataHandler is called
–Call Function (Fail)	–Data does not get sent, EditFund of DataHandler is not called

Test-case Identifier: TC-8 **Function Tested:** Controller::RequestHistory(String : investor) : Void **Pass/Fail Criteria:** The test passes if the request for the correct investor history is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, RequestHistory of DataHandler is called
–Call Function (Fail)	–Data does not get sent, RequestHistory of DataHandler is not called

Test-case Identifier: TC-9 **Function Tested:** Controller::RequestInvite(String : investor, String : League) : Void **Pass/Fail Criteria:** The test passes if the request to invite the correct investor to join the correct league is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, Invite of DataHandler is called
–Call Function (Fail)	–Data does not get sent, Invite of DataHandler is not called

Test-case Identifier: TC-10 **Function Tested:** Controller::RequestBuy(Ticket : ticket) : Void **Pass/Fail Criteria:** The test passes if the request to buy stock is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, ExecuteOrder of DataHandler is called
–Call Function (Fail)	–Data does not get sent, ExecuteOrder of DataHandler is not called

Test-case Identifier: TC-11 **Function Tested:** Controller::RequestSell(Ticket : ticket) : Void **Pass/Fail Criteria:** The test passes if the request to sell stock is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
–Call Function (Pass)	–Correct data gets sent, ExecuteOrder of DataHandler is called
–Call Function (Fail)	–Data does not get sent, ExecuteOrder of DataHandler is not called

12.2.2 PageRenderer

Test-case Identifier: TC-12 **Function Tested:** PageRenderer::generatePageOrder(TicketLticket, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the test stub enters a request for an order and an order page is generated. Unsuccessful if page is not generated.	
Test Procedure	**Expected Results**
–Submit an order ticket (Pass)	–Page Renderer returns true and order result page is successfully created
–Request an order page (Fail)	–Page Renderer returns false if the page could not be generated

Test-case Identifier: TC-13 **Function Tested:** PageRenderer::generatePageStock(StockData:data, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the test stub enters a request for a stock and a stock page is generated. Unsuccessful if page cannot be generated.	
Test Procedure –Request a stock page (Pass)	**Expected Results** –Page Renderer returns true and market data page is successfully created
–Request a stock page (Fail)	–Page Renderer returns false if the page could not be generated

Test-case Identifier: TC-14 **Function Tested:** PageRenderer::generatePagePortfolio(Portfolio:portfolio, StockData* data, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for a portfolio and a portfolio page is generated. Unsuccessful if page cannot be generated.	
Test Procedure –Request a portfolio page (Pass)	**Expected Results** –PageRenderer returns true and user portfolio page is created
–Request a portfolio page (Fail)	–PageRenderer returns false if portfolio page could not be generated

Test-case Identifier: TC-15 **Function Tested:** PageRenderer::generatePageFront(UserInfo:userinfo, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for home and a home page is generated. Unsuccessful if page cannot be generated.	
Test Procedure –Request a front page (Pass)	**Expected Results** –PageRenderer returns true and home page is generated
–Request a front page (Fail)	–PageRenderer returns false if front page could not be generated

Test-case Identifier: TC-16 **Function Tested:** PageRenderer::generatePageLorF(Fields: fields, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for a league or fund and a correct league or fund page is generated. Unsuccessful if page cannot be generated.	
Test Procedure	**Expected Results**
–Request a league page (Pass)	–Page Renderer returns true and league page is created
–Request a league page (Fail)	–Page Renderer returns an false if page could not be created
–Request a fund page (Pass)	–Page Renderer returns true and fund page is created
–Request a fund page (Fail)	–Page Renderer returns an false if page could not be created

Test-case Identifier: TC-17 **Function Tested:** PageRenderer::generatePageJoin(String: league, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for joining a league and a join page is generated. Unsuccessful if page cannot be generated.	
Test Procedure	**Expected Results**
Request a join page (Pass)	Page Renderer returns true and league join page is created
Request a join page (Fail)	Page Renderer returns false and page is not generated

Test-case Identifier: TC-18 **Function Tested:** PageRenderer::generatePageInvite(Investor: investor, String: league Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for a league invite and a league invite page is generated. Unsuccessful if page cannot be generated.	
Test Procedure	**Expected Results**
–Request a league invite page (Pass)	–Page Renderer returns true and request invitation page is created
–Request a league invite page (Fail)	–Page Renderer returns false and page is not generated

Test-case Identifier: TC-19 **Function Tested:** PageRenderer::pageType(Integer : Type, void* : data, Integer: valid):Boolean **Pass/fail Criteria:** The test passes if the page renderer calls the correct	
Test Procedure	**Expected Results**
–Request page type (Pass)	–Page Renderer returns true if a corresponding generate page function is called
–Request page type (Fail)	–Page Renderer returns false if a corresponding generate page function is not called. For example if the integer Type is out of range.

