## Run Game

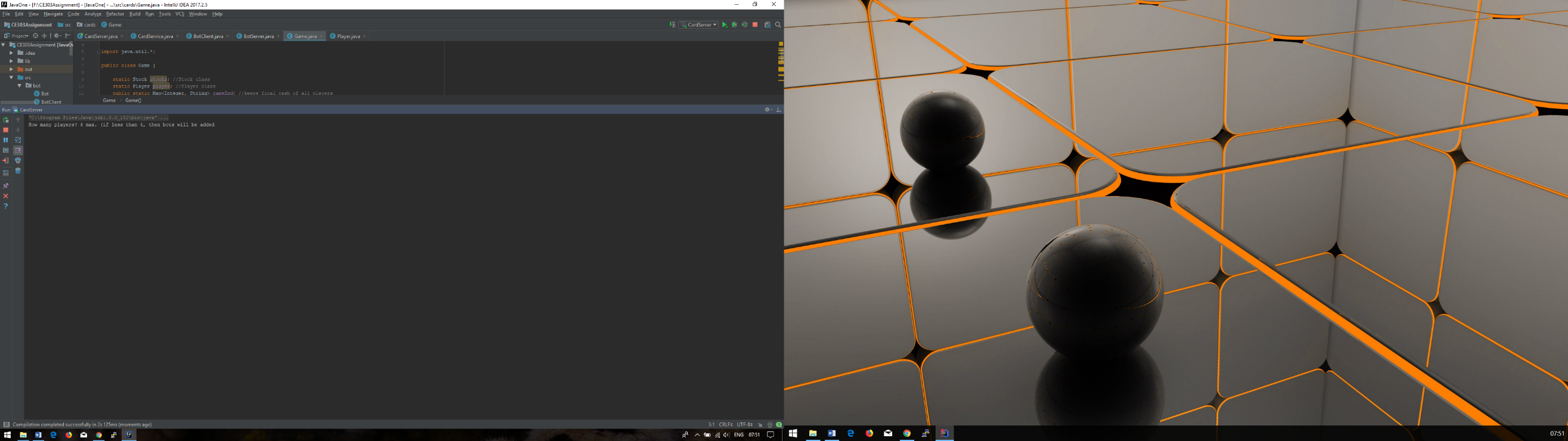


Figure Console Player input

To run the game:

* Run the CardServer, which will open a console where the number of players who wishes to play is asked. See figure 1 for reference.
  + The game allows maximum of 4 players or minimum of 1 players to play the game. Every game must have a total of 4 players to play the game.
  + Missing players will be filled in using Bots. Maximum of 3 Bots can be created as at least one player is required to execute end of round.



Figure PuTTY client

* If there was less than 4 players playing, please wait few seconds for the Tomcat server to start. Each game has 5 rounds and at the end the winner will be announced.
* If player responded anywhere between 1-4 players are going to play the game, they may start a PuTTY [1]service (figure 2), and in “Host Name (or IP address)” box enter “localhost” and in “Port” type “8888” and choose “Raw” connection type. You may save the session for future quick access by giving a filling in “Saved Sessions” and clicking on “Save”. In future, the players can select the saved game to start socket quickly. Afterwards you may click on “Open” to start the game. All other players must do this. Any more players than 4, will lose connection when they try to connect to the socket server.

