

Software Engineering Project: Assessment 2

Gandhi-Inc. - Project Blind Eye

Project Testing

We have used a wide variety of testing methods and approaches for different stages of our project development; hoping to use the most appropriate method for each stage. We used some methods outlined in the “ISO standard 29119 - 2013” [1] and others we found.

During the first round of development, we used ad Hoc testing [2]. This allowed us freedom to develop any tests we deemed necessary, or leave out any tests we felt were superfluous. In addition to this, the lack of any formally defined tests sped up the development process: this is important in agile development (the methodology we are following) as it allows us to perform more development cycles.

After the initial development cycle, we moved on to more formal testing procedures. We started by doing unit tests on our individual classes. To do this we are using structure based testing (also known as white-box testing) [4] as we have written the source code. This allows us to develop more comprehensive tests based on how the code actually works rather than an estimate as in black-box testing. The unit tests were written by the people who designed the module. We used the JUnit test framework [3] to improve the speed at which we could write the tests.

For the next development cycle, we moved onto integration testing [1]. In integration testing, multiple components are tested together to validate whether they work correctly together.

The compatibility testing of our game was accomplished implicitly; we are using both Windows and Ubuntu Linux to run our game during our development process, this means that we can accomplish testing of our requirement that our game should be able to run on both of these platforms. There was not expected to be any difficulty in this as we are using Java, which uses the Java Virtual Machine to execute its binaries.

The test that we have written to test the requirement that “The game should boot properly 98% of the time, after being tested 50 times” [5] is an example of smoke testing. Smoke testing is a test to determine if there are any catastrophic failures in the build of software.

It is seen as one of the most cost effective ways to test code, it is seen as an industry best practice.[4] It was a relatively quick and easy test to implement, however would have proved invaluable had there been any failures in the product.

The final step of testing was split into 2 segments:

- Firstly, alpha testing of the product required the product design team, ie Gandhi Inc. , tested the whole project as a whole. Testing gameplay to check that there is no bugs/ errors in the gameplay.[6]
- Secondly, beta testing of the project involved people not involved in the development process, testing the product.

Alpha testing is an example of integration testing [1] as it tests all completed systems running together how they should. Beta testing is also an example of this, however it also allows the system to be used in ways in which the development team may not have intended or thought of. This provides a more comprehensive test of the product.

Software Engineering Project: Assessment 2

Gandhi-Inc. - Project Blind Eye

Report on tests:

Test for "Pub Class":

The pub class is used for the gambling system, it provides, for three different game types, information which is interpreted by the market place on winnings or losses when a user gambles any of their money. As the system uses random numbers, the same input done multiple times would produce different results. For this reason, the test for the pub is different than the rest of the tests. We have used the fact that $\lim_{x \rightarrow \infty} \sum (\text{output of Pub.game}) = 0$ to test our gambling system. We are running the pub game 1,000,000,000 times then dividing by 1,000,000,000. If the game has an equal chance of winning as it does losing, then the result should be very close to zero.

When we performed this test on the class, the results we obtained was not close to zero indicating that our code was not correct. We updated the code and ran the test again, this time getting the result we expected.

Tests for "Roboticon Class":

The tests for the Roboticon class were written using the JUnit framework.

- The first test tests whether a Roboticon that has just been created has all zero values for its properties.
- Test 2 gives the Roboticon a base production rate, then tests if these production rates were correctly updated in the instance of the roboticon.
- Test 3 tries to ask for the production rate of a value that does not exist, this should raise an error.
- Test 4 tries to set the production rate to a negative number, this should raise an exception.
- Test 5 gives the robot new values and then tests that they have been updated correctly.
- Test 6 sets the specialisation of the roboticon and then tests if the specialisation has been set correctly.
- Test 7 gives the roboticon a plot then checks the plot is correct.
- As the roboticon's production has an amount of variance built into it, test 8 checks whether the production rate of the the roboticon is within this variance.
- Test 9 checks if an exception is produced if you ask for a production of a resource that does not exist.

All tests have passed.

Tests for the "MarketPlace class":

We have decided not to implement any tests on trivial getters and setters for the market place as this would take a significant amount and provide little benefit.

- Test 1 tests the ore buying code by setting a price to the ore and then buying some ore. Test 2 and 3 are the same, only they are testing buying energy and roboticons retrospectively.
- Test 4 and 5 test the selling mechanism of the system. They do this by giving the player some ore or energy and then trying to sell to the market.
- Test 5 will test production of roboticons by giving the market ore but no roboticons to see if it makes any.
- Test 6 will produce roboticons based on how much ore is in the market and test that the right amount of roboticons are produced.
- Test 7 will test for errors in values that are not allowed, for example, if the prices are set to negative numbers. It will also test to make sure that the player has enough money to buy something.

Tests 1-6 inclusive run and pass. Test 7 does not pass as we have not yet implemented the exchange rates.

Software Engineering Project: Assessment 2

Gandhi-Inc. - Project Blind Eye

Tests for “Player class”:

- Test 1 will create an example player, give it money, ore, energy and a name. It then tests if the test layer has these values.
- Test 2 creates a plot and player and assigns that plot to the player. It then tests whether it has been assigned correctly.

Both tests have passed.

Tests for “Game class”:

The test for the game class produces 2 AI players and plays the game through with just the AI players to test whether all of the different components work correctly together.

The test has passed

Tests for the whole system:

In our requirements specification, there was a requirement that the game should boot correctly 98% of the time after testing 50 times. We are testing this using a shell script that loads the program 10 times. This was ran 5 times, to give the required 50 tests. This is an example of Smoke testing. There haven't been any failures during testing, therefore the game has passed this test.

The test passed.

We are also using Alpha and Beta testing to test the completed game. This involved the team and outside participants playing the completed game to validate whether there are any bugs in the system. During alpha testing, a bug was discovered in which, there could be zero players in the game, at which point the game would crash. This has been fixed.

No bugs were found during beta testing.

Completeness of testing: Our tests check all functions within our software however we have limited error handling tests. We expect more error handling tests to be implemented later, as it becomes necessary.

[1] "ISO/IEC/IEEE 29119:2013" software and systems engineering - software testing"

[2] "Ad hoc Testing – Software Testing Fundamentals", Softwaretestingfundamentals.com. [Online]. Available: <http://softwaretestingfundamentals.com/ad-hoc-testing/>. [Accessed: 17- Jan- 2017].

[3] "JUnit", Junit.org. [Online]. Available: <http://junit.org/junit4/>. [Accessed: 17- Jan- 2017].

[4] "Guidelines for Smoke Testing", Msdn.microsoft.com, 2017. [Online]. Available: [https://msdn.microsoft.com/en-us/library/ms182613\(VS.80\).aspx](https://msdn.microsoft.com/en-us/library/ms182613(VS.80).aspx). [Accessed: 21- Jan- 2017].

[5] W. Wood, et al. (2016, Nov. 9). "Updated Requirements". Downloads – Gandhi Inc. – SEPR Project group [Online]. Available: <http://gandhi-inc.me/downloads/Gandhi-Inc1.zip>. [Accessed: Jan. 23, 2017].

[6] "Alpha Testing Vs Beta Testing", Guru99.com, 2017. [Online]. Available: <http://www.guru99.com/alpha-beta-testing-demystified.html>. [Accessed: 22- Jan- 2017].