Bears & Bulls

332:452 Software Engineering
Report 2

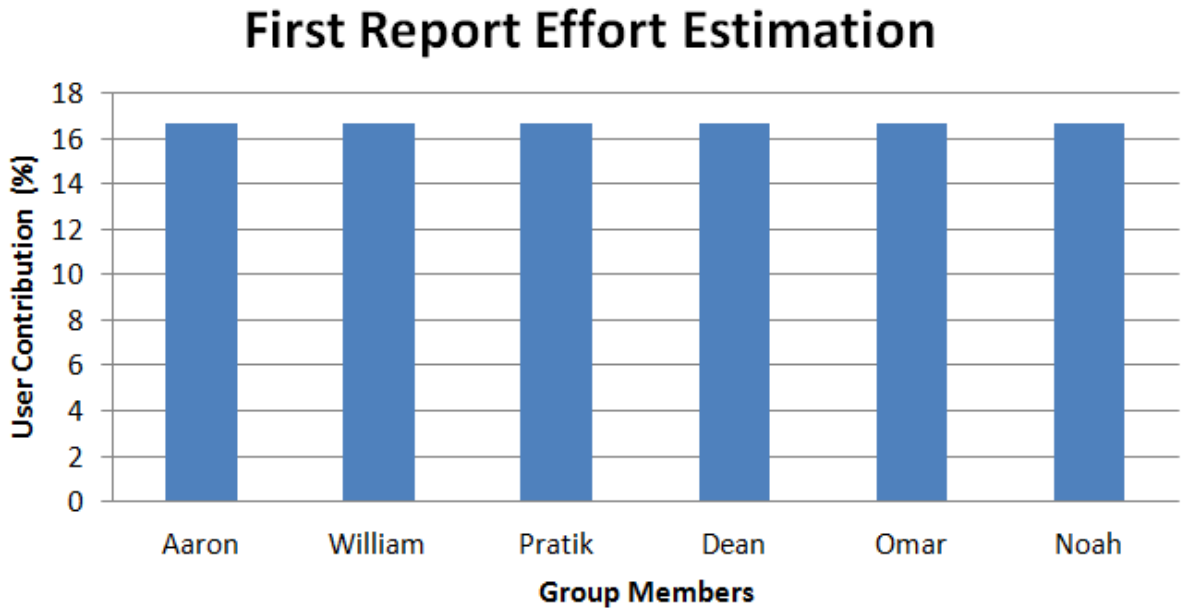
Group 6:
William Pan, Aaron Sun, Pratik Ringshia
Dean Douvikas, Omar Raja, Noah Silow-Carroll

URL: To Be Determined

April 23, 2012

Contributions Breakdown

Task	William	Aaron	Pratik	Dean	Omar	Noah
Cover Page	x					
Table of Contents	x					
Interaction Diagrams		x				
Class Diagram	x					
Data Types and Op. Signatures		x				
Architectural Styles			x	x		
Identifying Subsystems			x	x		
Mapping Subsystems to Hardware					x	x
Persistent Data Storage					x	x
Network Protocol					x	
Global Control Flow					x	x
Hardware Requirements						x
Algorithms	x					
Data Structures		x				
UI Design and Implementation			x			
Design of Tests		x		x	x	
References	x	x	x	x	x	
Progress Report		x				
Plan of Work		x				



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

Thank you.

Contents

NOTE: If viewing as PDF, please zoom in to see images. In addtion, design principles for various objects are stated in the first caption, and not repeated in subsequent captions to avoid repetition.

1 Interaction Diagrams

1.1 Use Case 1/2: Buy/Sell Stocks

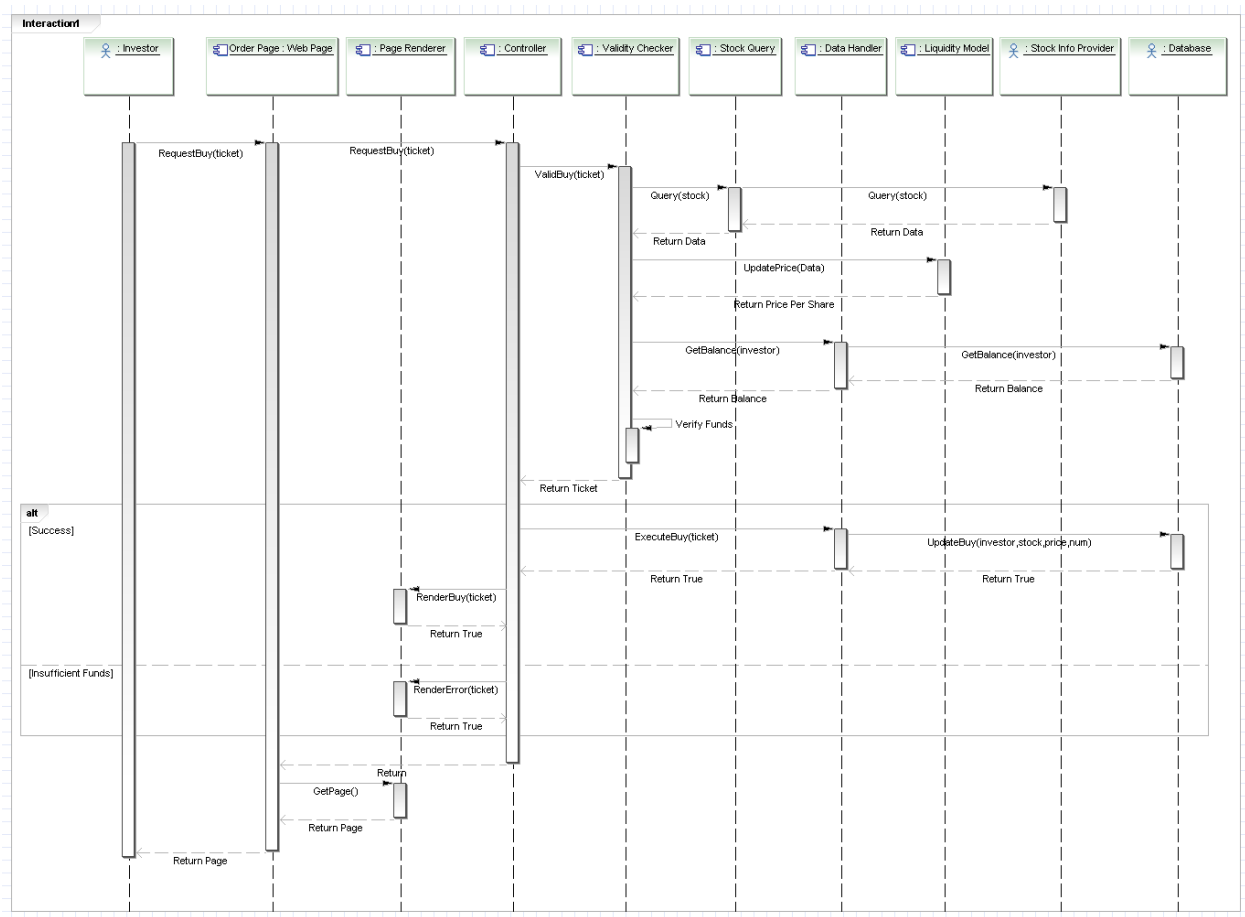


Figure 1: Interaction Diagram for Buy Stocks (From Report 1 Fig. 2)

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

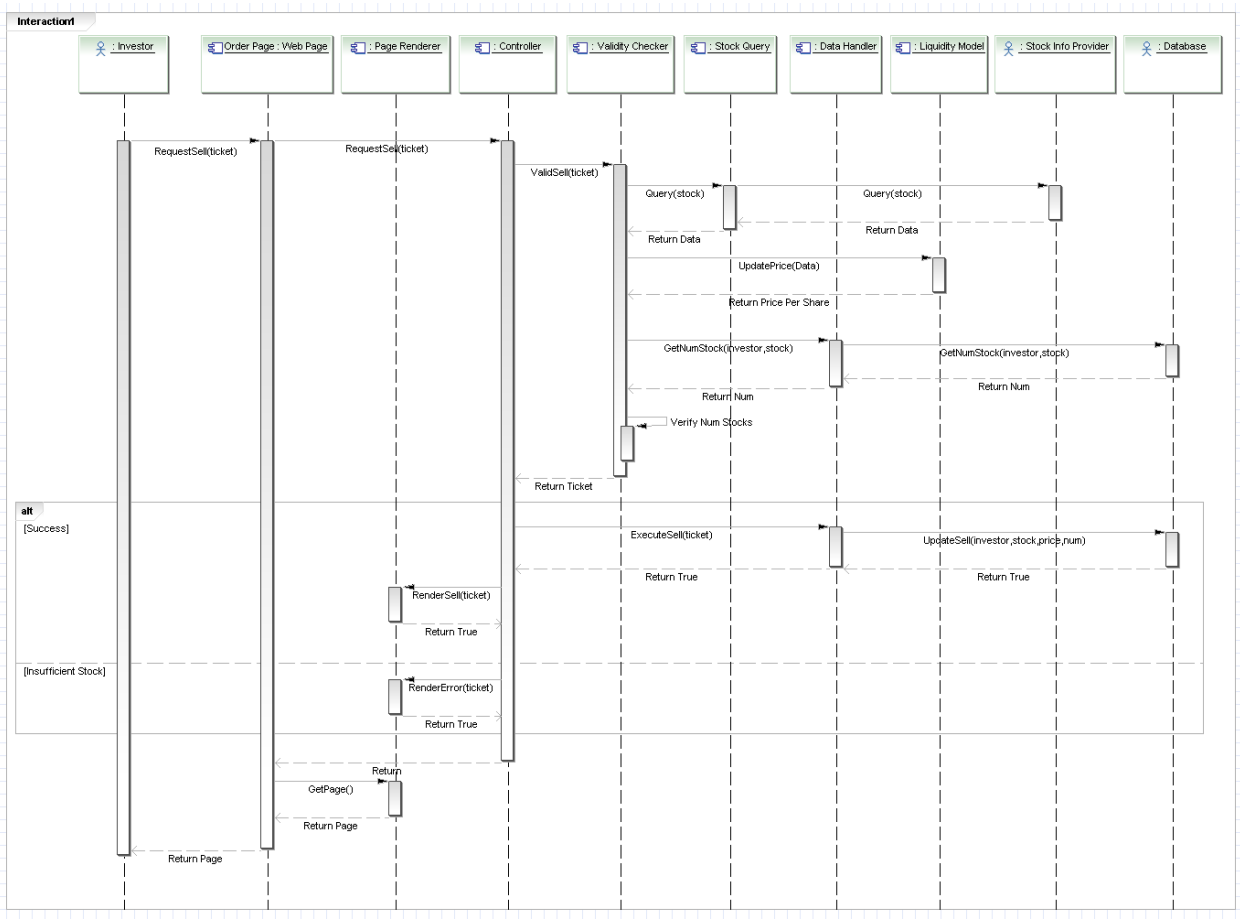


Figure 2: Interaction Diagram for Sell Stocks (From Report 1 Fig. 3)

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 DataHandler to update the Database to reflect the new state after the transaction has been conducted. While creating the DataHandler 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 DataHandler 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 DataHandler 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.

1.2 Use Case 3: Query Stocks

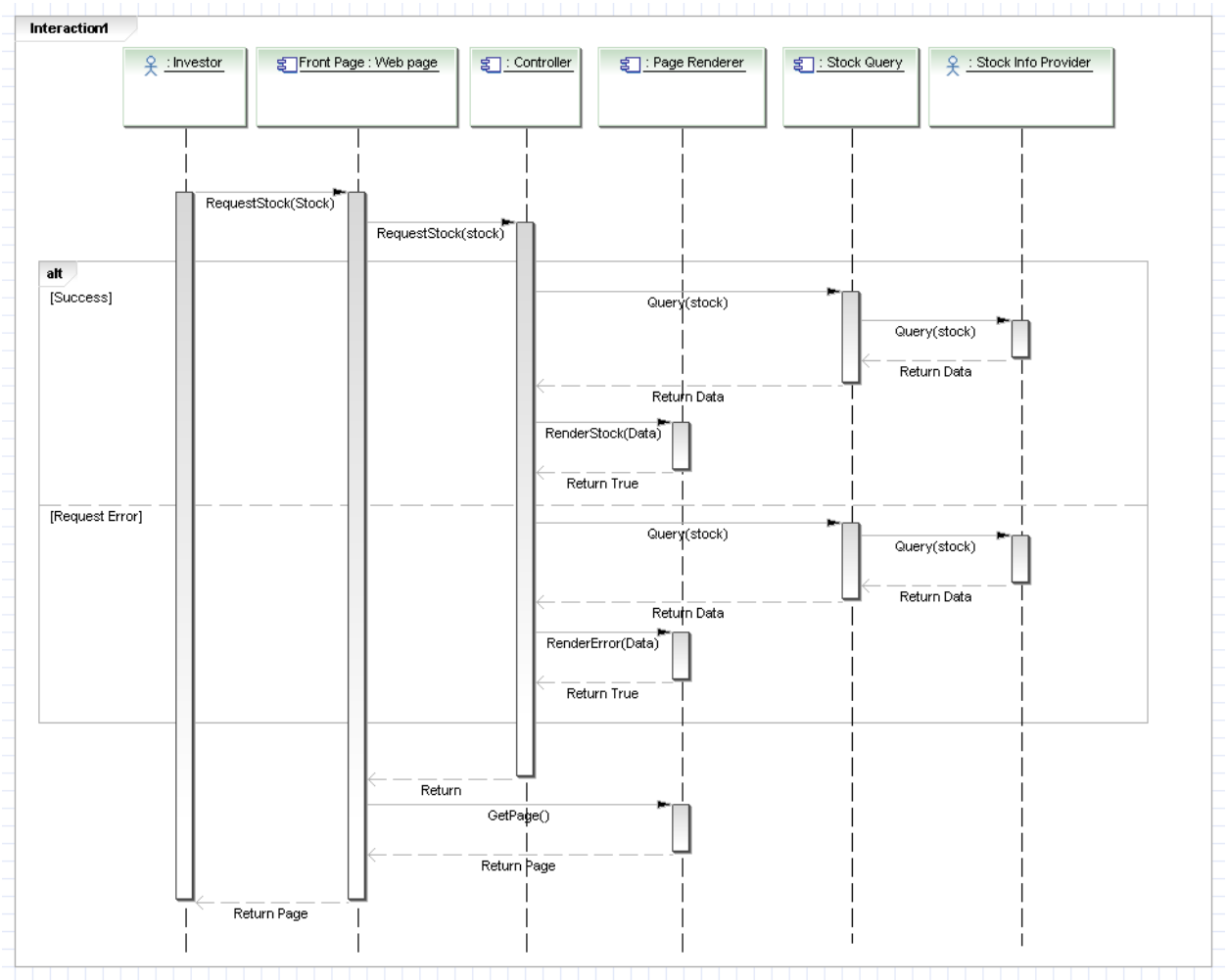


Figure 3: Interaction Diagram for Query Stocks (From Report 1 Fig. 4)

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.

