AgriCom Training

Report #2

CSCI 441 - Team B

David Gladden, Christopher Katz, David Schiffer, Calvin Ku, Alexis Angel

Contents

Contents

1. Analysis and Domain Modeling

1.1. Conceptual Model. 3
1.2. System Operation Contracts. 8
1.3. Data Model and Persistent Storage 9
1.4. Mathematical Model 12

2. Interaction Diagrams 13

3. Class Diagram and Interface Specification

3.1. Class Diagram 18
3.2. Data Types and Operation Signatures. 19
3.3. Traceability Matrix. 21

4. Algorithms and Data Structures

4.1. Algorithms. 22
4.2. Data Structures. 22

5. User Interface Design and Implementation 23

6. Design of Tests

6.1. Test Cases 24
6.2. Test Coverage 29
6.3. Integration Testing 29

7. Project Management and Plan of Work

7.1. Merging Contributions from Team Members 30
7.2. Project Coordination and Project Report 30
7.3. Plan of Work 30
7.4. Breakdown of Responsibilities 32

8. Report 2 Contributions 34

References 35

1 Analysis and Domain Modeling

1.1 Conceptual Model

Concept Definitions

The Domain Model Concepts are derived from the elaborated Use Cases described in Report #1.

Responsibility	Type	Concept
R1: Let new user create a trainee account to partake in exercises for stock trading training. Login existing user.	D	Account Controller
R2: Initialize new accounts with a fixed balance.	D	Account Controller
R3: View the user's portfolio and display info about stocks, trades, and balance.	K	User Controller
R4: Retrieve information about stocks, trades, and commodities.	K	Commodity API
R5: Buy/Sell Stock trades and record into the database.	D	System Controller
R6: Display Home screen to create an account or login	K	Home View
R7: Display end of training feedback to user	K	User Controller

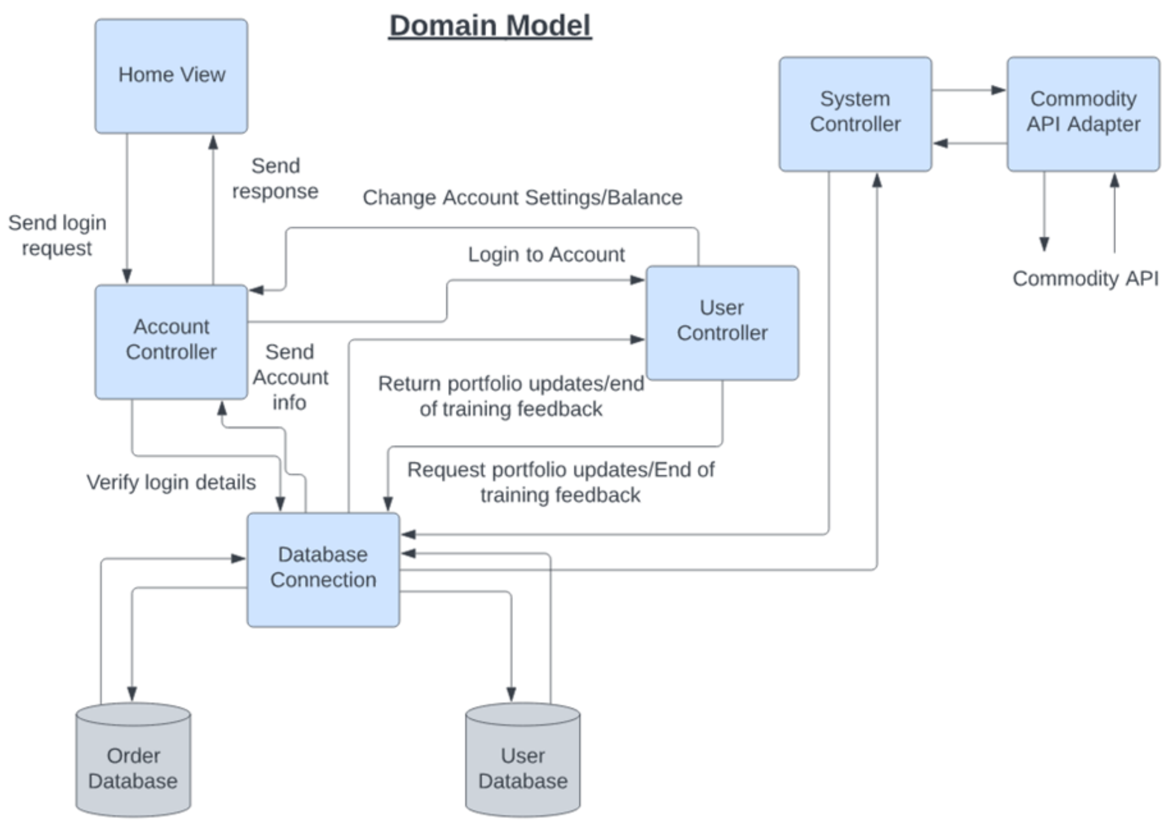


Figure 1.1 The Domain Model

Account Controller

In order to access this system at all, one has to make an account. Account creation will happen on the login screen and will check to see if an account has already been made with those credentials and if there if not it will proceed to make either a trainee or manager account depending on the preference. These details are then stored into the database.

User Controller

The User Controller accesses users' portfolios and displays relevant information such as stock, balance, and trade history. User Controller should also pull end of training feedback once training has been completed by a trainee. It will query the Commodity API to pull relevant information about any searched stocks as needed.

Commodity API

The Commodity API provides the almost real time stock data that AgriCom Training is dependent on for training new stock market users. We will have to access the market API through an adaptor that serves as a translator between our server and the API. If we did not have access to this data, the stock market training would not be realistic as it would not have real time information.

System Controller

Any order that is placed will have to go through the System Controller as it will have to record the order and input it into the database so we have a history to pull later and are able to make calculations based on account balances. This system will also have to communicate with the Commodity API as it will require current stock prices in order to record accurate account balances and if the trade is allowed under how much funds the account has.

Home View

The Home View displays a UI that allows the user to create an account or login with existing credentials. This will send a request to the Account Controller to pull or add to the database and if it fails it will reflect this on the Home View or if it succeeds the user will proceed to the Portfolio.

Association Definitions

Concept Pair	**Association Description**	**Association Name**
Home View <> Account Controller	Home View sends a login request. Account controller responds with success or failure.	Sends
Account Controller <> Database Connection	Account Controller sends user login details. Database sends account information or login failure.	Sends
Account Controller <> User Controller	Account controller shows that user logs into account and is able to access information and make changes. User controller makes those changes and sends them back to the Account Controller.	Updates

User Controller <> Database Connection	User Controller requests portfolio updates from database or end of training review. Database sends required information over to User controller.	Sends, Updates
System Controller <> Database Connection	System Controller Buy/Sell stocks and send information to Database for storage.	Sends, Updates
System Controller <> Commodity API	System Controller sends requests to API about stock prices. API returns stock information.	Sends

The associations of domain concepts are derived from the table above. First, the Account Controller takes the information that a user enters in the Home View and verifies it with the Database to see if the login is valid. Once that is validated, the Account Controller can change the view over to the Portfolio based on information that is stored in the database. The User Controller allows them to navigate the site and request information located on the site and updates from the database as needed. When a user makes a trade, the System Controller steps in and Buy/Sells stocks and sends that information to the database for storage while also pulling current stock information and prices from the Commodity API for use. The System Controller must see if a user has sufficient funds however for a Buy/Sell action.

