Financial Portfolio Tracking System: Release 2

Submission Date: April 15th, 2016

Design Documentation
Prepared by Group B:

Kayla Nussbaum <kxn3867@rit.edu>
Philip Bedward <pxb2841@rit.edu>
Mica Lewis <mxl7287@rit.edu>
Daniil Vasin <dxv1489@rit.edu>
Ryan Hartzell <rxh4133@rit.edu>
William Estey <wpe2632@rit.edu>

Table of Contents

Summary

Financial Portfolio Tracking System (FPTS) provides the ability to track financial portfolios containing equities such as stock, bond, and mutual fund holdings, and cash in one or more cash accounts. FPTS can track multiple portfolios each with individual ownership. This system can simulate market conditions to show the effect on the user's portfolio. After understanding the requirements and functionality of the system, our team began by creating diagrams to model the system architecture.

Architecture and Diagrams

Our system architecture started with a very high-level design, noted on Page 8. This design represents our domain model that is primarily used for the basic understanding of the system as a whole. All of the components are shown here, as well as their relationship to one another in the system (Portfolio, Equity Info, Holdings, Transaction, Importing/Exporting Component, Simulation, User, Cash Accounts, and Equities). Upon implementation, we developed our design into something on a much lower-level; This is reflected in our large, UML Class diagram, noted on Page 7. This diagram holds specific method names, state information, return types, and attributes for each of the classes. Our team decided that this was a fundamental diagram that we needed to include for our total understanding of how each of these classes cohesively worked together in our system.

For the purpose of system architecture and design documentation, we partitioned the composition of our designs into five different subsystems: User Info, GUI, Log In/Log Out, Search, Simulation and Undo/Redo. The user info subsystem keeps track of all elements of the portfolio. The GUI displays information from the User Info to the user. The Log In/Log Out subsystem determines the behavior of the opening and closing of the program. The Search system maintains and retrieves the information about equities from web services. The Simulation subsystem projects the value of the current portfolio under multiple market environments. The Undo/Redo subsystem applies and keeps track of a long list of commands that manipulate the data of User Info. By having the classes listed out in their own subsystem within the large, low-level abstraction UML, we were able to closely see the separation and relationship between apparent associations within our system. By partitioning our structure into subsystems we we able to recognize the importance of using software patterns to develop a more robust system.

There is a final UML diagram that is composed of only the system's subsystems, noted on Page 9. Including this makes it very simple for the audience to interpret the way this system's components are working together. More information about the architecture of our subsystems is located on Page 9.

Apart from the architecture of the system as a whole, there are several sequence diagrams within this document that walk through the functionality noted in the problem statement of the project. These diagrams essentially satisfy how a user can log in to their account, view and manipulate their profile information, search for holdings and their information, and describes how the simulation handles all of this information. In conclusion, the main architecture of this system is represented by the GUI which conveys all the back-end information (that is detailed and described in the diagrams and models) to the user.

Design Pattern Usage and Rationale

Three major design patterns were used in the various subsystems of this project. Certain subsystems reflected these patterns and were implemented according to standards of those patterns.

Each of these patterns are specified in this document. The Simulation subsystem contains a memento design pattern and a strategy design pattern. The GUI subsystem contains the an observable pattern mostly built using the default java observable class. All of the design is vaguely contained within the philosophy of model view control, but these three patterns have been implemented more concretely.

The memento pattern in the simulation subsystem is made mostly within the simulation class, but still fulfills all of the requirements of the memento pattern using default java data structures. The memento itself is an arraylist of holdings that are modified by the simulation method. The caretaker of the mementos is a java stack of arraylists of holdings. The use of a stack as a caretaker is in response to the requirement of the system to allow the user to go back a certain number of simulation executions. Each simulation creates a new memento which is placed on the stack. When the user wants to go back to previous executions of the simulation, the stack will pop the head of the stack and thus a previous memento will be the new current memento. This implementation is further specified in the Design pattern on Page #19.

The Strategy pattern allows the user to choose a certain implementation of the market simulation algorithm. The interface MarketSimulationStrategy specifies implementations of the simulation. Three classes extend this interface and provide the methods that the memento pattern uses to create new mementos by running simulations. This satisfies the requirement to include simulations for Bear, Bull and no-growth markets. The implementation of this is further specified on Page #18.

The Iterator pattern allows us to further encapsulate data. We use it to protect the list of equity metadata (Name, ticker symbol, current value, market averages) from being modified. Instead of getting the list, you can only get the iterator over it, stopping the list from being modified.

The Observer pattern, as part of the GUI subsystem, ensures that the FXML view is kept up to date with the data of portfolio. The abstract classes required for the pattern are part of the default javaFX implementation of the observer pattern. While this doesn't fulfill any requirement, it allows the system to display all information the portfolio contains. The implementation of this class is further specified on Page #9.

The Undo/Redo Subsystems makes use of a few design patterns. The Command Pattern is used to perform user actions that affect the data of the equities persisted in their portfolio. The Composite pattern is used in conjunction in order to build MacroCommands such as the TransactionCommand. A slight version of the Memento pattern is used in order to store the commands in order and undo/redo them whenever the user makes the request. The SingletonPattern is used to ensure that only one command care taker exists in our system. We did this in order achieve the requirement of allowing the user to undo and redo their commands up to five times.

Design Principles

Good cohesion and low coupling are achieved by separating minor tasks into classes appropriate to which are relevant to those tasks. Good coupling is achieved by separating the major tasks into Subsystems where classes would then only communicate to classes outside of their subsystem if they absolutely have to. This is almost the definition of the Law of Demeter with the other components relevant being a part of the same subsystem; LoD focuses on loose coupling. All information specific to each class is designated as private, whereas getters and setters are only implemented as necessary satisfying proper information hiding. Finally, the code is as modular as possible, with much room for expansion on existing class if new requirements arise later on.

Requirements

The primary functionality of the FPTS is to persist multiple portfolios of financial information. A portfolio is protected by associated log in information that is created with formation of a new portfolio. Each portfolio contains a transaction history and holdings that are either both empty or imported upon creation. Portfolios can be imported from previous portfolios or CSV files generated by other Financial Portfolio software. At the end of a given invocation of the program, the user will log out and be prompted to save the portfolio for future use or to be used by a different financial portfolio system.

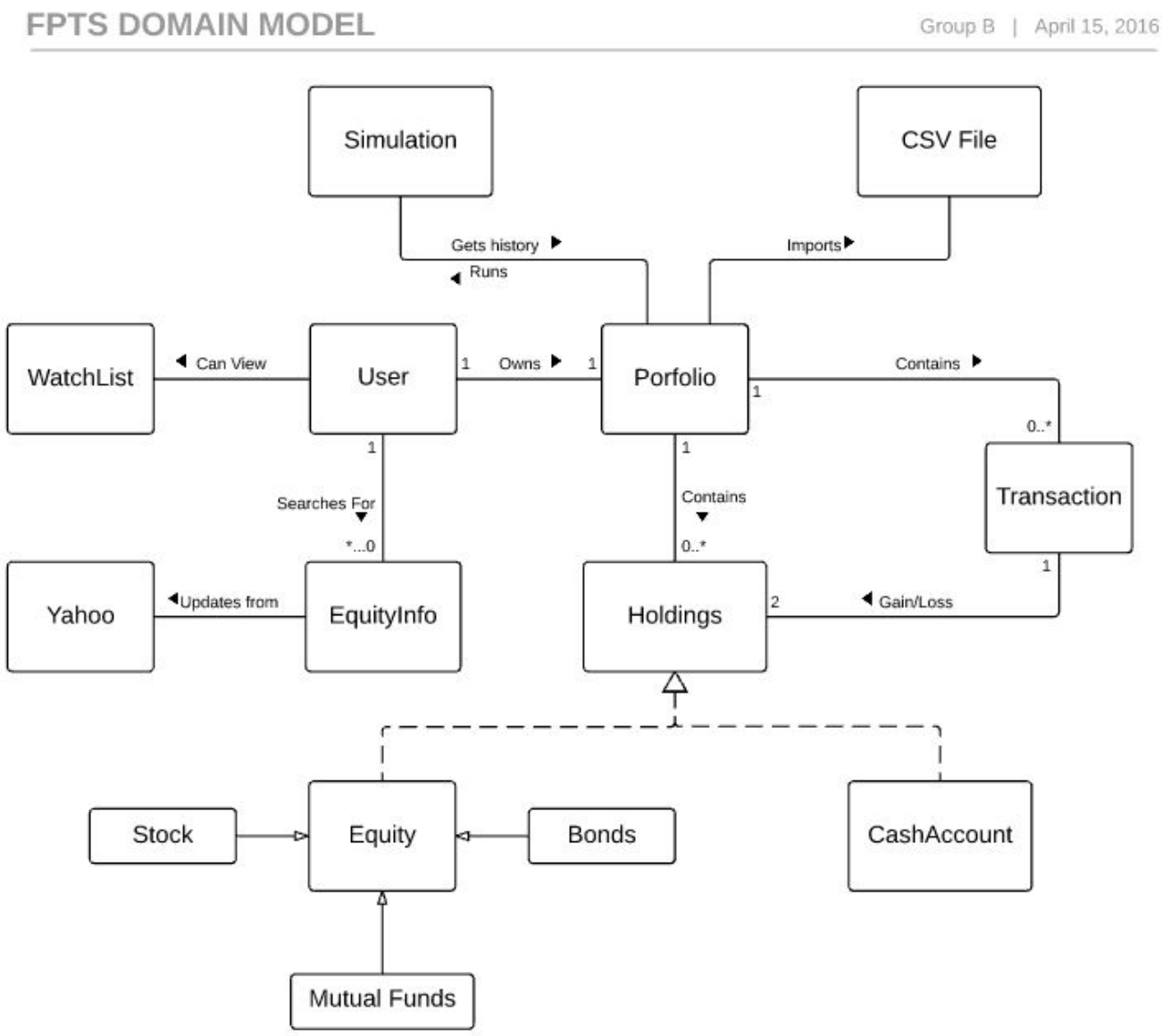
Holdings can be either Cash Accounts with a bank name and value, or equities with a ticker symbol and quantity of shares. Cash Accounts and equities can be added to or subtracted from. Transactions can be added that exchange a value from a cash account for a quantity of equities, or vice versa, or record an exchange without affecting the current holdings. Holdings can also be added via the same import system from the creation of the portfolio. All actions done to the system can be undone or redone.