1.3 Use Case 5: View Portfolio

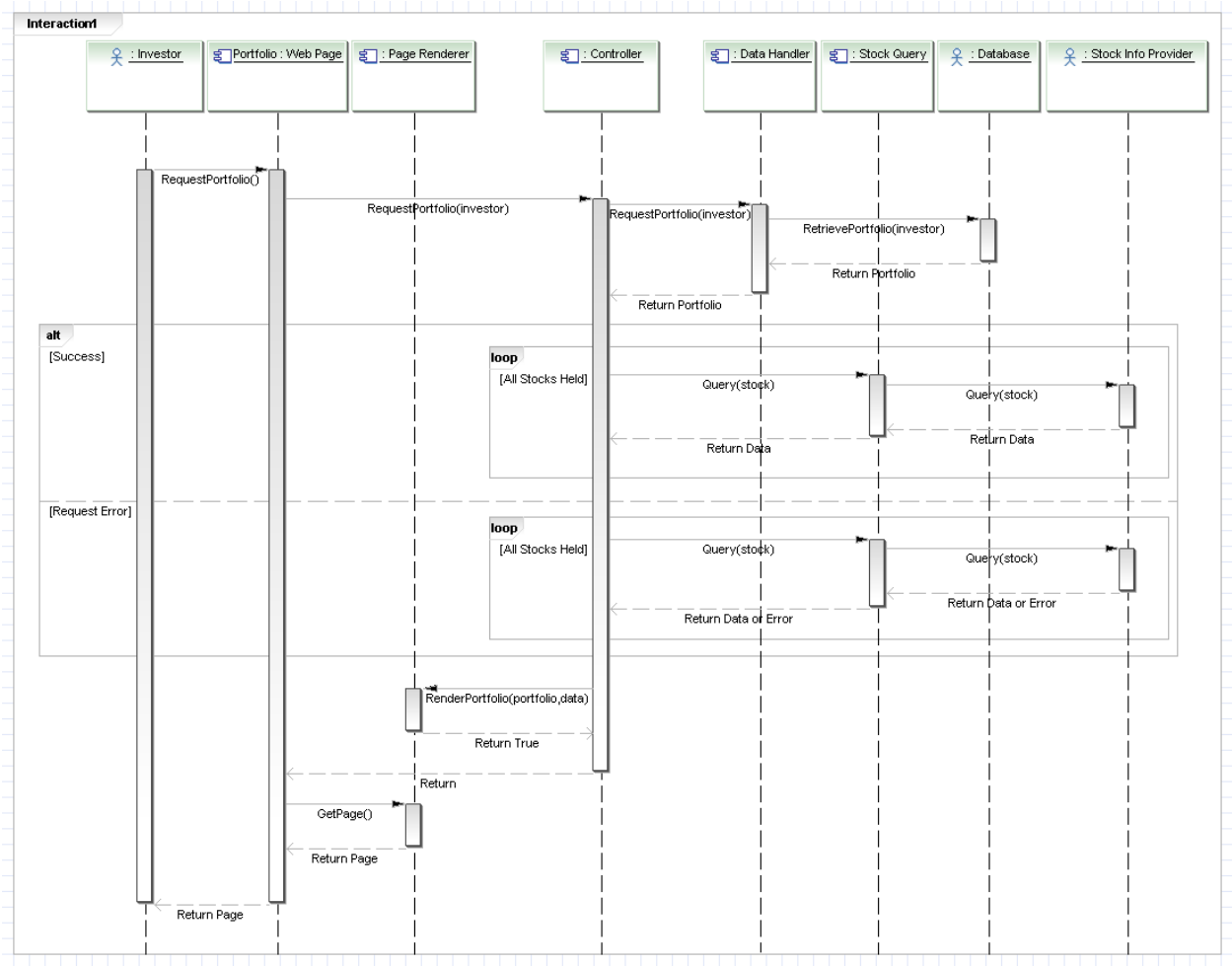


Figure 4: Interaction Diagram for View Portfolio (From Report 1 Fig. 5)

When the Investor wants to view their Portfolio, they notify the Web Page, which communicates with a Controller. The Controller queries the DataHandler 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.

1.4 Use Case 7: Register

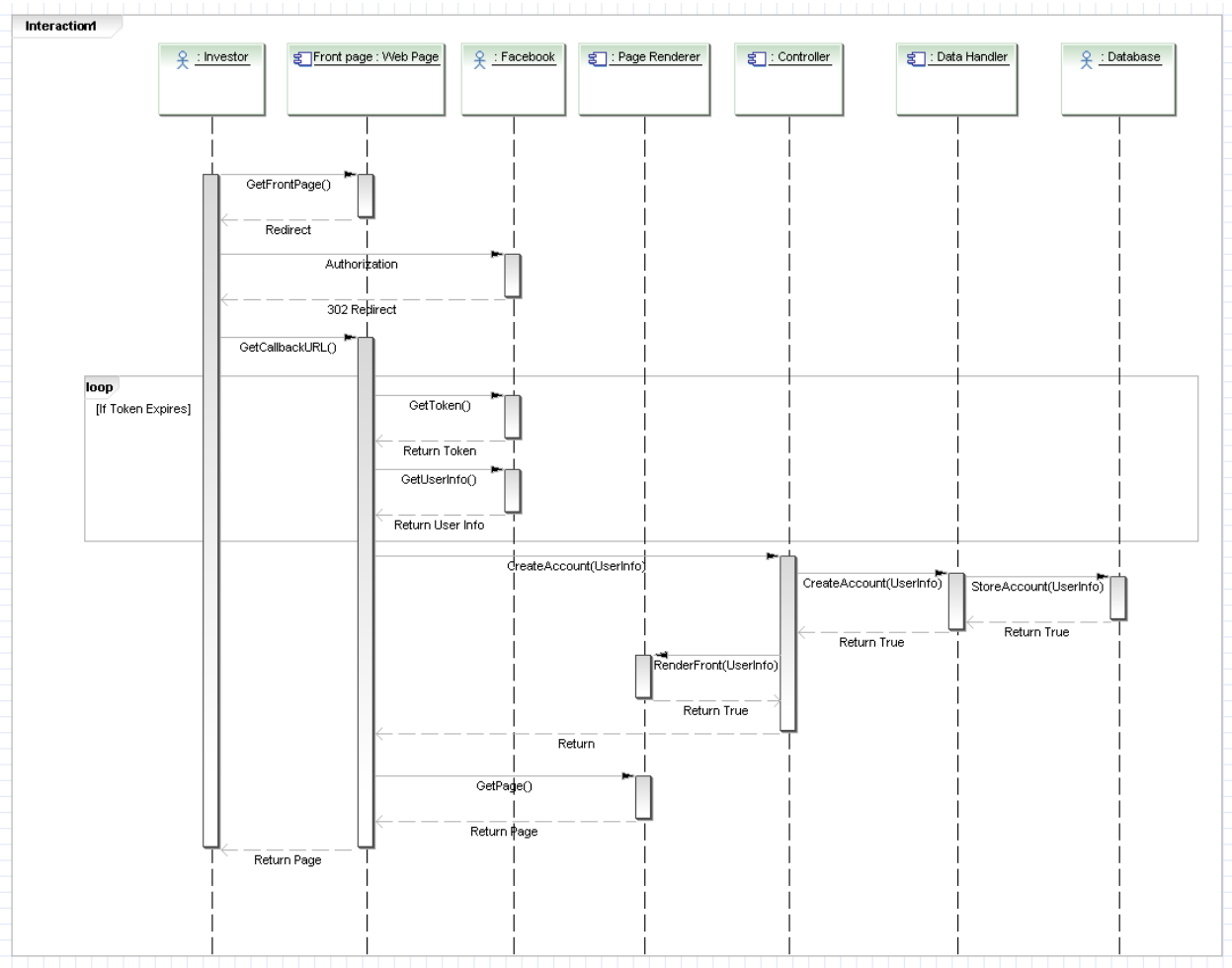


Figure 5: Interaction Diagram for Register (From Report 1 Fig. 6)

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 DataHandler (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.

1.5 Use Case 8/11: Create League/Fund
Use Case 13/20: Manage League/Fund

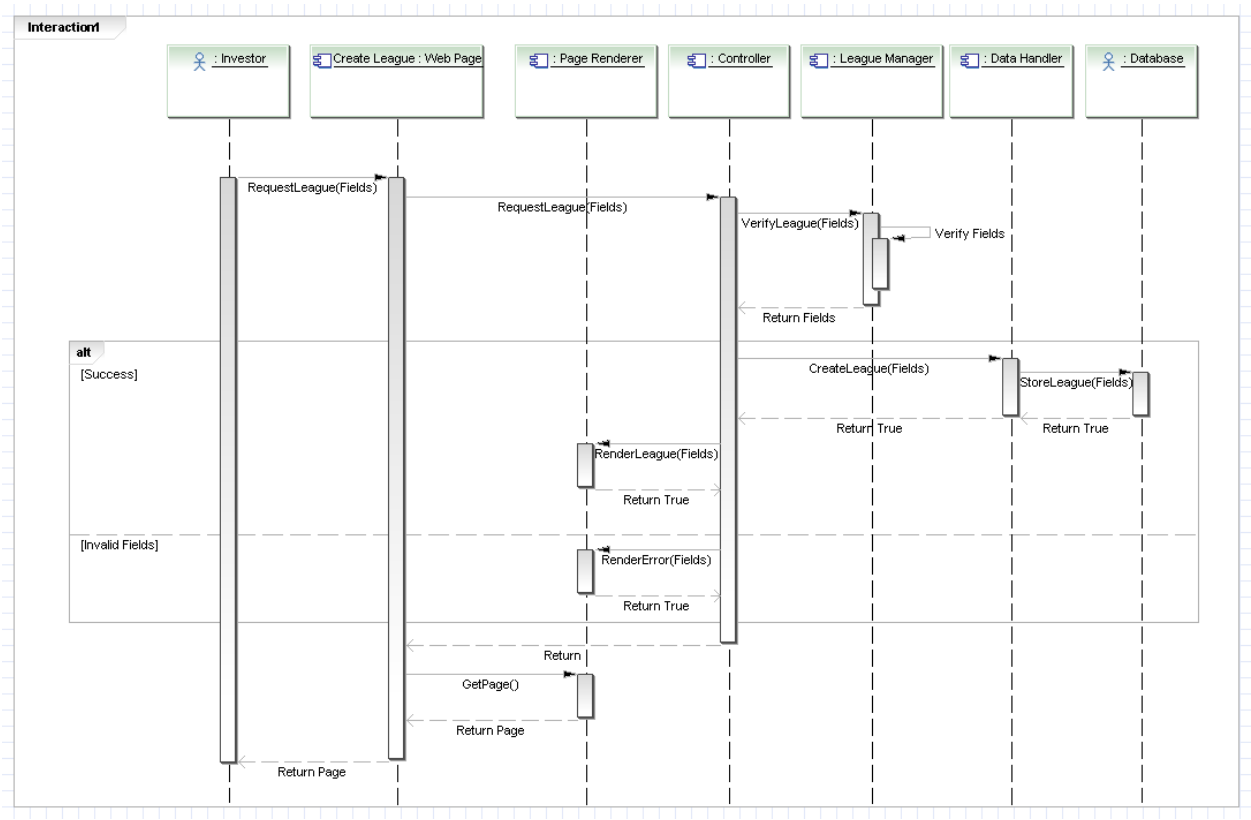


Figure 6: Interaction Diagram for Create League (From Report 1 Fig. 7)

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 FundHandler or the LeagueManager 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 DataHandler 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.