Test-case Identifier: TC-20 **Function Tested:** PageRenderer::getPage():Page **Pass/fail Criteria:** The test passes if the system returns a page and unsuccessful if no page is returned.	
Test Procedure	**Expected Results**
–Request a page (Pass)	–System displays Returns true if a page is loaded and returned
–Request a page (Fail)	–Incorrect if a page isnt loaded, record that no page loaded

12.2.3 DataHandler

Test-case Identifier: TC-21 **Function Tested:** DataHandler::executeOrder(Ticket: ticket): Boolean **Pass/Fail Criteria:** The test passes if the test stub executes the ticket by updating the investors portfolio accordingly	
Test Procedure	**Expected Results**
–Execute order (Pass)	–DataHandler executes order and updates investors portfolio and returns true.
–Execute order (Fail)	–If unable to execute order, return false.

Test-case Identifier: TC-22 **Function Tested:** DataHandler::RequestPortfolio(String: Investor): Portfolio **Pass/Fail Criteria:** The test passes if the test stub requests for portfolio data and it is retrieved from the database	
Test Procedure	**Expected Results**
–Request portfolio data (Pass)	–DataHandler requests portfolio data and returns it from the database.
–Request portfolio updaten (Fail)	–If there is an error retrieving the data from the database, it should display an error that no pertinent data was returned.

Test-case Identifier: TC-23 **Function Tested:** DataHandler::CreateAccount(UserInfo: userinfo): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests an account creation and the request is granted.	
Test Procedure	**Expected Results**
–Request to create an account (Pass)	–DataHandler requests account creation and returns true if account creation successful
—Call Function (Fail)	–If the request for account creation is unsuccessful, return false

Test-case Identifier: TC-24 **Function Tested:** DataHandler::CreateLeague(Fields: fields): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests a league creation and it is created. Unsuccessful if league isnt created.	
Test Procedure	**Expected Results**
–Request to create a league (Pass)	–DataHandler requests league creation and returns true if league creation successful
–Request to create a league (Fail)	–If the request for league creation is unsuccessful, return false

Test-case Identifier: TC-25 **Function Tested:** DataHandler::EditLeague(Fields: fields): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests to modify league settings in database and is successful. Unsuccessful if settings not changed.	
Test Procedure	**Expected Results**
–Request to edit league settings (Pass)	–DataHandler modifies league settings and returns true.
–Request to edit league settings (Fail)	–DataHandler unable to modify league settings, returns false.

Test-case Identifier: TC-26 **Function Tested:** DataHandler::CreateFund(Fields: fields): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests a fund creation and it is created. Unsuccessful if fund isnt created.	
Test Procedure	**Expected Results**
–Request to create a fund (Pass)	–Fund Handler requests fund creation and returns true if fund creation successful
–Request to create a fund (Fail)	–If the request for fund creation is unsuccessful, return false.

Test-case Identifier: TC-27 **Function Tested:** DataHandler::EditFund(Fields: fields): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests to modify fund settings in database and is successful. Unsuccessful if settings not changed.	
Test Procedure	**Expected Results**
–Request to edit fund settings (Pass)	–DataHandler modifies fund settings and returns true.
–Request to edit fund settings (Fail)	–DataHandler unable to modify fund settings, returns false.

Test-case Identifier: TC-28 **Function Tested:** DataHandler::RequestHistory(String: Investor): History **Pass/Fail Criteria:** The test passes if the test stub requests to view transaction history from the database and it is successful.	
Test Procedure	**Expected Results**
–Request to view transaction history –Request to view transaction history (Fail)	–DataHandler returns the transaction history. –DataHandler is unable to return transaction history, display error saying unable to retrieve transaction history.

Test-case Identifier: TC-29 **Function Tested:** DataHandler:: JoinLeague(String: Investor, String: League): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests investor to be added to a league in database and is successful. Unsuccessful if it doesnt occur.	
Test Procedure	**Expected Results**
–Request to join league (Pass) –Request to join league (Fail)	–DataHandler updates information in database about the league and returns true. –If joining the league encounters a problem, return false.

Test-case Identifier: TC-30 **Function Tested:** DataHandler::Invite(String: Investor, String: League): Void **Pass/Fail Criteria:** The test passes if the test stub requests that an invite be added to the investors account.	
Test Procedure	**Expected Results**
–Add invite to investors account (Pass) –Add invite to investors account (Fail)	–DataHandler adds the invite to investors account in database. –If unable to add invite in database, display error saying that invite wasnt added to investors account.

Test-case Identifier: TC-31 **Function Tested:** DataHandler::ExecuteOrder(Ticket: ticket): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests that the database allocate money to the specified investor and is able to.	
Test Procedure	**Expected Results**
–Allocate money to investor (Pass)	–DataHandler allocates money to the specified investor and returns true.
–Allocate money to investor (Fail)	–If unable to allocate money to investor, return false.

12.2.4 ValidityChecker

Test-case Identifier: TC-32 **Function Tested:** ValidityChecker::ValidateBuy(Ticket: ticket): void **Pass/Fail Criteria:** The test passes if the test stub determines that a buy is valid. Unsuccessful if buy is not valid.	
Test Procedure	**Expected Results**
–Submit buy request (Pass)	–Validity Checker verifies that buy is valid.
–Submit buy request(Fail)	–If attempted buy is invalid, Validity Checker should display an error code that buy is invalid.

