Money Machine

Report #2

Group No. 6

Team Members

Name	**Email**
Rylan Uherek	rylan@scarletmail.rutgers.edu
Avinash Oza	avioza@scarletmail.rutgers.edu
Aakash Patel	Aak4shpatel@gmail.com
Mozam Todiwala	tmozam@scarletmail.rutgers.edu
Mandeep Desai	Mandeep.desai111@gmail.com
Pintu Patel	Php28@scarletmail.rutgers.edu

Instructor: Prof. Ivan Marsic

Project URL: https://sites.google.com/site/sespring13/

Revision History:

Version No.	**Date of Revision**
v.1 – Part #1	3/1/2013
v.2 – Part #2	3/9/2013
v.3 – Final Report #2	3/16/2013

Individual Contributions Breakdown

Task/Group Member	Rylan	Avinash	Aakash	Mozam	Mandeep	Pintu
Interaction Diagrams (30 Points)		50%		50%		
Class Diagram and Interface Specification (10 Points)					50%	50%
System Architecture and System Design (15 Points)					50%	50%
Algorithms and Data Structures (0 Points)[1]	---	---	---	---	---	---
User Interface Design and Implementation (15 Points)			100%			
Design of Tests (12 Points)		20%	20%	20%	20%	20%
Project Management and Plan of Work & References (18 Points)	90%				10%	

** *Underlined & Italicized Percentages* indicate that the team member will in the future produce the specified work indicated in the box. Boxes which are *not* italicized or underlined indicate that the team member has already completed the specified work.

Individual Point Allocation

Team Member	Points / Estimated Points
Rylan	17
Avinash	17
Aakash	16
Mozam	16
Mandeep	16
Pintu	16

[1] Our project does not any real algorithms or data structures. Therefore, we moved those points to the UI section.

Individual Work Description, Project Management, & Notes

The following is a brief description of what each team member completed for Report #2:

Rylan:

- Project Management
    - Coordinated meetings / meeting times
    - Collated reports, documents, etc.
    - Dropbox / Document Control management
    - Represented group / contact point with TA & Dr. Marsic
    - Edited, modified styling, etc. on submitted documents
- Project Management & Plan of Work Sections of Report

Avinash:

- Developed interaction diagram captions and descriptions
- Wrote test cases as needed

Aakash:

- Created documentation and updates for UI Spec & Implementation
- Updated team website with new documents & updates as needed
- Wrote test cases as needed and developed *Test Coverage* and *Integration Testing* sections of Test Section

Mozam:

- Developed interaction diagrams and coordinated with Avinash for descriptions
- Wrote test cases as needed

Mandeep:

- Worked on System Architecture and System Design, and Class Diagram and Interface Specification
- Updated references list
- Wrote test cases as needed and assisted in *Test Coverage* and *Integration Testing* sections by doing research on sections
- Developed *Testing State Diagrams*

Pintu:

- Worked on Class Diagram and Interface Specification, and System Architecture and System Design
- Wrote test cases as needed
- Developed *Testing State Diagrams*

NOTES:

- As project is coded, modifications will be made to the diagrams and documentation provided in this report. The fully updated documentation will be reflected in Report #3. The documentation contained in this report is current as of early March, 2013.

- Our project does not any real algorithms or data structures. Therefore, we moved those points to the UI section.

- All team members played a role in document review before the report was submitted.

Table of Contents

1.0 Interaction Diagrams

UC-1: Register

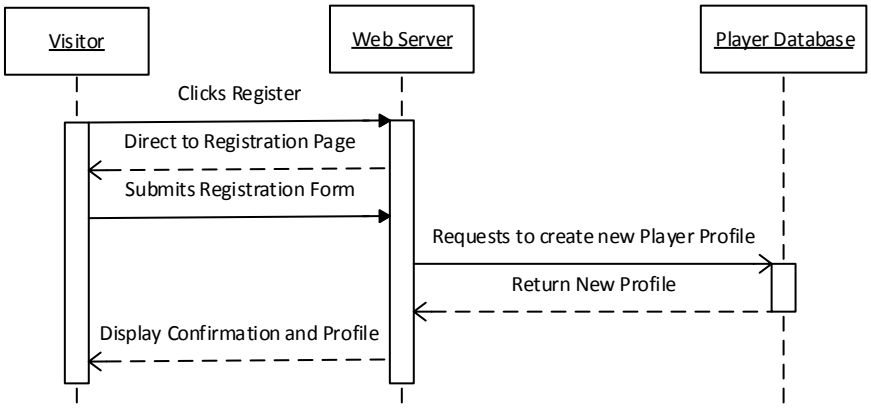
In this system sequence diagram, the visitor first navigates to the website. After reaching the website, the visitor clicks “Register”. After this, the visitor is presented with the registration page.

Once the user has submitted the registration page, the information provided is validated and is sent to the Player Database. The system then requests for a new player profile to be created for the visitor. The system then returns to the visitor that their profile creation was complete, and that they are now logged into the system.

The only alternate scenario to the main success scenario would be if any of the information entered by the user was invalid. In this situation, the system would return an error to the Visitor letting them know that there was an error in their submission. It would give the user another chance to submit the registration form.

As of now, no other implementations have been discussed, as the current one seems to be the most logical flow of events.

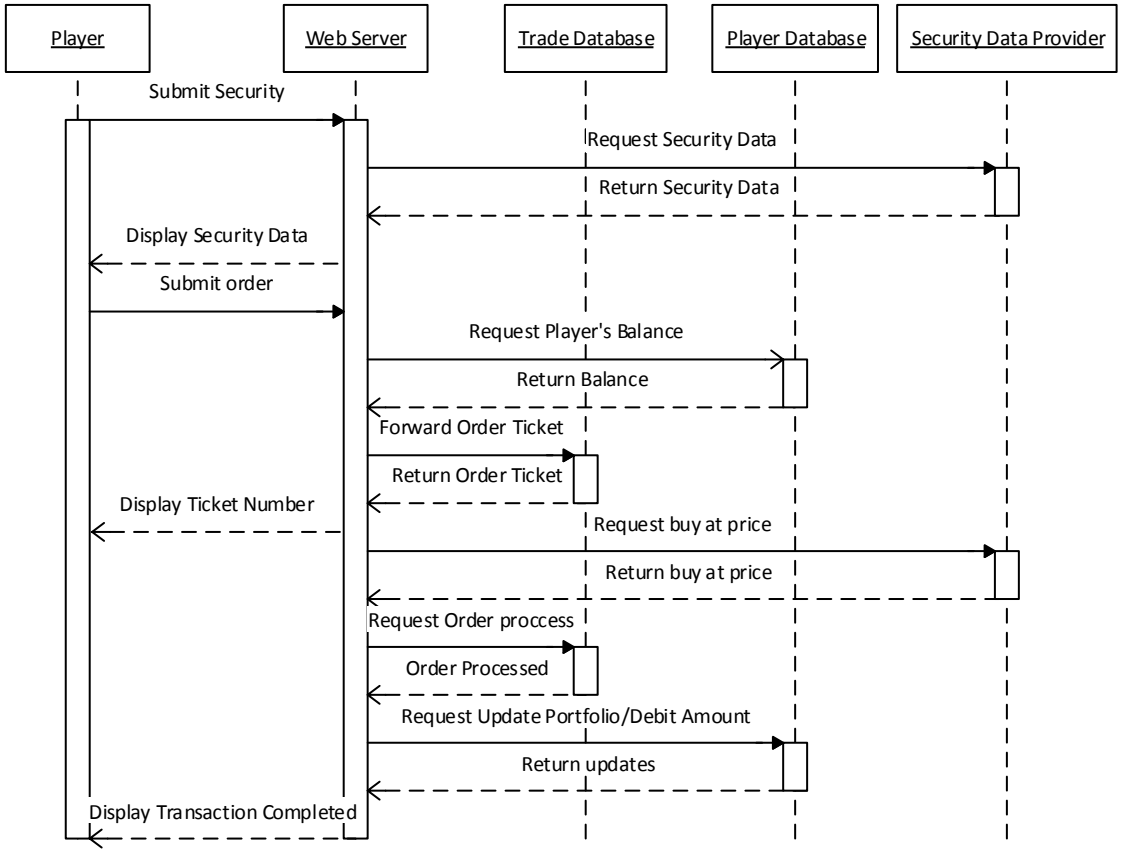
UC-1 System Sequence Diagram:



UC-3: Buy Security

In this system, the player selects a security of interest and in return the system should display the latest data of the security. This is done when the web server connects to the security data provider to read the data. The data is then displayed to the player. Then, the player must fill out order form which most importantly includes the buying price of the security. Upon clicking submit, the web server verifies if the player has enough balance to purchase. If the player has enough balance then the system requests an order ticket from the trade database for the particular security (securities). The player is displayed with a confirmation of order and order ticket number. The Web Server constantly reads the data of the particular security through the security data provider. Once, the parameters of the player match the current data, the system requests the trade database to process the order. Order is then processed and player's portfolio is updated. A notification is also sent to the player informing that the transaction is completed.

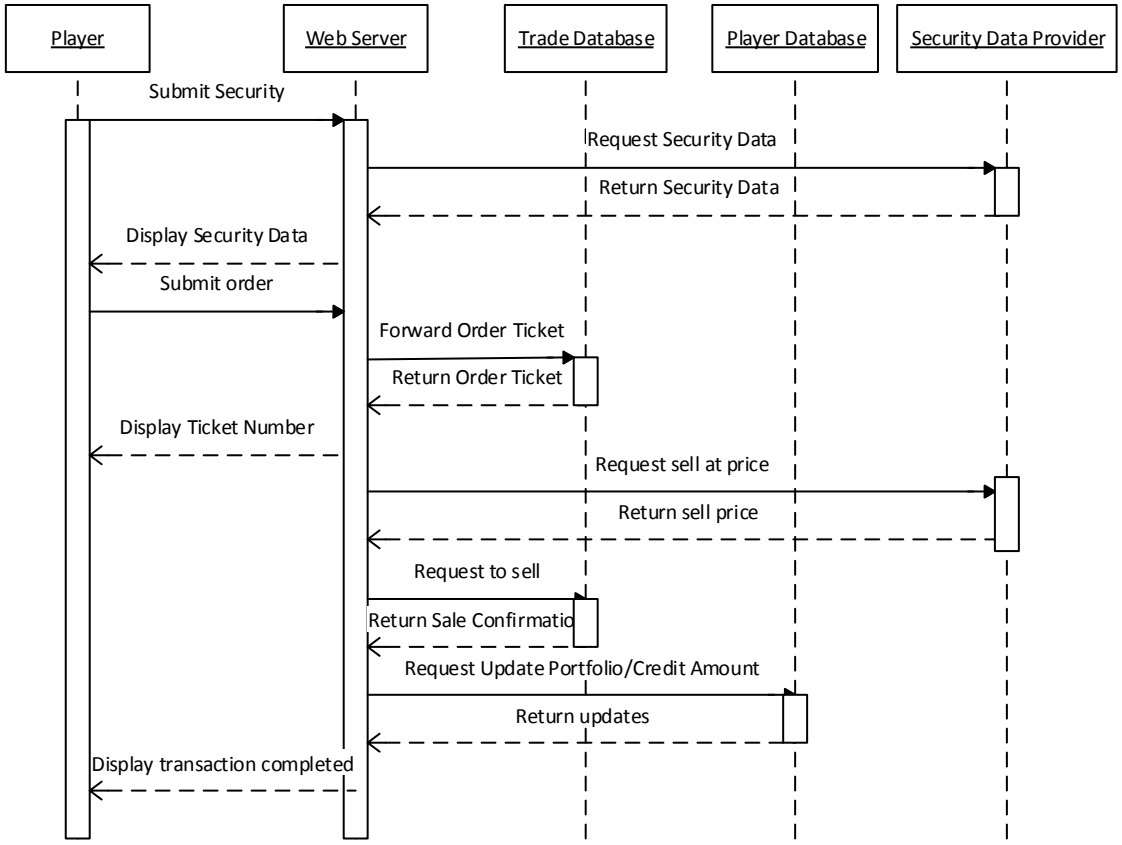
UC-3 System Sequence Diagram:



UC-4: Sell Security

In this system, the player selects a security of interest and in return the system should display the latest data of the security. This is done when the web server connects to the security data provider to read the data. The data is displayed to the player. Then, the player must fill out order form which most importantly includes the selling price of the security. A request for generating an order ticket is then sent out to the trade database. Once the order ticket has been generated the player is displayed confirmation of the order and the ticket number. The system constantly reads the data through the security data provider and once the price to sell his matched with the user's parameters the order is sent to be processed to the trade database.
Order is then processed and player's portfolio is updated. A notification is also sent to the player informing that the transaction is completed.

UC-4 System Sequence Diagram:

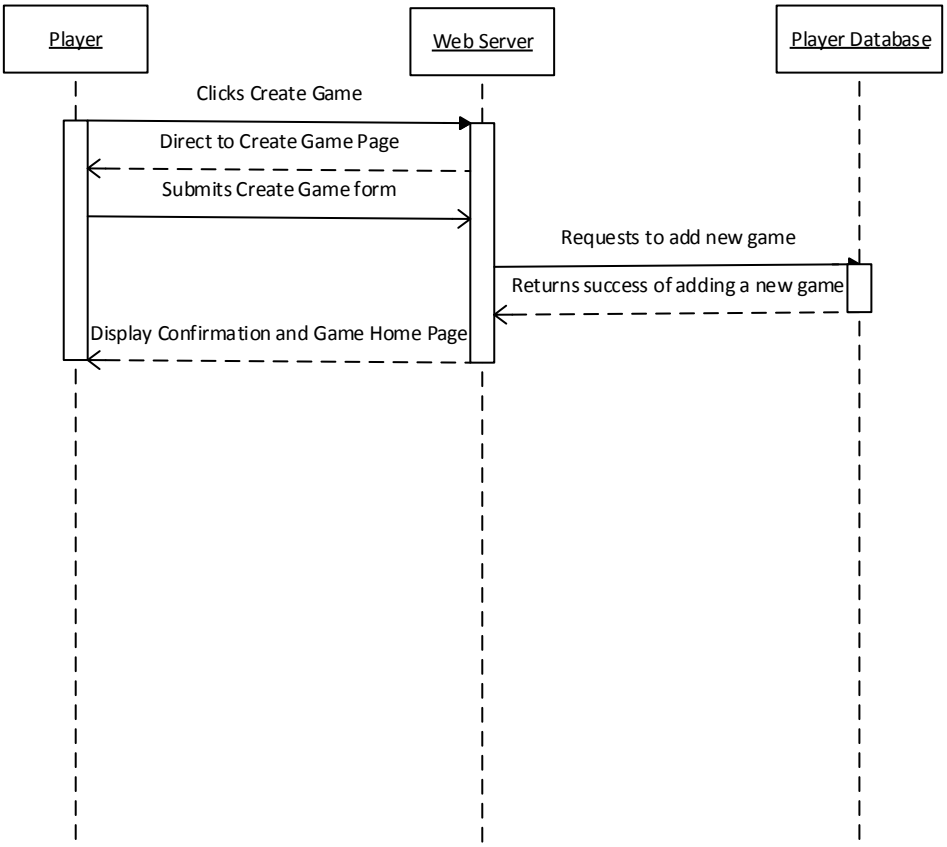


UC-7: Create Game

In this system sequence diagram, the Player requests for a new game to be created. The system then presents the user with the Create Game page. One the user has submitted the registration page, the information is validated and a request to create a new game is sent to the Player Database. The system then updated required fields in the Player Database and signals a success to the Web Server. The Web Server signals this back to the user with a confirmation that their game has successfully been create.