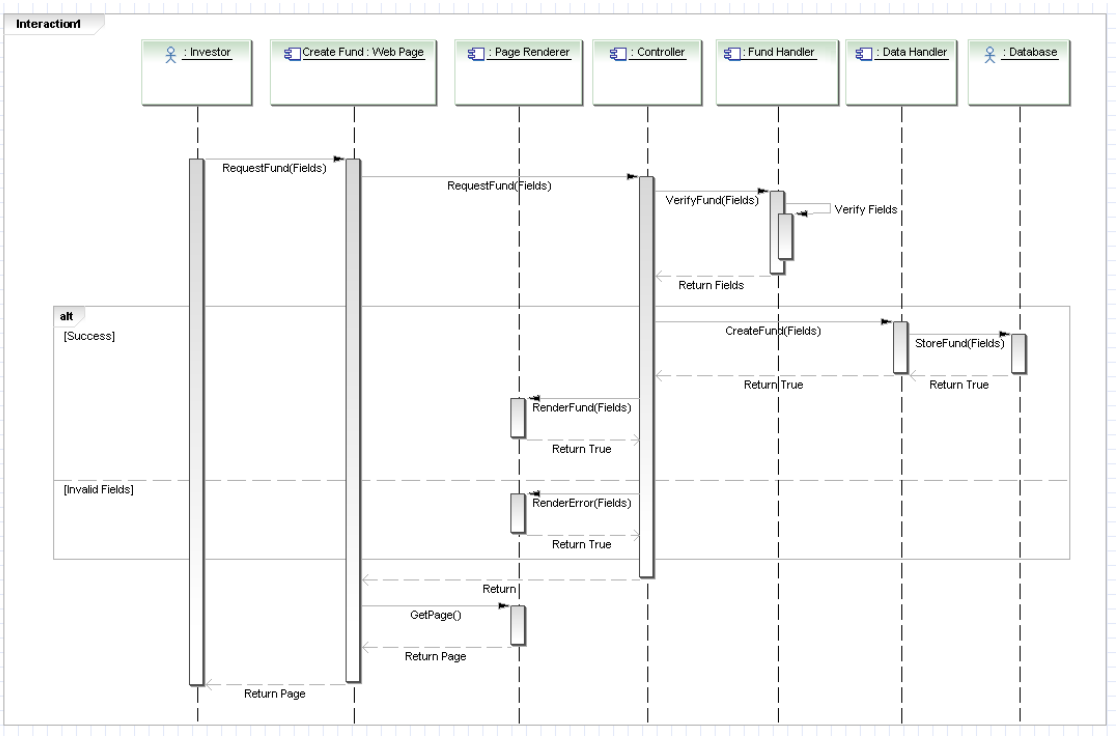


Figure 7: Interaction Diagram for Create Fund (From Report 1 Fig. 8)

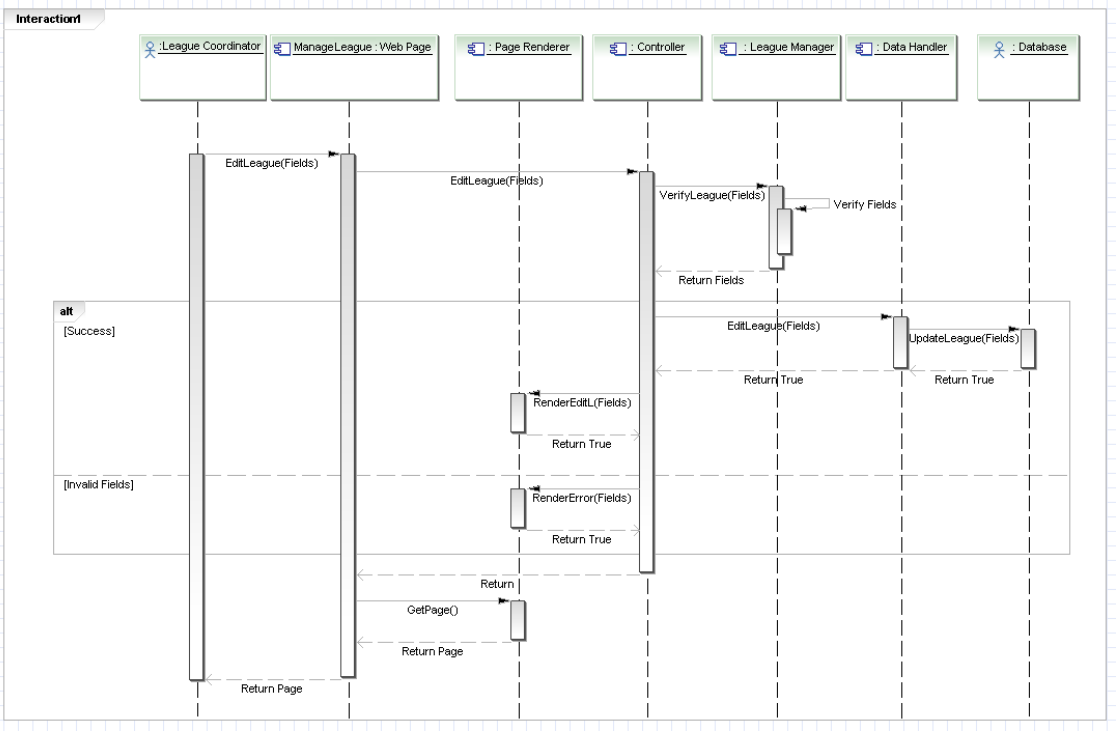


Figure 8: Interaction Diagram for Manage League (From Report 1 Fig. 9)

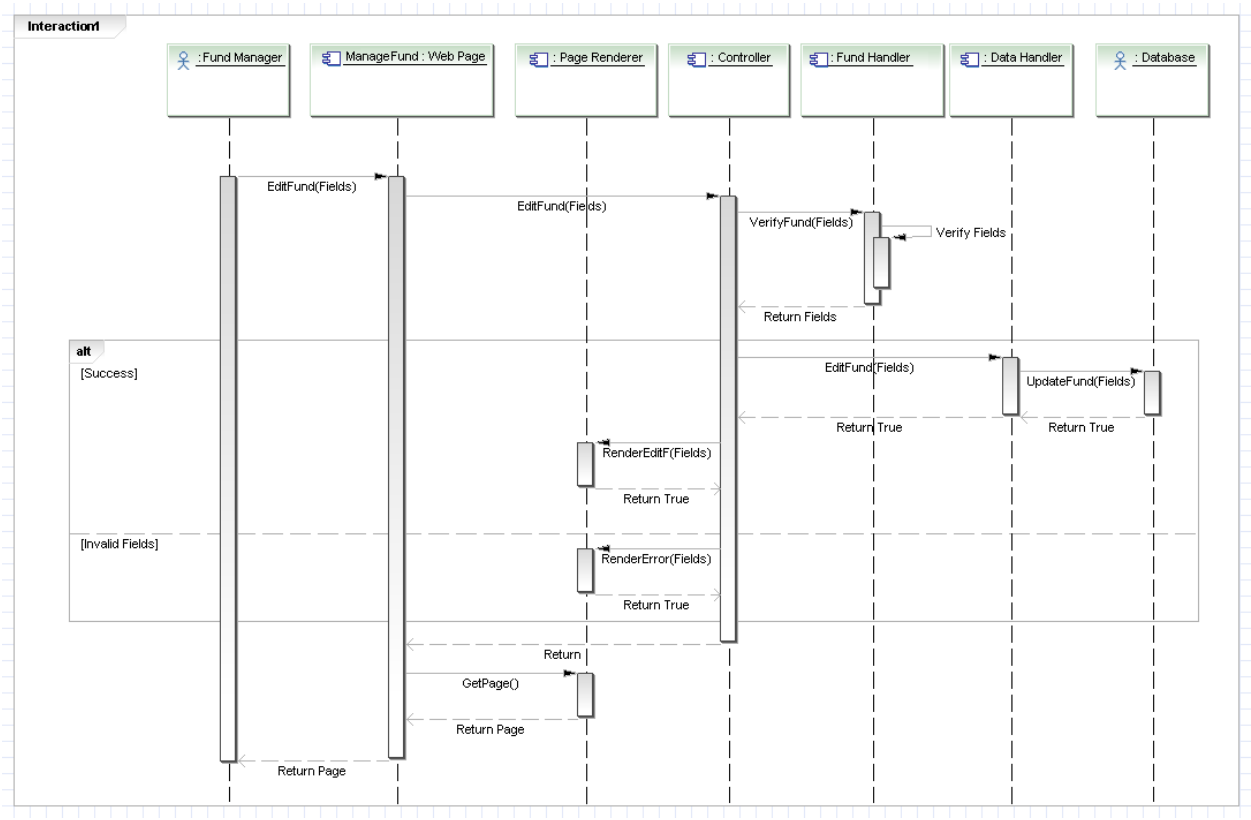


Figure 9: Interaction Diagram for Manage Fund (From Report 1 Fig. 10)

2 Alternative Designs

As you can see, there are many objects that are reused multiple times in each of the interaction diagrams above. Since each object was only assigned the responsibilities need to accomplish a specific purpose, it is able to be used many times for different scenarios. This is the benefit of using the design decisions of Expert Doer and High Cohesion, at the expense of Loose Coupling.

The only new object that was introduced from our previous iterations (report 1) was the Controller. Initially, we had the functionality of validity checking built into the Controller (which was then named the Validity Checker) as shown in the diagrams below. However, in spirit of high cohesion, we allowed the Validity Checker to carry out the sole task of verifying the ticket to see if it supports a valid transaction. This frees up the Validity Checker to also be flexible so it can be use to both buy and sell stocks. The need for a seperate controller object became apparent when creating the interaction diagrams because we realized that the Web Site needed one object that initiated multiple requests rather than many objects initiating some requests to get the task done. This falls in line with the idea of

the Expert Doer principle. With this new model, objects do not take on too many Type 2 responsibilities since the Controller now specializes in doing it for them.

(Please see below for the alternate diagrams without the controller)

Some other alternatives (not pictured) were also considered during the design process. For example, in one design, we considered having Stock Query query the Stock Info Provider periodically to get the prices for all stocks and store them in the database. For our app to get a stock price, it would get it from our database instead. We ultimately decided against this since it would unnecessarily increase the size of our database and would be too complicated an implementation in the early stages of development. We will possibly revisit this implementation in the future since it would reduce our application's latency by not having to connect to an external server each time a trade needs to be made, and to reduce the overall volume of queries made to the external API once the app becomes popular.

2.1 Use Case 1: Buy Stocks

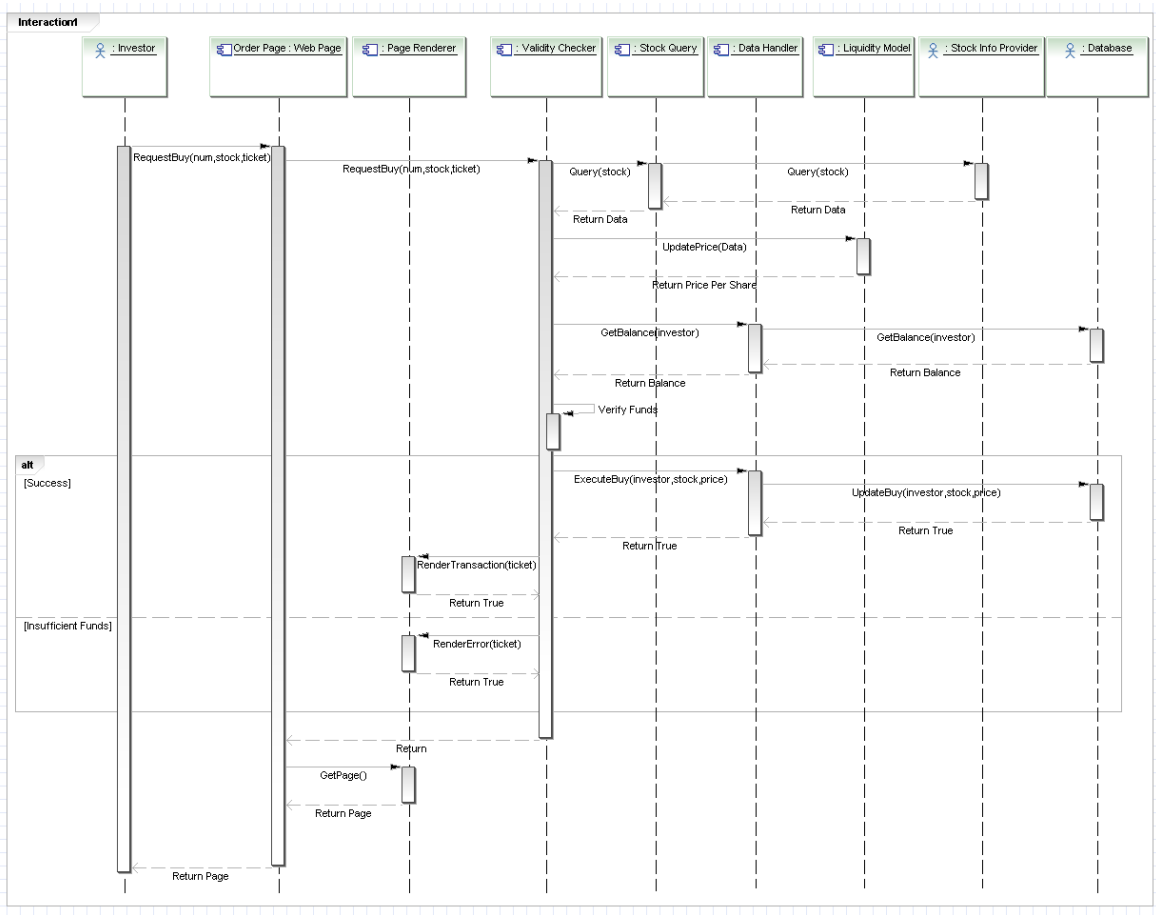


Figure 10: Interaction Diagram for Buy Stocks (Alternate)