Test-case Identifier: TC-33 **Function Tested:** ValidityChecker::VerifyFunds(): void **Pass/Fail Criteria:** The test passes if the test stub determines that the investor has sufficient funds for the transaction. Unsuccessful if insufficient funds for the transaction	
Test Procedure	**Expected Results**
–Submit buy order ticket (Pass)	–Validity Checker verifies that cash balances are sufficient for order.
–Submit buy order ticket (Fail)	–If the investor has insufficient funds for the transaction, Validity Checker should display an error that there are insufficient funds for the transaction.

Test-case Identifier: TC-34 **Function Tested:** ValidityChecker::ValidateSell(Ticket: ticket): void **Pass/Fail Criteria:** The test passes if the test stub determines if a sell is valid. Unsuccessful if its invalid.	
Test Procedure	**Expected Results**
–Submit sell request (Pass) –Submit invalid sell request (Fail)	–Validity Checker verifies that sell is valid. –If attempted sell is invalid, Validity Checker should display an error code that sell is invalid.

12.2.5 StockQuery

Test-case Identifier: TC-35 **Function Tested:** StockQuery::Query(String: stock):StockData **Pass/Fail Criteria:** The test passes if the system queries a stock and that stock is returned	
Test Procedure	**Expected Results**
–Request to query a stock (Pass) –Request to query a stock (Fail)	–Stock Query returns the stock data –If the attempted query was for a stock that does not exist, Stock Query should return an error code that the stock does not exist. If stock information was not attainable, it should display an error that no pertinent data was returned.

12.2.6 LiquidityModel

Test-case Identifier: TC-36 **Function Tested:** LiquidityModel::AdjustPrice(StockData: data, Ticket: ticket): Integer **Pass/Fail Criteria:** This test passes if the system requests a price adjustment and the new price is returned	
Test Procedure	**Expected Results**
–Request a price adjustment (Pass) –Request a price adjustment (Fail)	–LiquidityModel returns the adjusted price –If the attempted price adjustment is invalid, LiquidityModel should return an error code that the adjustment was invalid. If price adjustment was not attainable, it should display an error that no pertinent data was returned.

12.2.7 FundHandler

Test-case Identifier: TC-37 **Function Tested:** FundHandler::verifyFields(Fields: fields): Fields **Pass/Fail Criteria:** The test passes if the system verifies that the settings for the fund are all valid and returns the fields	
Test Procedure	**Expected Results**
–Request to verify fields (Pass) –Request to verify fields (Fail)	–FundHandler returns the valid fields. –If incorrect fields are loaded, FundHandler should return an error code that the fields are invalid. If any fields are not filled in, FundHandler should return an error code that there are empty fields.

12.2.8 LeagueHandler

Test-case Identifier: TC-54 **Function Tested:** LeagueHandler::verifyFields(Fields: fields): Fields **Pass/Fail Criteria:** The test passes if the system verifies that the settings for the league are all valid and returns the fields	
Test Procedure	**Expected Results**
–Request to verify fields (Pass) –Request to verify fields (Fail)	–League Handler returns the valid fields. –If incorrect fields are loaded, League Handler should return an error code that the fields are invalid. If any fields are not filled in, League Handler should return an error code that there are empty fields.

12.3 Test Coverage

The test cases are envisioned to cover all states and transitions for every class. This is attained through the testing of every function of every class. Because the transitions and states are all attained in some way or another through a function call, the test coverage is very high and accounts for all of these states and transitions. For example, for the order ticket class, the empty case is the initial default case. The filled state is attained by the investor filling out the form and submitting it. The transition between these two is tested by TC-16 and TC-17, when the web page calls the controller to request a buy and sell. For the transition for pending and execute, TC-46, TC-49, and TC-50 cover the necessary transitions between the states (as well as testing that the states exist). For the archive state, TC-50 by the DataHandler covers the transition as well as the archived state.

In a similar fashion, the states and transitions of the league are also accounted for. Although doing these tests will not ensure that the flow through the entire system is guaranteed, it will make sure that each transition and state is tested.

For the other classes who have trivial states (idle and active), testing the functions will again cover these states because a function call puts the class into an active mode, and exiting the call puts it back in idle state.

12.4 Integration Testing

For integration testing, Bears & Bulls will undergo bottom-up integration testing. Each component in a lower level of the systems hierarchy will be tested individually. After that occurs, the components which rely upon these are tested. Integration testing needs to take place after we conduct the entire unit testing. There is a need to use the higher model to test its interactions with its lower level components. For example, with the PageRenderer, it is necessary to test that the Page Renderer is able to interact with each of its methods correctly. If any problem occurs, testing can pinpoint that the problem is either in the interface between PageRenderer and its method. If a problem is pinpointed, it needs to be reviewed and corrected. By following this strategy, problems can be pinpointed more easily. Drivers need to be implemented to simulate the higher level components. Test stubs will be needed to simulate lower level components. Drivers will need to be implemented for the PageRenderer, ValidityChecker, LiquidityModel, DataHandler, Controller, FundHandler, LeagueHandler, StockQuery, and LeaderBoard. Interactions need to be tested with funds, leagues, investor accounts, portfolios, history, league coordinators, fund managers, order list, shares, tickets, stop orders, limit orders, and market orders. The top level components are the most important, yet they are tested last. This is the last main testing portion where it is determined whether interactions can be made throughout the system without errors. At this point, most of the bugs should be fixed and the system should operate as its operation contracts state. Testing is a major part of software engineering. Due to time constraints, testing may have to be cut short if it consumes too many resources (developers time) and if deadlines are approaching. The more faults found at the beginning of the testing stage increases the probability of finding further faults if testing goes on for an extended period of time.