The only alternate scenario to the main success scenario would be if the game name the Player is trying to make is already taken. In this situation, the system would return an error after form submission letting the user know that their game name is already taken. It would ask the Player to choose another game name and go through the same process of revalidating.
As of now, no other implementations have been discussed, as the current one seems to be the most logical flow of events.

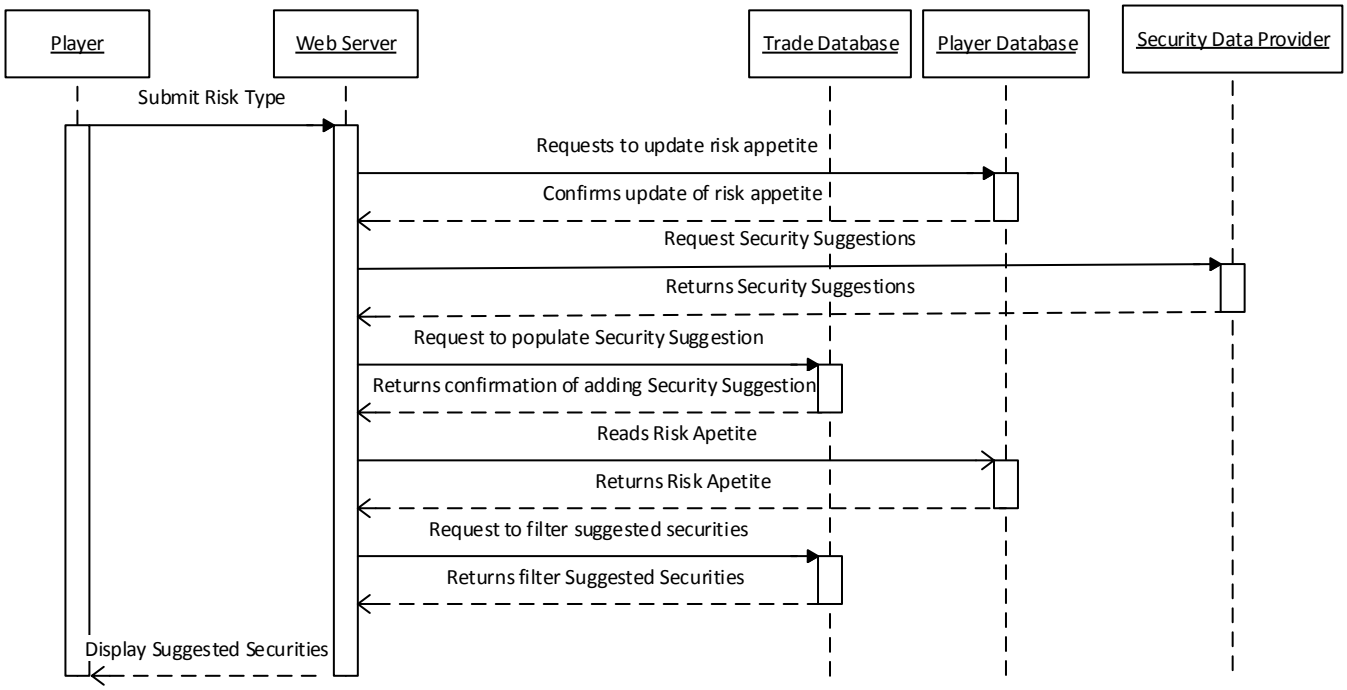
UC-7 System Sequence Diagram:



UC-14: Suggest Security

Each player has to select a risk appetite and submit to the user. This risk appetite is added to the player's database. Then, the system requests for security suggestions depending on the latest news, technical and fundamental analysis etc. through the Security Data Provider. A collection of the data is fed to the trade database. The system then reaches out to the player database to read the risk appetite and requests the trade database to filter out suggestions according to priority and the risk type requested. Finally, the suggestions are displayed to the user.

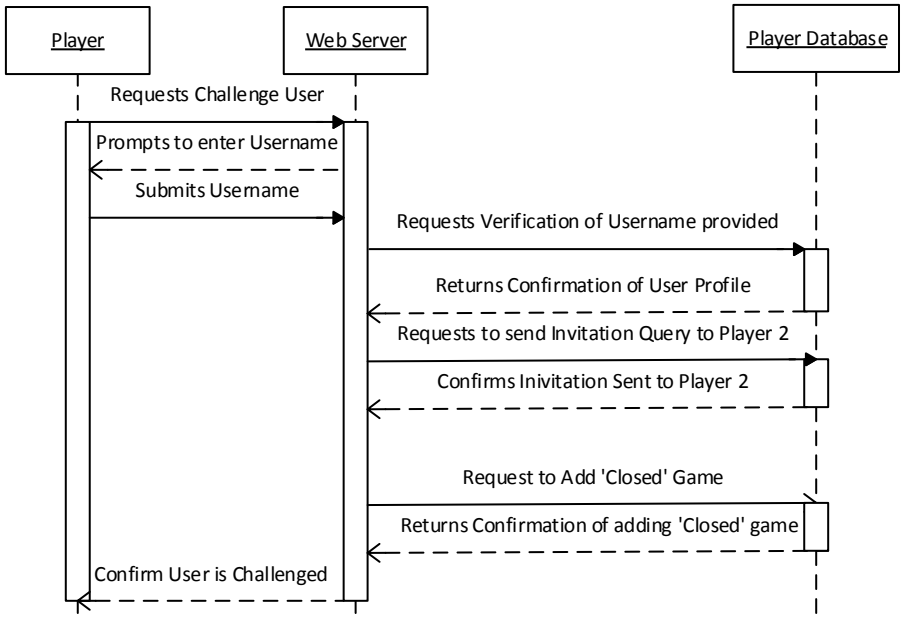
UC-14 System Sequence Diagram:



UC-15: Challenge User

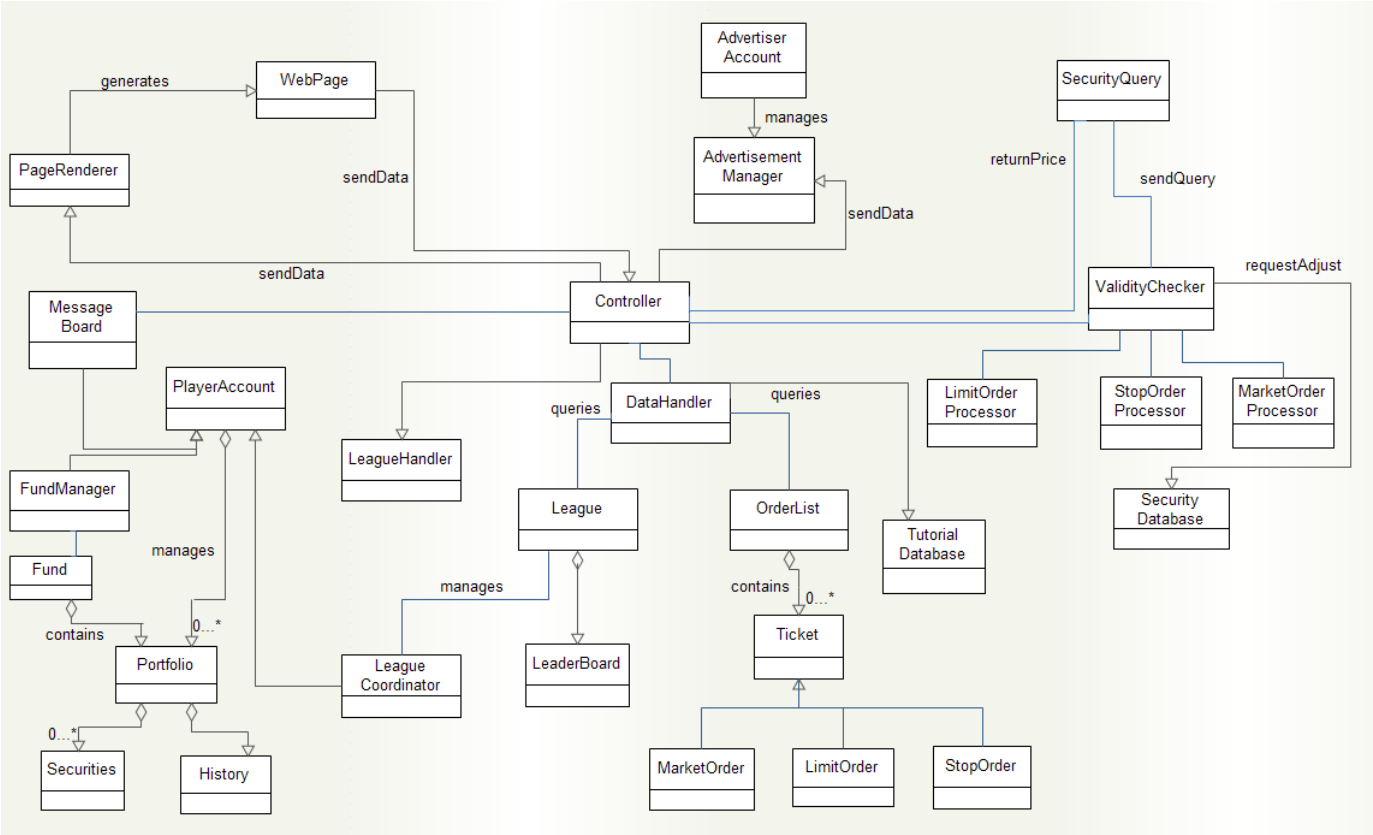
A player should be able to challenge other player(s). When the player requests to challenge another player, the system prompts the username of the player. The system verifies if the player exists in the player database. Once the verification is done the system sends a notification of challenge to Player 2. The portfolio of the two players get updated to yet another inner game (Closed Game) between the two. A notification is sent out to the initiating player saying that the challenge has started.

UC-15 System Sequence Diagram:



2.0 Class Diagram & Interface Specification

2.1 Class Diagram



A summary of important classes, their variables, and functions are shown in the *Data Types & Operation Signatures*.

2.2 Data Types & Operation Signatures

Controller

– ticket : Ticket
– data : StockData
– fields : Fields
– player : string
– portfolio : Portfolio
– history : TransactionHistory
– stock : String
– league : String

– Render (Integer : type, void* : data) : Boolean
– RenderError (Integer : type, void* : data) : Boolean
+ RequestBuy (Ticket : ticket)
+ RequestSell (Ticket : ticket)
+ RequestPortfolio (String : player) : void
+ RequestCreateLeague (Fields : fields) : void
+ RequestEditLeague (Fields : fields) : void
+ RequestCreateFund (Fields : fields) : void
+ RequestEditFund (Fields : fields) : void
+ RequestHistory (String : player) : void
+ RequestJoin (String : player, String : league) : void
+ RequestChallenge (String : player) : void
+ RequestInvite (String : player, String : league) : void
+ RequestShare (String : player) : void
+ RequestAddCoord (String : player) : void

AdvertiserAccount

advertiser : String
balance : Double
fields : Fields

– setAd( ) : void
– payBalance(balance : Double) : void
– editSettings(fields : String) : void

PageRenderer

– page :Page

– generatePageOrder (Ticket : ticket, Integer : valid) : boolean
– generatePageSecurity (SecurityData : data, Integer : valid) : boolean
– generatePagePortfolio (Portfolio : portfolio, SecurityData* : data, Integer : valid) : boolean
– generatePageFront (PlayerInfo : playerinfo, Integer : valid) : boolean
– generatePageLeague_Fund (Fields : fields, Integer : valid) : boolean
– generatePageTutorial (Fields : fields, Integer : valid) : boolean
– generatePageAdvertisement (Fields : fields, Integer : valid) : boolean
– generatePageJoin (String : league, Integer : valid) : boolean
– generatePageInvite (Player : player, String : league, Integer : valid, String : which) :boolean
– generatePageShare (Player : player, Integer : valid, String : which) : boolean
+ pageType (Integer : Type, void* : data, Integer : valid) : boolean
+ getPage() : Page

SecurityQuery

+ Query (String : security) : SecurityData

Validity Checker

– data : SecurityData
– ticket : Ticket
– pricepershare : Double
– balance : Double

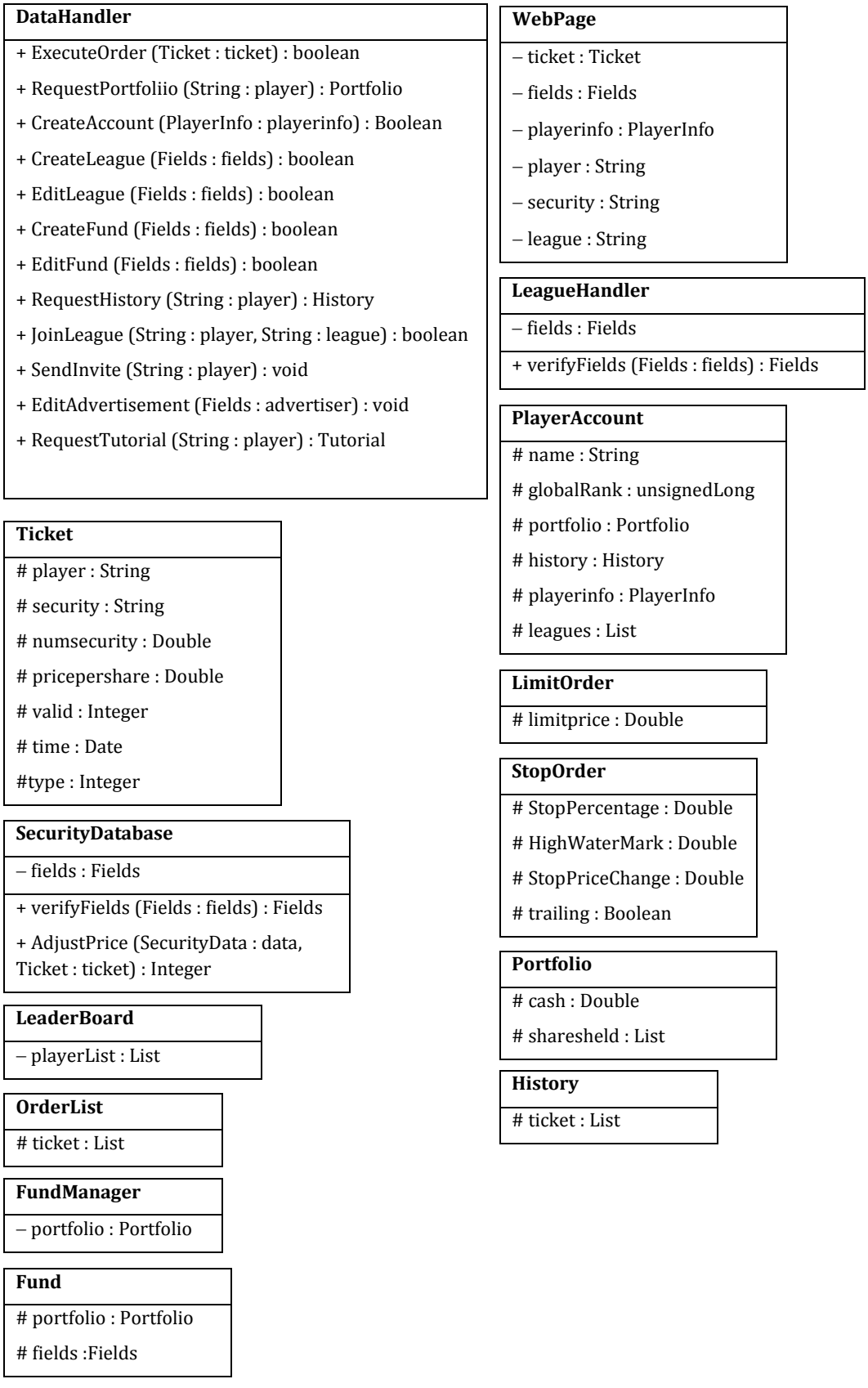
+ ValidateBuy (Ticket : ticket) : void
– VerifyFunds() : void
+ ValidateSell (Ticket : ticket) : void