In addition to maintaining information about a portfolio, the FPTS will also maintain information about equities including ticker symbol, name, and market indices. This information will be updated on regular intervals by connecting to the Yahoo! Financial Web Service. These equities can be searched for when doing some action on the portfolio by means of ticker symbol, equity name, index and sector. Portfolio can also create a watchlist of some equities that will notify the user when an equity goes past a certain point.

Another capability of the FPTS is to run a simulation on the portfolio to show its value across across different types of market. These simulations grow or shrink in value linearly according to some user specified interest rate. After a simulation has been run, the user can specify a new simulation with different rates of growth or shrinkage, or they can revert to any previous execution of the simulation.

Domain Model

This section provides a domain model for the project. This diagram follow the guidelines discussed in class and the design project activity sheets. Below is a simplified model of this system; It provides a simple structure replication for how the system will run and how each module interacts within the system. A Portfolio is related to the Simulation, the file of which it imports, a user and its information in the system, specific holdings, and transaction history.



System Architecture

UML Class Diagram (see UML R2.pdf for diagram)

Simple UML Class Diagram (see SimpleUML.pdf for diagram)

Our system relies heavily upon the MVC pattern. All changes to the model are ultimately controller driven, which changes the model, which then updates the view. For example, when logging in and logging out, the user enters their login and password in text fields and then presses a button to initiate the login process. The login controller for the login screen view is the cornerstone of the Login/Logout subsystem, and prompts the User Info subsystem to load the appropriate user portfolio. The User Info subsystem then returns whether it was successful in loading or not, and if it was, the GUI subsystem then displays the portfolio information gotten by the User Info subsystem.

When the user wants to buy or sell Equities, the GUI subsystem (which includes the controllers) communicates with the Search subsystem (which works with the User Info subsystem) to get all the possible Equities to buy or sell. The user then selects which one out of the list that they are buying or selling, and fill in the rest of the transaction info. The GUI then tells the User Info subsystem to execute a transaction with this information. The User Info subsystem then adds the gained holding to the holdings it holds, and subtracts the lost holding from the holdings it holds. It then adds the transaction object to the list of transactions it holds as the record of transaction history.

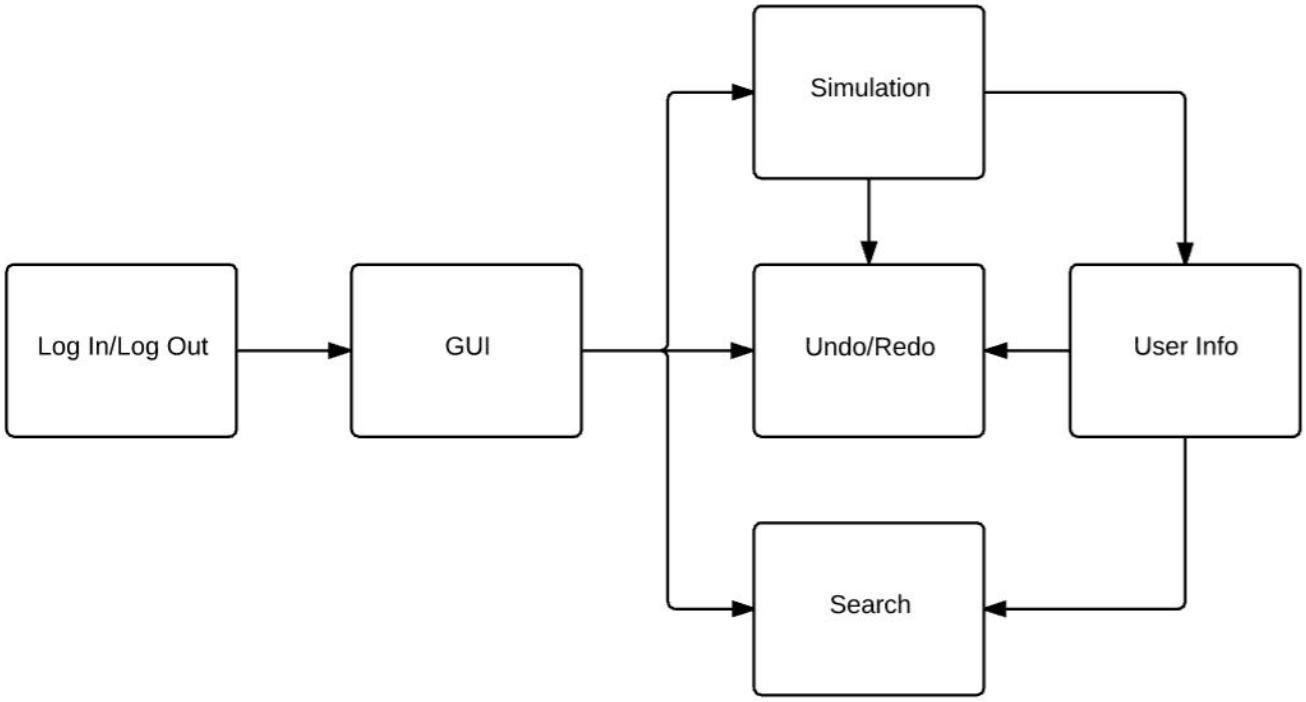
When the user wants to run a simulation, they first enter all the information necessary for the simulation (number of steps, run step by step, strategy, etc). They then press either the “Run All” button or the “Run Step by Step” button, and the GUI tells the Simulation subsystem to begin a simulation. The Simulation subsystem then gets the initial equities from the User Info subsystem. If the user requested a step by step simulation, the Simulation subsystem then returns the first step of the simulation to the GUI subsystem to display. The GUI then gets the next steps of the simulation from the Simulation subsystem when the user presses the next step button.

The GUI also asks the Simulation subsystem for the number of steps that have passed, and displays it to the user. If the user did not want the simulation run step by step, then the Simulation subsystem simply returns the end result of the simulation. Regardless of steps, once the final state of the simulation has been displayed, there are 3 options for the user. They can either choose to end the simulation, in which case the GUI subsystem tells the Simulation subsystem to clear the simulation stack, restore the simulation to the end of the previous simulation, in which case the GUI tells the Simulation subsystem to backtrack, and the Simulation subsystems returns the result of the previous simulation for GUI to display, or the user can select to run another simulation, in which case the cycle starts all over again.

Subsystems

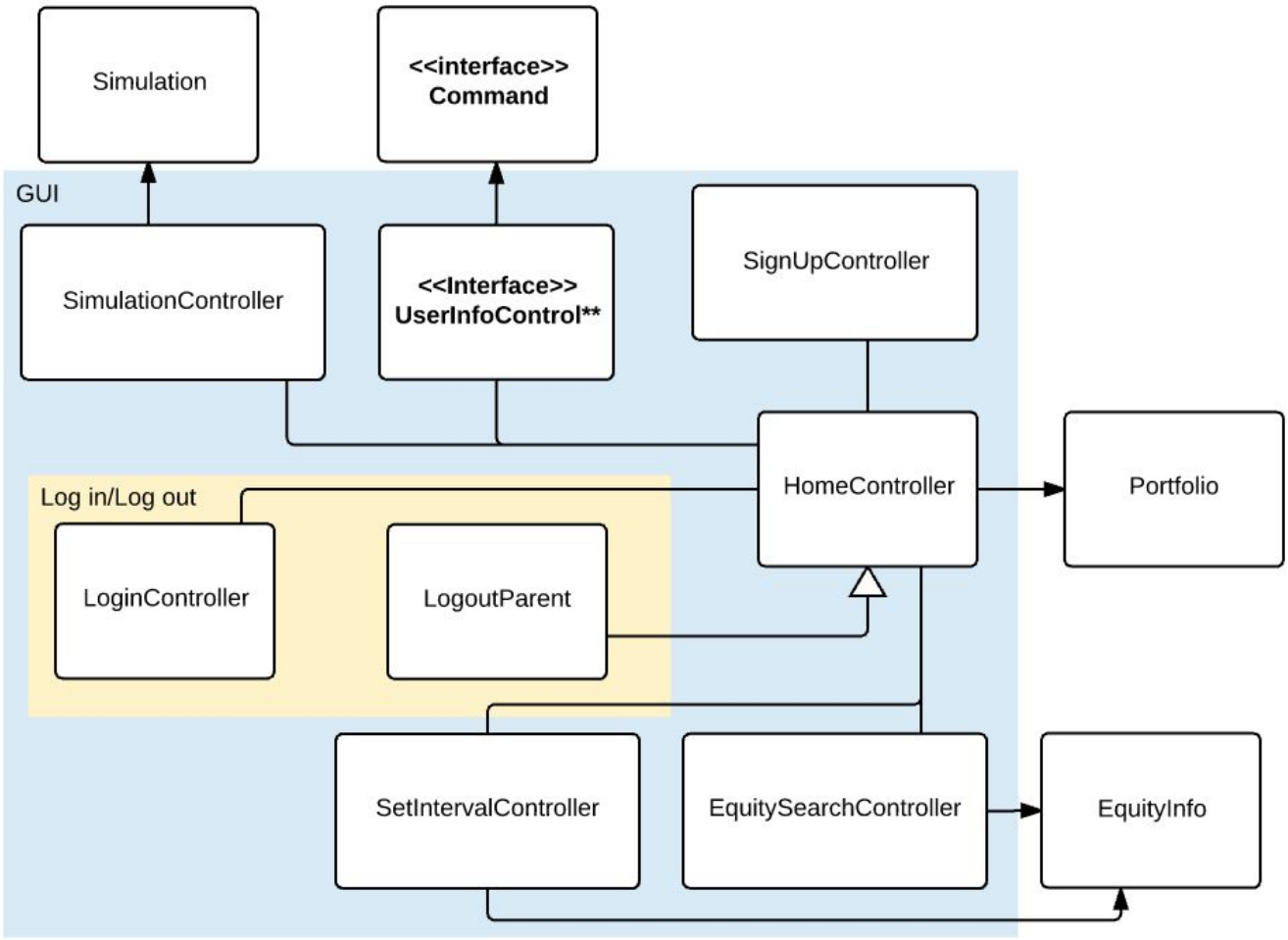
This system is broken up into five subsystems: Log In/Log Out, GUI, Simulation, User Info, and Search. Upon running the application, the user would access the Log In/Log Out subsystem. The GUI is then updating with their Login information or prompting them to sign up. This then accesses the User Info subsystem that holds two user actions: Simulation and Search (which are the remaining subsystems in our application). These subsystems seamlessly work together throughout the entire program and constantly update the GUI.

