The Paramount Investments League

Report 2
Software Engineering
14:332:452

Team 1:

David Patrzeba
Eric Jacob
Evan Arbeitman
Christopher Mancuso
David Karivalis
Jesse Ziegler

March 19, 2014

Hyperlinks:

Webapp Link
Project Repository
Reports Repository

Revision History:

Version No.	Date of Revision
v.2.1	3/2/2014
v.2.2	3/9/2014
v.2.3	3/16/2014
v.2.4	3/19/2014

Contents

1 System Interaction Diagrams

1.1 Introduction

PLACEHOLDER TEXT –REPLACE WITH OUR OWN INTRO–

The following is an analysis of the interaction of UC-1 to UC-5. Within these use cases we touch on two of the most integral parts of our system, the persistant database, and the Yahoo! Finanace API adapter. The diagrams clearly describe the interactions of different subsystems to satisfy the use cases intent. Beyond the afformentioned subsystems, other subsystems are introduced

1.2 Diagrams

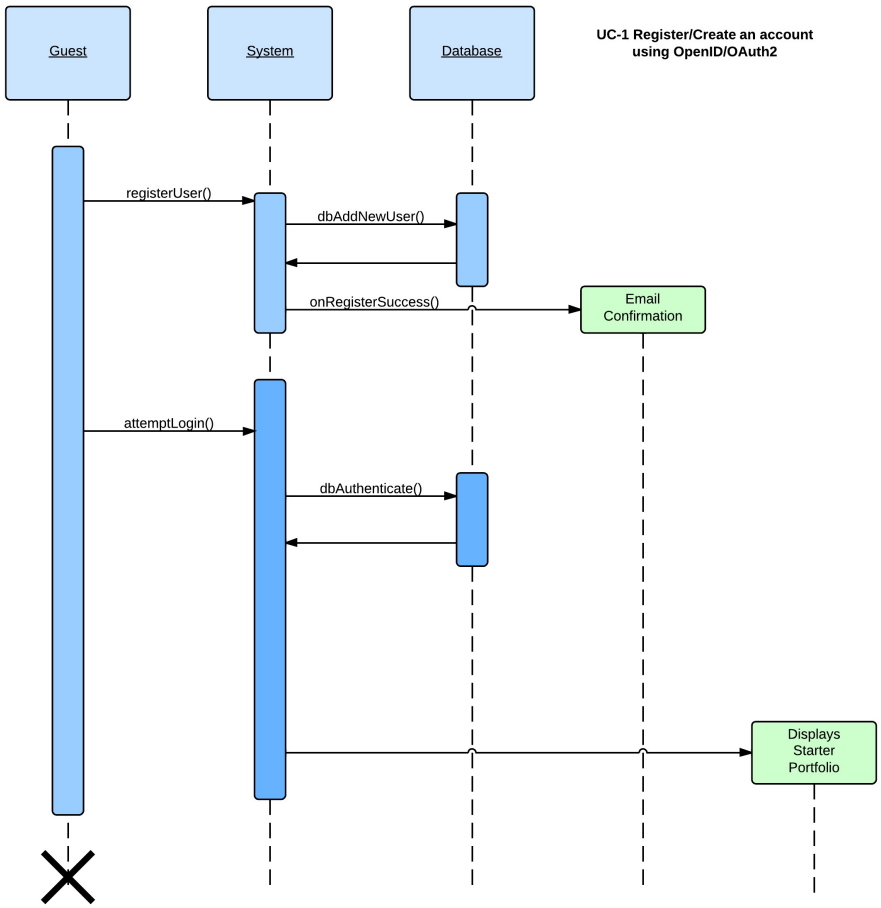


Figure 1.1: Shown in the sequence diagram for UC-1 begins with two options for the Guest. Either login or register an account. If a user attempts to register a new account the system is contacted with the users information. Then the system can attempt to check to make sure no duplicate login information exists in the database and if not it will store the new user information into the database. After this happens the user will be sent a confirmation email. If a user attempts to login, the system will attempt to authenticate the login details with details found in the database. If the details match correctly then the system will send the guest into investor mode and continue to upload their portfolio setting.

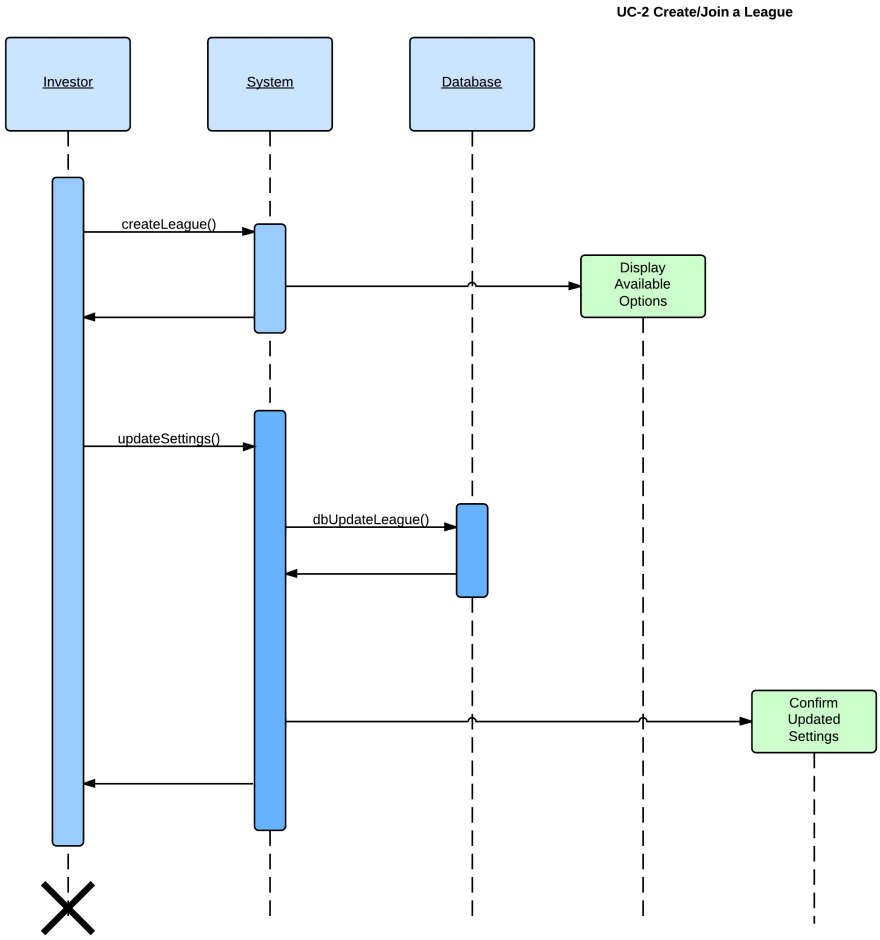


Figure 1.2: Shown in the sequence diagram for UC-2 is the flow of how to create an investment league. When an investor selects to create a league the system and more specifically the league controller will be contacted. The system will display the available options for creating a league. After there is a function updateSettings() which will create the league and process it in the database and also allow settings to be updated for a league. Not shown in the diagram is the alternative case of joining a league. The process to join a league is straightforward, where the league controller will show available leagues and then if an investor chooses to join they will be entered into the list in the database to associate with this league.