Securties

pricepaid : Double
executiondate : Date
lasttrade : Double
change : Double
changepercent : Double
daysgain : Double
quantity : Double
totalgain : Double
totalgainpercent : Double

League

player : List
ranking : List
fields : Fields


DataHandler
+ ExecuteOrder (Ticket : ticket) : boolean
+ RequestPortfoliio (String : player) : Portfolio
+ CreateAccount (PlayerInfo : playerinfo) : Boolean
+ CreateLeague (Fields : fields) : boolean
+ EditLeague (Fields : fields) : boolean
+ CreateFund (Fields : fields) : boolean
+ EditFund (Fields : fields) : boolean
+ RequestHistory (String : player) : History
+ JoinLeague (String : player, String : league) : boolean
+ SendInvite (String : player) : void
+ EditAdvertisement (Fields : advertiser) : void
+ RequestTutorial (String : player) : Tutorial
Ticket
player : String
security : String
numsecurity : Double
pricepershare : Double
valid : Integer
time : Date
#type : Integer
SecurityDatabase
– fields : Fields
+ verifyFields (Fields : fields) : Fields
+ AdjustPrice (SecurityData : data, Ticket : ticket) : Integer
LeaderBoard
– playerList : List
OrderList
ticket : List
FundManager
– portfolio : Portfolio
Fund
portfolio : Portfolio
fields :Fields
WebPage
– ticket : Ticket
– fields : Fields
– playerinfo : PlayerInfo
– player : String
– security : String
– league : String
LeagueHandler
– fields : Fields
+ verifyFields (Fields : fields) : Fields
PlayerAccount
name : String
globalRank : unsignedLong
portfolio : Portfolio
history : History
playerinfo : PlayerInfo
leagues : List
LimitOrder
limitprice : Double
StopOrder
StopPercentage : Double
HighWaterMark : Double
StopPriceChange : Double
trailing : Boolean
Portfolio
cash : Double
sharesheld : List
History
ticket : List

1. Controller

Attributes

The controller plays the role of a town center, conveying messages back and forth between different domain concepts in the domain model. The role of controller is best accomplished, if the controller has a copy of all data that it handles as an attribute. Doing so will lower the chance of corrupting data.

– ticket : Ticket

This is a copy of the order ticket that the player has just submitted.

– data : StockData

This is a copy of the data that the system queries from the Security Info Provider.

– fields : Fields

This is a copy of the fields that a Player or League Coordinator fills out during a creation / editing request.

– player : string

This a copy of the player's username that the controller passes along to the data handler. It is used to find the Player object from inside the database.

– portfolio : Portfolio

This a copy of a Portfolio object that the controller passes along.

– history : TransactionHistory

This is a copy of a TransactionHistory object that the controller passes along.

– stock : String

This is a copy of the security symbol that is passed to the Security Query for it ot get info on the security.

– league : String

This is a copy of the name of the 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

The controller calls this method when it is ready to render a page. The arguments Integer represents the type of page that is displayed, and the pointer, points to a data structure containing the data necessary to construct the page.

– RenderError (Integer : type, void* : data) : Boolean

The purpose of this method is same as Render method, but instead of rendering the correct page it renders an error version of the page.

+ RequestBuy (Ticket : ticket)

    Method used to request a buy security.

+ RequestSell (Ticket : ticket)

    Method used to request a sell security.

+ RequestPortfolio (String : player) : void

    Method used to view a portfolio.

+ RequestCreateLeague (Fields : fields) : void

    Method used to create a league.

+ RequestEditLeague (Fields : fields) : void

    Method used to edit league settings.

+ RequestCreateFund (Fields : fields) : void

    Method used to create a fund.

+ RequestEditFund (Fields : fields) : void

    Method used to edit fund settings.

+ RequestHistory (String : player) : void

    Method used to obtain/view transaction history.

+ RequestJoin (String : player, String : league) : void

    Method used to request, join a league/game.

+ RequestChallenge (String : player) : void

    Method used to challenge another player.

+ RequestInvite (String : player, String : league) : void

    Method used to invite a player into the league/game. This method will also be used to invite non-registered players to the investment game.

+ RequestShare (String : player) : void

    Method used to share player current standings and game statistics to registered and non-registered players.

+ RequestAddCoord (String : player) : void

    Method used to add a coordinator to a league.

2. PageRenderer

Attributes

– page :Page

    This is the current page that the web browser is displaying/will be displayed.

Methods

The valid parameter lets the page rendered know if the page that it should be generating is a success page or an error page.

– generatePageOrder (Ticket : ticket, Integer : valid) : boolean

Method called to render a page displaying the results of an order.

– generatePageSecurity (SecurityData : data, Integer : valid) : boolean

Method called to render a page displaying result of a security data query.

– generatePagePortfolio (Portfolio : portfolio, SecurityData* : data, Integer : valid) : boolean

Method called to render a page displaying the results of a portfolio viewing.

– generatePageFront (PlayerInfo : playerinfo, Integer : valid) : boolean

Method called to render a page displaying the result of an account creation.

– generatePageLeague_Fund (Fields : fields, Integer : valid) : boolean

Method called to render a page displaying the results of a creation of a fund or league or editing of a fund or league.

– generatePageJoin (String : league, Integer : valid) : boolean

Method called to render a page displaying the results of joining of a league.

– generatePageInvite (Player : player, String : league, Integer : valid, String : which) : boolean

Method called to render a page displaying the results of inviting a player or non-player to a league.

– generatePageShare (Player : player, Integer : valid, String : which) : boolean

Method called to render a page displaying the results of sharing of game statistics to other players and non-players.

+ pageType (Integer : Type, void* : data, Integer : valid) : boolean

Method 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

Method called by the web page in order to retrieve the page it must display.

3. DataHandler

Methods

Methods called by the controller to access the information in the database.

+ ExecuteOrder (Ticket : ticket) : boolean

Method executes the ticket order by updating the player's portfolio accordingly.

+ RequestPortfoliio (String : player) : Portfolio

Method called to request the portfolio data from the database.

+ CreateAccount (PlayerInfo : playerinfo) : Boolean

    Method called to request creation of new account and data to be stored in the database.

+ CreateLeague (Fields : fields) : boolean

    Method called to request a new league creation and data to be stored in the database.

+ EditLeague (Fields : fields) : boolean

    Method called to request the league settings be modified in the database.

+ CreateFund (Fields : fields) : boolean

    Method called to request a new fund creation and data to be stored in the database.

+ EditFund (Fields : fields) : boolean

    Method called to request the fund settings be modified in the database.

+ RequestHistory (String : player) : History

    Method called to request the transaction history from the database.

+ JoinLeague (String : player, String : league) : boolean

    Method called to request that a player be added to a league in the database.

+ SendInvite (String : player) : void

    Method called to request that a invite to be send a player or non-player.

4. SecurityQuery

Methods

+ Query (String : security) : SecurityData

    Method called to request security data from the Security Info Provider. The data is forwarded straight to the class requesting it, and a copy is made within the Security Query.

5. Validity Checker

Attributes

The validity checker holds the below attributes that it uses in calculations to determine if an order is valid.

– data : SecurityData

    Copy of the stock data obtained from Security Query.

– ticket : Ticket

    Copy of the order ticket that the player fills out.

– pricepershare : Double

    Copy of the price per share of the security, which the TradeDatabase determines.

– balance : Double

    Copy of the investor's current account balance.

Methods

+ ValidateBuy (Ticket : ticket) : void

Method called by the controller to determine if a buy is valid or not.

- VerifyFunds() : void

Method called by the Validity Checker in order to determine if the investor has sufficient funds for the transaction.

+ ValidateSell (Ticket : ticket) : void

Method called by the controller to determine if a sell is valid or not.

6. SecurityDatabase

Methods

+ AdjustPrice (SecurityData : data, Ticket : ticket) : Integer

Method called by the Validity Checker to modify the security price per share in accordance to how many the player plans to buy or sell.

7. WebPage

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

Attributes

– ticket : Ticket

This is a copy of an order ticket that the player fills out.

– fields : Fields

This is a copy of the league or fund settings the player fills out.

– playerinfo : PlayerInfo

This is a copy of the player info that the database provides.

– player : String

This is a copy of the player's username.

– security : String

This is a copy of the particular security that is requested by the player.

– league : String

This is a copy of the league name that the player enters.

8. TutorialDatabase

This class contains list of tutorials available to the player.

9. LeaderBoard

Attributes

– playerList : List

This is a list of the top ranked players in the game.

10. LeagueHandler

Attributes

– fields : Fields

This is a copy of the fields for the league

Methods

+ verifyFields (Fields : fields) : Fields

Method the controller calls that verifies that the settings for the league are all valid.

11. Ticket

Attributes

player : String

This is the player's username.

security : String

This is the stock symbol.

numsecurity : Double

This is the amount of security that is being exchanged.

pricepershare : Double

This the price per share of the security.

valid : Integer

This 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).

12. Securities

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

Attributes

pricepaid : Double

This 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 latest trade on the market for the stock.

change : Double

This is the change in the stock form 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 security in the current day.

quantity : Double

This is the amount of security that is owned.

totalgain : Double

This is the total gain from the security when it was first bought.

totalgainpercent : Double

This is the percentage gain from the security out of the gains from all security the player holds.

13. Portfolio

Attributes

cash : Double

This is the player's balance.

sharesheld : List

This is a list of class shares that the player owns.

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.

15. LimitOrder

Attributes

limitprice : Double

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

16. MarketOrder

This class is the default order type and has no special requirements.

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.

18. History

Attributes

ticket : List

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

19. FundManager

Attributes

– portfolio : Portfolio

This is the portfolio of the fund, which the fund manager maintains.

20. LeagueCoordinator

The league coordinator does not have any special attributes or methods that make it different from a player.

21. PlayerAccount

Attributes

name : String

This is the username of the player.

globalRank : unsignedLong

This is the global rank of the player.

portfolio : Portfolio

This is the player's portfolio.

history : History

This is the investor's transaction history.

playerinfo : PlayerInfo

This the player's personal info.

leagues : List

This is the list of leagues that the player is currently a member of.

22. Fund

Attributes

portfolio : Portfolio

This is the fund's portfolio

fields :Fields

This the various settings of the fund, including fund name.

23. League

Attributes

investor : List

This is the list of players that currently in the league

ranking : List

This is the list of rankings for each player (it runs parallel to the player list).

fields : Fields

This is the various settings of the league, including the league name.

24. AdvertiserAccount

Attributes

advertiser : String

This is the username of the advertiser.

balance : Double

This is the balance that the advertiser owes (unpaid balance).

fields : Fields

This are the fields that the advertiser fills out when editing settings.

Methods

– setAd( ) : void

– payBalance(balance : Double) : void

– editSettings(fields : String) : void

25. AdvertisementManager

This class contains all the advertisements that need to be displayed for a given advertiser.

26. NewsFeeder

This class contains all the recent news feeds relating to financial sector, economy, and world.

27. MessageBoard

This class contains all the messages, or posts that other players have posted on the message board.

2.3 Traceability Matrix

Class / Domain Concept	WebPage	PageRender	ValidityChecker	StockQuery	DataHandler	GameHandler	TradeDatabase	AdvertisementHanler
WebPage	X							
PageRender		X						
Controller	X	X						
ValidityChecker			X					
StockQuery				X				
SecurityDatabase							X	
DataHandler					X			
LeagueHandler						X		
League					X			
LeaderBoard					X			
PlayerAccount					X			
Portfolio					X			
FundManager					X			
Fund					X			
Securities					X			
History					X			
OrderList					X			
Ticket					X			
Market Order					X			
Limit Order					X			
Stop Order					X			
MarketOrderProcessor		X						
LimitOrderProcessor		X						
StopOrderProcessor		X						
TutorialDatabase					X			
AdvertiserAccount					X			
AdvertisementManger								X
MessageBoard					X			

Many of the classes map back to the DataHandler concept they contain data that is queried by the DataHandler. The domain model represented the all the classes in single entity, but now they are show as separate entities in the class diagram. The class diagram gives more insight on the inner workings and details of our program.

2.3.1 Object Constrain Language (OCL)

context Controller::RequestPortfolio(string : investor) void

pre: (investor → InvestorAccount.portfolio = this.portfolio)

- You can only view your own portfolios

context Controller::RequestEditLeague(Fields : fields) void

pre: (InvestorAccount → LeagueCoordinator = true)

- You can only edit a league if you are the league coordinator

context Controller::RequestEditFund(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(PlayerInfo : playerinfo) : Boolean

post: (hasPortfolio = true AND inGlobalLeague = 1)

- The investor will have a portfolio for the Global Public League upon registration

context DataHandler::CreateLeague(Fields : _elds) : 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 ≥ 0

- The User will not have a negative balance

context ValidityChecker::ValidateBuy(Ticket : ticket)

pre: (ticket → 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 _ 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 LeagueHandler::verifyFields inv:

self!_elds.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 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 number between 0 and 100

context LimitOrder inv:

self.limitprice > 0

- User cannot purchase a stock at a price of zero

3.0 System Architecture & System Design

3.1 Architectural Styles

Money machine utilizes architectural styles with a main focus on Model/View/Controller approach. In this part, we will take closer look in to how Money machine incorporates these techniques in its implementation.

3.1.1 Model/View/Controller

Money Machine relies heavily on the Model/View/Controller architecture. The main view is the 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 Python. For the models, the site database will be created using Django (Sqlite abstraction) 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.

3.1.2 Front and Back Ends

The front end component mainly involves Web UI, which will be mainly seen by public. The back end consists of all the behind the scenes business logic for our application. For example, the controller to communicate with the database, controller must go through DataHandler. So, in that situation, we can conclude that DataHandler is working as front end of the database to controller.

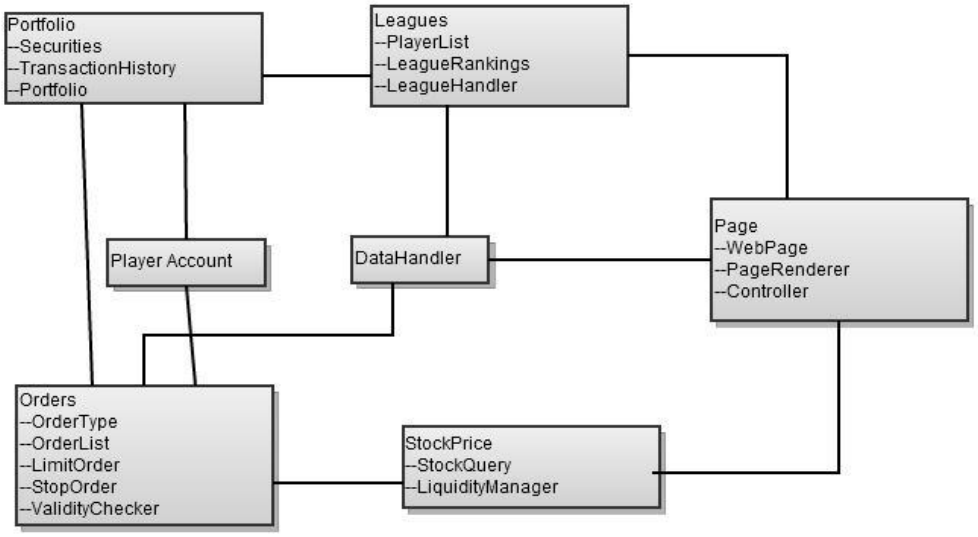
3.1.3 Event-driven Architecture

A Player acts as an event generator. A Player can buy securities, sell securities, create new leagues, and participate in different leagues. Another type of event is driven when the stock prices changes, or a company merges into another company or a company doesn't have enough securities to sell. In addition, there will be another event/feature called tutorial, in which our players choose their skill levels and based on their skill level, they will be able to learn more about the different types of securities and their behavior. Players can send referral to join our game by providing E-mail address of the other people and E-mail will be sent out to other people.