12.5 Non-functional Requirements Testing

In order to test the system's nonfunctional requirements, a focus group will be surveyed and the results compiled to determine the overall usability and ease of use of the application. Surveyors will be asked such questions as "Did you ever feel lost or overwhelmed at any particular screen?", "Were you able to get where you wanted?", "Did the app produce any unexpected behavior?" (REQ-12, REQ-14). Additionally, the time it takes a user to carry out a predetermined task can be measured to ensure that the app is not needlessly complex, and that it is also loading fast enough (REQ-14, REQ-15). Additionally, the app must be tested on a

variety of browsers (especially the industry leading Chrome, Firefox, and Internet Explorer) on various operating systems to ensure that all users receive a similar experience (REQ-16). Since Bears & Bulls is using Heroku, cases such as system and disk failures will be managed and tested by their team. Heroku provides a system status page on https://status.heroku.com/ which provides the current working conditions of their servers (REQ-17).

13 History of Work, Current Status, Future Work

13.1 History of Work

The planned milestone of completing the First Report within 15 days was successfully accomplished. The planned milestone of completing the Second Report which consisted of Class Diagram and Interface Specification, System Architecture and System Design, Algorithms and Data Structures, User Interface Design and Implementation, Progress Report and Report Editing went faster than we expected in Report 1 by 33% of the time (15 days to 10 days). When creating planned deadlines for the Demos, we assumed team members would spend three to five hours a day working on Bears & Bulls. Fixing errors that occurred and implementing some of the functionalities took a little longer than expected. As the Demo 1 due date approached, the amount of hours put into working on Bears & Bulls increased significantly resulting in some team members spending north of ten hours a day working on Bears & Bulls. The time frame preceding the days of the Demo 1 due date (March 27, 2012) was heavily concentrated. Functionalities from Report 1 which we did not implement were a watch stock list, league entrance fees, adding league coordinators, and Facebook credits from league participants for leagues with entrance fees and paying league winners. We decided to remove these functionalities both because some were determined to be no longer necessary and also time constraints came upon us. We decided as a team that the actual payment would be too complicated to implement given the allotted time frame. The user interface is one of our biggest strengths. Instead of the simply display we had planned as of Report 2, we were able to add more functionalities than previously expected. We were able to have market snapshots on the main screen, giving the user an indication of the markets progress that day. Querying a stock gives not only pertinent stock information from Yahoo Finance, but also news related to the company also retrieved from Yahoo Finance. This was all placed in a user-friendly format which is informative and inviting. The option to buy a stock is not limited to the trade page, but can be purchased straight from querying a stock (simple link to the trade

page).

13.2 Current Status

Currently, Bears & Bulls is a functioning application within Facebook. Users can create multiple portfolios, join leagues and funds, and the user interface has been updated significantly to be more user friendly. Most of the functionalities have been implemented and the current status is just debugging minor bugs and upgrading the efficiency of the application and easy-to-use interface.

13.2.1 Key Accomplishments

The following are the key accomplishments of Bears & Bulls that were implemented.

- Facebook integration
- Posting activity of Bears & Bulls on main Facebook page
- Posting activity of Bears & Bulls within League Details
- Interacting directly with members of a league via Facebook social plugin
- Implementing Leaderboards of each league
- Implementing both Mutual and Hedge Funds
- Advanced research page
- Searching web for news related to stocks queried and displaying them

13.3 Use Cases

The following are the use cases of Bears & Bulls that were implemented.

- Buy Stock
- Sell Stock
- Query Stock
- View History
- View Portfolio
- Register
- Create League
- Submit Comment
- Create Fund
- Join League
- Manage League

- Invite to League
- Update Models
- View Comment
- Manage Fund

13.4 Future Work

Goals we had in mind that werent able to be implemented due to time constraints are several. As indicated in Report 1, a stock watch list within the Research page would be a bonus. It gives investors another option to research stocks before they decide to make or not to make an investment in a given firm. Payment leagues are another key feature that would be beneficial to add. Implementing a league system similar to Yahoo! Fantasy Sports would give the option of testing your skill against serious competition. With options to submit payment prior to entry in a league, financial incentives are now offered to perform well. To ensure that Bears & Bulls remains user-friendly and doesnt become a gambling den, limits would be placed on how large payment to leagues would be to ensure that friendly competition still exists. These payments would be submitted via Facebook credits. Management of this money system would also have to be implemented. Another goal that would be beneficial is to introduce a mobile version of Bears & Bulls. In the modern world where smartphones dominate the market and many people have access to Facebook on the go, enabling access to Bears & Bulls on the go would increase the traffic Bears & Bulls incurs and thus activity would soar. It would also give users multiple interfaces to interact with Bears & Bulls. Enhancing the notification system would be an added bonus. In addition to the existing capabilities, sending notifications when stocks are doing well, stocks are doing bad, movement among the leaderboards, etc. would give the user updates on their activity within Bears & Bulls.

14 Appendix

14.1 Original Domain Model

NOTE: Alternates Omitted.