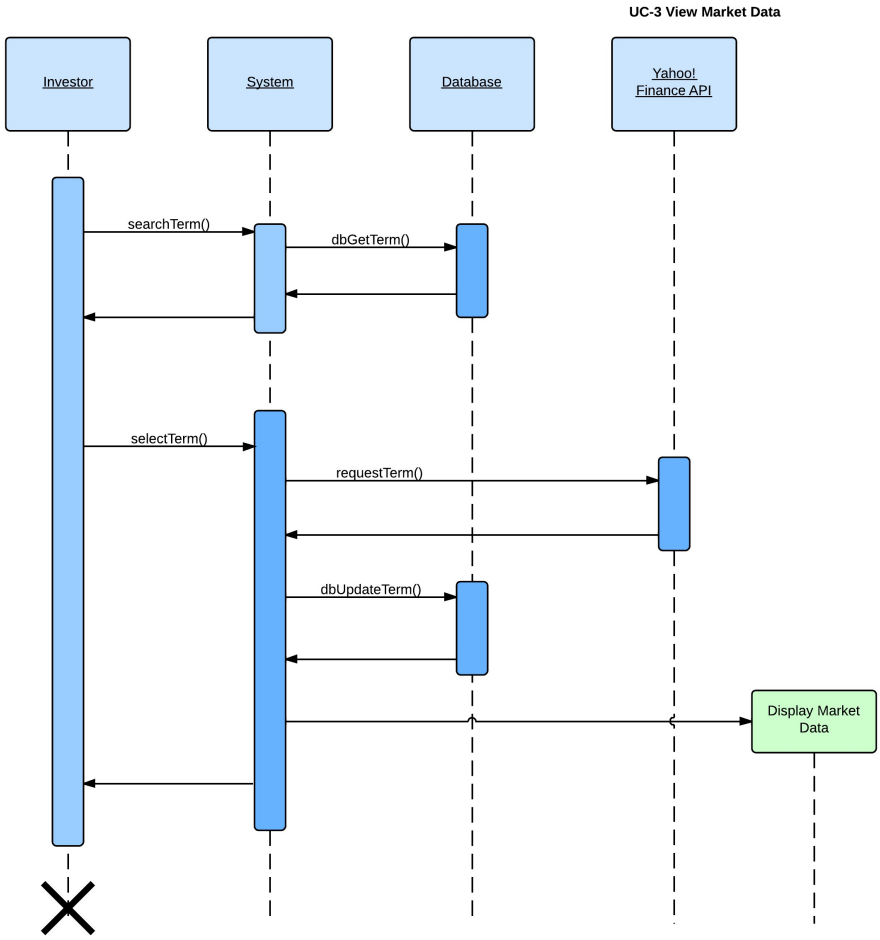


Figure 1.3: Viewing market data is accomplished by an investor searching a term. The system then finds this term which is most likely a company name or stock symbol. The system will fetch matches from the database and display them from the user. The investor will choose a match. The system takes the chosen term and requests its data from the Yahoo! Finance API. The system will update the database for this term, and then continue to display its market data to the investor.

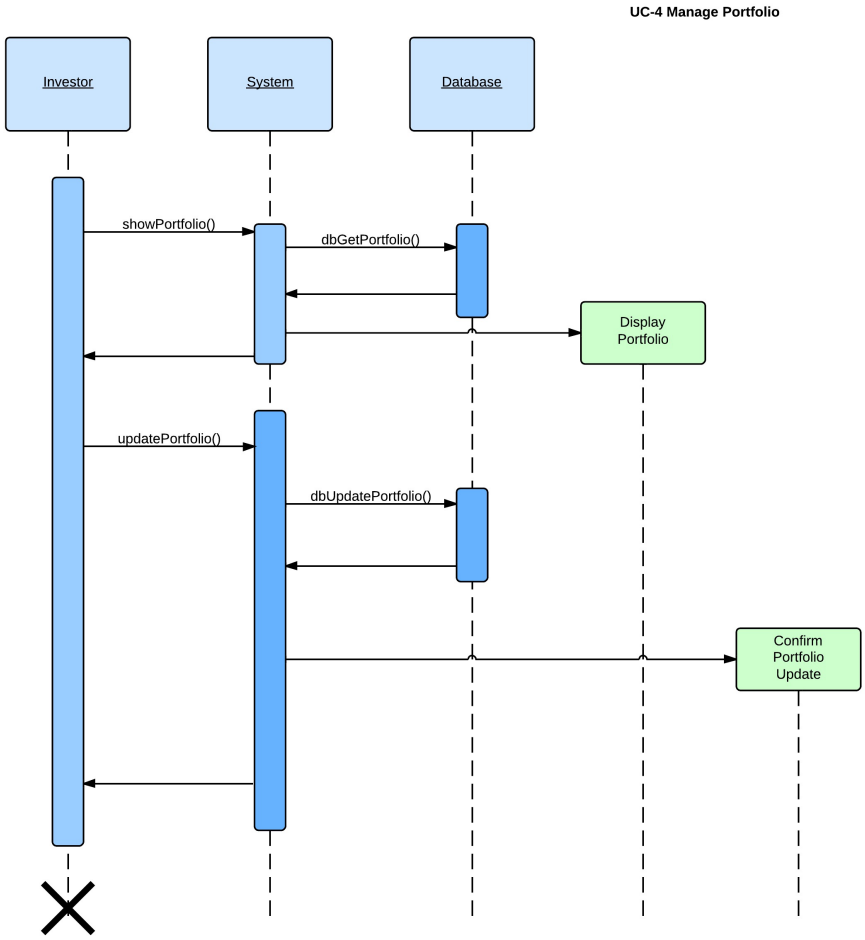


Figure 1.4: The investor should be able to view and make changes to their portfolio. The investor should be able to view their portfolio. The system will fetch the investors portfolio stocks from the database. The investor can also update their view of the portfolio and other settings.