3.1.4 Object-oriented

In object-oriented design, the responsibilities are divided into different objects, which contains relevant information/data and behavior. In our application, we are planning to use object oriented approach, because it will make our work easier as well as efficient. We can represent Portfolio, Securities, League, and Orders as objects. These objects are most important things in our project.

3.2 Identifying Subsystems



Page: (WebPage, PageRenderer, Controller)

Page is responsible for taking Player's requests and executing or transferring those requests to other subsystems. Page is subsystem which has broad relationship with the Player.

League: (PlayerList, LeagueHandler,LeagueRankings)

League is subsystem that is responsible for creating as well as maintaining /updating all the things that are associated with the League such as, updating information about different kind of Leagues and their Players.

Portfolio: (Portfolio, TransactionHistory, Securities)

This subsystem is responsible for keeping track of Player's portfolio, securities, account balance and their past transactions.

Orders: (OrderType, OrderList, LimitOrder, StopOrder, ValidityChecker)

Orders keep track of the all the orders that have been placed in the past. In addition, It lets Player to stop order when the Player does not want to sell or buy Securities which he planned to buy for that price. In addition, this subsystem is responsible for validating the Player's buying securities request based upon available balance in Player's account. In addition, Players will have to choose the order type such as stocks, bonds, funds etc.

StockPrice: (StockQuery, LiquidityManager)

This subsystem is mainly responsible for getting the updated stock prices and alter them based on liquidity.

3.3 Mapping Subsystems to Hardware

The capabilities allow Django and Sqlite 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 server.

3.4 Persistent Data Storage

Our database will store user's name, user's account balance, current stock prices and history of user's past transactions. For current stocks, the database will save current stock name, stock's ticker symbol, price and available quantity, price, date and time of the transaction. The system will be able to calculate Player's net worth, his stock holdings, his account balance, his standing in league, and his past transactions. In addition, the system will also suggest Player some securities based on his stock holdings. The database will store the information about the past transactions and different types of stocks sold as well as bought.

Name: Ivan Marsic

CashBalance: $26,615.00

Market Value: $73385.00

Stocks

Symbol	Qty	Price Paid	Date Bought	Current Price
GOOG	100	652.55	11/14/12	806.19
YHOO	100	19.35	12/20/12	21.94
XOM	70	88.50	2/27/13	89.23

Transaction History

Symbol	Transaction Type	Price	Quantity	Date
YHOO	Buy	19.35	100	12/20/12
XOM	Buy	88.50	70	2/27/13
F	Sell	34.83	200	2 /24/ 12
GOOG	Buy	652.55	100	11/14/12

3.5 Network Protocol

Money Machine will communicate with our application via HTTP. If the user is at our website, then they will be able to get the latest updates about the United States markets as well as some International markets. In addition, if the user scrolls down on the home page, then they will be able to get the latest news related to stock markets. If the user are registered with our system then they can log in and then will be directed to their game, and if the user are not registered with our system then user will be asked to register with their background information. The system will validate the log in information and upon successful completion, login cookie stored on the user's browser which authenticates the user to experience game.

3.6 Global Control Flow

Our system is an event driven system in which it will wait for certain actions and then responds accordingly. A Player's portfolio will be updated every time webpage is updated. The database will be

updated every time it will receive request for StockInfoProvider and then Player's portfolio will be updated. This process is similar for League updates. If a Player wants know about the securities then StockInfoProvider will be requested and then up to date information will be sent based on Player's request. In addition, there will be additional feature called Invite, in which Players will be able to send the referral to join the game. Most of the events in our system are related to each other. Players' request are executed in the order they were received, which is like putting them in queue, and when the orders are executed then they will be removed from queue.

3.7 Hardware Requirements

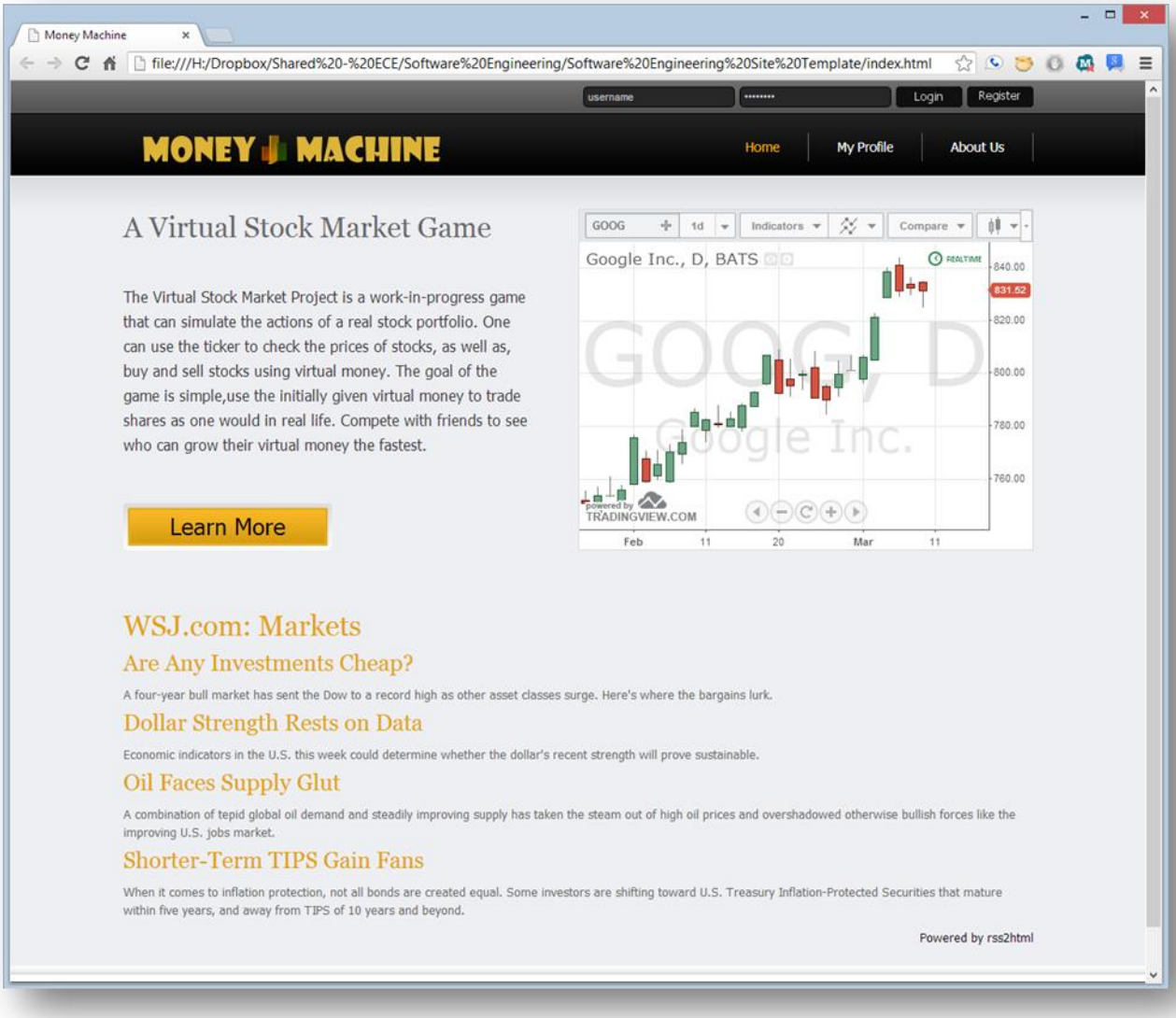
Our system will require only Internet connection and web browsers from our users. Our system will run on any web browser. Our system won't require any hardware space for this application. It will save all the information on our 'MoneyMachine' servers. Using their preferred browsers, without installing any software, Users easily connecting to their Internet, and enjoy and experience real life Stocks, and It will be an amazing experience for our users. In addition, our application will be accessible by any mobile device platforms such as iOS, Android, and Blackberry.

4.0 Algorithms & Data Structures

Our project does not have any true algorithms or data structures. We have chosen to forgo this section. Points for this section have been re-allocated into the User Interface Design & Implementation section as required. Please see the breakdown of responsibility to note the changes.

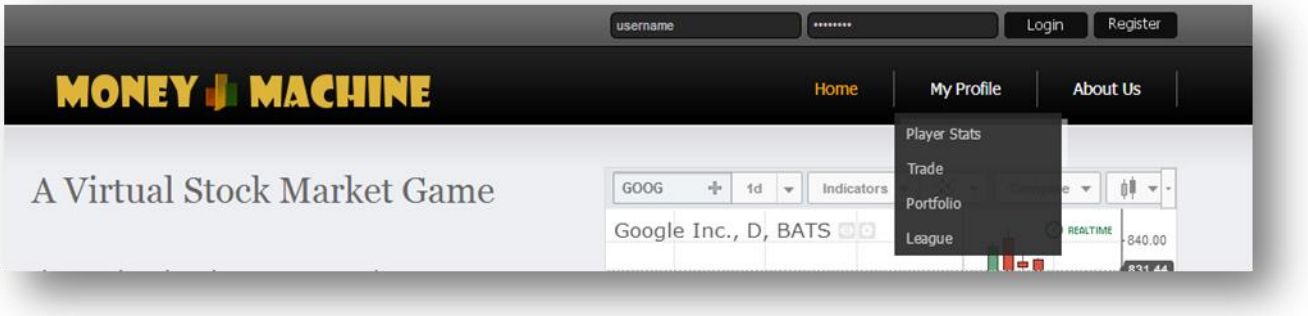
5.0 User Interface Design & Specification

5.1 Home Page



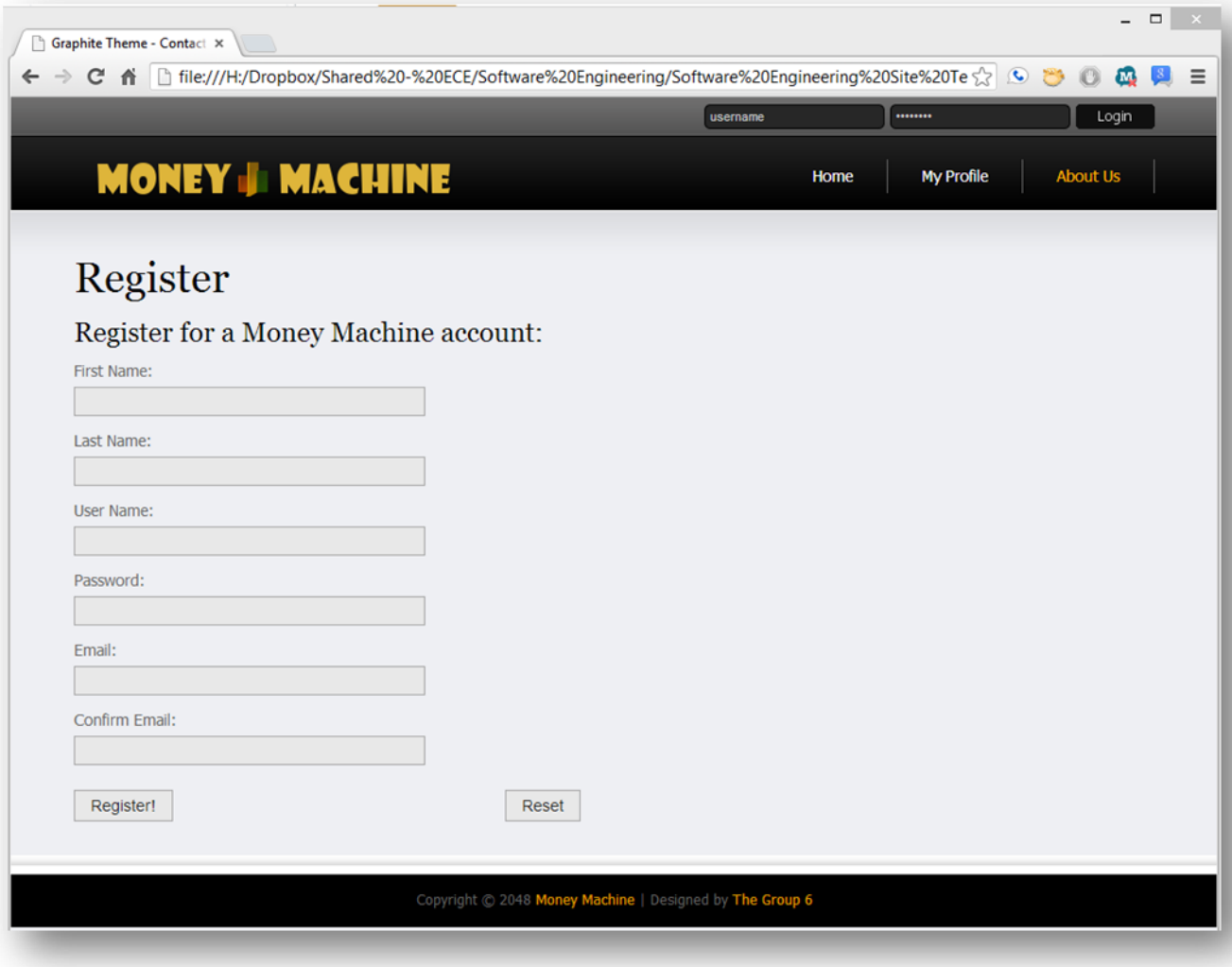
This is the home page of Money Machine, this is the first page that will be loaded when a user visits the site at first. The new UI offers a much cleaner look and provides quick access to stock ticker information directly from the home page which the previous UI design was not meant to do. The buttons have changed as well providing a more intuitive design. The original mockup of the home page was used as a skeleton for the content that was going to be displayed on the home page, major change in this UI is the layout of the content. Originally the News feeds were on the left half of the webpage while the stock prices and world markets information was on the right half of the page, but this has been changed with news feeds being on the bottom half of the page while displaying the welcome message and the stock information on the top half of the page.

5.2 Header Layout



The header has been changed which provides a log in system directly accessible within any page on the website given that the user is not currently logged in. Next change is the drop-down menu which has the options: Players Stats, Trade, Portfolio, and League. The drop-down menu appears whenever the cursor is hovered over the “My Profile” button. This is different than the originally planned mockup of having just a single “My Profile” page which just had the tabbed panels that showed the 4 information tabs. The amount of clicks necessary with the new UI is the same as the previous UI; the user still has to click on 1 of the 4 options within the menu to perform the desired task. To access the subpages within the “My Profile” page it will only take 1 click from anywhere on the “Home” page or the “About Us” page. The “Register” button has also been added next to the “Login” button which has reduced the number of clicks required for the user to access the registration page from 2 clicks to 1 click.