2.2 Use Case 2: Sell Stocks

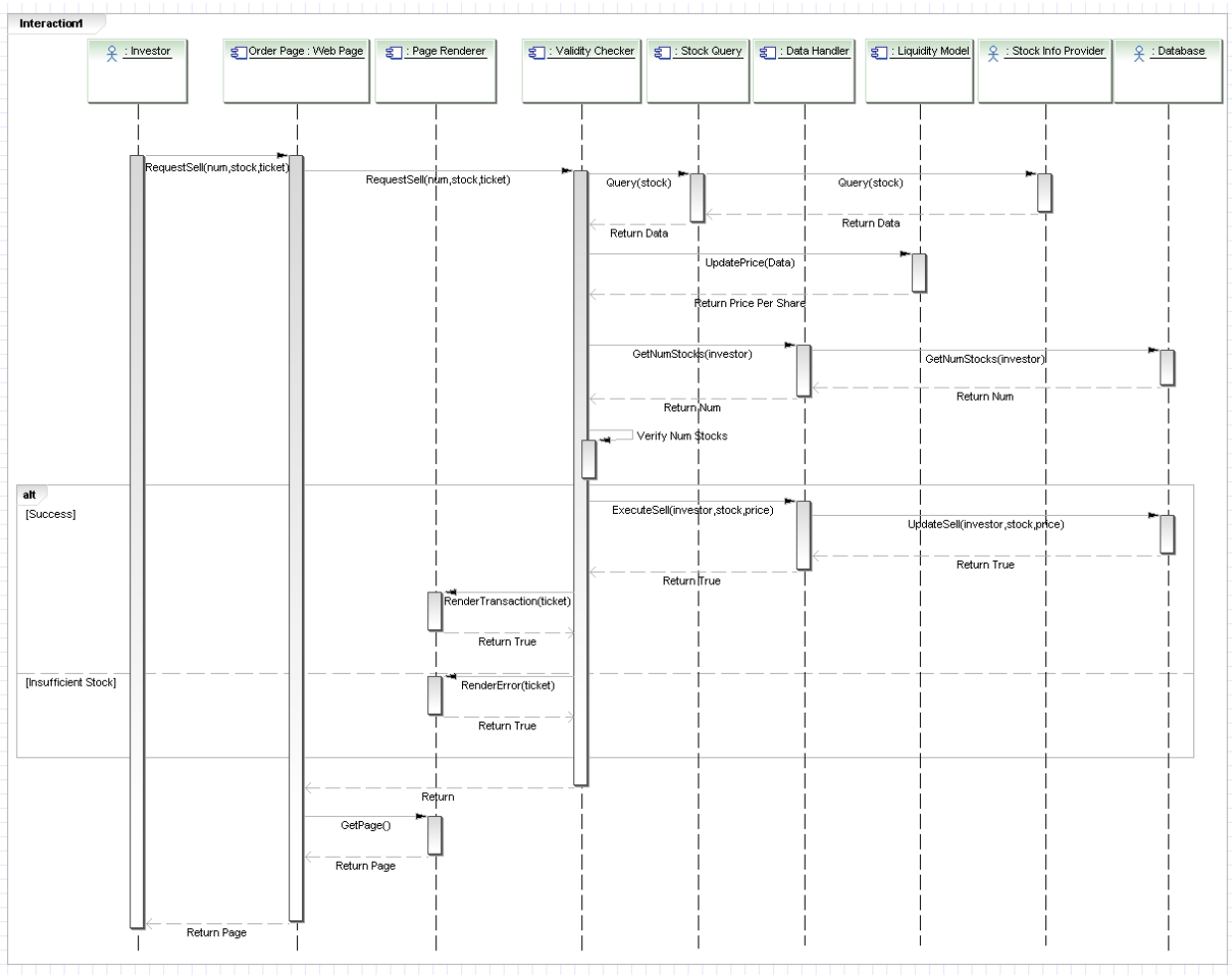


Figure 11: Interaction Diagram for Sell Stocks (Alternate)

2.3 Use Case 3: Query Stocks

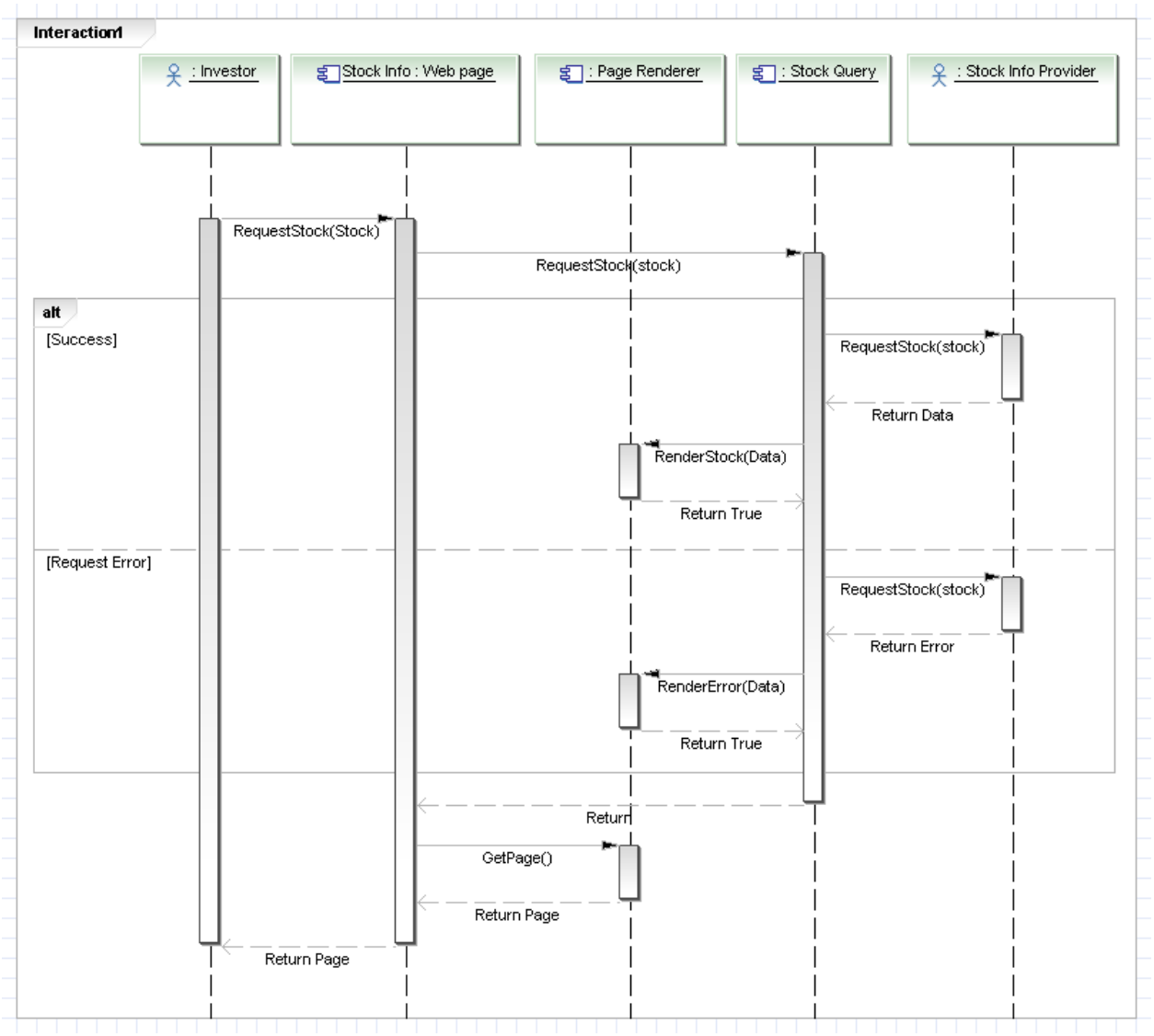


Figure 12: Interaction Diagram for Query Stocks (Alternate)

2.4 Use Case 5: View Portfolio

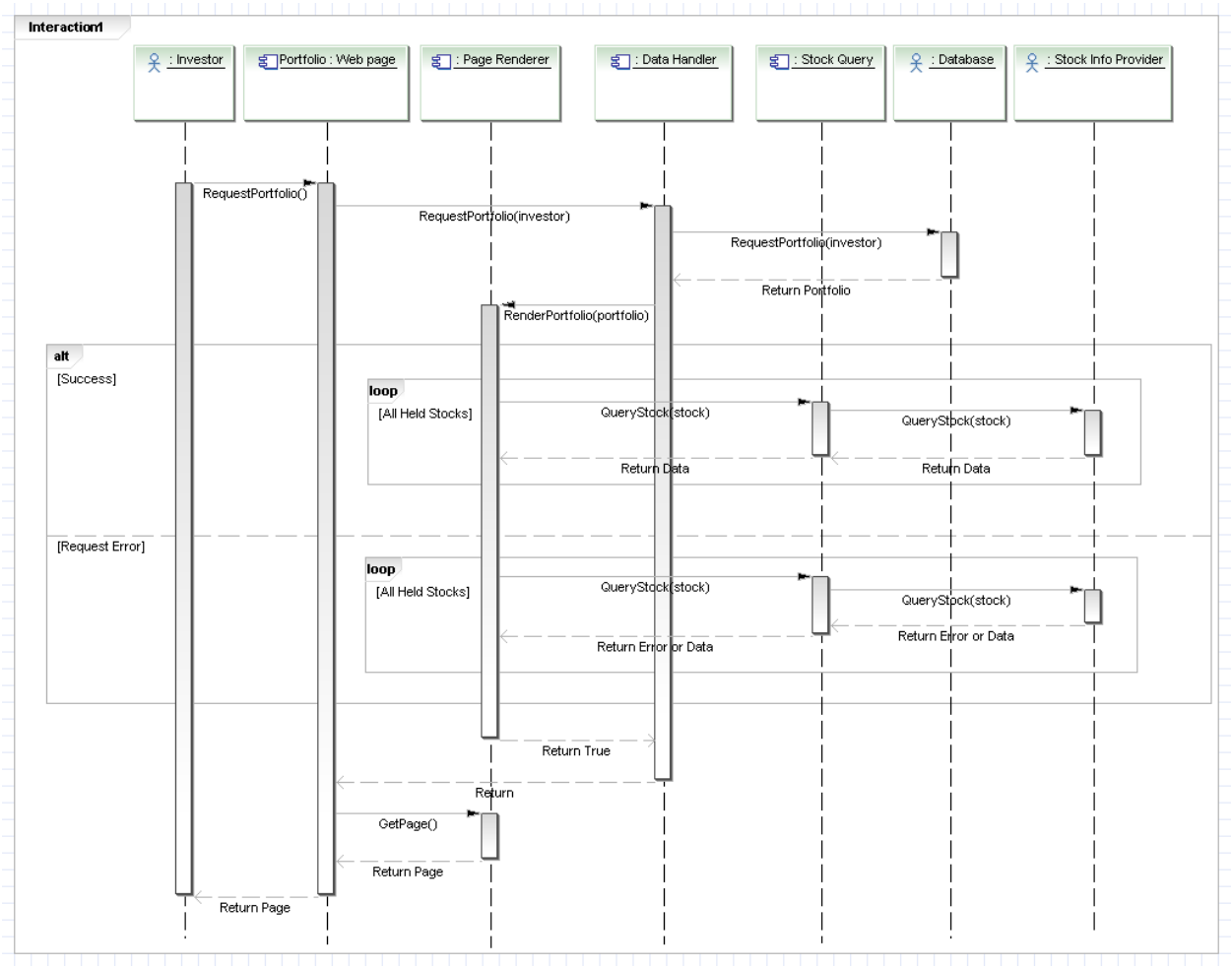


Figure 13: Interaction Diagram for View Portfolio (Alternate)