Attribute Definitions

Responsibility	Attribute	Concept
R9: Know if a user login failed	LoginFailed	Home View
R10: Know if a user is logged in	isLoggedIn	Portfolio
R11: User Name	userName	Portfolio
R12: Account Balances	Account Summary	Portfolio

R13: Stock Bought	buyStock	System Controller
R14: Stock Sold	sellStock	System Controller
R15: Manager Account	isManager	Account Controller
R16: Trainee Account	isTrainee	Account Controller
R17: Reached end of training	endOfTraining	User Controller

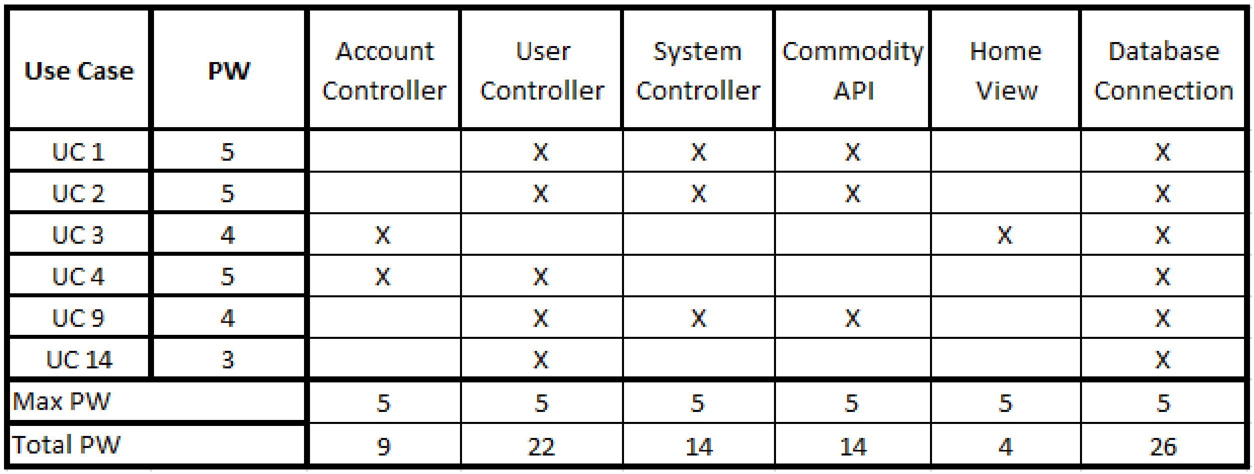
At the Home View it is very important to know if a login attempt failed as we can notify the user that way. When the login attempt is successful it will set the attribute of isLoggedIn in the Account Controller which then interacts with the database to pull all relevant user information to populate the user Portfolio.

At the Portfolio view a user may attempt things such as making a buy/sell order, view account history, or view account information. The Account Summary will show the user balances on the Portfolio view and the System Controller will store previous stocks bought or sold for the database to be able to pull here for the history.

Once the user has made it this far into the system they are ready to make an order. The System Controller will interact with the Commodity API and retrieve data for a particular stock such as its current price or price history and then report it back to the System Controller so it can perform the action of Buy/Sell on the stock.

The Database Connection also should have a retrieveData attribute to be able to read data from the database. There should also be the inverse of that attribute so that it can store data about the user and their profiles such as writeData.

Traceability Matrix



Use Case	PW	Account Controller	User Controller	System Controller	Commodity API	Home View	Database Connection
UC 1	5		X	X	X		X
UC 2	5		X	X	X		X
UC 3	4	X				X	X
UC 4	5	X	X				X
UC 9	4		X	X	X		X
UC 14	3		X				X
Max PW		5	5	5	5	5	5
Total PW		9	22	14	14	4	26

Figure 1.2 Traceability Matrix

1.2 System Operation Contracts

UC-1 Buy Commodity	
Preconditions:	• Account Controller's LoginStatus = Trainee Logged In • Page Renderer's ViewType = Commodity View • Order System Controller's OrderType = Buy • Database Connection's IsConnected = Success • Commodity API Handler's RetrievalStatus = Success
Postconditions:	• Order System Controller's ValidOrder = Success/Fail • Database is updated if ValidOrder = Success.

UC-2 Sell Commodity	
Preconditions:	• Account Controller's LoginStatus = Trainee Logged In • Page Renderer's ViewType = Commodity View • Order System Controller's OrderType = Sell • Database Connection's IsConnected = Success • Commodity API Handler's RetrievalStatus = Success
Postconditions:	• Order System Controller's ValidOrder = Success/Fail • Database is updated if ValidOrder = Success.

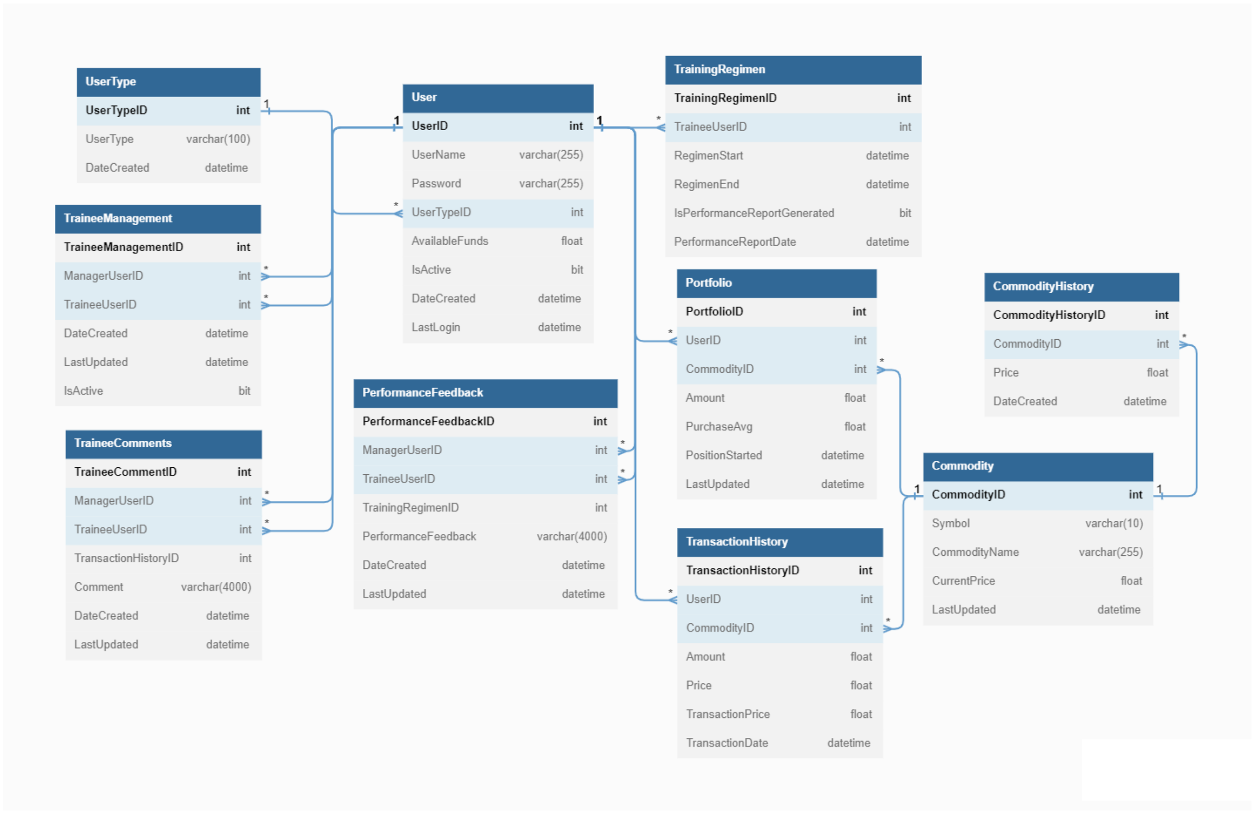
UC-3 Register as Trainee	
Preconditions:	● An unregistered Trainee (Guest) visits the system. ● Account Controller's LoginStatus = NULL or Logged Out ● Page Renderer's ViewType = Login View ● Database Connection's IsConnected = Success
Postconditions:	● Account Controller's AccountCreationStatus = Success/Fail ● Database is updated if AccountCreationStatus = Success.

UC-4 View Portfolio	
Preconditions:	● Account Controller's LoginStatus = Trainee Logged In ● Database Connection's IsConnected = Success ● Commodity API Handler's RetrievalStatus = Success
Postconditions:	● Account information is retrieved from the Database and updated with information from Commodity API Handler. ● Page Renderer's ViewType = Trainee Profile View

UC-14 Provide end of training Feedback	
Preconditions:	● Account Controller's LoginStatus = Manager Logged In ● Database Connection's IsConnected = Success ● Database's ManagedTraineeID is not NULL ● Database's Trainee account's TimerTask is expired
Postconditions:	● Database updates comments for the trainee to review

1.3 Data Model and Persistent Storage

Data is a fundamental component of the AgriCom Training System and therefore will utilize a series of SQL relational database tables to persist a majority of the required user and trading data. Currently this data will exist in set 10 different structured SQL tables. Each table can be used for multiple portions of the training system as detailed below. The relational data diagram below details the different tables and includes the field names and data types:



There are 2 central tables within this database schema that will be utilized as the core of the data system. These main tables are the User and Commodity tables. The User table will house all of the registered user information including their username , password, user type, current available funds, the date the user was created, the last login date, and if they are currently active. The Commodity table will store the various commodities that can be traded. This includes the symbol, name, current listed price, and last updated date. When the commodity API is called to pull the current price information the Commodity table is where this data will be stored.

