Top Right Corner

Evaluation and testing report

Evaluation of the final product

Our requirements (https://sepr-topright.github.io/SEPR/documentation/assessment4/reqs.pdf) were developed using the brief, conversations with the client and changes issued by the client. We've also had them confirmed by the client so we can be sure that they match the client's view of what the system should be and as we wrote a rationale for each requirement they can be traced back to their origins. When the requirements were developed we also devised fit criterion that could be used to derive tests to ensure that the final system does in fact meet said requirements[1]. Many of our system tests and automated tests were developed to test individual requirements and more importantly can be traced back to the requirements using the traceability matrix we produced (https://sepr-topright.github.io/SEPR/documentation/assessment4/TestMatrix/Sheet1.html). Clearly it is possible to trace our tests all the way back to the original brief (or some other interaction with the client) and we can therefore use all of these tests to show that the system matches its brief[1,2]. As all of our tests pass, we can be confident that all of our requirements have been correctly implemented and that therefore the system matches its brief.

We would have liked to have gotten the client to perform some acceptance testing. Where we would have created some basic tests for them to perform to ensure that all requirements were examined, but allowed them to use the system as they see fit using their own input data etc so that they can be sure that they are happy with what we've produced[1,3]. These tests would have made it absolutely clear whether or not the client was happy with our work, but due to the nature of assessment and the clients busy timetable they were unable to participate.

Testing of the final product

We feel that in order for us to deem our code to be of appropriate quality there must be no "showstopper" bugs that rendered the system totally unusable (i.e. crashing on startup) or "high-level" bugs that render a major component of the system unusable (i.e. if the gambling system would cause the system to crash every time the player lost)[4]. However, we did not aim to eliminate every single error that exists, it's normal for software to ship with minor errors such as cosmetic errors (e.g. spelling errors) or *a small number of* minor bugs that cause the system to behave in a strange way but does not cause the system to crash and for which a workaround exists. If someone is determined to break the game they most likely will find a way, the elimination of all defects is near impossible and if we had focussed too much of our efforts on testing we wouldn't be able to make the overall product as good it is, also fixing minor bugs carries the risk of introducing new, potentially more serious ones. We're confident that we've show that no high-level/showstopper bugs exists that can be triggered under normal use of the system and as such we believe that our code is of appropriate quality.

Our testing methods haven't changed since assessment 3 (as we found them to be very effective) as such the description of our testing methodology has largely been derived from what was submitted previously (https://sepr-topright.github.io/SEPR/documentation/assessment3/updatedTestingreportnocolouring.pdf). We utilised three "levels" of testing starting with automated unit tests for every back-end class (every class that did not involve any UI code). These tests were divided into a test case for each class that contained unit tests for every public method in the class under test. As before we used Junit to write all automated tests and JMockit to mock out any dependencies when unit testing (create fake versions of objects that the class under test depended on that would always behave in a specified way[5]). Without mocking our unit tests would not have been true unit tests. A unit test is meant to test only one thing[6], if we did not mock out these dependencies we would also have been testing the functionality of the other objects[5]. Using JMockit also allowed us to be sure that mocked objects were being used correctly by the class under test ensuring that the

correct methods from the mocked object were called with the correct arguments, thus increasing the power of our tests. In many cases several tests had been written to test one method using the same input data to test different aspects of the method call such as the result produced and the methods of various mocked objects that were called. Although it massively increased the number of tests, when any one test failed it was far easier to find out exactly what went wrong, making it far easier to debug our code.

We also utilised integration tests, testing many objects together to ensure that they worked well together, we did this as some behaviour only becomes obvious once several components have been combined together[1]. In many cases we had taken our unit tests that involved several mocked objects and rewrote them using 'real' (not mocked) objects, saving time and ensuring that the same functionality was tested using real objects.

Finally, we utilised some manual system testing, testing the system 'as a whole' rather than individual components[1]. We did this to be sure that all the components behaved properly together (some behaviour only becomes apparent once components have been combined[1]). Automated system testing is very difficult[1] and the tests would have been very complex due to the large sequence of operations that would have to be performed to simulate e.g. a single player's turn, we would likely have made mistakes when writing them. We also wanted to convince ourselves that the GUI worked correctly as we did not perform automated UI testing, it was necessary for us to test this functionality using manual methods. The large quantity of automated testing that we'd carried out meant that we didn't have to carry out a large number of manual tests as e.g. the majority of the validation tests etc had already been fully tested.