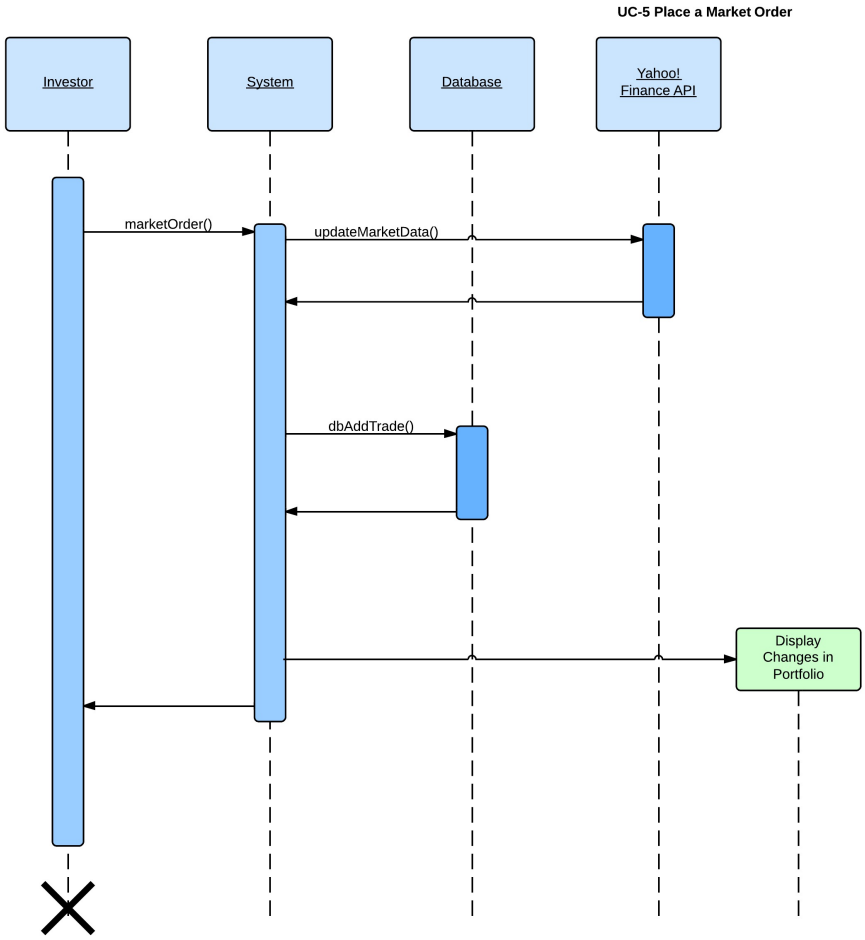
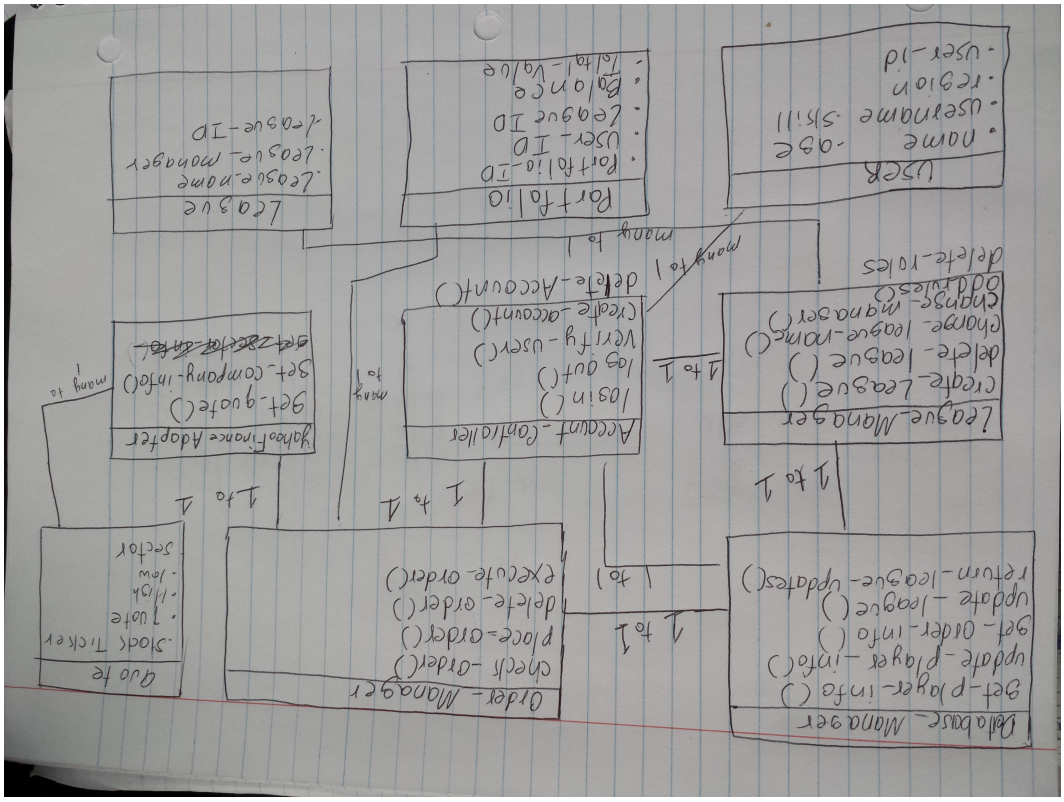


Figure 1.5: The investor needs to be able to place market orders. As soon as the investor places an order the system contacts Yahoo! Finance API to retrieve the current price of the stock. After the current price is found the system must confirm with the database that the user has enough funds to make a buy offer or enough stock to make the sell offer. After the trade is confirmed information will be stored about it in the database and the changes will be displayed in the investors portfolio.

2 Class Diagrams and Interface Specifications

2.1 Class Diagram



2.2 Class Data Types and Operation Signatures

Database Manager

Attributes

Our database manager performs the function of managing the database. This can mean anything from adding user information into the database, retrieving information from the database and updating information in the database, regardless of whether the information deals with users/accounts, leagues, orders.

Methods

+ get_player_info(in user_id : int) : class User

This method is used when information needs to be retrieved for a specific player

+ update_player_info(in user_id : int, in upd: class user) : bool

This method is used to update a players information, whether it be administrative or game related

+ get_order_info(in transaction_id : int) : class transaction

This method takes in a transaction_id and returns the information associated with that specific transaction

+ update_league(in league_id : int, in leagueInfo : class league) : bool

This method is used when a league needs to be updated with the newest information provided in the league in the input

+ return_league_updates(in league_id : int) : class league

This method returns the latest updates in the league

Order Manager

Attributes

Our order manager class is responsible for handling all the tasks related to orders/transactions. It is responsible for placing the order in the system and for moving old orders to the archive_transactions table.

Methods

+ Check_order(in symbols: class Order) : bool

This method is simple used to check and make sure that the input order can be processed. It will check the users balance, etc.

+ place_order(in symbols: class Order) : bool

As the name suggests, this method is used to place an order in the system, with the information given by the input Order class

+ delete_order(in symbols: transaction_id): bool
As the name suggests, this method deletes an order from the system, assuming that it hasnt already been processed. If it has then this function will return a false value.

+ Execute_order(in transaction_id : int) : bool
This method is responsible for actually getting the stock information from Yahoo Finance API and then changing the account/portfolios to reflect it accordingly.

League Manager

Attributes This class is responsible for managing all the leagues in the system. It has the authority to create leagues, delete leagues, and modify leagues as it is instruction to do so.

Methods

+ Create_league () : Class league
This function is used to create a league from scratch so that the user can create a league.

+ Delete_leagues(in league_id : int) : bool
As the name suggests, this method will delete the league matching the input

+ change_league_name(in league_id : int) : bool
This method is here solely for the purpose implied by its name. Its only function is to change the name of the league.

+ Change_league_manager (in league_id : int, in usr : class User) : bool
This function replaces the current league manager stored in the input league with the user specified in the input.

+ add_rules(in league_id : int) : bool
This function is here for the reason its name suggests. It is here just to add rules to a given league.

+ Delete_rules(in league_id : int) : bool
This method exists just to delete the rules in a league.

Account_Controller

Attributes
This class exists to take care of any function that relates to accounts. This can mean creating an account, modifying an account, or even deleting an account.

Methods

+ Login(in user_id : int) : bool
This function is used by the user to log into the system
+ logout(in user_id : int) : bool
This function is the opposite to the one above it, it is used by the User to log out of the system.

+ Verify_User(in User_id : int) : bool
Method to make sure that the person logging in or that the person who is logged in is not an imposter/fake.

+ Create_account() : class User
Creates an account with the current user

+ delete_account(in suser_id : int) : bool
Used to delete an account from the database/ system.

Yahoo Finanace Adapter

Attributes
This class is responsible for obtaining market data from Yahoo Finance API. It consists of 3 functions to get quotes, get company information, and to get sector information.
Methods

+ Get_quote(in stock_ticker_id : string) : class quote
As the name suggests, this method is responsible for obtaining quote information about a given stock_ticker

+ get_company_info(in stock_ticker_id : string) : class Company
This method is responsible for getting market information about a specified company.