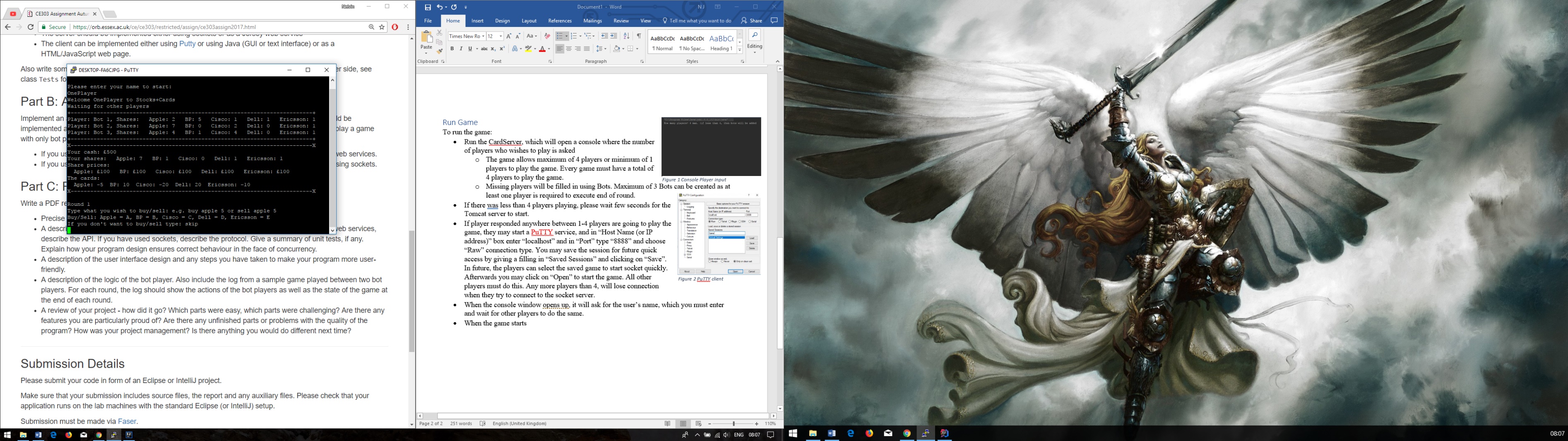


Figure Game details

* When the console window opens, it will ask for the user’s name, which you must enter and wait for other players to do the same.
* The game will start after all the players have started their PuTTY client and have entered their game in. At the start of game (figure 3), all the users will be shown other players name and their shares at start of game. Followed by yours cash, shares, share prices, and the cards. Every player see this same detailed information.

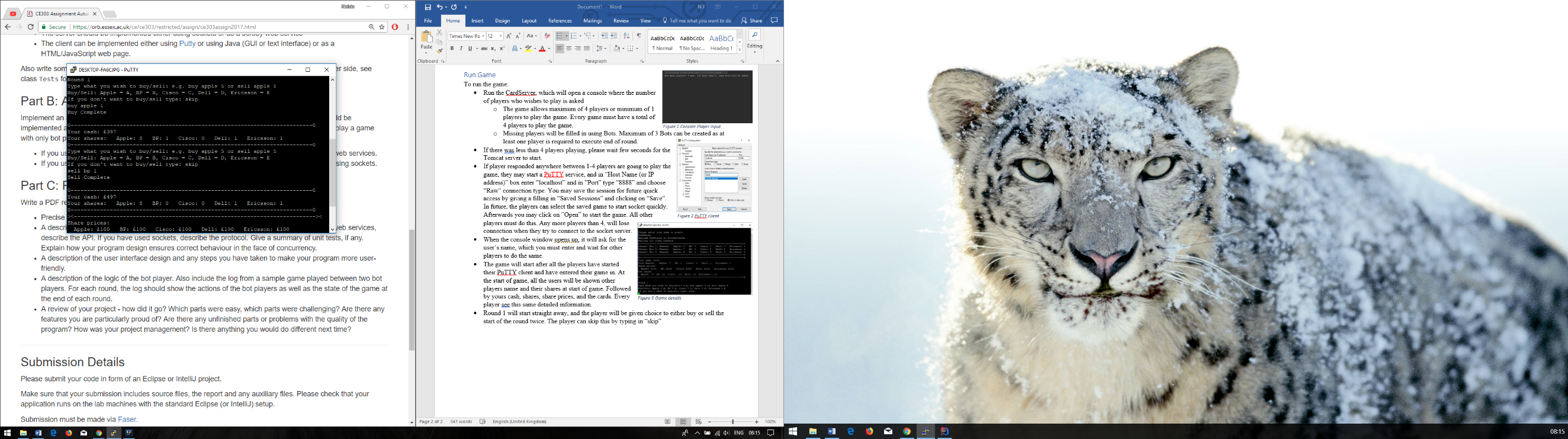


Figure Round buy/sell

* Round 1 will start straight away, and the player will be given choice to either buy or sell the start of the round twice. The player can skip this by typing in “skip”. To buy from any stocks, type in “buy stock amount”, e.g. “buy apple 1” as shown in figure 4 and to sell a share, type “sell stock amount”, e.g. “sell bp 1”. If you do not have enough cash, you will be unable to buy the stock share requested and if you do not have the share to sell, it will be denied.
* After buy/sell has been used twice, you will be asked to vote twice, as shown in figure 5, you may vote by typing in “stock yes” or “stock no”. This must be for two different companies. After voting, the votes will be executed and the next round will begin.