2.5 Use Case 8: Create League

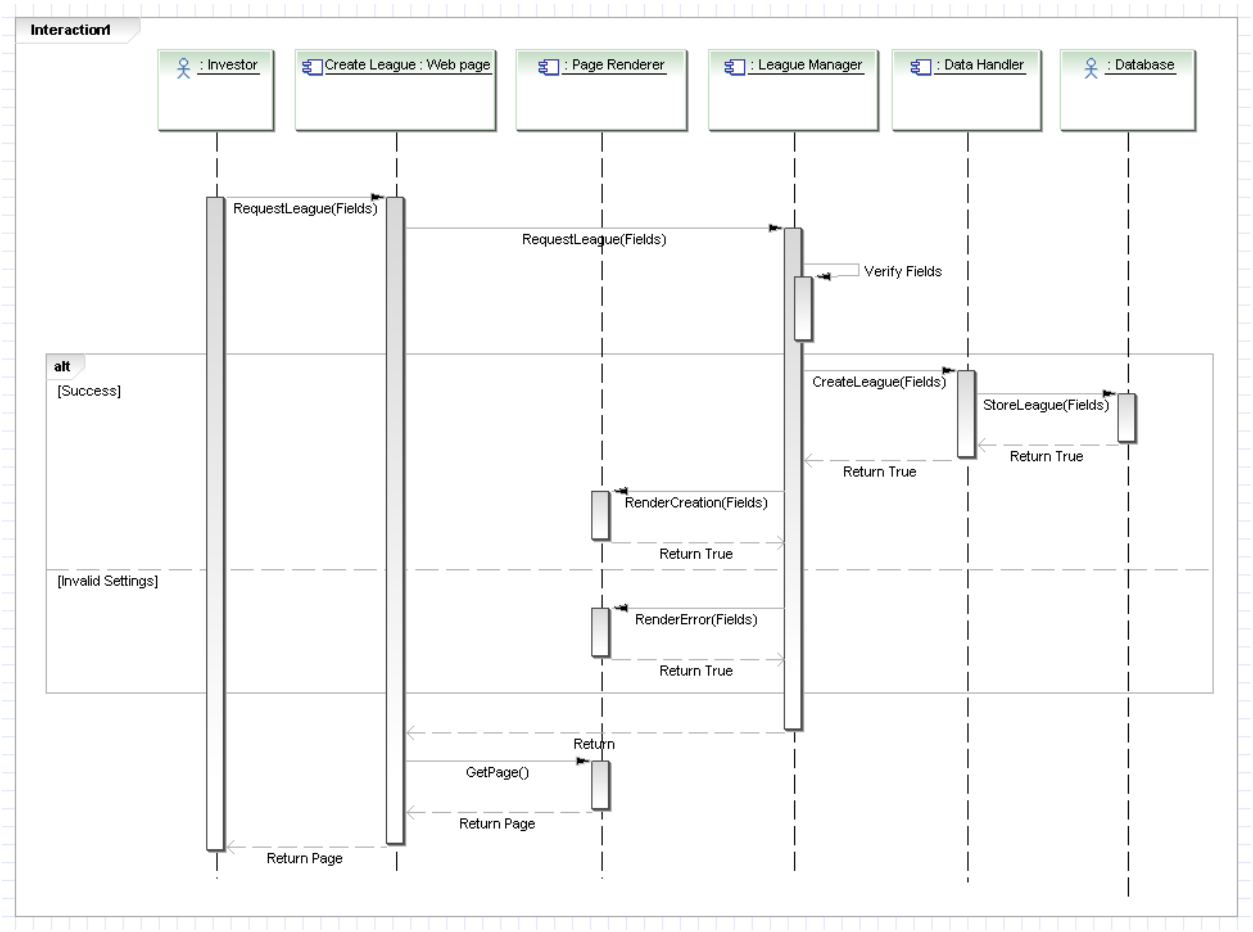


Figure 14: Interaction Diagram for Create League (Alternate)

2.6 Use Case 11: Create Fund

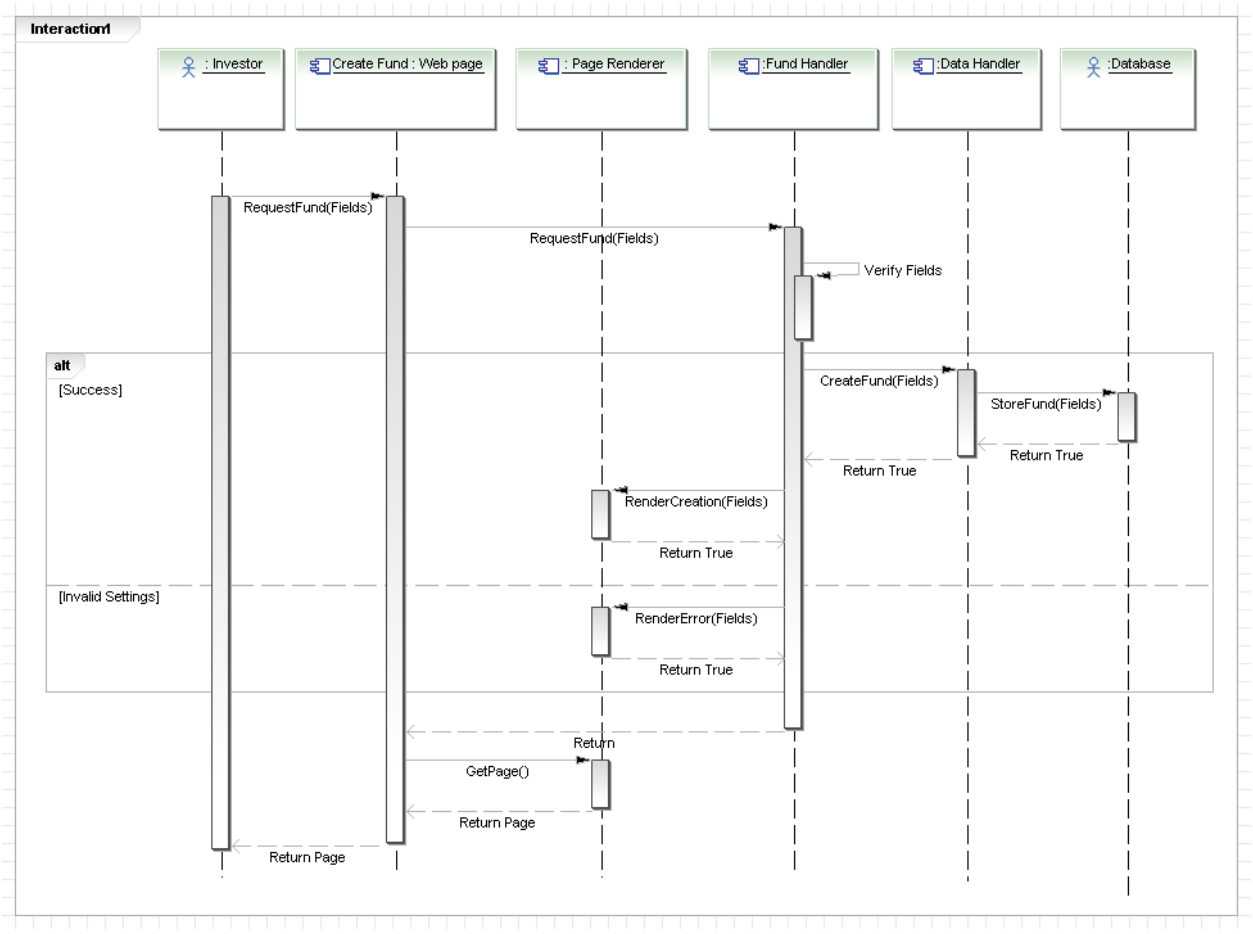


Figure 15: Interaction Diagram for Create Fund

2.7 Use Case 13: Manage League

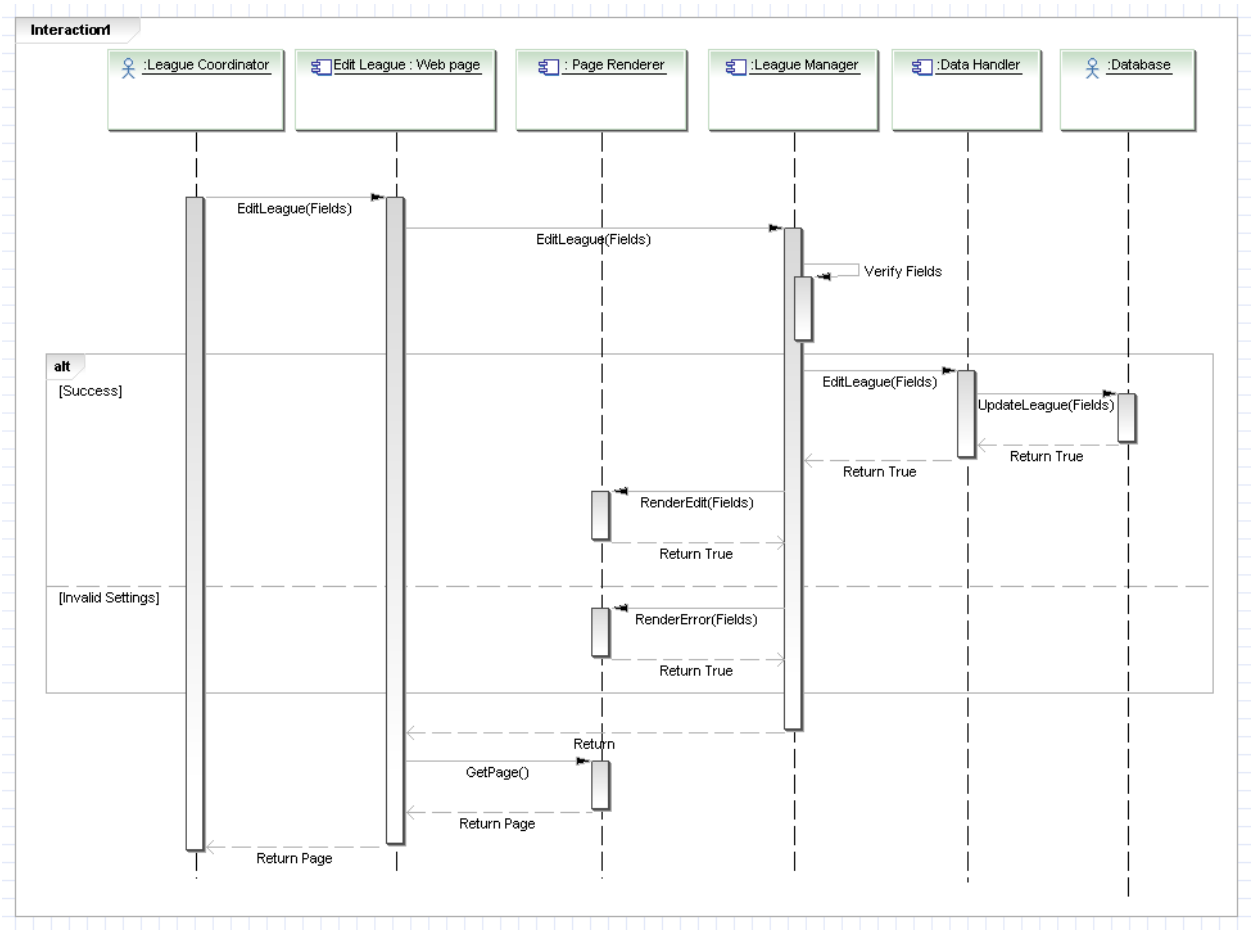


Figure 16: Interaction Diagram for Manage League (Alternate)