Automating most of our testing made regression testing (retesting the code when changes were made[7]) far easier as the tests can be run in seconds. This allowed us to continually refactor our code whenever we saw something that could be improved (as we stated we would in the methods and planning document of assessment 1 https://sepr-topright.github.io/SEPR/documentation/assessment1/deliverables/Plan1.pdf) without fear of breaking anything or spending time manually testing code. This made us more comfortable refactoring our code and likely improved its overall quality[1].

We choose to reuse most of our automated tests (https://github.com/SEPR-TopRight/Roboticon-Quest/tree/master/core/test/io/github/teamfractal) and system tests (https://sepr-topright.github.io/SEPR/documentation/assessment3/ManualTest.pdf) from assessment 3. Both our assessment 3 game and the game we took on for assessment 4 were based on the same code base. Allowing many of our tests (that we believe are of high quality) to be reused saving massive amounts of time (that we could use to improve the game). We had to delete some of the tests as they related to methods that we had implemented in our old system but were not present in this one. The automated tests we inherited (https://github.com/Jormandr/Roboticon-Quest/tree/master/test/src) were fairly limited (didn't cover much of the code base) and often of poor quality (no unit tests, only integration tests testing multiple different things at once). However in a few places, some of these tests were OK and were re-used.

We didn't follow a test-driven development cycle as we planned to in assessment 1 (https://sepr-topright.github.io/SEPR/documentation/assessment1/deliverables/Plan1.pdf) as most of the changes we made were to the UI or to classess that involved UI elements that were difficult to automatically test (as LibGDX, the library used to implement the UI, has always caused us issues when it comes to automated testing). These new elements were of course tested using manual methods.

We feel that our tests were fairly complete and as they all passed our code must be of "appropriate quality". As we started from the simplest level with unit testing and tested every public method using normal, extreme and erroneous data we can be sure that every line of code has been executed at least once under both normal and extreme conditions and every exception that could be thrown by our system has been thrown. By then going up a level and performing integration tests we can be sure that the objects in our system interact well together when combined to form composite components and that there is no unexpected behaviour that results from combining these objects. Finally by testing the system as a whole via system testing we can be sure that all these composite system components interact well together and that once again is no unexpected behaviour revealed by combining said components. Testing in this way, working from the level of single methods up to the entire system we were able to test our system far more completely than if we had just performed system testing as we can be sure all the code has been thoroughly tested rather than just that

which is easily testable using the user interface. For example players cannot try to buy more ore than the market has in stock (as the spinbox used to select the quantity of ore that they wish to buy does not allow them to). If we had not also performed unit testing we would not have been able to test what happens if a user does manage to try buy more ore than is available, if the ore buying code was reused later in another project or the system modified in a way that did allow users to try and buy more ore than is available we could not be confident that our code would work as expected. It is worth noting that most of our non-functional, user interface and constraint requirements were untested as we could not develop precise tests to test them. We do however feel certain that requirement 3.1 holds as we have played the game many times on the computers in CSE 069/070. Similar automated tests were carried out using both real and mocked objects and several different data sets and then in some cases the same functionality was tested again in the manual system testing and all these tests passed helps to show the correctness of our tests. It's unlikely that at all three 'layers' of testing, tests that tested the same functionality could all have been incorrect yet passed.

We carried out a total of 1907 automated tests, all of which passed, test results can be found here: https://sepr-topright.github.io/SEPR/documentation/assessment4/test/index.html and a JavaDocs describing the purpose of the tests and the expected outcomes can be found https://sepr-topright.github.io/SEPR/documentation/assessment4/testJavaDocs/doc/index.html. We also carried out 106 manual system tests, our test plan and results can be found https://sepr-topright.github.io/SEPR/documentation/assessment4/manualTest.pdf. Our traceability matrix showing which tests test which requirements can be found here: https://sepr-topright.github.io/SEPR/documentation/assessment4/TestMatrix/Sheet1.html:. A version of our traceability matrix with additions (as compared to assessment 3) highlighted in green and deletions in red can be found here: https://sepr-topright.github.io/SEPR/documentation/assessment4/TestMatrixChanges/Sheet1.html:. Similarly a version of our manual system tests with changes highlighted in yellow can be found here: https://sepr-topright.github.io/SEPR/documentation/assessment4/manualTestChanges.pdf.