The remaining tables outside of the central tables above are the satellite tables that store a variety of different data from configurations, user relationships, comment information, and history data. The UserType table will store the different types of user accounts to be associated with the User table. Currently the types planned at this time are trainee and manager but having a lookup table will allow many other user types to be created without impacting the established schemas.

Each user will have their current account holdings stored in the Portfolio table. This table will maintain the commodity id, amount currently held, the purchase average, data the position was started, and the last updated date for the commodity in their portfolio. This table, along with the Commodity and User table, will also be used to calculate a given user’s current account balance from the product of each held commodities amount * current price in addition to the users' current available funds.

TrainingRegimen will store the trainee users' start and end date for their monthly training regimen. This also includes a flag for if their performance report has been generated for the given regimen as well as the date this report has been generated.

The TraineeManagement table will store the relationships between trainee users and manager users. As managers add trainees to their list of users to manage, this table will be updated with their manager user id, the trainee user id, the date the relationship was established, if the relationship is currently active, and the date the relationship was last updated. This will allow the system to retain previous manager to trainee relationships even if a trainee is no longer being monitored by a manager user going forward. This setup also allows for multiple manager accounts to be monitoring the same trainee accounts rather than only allowing one trainee to one manager relationship.

There are two tables created for the storage of comments and feedback at both a transactional and performance level. The TraineeComments table will hold comments from either manager and/or trainee accounts that are created for transactions that the trainee account has made. TransactionHistoryID is included in this table to maintain the relation to the TransactionHistory table. The PerformanceFeedback table, however, will store managerial comments for a trainee at the end of their monthly training regimen. TrainingRegimenID is included in this table to maintain the relationship back to the TrainingRegimen table.

As far as history tables are concerned, there are two different tables created for retaining historical information, CommodityHistory and TransactionHistory. CommodityHistory will contain previous commodity price information so that as new prices are pulled into the system the previously retrieved values will be stored. TransactionHistory will include all the transactions for a given user. Every transaction that a user places will be stored in the TransactionHistory table and will include the commodity id, amount, transaction price, and transaction date.

1.4 Mathematical Model

Commodity Prices

There are no complicated mathematical models behind how the commodity prices are determined on our platform. The market prices that are retrieved from Commodity API Handler are the current price in the given update timeframe allowed by the API provider.

End-of-Training Regiment Report

The report at the end of a training regiment has two mathematical models. There are no complicated algorithms behind how the report is calculated. The report will provide a simple statistical summary of the Trainee's account and transactions, such as percentage gain or loss, average account balance, and average gain or loss per transaction.

Percentage gain or loss

$$(End\,Total\ -\ Start\,Total) \div Start\,Total$$

Average account balance

$$\sum_{i=1}^{Number\ of\ Days} \div\ Number\ of\ Days$$

Average gain or loss per transaction

$$(End\,Total\ -\ Start\,Total) \div Number\ of\ Trades$$

2 Interaction Diagrams

Use Case 1

The workflow for how to buy a commodity is shown in the diagram for UC-1. A trainee that is logged into the system can click the "Buy Commodity" link which will call the System Controller to prompt the trainee for the commodity and amount wished to be purchased. The trainee will then select a valid commodity and enter in an amount which is returned to the System Controller and the system then calls the Commodity API to query the current market price and returns the total cost so that the trainee can confirm the order. Once confirmed this information will be saved into the database via the DB connection and a notification is returned to the trainee from the System Controller that the transaction has been completed.

Not shown are workflows for two alternate scenarios which cause an error notification to return to the trainee. In alternate scenario 1 the trainee attempts to enter an invalid commodity name or symbol and the system returns a notification that the name or symbol is invalid. In alternate scenario 2 the trainee attempts to complete a purchase using more funds than the trainee currently has in their portfolio. This again is returned with a system notification alerting the trainee that there is not enough funds to complete the purchase.

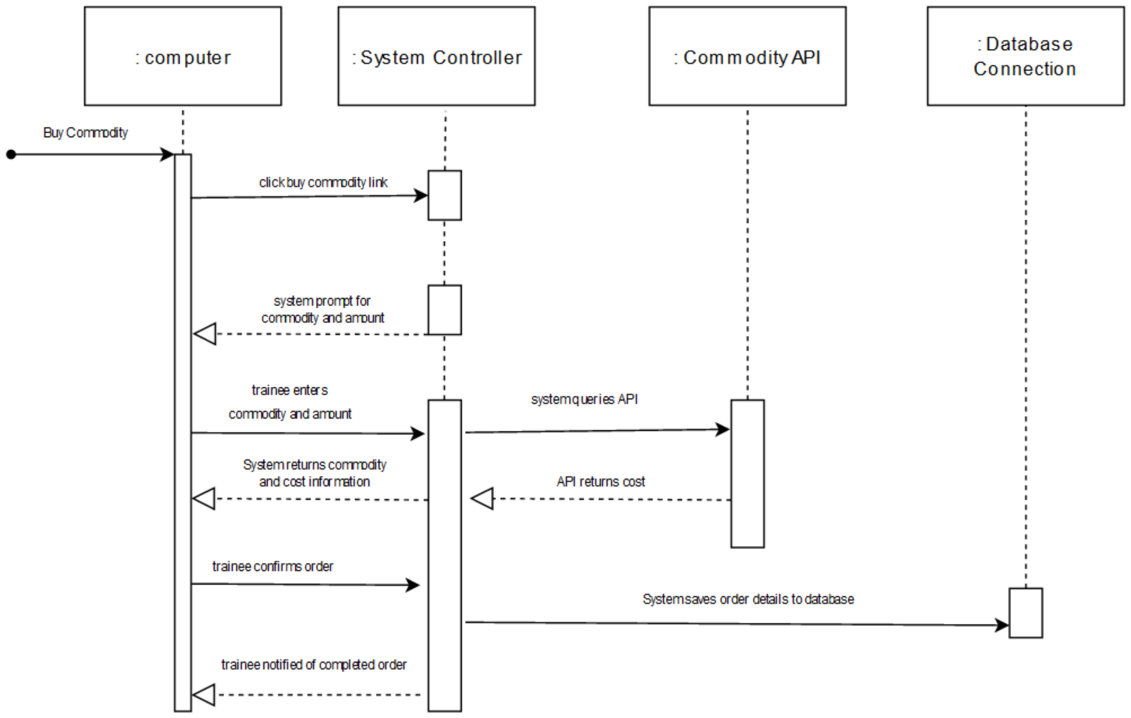

: computer
: System Controller
: Commodity API
: Database Connection
Buy Commodity
click buy commodity link
system prompt for commodity and amount
trainee enters commodity and amount
system queries API
System returns commodity and cost information
API returns cost
trainee confirms order
System saves order details to database
trainee notified of completed order

Figure 2.1: Use Case 1 Diagram

Use Case 2

The workflow for how to sell a commodity is shown in the diagram for UC-2. A trainee that is logged into the system can click the "Sell Commodity" link which will call the System Controller to prompt the trainee for the Commodity and amount wished to be sold. The trainee will then select a valid commodity and enter in an amount wished to be sold. This is returned to the System Controller and the system then calls the Commodity API to query the current market price. The projected total return from the sale is returned so that the trainee can confirm the order. Once confirmed this information will be saved into the database via DB Connection and a notification is returned to the trainee from the System Controller that the transaction has been completed.

Not shown are workflows for two alternate scenarios which cause an error notification to return to the trainee. In alternate scenario 1 the trainee attempts to enter an invalid commodity name or symbol and the system returns a notification that the name or symbol is invalid. In alternate scenario 2 the trainee attempts to sell a larger amount of shares than the trainee currently has in their portfolio. This again is returned with a system notification alerting the trainee to enter in an available amount of shares to be sold.