2.8 Use Case 20: Manage Fund

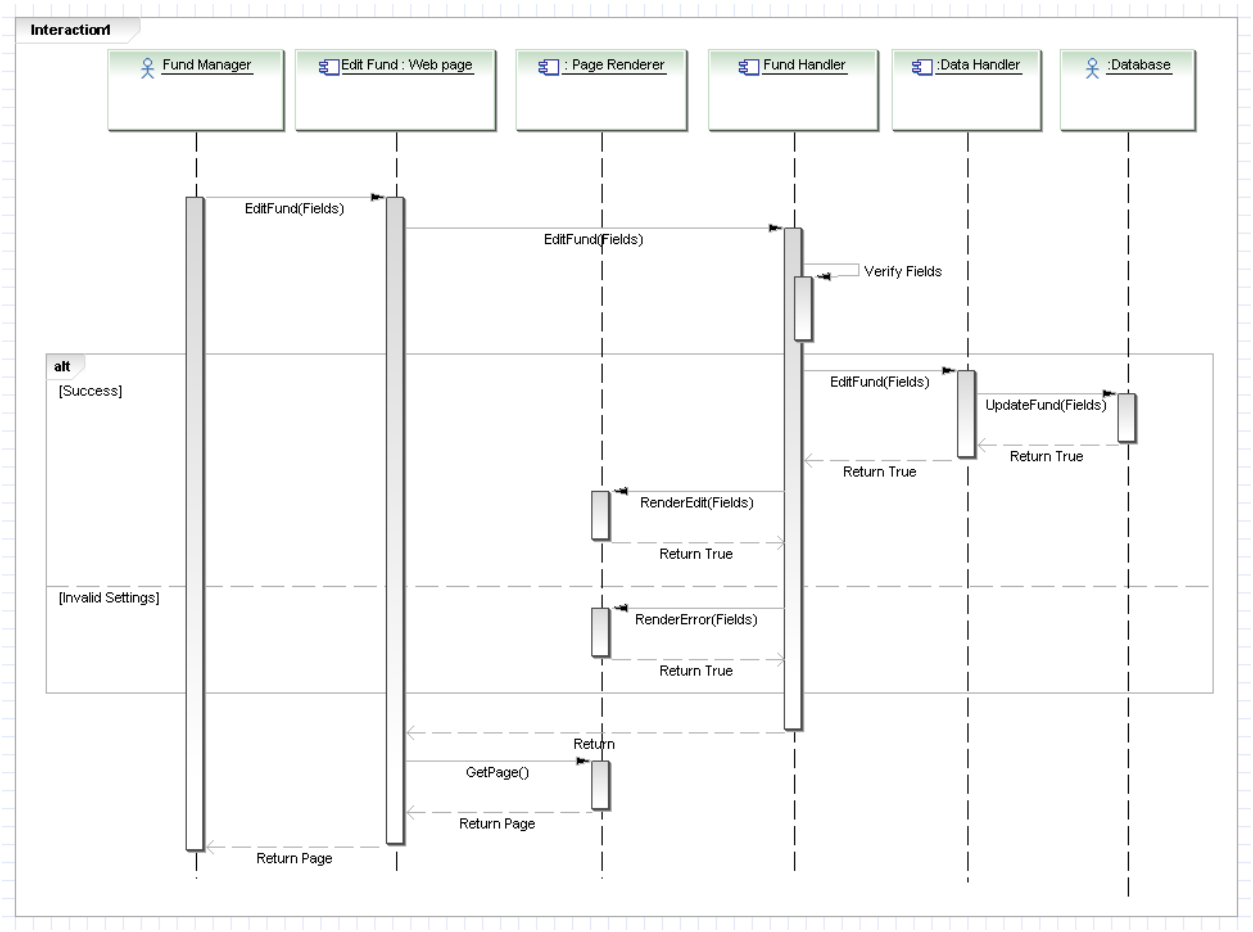
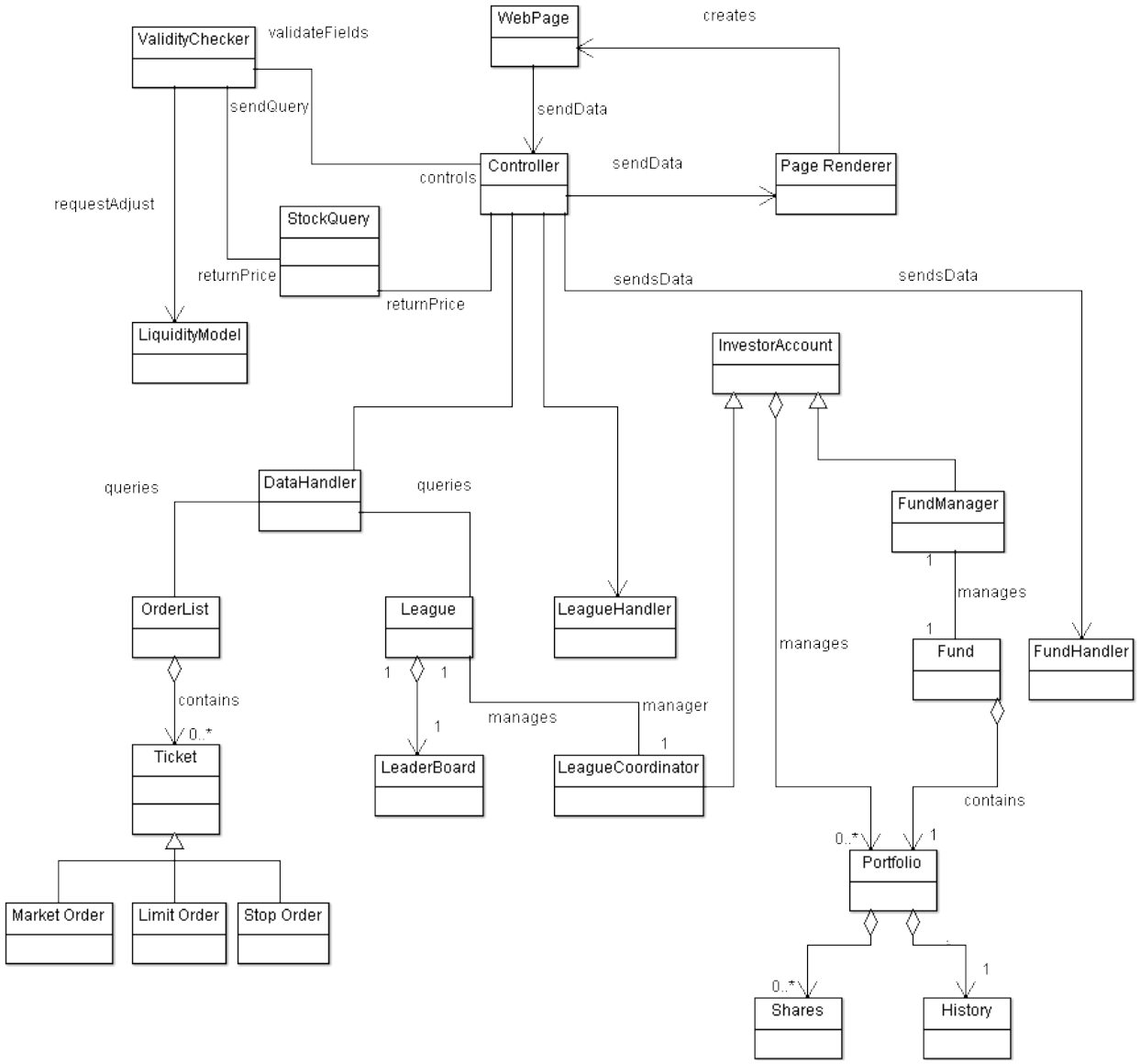


Figure 17: Interaction Diagram for Manage Fund (Alternate)

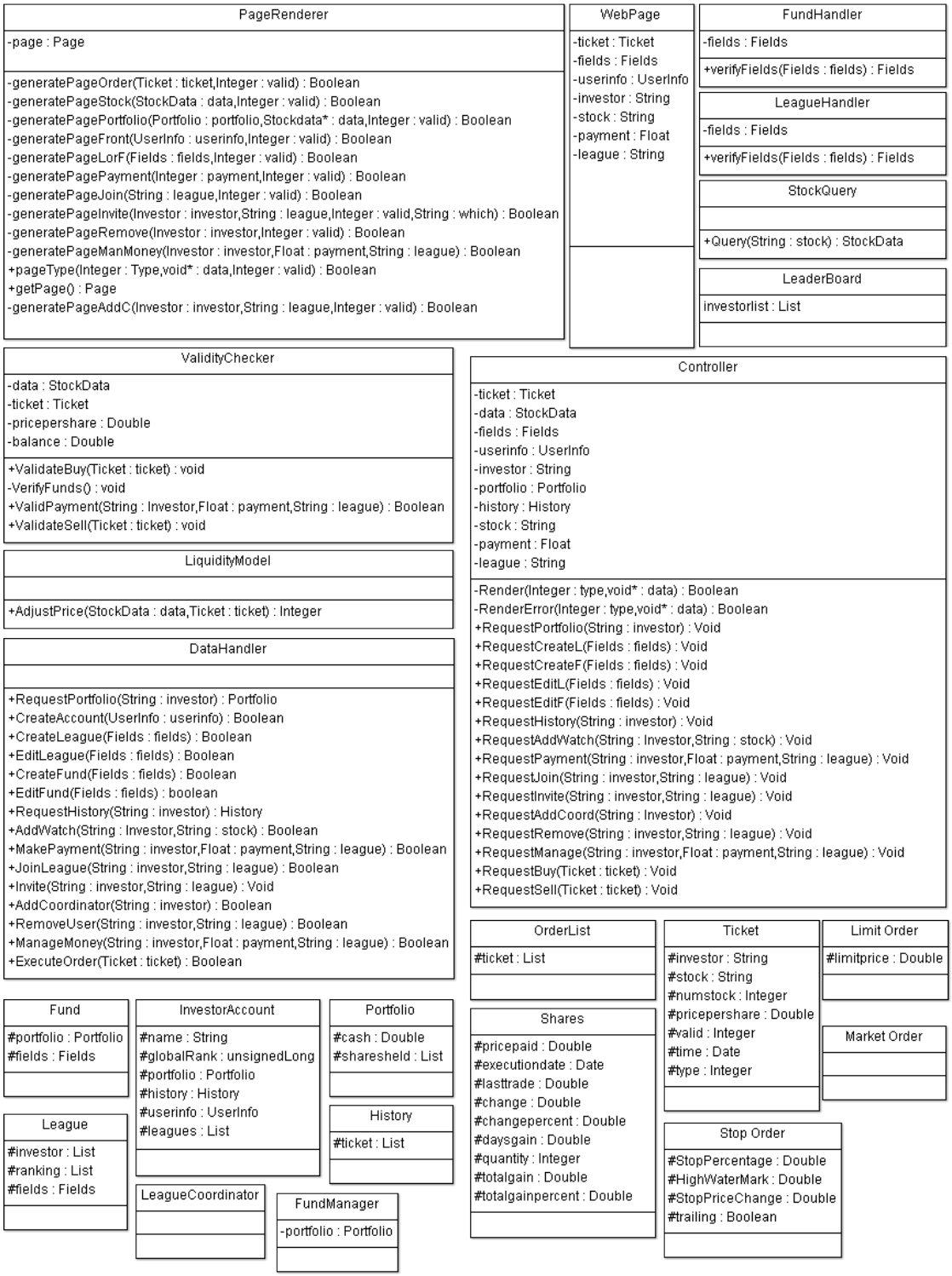
3 Class Diagram and Interface Specification

3.1 Class Diagram

The following class diagram shows the relations between classes. The class diagrams are collapsed to show only the names of the classes. The class attributes and methods are listed in Section 3.2. Several classes shown here were not present in the domain model. This is explained under the Traceability Matrix.



3.2 Data Types and Operation Signatures



3.2.1 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
- **payment : Float**
  This is a copy of the amount that the investor is submitting as payment to a league
- **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 (for example portfolio page), 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.
+ **RequestAddWatch(String : investor, String : stock) : Void**
  This method is the method that the web page calls in order to add a stock to the watchlist.
+ **RequestPayment(String : investor, Float : payment, String : league) : Void**
  This method is the method that the web page calls in order to make a payment to a league.

+ **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.
+ **RequestRemove(String : investor, String : league) : Void**
  This method is the method that the web page calls in order to remove an investor from a league.
+ **RequestManage(String : investor, Float : payment, String : league) : Void**
  This method is the method that the web page calls in order to distribute money to the winners.

3.2.2 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.
– **generatePagePayment(Integer : payment, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of a payment to a 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.
– **generatePageRemove(Investor : investor, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of removing an investor from a league.
– **generatePageManMoney(Investor : investor, Float : payment, String : league) : Boolean**
This method is called in order to render a page displaying the results of distributing money to the winners.
– **generatePageAddC(Ingestor : investor, String : league, Integer : valid) : Boolean**
This method is called in order to render a page displaying the results of adding a coordinator to a league.
+ **pageType(Inveger : 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.

3.2.3 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.
+ **AddWatch(String : investor, String : stock) : Boolean**
  This method is called to request that the database add a stock to the watchlist of an investor.
+ **MakePayment(String : investor, Float : payment, String : league) : Boolean**
  This method is called to request that the payment for a league be updated in 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.
+ **AddCoordinator(String : investor) : Boolean**
  This method is called to request that a coordinator be added to a league in the database.
+ **RemoveUser(String : investor, String : league) : Boolean**
  This method is called to request that an investor be removed from a specific league in the database.
+ **ManageMoney(String : investor, Float : payment, String : league) : Boolean**
  This method is called to request that the database allocate money to the specified investor.

3.2.4 ValidityChecker

Attributes

The validity checker holds the below attributes that it uses in calculations to determine if an order is valid or not.

− **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.
+ **ValidPayent(String : Investor, Float : payment, String : league) : Boolean**
  This method is called by the controller to determine if the investor can pay the specified amount to the league.
+ **ValidateSell(Ticket : ticket) : void**
  This method is called by the controller to determine if a sell is valid or not.

3.2.5 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.

3.2.6 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.

3.2.7 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.
− **payment : Float**
  This is the amount of payment that the investor enters to pay a league.
− **eague : String**
  This is the name of the league that the investor enters.

3.2.8 FundHandler

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.

3.2.9 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.

3.2.10 Leaderboard

Attributes

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

3.2.11 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).

3.2.12 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.

3.2.13 Portfolio

Attributes

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

3.2.14 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.

3.2.15 LimitOrder

Attributes

limitprice : Double

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

3.2.16 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.

3.2.17 OrderList

Attributes

ticket : List

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

3.2.18 History

Attributes

ticket : List

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

3.2.19 FundManager

Attributes

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

3.2.20 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

3.2.21 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.

3.2.22 Fund

Attributes

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

3.2.23 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.

3.3 Traceability Matrix

Class	WebPage	Page Renderer	ValidityChecker	StockQuery	DataHandler	LiquidityModel	LeagueManager	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			
FundManager					x			
Fund					x			
Shares					x			
Histor					x			
OrderList					x			
Ticket					x			
Market Order					x			
Limit Order					x			
Stop Order					x			

Many of the classes map back to the DataHandler concept since they contain data that is queried by the DataHandler. These classes should have been represented as separate entities in the domain model and shows a shortcoming of the domain model that was derived earlier. These concepts will be added to the domain model to reflect the classes needed to store the data.

4 System Architecture and System Design

4.1 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.

4.1.1 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.

4.1.2 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.

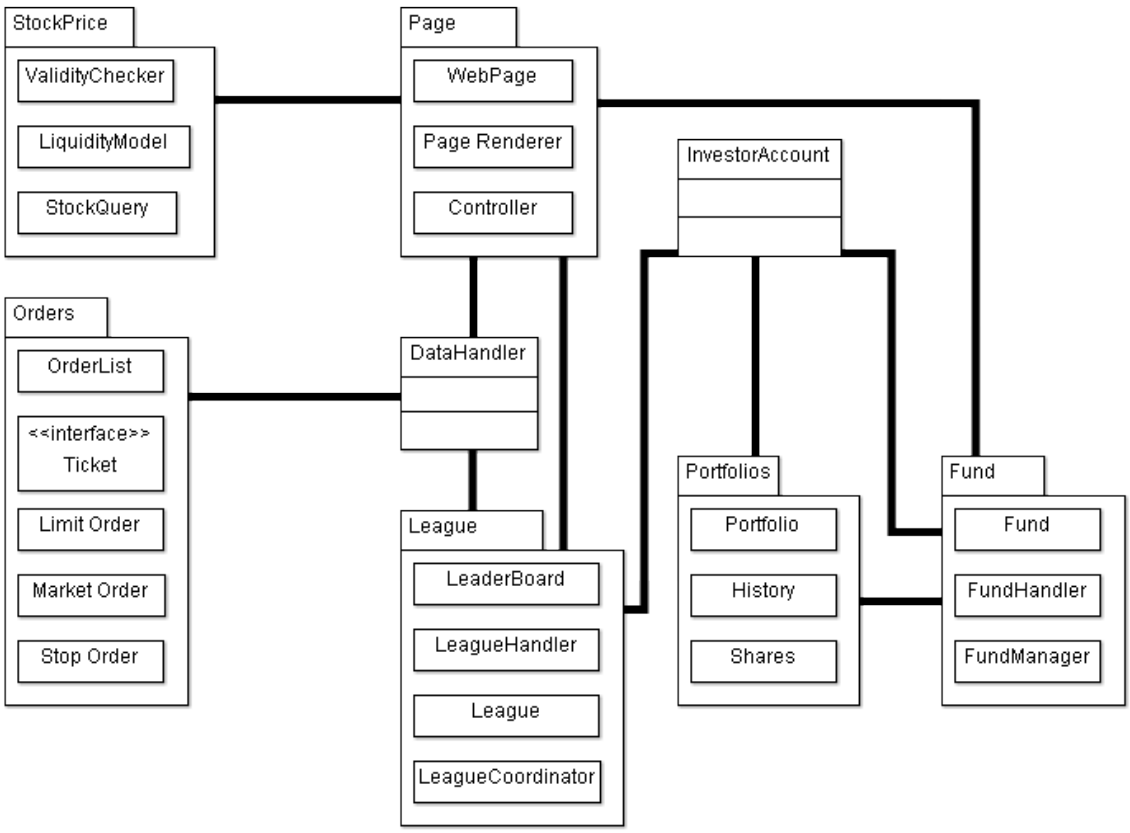
4.1.3 Event-driven Architecture

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.

