CE303 Report - Jason English 1404665

Instructions

In order to connect to the game server, you must use PuTTY, with the host address of "localhost", port number 8888 and a raw connection. Once you have connected to the server you must specify the number of players who will play the game. Once the game has started there are a series of commands you can use to play the game.

Command	Stock option	Modifier	Effect	Notes
Buy	Apple, BP, Cisco, Dell, Ericsson	1 – total amount user can afford	Will buy x amount of stock, as long as user can afford it.	Will cost the amount of the stock plus an additional cost of 3
Sell	Apple, BP, Cisco, Dell, Ericsson	1 – total amount user has	Will sell x amount of a stock at the rate displayed by the game	
Vote	Apple, BP, Cisco, Dell, Ericsson	Yes, No	Vote to use (yes) or discard (no) a card displayed	Any cards with a +ve amount of votes wil be played. -ve votes will be destroyed. Cards with no cumulative votes will be left for the next round.
Done			Will skip all remaining turns	

Players get a total of 2 commands (Buy, Sell, Done) and 2 votes per round, players can use commands as they please as long as they can provide the cost or stock required for the command. A player can vote twice in total but only once on a specific stock each round. Once 5 rounds have occurred the game will briefly display the results before closing. The final score is equal to the total amount of money a player has after all of the stocks are sold in round 5.

Program Architecture and Socket communication

The program uses a series of classes to communicate between the various parts of the game logic and the data stored on the server.

The Card class defines a comparable card class, this is done so that the card object can be defines with the necessary attributes. The Deck object builds a deck of card objects, along with the corresponding stock the deck was built for. By having a specific container for the cards while denoting the stock it acts on it allows for relevant information for the cards to be accessible where necessary.

The Constant class acts as a set of information that can be defined about the program, this is used to set the IDs of the users that join the game. This information helps keep the right number of players in the game.

The Game class initializes the game state and contains all of the logical operations needed for the game to work. The main components of the game are created here in the constructor. The body contains the methods which are used to update the state of the game, the attributes are kept static to stop multiple instances of the game state running concurrently when multiple users connect. By keeping the values static, it removes the chance of multiple versions of important game data being created. Getter methods are used to access the private methods of the class. The information is then passed to another class to be displayed in the GUI. Some methods within the class are synchronized, this is due to the functionality of the program. The multiple threads of the users playing the game will interact with the methods multiple times, to stop this causing differences across the clients the synchronization is used.

The Player class is used to define and construct the player object used within the Game class, the separation of these objects allows for independent constructions of the object. Methods specific to Player are separated from Game, where they would not be useful and could cause un-wanted interactions between they classes. The Stock class creates a stock object which relies on ENUMs for construction. This allows for a reference for what stocks are available within the game. The Test class creates an environment to test specific methods throughout the program. The test environment checked expected results against outputs generated from the program itself with test data.

The Socket classes are used to communicate between the front end of the program and the server and socket services that update the game server. Classes request and request type are used to define the actions a player can take on the server interface. The requests form the method call that will update the state of the game based on the action being performed.

GameServer is used to set up the connection to the server socket and link the program to the server you are attempting to host the program on. GameService details all the interactions between the server and the game. It contains the methods which take a request and create method call for the game client. The Game uses PuTTy to host a server on the local host. The main method of communication between the program and the server are through param and request type statements. A request takes the information from the input on the server and defines the parameters to be passed to the program in a way that is useful to the methods it contains.

Concurrency

The main steps used to maintain the correct operation in the face of concurrency is the used of static objects, guarantying only a single instance of important objects used within the games code. This allows for many functions to be called without worrying about grabbing a specific instance of the object. The game class makes used on an atomic integer to track the number of rounds over the course of the game, this allows for the value to be updated without having to worry about multiple instances attempting to increment at various points. There is limited use of synchronized threads, they either focus on a block of code that will be used multiple times by various threads or specific object which is operated receive multiple operations from various threads at the same time.

Unit tests

There are three unit-tests which test the use of the buy and sell methods of the game interface. These tests successfully compile and produce successful test results. The setup of the classes made for a difficult implementation for the test class, ideally more difficult and rigorous testing methods would be use. The tests classes that currently exist give a reasonable test of the buy and sell methods used by multiple players. Due to time constraints there is unfortunately no test class to explore the functionally of the voting and vote execution methods, these methods would be worth testing extensively to ensure only reasonable commands occur. This would have allowed for better testing of the synchronized methods outside of observing the operation of the game on the server.

User Interface

The output of the information on the user interface is intended to allow for easier readability for the user. The spreading of information with sub-headings and delimiters between different value types improve interface greatly. A tabular format might have been useful as information such as stock or player is shared across multiple values, a table would remove repetitive data. Error messages are posted with information intended to help the user understand the specific error with a command. Messages are also posted to update with the status of the command just issued. The error checking on the inputs is quite strict as to stop unintentional commands being issued. Incorrect commands are not counted as actions so the player gets the opportunity to retry the action they intended to do. When and extra player attempts to join they get a message showing the inability to join a full game.

A message tells the user which player they are logged on as to allow them to know which player profile is theirs, giving this information stop confusion between which player is which as users are not expected to remember in what order players connected. A message is given when users attempt to vote on the same stock twice, informing them that they are attempting to do something the game rules do not allow.

Project Review

Over the progress of the project was good, after spending time to learn the operation of the provided source code the follow inclusions to the program became more obvious. The logical operations of the commands and connections were straightforward, although complex object times for classes caused difficulty in readability of operations causing harder to solve semantic issues in the code. Once the request functions were fully understood their inclusion to the program was straightforward. Connecting to the PuTTy server proved easy.

Debugging issues that occurred after a second user connected to the server were frustrating due to the debugger not following the second connection in any regard. This lack of information caused small problems being hard to track and solve. Attempting to implement web services proved most difficult due to a lack of experience with their operations compounded with the lack of time near the end of the assignment. The efficiency of the code could be improved in certain areas along with a more complete set of unit tests. I am proud of the handling the multiple threads through important sections of code and the param requests translating server commands to method calls. Give another attempt at the project more time would be focused on implementing and understanding web services as I wanted to create a well-designed bot, but due to time miss-management early in the project it didn't leave enough time to explore that area of the program.