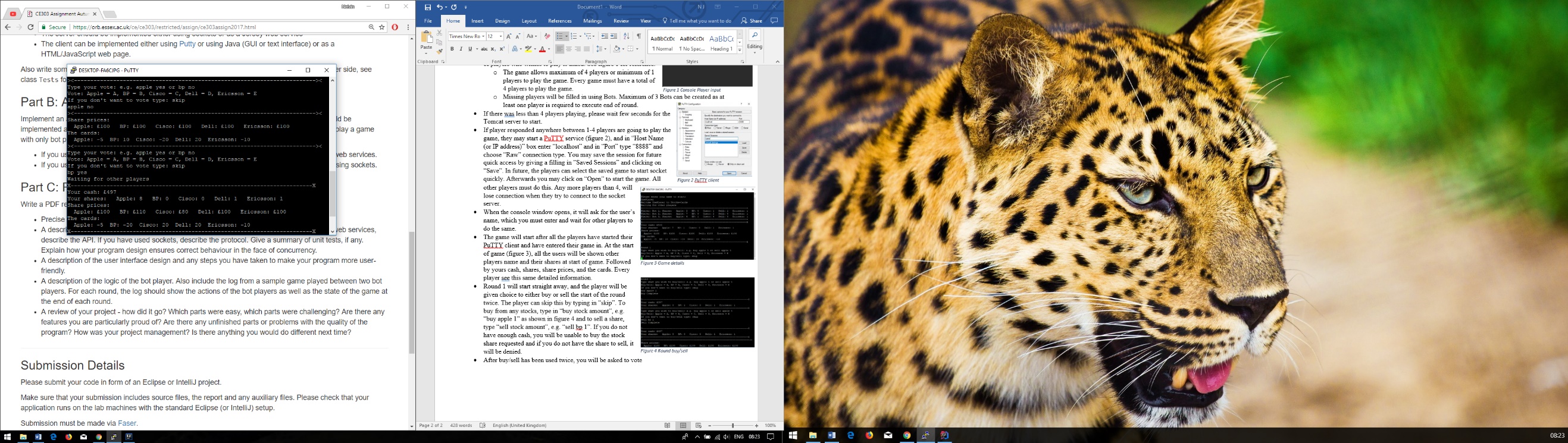


Figure Vote

* At the end of the game, every player total cash from selling their shares for the share price along with their remaining cash and their rank depending on their total cash will be shown.

## Program architecture

### Web Services

Jersey web services were used for bots within the game. The web service run off Tomcat client, which is connected to port 8080 and is given path “/bot” to BotServer class. WebTarget is used to create a path for accessing the server from the client. The Game object was shared with web service to access the same objects as the socket, which made it possible for the bot to communicate with the socket in concurrency. The BotServer class acted as a JSON application, as within the class there were methods that had a path to either GET or POST depending on what the user called. When the user called a GET method, for example, getShare(id), it retrieved Player.shares[id] from the Player class, which held each players shares or using the POST method, to update the game object, for example buyStock(id, stockid, amount), a form was used to send the data from client to server due to three parameters were needed to pass to Game buy( ) method and this was passed into the Game buy, which executed the buy for purchasing shares.

The web service communicates with same game objects as sockets, so every update by either services would affect the game. In my BotTest, one of the first tests I did to test if I was connected to same Game object as socket was retrieving the cash for first player, which was £500, afterwards I used the sell post web service command to sell one apple stock for the first player, and checked the cash afterwards, which should be £600 and I was able to retrieve £600. This meant I should be careful with bot, as giving it wrong id can affect the socket player games. So I created a method for creating and storing the bots id, to only execute bots game objects.