4.1.4 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.

4.2 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.

4.3 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. A software engineering computer was provided by the ECE department (sweng-1.engr.rutgers.edu) which can be accessed within the Rutgers network via ssh group6@sweng-1.engr or by visiting apps.rutgers.edu and then http://sweng-1.engr.rutgers.edu/g̃roup6. 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 these two servers. For capabilities stored on the Heroku server, the information is stored in the cloud provided by Heroku. For capabilities stored on the ECE server, the data is stored on sweng-1. The exact location of each subsystem has not been fully determined yet, but they will be between the two choices discussed above.

4.4 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 data base 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

4.5 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.

4.6 Global Control Flow

Bears & Bulls is an event-driven system which waits for certain actions to occur and responds the them. To update a users portfolio, the system waits for the user to attempt to view his portfolio. 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.

4.7 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.

5 Algorithms and Data Structures

5.1 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. For information regarding the mathematical model, please refer to the Bears & Bulls Report 1. 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.

5.2 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.

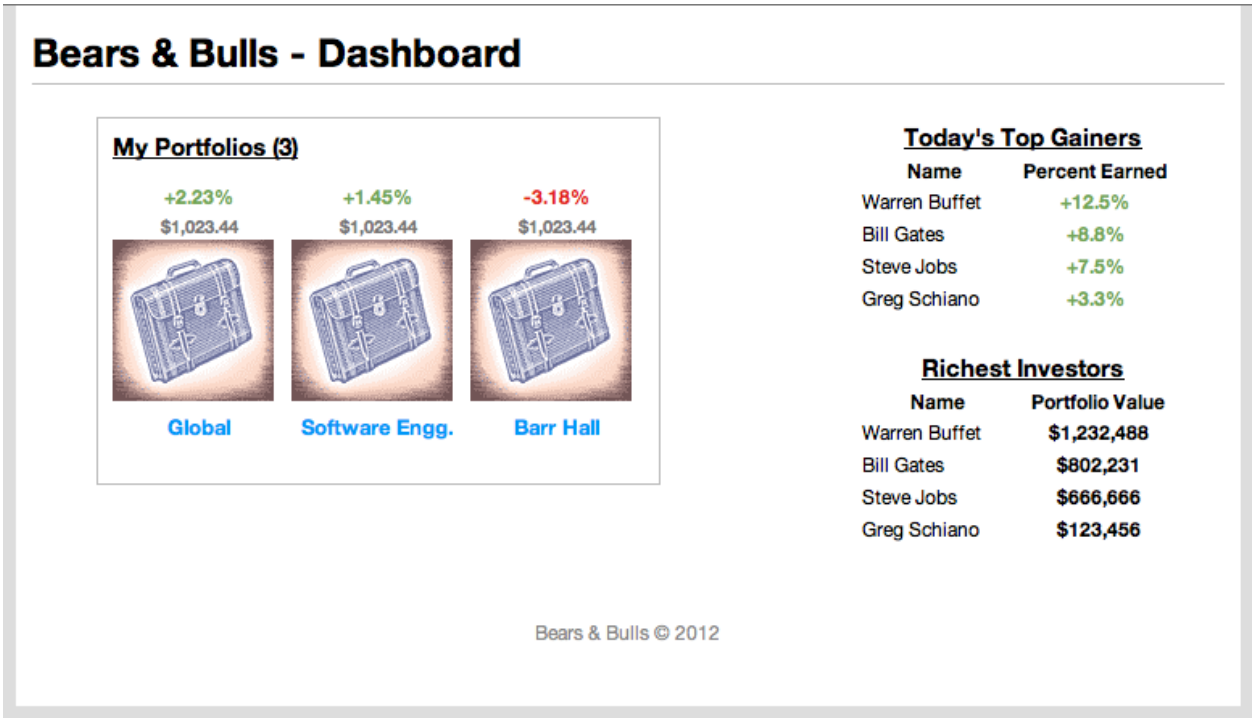
The linked-list structure was chosen because it supports easy insertion and deletion from the list. The linked list was chosen over the queue because the queue does not support deletion from any point within the queue. This is necessary because the orders are not necesarrily executed in the order they 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.

6 User Interface Design and Implementation

The user interface and user effort approximation has not yet varied from what was outlined in Report 1. The initial mockups provided in Report 1 are in the process of being implemented in HTML and CSS. Thus far, the basic page skeleton that will be reused for most pages on the site and the Dashboard page have been completed. An HTML implementation of the entire website is scheduled to be completed at a later time before Demo 1. Currently, the site front-end is temporarily uploaded and can be interacted with at ¡http://high-window-8945.herokuapp.com/temp/¿.



7 Design of Tests

7.1 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.

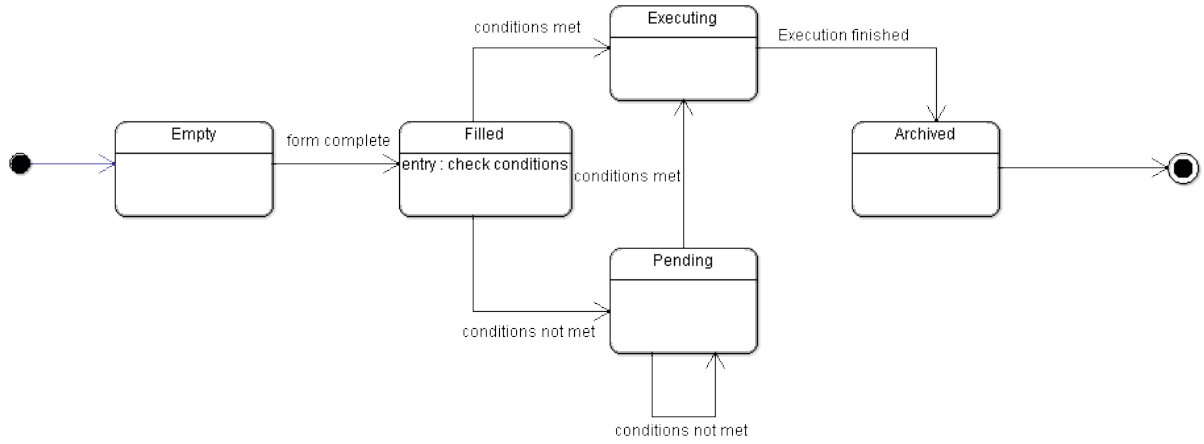


Figure 18: State Diagram of Order Ticket

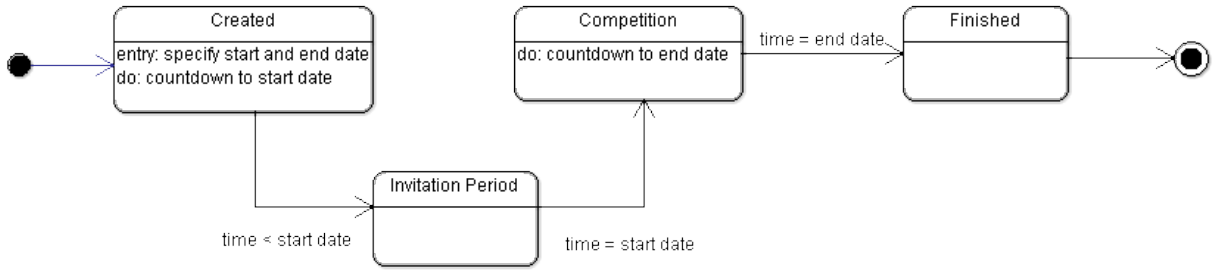


Figure 19: State Diagram of League

7.2 Unit Tests

7.2.1 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::RequestAddWatch(String : investor, String : stock) : Void **Pass/Fail Criteria:** The test passes if the request to add a watch on the correct stock for the correct investor 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, AddWatch of DataHandler is called
–Call Function (Fail)	–Data does not get sent, AddWatch of DataHandler is not called

Test-case Identifier: TC-10 **Function Tested:** Controller::RequestPayment(String : investor, Float : payment, String : League) : Void **Pass/Fail Criteria:**The test passes if the request to make the correct payment for the correct investor in 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, MakePayment of DataHandler is called
–Call Function (Fail)	–Data does not get sent, MakePayment of DataHandler is not called

Test-case Identifier: TC-11 **Function Tested:** Controller::RequestJoin(String : investor, String : League) : Void **Pass/Fail Criteria:** The test passes if the request to make the correct investor 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, JoinLeague of DataHandler is called
–Call Function (Fail)	–Data does not get sent, JoinLeague of DataHandler is not called

Test-case Identifier: TC-12 **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-13 **Function Tested:** Controller::RequestAddCoord(String : investor) : Void **Pass/Fail Criteria:** The test passes if the request to add the correct coordinator 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, AddCoordinator of DataHandler is called
–Call Function (Fail)	–Data does not get sent, AddCoordinator of DataHandler is not called

Test-case Identifier: TC-14 **Function Tested:** Controller::RequestRemove(String : investor, String : league) : Void **Pass/Fail Criteria:** The test passes if the request to remove the correct investor from 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, RemoveUser of DataHandler is called
–Call Function (Fail)	–Data does not get sent, RemoveUser of DataHandler is not called

Test-case Identifier: TC-15 **Function Tested:** Controller::RequestManage(String : investor, Float : payment, String : league) : Void **Pass/Fail Criteria:** The test passes if the request to distribute money to the winners 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, ManageMoney of DataHandler is called
–Call Function (Fail)	–Data does not get sent, ManageMoney of DataHandler is not called

Test-case Identifier: TC-16 **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-17 **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

7.2.2 PageRenderer

Test-case Identifier: TC-18 **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-19 **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	**Expected Results**
–Request a stock page (Pass)	–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-20 **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	**Expected Results**
–Request a portfolio page (Pass)	–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-21 **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	**Expected Results**
–Request a front page (Pass)	–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-22 **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-23 **Function Tested:** PageRenderer::generatePagePayment(Integer:payment, Integer:valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request for a payment page and a payment page is generated. Unsuccessful if page cannot be generated.	
Test Procedure –Request a payment page (Pass)	**Expected Results** –Page Renderer returns true and league page is created
–Request a payment page (Fail)	–Page Renderer returns false and page is not generated

Test-case Identifier: TC-24 **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 Request a join page (Pass)	**Expected Results** 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-25 **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 –Request a league invite page (Pass)	**Expected Results** –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-26 **Function Tested:** PageRenderer::generatePageRemove(Investor: investor, Integer: valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request to remove a user and a remove user page is generated. Unsuccessful if page cannot be generated.	
Test Procedure	**Expected Results**
–Request a remove user page (Pass)	–Page Renderer returns true and remove user page is created
–Request a remove user page (Fail)	–Page Renderer returns false and page is not generated

Test-case Identifier: TC-27 **Function Tested:** PageRenderer::generatePageManMoney(Investor: investor, Integer: payment, String: which):Boolean **Pass/fail Criteria:** The test passes if the system enters a request to manage money and a correct money management page is generated. Unsuccessful if page cannot be generated.	
Test Procedure	**Expected Results**
–Request a money management page (Pass)	–Page Renderer returns true and manage money page is created
–Request a money management page (Fail)	–Page Renderer returns false and page is not generated

Test-case Identifier: TC-28 **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-29 **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

Test-case Identifier: TC-30 **Function Tested:** PageRenderer::generatePageAddC(Investor: Investor, String: league, arg4, Integer: valid):Boolean **Pass/fail Criteria:** The test passes if the system enters a request to add a page and it is created successfully. Unsuccessful if page cannot be generated	
Test Procedure	**Expected Results**
–Request a page to be created (Pass)	–Page Renderer returns true if add coordinator page is generated
–Request a page to be created (Fail)	–Page Renderer returns false and page is not generated

7.2.3 DataHandler

Test-case Identifier: TC-31 **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-32 **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) –Request portfolio updaten (Fail)	–DataHandler requests portfolio data and returns it from the database. –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-33 **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) —Call Function (Fail)	–DataHandler requests account creation and returns true if account creation successful –If the request for account creation is unsuccessful, return false

Test-case Identifier: TC-34 **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) –Request to create a league (Fail)	–DataHandler requests league creation and returns true if league creation successful –If the request for league creation is unsuccessful, return false

Test-case Identifier: TC-35 **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-36 **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-37 **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-38 **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-39 **Function Tested:** DataHandler::AddWatch(String: Investor, String: Stock): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests to addstock to watchlist and it occurs. Unsuccessful if stock not added.	
Test Procedure	**Expected Results**
–Request to watch a stock (Pass) –Request to watch a stock (Fail)	–DataHandler adds stock to watchlist and returns true. –DataHandler is unable to add stock to watchlist and returns false. If the stock entered is invalid, return false.