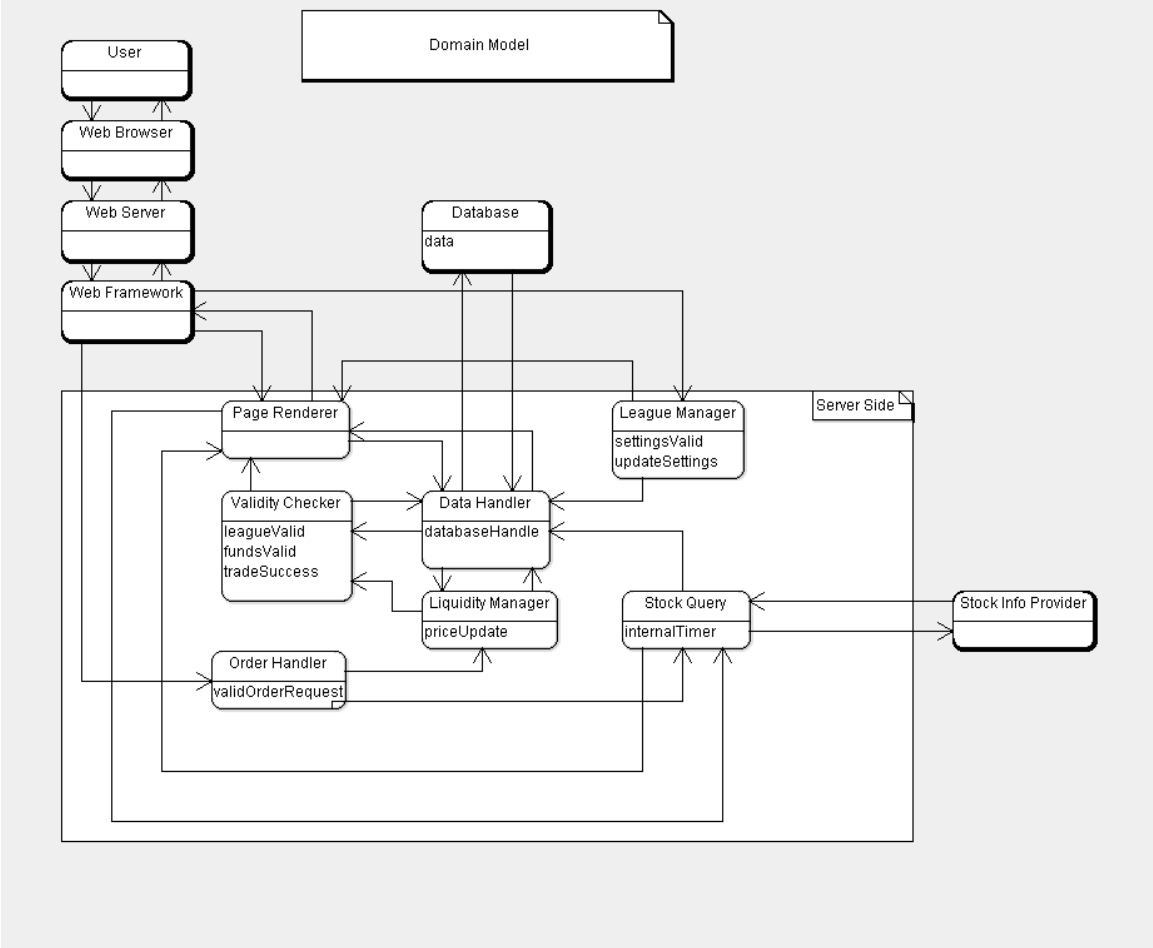


Figure 30: Domain Model

Figure 30 shows Bears & Bulls' general orignal plan for the Domain Model. The subsequent diagrams give insight into how we planned for the concepts to work to satisfy the key use cases of the website. Alternate models that we considered 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. We ended up pursuing the former option.