Below is a simplified UML diagram showing the relationships between each of the subsystems and how they work together in our system. This is a high-level design made to simplify the idea of the flow of our system's classes.



Subsystem #1 – GUI

The system's Graphical User Interface (GUI) is represented by this subsystem modeled below. The GUI is created using the Java FX Library. This subsystem is responsible for portraying a system view of all the user's information logged in our database from their personal Financial Portfolio. Shown in this diagram are the three main controllers that organize the information to be represented by the GUI. LogoutParent provides a view for the system when needing to access logging in, logging out, or being on the home screen. Once a user signs up and logs in, they are systematically redirected to the HomeController that is responsible for conveying the information to the GUI view (LogoutParent).



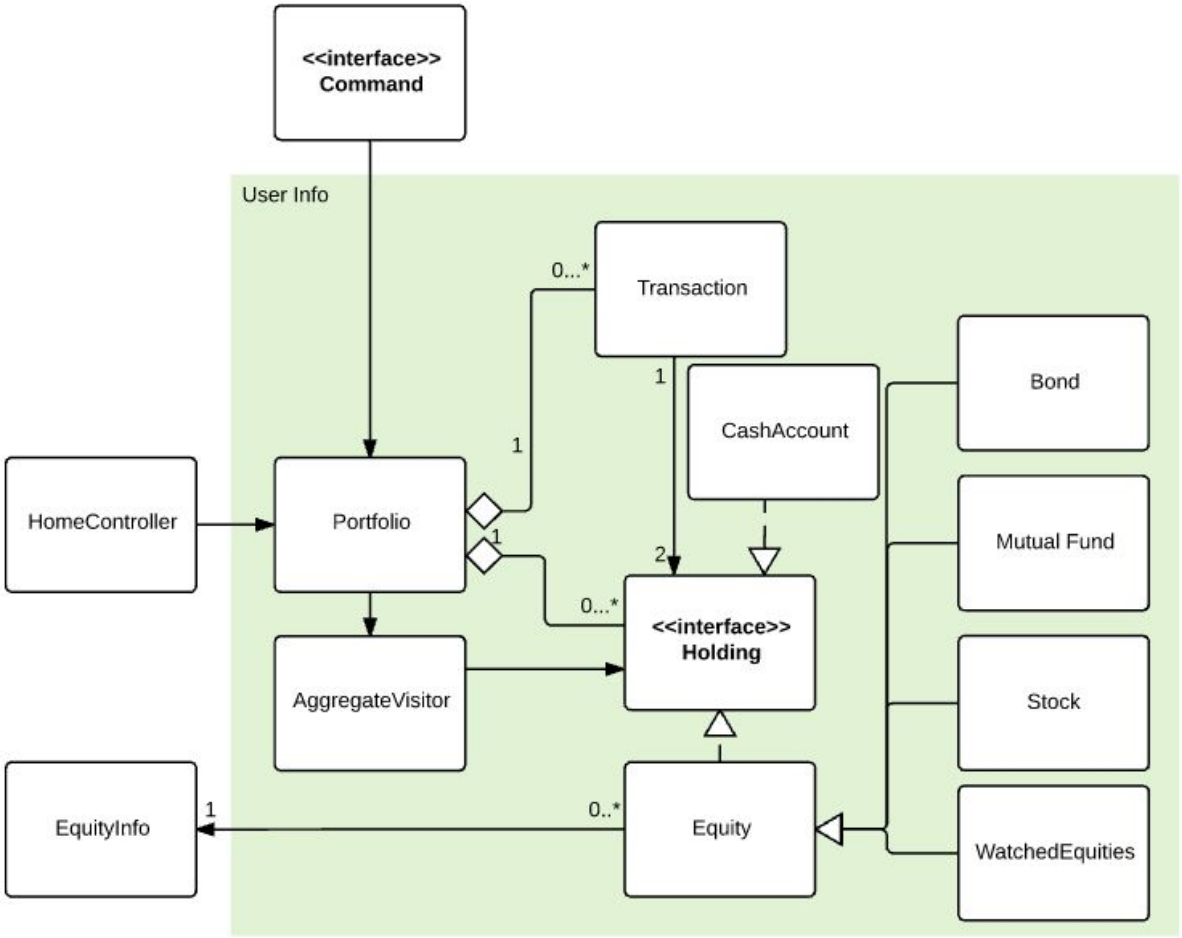
**represents many classes of contoller that all have similar function

GUI: Design Pattern

Name: GUI		**GoF pattern: Observer**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Observable	Subject	This is an abstract predefined by the Java language. Allows the changes made to a subject that extends this class to be seen by an Observer
Portfolio	ConcreteSubject	This class maintains reference to the Portfolio GUI. The changes in it can be seen by the GUI.
Observer	Observer	This is an interface predefined by Java. This provides methods that the ConcreteObserver has to override in order to reflect the changes that are made in the GUI.
HomeController	ConcreteObserver	This class maintains reference to the Portfolio and portrays a visual representation of said object. This directly implements the Observer updating interface.
Deviations from the standard pattern: Due to Java's libraries for the Observer pattern, our implementation should follow most of the characteristics of the pattern. However, as shown above we will separate some control functions into a separate class to implement the Command Pattern		
Requirements being covered: The requirements being covered are any that include an implicit reference to having a GUI for the user including, but not limited to as logging, in and logging out, the user viewing portfolio and all of its information, and the user searching for items through the system.		

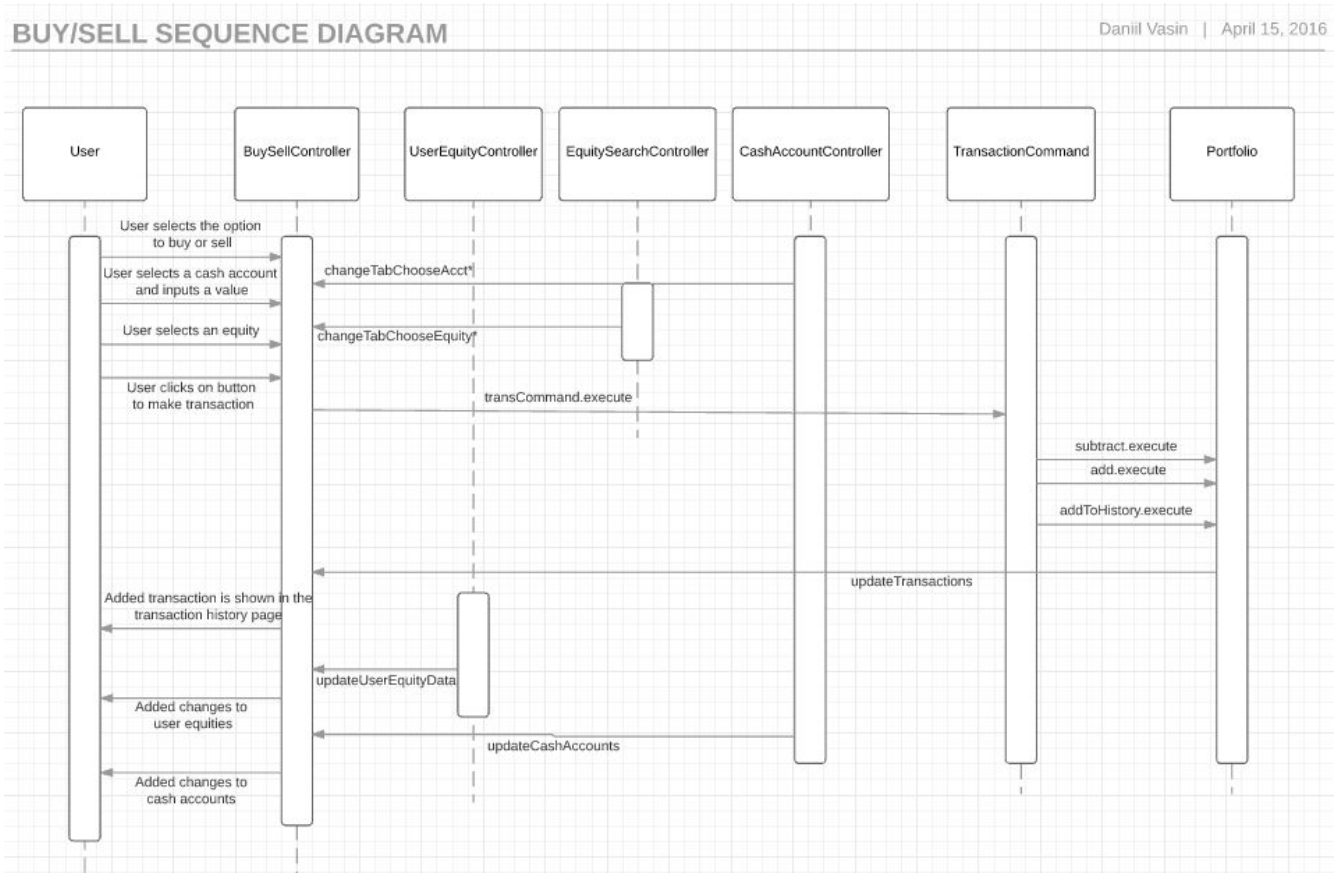
Subsystem #2 – User Info

This subsystem is responsible for keeping track of all user information, such as the user's user ID, password, transaction history, and holdings. It is also responsible for adding transactions and holdings. Each holding has a cash account and and equity that is composed of stocks, bonds, and mutual funds. All of these entities are reflected within a portfolio that also contains transaction history and actions.



