Evaluation & Testing Report

Evaluation

Initially, before we began any work on this part of the project, we went through the previous team's requirements documentation and outlined any requirements which they had marked to be implemented but had not. This was so that we knew exactly what we had to implement, as to not waste time implementing things that were not necessary to the product brief. These requirements were then adapted to fit within our implementation: this is to reduce ambiguity in how a requirement may be interpreted.

For example, when deciding upon how the capture the Chancellor mini game should be developed, we decided how the Chancellor would be captured; the player would click the tile the Chancellor was currently located in, while the Chancellor would also be frequently changing randomly between the tiles to increase the challenge of the mini game. We also decided that the reward for capturing the chancellor would be money since this would provide the player with a way of getting Roboticons and also buying resources from the market which would help them increase their score.

During the implementation, we kept 3 lists of tasks that had to be completed; those which had been completed; those which were ongoing; and those which were to be started. This was to help us keep track of what development stage we were in for the requirements, this therefore gave us a good indication of what remaining work was left to do.

Once all the tasks were completed on the list, we then knew that all the requirements were implemented correctly and the brief had been concluded. All that was left to do was testing to check that everything was working.

Testing

For the initial stage of testing, we ran all the unit tests that had already been created by the other group, to make sure that all the code that was implemented was working as expected. This ensured that none of the requirements or aspects of the brief were missed out.

Types of Testing

The testing procedure inherited through the previous assessment was not strict. We decided to implement strict testing only for our new classes. Once our new classes were implemented with our testing passing we deemed the game to be working if it was playable and everything was working as it should. Most of the testing was done therefore through unit test and ad hoc testing.

Unit Tests

Unit tests were used to ensure that each procedure and class was working as expected. Since unit tests are used to isolate a piece of code and make sure it behaves as you would expect it to[1]: we decided that this would be a very useful way of testing. Throughout all our implementation, one of our main goals was to make everything as modular and reusable as possible so that one part could be changed without having massive impacts elsewhere in the code. The unit tests were designed with our architecture in mind; given we already knew what functions would exist and the expected output from a given input of a function. However, we did not solely use unit tests since we may want to change the architecture during the implementation due to unexpected restraints of the system that we inherited, as well as changes to the architecture due to refactoring.

Ad-hoc Testing

We used ad-hoc testing[2] as a way of testing newly implemented parts of the system while we were still developing them to make sure that the function's state and outputs were as expected. This allowed us the freedom to implement and try different ways of implementing features while we were getting used to the code base that we had inherited from the previous assessment. Then, once we were familiar with the code base, we switched to a stricter approach to testing. We continued using an ad-hoc approach when we were debugging sections of code; since it was not strict, it made it easy to try different things when trying to the fix the bug that was occurring in the code.

Continuous Integration

We used continuous integration throughout the whole of this assessment. Continuous integration is where code changes are committed to a repository, in this case our git repository on github, and then automated unit tests are run on the current build of the system with the newly committed source code [3].

This means that when a code change is committed, we will know whether it is performing the correct function based upon if the tests passes or fails. If they fail then we can start to debug the code and try to find out what is causing the issue, and then proceed to try to fix it by using our ad-hoc testing approach described above.

Travis-CI

The software used in order to perform continuous integration on our project was Travis-CI. This was because it has excellent Github integration, includes our version control repository and is extremely easy to use and understand; therefore, no time was wasted in trying to setup the system.

Travis-CI builds the system for both the OpenJDK and OracleJDK, which are the two main Java Development Kits, so that we know for both versions it build and runs successfully and if one of them fails it will inform us as to which one has failed to build and run. The unit tests that have been defined to be run by Travis-CI are run on both version of the JDK so that we know the game code will perform the same results. This makes it very easy to use with minimal adaptation.

When Travis-CI runs the unit tests, it runs them through the Gradle build tool as to emulate our own build process as closely as possible. We are then able to see the console output of the Gradle command so if the tests fail we will be able to identify which test has failed by looking at the output from the console.

Quality of Code

Software was deemed to be of satisfactory quality if it:

1) Has good comments.

2) Is easy to read and understand so other people can work with the code if they need to.

3) Fulfils its purpose based on the requirements documentation.

4) Passes unit tests that have been written for it.

5) Passes continuous integration testing with the unit tests that have been written.

It must also be well documented and easy to understand, so when another member of the group had to work on the code. This was very common to fix bugs that were detected during testing; or if they had to modify it in any way, then the person working on it does not have to spend a large amount of their time figuring out how it works, they can simply get on with the change or fix.

Meeting the requirements

We inherited a game, which had a mostly fully implemented list of requirements. It had support for up to nine players including AI players. When the new requirements [4] were released for assessment 4, we made sure to add these to the requirements and add them into the game. This included a chancellor mini game and the ability to have up to four players along with AI players. Seeing as though we already had the four player functionality already implemented in the inherited game from the other team, we concentrated on the chancellor requirement.

We implemented the chancellor mini game by adding an additional stage to phase 3, referred to as phase 3.5 in the requirements documentation under requirement 5.1.2. We decided that this would occur for every human controlled player during phase 3. During this stage, the chancellor would appear on a random tile for half a second and then disappear for another half a second to later appear on another random tile and this would happen for a total of 15 seconds. The chancellor would be deemed as caught if the player had successfully clicked on the tile that the chancellor was occupying. Since the score was calculated at the end of the game by adding up all resources the player had, we decided it would be easiest if the reward for capturing the chancellor was 50 monetary units. This would give the player an advantage by allowing them to buy and upgrade roboticons or extra resources from the market to increase their score.

Since all of the requirements from the previous team had already been implemented the only requirements left to do were those specified by the change in requirements given out at the beginning of the assessment, hence why we have only discussed one requirement since multiplayer was already completed so only the chancellor mini game was left to complete.

References

[1] "Unit Testing", msdn.microsoft.com, (2017). [Online]. Available: https://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx. [Accessed: 30-Apr-2017].

[2] "Ad Hoc Testing", Guru99.com, (2017). [Online]. Available: http://www.guru99.com/adhoc-testing.html. [Accessed: 29-Apr-2017].

[3] "What is Continuous Integration? - Amazon Web Services", Amazon Web Services, Inc., 2017. [Online]. Available: https://aws.amazon.com/devops/continuous-integration/. [Accessed: 01-May-2017].

[4] W. Wood, et al. (2017, May. 2). "Requirements". Downloads – Gandhi Inc. – SEPR Project group [Online]. Available: http://gandhi-inc.me/downloads/Req4.pdf. [Accessed: 2-May-2017].