5.3 Registration Page

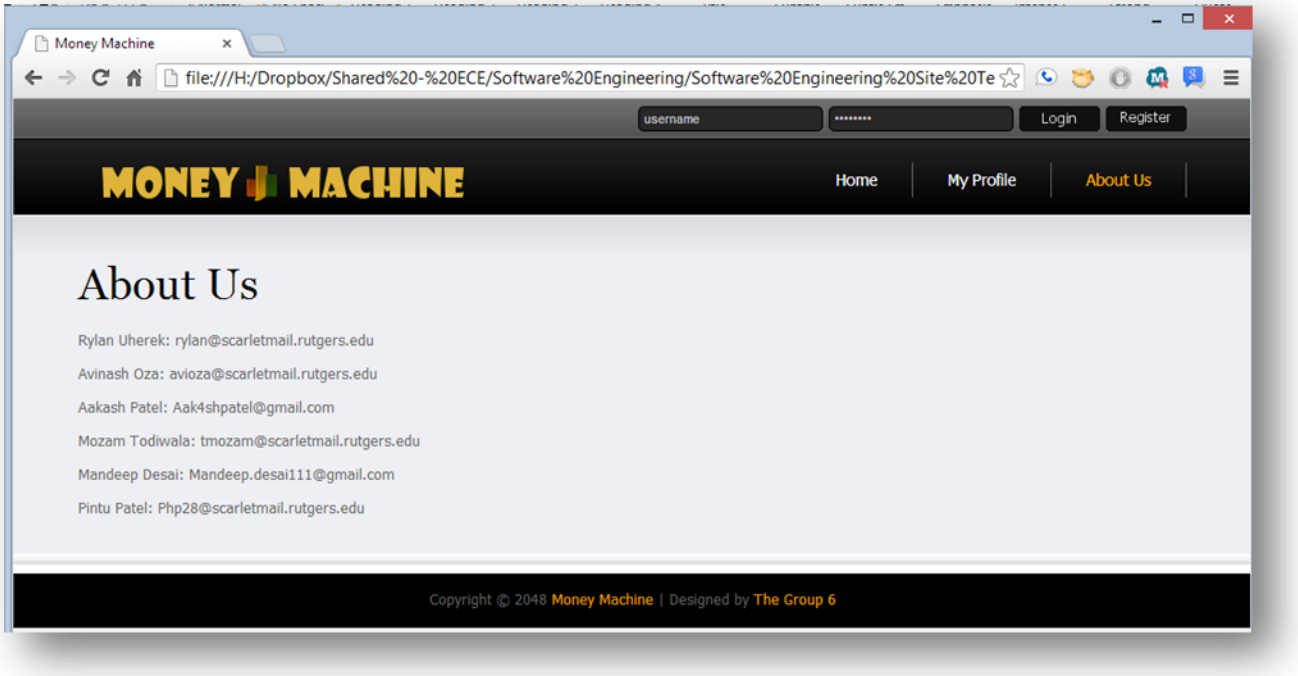


The registration page has been revised that provides more fields of input. The user has to enter an extra field, "User Name" and the "Confirm Password" from the previous UI Design has been changed to "Confirm Email" instead. This change has been made because the user can type the wrong password at first which can be recoverable via email, but if the user enters the wrong email address then that account can potentially belong to someone else or the user unable to access their account. So it is very important that the user enters the correct email address and have it be verified. A "Reset" button has been implemented should the user choose not to register for Money Machine. Once the user has registered he/she will be brought to the "My Profile" page which will implement Use Case 2, Use case 5 and Use Case 16.

HOW TO ACCESS THIS PAGE:

1) From the **Home** page, click on the **Register** button.

5.4 About Us Page



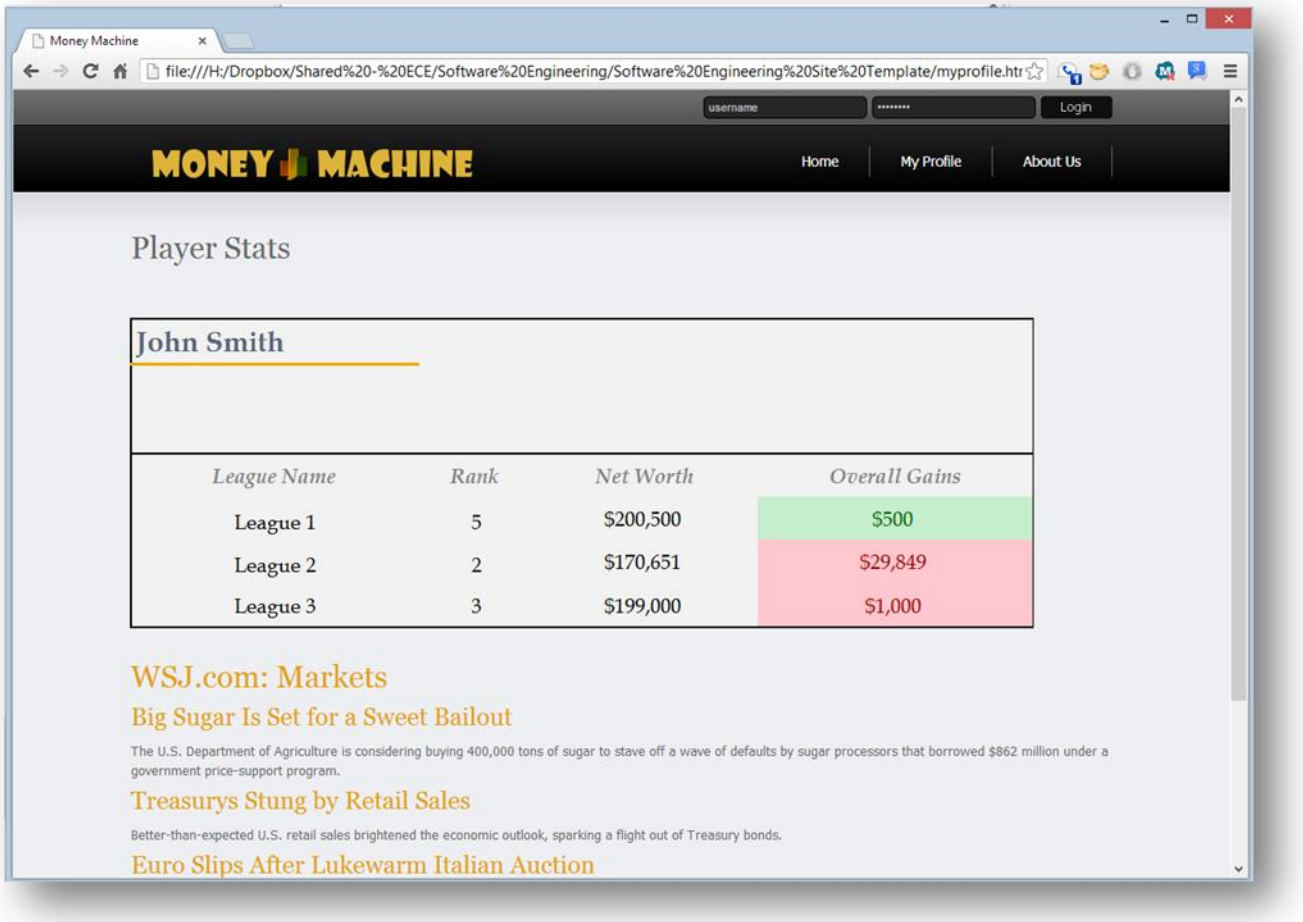
The "Help" page from the mockup has been changed to the "About Us" page instead which has the teams email addresses so it is easier for the player to contact one of the site administrators.

All the other changes that have been made are pure aesthetic changes which still provide the same number of clicks and menu traversals as before. UI minimizes the user effort in the sense that it is a simple interface while providing the tools necessary for the player to go through the registration process and enter a league in little to no time.

HOW TO ACCESS THIS PAGE:

1) From the **Home** page (or any page), click on the **About Us** button at the top.

5.5 Player Stats Page



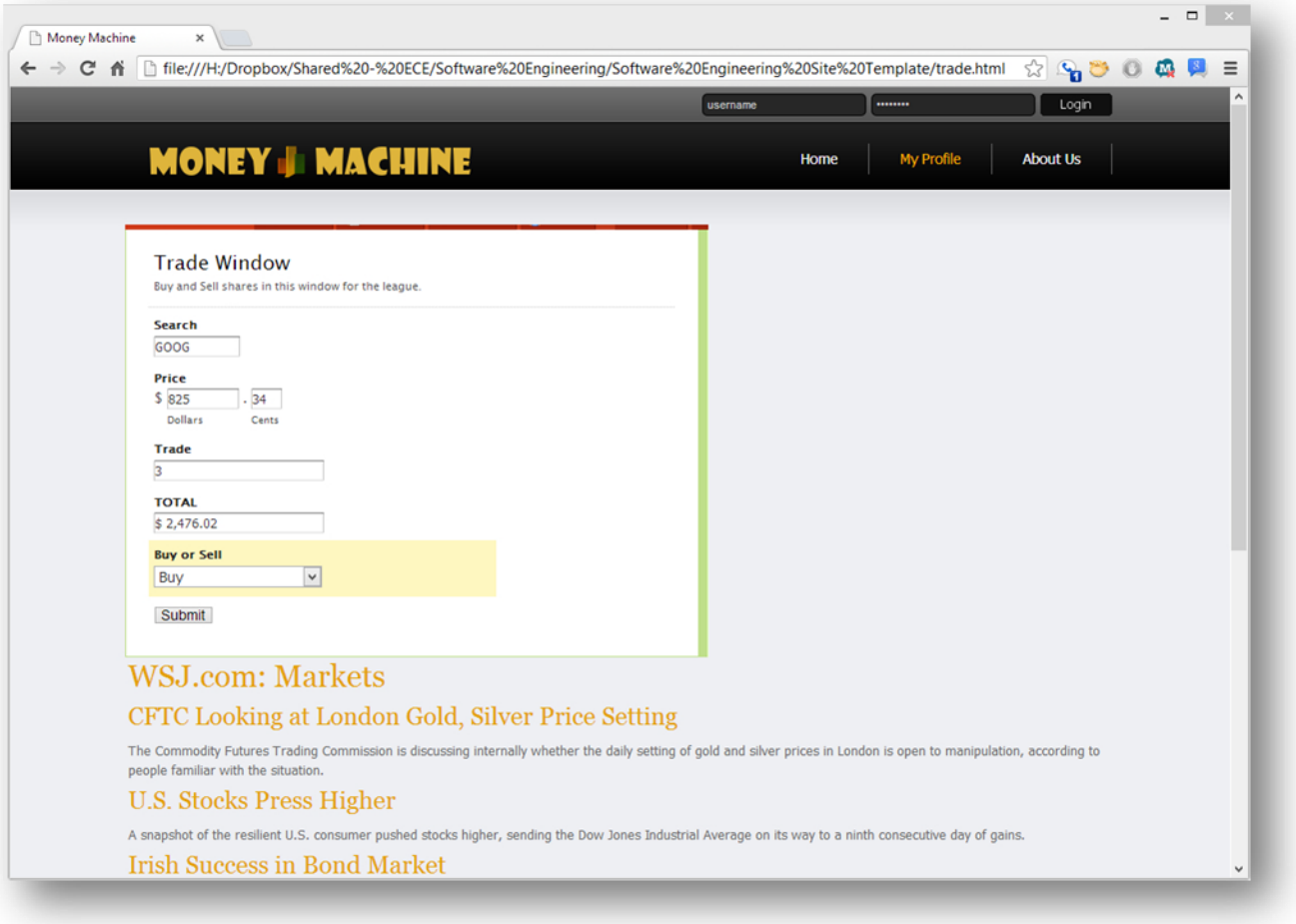
Player Stats Page: This view is created for demonstration purposes only; the actual table has not been coded yet.

The "Player Stats" page can be viewed by hovering over the "My Profile" button and clicking on the "Player Stats" page or via 1 click by simply clicking on the "My Profile" button. This page shows the statistics of a player based on the league the player has joined and the players name, league name, league rank, net worth and the over gains for that particular league.

HOW TO ACCESS THIS PAGE:

1) From the **Home** page (or any page), hover over the **My Profile** button.
2) Click on **Player Stats.**

5.6 Trade Page



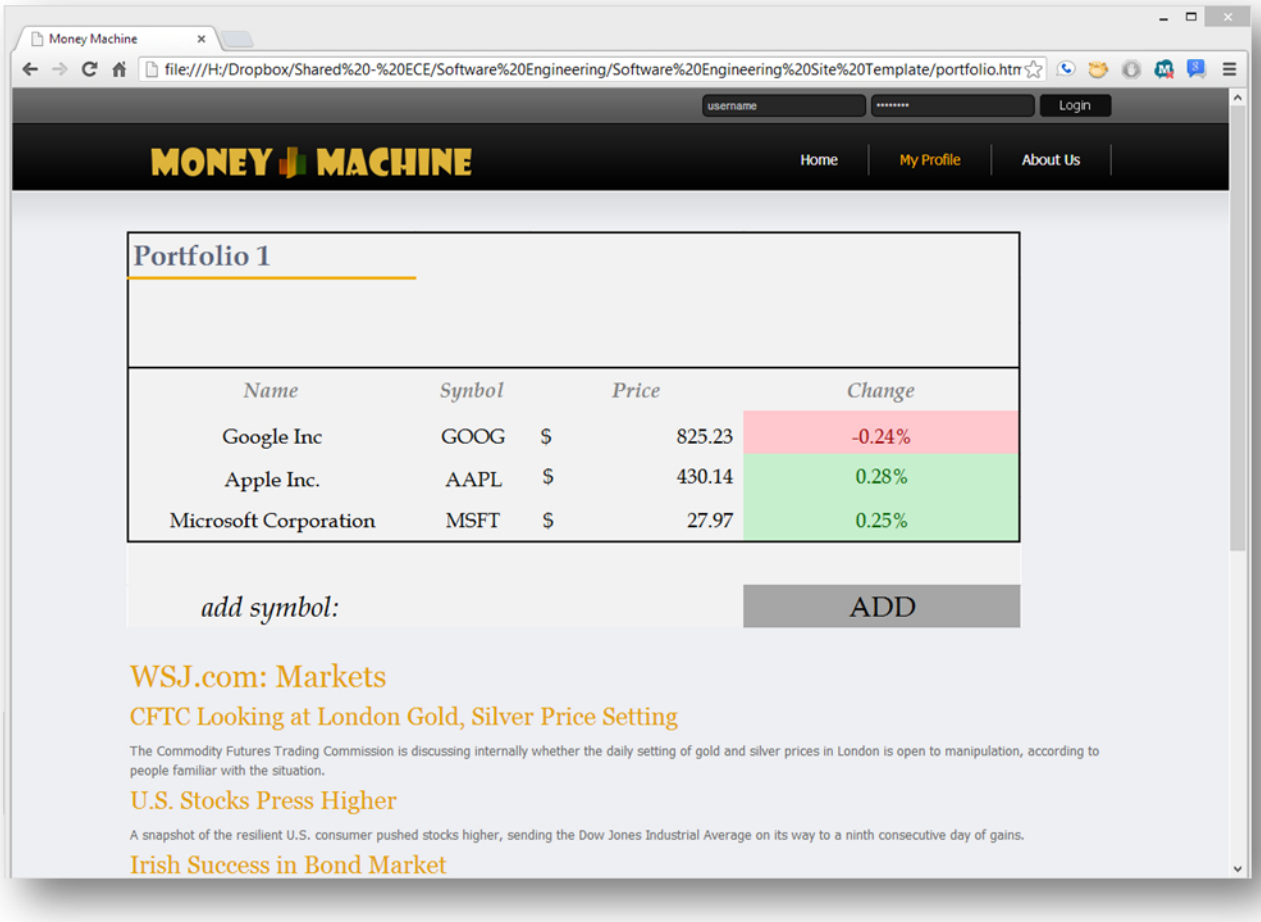
Trade Page: This view is created for demonstration purposes only; the actual table has not been coded yet.