Buy/Sell: Sequence Diagram

A user selects the option to buy or sell an equity by going to the buy/sell tab. They then press a button to select an account, which switches them to the cash account tab. They then select an account on the table and click select, which takes them back to the buy/sell tab and populates the field designated for the CashAccount. They then click the select equity button, which opens a window with a list of all equities available to be bought, along with 3 textfields allowing the user to filter the results on the table. The user then selects an equity on the table, and presses the select button. The window then closes, and the text field designated to display the equity to be bought is populated. The user then enters the number of shares they want to buy/sell, and presses the buy/sell button. A transaction command is then created, which executes subHolding() or addHolding() on the currentPortfolio with the Holdings gained and lost. The currentPortfolio then tells the BuySellController to update the list of transactions which tells the user equity controller to display an updated list of user equities. The user is then moved to the user equity tab, where they can see that they have a new equity or the equity they sold is no longer there.



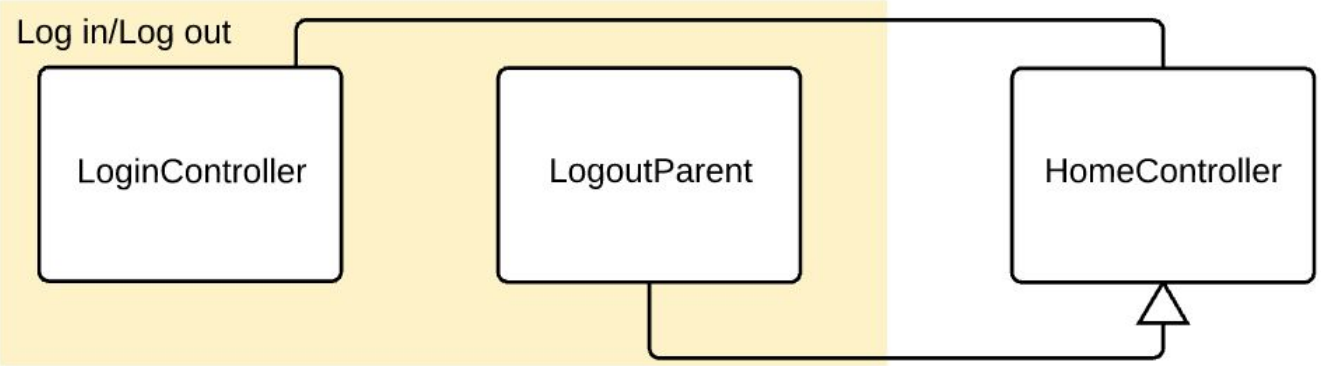
User Info: Design Patterns

Name: Statistics		**GoF pattern: Visitor**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Visitor	Visitor	This is the interface implemented by all ConcreteVisitors. Allows the ConcreteElements to send requests to the ConcreteVisitor object
AggregateVisitor	ConcreteVisitor	The AggregateVisitor implements all methods defined in the Visitor, this includes the visit() method. The method takes a Holding in and provides some operation on it
Holding	Element	Defines the accept operation and other methods for the Concrete Elements to implement.
Equity	ConcreteElement	This class implements the accept method and all other operations defined by the Holding interface.
Cash Account	ConcreteElement	This class implements the accept method and all other operations defined by the Holding interface.
FPTSRunner	ObjectStrucutre	(usually the program)
Deviations from the standard pattern: There are not many deviations from the standard pattern in our implementation that we are aware of.		
Requirements being covered:		

Name: Equity Information		**GoF pattern: Iterator**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Container	Aggregate	Ensures that all aggregate classes that want to use iterators have the getIterator() method.
Iterator	Iterator	Provides the method headers for iterator, ensuring that all iterators have the two essential methods, hasNext() and next(), for determining if there is more data in the aggregate and providing a way to get the next piece of data.
EquityInfoList	ConcreteAggregate	An internal class in EquityInfo that provides a very efficient way to improve encapsulation. It lets us pass an iterator over this list to people who need to display all the equity metadata. If we give them the list, there's the chance they can edit it. If we give them an iterator, they can access all the elements of the aggregate, but they cannot modify it directly.
EquityInfoList Iterator	ConcreteIterator	An internal class inside EquityInfoList that provides iteration behavior. Implements Iterator, and is provided when an iterator over an EquityInfoList is requested.
Deviations from the standard pattern: All of the concrete classes are private internal classes, used to preserve and heighten encapsulation.		
Requirements being covered: The user has the ability to store their information in the Portfolio. This pattern enforces encapsulation in order to achieve the customer's request.		

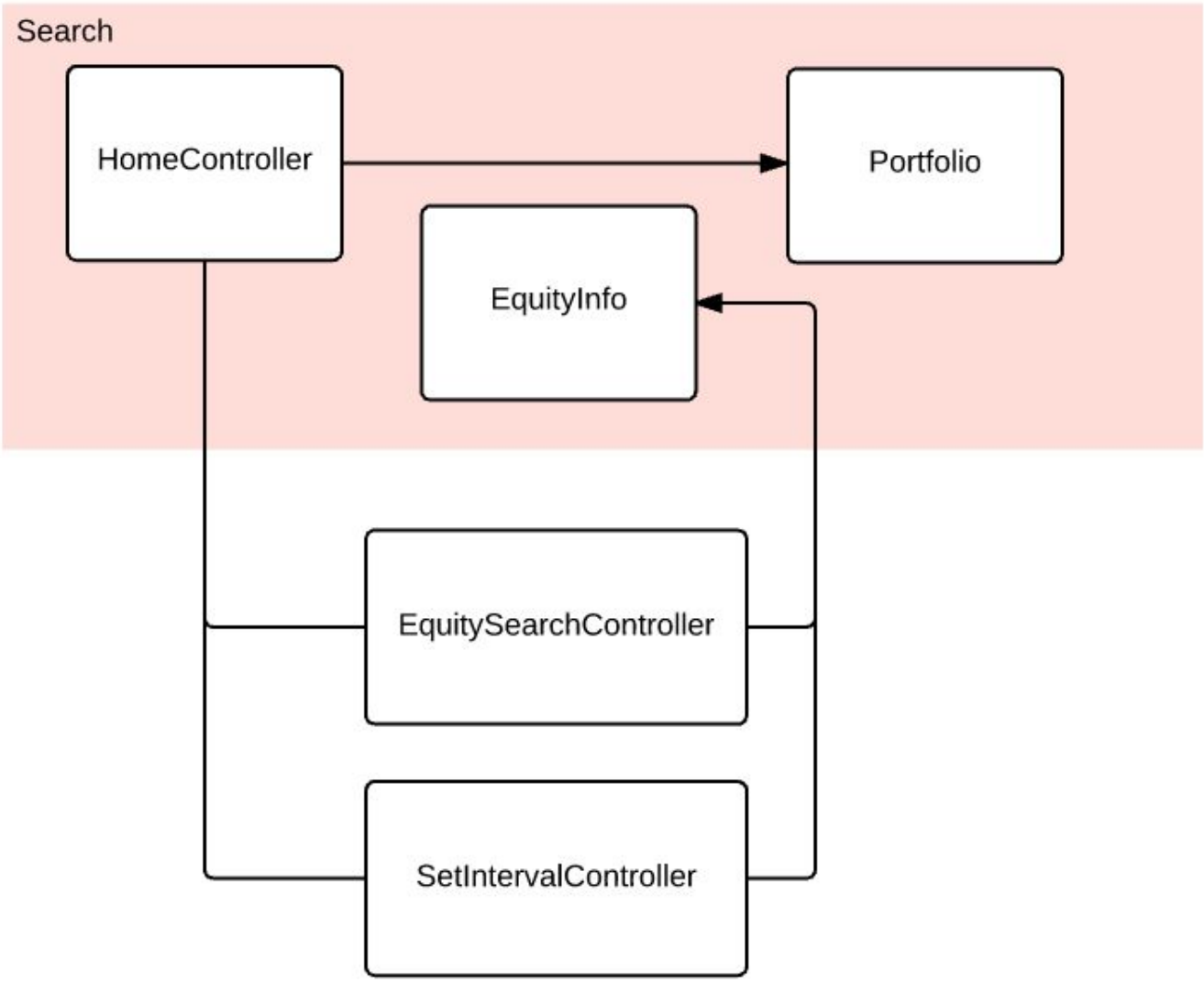
Subsystem #3 - Log In/Log Out

This subsystem is responsible for handling the saving and loading of user profiles, as well as switching the current GUI view to the home view when logging in and the login screen when logging out and prompting the user to save when they want to quit or logout. The LogoutParent class shown below provides the logout button and functionality to all GUI classes that are only shown while logged in. The other controllers that interact with this subsystem all extend LogoutParent as it is one of the main components to how the system functions.