Test-case Identifier: TC-40 **Function Tested:** DataHandler::MakePayment(String: investor, Float: payment, String: League): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests that payment is updated in database and it occurs. Unsuccessful if it doesnt happen.	
Test Procedure	**Expected Results**
–Submit payment (Pass) –Submit payment (Fail)	–Data handler submits payment to the league and returns true. –If payment isnt processed, return false.

Test-case Identifier: TC-41 **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)	–DataHandler updates information in database about the league and returns true.
–Request to join league (Fail)	–If joining the league encounters a problem, return false.

Test-case Identifier: TC-42 **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)	–DataHandler adds the invite to investors account in database.
–Add invite to investors account (Fail)	–If unable to add invite in database, display error saying that invite wasnt added to investors account.

Test-case Identifier: TC-43 **Function Tested:** DataHandler::AddCoordinator(String: Investor): Boolean **Pass/Fail Criteria:** The test passes if test stub requests that a coordinator be added to a league in the database and is added.	
Test Procedure	**Expected Results**
–Request for an investor to be promoted to coordinator (Pass)	–DataHandler adds investor to list of coordinators and returns true.
–Request for an investor to be promoted to coordinator (Fail)	–If unable to add investor to coordinator list in database, return false.

Test-case Identifier: TC-44 **Function Tested:** DataHandler::RemoveUser(String: Investor, String: League): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests that an investor be removed from a specific league in the database.	
Test Procedure	**Expected Results**
–Request to remove an investor from the league (Pass) –Request to remove an investor from the league (Fail)	–DataHandler removes user from league in database and returns true. –If the DataHandler is unable to remove user from the league, return false.

Test-case Identifier: TC-45 **Function Tested:** DataHandler::ManageMoney(String: Investor, Float: Payment, String: League): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests that the order is executed and the investor's portfolio is properly adjusted.	
Test Procedure	**Expected Results**
–Execute a ticket (Pass) –Execute a ticket (Fail)	–DataHandler changes investor's porfolio to reflect trade. –If unable to execute the trade, return false.

Test-case Identifier: TC-46 **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) –Allocate money to investor (Fail)	–DataHandler allocates money to the specified investor and returns true. –If unable to allocate money to investor, return false.

7.2.4 ValidityChecker

Test-case Identifier: TC-47 **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-48 **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-49 **Function Tested:** ValidityChecker::ValidPayment(String: Investor, Float: payment, String: league): boolean **Pass/Fail Criteria:** The test passes if the test stub determines the investor can pay the specified amount to the league. Unsuccessful if the investor cannot pay.	
Test Procedure	**Expected Results**
– Submit valid payment (Pass)	–Validity Checker determines payment is valid and returns true.
– Submit invalid payment (Fail)	–Validity Checker determines payment is invalid and returns false.

Test-case Identifier: TC-50 **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.

7.2.5 StockQuery

Test-case Identifier: TC-51 **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.

7.2.6 LiquidityModel

Test-case Identifier: TC-52 **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)	–LiquidityModel returns the adjusted price
–Request a price adjustment (Fail)	–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.

7.2.7 FundHandler

Test-case Identifier: TC-53 **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)	–FundHandler returns the valid fields.
–Request to verify fields (Fail)	–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.

7.2.8 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.

7.3 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.

7.4 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.

7.5 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).

7.6 Mathematical Model Testing

The following shows various test cases for the mathematical model governing price slippage. The model works adjust prices as expected to account for market conditions, however the price change is heavily dependant on the number of blocks that is being purchased. The largest percent changes occur when over 300 block sizes are being bought. This isn't too surprising since large sell offs or buys will likely signal to other traders to jump in on the trade on drive prices down or up respectively.

Case #	Price Total	Price Per Stock	Ask Price	Percent Change	Volatility	Liquidity	Block Size	Total Shares	# of Full Blocks	Bid-Ask Spread	Beta
Case 1	101.21	0.1012	0.1	1.21	0.5	50	300	1000	3	0.002	50
Case 2	1012.06	1.0121	1	1.21	0.5	50	300	1000	3	0.02	50
Case 3	10120.60	10.1206	10	1.21	0.5	50	300	1000	3	0.2	50
Case 4	101206.01	101.2060	100	1.21	0.5	50	300	1000	3	2	50
Case 5	1012060.10	1012.0601	1000	1.21	0.5	50	300	1000	3	20	50
Case 6	10120601.00	10120.6010	10000	1.21	0.5	50	300	1000	3	200	50
Case 7	1012.06	101.2060	100	1.21	0.5	50	3	10	3	2	50
Case 8	10120.60	101.2060	100	1.21	0.5	50	30	100	3	2	50
Case 9	541187.51	108.2375	100	8.24	0.5	50	300	5000	16	2	50
Case 10	1179957.16	117.9957	100	18.00	0.5	50	300	10000	33	2	50
Case 11	79714941.17	797.1494	100	697.15	0.5	50	300	100000	333	2	50
Case 12	100.12	0.1001	0.1	0.12	0.5	500	300	1000	3	0.0002	50
Case 13	1001.20	1.0012	1	0.12	0.5	500	300	1000	3	0.002	50
Case 14	10012.01	10.0120	10	0.12	0.5	500	300	1000	3	0.02	50
Case 15	100120.06	100.1201	100	0.12	0.5	500	300	1000	3	0.2	50
Case 16	1001200.60	1001.2006	1000	0.12	0.5	500	300	1000	3	2	50
Case 17	10012006.00	10012.0060	10000	0.12	0.5	500	300	1000	3	20	50
Case 18	1001.20	100.1201	100	0.12	0.5	500	3	10	3	0.2	50
Case 19	10012.01	100.1201	100	0.12	0.5	500	30	100	3	0.2	50
Case 20	503939.27	100.7879	100	0.79	0.5	500	300	5000	16	0.2	50
Case 21	1016340.25	101.6340	100	1.63	0.5	500	300	10000	33	0.2	50
Case 22	11861404.57	118.6140	100	18.61	0.5	500	300	100000	333	0.2	50
Case 23	100.01	0.1000	0.1	0.01	0.5	5000	300	1000	3	0.00002	50
Case 24	1000.12	1.0001	1	0.01	0.5	5000	300	1000	3	0.0002	50
Case 25	10001.20	10.0012	10	0.01	0.5	5000	300	1000	3	0.002	50
Case 26	100012.00	100.0120	100	0.01	0.5	5000	300	1000	3	0.02	50
Case 27	1000120.01	1000.1200	1000	0.01	0.5	5000	300	1000	3	0.2	50
Case 28	10001200.06	10001.2001	10000	0.01	0.5	5000	300	1000	3	2	50
Case 29	1000.12	100.0120	100	0.01	0.5	5000	3	10	3	0.02	50
Case 30	10001.20	100.0120	100	0.01	0.5	5000	30	100	3	0.02	50
Case 31	500392.19	100.0784	100	0.08	0.5	5000	300	5000	16	0.02	50
Case 32	1001618.69	100.1619	100	0.16	0.5	5000	300	10000	33	0.02	50
Case 33	10168017.49	101.6802	100	1.68	0.5	5000	300	100000	333	0.02	50

8 Progress Report

As of now, none of the use cases have been implemented. However, we do have the framework for much of the system and the database done, so although none of these have been implemented it is not a far leap to implement them. We now have to take the framework and build functionality onto it, which includes the use cases as well as interactions between the system and the database. This includes connecting to the database as well as implementing the correct retrieval functions from the database. We hope to have all this done by March 18th so that we have ample time for debugging and testing the code.

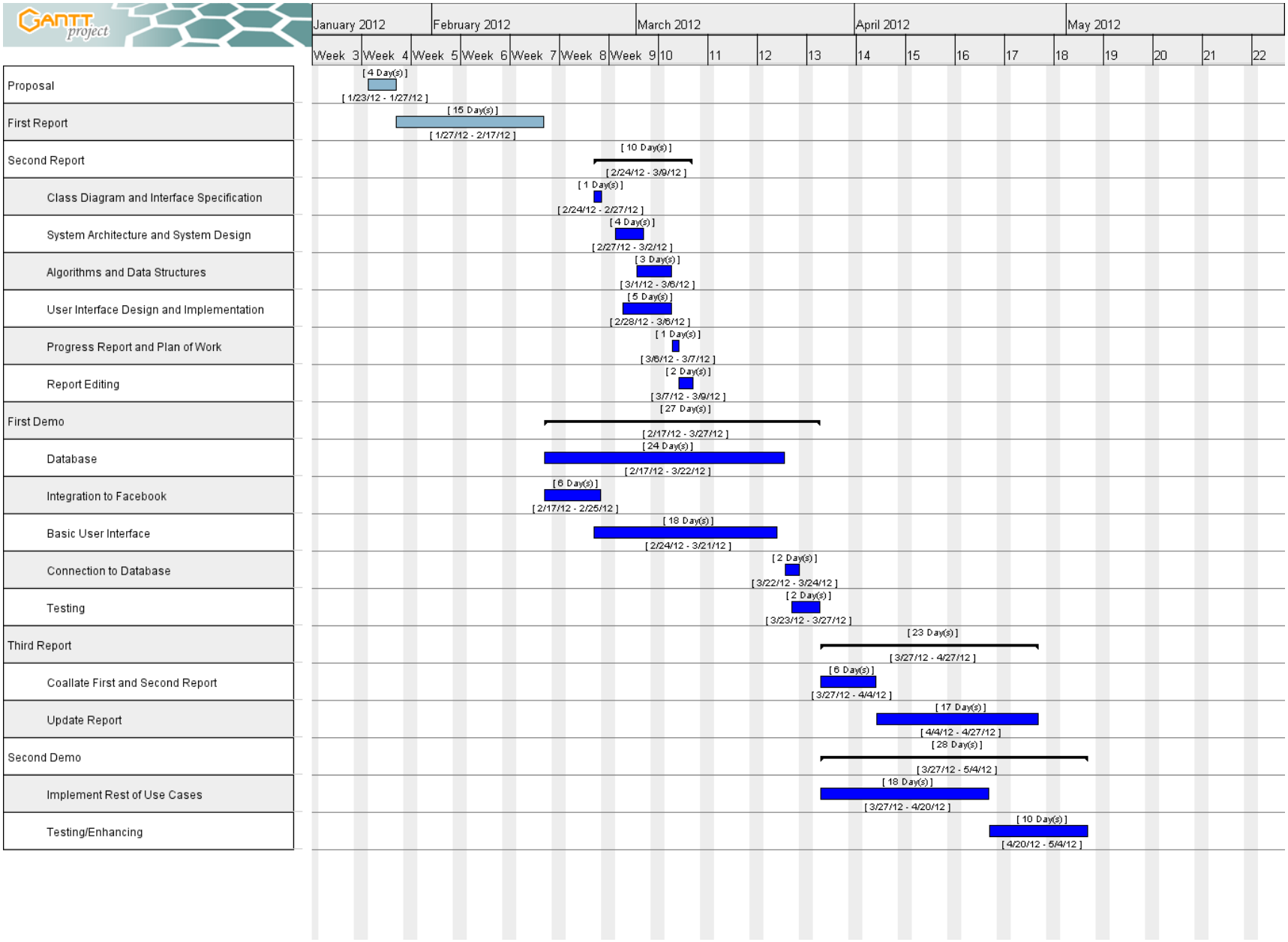
9 Plan of Work

Contribution Breakdown

Responsibilities Matrix	William	Aaron	Pratik	Dean	Omar	Noah
First Demo						
Database	x	x				
Facebook Integration			x	x	x	x
Basic User Interface	x	x	x	x	x	x
Connection to Database	x	x	x			
Testing	x	x	x	x	x	x
Third Report						
Collation of Reports	x	x	x			
Update Report	x	x	x	x	x	x
Second Demo						
Implement Rest of Use Cases	x	x	x	x	x	x
Testing and Enhancing	x	x	x	x	x	x

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

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


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

10 References

References

[Bell, 2004] Bell, D. (2004). Uml basics: The class diagram. www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/.

[Company, 2012] Company, O. T. L. U. (2012). Eclipse uml. http://omondo.com.

[Marsic, 2012] Marsic, I. (2012). *Software Engineering*. Unpublished, first edition edition.

[Unknown, a] Unknown. Architectural patterns and styles. http://msdn.microsoft.com/en-us/library/ee658117.aspx.

[Unknown, b] Unknown. Bottom-up integration in integration testing. http://www.freetutes.com/systemanalysis/sa9-bottom-up-integration.html.

[Unknown, c] Unknown. Communications protocol. http://en.wikipedia.org/wiki/Communications_protocol.

[Unknown, d] Unknown. Geometric series. mathworld.wolfram.com/GeometricSeries.html.

[Unknown, e] Unknown. Integration testing. http://softwaretestingfundamentals.com/integration-testing/.

[Unknown, 2011] Unknown (2011). Bull n bear. http://www.cjssecurities.co.za/main/Portals/0/bull_n_bear\%5B1\%5D.jpg.