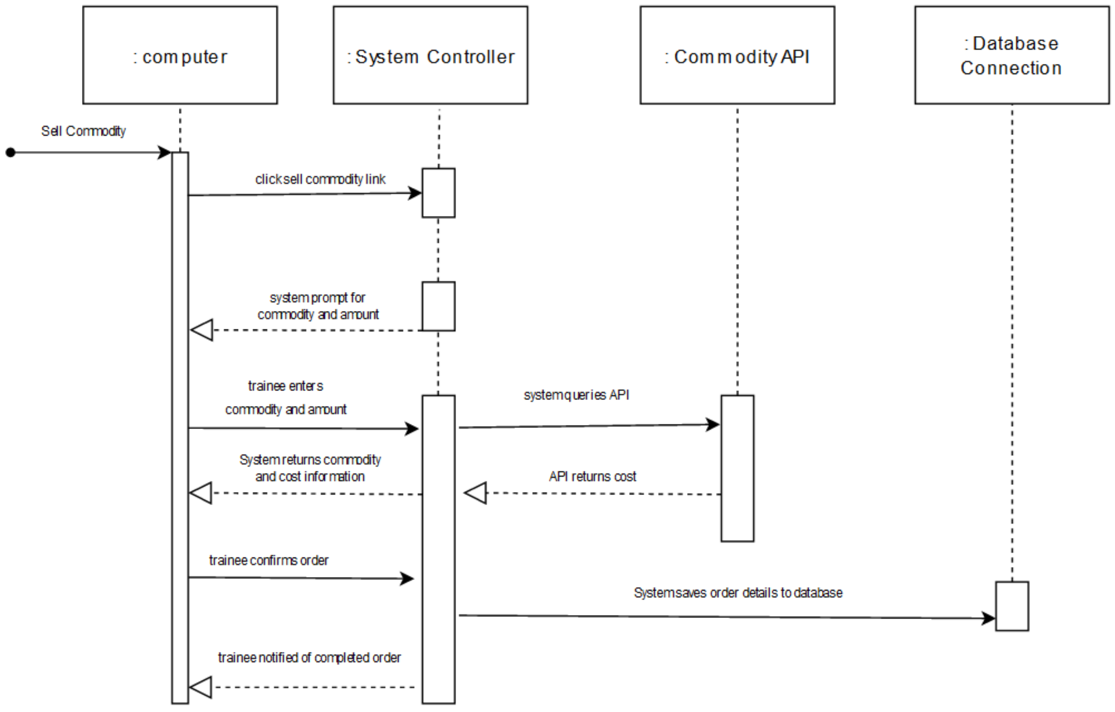

: computer
: System Controller
: Commodity API
: Database Connection
Sell Commodity
click sell commodity link
system prompt for commodity and amount
trainee enters commodity and amount
system queries API
System returns commodity and cost information
API returns cost
trainee confirms order
System saves order details to database
trainee notified of completed order

Figure 2.2: Use Case 2 Diagram

Use Case 3

The workflow for how to register as a trainee is shown in the diagram for UC-3. As a guest within the system, the guest user will click on the "Register" link which calls the System Controller and a prompt is returned to the guest for their user information. The guest then enters the information into the Trainee registration form and, once submitted, the System Controller will verify that the user name entered is unique via DB Connection. If unique, the System Controller will save the new user into the database via DB Connection and provide a notification to the now registered trainee that the account was successfully created.

Not shown in the workflow is an alternate scenario which causes an error notification to return to the guest attempting to register. In alternate scenario 1 the guest attempts to create a trainee account with a user name that already exists in the database. The system then returns a notification for the guest to attempt to register with a different user name.

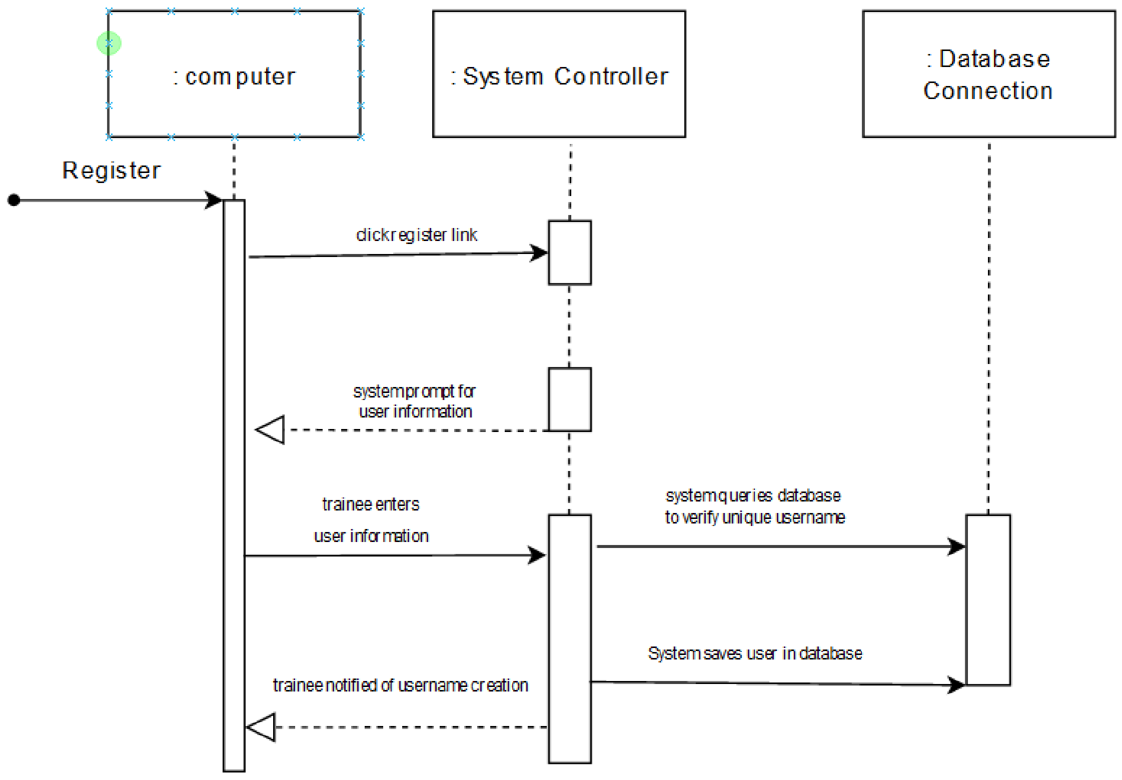

: computer
: System Controller
: Database Connection
Register
click register link
system prompt for user information
trainee enters user information
system queries database to verify unique username
System saves user in database
trainee notified of username creation

Figure 2.3: Use Case 3 Diagram

Use Case 4

The workflow for how to view a portfolio is shown in the diagram for UC-4. A trainee that is logged into the system can click on the "View Portfolio" link. If the user is a manager then they will first select the managed trainee account they wish to view before clicking on the "View Portfolio" link. In both scenarios the “View Portfolio” link will call the System Controller which will then call the Commodity API to query the most recent commodity prices. This is returned to the System Controller which stores this information in the database via DB Connection. Afterwards the trainee, or selected trainee, account information will be pulled from the database and the System Controller will display the current portfolio information back to the trainee or manager user.