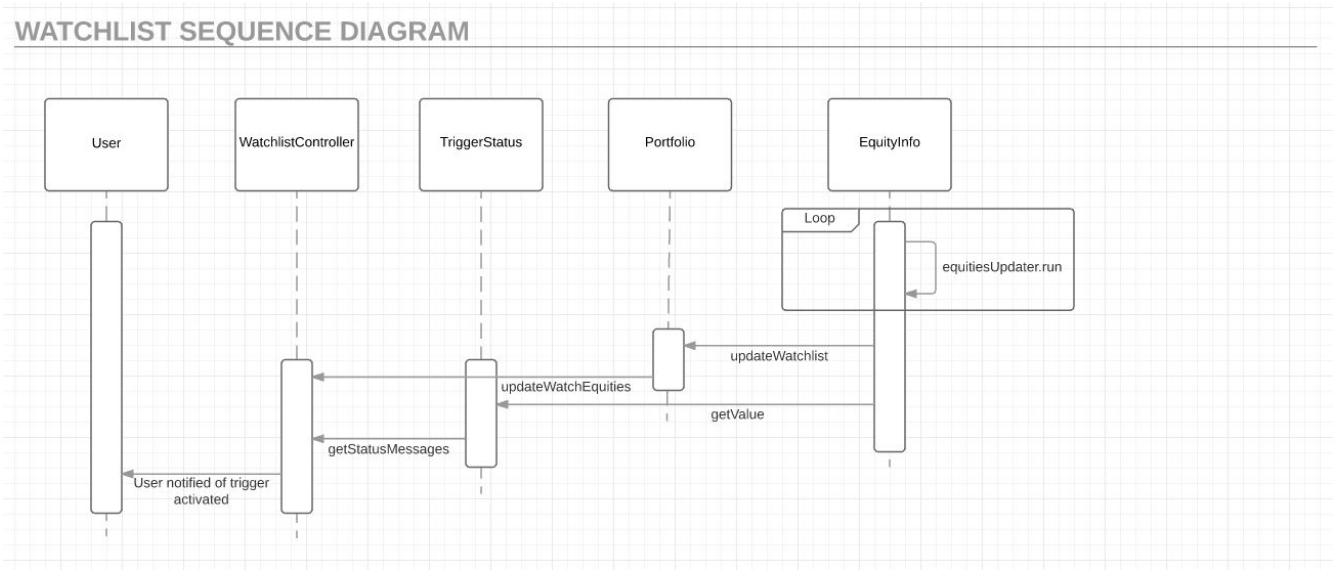
Subsystem #4 – Search

This subsystem is responsible for finding an Equity from equities.csv that matches a desired name fragment, ticker symbol fragment, and market index fragment. It is also responsible for finding an Equity that a user owns that matches those same fragment types. If all fields start with or contain their respective fragments, then that Equity is considered a match. This subsystem is mainly used by the HomeController class in the GUI subsystem. It uses the search module to populate a table filled with Equity information, such as ticker symbol (Ticker), equity name (Name), equity value (Value), and market indexes (Market).



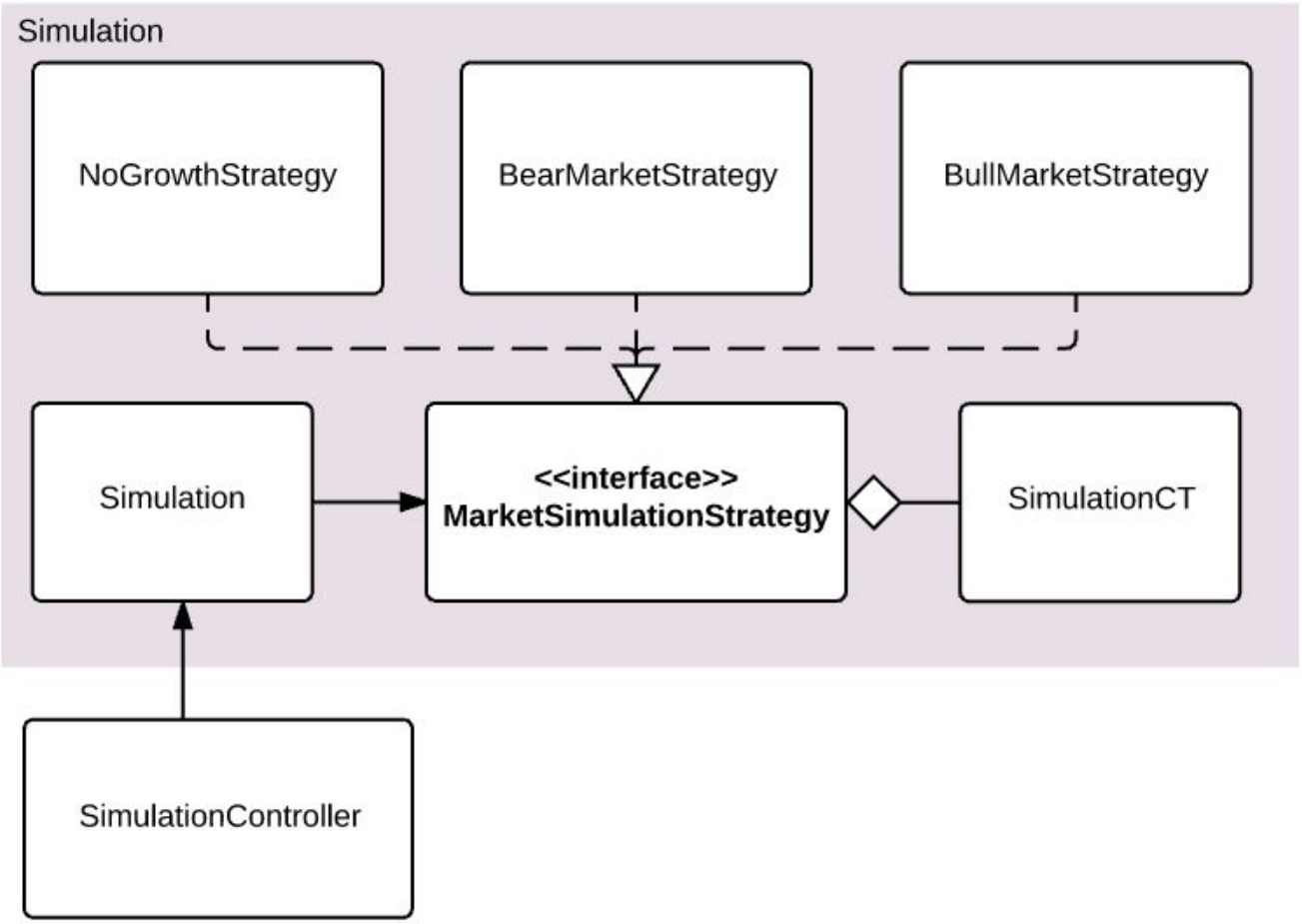
Watchlist: Sequence Diagram

The watchlist is one of the more complex pieces of the program. The values of the EquityInfos are updated at an interval specified by the user. Once it's updated, it tells the currentPortfolio to update its watchlist before putting its thread back to sleep. When the values of the EquityInfos change, the TriggerStatus objects in the watchlist that are listening to their respective EquityInfo objects are updated with their new values. Depending on the high and low trigger values, it sets its status message accordingly. The watchlist controller is listening to the TriggerStatus objects of the WatchedEquity objects, and in turn the TriggerStatus listens to the value of the WatchedEquity and updates its message when a trigger(either high or low) is activated. When they update, it gets the status messages from the TriggerStatus object and displays it.



