Development Testing Draft

To produce a high-quality product for the client, a rigorous testing of our solution was put at a high priority. Development testing is vital in guaranteeing a reliable product as it discovers any hidden bugs or defects causing our solution to not meet our requirements. We will be using the built-in test class created by our project by springboot to hold all our Junit units tests.

We will be using Test-Driven development which involves writing tests which initially fail before creating the code to make the tests pass. This process allows better program design by making us focus on one aspect at a time, planning how will be integrated into the project before implementation. It also saves a lot of time in the later refactoring and debugging process as the project develops. Detailed unit tests will be made for any major functions, ensuring they pass before committing to the main. We want complete and comprehensive test cases which balance coverage and practicality. We will do this by ensuring common, unusual, and marginal cases are all covered.

The core functions we will be testing here are the functions addNumberOfStonk() and reduceNumberOfStonks(). These functions have been chosen because when considering trading buying and selling stocks are the main aspect of the process. The player is likely to be calling these functions the most and the game is developed around the fact stocks can be bought and sold to create profit. In addition many other functionalities of the program, such as shorting and covering stocks, require these functions making it vital that these aspects work correctly. There is also a wide range of different player scenarios in which there needs to be tests for to ensure no errors.

What each function should do:

* addNumberOfStonk(): Take the desired stock and number of stocks as parameters then give the player the desired number of stocks, reducing their money depending on the current stock price. If the player does not have enough money or the stock does not exist it should not allow a purchase.
* reduceNumberOfStonks(): Take the desired stock and amount to sell as parameters. It should then remove the desired number of stocks from the player, increasing their wealth depending on the value of the stock at the time. If the player does not own enough stocks or the player does not own the stock, it should not allow it to sell anything

For ease of explanation consider stocks with the following prices: {Apple=2400,Nokia=450,AMC=250}

Below are the Tests for the addNumberOfStonk() function:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Player Wealth | Data Used  (Function Parameters) | Expected  (Wealth) | Expected (Owned Stonks) |
| New player with enough money buying a single stonk | 10,000 | (“Apple”, 1) | 7,600 | [{Apple,1}] |
| New player with enough money trying to buy 0 stonks | 10,000 | (“AMC”, 0) | 10,000 | [] |
| New player with enough money buying multiple stonks | 10,000 | (“Nokia”,5) | 10,000 | [{Nokia,5}] |
| New player with insufficient funds trying to buy a stonk | 1,000 | (“Apple”, 1) | 1,000 | [] |
| New player buying a stonk that does not exist | 10,000 | (“Fake Stonk”, 1) | 10,000 | [] |
| Player who already owns one Apple stonk buying another | 7,600 | (“Apple”, 1) | 5,200 | [{Apple,2}] |
| player buying a stonk costing their exact current wealth | 250 | (“AMC”, 1) | 0 | [{AMC,1}] |