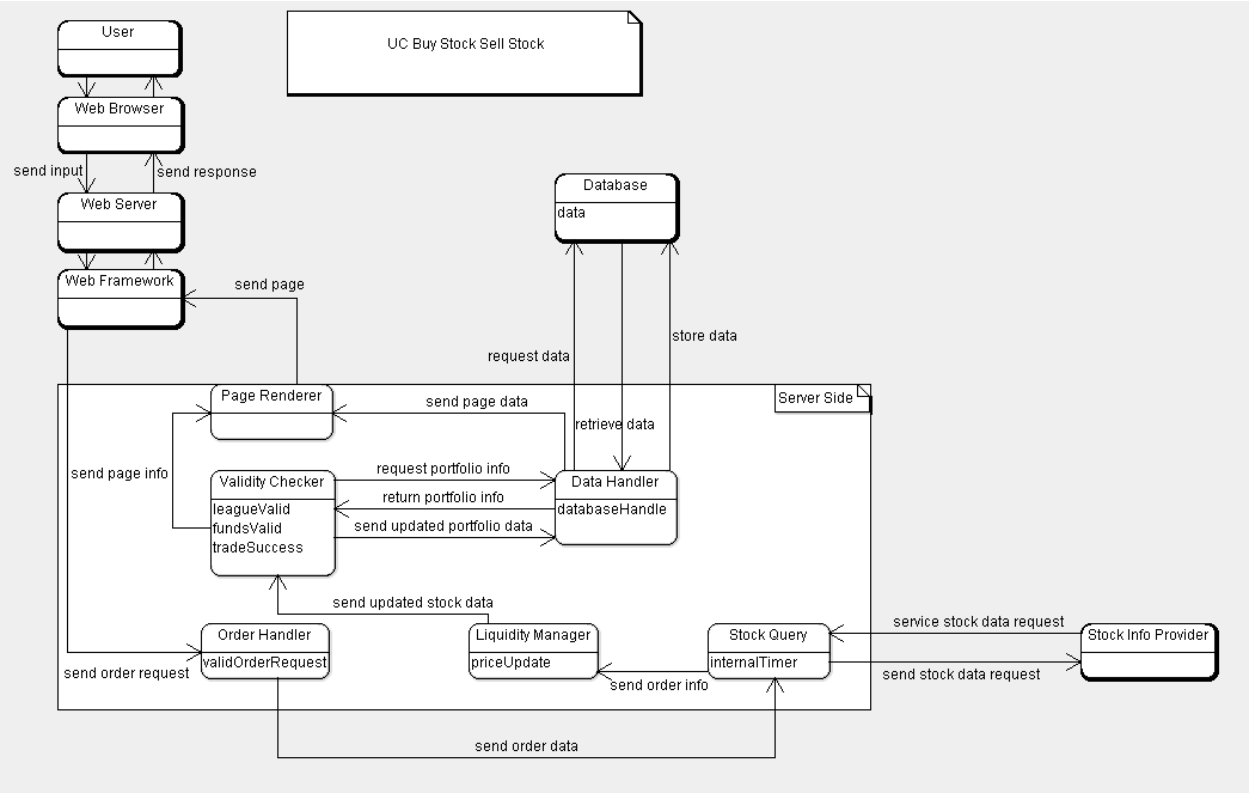


Figure 31: Place Order

Figure 31 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.

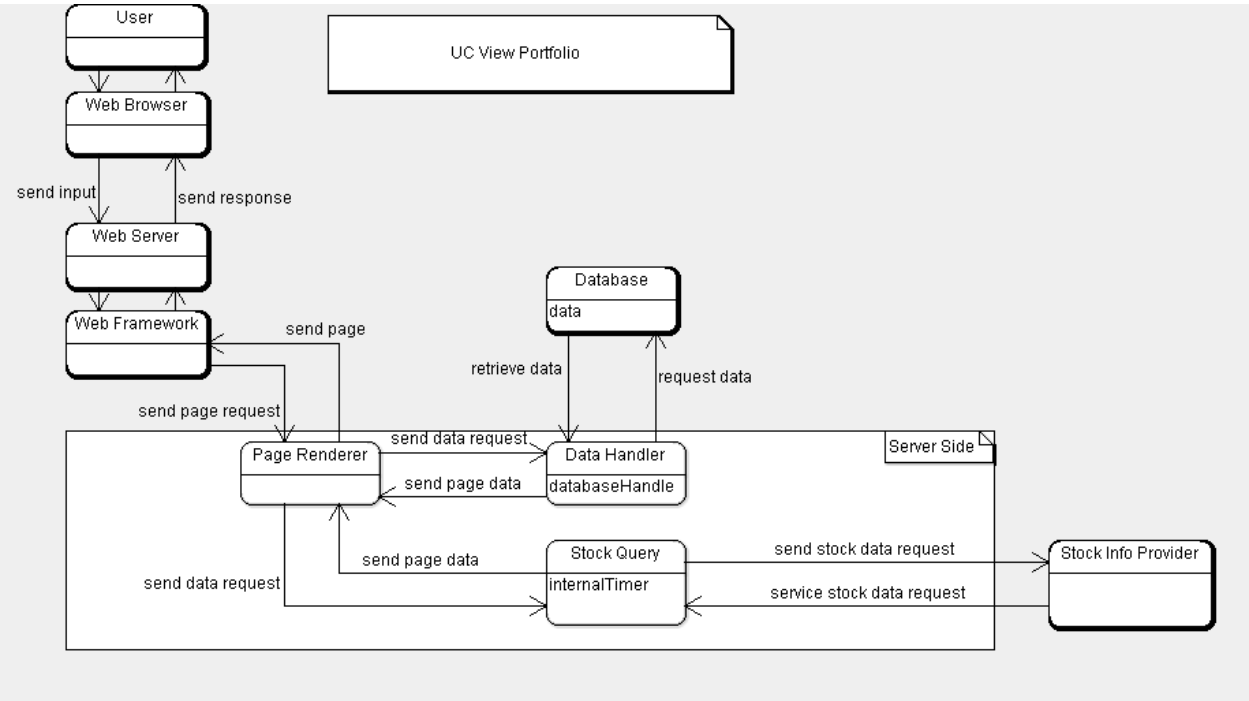


Figure 32: View Portfolio

Figure 32 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.

