**Brief on Quality Assurance**

**Acceptance Tests**

During this sprint our team hoped to improve upon our performance from the last sprint by tackling issues brought up during that sprint, one of which was the issue of acceptance testing. By using acceptance tests our team was able to split up user stories into easy to check tasks which when brought together would reflect each aspect of the user story. Previously we had been slightly vague with our acceptance tests and had only met with the client once during confirmation of these tests. During our sprint review with the client for the previous sprint some fine details were discovered to be not what the client has envisioned, specifically the font, UI layout, button size etc.

We resolved to ensure that the highest quality of our product was delivered this time, and thus we set about resolving the two major issues with the previous sprint’s acceptance tests. This time after gathering initial acceptance tests in discussion with the customer the team went back to refine and broaden these tests into fully fledged, specific and testable acceptance tests. After this process of refinement and brainstorming was completed we returned to the client with the new acceptance tests to talk them through the exact features for each story to be developed. This second meeting helped clarify points even more, and as both parties had more time to think on the project, more features pertaining to error handling were discovered. Again these new acceptance tests were further refined and broadened by the team, which at this point felt that the features of the user story were being exhausted.

These practices built upon lessons learned at the end of sprint one. By preforming acceptance test discussion and development in this fashion we strongly believe that this sprint’s deliverables are of a much higher quality and truly reflect the client’s vision.

**Refactoring**

Another new feature of this sprint was the extensive use of refactoring in order to adapt our code structure more fluidly and quickly to any changing needs of the client. After looking through materials given out through the lecture it was decided that one person take on the role of refactoring all the code in order to maintain a single style of refactoring. That style was determined by looking through online Open Source code which by its very nature has to be extremely friendly and intuitive to all parties involved. Refactoring was essential for the unit testing which occurred in our code, it helped provide a solid understanding of where other team members had gotten to and there process for getting there, incidentally this incremental refactoring effort identified relic code before unit testing for code coverage, which was swiftly removed. By providing a clearer foundation into the code base through changing method and variable names, removing dead code, and commenting in useful places like at the end of any statement, the team was able to proceed much swifter with fewer code duplicates for tasks being written due to the team work nature of the project. In these respects the code given to the client for their app will definitely have fewer bugs and provide a solid intellectual foundation for the next sprint.

It is important to note that during this refactoring process working code was left untouched, refactoring through our understanding should never change the outcome of the unit tests assigned to code.

**Unit Testing**

Through unit testing using djunit the objective of any project is to identify relic code for removal, determine errors which may occur in hypothetical circumstances which may indeed become reality in the future. Unit testing provides a basis for testing acceptable data passed to variables as well as providing a simulation of how your code will respond should invalid entries be passed to data which may not be logical ie. If a stock price is returned as a negative number, how will the code respond?

Our team used both black box testing and white box testing approaches during this sprint, white box when we already had working code and wanted to ensure that it was as error free as possible, and black box testing when developing new code to be implemented ie. We didn’t know what code to test upon and so the tests came first.

The software being used for unit testing did leave a lot of gaps which were uncovered during the team discussions on suitable testing; no android elements could be tested using the junits tests eg. What would happen if the client swiped a button instead of pressing it, would it still function? These had to be done manually, with the most common case being that an error message would be displayed should this case occur.

Through unit testing we were able to ensure that the client would be able to operate their application in the event that there would be no internet connection, no stock data, invalid stock data or incomplete stock data. Because these situations are taken care of the user experience is enhanced and thus the quality of our work could be assured.

**Metrics**

After preforming unit testing our group turned to the issue of using metrics to decrease the complexity of our code and attempt to increase code coverage by removing any (very few at this point) relic code. The code coverage percentage agreed by the group was …. The primary metrics used and agreed on by our team was to increase the maintainability Index of our code and decrease as much as possible the cyclomatic complexity (number of branches within the code) as well as the coupling between classes. By ensuring that our code was of low complexity and ensuring that each class has a unique responsibility we increased the re-usability and ability to grow our code over time.