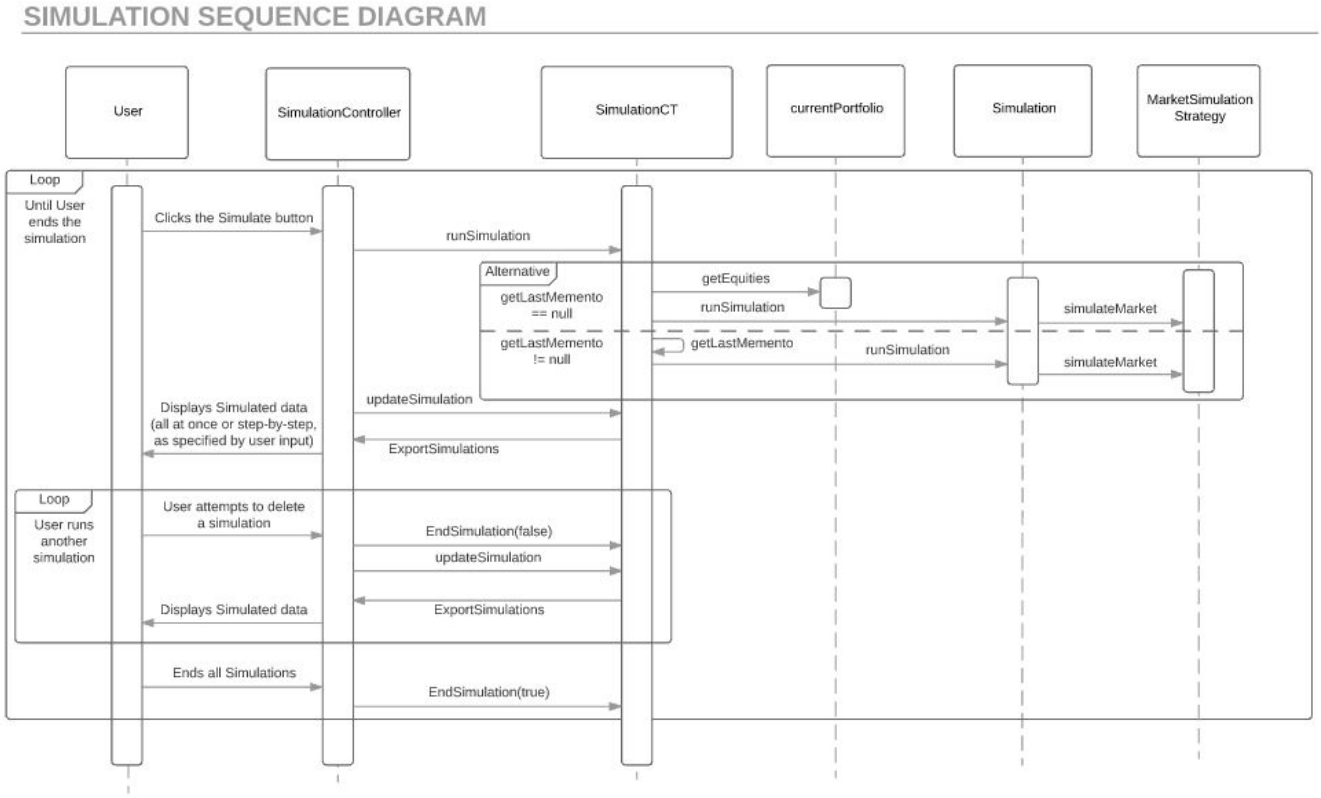
Subsystem #5 – Simulation

This subsystem is primarily implemented in the Simulation class. The Simulation subsystem utilizes the Memento design pattern, which allows the system to create mementos of the different moments in the simulation, which provides the ability to go backwards and forwards in the simulation timeline to get a detailed view. It also utilizes the Strategy design pattern in that the Simulation class acts as the context and that there are 3 possible market simulation strategies that implement the MarketSimulationStrategy interface: the BearMarketStrategy (loss), BullMarketStrategy (gain), NoGrowthStrategy (status-quo). The caretaker of the simulation mementos is a caretaker class which holds an ArrayList of simulation objects which act as mementos. The ArrayLists of Equities which act as the mementos are created by the Portfolio class and are modified by the Simulation class.



Simulation: Sequence Diagram

The user begins the Simulation subsystem by providing the necessary parameters; chosen strategy (BearMarket, BullMarket, or NoGrowthMarket), size of steps (day, month, or year), number of steps, and percent change per annum. The user will then select either to simulate all steps at once or run step-by-step. The SimulationController class, via the runSimulation() class, passes the parameters over to the SimulationCT class to create a new simulation memento. The SimulationCT class acts as the memento caretaker and manages all Simulations of the data. Simulation uses the user selected MarketSimulationStrategy in order to Simulate the data. The data used by the Simulation is either the currentPortfolio Equities if no other Simulation mementos exist, or the values from the last memento created. The results are then received by the SimulationController class, via the updateSimulation() method, which are then displayed to the user in Simulation tab in the GUI. The user is then able to move forwards and backwards through the steps of current and past Simulations with buttons. They can also press buttons to jump straight to the first Simulation or the last Simulation. From here, the user can either run another Simulation of the data, delete the last Simulation created, or end the entire process and clear the Simulations.



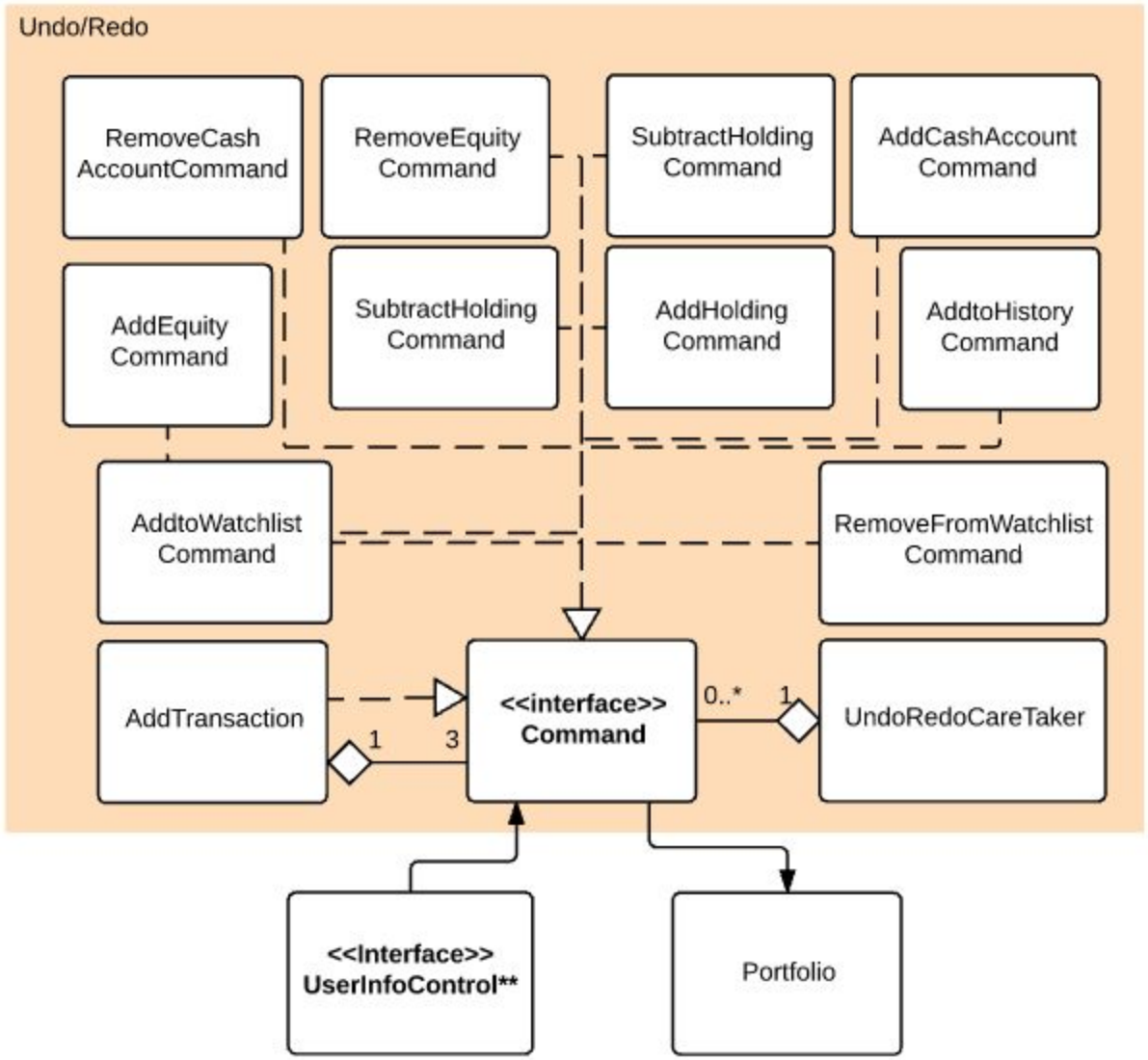
Simulation: Design Pattern

Name: Simulation Strategy		**GoF pattern: Strategy**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Simulation	Context	This will hold a reference to a simulation strategy object.
MarketSImulationStrategy	Strategy	Indicates to all strategies that they must take in a portfolio (and some percentage?).
BearMarketStrategy	Concrete Strategy	Decreases the value based on the percentage and the transaction history.
BullMarketStrategy	Concrete Strategy	Increases the value based on the percentage and the transaction history.
NoGrowthStrategy	Concrete Strategy	Returns the same as the input
Deviations from the standard pattern: Pulls information from a memento object		
Requirements being covered: The system shall demonstrate how the portfolio will perform over time by allowing the user to specify a time period and market simulation algorithm to use. The system shall allow the user to indicate the number of time steps to move forward, and the time interval for each step. The allowed time intervals are one day, one month, and one year. The user shall be able to specify running the simulation step-by-step, or without any pauses to the end. The user can select a no-growth-market simulation algorithm in which the equity prices never change. The user can select a bull-market simulation algorithm. With this simulation, the user shall indicate a per annum percentage that each equity will increase. The user can select a bear-market simulation algorithm. With this simulation, the user shall indicate a per annum percentage that each equity will decrease At the end of a simulation, the system shall show the portfolio value at that point.		

Name: Simulation Memento		**GoF pattern: Memento**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Simulation	Originator	Pulls all of the equites from a portfolio into an arraylist.
CurrentMemento	Memento	A representation of the portfolio as an arraylist of equities. Exists as a field within the simulation class
SimulationCT	Caretaker	A stack data structure that exists within the simulation class. Contains all of the arraylists of already simulated equities.
Deviations from the standard pattern: Our Memento Pattern deviates from the standard pattern in that it is contained entirely within the Simulation class, otherwise the pattern follows the standard.		
Requirements being covered: The system will allow the user to run another simulation from this point, or reset the equity prices to their current prices or the prices at the start of the simulation that just finished.		

Subsystem #6 – Undo/Redo

The Undo/Redo subsytem uses an array of design patterns in order to achieve its desired functionality. It uses the command/composite pattern to make command objects and macro command objects. These objects can be executed or unexecuted. The method signatures for those methods are defined in the command interface. The macro command, transactionCommand, is an aggregate of three commands which act as leaf commands. It uses these commands to achieve its larger task. The Undo/Redo caretaker is not a full meneto pattern but it uses the concept of having a caretaker class from the memento pattern. It holds the command objects so they may be undone or redone at any point in the system. The Undo/Redo caretaker uses the singleton pattern to ensure that only one instance of it exists in the system so that commands are always stored in one location.