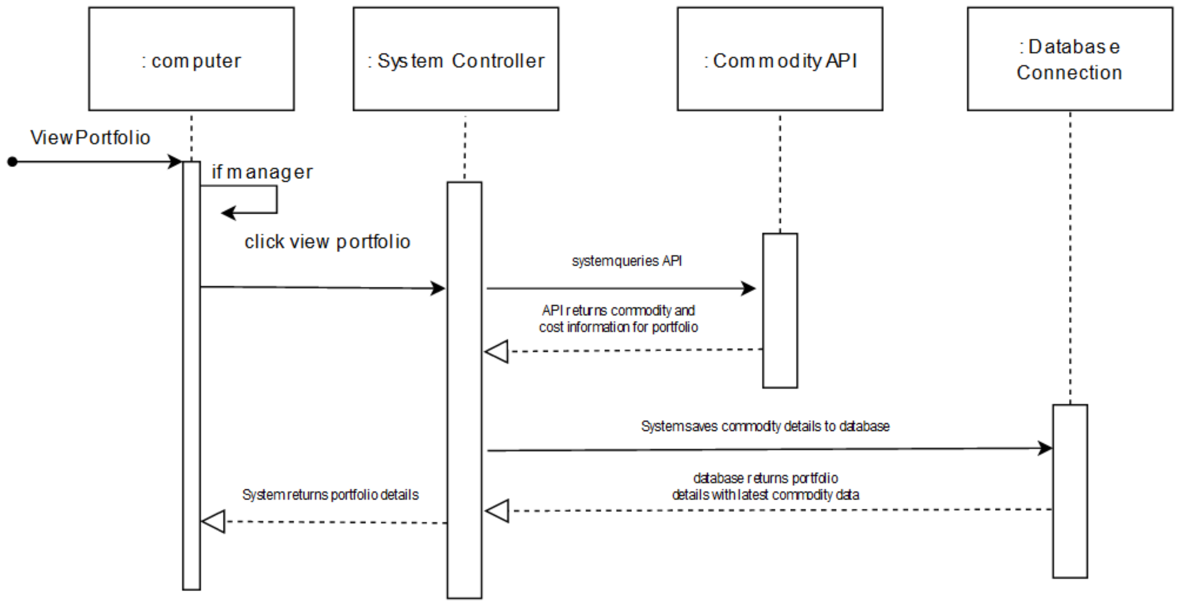

: computer
: System Controller
: Commodity API
: Database Connection
ViewPortfolio
if manager
click view portfolio
systemqueries API
API returns commodity and cost information for portfolio
Systemsaves commodty details to database
database returns portfolio details with latest commodity data
Systemreturns portfolio details

Figure 2.4: Use Case 4 Diagram

Use Case 14

The workflow for how end of training feedback is provided is shown in the diagram for UC-14. A manager that is logged into the system will select a trainee account from their management list and click on the "View Performance Report" link. The System Controller will then query for the selected account's portfolio and historical transactions from the database via DB Connection which is returned back to the System Controller which then returns the performance report information back to the manager user. After reviewing this information, the manager is then able to click the "Provider Feedback" link and the System Controller will prompt for feedback on the trainee accounts performance. The manager user will enter in their feedback and click "Submit" which calls the System Controller to save the feedback into the database via DB Connection and then returns a system notification to the manager that the feedback has been successfully saved.

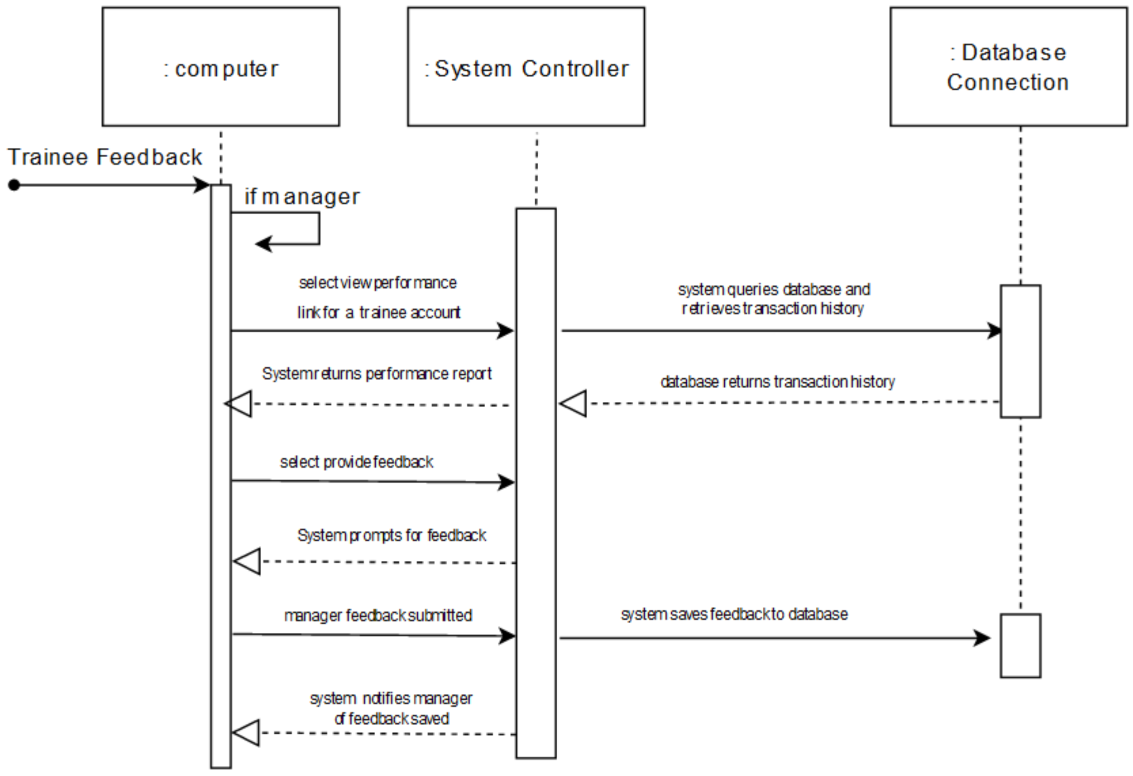
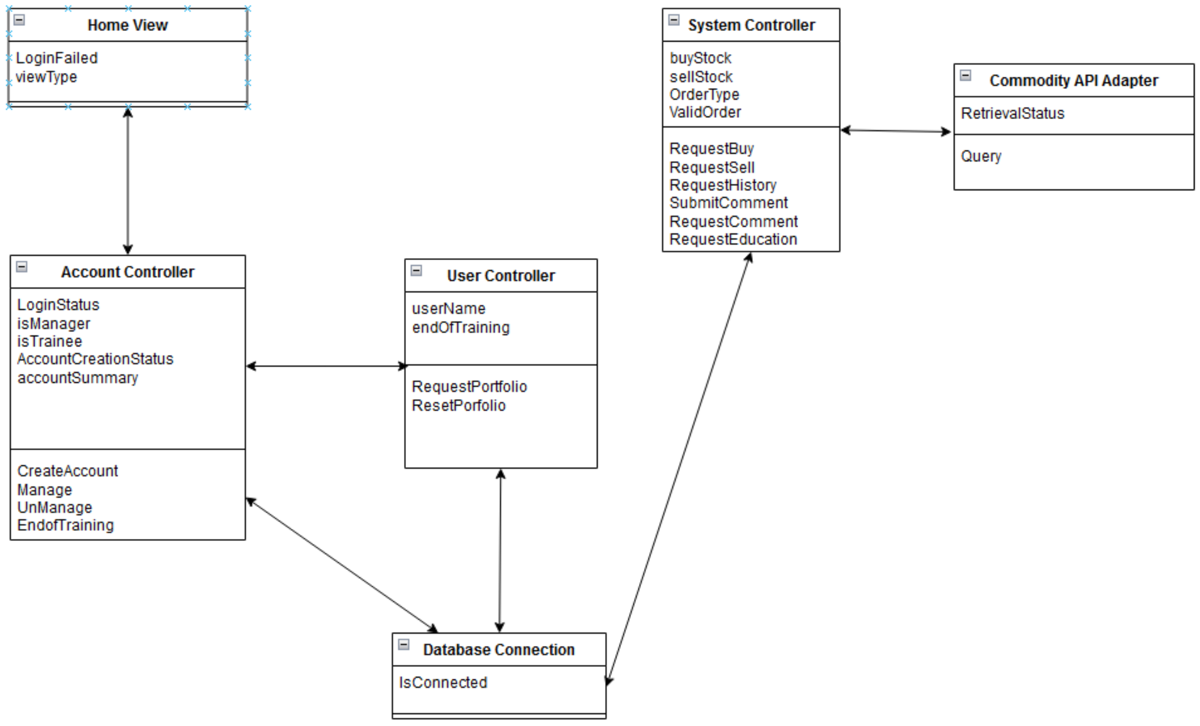

: computer
: System Controller
: Database Connection
Trainee Feedback
if manager
select view performance link for a trainee account
system queries database and retrieves transaction history
Systemreturns performance report
database returns transaction history
select provide feedback
System prompts for feedback
manager feedback submitted
system saves feedback to database
system notifies manager of feedback saved