3 System Architecture and System Design

3.1 Architectural Styles

In order to make the most efficient use of our software, we will couple several known software tools and principles into our design. The follow architecture types will be expanded in detail to not only reflect general functionality, but also to reflect functionality of the software as a whole. As explained, each will play a crucial role in the success of our software and will be largely derived from the necessities of the software. That being said, architectural systems will include (and may be expanded upon in the future) the Model View Controller, Data-Centric Design, Client-Server access, and RESTful design, with each architecture serving a small part of the whole result.

Model-View-Controller

The Model View Controller is a User Interface implementation method which will separate the software into 3 specific groups; that is: the model, view, and controller subsections. The view category is typically limited to UI specific output, i.e. a webpage with stock information. That being said, the model remains the core component of the MVC method which holds all of the data, functions, and tools. The controller simply takes the input and converts it into a command for either the model or the view.

The MVC method is ideal for this particular software because it allows the design to be broken down into smaller sub-problems. By splitting into 3 parts, we can separate UI functions, from database functions, and have all of them handled ultimately by the controller. Thus in terms of fluidity of the design, adding in the MVC allows each to be distinct and allows for the programming to be made far easier.

Data-Centric Design

Data is the fundamental backbone of Paramount investments. Stored within our database, will be numerous bouts of data, which will be necessary for all aspects of the software. The database needs to contain not only data pulled from the Yahoo! Finance API, but more importantly user specific data. Whenever the user logs in, they need to have access to a personal host of their own data. That includes but is not limited to complete portfolio, leagues, achievements, leaderboard, and settings. More importantly, the data needs to be stored in a way that it can be accessed by multiple subsystems whenever necessary. So in using this method, we can keep the data specific parts in the software abstract and easily accessible.

Client-Server Access

The user will be constantly interacting with the interface. All of the interactions are occurring, thus, on a client server basis. The user remains the primary client, and as such, constantly must interact with the other subsystems. All of the infrastructure provided by Paramount Investments will need to be accessed by the user. This ensures a smooth communication between each of the parts of the MVC and between client and infrastructure. Further, the infrastructure provided by Paramount investments will be able to access infrastructure of non-associative systems.

Representational State Transfer

As a software implementing a client server Access system, a REST system is also inherently implied. The RESTful design principles state that in addition to having a Client-Server Access system, the system has a scalability of components, that the interface is uniform, stateless, and cacheable. Using this method will employ a smooth, modular set of code. Using the interface specifications within the RESTful outline allows both the user and the designers to have streamline interactions with the interface. That is the user knows quite clearly what he or she is doing when say a link is clicked on a web page. The request is converted and sent out to the controller.

Importantly, the RESTful implementation can be implemented on multiple levels. And as is desired, this system will be able to work on Android and iOS as well as through standard web interfaces. Thus a smooth transition between these mediums is incredibly important. Thus whether a user places an order on his cell phone or online, he should be able to experience a uniform experience across all mediums. Using the RESTful system will help in this process.

3.2 Identifying Subsystems

Paramount investments aims to set its platform on multiple interfaces. As such, subsystem identification becomes an integral part of initial analysis of the software. On a thick layer, our platform exists with a front-end system and a back-end system. But on a much deeper level, we can see that, each of these subsystems can be broken down into still greater detail. Front end systems typically involve user interface, and object interactions with the user. Back-end will refer to all database schema, implementation and interactions with relevant hardware. Also included are non-associative items which are necessary to the success of our system.

Front-end systems are formally plain. The user interface which displays views and specific data to the user on multiple platform is included here. That is, it will contain different mappings and specific implementations for iOS and Android as well as natively for the Web. The front-end system will have to maintain constant communication with the back-end system to maintain consistency and retrieve data regularly. It needs to be able to successfully communicate information from commands given by the user and communicate them to the back end. The back end system will retrieve necessary data and information and return the data to the UI and user to project the page or information requested.

Our back end system will be broken down further and is easily considered the most important part of our infrastructure. Since we are using the MVC framework, the back end system is to be broken down into controller and database subsystems. Additionally, we will have the financial

retrieval system and queuing systems as previously outline. Thus, the bulk of the command processing is handled by our back-end subsystem. The back-end system must not only communicate among the subsystems within itself, but it must also communicate with the front-end UI system to respond to commands and also communicate with the non-associate systems as well.

Breaking down the subsystem further, we highlight the importance of the financial retrieval system, and the queue system. The financial retrieval system will communicate with Yahoo! Finance to retrieve relevant information as requested by the controller (whenever the controller receives an input from the front-end user). The queuing system will handle other processes and largely communication with non-associative systems. It will also be involved in queuing and handling all back-end processes and monitors to ensure that the correct commands are processed at the correct time. The success of these modules, the success of the entire back-end system, and the success of communication amongst the systems will be crucial for the overall success of the software.

3.3 Mapping Hardware to Subsystems

The Paramount Investments League is contained on a MySQL database server, which is stored on one machine. However, the system as a whole is spread across several machines. The system to be is divided into two separate sections: a front-end side that is run on the clients web browser of choice, and a back-end that runs on the server side of the database. The front-end is the main graphical user interface (GUI) between the system and the client. The front-end is responsible for communication between the GUI and the database for purposes such as confirming market orders and updating an investors portfolio. These changes in the front-end are reflected in the back-end side of the server. The back-end will handle proper execution of market orders and will updates users on each of their transactions.

3.4 Persistent Data Storage

The plan for data storage exists at the core of Paramount Investments. Since so much of our software depends on properly developed and updated data, it is of the utmost important that our database schema represent accurately all objects involved. That is, the data must accurately (at all times) reflect all relevant user data, stock information, ticker variables, league settings, achievements, leaderboards, and all other relevant objects.

Paramount Investments will make heavy use of the relational database MySQL. Relational databases are far more practical for the needs of this particular software. That is, relational databases consist of several indexed tables filled with various object attributes. As can be viewed in the class diagrams on the previous page, this is necessary for the large quantity of objects which will be present in the software. Tables will need to exist not only for user data and settings such as log in and league profiles, but also for stock and portfolio information. Further, these databases need to be constantly written and rewritten to ensure constantly updated and accurate information. Items such as leaderboards, and information which will be able to be viewed on each users portfolio need to constantly reflect accurate data.

The data will be retrieved from the respective database table in the form of a query. When a user inputs a command to retrieve data, a query must be placed, the table searched, and the eventual correct data value (or values) returned. For example if a user requests his or her settings,

it can query currently selected settings and return those values to the UI and to the user. If the user elects to make a change this will be sent back to the database, updated and saved for further access later. The same process can be mirrored and applied to all facets of the software. Several tables will be used for varying data as has been outlined in the diagrams above. The success of the software is dependent on the values being returned accurately and in the most updated form at all times. Because of that, the database must receive a regular feed from the Yahoo! Finance API in order to constantly update and reflect data when queries are placed. In doing so, users will have constantly accurate views of their portfolio performance, leaderboards, achievements, stock tickers, and recent trades going on throughout the league and entire user base. It is in this way that the Paramount Investment software will distinguish itself from others and retain functionality and efficient realization of its ultimate goals and requirements.

3.5 Network Protocol

As is standard for software of this type, Paramount Investments will uses the standard Hypertext Transfer Protocal (HTTP). HTTP acts by structuring text which is uses hyperlinks to communicate messages through text between nodes. While not necessarily unique or particular to our situation, it is still important to note that this will be the primary protocol between user and software interface. More importantly, the HTTP protocol will be used not only on web-based devices but also on Android and iOS devices as well. From any of these mediums, the users can access various webpages and links from the Paramount Investment website. They will be able to access, through this protocol, all relevant stock, portfolio, and relevant information through these pages and by using the HTTP protocol.