Undo/Redo: Design Pattern

Name: Undo/Redo		**GoF pattern: Command**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Command	Command	Declares an interface for users performing various tasks on the contents of their portfolio
AddEquityCommand(all other Concrete Comands)	ConcreteCommand	Implements the Execute command that is defined in the Command interface by invoking functions defined in the Portfolio class
FPTSRunner	Client	This is the main application that allows the ConcreteCommand to be instantiated. The receiver is set upon instantiation
HomeController(all BuySellContr oller and CashAccount Controller	Invoker	Where the execute operation is called. Carried out when a
Portfolio	Receiver	Has the underlying implementation associated with carrying out the Command. This is done in order to maintain separation of concerns and cohesion within our classes.
NOTE: This is not all of the ConcreteCommand objects that exists in our system, but they are not all listed here because there is a significant number of them and listing them here would decrease the readability of this table.		
Deviations from the standard pattern: The receiver does not have an Action operation. Instead it all of the actions needed to update the contents, and/or the information related to the contents of a portfolio.		
Requirements being covered: Requirement 3E - The system shall provide undo and redo capability to the user. The user shall be able to undo at least the last five portfolio modifications that were performed without an intervening operation that is not undoable.		

Name: Undo/Redo		**GoF pattern: Memento**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Command	Memento	Commands are the memento. Their state will be so that their affects can be undone at later point in the program if such is requested by the user. It does not pass itself to other objects since commands can only be used to either perform or undo tasks, nothing else.
UndoRedo Caretaker	Caretaker	An aggregation of Command objects. Contains two stacks, the undo stack and the redo stack. All commands that are executed are pushed to the undo stack. There you can pop off the top of the stack(the most recently performed command) and un-execute it. All undone commands are sent to the redo stack. There you can perform the same sequence in order to re-execute a command
Deviations from the standard pattern: This pattern has no Originator participant because Command objects are not objects to be used to store another existing object's state. They hold methods that perform operations that are defined in Receiver objects and those methods update the Receiver's state and any other participating objects.		
Requirements being covered: Requirement 3E - The system shall provide undo and redo capability to the user. The user shall be able to undo at least the last five portfolio modifications that were performed without an intervening operation that is not undoable.		

Name: Undo/Redo		**GoF pattern: Composite**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Command	Component	The interface defined for these
AddToHistory Command	Leaf	A ConcreteCommand that has no child Commands. This will add the transaction to the history
Subtract Holding Command	Leaf	A ConcreteCommand that has no child Commands. This will subtract value from the appropriate holding.
AddHolding Command	Leaf	A ConcreteCommand that has no child Commands. This will add value to the appropriate holding.
Transaction Command	Composite	A ConcreteCommand that has the three above objects has its leaf components. When this Command is executed and/or un-executed the corresponding operations are just called on its children in the appropriate sequence.
BuySell Controller	Client	This controller object is what calls methods when the user interacts with our buy/sell interface in the GUI. It will execute a TransactionCommand when the user performs one.
Deviations from the standard pattern: There are not many deviations from the standard pattern in this example. This pattern is being used to design a MacroCommand.		
Requirements being covered: Requirement 3E - The system shall provide undo and redo capability to the user. The user shall be able to undo at least the last five portfolio modifications that were performed without an intervening operation that is not undoable.		

<table>
<tr><td colspan="2">Name: Undo/Redo</td><td>GoF pattern: Singleton</td></tr>
<tr><td colspan="3">Participants</td></tr>
<tr><td>Class</td><td>Role in pattern</td><td>Participant's contribution in the context of the application</td></tr>
<tr><td>UndoRedo Caretaker</td><td>Singleton</td><td>This is a Singleton because we only want one/the same instance of the UndoRedoCaretaker class to be used for Command storage. If the commands are being stored in different instances of the UndoRedoCaretaker then there will be no way for us to know the order that the users performed the commands is.</td></tr>
<tr><td colspan="3">Deviations from the standard pattern: Specific to the GOF design patterns book there is no getter for the singleton data and there is no singleton data field. There are just operations to push and pop from the appropriate stacks(undo or redo stack), and there is a method to get the instance of the singleton.</td></tr>
<tr><td colspan="3">Requirements being covered: Requirement 3E - The system shall provide undo and redo capability to the user. The user shall be able to undo at least the last five portfolio modifications that were performed without an intervening operation that is not undoable.</td></tr>
</table>

Transition to R2

One of the major changes in R2 was getting real time stock values from yahoo. This had a large effect on our design. We use a class called EquityInfo to store Equity metadata (Company name, Ticker symbol, current share value, market averages). The value variable in EquityInfo stores the the current value of the equity share, and is updated regularly from a thread with a user defined interval between updates. As a side effect of making the value variable within EquityInfo much more important we improved encapsulation regarding EquityInfos. We restricted access to who could make them, removed the copy constructor, and restricted access to the static list of EquityInfos. Instead of getting the list, you can now only get an iterator over the list. This lets you access the values in the list, but not be able edit the list directly.

Status of the Implementation

For Release 2 of the Financial Portfolio Tracking System, mostly all requirements have been satisfied as per R2 rubric.

Incompleted Requirements:

- Our system does not prompt the user to save before they logout. To fix this we would have to add an alert box when the logout button is clicked. This alert box would have to have prompt text with yes or no buttons. On “no”, the box closes, on “yes”, there would have to be an event handler to save the portfolio. After a successful save, the user would be logged out.

Known Bugs

- Importing portfolio is not functional on Unix/Linux systems due to lack of wildcard support (the kleene star).

UML State Activity Diagram

*** See ***System Flow.pdf*** to view this diagram***

Appendix

This section provides fine-grained design details for all of the classes in your design.

Class: Portfolio	
Responsibilities: This class is responsible for holding user information. It keeps a list of the user's holdings, it keeps the a history of the user's transactions, and stores the user's login user-ID and password. It stores the password encrypted, and only decrypts when needed, immediately encrypting afterwards. It is also responsible for saving itself to a file. Can import outside portfolio.	
Collaborators	
Uses: Holding Transaction CashAccount Equity	**Used by:** LoginController LogoutParent SignUpController MarketSimulationStrategy
Author: R. Hartzell	

Class: Equity	
Responsibilities: This class is the representation of a quantity of an Equity that a user has bought. It uses an EquityInfo object to specify what equity it is (Company, ticker symbol, etc), and keeps track of how many shares the user owns, the date it was acquired, whether it was acquired using cash from a cash account, and the price that the user payed per share.	
Collaborators	
Uses: EquityInfo	**Used by:** Portfolio Transaction MarketSimulationStrategy Holding
Author: R. Hartzell	

Class: Cash Account	
Responsibilities: This class is a representation of a bank account that the user owns. It keeps track of the date the account was opened, the name of the account, and the money in the account.	
Collaborators	
Uses:	**Used by:** Portfolio Holding MarketSimulationStrategy
Author: R. Hartzell	

Class: Transaction	
Responsibilities: This class represents a Transaction where a Holding is lost and a Holding is gained. Partial losses are acceptable. We have methods to subtract two Holdings that are of the same type (That is to say, two cash accounts with the same name or two Equities with the same EquityInfo) that simply removes the loss from the existing Holding.	
Collaborators	
Uses: Holding	**Used by:** Portfolio HomeController
Author: R. Hartzell	

Class: Bond	
Responsibilities: This class is simply a subclass of Equity that adds no functionality, just a way to specify what kind of Equity it is.	
Collaborators	
Uses: Holding EquityInfo	**Used by:** Equity
Author: R. Hartzell	

Class: EquityInfo	
Responsibilities: EquityInfo is the class used to keep track of what an Equity can be. EquityInfo represents an actual company or corporation that you can buy shares of. It has a private constructor, so that it can only be created inside its own class. The only time it will ever be created is when we parse the formattedEquities.csv file for the list of equities we are supposed to persist. Once the formattedEquities.csv file is parsed, the EquityInfos created are kept privately, only able to be accessed by calling a search method with text fields that match (Name, tickersymbol, and a market index), or by being iterated over with an Iterator<EquityInfo> object obtained by the static method getEquityInfoIterator().	
Collaborators	
Uses: EquityInfoList	**Used by:** Equity HomeController
Author: R. Hartzell	

Class: MutualFund	
Responsibilities: This class is simply a subclass of Equity that adds no functionality, just a way to specify what kind of Equity it is.	
Collaborators	
Uses: EquityInfo	**Used by:** Equity
Author: K. Nussbaum	

Class: Stock	
Responsibilities: This class is simply a subclass of Equity that adds no functionality, just a way to specify what kind of Equity it is.	
Collaborators	
Uses: Holding EquityInfo	**Used by:** Equity
Author: K. Nussbaum	

Class: Market	
Responsibilities: This class is simply a subclass of Equity that adds no functionality, it simply represents an equity that is only described as being a market/sector	
Collaborators	
Uses: Portfolio MarketSimulationStrategy	**Used by:** Home Controller
Author: Daniil Vasin	

Class: BearMarketStrategy	
Responsibilities: Implements the MarketSimulationStrategy interface and simulates the value of the portfolio and all of the stocks contained within it in a market where values increase exponentially. Takes in a portfolio copy (an arraylist of equities) and an amount of time over which the simulation is executed. It then returns a copy of the array list but modified according to the simulation**.**	
Collaborators	
Uses: MarketSimulationStrategy Equity	**Used by:** Home controller
Author: W. Estey	

Class: BullMarketStrategy	
Responsibilities: Implements the MarketSimulationStrategy interface and simulates the value of the portfolio and all of the stocks contained within it in a market where values increase exponentially. Takes in a portfolio copy (an arraylist of equities) and an amount of time over which the simulation is executed. It then returns a copy of the array list but modified according to the simulation**.**	
Collaborators	
Uses: MarketSimulationStrategy Equity	**Used by:** Home controller
Author: W. Estey	

Class: NoGrowthMarketStrategy	
Responsibilities: Implements the MarketSimulationStrategy interface and simulates the value of the portfolio and all of the stocks contained within it in a market where values increase exponentially. Takes in a portfolio copy (an arraylist of equities) and an amount of time. It then returns a copy of the array list of equities**.**	
Collaborators	
Uses: MarketSimulationStrategy Equity	**Used by:** Home controller
Author: W. Estey	