Figure 2.5: Use Case 14 Diagram

3 Class Diagram and Interface Specification

3.1 Class Diagram



3.2 Data Types and Operation Signatures

- **Home View**
  - **Attributes**
    - **LoginFailed : Boolean**
      - **Lets the user know if a login attempt failed**
    - **viewType : string**
      - **tells the page renderer which view to display choosing between the login screen, commodity screen, management screen, or trainee profile.**
- **Account Controller**
  - **Attributes**
    - **LoginStatus : Boolean**
      - **Holds the value if an account is currently logged in or not**

        - **isManager : Boolean**
            - **Shows if an account is a manager account**
        - **isTrainee : Boolean**
            - **Shows if an account is a trainee account**
        - **AccountCreationStatus : Boolean**
            - **Holds the information if an account was successfully created**
        - **accountSummary : string**
            - **Holds the information regarding the account such as account balances and purchase/sales history**
    - **Operations**
        - **CreateAccount(UserInfo: userinfo) : Boolean**
            - **This function creates a new account with user info and passes a success/fail**
        - **Manage(String: traineeID): Void**
            - **This function will add a trainee to a specified manager's account**
        - **UnManage(String: traineeID): Void**
            - **This function will remove a trainee from a specified manager's account**
        - **EndofTraining(String: traineeID): Void**
            - **Checks in the database of users to find accounts that have expired training sessions.**
        -
- **User Controller**
    - **Attributes**
        - **username : string**
            - **holds the username**
        - **endOfTraining : Boolean**
            - **holds the value if the user has a current training session or not**
    - **Operations**
        - **RequestPortfolio(String: Investor):Portfolio**
            - **This function retrieves the portfolio data for a user from the database and sends a success/fail signal**
        - **ResetPortfolio(String: traineeID): Boolean**
            - **This function will reset the portfolio data for a user in the database and sends a success/fail signal**
- **System Controller**
    - **Attributes**
        - **buyStock : Integer**
            - **holds the value of how much of a stock the user wants to buy**
        - **sellStock : Integer**
            - **holds the value of how much of a stock the user wants to sell**
        - **OrderType : String**

        - **Holds the value of either Buy or sell**
      - **ValidOrder : Boolean**
        - **Holds the value of true/false if the order satisfies all requirements in order to be bought or sold**
  - **Operations**
    - **RequestBuy(Ticket : ticket) : Void**
      - **This function allows the user to execute a buy request of stock**
    - **RequestSell(Ticket : ticket) : Void**
      - **This function allows the user to execute a sell request of stock**
    - **RequestHistory(String : investor) : Void**
      - **This function allows the user to execute a request for the history of the user portfolio**
    - **SubmitComment(TransactionID: transactionID, String: comment): Boolean**
      - **This function allows the user to enter a comment for a transaction**
    - **RequestComment(TransactionID: transactionID) : Void**
      - **This function allows the user to request the entered comments for a transaction**
    - **RequestEducation() : Void**
      - **This function allows the user to request education for a trainee**
- **Database Connection**
  - **Attributes**
    - **IsConnected : Boolean**
      - **This value holds the success/fail value if the database connected properly**
- **Commodity API Adapter**
  - **Attributes**
    - **RetrievalStatus**
      - **This value holds the success/fail value if the API connected properly**
  - **Operations**
    - **Query(String: commodity):CommodityData**
      - **This function retrieves the market commodities data from the internet**

3.3 Traceability Matrix

REQ	**PW**	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11	UC12	UC13	UC14
REQ-1	5			X							X				
REQ-2	5	X	X												
REQ-3	5	X	X												
REQ-4	5	X	X	X	X	X	X	X	X	X	X	X	X	X	X
REQ-5	5	X	X		X				X	X					
REQ-6	5	X	X	X	X	X	X	X				X	X		X
REQ-7	4											X	X	X	X
REQ-8	4														X
REQ-9	3													X	
REQ-10	2						X	X							
REQ-11	1								X						
Max PW		5	5	5	5	5	5	5	5	5	5	5	5	5	5
Total PW		25	25	15	15	10	12	12	11	10	10	14	14	12	18

4 Algorithms and Data Structures

4.1 Algorithms

Most of the functions of AgriCom take user inputs and return outputs with minimal data manipulation other than page rendering. Commodity prices will be obtained through Commodity API Handler and each request will obtain a historic range of data to be able to provide enough information for displaying a graph. A relational database will be used for persistent data storage. Most of the data in the system will be entered into the database and so algorithms for manipulating the data, such as sorting and searching, are handled by the database. Thus search and sorting algorithms are not in the scope of the system and will not be discussed.

4.2 Data Structures

The AgriCom system will utilize various data structure elements to assist in storing and processing system data for different scenarios. These planned data structures are ArrayList, Queue, and Entity Classes based on the database table structures.

ArrayList

ArrayLists data structures will be utilized to hold and process data that can contain more than one element for a given item. An example of this usage would be for calculating averages from transaction history. For this example there will be an unknown amount of transactions for a specific user and commodity, so an ArrayList will be used to pull this type of data and as this structure does not have a fixed size and can grow to accommodate the data needed.

Queue

Queue data structure will be used as the foundation for the AgriCom order system. For all order tickets, buy or sell, once placed by the user the order ticket will be placed into a queue for processing based on a FIFO (First-In-First-Out) principle. Since there may be multiple users placing orders during the same time domain, this will enable the system to follow closely with trading in real world conditions.

Entity Class

Entity Classes will be created to represent each table in the SQL relational database structure given that the tables' data will need to be referenced by the system. Tables such as User, Portfolio, Commodity, TransactionHistory, etc. will have entity classes created so the various columns and data elements can easily be pulled from the database and referenced in the application layer. Each entity will contain all of the columns and their corresponding data types, as well as getter and setter methods to access the data within each entity class.

5 User Interface Design and Implementation

5.1 Updated Design

At this moment, we are currently making the user interface and while it is not finished we are going off of the original mockups made in Report 1. This being said, we do have a few small changes that we are going to make for ease of use as well as consistency. There may also be some minor changes before the first demonstration but this has not been finalized or implemented yet.

The few changes that we do for sure plan on making include: adding an option on the login page to recover a lost password and changing the page name on the mockup of “Orders” to “Portfolio” to keep it consistent with the rest of the report and referencing. If we were to not include a way to recover a password it would be very difficult for anyone to recover their account and they would unfortunately have to make a new one in order to keep using the training.

5.2 Efficiency

A very important concern is making sure that the website is easy to use and fast for all users who attempt to use our website. For fast loading times we can and will logically break down our website into different elements so that the load time is more efficient especially for those elements that are unchanging and can be cached. For example, we can separate the header and the content of a given page. Since the header is the same across all pages, they only need to be loaded a single time to the client and can be cached on the client side for the duration of the visit to the website.

To be able to show that a user is interacting with the webpage without reloading it, we can take advantage of technologies such as Comet [5]. This will allow us to keep the pages dynamic while reducing load times by caching more information on the client side and perform updates only when needed with lower delay.

In order to reduce client load we will be using HTML[6] and CSS[7] to display our user interface while minimally using pictures if any. Any picture displayed will be contained in appropriate web format and resized to fit the container as needed. Our main audience is those training in the stock market so the user interface is made to be used in browsers. While we will make the pages responsive to browser size it will be optimized for those with a laptop or desktop computer. This also means that it is expected to have a browser web compliant with modern web standards. This should have minimal impact however as most people in our target audience will have those requirements met.

6 Design of Tests

6.1 Test Cases

Test-case Identifier: TC-1 **Function Tested:** Buy Commodity, SystemController::RequestBuy(Ticket : ticket) : Void **Pass/Fail Criteria:** The test passes if the request to buy commodity is sent to the DatabaseConnection. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-Correct data gets sent, correct commodity is added to the investor's portfolio -Data does not get sent, no new commodity was purchased

Test-case Identifier: TC-2 **Function Tested:** Sell Commodity, SystemController::RequestSell(Ticket : ticket) : Void **Pass/Fail Criteria:** The test passes if the request to sell stock is sent to the DatabaseConnection. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-Correct data gets sent, correct commodity is removed from the investor's portfolio -Data does not get sent, no commodity was sold

Test-case Identifier: TC-3 **Function Tested:** Register as Trainee, AccountController::CreateAccount(UserInfo: userinfo): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests an account creation and the request is granted.	
Test Procedure	**Expected Results**
-Request to create an account (Pass) -Call Function (Fail)	-AccountController requests account creation and returns true if account creation successful -If the request for account creation is

	unsuccessful, return false

Test-case Identifier: TC-4 **Function Tested:** View Portfolio, UserController::RequestPortfolio(String: Investor): Portfolio **Pass/Fail Criteria:** The test passes if the test stub requests for portfolio data and it is retrieved from the database	
Test Procedure	**Expected Results**
-Request portfolio data (Pass) -Call Function (Fail)	-UserController requests portfolio data and returns it from the database. -If there is an error retrieving the data from the database, it should display an error that no pertinent data was returned.