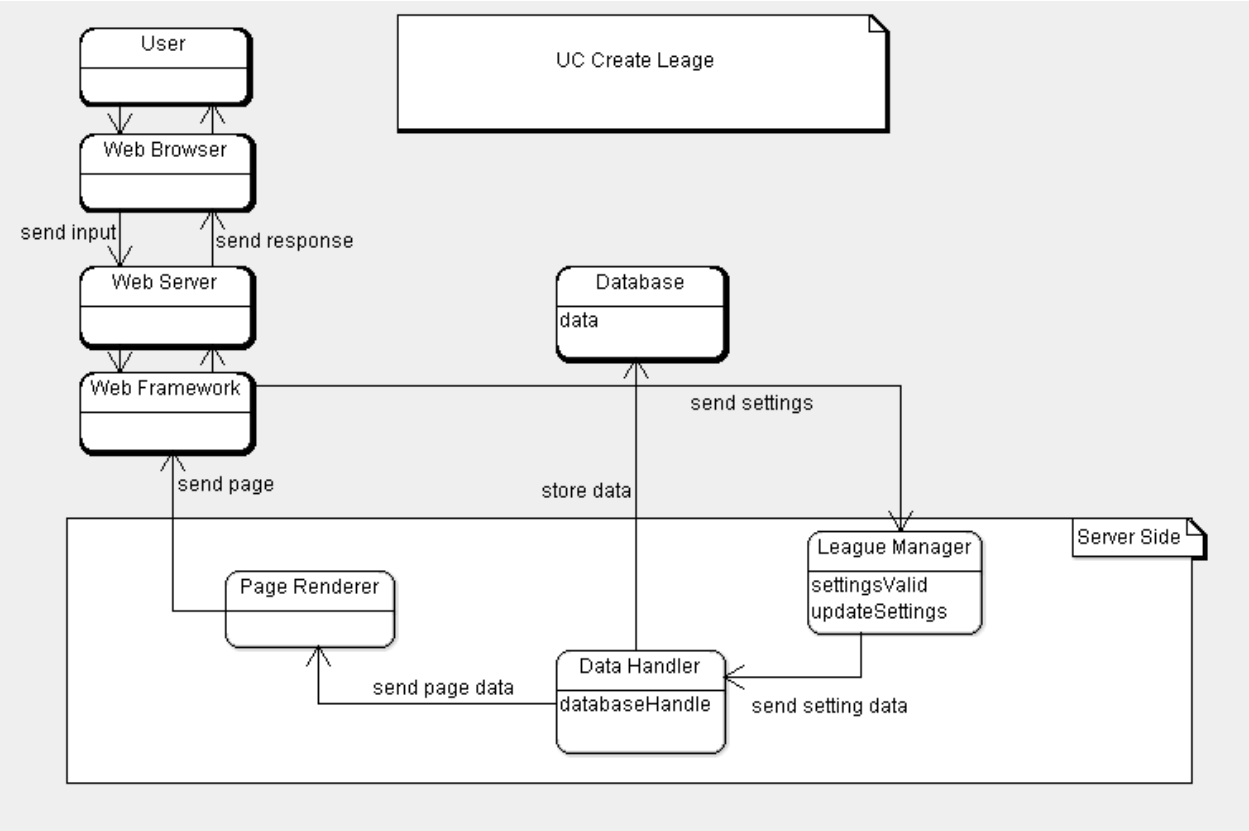


Figure 33: Create League

UC-8 Create League is represented shown in Figure 33. 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.