3.6 Global Control Flow

Execution Order

In general, the implementation of the system at Paramount Investments is for the most part, event-driven. All the features that the system has to offer must be triggered by some entity, whether it be the user themselves or some other part of the system. Overall, most of the event-driven characteristic comes from the user end of the system. Many of the functionalities (stock trading, portfolio viewing, league joining, etc..) can only be triggered by the user. There are however, some event-driven functionality that are initiated by the system. When the user places an order, it is processed and added into the database. From here, the system initiates the process of checking the order and then it uses Yahoo Finance API to retrieve market information about the stock, obtain a quote and then actually process the order.

There is some functionality that have to be executed in a defined order. Before the user starts investing, they must a few steps:

- Registration/creating an account: any user must register within our website before joining a league.
- Join a league: any user must first join a league before they can start investing.
- Achievements: any user must first complete the required criteria before they can be awarded with the achievement trophy.

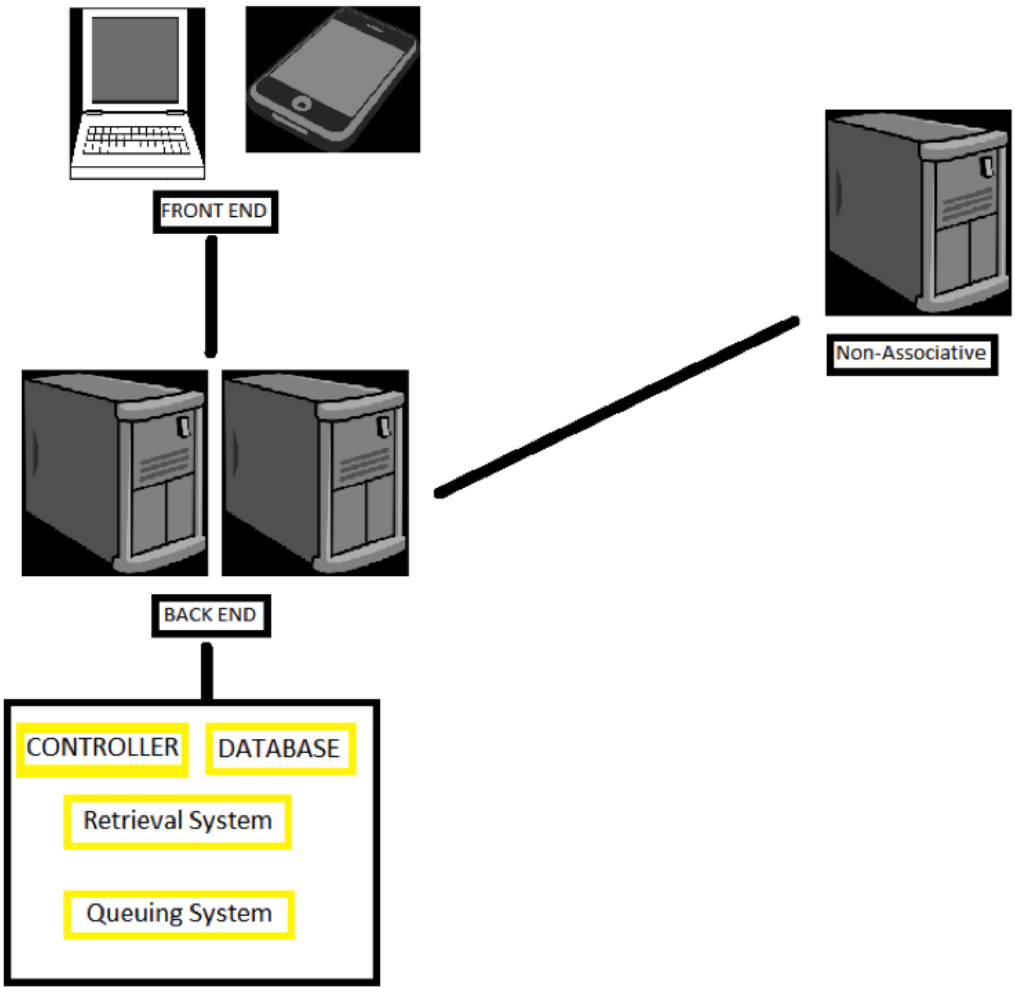


Figure 3.1: Diagram of the network protocol.

However, on the whole, our system is still definitively an event-driven one.

Time Dependency

In general, the system at Paramount Investments is very much a real-time system, but there are features that do not depend on time. The real-time system is very reliant on the stock market, which itself has certain times of operation. As the user is browsing the website, there are real-time timers that help the system process information that it is receiving.

- Achievement Timer: This timer is used at the end of the day to check for achievement specs for all the users. Achievements/ rewards will then be dished out accordingly.
- Stock Market open and close: The stock market has a time interval between when its open and when it is closed.
- Queuing system: the orders placed by users are placed into a queue. Depending on market conditions, this can place high loads on the server. To balance the server load, we must split the orders effectively. The timer in this system helps to check for unexecuted/ outstanding orders and then processes them.

Concurrency

There are sub-systems in our main system which have to be carefully thought of due to concurrency. The biggest of these is the queuing subsystem. This produces a concurrency issue because we have to make sure that no more than 1 order is being inserted into the queue at any given time. Likewise, we also have to make sure that no more than 1 order is being dequeued from a given queue. Other than this our system really does not need any synchronization. However, this may change as we are implementing our system.

3.7 Hardware Requirements

The hardware requirements on the server side are the main contribution to the operation of Paramount Investments League, leaving the client-side with minimal requirements. In fact, the only requirement of a client will that it runs a browser that is capable of running a modern web browswer.

Internet Connection

In order for Paramount Investments League to use any of its core functions (trading stocks, updating user portfolio, tracking administrative actions, etc.), an internet connection is required. Since most of the data being transferred is text (executable instructions), a low band of frequency is required. Note that a complete scalable analysis has not been performed on the system, so a low band of frequency is based off of the needs of the current website. For ideal performance, higher bandwidths of frequency should be used in order to reduce any overhead. A network connection between the server and the Yahoo Finance API is necessary during trade hours (9:30am - 4:00pm Monday through Friday), otherwise, no investors can perform a transaction.

Disk Space

The server must have adequate hard drive space to be able to store all of the database information. All data being stored is the sum of all program instructions for the system. 10 GB of storage space should be sufficient for the system.

System Memory

Since this system is in active development, there is limited concrete evidence that supports the overall performance of the system. The system will load copies of database stored information in order to operate over it. For better throughput, the memory should be managed using a Least

Recently Used scheme (LRU) in order to keep the system memory populated with useful information. A LRU scheme will release any bits of memory that havent been accessed in a long time, and it will replace it with information that is used more often. Also, any operations used on loaded information will also use up system memory. A minimum of 512 MB should be used for testing our system. In addition, as our user based expands, it is obvious that the system memory will also have to grow with it.

Client-side Hardware Requirements

The core hardware requirement on the client-side of the system will be an internet connection. This is essential for the client to be able to remotely connect to the server in order to access the database. Without an internet connection, no client will be able to use a web browser to visit the Paramount Investments League website. In addition to an internet connection, and for a friendly user experience, anyone on the client-side should have a functional mouse and keyboard, as well as a graphic display to see their portfolio. To display the Paramount Investments League website, a screen with a minimum resolution of 800x600 pixels is adequate.

4 Data Structures & Algorithms

4.1 Data Structures

In the implementation of the system at Paramount Investments League, there will be 3 main data structures in use. These 3 data structures are a Queue, ArrayList, and the HashMap.