Test-case Identifier: TC-5 **Function Tested:** View Transaction History, SystemController::RequestHistory(String : investor) : Void **Pass/Fail Criteria:** The test passes if the request for the correct investor history is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-Correct data gets sent, RequestHistory of SystemController is called -Data does not get sent, RequestHistory of SystemController is not called

Test-case Identifier: TC-6 **Function Tested:** Submit Comment, SystemController::SubmitComment(TransactionID: transactionID, String: comment): Boolean **Pass/Fail Criteria:** The test passes if the comment is saved to the transaction in the database. The test fails if the request is not saved.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-Correct data gets sent, Database returns true. -Data does not get sent, Database returns nothing or false.

Test-case Identifier: TC-7 **Function Tested:** View Comment, SystemController::RequestComment(TransactionID: transactionID) : Void **Pass/Fail Criteria:** The test passes if the request for the correct comments under a transaction is sent to the DataHandler. The test fails if the request does not go out, due to an incorrect argument.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-Correct data gets sent, RequestComment of SystemController is called -Data does not get sent, RequestComment of SystemController is not called

Test-case Identifier: TC-8 **Function Tested:** View Educational Information, SystemController::RequestEducation() : Void **Pass/Fail Criteria:** The test passes if the request is called. The test fails if the request does not go out.	
Test Procedure	**Expected Results**
-Call Function (Success) -Call Function (Fail)	-RequestEducation of SystemController is called -RequestEducation of SystemController is not called

Test-case Identifier: TC-9 **Function Tested:** View View Commodity, CommodityAPI::Query(String: commodity):CommodityData **Pass/Fail Criteria:** The test passes if the system queries a commodity and that commodity is returned	
Test Procedure	**Expected Results**
-Request to query a commodity (Pass) -Request to query a commodity (Fail)	-CommodityQuery returns the commodity data -If the attempted query was for a commodity that does not exist, CommodityQuery should return an error code that the commodity does not exist. If commodity information was not attainable, it should display an error that no pertinent data was returned.

Test-case Identifier: TC-10 **Function Tested:** Register as Manager, AccountController::CreateAccount(UserInfo: userinfo): Boolean **Pass/Fail Criteria:** The test passes if the test stub requests an account creation and the request is granted.	
Test Procedure	**Expected Results**
-Request to create an account (Pass) -Call Function (Fail)	-AccountController requests account creation and returns true if account creation successful -If the request for account creation is unsuccessful, return false

Test-case Identifier: TC-11 **Function Tested:** Add Trainee to manage, AccountController::Manage(String: traineeID): Void **Pass/Fail Criteria:** The test passes if the test stub requests that a Trainee account be added to the Manager's account.	
Test Procedure	**Expected Results**
-Add Trainee account to the Manager's account. (Pass) -Add Trainee account to the Manager's account. (Fail)	-AccountController adds Trainee account to the Manager's account -If unable to add Trainee in database, display error saying that the operation was unsuccessful

Test-case Identifier: TC-12 **Function Tested:** Remove Trainee to manage, AccountController::UnManage(String: traineeID): Void **Pass/Fail Criteria:** The test passes if the test stub requests that a Trainee account be removed from the Manager's account.	
Test Procedure	**Expected Results**
-Remove Trainee account from the Manager's account. (Pass)	-AccountController removes Trainee account from the Manager's account

-Remove Trainee account from the Manager’s account. (Fail)	-If unable to remove Trainee in database, display error saying that the operation was unsuccessful

Test-case Identifier: TC-13
Function Tested: Reset Trainee’s account, AccountController::ResetPortfolio(String: traineeID): Boolean
Pass/Fail Criteria: The test passes if the test stub requests to reset a trainee’s portfolio settings in database and is successful. Unsuccessful if settings not changed

Test Procedure	**Expected Results**
-Request to reset a Trainee’s portfolio from the Manager’s account. (Pass) -Request to reset a Trainee’s portfolio from the Manager’s account. (Fail)	-AccountController modifies Trainee’s portfolio settings and returns true -AccountController unable to modify portfolio settings, returns false.

Test-case Identifier: TC-14
Function Tested: Provide end of training feedback, AccountController::EndofTraining(String: traineeID): Void
Pass/Fail Criteria: The test passes if the test stub calls EndofTraining when the Trainee’s training regiment expires.

Test Procedure	**Expected Results**
-TimerTask calls function. (Pass) -TimerTask calls function. (Fail)	-Checks with Database to find accounts that is expired and calls EndofTraining function -EndofTraining function fails to be called despite having accounts expiring in database

6.2 Test Coverage

The ideal test coverage would be to have a test that covers every edge case of every method. This is not only not feasible, it is impossible since it is not possible to actually know all the edge cases. Because of this we plan to test core functionality to provide a core amount of testing. Through the use of alpha and beta build interactions with end users, we will be able to identify ways that users may interact with the system that were not foreseen.

We can then add additional testing to cover these new edge and use cases which will also help debug and prevent regression in the future.

6.3 Integration Testing

Integration testing will be done on a local developer machine by emulating the server environment. The system may not go live until the current system works in the integration environment.

We accomplish this by having two branches of source code on GitHub, master and dev. dev is the branch that all new work will be done on. From there, it will be pulled down into the local integration machine to be tested and debugged. Once the system has been debugged with detailed logs of any system config changes, the source code will be pushed to master.

Once the source code is pushed to master, any system config changes will be made on the production server in order to accommodate the new branch. Once those changes are made, master will be pulled into the production machine and a second round of integration testing will begin by launching the service on a developer port. If it passes all the tests, then the developer port will be shut down, and the system will relaunch the website on the normal http port.

7 Project Management and Plan of Work

7.1 Merging Contributions from Team Members

In order to keep reports simple and up to date we have been keeping a google document open and shared to all team members so that everyone can edit and contribute in real time. This keeps us all up to date on the progress of each team member and if they end up working in a separate space it is easy to implement their work into this document. Reports are divided up between team members and to each member's strengths as evenly as possible. Compiling the report has also been made easy as the style has been kept the same from the first report and each team member remains consistent throughout writing the report while contributing. If anyone spots a formatting error they can fix it easily without hurting anyone's work and it is backed up as well in case something important gets deleted.