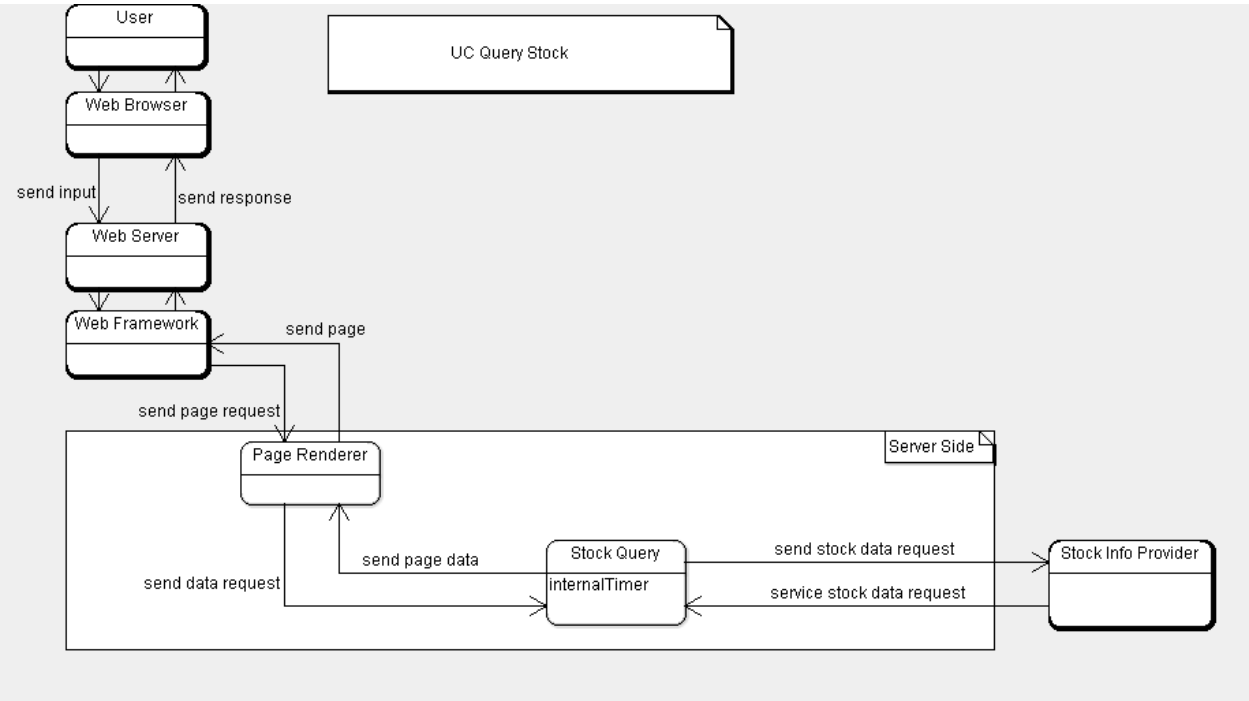


Figure 34: Query Stocks

Figure 34 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.

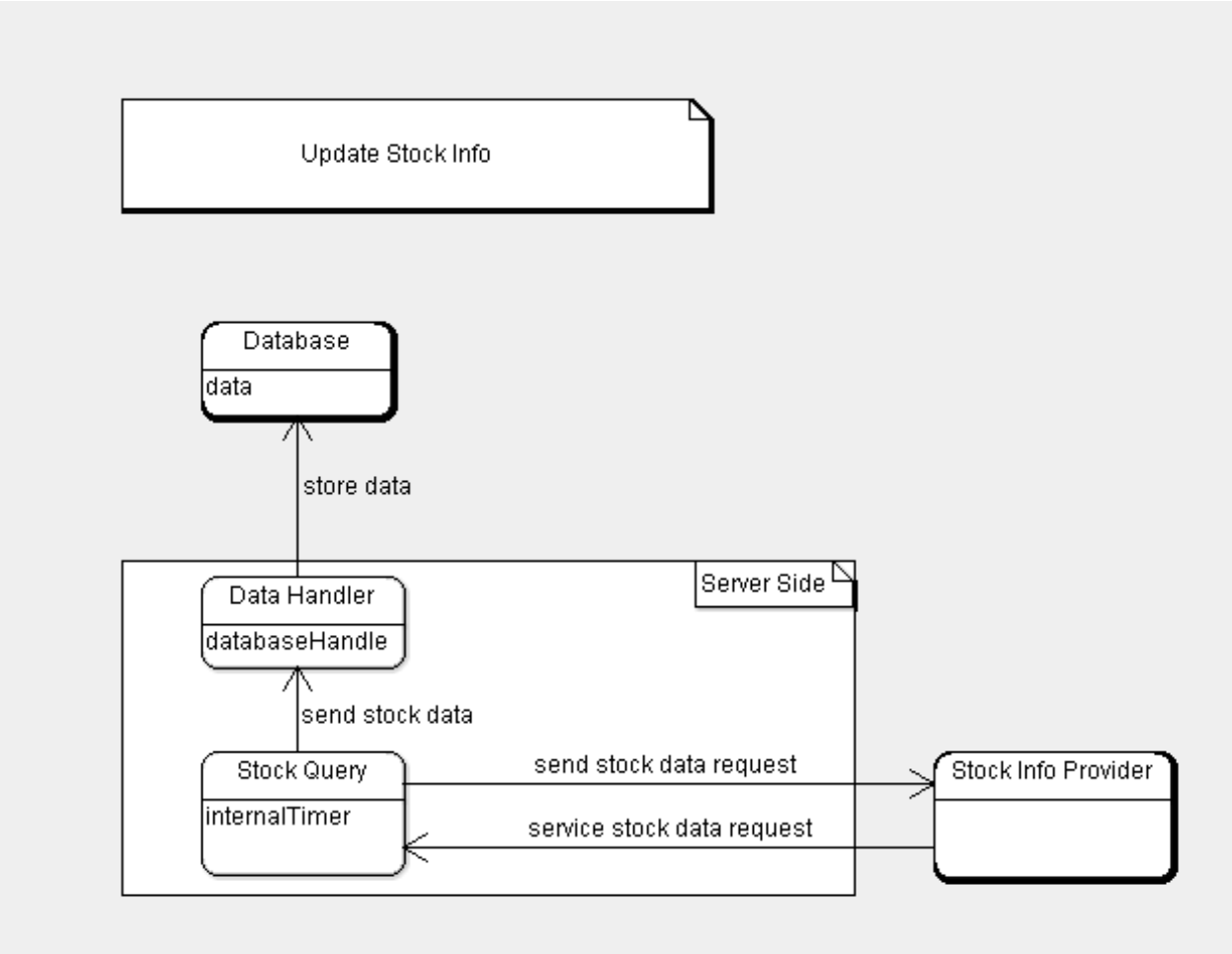


Figure 35: Updating Stock Info

The last diagram (Figure 35) 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.

14.1.1 Original Concept Definitions

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

14.1.2 Original 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

14.1.3 Original 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.

14.1.4 Original Traceability Matrix

Use Case	PW	User	Web Browser	Web Server	Web Framework	Page Renderer	Order Handler	Stock Query	Availability 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

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#axzz1m0ap5Tqp.

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

[4] Facebook Developers. http://www.facebook.com.

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

[6] Technical Indicators and Overlays. http://stockcharts.com/school/doku.phpid=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.