Queue

The system at Paramount Investments league uses a queue data structure to hold all the orders/transactions that users may place during in the system during the day. Because of the FIFO (First in First out) property of the queue the orders that were placed first will the ones that are executed/processed first. This mimics the real world scenarios and will help capture part of the essence of stock market trading. At this stage in the implementation, we will be trying to accomplish this using the interface provided to use by the Queue Interface in Java. One of the requirements we require of this queue is that it be thread safe since there can be multiple users placing their orders at the same time into the same queue. If we find that there is a space limitation or lack of thread synchronization of this queue implementation, we will make an attempt to code the queue ourselves.

ArrayList

The system at Paramount Investments league will also be using an array data structure to keep track of the positions that an investor might have in a single stock. In most case, investors will have only one position per stock but there are scenarios where this is not true and the investor may have more than one position in a single stock. Because we are unsure of how many positions the investor might want, we need to be able to account for this using a data structure that has quick random reads but also has the capability to grow in size without restriction. This is achieved using the ArrayList class provided in Java. The ArrayList class has random read capability just like an array, but it also has the capability to grow indefinitely just like a linked list. Hence we will be using the ArrayList to keep track of the positions per stock.

HashMap

The system at Paramount Investments League will be using a HashMap to keep track of all the stocks that a user chooses to invest in for a given portfolio. This data structure needs to have quick insertion, delete, and read times. We chose the HashMap to accomplish this task because of the speed at which it is possible to insert, delete, and access information in a HashMap. Because there

are hundreds of different stocks, quick access to information is a necessity. We plan to accomplish this task using the HashMap class in Java.

4.2 Algorithms

At the time of this report there is only one interesting algorithm that has been designed and implemented. We expect as the project progress for this section to flesh out more and more, and include additional algorithms.

A Method For Reducing API Calls in a Highly Concurrent Environment

Our system relies on external API's[1] in order to accomplish the most central tasks, namely retrieving up-to-date stock data. Since we are using a free API, their are limits to the number of times that we can request information from the API without having our IP address[2] blocked. In order to limit the number of calls that are made, we need to cache the results on our servers.

In order to accomplish this we wrote a service that is concurrent and maintains a cache of stocks values on our server updating them periodically. Here is a brief overview of the algorithm:

Algorithm 1: Retrieve and Cache Stock Values

```
1  stockTicker ← End user does an operation that requires a stock value;
2  stock ← HashMap.get(stockTicker);
3  if stock exists then
4      return stock;
5  Synchronize;
6  if stock exists then
7      return stock;
8  stockValues ← YahooAPICall(stockTicker);
9  stock ← newStock(stockValues);
10 HashMap.put(stockTicker, stock);
11 return Stock;
```

In order to make the above algorithm work in a Concurrent environment, we synchronize[**?**]wiki:sync) it in the critical section, that is, we only allow one user at a time to add something to the HashMap.

In order to update the HashMap periodically, we run a background thread that sleeps for some defined amount of time, then runs. This background process, builds an entirely new HashMap, and once complete, replaces the out of date HashMap.

5 Design of Tests

No application is ever complete, but a big part of driving a project to a viable project is testing. Testing allows us to ensure expected functionality, check for possible security vulnerabilities, and prevent regression as the project moves forward. Attempting to launch a product without performing unit and integration testing, as well as "dog fooding"[3] an alpha version is a guarantee to have to putting out a buggy and sub par product. However, even with performing all the aforementioned, it is not possible to find and resolve every flaw before shipment. To this end, developers utilize *testing suites* in order to perform integration and unit testing in an efficient and effective manner.

A modern approach to this trade off is to build the feature set of an application around measurable, predefined tests. In this technique, known as Test-driven Development[4], developers iteratively define tests for intended future features, confirm that those features are not yet implemented (by running those tests), and then implementing the solutions. Though this approach does not test for all possible interplay between components, it is usually employed in high-paced development environments such as ours, where the coverage provided is usually respectable enough to prevent most problems.

Accordingly, we first define the features and tests we plan on developing around, proceed to analyze the coverage offered by these tests, and then briefly discuss how we intend to test the integration of the components.

5.1 Test Cases

The Paramount Investments League application is in active development, therefore, each test case specified is only applicable to existing functions during this stage of development. For the most thorough testing, we will perform unit tests on each component of the system currently in existence. The Paramount Investment League requires communication between Yahoo! Finance, our MySQL database, and our server, but unit testing these components in not efficient. Instead, we will perform integration tests on these units to see how they interact with each other.

Paramount Investments League will be using a Java/Scala Play Framework to develop our web application. The main reason for choosing Play Framework provides minimal resource consumption (CPU, memory, threads) and also supports big databases. Also, most of the team members are proficient in C++, so the transition to Java is doable.

5.2 Unit Tests

Database Manager

The tests listed below interact with our MySQL database, however they have no correlation to the implementation of the database.

1. Test Case Identifier TC-1:
   Function Tested: get player info(in user id : int) : class User
   Success/Fail Criteria A successful test is one that retrieves information about the requested player.

Test Procedure:	Expected Results
Call Function (Success)	Information requested matches the search criteria.
Call Function (Failure)	Information requested does not match the search criteria.

2. Test Case Identifier TC-2:
   Function Tested: update player info(in user id : int, in upd : class user):bool
   Success/Fail Criteria - A successful test is one that updates a player's information, whether it be an administrative action or game related.

Test Procedure:	Expected Results
Call Function (Success)	Player's profile is updated with new information. A value of true is returned
Call Function (Failure)	Player's profile is not affected after attempted update. A value of false is return

3. Test Case Identifier TC-3:
   Function Tested: get order info(in transaction id : int):class transaction
   Success/Fail Criteria - A successful test is one that returns the information associated with a specific transaction.

Test Procedure:	Expected Results
Call Function (Success)	Transaction information returned corresponds to transaction.id.
Call Function (Failure)	Transaction information isn't returned to the user.

4. Test Case Identifier TC-4:
   Function Tested: update league(in leagueInfo : class league):bool Success/Fail Criteria - This method is used when a league needs to be updated with the newest information provided.

Test Procedure:	Expected Results
Call Function (Success)	League information has been successfully updated. A value of true is returned a
Call Function (Failure)	League information has not changed from before. A value of false is returned af

5. Test Case Identifier TC-5:
   Function Tested: return league updates(in league id:int):class league Success/Fail Criteria - A successful test will return any league updates to the requested user.

Test Procedure:	Expected Results
Call Function (Success)	League updates are presented to the requesting user.
Call Function (Failure)	No data is presented to the user after function call.

Order Manager

The Order Manager is responsible for handling all tasks related to orders and transactions. The Order Manager is responsible for placing new orders in the system, as well as archiving old

transactions in a table.

1. Test Case Identifier TC-6:
   Function Tested:Check order(in symbols: class Order) : bool
   Success/Fail Criteria A successful test will return a Boolean value of true corresponding to a valid user trade requests (buy, sell short, stop etc.).

Test Procedure:	Ex
Call Function (Success)	User is able to perform a valid transacti
Call Function (Failure)	User will be notified that he/she will not be able to perform a valid transaction.

2. Test Case Identifier TC-7:
   Function Tested: place order(in symbols: class Order) : bool
   Success/Fail Criteria - A successful test will allow the user to place a market order.