The "Trade" page can be accessed from the "My Profile" button and selecting the Trade option. This page provides access to the stock ticker prices as well as the amount of shares the player wishes to trade or purchase. In example shown it can be seen that the player is trying to make a purchase of 3 Google shares priced at $825.34 per share making the total $2,476.02.

<u>HOW TO ACCESS THIS PAGE:</u>

1) From the **Home** page (or any page), hover over the **My Profile** button.
2) Click on **Trade.**

5.7 Portfolio Page



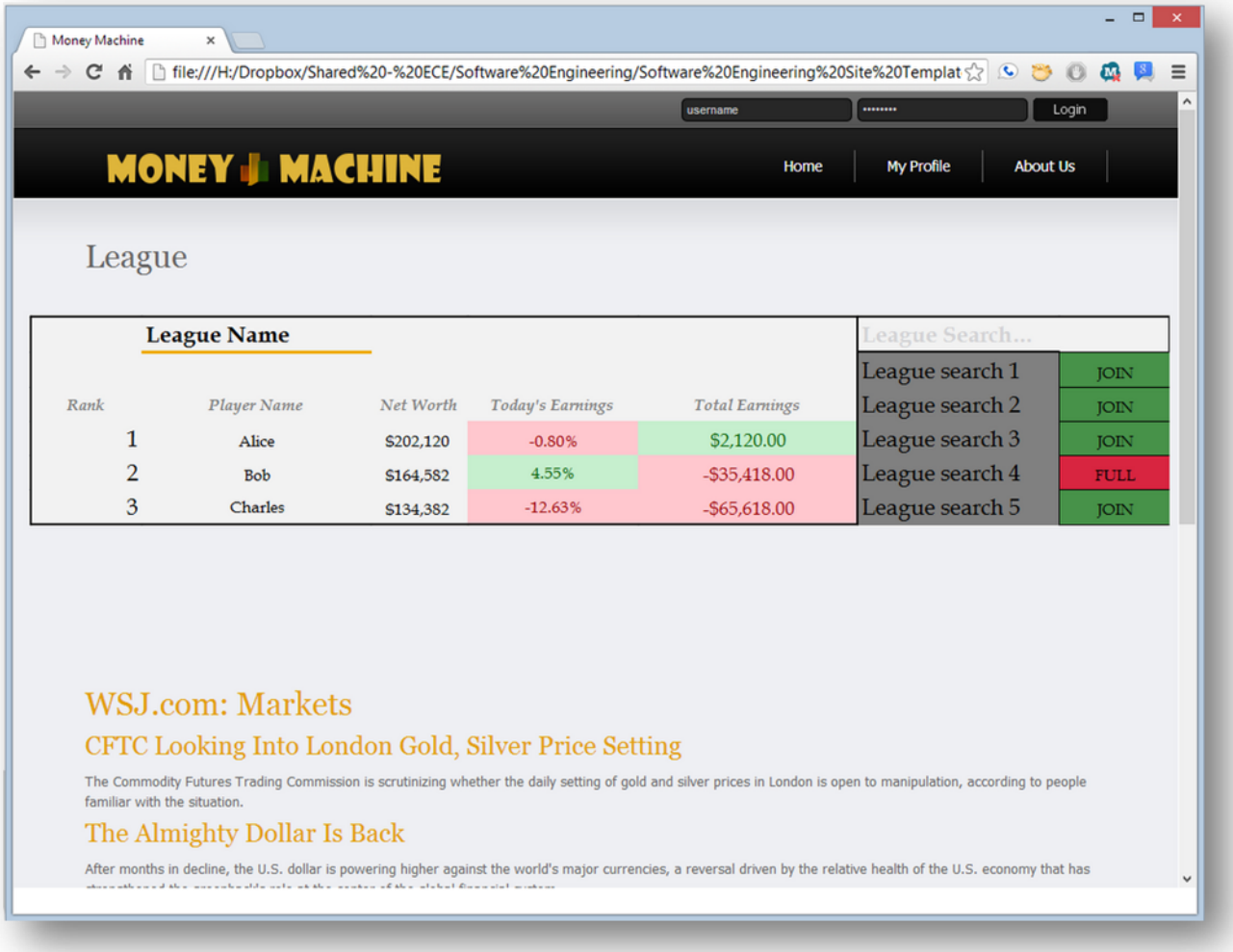
Portfolio Page: This view is created for demonstration purposes only; the actual table has not been coded yet.

The portfolio page offers the player to maintain a stock portfolio for demonstration purposes a sample portfolio has been created called *Portfolio 1* which contains the following stocks, *Google Inc, Apple Inc,* and *Microsoft Corporation.* Each name shows the associated symbol along with its price and the market change price in percentage. Within this page multiple portfolios can be managed and viewed. The add symbol allows the player to search for a symbol and add it to the portfolio.

<u>HOW TO ACCESS THIS PAGE:</u>

1) From the **Home** page (or any page), hover over the **My Profile** button.
2) Click on **Portfolio.**

5.8 League Page



League Page: This view is created for demonstration purposes only; the actual table has not been coded yet.

The league page offers players the ability to join different leagues and view the statistics of the currently joined league. To get to the league page the player has to hover over the *My Profile* button and selecting the *League* option. There is a search option on the right side of the *League* page and it also shows the player if the league is joinable or if it is full, meaning that the host of that particular league has set a cap on the amount of players allowed in the league.

HOW TO ACCESS THIS PAGE:

1) From the **Home** page (or any page), hover over the **My Profile** button.
2) Click on **League.**

6.0 Design of Tests

The following are the tests designed for our system. We plan on updating the tests as we continue to develop our software. These tests primarily encompass our unit testing scheme, however, there is a brief discussion of our integration testing technique.

6.1 Test Cases

Test Case: TC-01 [Log-In Page]

View Tested: virtualstockmark.views.login

Pass/Fail Criteria: This test case will check if the Player has provided correct user name and password successfully. However, the user must be registered with system.

Test Procedure	Results	Actions
Call Function	Pass	User should be able to log in to the system and able to see his portfolio.
Call Function	Fail	If user haven't provided correct user name and password that is registered with the system. This can also be a case when user click log-in button without providing any information. As results, it should notify user and request for the correct information showing (*) next to mandatory fields.

Test-case TC-02: [Validity Checker]
Function Tested: ticket_system.valid()
Pass/Fail Criteria: This test determines if Player has been able to successfully place an order in the market.

Test Procedure	Results	Actions
Call Function	Pass	Player has been successfully able to place an order in the market. The order will be placed only if Player has enough cash balance.
Call Function	Fail	If Player doesn't have enough cash balance, then it won't let Player to place an order.

Test Case: TC-03 [Create League]

View Tested: league.views.create_league

Pass/Fail Criteria: This test case will check to see if the inputs provided by the Player are valid or not.

Test Procedure	Results	Actions
Pass League Name Input	Pass	Player should be presented with the manage league page for the newly created league.
	Fail	The Player will be presented with an error page letting them know that the league name is already taken and will be given an opportunity to try again.

Test Case: TC-04 [Registration Page]

View Tested: virtualstockmark.views.register

Pass/Fail Criteria: This test case will check if the Player has provided correct values for the specified fields.

Test Procedure	Results	Actions
Pass Inputs for First Name	Pass	Player has entered correct values and can proceed to the next field.
	Fail	Display error message and player has to modify the field.
Pass Inputs for Last Name	Pass	Player has entered correct values and can proceed to the next field.
	Fail	Display error message and player has to modify the field.
Pass Inputs for User Name	Pass	Player has entered a unique username and can proceed to the next field.
	Fail	User name is taken and Player must choose a different username
Pass Inputs for Password	Pass	Player has entered a valid password and can proceed to the next field.
	Fail	Display error message and player has to modify the field.
Pass Inputs for Email	Pass	Player has entered correct email address and can proceed to the next field.
	Fail	Display error message and player has to modify the field.
Pass Inputs for Confirm Email	Pass	Player has re-entered the correct email address and can proceed to the next field.
	Fail	Display error message, player has not entered the same email address as in the previous field and player has to modify the field.
Click Register button	Pass	Player has entered all correct information and account is created and is forwarded to the Portfolio screen.

Test Case: TC-05 [Challenge Player]

View Tested: virtualstockmark.views.league

Pass/Fail Criteria: This test case will check if a challenge player request is successful or unsuccessful.

Test Procedure	Results	Actions
Pass Input Challenge	Pass	Player should be presented with a page with time until challenge begins and challenged player's name.
	Fail	The Player will be presented with an error page stating the player is already in a challenge with another player, or invalid challenge request.

Test Case: TC-06 [Data Handler]

View Tested: virtualstockmark.views.orderticket

Pass/Fail Criteria: The test passes if the test stub executes the ticket by updating the investor's portfolio accordingly

Test Procedure	Results	Actions
Execute Order	Pass	DataHandler executes order and updates investors portfolio and returns tree
	Fail	If unable to execute order, return false.

Test Case: TC-07 [Data Handler]

View Tested: virtualstockmark.views.portfolio

Pass/Fail Criteria: The test passes if the test stub request for portfolio data and it is retrieved from the database.

Test Procedure	Results	Actions
Request Portfolio Data	Pass	DataHandler request portfolio data and returns it from the database.
	Fail	If there is an error retrieving the data from the database, it should display an error that no data was returned.

Test Case: TC-08 [Data Handler]

View Tested: virtualstockmark.views.league

Pass/Fail Criteria: This test case will check if a challenge player request is successful or unsuccessful.

Test Procedure	Results	Actions
Request to Edit League Settings	Pass	DataHandler modifies league settings and returns true.
	Fail	DataHandler unable to modify league settings, returns false.

Test Case: TC-09 [DataHandler]

View Tested: virtualstockmark.views.portfolio

Pass/Fail Criteria: The test passes if the test stub request to view transaction history from the database is successful.

Test Procedure	Results	Actions
Request to View Transaction History	Pass	DataHandler returns the transaction history.
	Fail	DataHandler displays an error message, unable to retrieve transaction history.

Test Case: TC-10 [Data Handler]

View Tested: virtualstockmark.views.league

Pass/Fail Criteria: This test case will check if a challenge player request is successful or unsuccessful.

Test Procedure	Results	Actions
Request To Join League	Pass	DataHandler updates information in database about the league and returns true.
	Fail	DataHandler returns false if joining league encounters problem.

Test Case: TC-11 [Data Handler]

View Tested: virtualstockmark.views.league

Pass/Fail Criteria: This test case will check if a challenge player request is successful or unsuccessful.

Test Procedure	Results	Actions
Send invite Player to League	Pass	DataHandler adds the invite to the Players account in database.
	Fail	If fails, displays message unable to send or add invite the player.

Test Case: TC-12 [Validate Login]

View Tested: virtualstockmarket.views.login

Pass/Fail Criteria: To verify that the player has entered either or both username/password and either of these fields are not left blank.

Test Procedure	Results	Actions
Pass input Username/Password	Pass	Player has entered the username/password and is able to login.
	Fail	The Player will be presented with an error on the login page stating the player username and password fields are left blank.

Test Case: TC-13 [Validate Logout]

View Tested: virtualstockmarket.views.logout

Pass/Fail Criteria: To verify that the player is not able to click the back button after clicking the logout button.

Test Procedure	Results	Actions
Pass input Logout	Pass	Player has clicked on logout and is not able to click back after arriving at the logout page.
	Fail	The player is able to click on the back button in the browser even after successfully logging out.

6.2 Coverage of Tests

The test cases are planned to cover all of the possible models and views for every "app" in the Money Machine Project. However, due to the nature of the language being a MVC style language, it is very hard to test individual classes. Nonetheless, it is envisioned that all of the test cases will address all aspects of the application. It is planned that about 75% of the test cases will focus on transition states (ex- making sure that the application transitions from the league app to the portfolio app when the user wants to view a portfolio in a league.) The remaining 25% will be devoted to creating tests that test the UI specification.

6.3 Integration Testing Plan

The integration testing will be done with each component individually at first and then with other components for the project. The basic template of the website was written to make sure that each individual page can be accessed from another page. Once the templates for the sites are done, the team will begin to develop the actually methods and models that will be used for the project such as the Portfolio System, League System, Ticker System, Login System, and Registration System. The Registration system was first developed separately to test if the databases are working properly and then it was integrated into the website templates that were originally created so any visitor to the website can register for Money Machine. Once the Registration System is working it can be used in conjunction with the Login System for user authentication. After the Login System is created, testing is done on the system to see if a player is able to register properly and also able to login using the username that was created. Further testing of the authentication of the system has to be done to maintain a secure logout. For example, if an authenticated user clicks on the *Logout* button then the player must be brought back to the home page of the website and must be re-authenticated if the player decides to click on the *Back* button in the web browser.

Once the login system is working properly, the Portfolio System and the Ticker System are going to be developed separately, with a higher priority on the Ticker System. The Ticker System is one of the most important aspects of this project because it will handle all the queries for buying and selling stocks. The Ticker System has to be tested thoroughly to make sure that each buy and sell query is handled properly. Once this system is working properly it can be integrated with the Portfolio System, League System, and Registration System. At first, it will be tested with the Registration/Login System to make sure that each individual player is able to buy or sell stocks and if the Ticker System is able to reflect the transactions to the

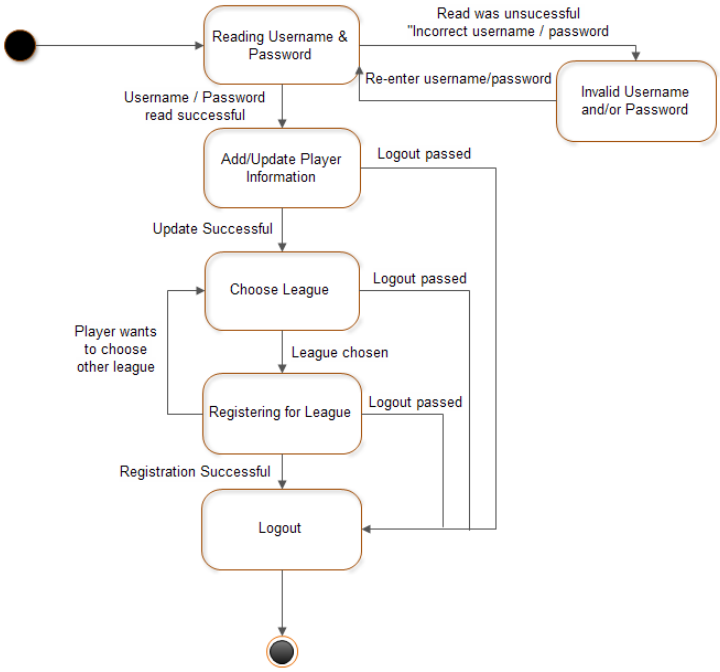
individual player. Once this test is done, it will be integrated with the Portfolio System, so the player can start building his/her portfolio. While this testing and debugging of the integrated system is being done, the League System will also start to be developed by 2-3 group members so there is no delay in the software development. By the time the League System is done, it is expected that the integration between the Ticker System, Portfolio System, and the Login System are working in unison. The League System will be tested at first if it can handle faux data that the team will generate, such as, player names, net worth, rank, and daily loss or gain. If this test passes then it will be integrated with the all the previous systems to full complete the project. In the end, if all the systems are working, a player should be able to register for Money Machine account, create and maintain a portfolio, join a league, obtain statistics about currently joined leagues, and buy and sell shares and have the transactions reflect within a portfolio.