Class: HomeController	
Responsibilities: Responsible for loading all of the tabs and displaying the aggregate values for the Cash Accounts and User owned Equities. Also displays a Table of the user's transaction history	
Collaborators	
Uses: FPTSRunner Portfolio CashAccountController UserEquityController BuySellController UndoRedoCaretaker LogoutParent SetIntervalController	**Used by:** EquitySearchController BuySellController CashAccountController UserEquityController WatchlistController SetIntervalController AddEquityController
Author: Philip Bedward	

Class: LogoutParent	
Responsibilities: This class is responsible for providing a logout button for classes that extend it. It provides a logout prompt as well, ensuring that the user did not press the logout button accidentally. Once logged out, it changes the scene back to the login window.	
Collaborators	
Uses: Portfolio	**Used by:** HomeController
Author: P. Bedward	

Class: SignUpController	
Responsibilities: This class is responsible for managing the sign up scene. It listens for the register button being pushed, and creates a new account accordingly. It listens for a cancel button being pushed and switches back to the login scene.	
Collaborators	
Uses: Portfolio FPTSRunner	**Used by:** FPTSRunner
Author: P. Bedward	

Class: LoginController	
Responsibilities: This class is responsible for listening to the login scene for 2 button pushes: the login button, and the sign up button. When the login button is pressed, it attempts to load a profile with the information entered in the text boxes. When the sign up button is pressed, it switches the scene to the sign up scene.	
Collaborators	
Uses: FPTSRunner Portfolio	**Used by:** FPTSRunner
Author: P. Bedward	

Class: Simulation	
Responsibilities: This class is responsible for handling simulation operations. When the user presses the button to run a simulation based on user specified MarketSimulationStrategy, type of step, number of steps, percentage to adjust by, and whether or not to run all steps at once or step-by-step. From here the user can either run another simulation, cancel the current simulation, or end all simulations.	
Collaborators	
Uses: Portfolio MarketSimulationStrategy	**Used by:** Home Controller
Author: W. Estey	

Class: EquityInfoList	
Responsibilities: This class is responsible for managing a list of EquityInfos. This class is only used in one place; the static list of EquityInfos contained in EquityInfo. Because this is the only place it is used, we made it a private internal class of EquityInfo. It uses a private internal class that implements iterator to let the contents be seen without allowing modification of the internal list.	
Collaborators	
Uses: EquityInfoListIterator Container	**Used by:** EquityInfo
Author: R. Hartzell	

Class: EquityInfoListIterator	
Responsibilities: This class is an Iterator. It is a private inner class of EquityInfoList. It iterates over an ArrayList inside EquityInfoList. It is primarily used so that anyone can access the contents of the list inside EquityInfoList without being able to modify the list.	
Collaborators	
Uses: Iterator	**Used by:** EquityInfoList EquityInfo

Author: R. Hartzell

Class: WatchedEquity	
Responsibilities: This class is a subclass of Equity that represents Equities that are being stored in the watchlist	
Collaborators	
Uses: EquityInfo	**Used by:** WatchlistController
Author: Daniil Vasin	

Class: BuySellController	
Responsibilities: This class is responsible for all Transactions (Buy/Sell) done by the user. This class also updates all graphs, charts, and all types of visual data pertaining to components of a transactions (equities and cash accounts).	
Collaborators	
Uses: FPTSRunner Portfolio HomeController CashAccountController	**Used by:** FPTSRunner
Author: P. Bedward	

Class: SimulationCT
Responsibilities: This class is responsible for handling simulation operations and maintaining all Simulation mementos. When the user presses the button to run a simulation based on user specified MarketSimulationStrategy, type of step, number of steps, percentage to adjust by, and whether or not to run all steps at once or step-by-step, a new Simulation is created and maintained by this class. From here the user can either run another simulation, cancel the current simulation, or end all simulations. They have the ability to step through each step of Simulations through the use of the memento pattern.
Collaborators

Uses: Simulation	**Used by:** Home Controller
Author: W. Estey	

Class: EquitySearchController	
Responsibilities: This class is responsible for actions that pertain to searching for equities within all equities available throughout the portfolio.	
Collaborators	
Uses: FPTSRunner Portfolio BuySellController HomeController WatchlistController	**Used by:** FPTSRunner
Author: P. Bedward	

Class: CashAccountController	
Responsibilities: This class is responsible for actions that pertain to a operations that can be done to cash accounts	
Collaborators	
Uses: FPTSRunner Portfolio BuySellController HomeController	**Used by:** FPTSRunner
Author: P. Bedward	

Class: WatchlistController	
Responsibilities: This class is responsible for actions that pertain to the management of the Portfolio’s watchlist	
Collaborators	
Uses: FPTSRunner	**Used by:** FPTSRunner

Portfolio HomeController EquitySearchController	
Author: Daniil Vasin	

Class: AddEquityController	
Responsibilities: Responsible for performing the tasks to add an equity to the portfolio, when the user clicks the ‘add’ button and selects the amount of shares that they want to add to the portfolio. The appropriate Command is carried out, executed and stored in the UndoRedoCaretaker.	
Collaborators	
Uses: EquitySearchController HomeController AddEquityCommand	**Used by:** EquitySearchController
Author: Philip Bedward	

Class: UserEquityController	
Responsibilities: Responsible for performing the tasks to search for user owned equities and to remove them from the system. The reason that this is separate from the AddEquityController because the other controller takes care of equities that are do not yet exist in the system. However, this controller handles actions performed on equities that the users already own.	
Collaborators	
Uses: EquitySearchController HomeController BuySellController UndoRedoCaretaker WatchlistController	**Used by:** BuySellController EquitySearchController
Author: Philip Bedward	

Class: SimulationController	
Responsibilities: Responsible for performing tasks within the simulationTab. Takes the user input and communicates to SimulationCT to update the model.	
Collaborators	
Uses: SimulationCT	**Used by:** HomeController
Author: W. Estey	

Class: AggregateVisitor	
Responsibilities: This class is responsible for actions that pertain to operations to aggregate the values of all of the Holdings in a user's portfolio	
Collaborators	
Uses: HomeController Equity CashAccount Market	**Used by:** FPTSRunner
Author: P. Bedward	

Class: RemoveCashAccountCommand	
Responsibilities: Responsible for the performing the task that removes a cash account from the portfolio, and also undos that command	
Collaborators	
Uses: Portfolio CashAccount UndoRedoCaretaker	**Used by:** CashAccountController
Author: P. Bedward	

Class: AddCashAccountCommand	
Responsibilities: Responsible for the performing the task that creates a cash account and adds it to the portfolio, and also undos that command	
Collaborators	
Uses: Portfolio CashAccount UndoRedoCaretaker	**Used by:** CashAccountController
Author: P. Bedward	

Class: AddEquityCommand	
Responsibilities: Responsible for performing the task that creates an equity and adds it to the portfolio, and also undos that command	
Collaborators	
Uses: Portfolio Equity UndoRedoCaretaker	**Used by:** HomeController
Author: P. Bedward	

Class: RemoveEquityCommand	
Responsibilities: Responsible for performing the task that removes an equity from the portfolio, and also undos that command	
Collaborators	
Uses: Portfolio Equity UndoRedoCaretaker	**Used by:** HomeController
Author: P. Bedward	

Class: AddHoldingCommand	
Responsibilities: Responsible for performing the task that represents one part of a transaction. This task will do one of three things. 1) Add value to a cash account(if something is being sold). 2)Add a brand new equity to the portfolio if it does not exist (if something is being bought). 3) Add shares to an existing equity (if something is being bought). Includes the capability to undo this task	
Collaborators	
Uses: Portfolio Holding	**Used by:** TransactionCommand
Author: P. Bedward	

Class: SubtractHoldingCommand	
Responsibilities: Responsible for performing the task that represents one part of a transaction. This task will do one of three things. 1) Subtract value to a cash account(if something is being bought). 2) Remove an existing equity from the portfolio if that number of shares being purchased is equal to the number of shares that the user owns. 3) Subtract the number of shares from an existing equity if the number shares being purchased is less than the number of shares that the user owns. Includes the capability to undo this task	
Collaborators	
Uses: Portfolio Holding	**Used by:** TransactionCommand
Author: P. Bedward	

‘

Class: AddToWatchlistCommand	
Responsibilities: Responsible for performing the task that adds an Equity to a watchlist Includes the capability to undo this task	
Collaborators	
Uses: Portfolio EquityInfo WatchedEquity	**Used by:** WatchlistController
Author: Daniil Vasin	

Class: RemoveFromWatchlistCommand	
Responsibilities: Responsible for performing the task that removes an Equity from a watchlist Includes the capability to undo this task	
Collaborators	
Uses: Portfolio EquityInfo WatchedEquity	**Used by:** WatchlistController
Author: Daniil Vasin	

Class: AddToHistoryCommand	
Responsibilities: Responsible for performing the task that will add a completed transaction to the transaction history.	
Collaborators	
Uses: Portfolio Holding Transaction	**Used by:** TransactionCommand
Author: P. Bedward	

Class: TransactionCommand	
Responsibilities: This class is responsible for carrying out an entire transaction by using the three command classes listed below in the ‘Collaborators’ section. If the transaction cannot be performed, an alert is generated.	
Collaborators	
Uses: Portfolio Holding AddHoldingCommand SubtractHoldingCommand AddToHistoryCommand UndoRedoCaretaker	**Used by:** BuySellController
Author: P. Bedward	

Class: UndoRedoCaretaker	
Responsibilities: This class is responsible for storing the commands that have been performed and undoed, so that they may be accessible for undoing and redoing upon the user’s request.	
Collaborators	
Uses: N/A	**Used by:** BuySellController HomeController CashAccountController TransactionCommand AddEquityCommand RemoveEquityCommand AddCashAccountCommand RemoveCashAccountCommand
Author: P. Bedward	