Test Procedure:	Expected Results
Call Function (Success)	Market order is placed, and a confirmation is sent to user. A value of true is ret
Call Function (Failure)	Market order is not placed, and the user will be notified. A value of false is retu

3. Test Case Identifier TC-8:
   Function Tested: delete order(in symbols: transaction id): bool
   Success/Fail Criteria - For a successful test, this method should delete an order from the system, assuming that it hasnt already been processed. If it has then this function will return a false value.

Test Procedure:	Expected Results
Call Function (Success)	Market order has been deleted from the queue. A value of t
Call Function (Failure)	Market order has already been recorded, user will be notified of the invalid tran

4. Test Case Identifier TC-9:
   Function Tested: Execute order(in transaction id : int) : bool
   Success/Fail Criteria - For a successful test, the system will obtain information from Yahoo! Finance and update a users portfolio accordingly.

Test Procedure:	Expected Re
Call Function (Success)	System retrieves data and updates the users portfolio. A
Call Function (Failure)	System either does not retrieve information from database and or the users port

League Manager

This class is responsible for managing all the leagues in the system. It has the authority to create leagues, delete leagues, and modify leagues as it is instruction to do so.

1. Test Case Identifier TC-10:
   Function Tested:Create league () : Class league
   Success/Fail Criteria - A successful test is when the user can create a league from scratch.

Test Procedure:	Expected Results
Call Function (Success)	User is now the league manager, and their new league is added to the
Call Function (Failure)	No new league is recorded in the system and the user will be notified that their

2. Test Case Identifier TC-11:
   Function Tested: reaturn league updates(in league id:int):class league

Success/Fail Criteria - A successful test will delete the selected league.

Test Procedure:	Expected Results
Call Function (Success)	Selected league is deleted from the users list of league. A value of true is returne
Call Function (Failure)	League will remain in the users list of league. A value of false is returned af

3. Test Case Identifier TC-12:
Function Tested: change league name(in league id : int) : bool
Success/Fail Criteria - A successful test will update the current league name with a modified one.

Test Procedure:	Expected Results
Call Function (Success)	League name has been changed and is reflected in the database. A value of true
Call Function (Failure)	League name has remained unchanged. A value of false is returned

4. Test Case Identifier TC-13:
Function Tested: Change league manager (in league id : int, in usr : class User) : bool
Success/Fail Criteria - A successful test will change the current league manager with the new input league manager.

Test Procedure:	Expected Results
Call Function (Success)	League has a new manager, and all changes are reflected in database. A value o
Call Function (Failure)	League manager remains unchanged. A value of false is returnec

5. Test Case Identifier TC-14:
Function Tested:add rules(in league id : int) : bool
Success/Fail Criteria -A successful test will add a new rule to the list of league rules already established.

Test Procedure:	Expected Re
Call Function (Success)	The newly added rule is reflected in the database. A va
Call Function (Failure)	The new rule to be added has not been added, and the database sees no change

6. Test Case Identifier TC-15:
Function Tested: Delete rules(in league id : int) : bool
Success/Fail Criteria - A successful test will delete a rule in the leagues list of rules.

Test Procedure:	Expected Results
Call Function (Success)	The selected rule is deleted, and the database is updated of the change. A valu
Call Function (Failure)	The selected rule has not been removed and the database sees no changes. A va

Account Controller

This class exists to take care of any functions that involve any user accounts. Functions include, adding, modifying, or deleting an account.

1. Test Case Identifier TC-16:
Function Tested: Login(in user id : int) : bool
Success/Fail Criteria - A successful test will allow the user to visit their Paramount Investments League global portfolio.

Test Procedure:	Expecte
Call Function (Success)	User is logged into the system and they can view their a
Call Function (Failure)	User is not logged into the website. User may not have entered password correc

2. Test Case Identifier TC-17:
   Function Tested: logout(in user id : int) : bool
   Success/Fail Criteria - A successful test will all the user to logout of their Paramount Investments League account.

Test Procedure:	Expecte
Call Function (Success)	User is logged into the system and they can view their a
Call Function (Failure)	User is not logged into the website. User may not have entered password correc

3. Test Case Identifier TC-18:
   Function Tested: Create account() : class User
   Success/Fail Criteria - A successful test will create a new user account.

Test Procedure:	Expected Re
Call Function (Success)	A former visitor to the Paramount Investments League website will now be a
Call Function (Failure)	The request to make a new account has failed, and no new account will be refle

4. Test Case Identifier TC-19:
   Function Tested: delete account(in user id : int) : bool
   Success/Fail Criteria - A successful test will delete the selected user account.

Test Procedure:	Expected Re
Call Function (Success)	An investor chooses to delete their account, and all portfolios will be deleted
Call Function (Failure)	The selected account remains in the system, the database doesnt lose the associ

Yahoo Finance Adapter

This class is responsible for obtaining market data from Yahoo Finance API. It consists of three functions to get quotes, get company information, and to get sector information.

1. Test Case Identifier TC-20:
   Function Tested: Get quote(in stock ticker id : string) : class quote
   Success/Fail Criteria - A successful test will return the requested quote (stock) information to the user.

Test Procedure:	Expected Resul
Call Function (Success)	Quote information is presented to the user. System requests
Call Function (Failure)	Quote information request does not go through and the user is notified of the er

2. Test Case Identifier TC-21:
   Function Tested: get company info(in stock ticker id : string) : class Company
   Success/Fail Criteria - A successful test will return the company information that the user requested.

Test Procedure:	Expecte
Call Function (Success)	Company information is presented to the user. System can access company info
Call Function (Failure)	Company information is not presented to the user. System failed to ret

Database Models

Because of the data-centric design of Capital Games, protecting the integrity of the database entries is of the utmost importance. The Ruby on Rails framework has safeguards and validation for this purpose, but we still need to thoroughly unit test each of the models to ensure that only

permissible combinations of attributes are able to be entered, and that proper error handling occurs to resolve attempts at improper attribute definition.

Queueing System

Capital Games heavily relies upon the queueing system to act as a computational highway for all asynchronous tasks. Due to the nature of this system we must prepare for race conditions; the different ways our data can be effected based upon the order of executing processes that are acting upon the queue. We will need to prepare a set of tests to express how the queue performs when open orders are altered by other processes during different phases of the queueing system. Based on our test results it might be necessary to implement data locking.

Finance Adapter

Whenever using external resources it is vital to understand the different ways in which they communicate not just when functioning as expected, but also when failing to perform properly. Since we do not have the ability to shut down the external Financial Adapter's Servers we can not run tests that give us feedback on what functionality to expect on failure. This leaves us without the ability to test the Financial Adapter and instead pro-actively safeguard against failure. Due to this we must build a wrapper that anticipates all perceivable failures coming from the Financial Adapter.

5.3 Test Coverage

In order to attain full functionality of Capital Games without bugs, we must be sure that none of its parts have errors themselves. Due to many dependencies such as other running processes, system states, and transitions, the same test will need to be preformed for each possible configuration to make sure that each part works in every possible scenario that it can be ran. This will require extending certain tests to run at the same time as background processes, and having parts called from all possible initiating parts. When working with integrated parts it is not simply enough to assume that parts will work once integrated just because they work independently. By extensively testing each possible run case we ensure that there are no points of failure once the system is launched.

5.4 Integration Testing

In order to achieve the most thorough testing, Capital Games will be tested using the bottom-up strategy. Each part of Capital Games that we wrote will be extensively tested individually first. However simply testing each part individually is not enough due to race conditions and other integration issues that may exist in the systems described above. Because of this, parts must be tested after integration as well. Knowing that functionality is state specific and transition specific for any state machine, each test must also be ran in all possible states. In addition to all previously listed conditions, tests need to be preformed at different times to make sure that functionality during backend asynchronous tasks do not have any bugs. We have chosen the bottom-up testing strategy based on the principle that bugs at the bottom level will dictate bugs at the top level, while bugs at the top level may very well be independent of bottom level performance. By carefully analyzing every part to part integration we can work our way up to a flawless design.