At first, each system will be tested individually with some sort of faux data that will be generated to make sure that individual system is working properly before being integrated with other systems. Testing and debugging is a major component of software development, however, if the debugging time is far too great than some aspects of the project might not get properly debugged due to project deadlines. Since each stage of the application development is being tested individually, hopefully this reduces the amount of errors when each system is integrated with one another.

6.4 Testing State Diagrams

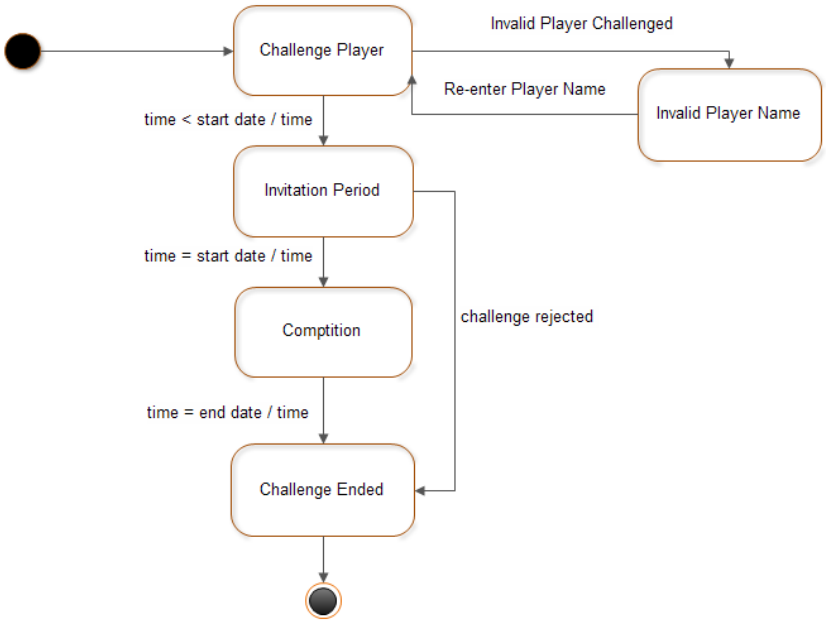
State Diagram: Registration / Login

The following state diagram below shows how registration and login by a player could be tested.



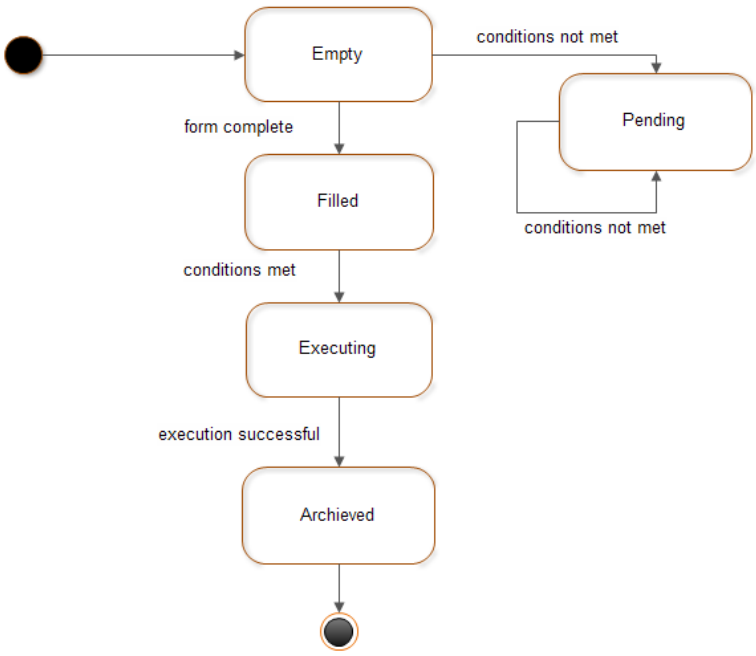
State Diagram: Challenge Player

The following state diagram below shows how the function challenge a player is tested.



State Diagram: Order Ticket

The following state diagram below shows how a order ticket is tested.



7.0 Project Management & Plan of Work

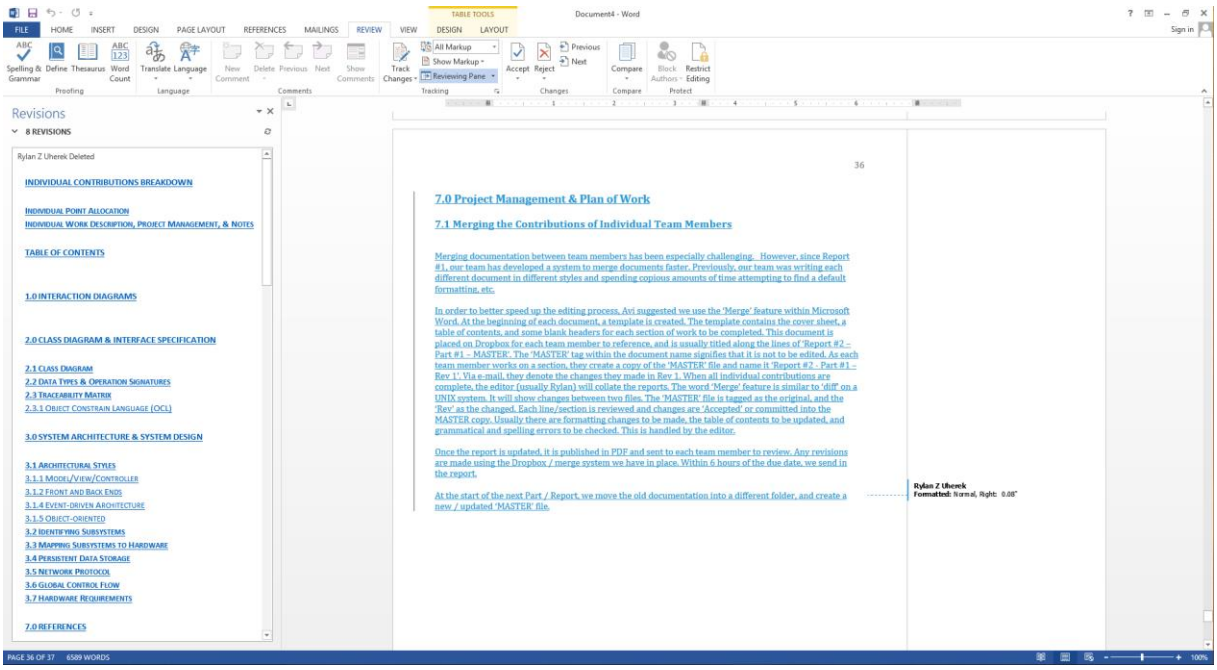
7.1 Merging the Contributions of Individual Team Members

Merging documentation between team members has been especially challenging. However, since Report #1, our team has developed a system to merge documents faster. Previously, our team was writing each different document in different styles and spending copious amounts of time attempting to find a default formatting, etc.

In order to better speed up the editing process, Avi suggested we use the 'Merge' feature within Microsoft Word. At the beginning of each document, a template is created. The template contains the cover sheet, a table of contents, and some blank headers for each section of work to be completed. This document is placed on Dropbox for each team member to reference, and is usually titled along the lines of 'Report #2 – Part #1 – MASTER'. The 'MASTER' tag within the document name signifies that it is not to be edited. As each team member works on a section, they create a copy of the 'MASTER' file and name it 'Report #2 - Part #1 – Rev 1'. Via e-mail, they denote the changes they made in Rev 1. When all individual contributions are complete, the editor (usually Rylan) will collate the reports. The MS Word 'Compare' feature is similar to 'diff' on a UNIX system. It will show changes between two files. The 'MASTER' file is tagged as the original, and the 'Rev' as the changed. Each line/section is reviewed and changes are 'Accepted' or committed into the MASTER copy. Usually there are formatting changes to be made, the table of contents to be updated, and grammatical and spelling errors to be checked. This is handled by the editor.

Once the report is updated, it is published in PDF and sent to each team member to review. Any revisions are made using the Dropbox / merge system we have in place. Within 6 hours of the due date, we send in the report.

At the start of the next Part / Report, we move the old documentation into a different folder, and create a new / updated 'MASTER' file.



If you notice in the above picture, you can see the blue text, which denotes a change to the original document. The markup on the right side shows which contributor made the change and which formatting differences exist. At the top of the document, an 'Accept' and 'Reject' button allow the editor to make the changes.

Team members have been using both Microsoft Word & Google Documents (a shared folder for our group) to complete their work. This doesn't matter, as both can export to the 'doc' file format for merging.

This document merging technique has sped up the merging and editing process for our team significantly. Because members have a copy of the formatting when they write their documents, as much time isn't spent reformatting pages to meet the visual consistency needed.

In terms of ensuring consistency, our team has a set format table of contents, cover page, and style for within the documents. This setup was created by Rylan and is periodically edited as needed. During the editing process, he ensures that all styles match, and the pages have a clean, consistent, and professional look to them.

7.2 Project Coordination & Progress Report

7.2.1 Project Coordination

Project coordination has been mainly Rylan's responsibility, however, all team members are a part of the coordination process. A brief description of the main responsibilities undertaken by the Project Manager (PM):

1. Coordinating Meetings – The PM is responsible for coordinating meetings. There are two different kinds of meetings, coding meetings (where coding and functionality changes are discussed) and report meetings, which focus on reports. At times, these topics are discussed in single meetings. For the most part, meetings are coordinated to be on Friday mornings, and Monday nights, with e-mail interchange facilitated throughout the week. We are currently experimenting with new meeting times to better coordinate our teams. Meeting coordination involves planning the meeting times, developing meeting agendas (usually pertinent upcoming due date tasks), and sending out 'meeting minutes' or a summary of meeting discussions for each member to have a copy of post-meeting.

2. Document Control – The document control task involves editing each member's contributions to reflect the template and styling elements of each report. After all, each report must *look* as though it came from one person. The PM also handles Dropbox & Google documents version tracking as needed. During the revision process, the PM assures that each individual contribution matches the requirements from the course website and notifies each contributor of any discrepancies.

3. Interface to Instructors – The PM serves as the point of contact for the instructors to the group. The PM is a group representative to contact the instructors with e-mail questions, report submitting, etc.

4. Group Status & Planning – The PM is also tasked with ensuring the schedule as set forth in Report #1 is met. This means working with the group to divide work amongst members, and setting up 'soft' due dates well before the due date. While each section is being worked on, the PM facilitates group

interaction, ensuring that each part will be completed in time, adjusting timelines as needed, and ensuring each group member is comfortable with the other's contributions. The PM ensures that all grade-related tasks are visible to each group member, and asks for a general consensus on all work before submitting. This allows each group member to voice pointed or general concerns, and make modifications as needed. In turn, our group works well together since open communication is fostered at each stage of the project process.

5. Code Planning – The PM has some background experience with Django and Python coding. This enables the PM to work with team members to develop a project execution process and work on developing the sections needed for coding. Tackling a large project in coding requires building in stages. Because testing needs to be completed at each stage, the PM has been working to schedule coding in stages which can be built off one another. For example, our team could have developed a portfolio and stock selling procedure first. However, we found it easiest to start by building a registration and login system. Next, a portfolio system will be developed in parallel with a league system. These systems will later be integrated. Developing such a timeline and implementation strategy is one of the responsibilities of the PM. The PM doesn't do this alone, but has a large say in this portion of project strategy.

6. Version Control – The PM works to ensure the version control systems are setup and initialized with the needed code. This involved setting up the needed CVS (BitBucket [similar to GitHub]) was ready for team members, and published the initial code to it. The code was setup for development, as well as the database. This allowed team members to make a 'pull' from the repository, and begin coding without wasting time developing a configuration. The PM is tasked with administrating the CVS as needed.

These are not the only tasks of the PM. The PM has a variety of other functions within the group to facilitate the project development process (PDP). However, these tasks are a bulk of the PM's responsibility.

7.2.2 Project Progress

Our team has begun coding and is currently following the project timeline developed for Report #1. One of the largest challenges so far has been learning Python and learning the Django framework (all of us are new to Python and to Django). However, there is a tutorial every member of the team completed (the link to the tutorial can be found in the references section for this report).

The greatest features of the Django framework for us have been the tools included in Django, and how data has been abstracted. As opposed to using vanilla PHP or vanilla Python, Django abstracts data for the user and handles all connections to and from the database. Similarly, Django formulates all SQL queries for the user. For example, instead of saying 'SELECT * from users', you can say Users.objects.all(). This OOP approach is much easier than handling raw data. Likewise, our group is using the sqlite database which stores data in a local file. This enables us to quickly share the database structure with each other (as opposed to using MySQL and sending SQL dumps to each other). Django also handles creating tables for data once each object is defined as a 'Model'. We've found this to limit SQL development time, and overall development time.

So far our team has a template in Django, and a Registration and Login system built. We are currently experimenting with development options for each of the subsystems we are planning on developing: the

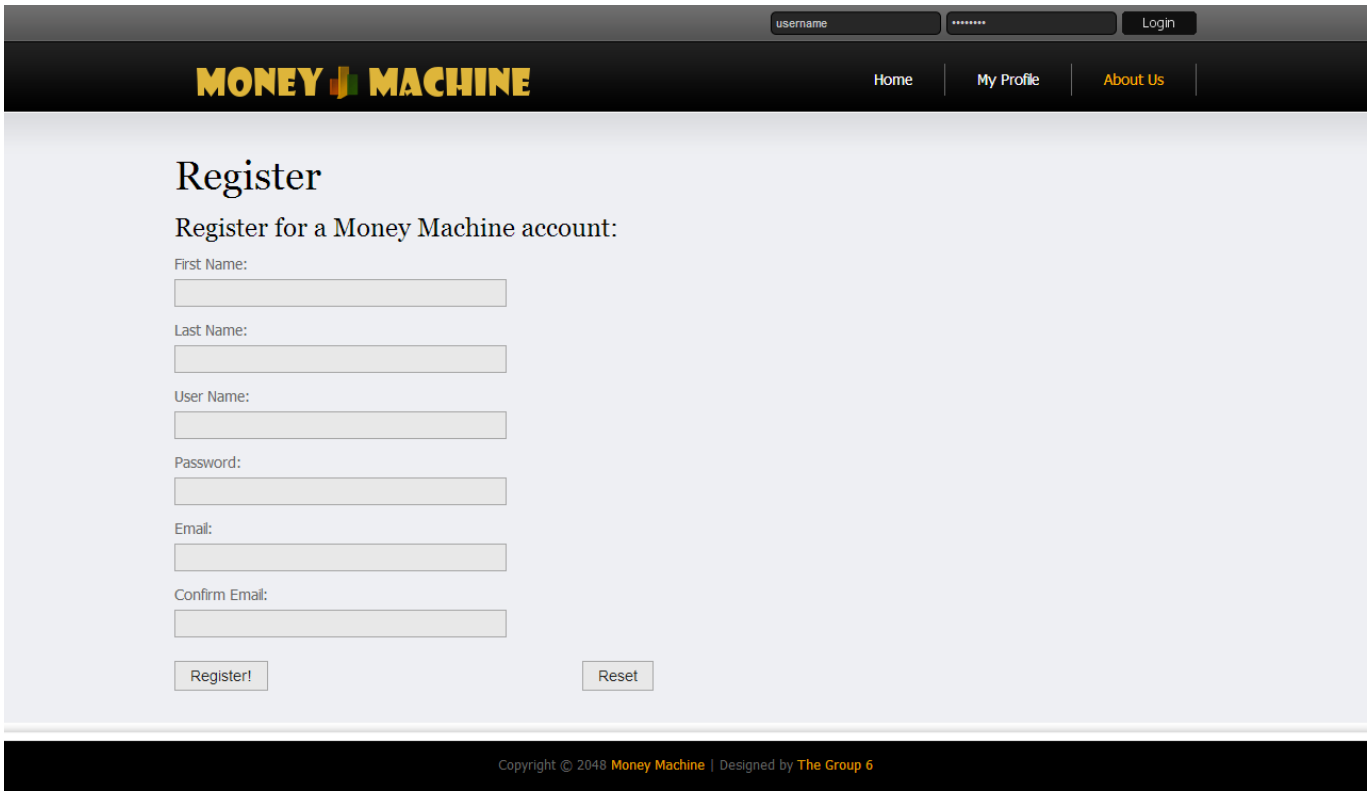
leagues and portfolios. Our goal is to have a basic, simple stock market league application ready for Demo #1. We plan on implementing our *Proposed* features with Demo #2. This is because most of our *Proposed* features rely on the basic features of the project. Likewise, if we spent most of our time preparing for the Demo #1 by developing proposed features it would be difficult to show any *working* project during the Demo.