Every bot actions were tested, to ensure web service communicated with the game object as intended. Concurrency for web is maintained by the socket giving response to execute the bot methods.

When the game begins the user is asked how many players are going to be playing and if its less than 4, a thread for Tomcat server and another thread for Bot is created. However, if 4 players are going to playing, the bot and tomcat threads are not created. The Tomcat thread is used for executing the tomcat and the Bot thread starts up the bot, and waits for a response from the socket. To make concurrency possible between the socket and web service, I use the first user on socket to communicate with web service by giving the responses to initialise an action from the bot. The bot thread already has a set of commands setup such as creating the bots id, which is very important so the bots are seen as a player.

### Socket

The socket uses “localhost” as IP address for time being and uses port 8888 to communicate. When the game starts, the number of players is asked, which is used to set how many threads will be using the CyclicBarrier, which is responsible for ensuring concurrency works between each player. Each socket has a thread created for it and each run() are synchronised for preventing thread interference and memory errors. The run executed few methods, which makes up the socket game from start till end. The socket is responsible for I/O operations as a Scanner is used to obtain input from the socket console and PrintWriter is used to print to the socket console. After some socket method finishes, such as the login() and playGame(), the CyclicBarrier is called to await(), so the other players also reaches the same position before all the players go on to next round within the game. Thread.sleep() is used before updating to see the bots actions or executing vote() as only one player is allowed to execute the votes to prevent incrementing the votes each execution.

The socket uses a while loop to commence each round. To make sure only one thread executes each rounds final votes and the final total cash, I store the user id at the start of the game, meaning each thread will hold a specific number, that is used throughout the whole game to ensure the whatever the user do, they do not have to type in their id to make commands. This meant I could say if user is 1, then execute the votes and game scores and bots may play. However, this also meant I would always need 1 player to play the game, as you would not progress from first round unless the first player existed.

I implemented few tests for the socket, such as all the main features where tested to ensure they work properly on the socket accordingly. Such as from creating the players, to retrieving the player cash, to buying, and selling shares and voting. I tested executing votes, to check if the cards also changed after votes. The tests were to check if my actions were reflected in socket, such as if I purchased a stock, will that update the shares the player held. All test received pass.

## User Interface

The design is quite simple as shown on figures 1, 3, 4 and 5. The socket game runs on a console interface, so I did my best to improve the layout of information such as I print out lines before and after details the player should know before making decisions to buy, sell and vote so separate it from the rest of the test. Each text is printed in new line to spread out the information and to reduce confusing the player. From my experience playing the game, I decided it was important the player knew their shares and cards when voting or their shares and cash when buying or selling. After each round the updated cards, and share prices are shown for the user to make use of. When the game ends, I print out all the players ranking in the end after sell all the shares and the first player says winner.

## Bots logic

The bots are only created if there are less than 4 players available to play the game. When the bot object is created, it waits until a Boolean response is given by user id 1. The cyclic barrier plays an important role as depending on the given number, if its less than 4, then when the game starts, there is await() before the if statement for user 1 that sends response to bot to begin, that waits for all the player to log into the game, and if there was less than 4 players.

Then a response is given by user 1, to execute botBegin(). Which would create the number of bots that fill up to 4 players, meaning if there are 2 players, 2 bots are needed to fill to get 4 player requirement for the whole game. The bots id are stored for further bot methods. For each round, the bot waits for a response from user 1, to execute the bot round.

When response is given. The bot round will be executed, where the bot id is passed to the method, where a random number between 0-5 is called four times and stored in a array. These numbers represent the company id. Then a buy method is executed using the bot id, and the first random number, to purchase 1 share. Then sell method is executed using the bot id, and second random number, to sell 1 share. However, if the bot doesn’t have any shares in that particular company, then no sell is made. Afterwards the third random value is used to make a yes vote and fourth random value to make a no vote. Though if the fourth value is same as third, then the no vote is not executed. The bot will not execute final votes, as only user 1 is responsible for total votes. Afterwards, the bot waits for the user to commence the next round. The bot log of two bots play is called ServerLog and should be in the same directory as the report.