6 Plan of Work

6.1 Development and Report Milestones

Illustrated on the next page is a chart reflecting our goals relative to the project dead-lines. It incorporates both core development and report items. For our initial stages we focus on environment and platform set-up (eg: deploying a development webserver) and the initial, core code implementation. At the same time we will finalize the details of our final product via the report milestones.

Development milestones have been spread out following the completion of Report 1 on 23 February 2014. It begins with deploying our development environment and server through Digital Ocean[5]. We concurrently will roll out developer images, the Play Framework[6], and develop database schema. Implementing user registration/login will follow shortly along with deploying a solution to use the Yahoo! Finance API. The development milestone finishes up with the implementation of user portfolios along with basic market operations and basic achievements.

Report milestones are also set concurrently. As we begin to initialize our development environment, we will also build on top of and expand on previous reports to expand upon and fully realize the details of *P*aramount Investment League.

Core goals leading up to Demo 1 include establishing all core functionality for *P*aramount Investment League. This includes the following:

- **Play Framework deployment :** This includes basic site navigation, user login/registration, and Twitter Bootstrap deployment.
- **Setting a foundation for the database:** Schema should be built to be extensible to support future enhancements.
- **Implement the Yahoo! Finance API**
- **A functional user interface:** The user interface should function across multiple platforms with a focus on experience and expectations.

6.2 Breakdown of Responsibilities Introduciton

Contributions leading up to the completion of this report are covered in the "Contributions" in Chapter 7. For the future division of labor, we all plan on subdividing aspects of both the next reports as well as the development of the *P*aramount Investment League Demo 1.

6.3 Breakdown of Responsibilities

Core server deployment will be the repsonsibility of David Patrzeba. Eric Jacob will be responsible for the database rollout. David Patrzeba will also be responsible for the core software rollout on the server including git, Play Framework, nginx, and other core libraries and software. David Karivalis will be repsonsible for integrating Twitter Bootstrap into Play Framework.

Routing will be headed by Eric Jacob and assisted by Chris Mancuso and Evan Arbeitman.

User Interface will be done by David Karivalis and Jesse Ziegler and they will integrate the REST API[7] to facilitate dynamic views.

The rest of the development workload will be divied up based around the Model, View, Controller design pattern. David Patrzeba and Eric Jacob will focus on the controllers, David Karivalis and Jesse Ziegler will focus on the Views, and Evan Arbeitman and Chris Mancuso will focus on models. David P., David K., and Eric will be made available for technical advising.

David Patrzeba will be responsible for formatting the report. David Karivalis will be responsible for digitization of paper diagramming for all reports. Report duties will be divied up based on percieved strengths of the team and availability.

Overall project success will be decided with how well the MVC[8] component teams communicate and work with each other, as *P*aramount Investment League will rely on the interactivity between the Model, Views, and Controller portions of the architecture.

6.4 Projected Milestones

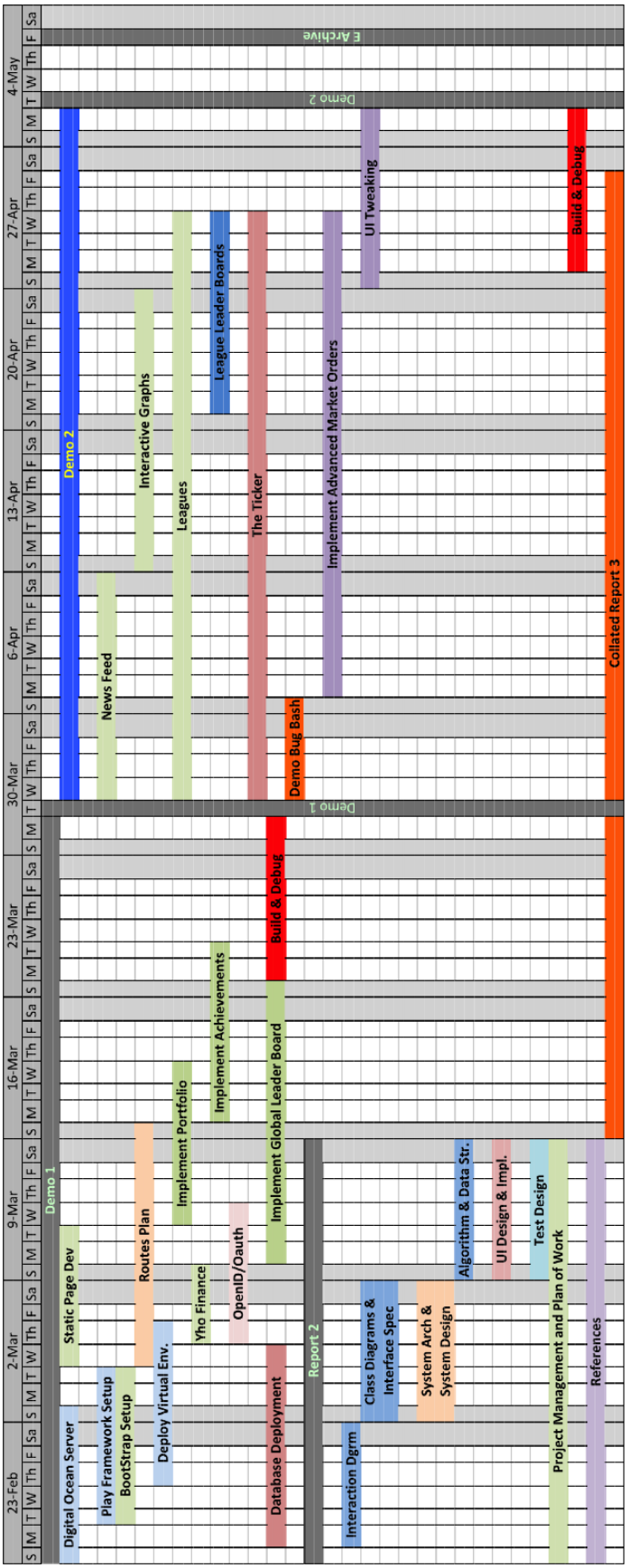


Figure 6.1: This chart is the roadmap to meeting all our milestones.

References

[1] Wikipedia, "Application programming interface." http://en.wikipedia.org/wiki/Application_programming_interface. [Online; accessed 19 March 2014].

[2] Wikipedia, "Ip address." http://en.wikipedia.org/wiki/Ip_address. [Online; accessed 19 March 2014].

[3] Wikipedia, "Eating your own dog food." http://en.wikipedia.org/wiki/Eating_your_own_dog_food. [Online; accessed 15 March 2014].

[4] Wikipedia, "Test-driven development." http://en.wikipedia.org/wiki/Test-driven_development. [Online; accessed 16 March 2014].

[5] Wikipedia, "Digital ocean." http://en.wikipedia.org/wiki/DigitalOcean. [Online; accessed 23 February 2014].

[6] Wikipedia, "Play framework." http://en.wikipedia.org/wiki/Play_Framework. [Online; accessed 23 February 2014].

[7] Wikipedia, "Representational state transfer - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Representational_state_transfer. [Online; accessed 23 February 2013].

[8] Wikipedia, "Model-view-controller - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Model-view-controller. [Online; accessed 23 February 2014].