As with any progress, there are some drawbacks. So far we've also developed a simple stock quote class, *SecurityDataProvider*. One of our major proposed features involved the ability to buy and sell stock options. However, we found that finding *retrievable* information on stock options is not easy. Because options change very frequently, the only way to access options data is to pay for it. Our team was considering footing the bill (if it were maybe $5 for 10,000 API calls), but it's over $200 a month for a contract to get basic options values. We looked into alternative methods such as downloading the HTML of a page, and parsing it for data. Yet, providers such as Yahoo Finance and Google Finance kept such data locked away from peering eyes. Therefore, we have no option but to forgo the inclusion of options from our project. Depending on our project timeline, we will either develop an alternative feature, or enhance the implemented features.

Great strides have been made with the UI development. Even though the project is meant to be functional (functionality is more important than visuals), the visuals our team has developed are of professional quality. The updates can be seen in the UI section of this report. However, here are some of the screenshots of pages which have already been implemented:

Home Page

User Registration Page



As mentioned earlier, our team has also made strides in coding. Below is our *SecurityDataProvider* code. We plan on expanding the code to provide full documented functionality as we continue development.

```
#!/usr/bin/python

# Broker API
# Written by Rylan Uherek

"""

NOTES:
API Makes calls to Yahoo Finance API for Stock Quotes
reference sheet: http://greenido.wordpress.com/2009/12/22/yahoo-finance-hidden-api/

USAGE:
import  broker
print broker.get_data('AAPL', 'p')

"""

import urllib

# s_get_data(symbol, stat)
# symbol -- ticker symbol (e.g. 'AAPL')
# stat -- statistic on ticker (see the yahoo finance API)
# RET: the data requested by the stat
```

```
def s_get_data(symbol, stat):
    # make a call to the yahoo finance api
    url = 'http://finance.yahoo.com/d/quotes.csv?s=%s&f=%s' % (symbol, stat)
    return urllib.urlopen(url).read().strip().strip('"')

# s_get_all_data(symbol)
# symbol -- ticker symbol (e.g. 'AAPL')
# RET: all relevant data requested by the stat in a dictionary

def s_get_all_data(symbol):
    # get all of the relevant data for the stock / etf
    stock_info = s_get_data(symbol, 'l1nghopxt8').split(',')

    data = {}
    data['price'] = stock_info[0]
    data['name'] = stock_info[1]
    data['low'] = stock_info[2]
    data['high'] = stock_info[3]
    data['open'] = stock_info[4]
    data['close'] = stock_info[5]
    data['exchange'] = stock_info[6]
    data['1y-target'] = stock_info[7]

    return data

dict = s_get_all_data('MSFT')
for entry in dict:
    print "%s %s" % (entry, dict[entry])
```

Some of our code used to generation the user registration page:

```
from django.template import Context, loader
from django.http import HttpResponse, HttpResponseRedirect
from django.shortcuts import render_to_response, get_object_or_404
from django.template import RequestContext
from django.contrib.auth.models import User
from django.contrib.auth import authenticate
from django.contrib.auth import logout as auth_logout
from django.contrib.auth import login as auth_login
from django import forms

from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse

from registration import RegistrationForm


from datetime import datetime
from django.utils.timezone import utc
from calendar import monthrange
import calendar

def home(request):
    context = RequestContext(request)
    return render_to_response('home.html', context)
```

```
# register view
def register(request):
    context = RequestContext(request)
    newForm = RegistrationForm()

    if request.POST:
        form = RegistrationForm(request.POST)
        if form.is_valid():
            new_user = form.create_user()


            return render_to_response('home.html', context)
        else:
            # return the old form and any errors
            newForm = form

    return render_to_response('register.html', {'form':newForm} , context)
```

Overall our team is working together well. We have implemented the basic features we've needed for our site and are now working on implementing more complex features. We are very proud of the UI and the features we have implemented so far.

7.3 Plan of Work

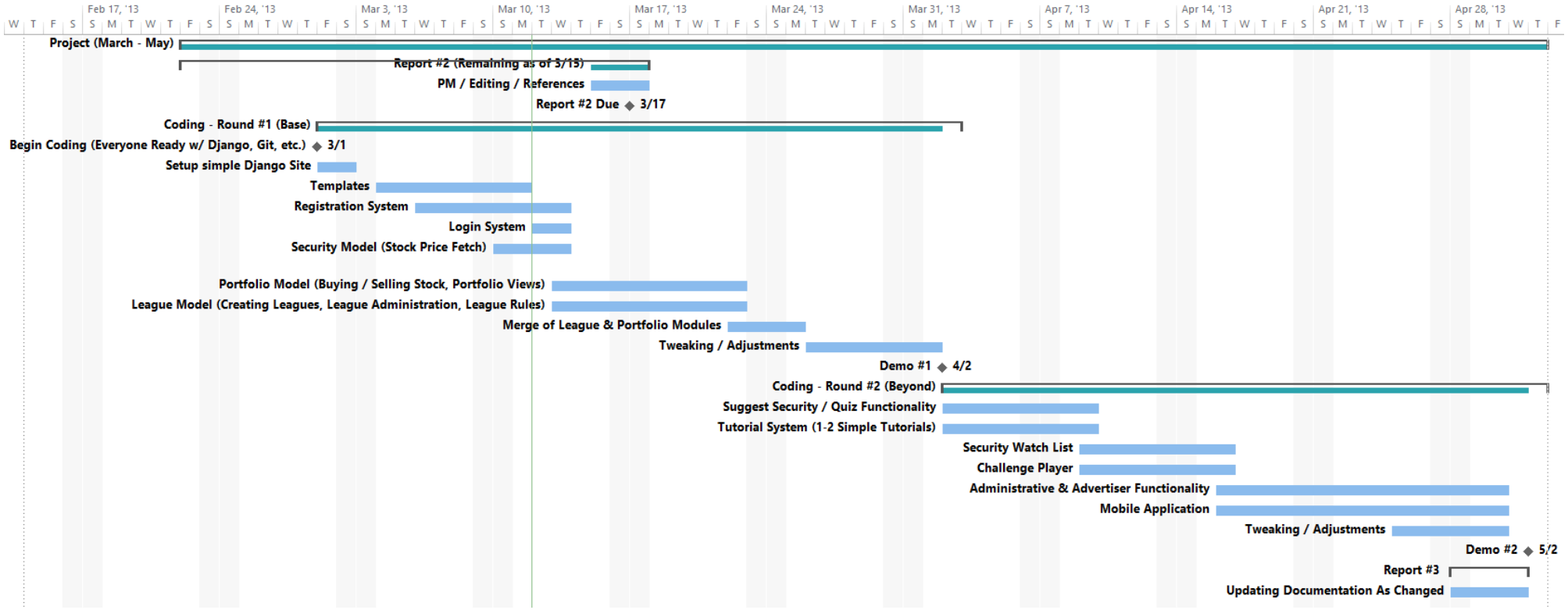
Below is our plan of work. It is a modification of the plan of work provided at the end of Report #1. Because report #2 is now complete, the roadmap is primarily concerned with coding and development. Report #3 is simply a collation and modification of Reports #1 & #2. It will be updated at the end of the development phase (the end of April before the project is due). This will allow the last Report to have the most accurate project documentation.

The Gantt chart has 4 Sections. The first section involves the completion of Report #2 (minimal). The second section involves Round #1 of coding. This is our 'base'. All *proposed* features will be implemented once this base works correctly. The Round #2 of coding is the next section, and involves the implementation of features we've proposed, such as a tutorial system, watch list, suggest-a-stock functionality, etc. Finally, the last section spans about 1 week, which will give us the time needed to update Reports #1 and #2 to meet the system we *actually* designed, and document it correctly.

Of course, this diagram is subject to change depending on how the project schedule plays out.

As much as possible, there are two major tasks being developed at each time. This allows our team of 6 to divide into 2 sub-teams and code independently of each other. At milestones in the project (TBD), we plan on meeting together, and merging our completed code as needed. This strategy is *entirely* different from the one in our group proposal. However, by separating into two teams, and now having a much more refined view of the work needed to complete this project, we see that having these split teams will benefit us in completing the most amount of work in the shortest time possible.

7.3.1 Gantt Chart



7.3.2 Task List

This task list goes with the Gantt chart of 7.3.1.

Task Name	Duration	Start	Finish
Project (March - May)	**50 days**	**Fri 2/22/13**	**Thu 5/2/13**
Report #2 (Remaining as of 3/15)	**17 days**	**Fri 2/22/13**	**Sun 3/17/13**
PM / Editing / References	2 days	Fri 3/15/13	Sun 3/17/13
Report #2 Due	0 days	Sun 3/17/13	Sun 3/17/13
Coding - Round #1 (Base)	**23 days**	**Fri 3/1/13**	**Tue 4/2/13**
Begin Coding (Everyone Ready w/ Django, Git, etc.)	0 days	Fri 3/1/13	Fri 3/1/13
Setup simple Django Site	2 days	Fri 3/1/13	Sat 3/2/13
Templates	6 days	Mon 3/4/13	Mon 3/11/13
Registration System	6 days	Wed 3/6/13	Wed 3/13/13
Login System	2 days	Tue 3/12/13	Wed 3/13/13
Security Model (Stock Price Fetch)	4 days	Sun 3/10/13	Wed 3/13/13
Portfolio Model (Buying / Selling Stock, Portfolio Views)	8 days	Wed 3/13/13	Fri 3/22/13
League Model (Creating Leagues, League Administration, League Rules)	8 days	Wed 3/13/13	Fri 3/22/13
Merge of League & Portfolio Modules	2 days	Fri 3/22/13	Mon 3/25/13
Tweaking / Adjustments	5 days	Tue 3/26/13	Mon 4/1/13
Demo #1	0 days	Tue 4/2/13	Tue 4/2/13
Coding - Round #2 (Beyond)	**23 days**	**Tue 4/2/13**	**Thu 5/2/13**
Suggest Security / Quiz Functionality	6 days	Tue 4/2/13	Tue 4/9/13
Tutorial System (1-2 Simple Tutorials)	6 days	Tue 4/2/13	Tue 4/9/13
Security Watch List	6 days	Tue 4/9/13	Tue 4/16/13
Challenge Player	6 days	Tue 4/9/13	Tue 4/16/13
Administrative & Advertiser Functionality	11 days	Tue 4/16/13	Tue 4/30/13
Mobile Application	11 days	Tue 4/16/13	Tue 4/30/13
Tweaking / Adjustments	4 days	Thu 4/25/13	Tue 4/30/13
Demo #2	0 days	Thu 5/2/13	Thu 5/2/13
Report #3	**3 days**	**Sun 4/28/13**	**Wed 5/1/13**
Updating Documentation As Changed	4 days	Sun 4/28/13	Wed 5/1/13

7.4 Breakdown of Responsibilities

The following table illustrates the responsibilities of each group member. The contents of the table are subject to change as coding continues. We have tried to provide as many links to the classes each section envelops. The classes each team member will implement are to be determined at the start of the coding section. Under the table are footnotes which denote the classes each Task requires for implementation. These classes can be found in *Section 2.2* of this report. At times, multiple tasks may need to interface with the same classes. Some classes are extensions of Django classes.

7.4.1 Responsibility Table

Task	Rylan	Avi	Aakash	Mozam	Mandeep	Pintu
Project (March - May)	X	X	X	X	X	X
Report #2 (Remaining as of 3/15)	X	X	X	X	X	X
PM / Editing / References	X				X	
Report #2 Due	X	X	X	X	X	X
Coding - Round #1 (Base)	X	X	X	X	X	X
Begin Coding (Everyone Ready w/ Django, Git, etc.)	X	X	X	X	X	X
Setup simple Django Site[2]	X	X				
Templates[3]			X			
Registration System[4]	X	X		X		
Login System[5]		X		X		
Security Model (Stock Price Fetch)[6]	X	X				
Portfolio Model (Buying / Selling Stock, Portfolio Views)[7]	X	X			X	
League Model (Creating Leagues, League Administration, League Rules)[8]			X	X		X
Merge of League & Portfolio Modules	X	X	X	X	X	X
Tweaking / Adjustments	X	X	X	X	X	X
Demo #1	X	X	X	X	X	X
Coding - Round #2 (Beyond)	X	X	X	X	X	X
Suggest Security / Quiz Functionality[9]	X	X			X	
Tutorial System (1-2 Simple Tutorials)[10]			X	X		X
Security Watch List	X	X			X	
Challenge Player			X	X		X

[2] WebPage, PageRenderer, AdministratorAccount
[3] NewsFeeder
[4] DataHandler, PlayerAccount, FundManager
[5] Controller
[6] Controller, SecurityQuery, ValidityQuery, SecurityDatabase, Ticket, OrderList, History
[7] Shares, Portfolio, StopOrder, LimitOrder, MarketOrder
[8] LeagueHandler, League
[9] Fund
[10] TutorialDatabase

Task	Rylan	Avi	Aakash	Mozam	Mandeep	Pintu
Administrative & Advertiser Functionality[11]	X	X			X	
Mobile Application			X	X		X
Tweaking / Adjustments	X	X	X	X	X	X
Demo #2	X	X	X	X	X	X
Report #3	X	X	X	X	X	X
Updating Documentation As Changed	X	X	X	X	X	X

Integration testing will be performed by the team as a whole. Each coder will be responsible for debugging their own code. When the code comes together within their branch (e.g. the Portfolio sub-team), the sub-team will designate one of their members to perform overall testing. Before the branch is merged into the master branch, the team will meet together and work together for integration. Integration testing will be performed by the PM under the supervision of the team.

Integration coordination will fall under the role of the PM. The PM has already developed a schedule for when coding blocks are to be completed. When the blocks are complete, the PM will work with the sub-teams to schedule integration before moving onto the next project phase.

[11] AdvertiserAccount, AdvertisementManager

8.0 References

1. Marsic, Ivan. Software Engineering. Rutgers University, unpublished. 2012. Web
2. “UML Class Diagram Help”, Class Draw. Macrospark Solutions, n. d. Web. 03 Feb. 2013.
3. MarketWatch. MarketWatch, 18 Oct. 2011. Web. 29 Jan. 2013.
4. Group 6. Bears & Bulls. 03 May. 2012. PDF file
5. Group 2. Stockhop: The Stock Market fantasy League Game. n. d. PDF file
6. “Create Great Diagrams”, Gliffy. Gliffy. Web. 2012
7. Django Documentation, Django, Django Software Foundation, 24 Feb. 2013.