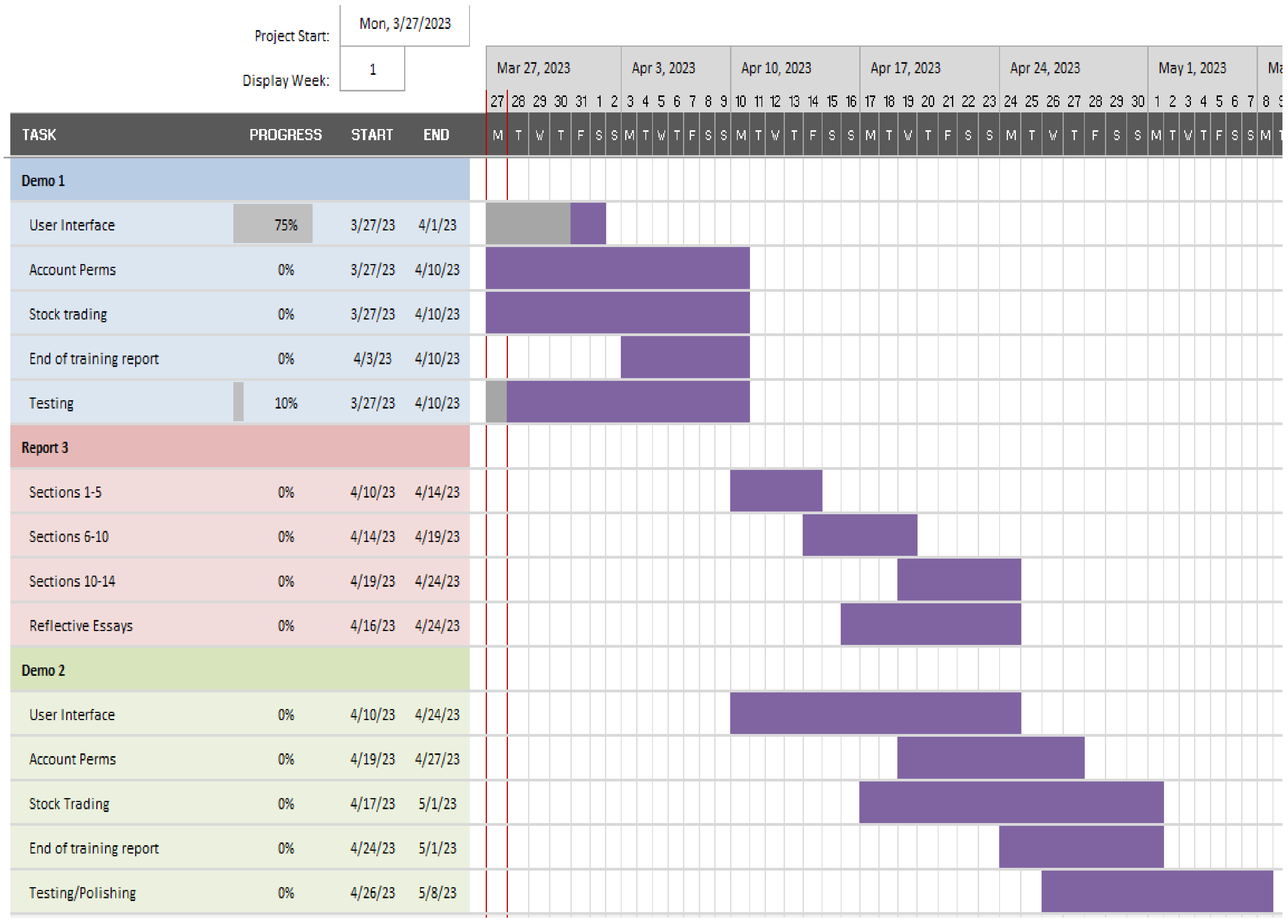
7.2 Project Coordination and Progress Report

So far, UC-3 and UC-10 are in progress and mostly finished, however development is still in progress. The database has been made and linked to the website so a user is able to create a user and log in to the website. The proposed databases for the other parts of the project have also been made. The functionality of the stock trading system is a work in progress but the user interface has been frameworked so that it is ready to be inputted into and formatted.

In order to keep track of progress and keep organized as a group, we have been scheduling meetings at least twice a week. This way we can set timeframes of when a section of work should be completed instead of waiting until the due date. We have set up a group chat where anyone can ask a question at any time and we can all see it and respond when needed. This lets us solve problems as they arise and keep the responsibilities evenly distributed instead of asking a single person to help.

7.3 Statement for Plan of Work

Project was broken down into two components during the duration of the time to meet the two due dates for Report #2 and the upcoming demonstration. Each member was able to use, develop, and code parts of the report into the demo to be presented for the following month. Parts of the report for the demo were included in the Discord server made by the team to better share and present information that was being developed throughout the course of the assignment as a procedure in place to track progress and gain perspective into the website being developed. The five people were able to compile the information into a readily made report with available backing to help define what we were to present but in essence had to ensure that communication and follow-up were present throughout the duration of the assignment. Future plan of work is outlined below in Gantt chart:



Project Start: Mon, 3/27/2023

Display Week: 1

TASK	PROGRESS	START	END
Demo 1			
User Interface	75%	3/27/23	4/1/23
Account Perms	0%	3/27/23	4/10/23
Stock trading	0%	3/27/23	4/10/23
End of training report	0%	4/3/23	4/10/23
Testing	10%	3/27/23	4/10/23
Report 3			
Sections 1-5	0%	4/10/23	4/14/23
Sections 6-10	0%	4/14/23	4/19/23
Sections 10-14	0%	4/19/23	4/24/23
Reflective Essays	0%	4/16/23	4/24/23
Demo 2			
User Interface	0%	4/10/23	4/24/23
Account Perms	0%	4/19/23	4/27/23
Stock Trading	0%	4/17/23	5/1/23
End of training report	0%	4/24/23	5/1/23
Testing/Polishing	0%	4/26/23	5/8/23

7.4 **Break Down of Responsibilities**

Working with a team of five can have its benefits and its detriments. Coordinating tasks and assignments help prospects and timelines create expectations to gauge team readiness. Report #2 has had a similar breakdown in duties and assignments in that each team member contributed towards the overall readiness of the report to be presented.

Breakdown of assignments are as follows:

- Alexis Angel – Spearheaded assignments and divided out tasks for the team to manage.
- Christopher Katz – Contributed when possible and took into account progress and planning.
- Calvin Ku – Handled mathematical modeling and diagramming.
- Dave Schiffer – Created blueprints and outlines of data types and diagrams.
- David Gladden – Modeling and diagramming throughout the report.

Future responsibilities for coding and testing are split into teams and each team is responsible for testing their own code initially and once integration is completed between sections both teams involved should test functionality. Teams are as follows:

User Interface Functionality - Alexis, Chris

End users should be able to experience a unified outcome across mobile, tablet, and desktop browsers. The customer should be able to use major modern browsers such as: Firefox, Chrome, Safari, and Edge to view this experience. Login should be simple and readily available once on the website, and a link to UI for trading and seeing account status should be accessible from the home page. Trainee users can view educational information as tooltips when hovering over certain UI elements. End users should also be able to access the newsfeed easily and navigate through news posts without issue or delay.

Account Portfolio Functionality - David, Alexis

End users should be able to log in and access their user profile and a defined list of tradable commodities. They can also view their account information, including cash balance, current holdings, and account totals.

News and Price Retrieval Functionality - Dave, Calvin, Chris

End users should be able to access and view newsfeed information about agricultural commodities. Current prices from the predefined list of items are considered as close to real-time as possible.

Account Permissions Functionality - Chris, Alexis

End users should have separate account permissions established and implemented relating to trainee or manager-level licenses. Manager-level permissions will have elevated rights to enable them to assist the trainee accounts they are supervising. The end user can log in very straightforwardly to access the system. The user is also able to update their account information as well as log out of the system. Managers can control access to information and have the ability to restrict user access.

Trading Functions Functionality - David, Calvin

End users should be able to place purchase orders for commodities, given their accounts have a positive cash balance. Users can also place sell orders for commodity positions their funds are currently holding, and managers can see what accounts under them are trading.

8 Report 2 Contributions

	Names				
Category	David Gladden	David Schiffer	Alexis Angel	Calvin Ku	Christopher Katz
Analysis and Domain Modeling	33%	0%	33%	33%	0%
Interaction Diagrams	50%	50%	0%	0%	0%
Class Diagram and Interface Specification	0%	100%	0%	0%	0%
Algorithms and Data Structures	50%	0%	0%	50%	0%
User Interface Design and Implementation	0%	0%	100%	0%	0%
Design of Tests	0%	0%	0%	100%	0%
Project Management and Plan of Work	0%	0%	80%	0%	20%

References

- *CommoPrices API*. API CommoPrices. (n.d.). Retrieved February 16, 2023, from https://api.commoprices.com/
- | *commodities prices and currency conversion JSON API.* Commodities-API. (n.d.). Retrieved February 16, 2023, from https://commodities-api.com/
- Lioudis, N. (2022, December 19). *Commodities trading: An overview*. Investopedia. Retrieved February 22, 2023, from https://www.investopedia.com/investing/commodities-trading-overview/
- Palmer, B. (2023, February 3). *A beginner's Guide to Precious Metals*. Investopedia. Retrieved February 22, 2023, from https://www.investopedia.com/articles/basics/09/precious-metals-gold-silver-platinum.ap
- Wikimedia Foundation. (2022, August 15). *Comet (programming)*. Wikipedia. Retrieved March 24, 2023, from https://en.wikipedia.org/wiki/Comet_(programming)
- Wikimedia Foundation. (2023, March 23). *HTML*. Wikipedia. Retrieved March 24, 2023, from https://en.wikipedia.org/wiki/HTML
- Wikimedia Foundation. (2023, March 23). *CSS*. Wikipedia. Retrieved March 24, 2023, from https://en.wikipedia.org/wiki/CSS#