Junit and JMockit can be found here: http://junit.org/junit4/ and here: http://jmockit.org/.

The following changes were made to our automated tests:

- Deleted the MarketTest class as it only tested gambling code that is not present in this version of the system. Similarly all tests that tested this gambling code were removed from the MarketUnitTest class
- Removed methods that tested the removeRoboticon method from the player class as that method does not exist in this version of the system
- The getScoreParameterisedTests were removed from both Player and PlayerUnitTest as the score is no longer calculated in the Player class
- Both the RandomEventsTests class and the RandomEventsUnitTests class were removed as random events are handled differently in this version.
- ScoreComparissonsTest and ScoreComparissonsUnitTest as the ScoreComparissons class does not exist in this version of the syste
- Tests were added to the PlayerUnitTest class for the methods event, gambleResult and giveMoney that were not present in our old system

We also kept the following automated tests that we inherited from the previous team:

- RPSAITest that tests their gambling code as we did not already have any tests written for this
- AdjustableActorTest that tests the code for spinbox style widgets used in the market screens

Does our product meet the requirements

In short we believe all of our requirements (https://sepr-topright.github.io/SEPR/documentation/assessment4/reqs.pdf) have been implemented. Our testing and evaluation (see above, specifically the traceability matrix and evaluation section) confirms that nearly all of our requirements have been implemented. The only requirements that have not been shown to have been implemented are 2.1, 2.2, 3.2 and 4.1. In order to show that the system was easy to learn (requirement 2.1) we would have to get a number of different people to attempt to perform a number of tasks using the system and give us their feedback so that we could analyse it. This is something that we just don't have time to do. However, we feel that the tips given in the top right hand corner of the screen that tell the user what they are supposed to be doing and the fact that users can only perform certain actions when they are appropriate (i.e. there is no way to access the roboticon market apart from in the roboticon purchasing phase and similarly no way to interact with the plots when you're supposed to be purchasing roboticons) guide the user in the right direction helping to make the system easy to understand and use. Please see these images https://sepr-topright.github.io/SEPR/documentation/assessment4/cantuseplots.png https://sepr-topright.github.io/SEPR/documentation/assessment4/install.png and the ones that we've linked to below.

We thoroughly believe that the system is reliable (requirement 2.2) and during testing we found no bugs that would cause the system to crash. There were no (Recorded) tests to ensure that the game was in fact capable of running on the computers in CSE/069&70 (requirement 3.1). However, as stated above, we have played the game on those computers a number of times and have no reason to believe that it does not work. Similarly, whilst there are no explicit tests to ensure that the in game map resembles the university of york campus we have met the fit criterion for requirement 3.2 as there are 3 clearly recognisable landmarks present: Heslington hall, central hall and the sports center (https://sepr-topright.github.io/SEPR/documentation/assessment4/landmarks.png). Again there are no explicit tests for requirement 4.1 that states that it must be clear which player (if any) owns each plot of land. However, again we believe that we've met this requirement by placing a square highlight around each acquired plot with each player's plots being highlighted in a separate colour (https://sepr-topright.github.io/SEPR/documentation/assessment4/colours.png).

References

[1]I. Sommerville, *Software engineering*, 10th ed. Harlow: Pearson, 2016, pp. 226-245. 245-251.

[2]"Evaluation in the software life cycle", *Issco.unige.ch*. [Online]. Available: http://www.issco.unige.ch/en/research/projects/ewg95//node94.html. [Accessed: 23- Apr- 2017].

[3]"Acceptance Testing – Software Testing Fundamentals", *Softwaretestingfundamentals.com*. [Online]. Available: http://softwaretestingfundamentals.com/acceptance-testing/. [Accessed: 23- Apr- 2017].

[4]"Product release readiness: Product testing & product bugs", *MaRS*, 2009. [Online]. Available: https://www.marsdd.com/mars-library/is-the-product-ready-for-release/. [Accessed: 30- Apr- 2017].

[5]. Nijssen, "Mock your Test Dependencies with Mockery", *SitePoint*, 2014. [Online]. Available: https://www.sitepoint.com/mock-test-dependencies-mockery/. [Accessed: 22- Jan- 2017].

[6]"Properties of Good Tests: A-TRIP – cavdar.net", *Cavdar.net*, 2008. [Online]. Available: http://www.cavdar.net/2008/10/25/properties-of-good-tests-a-trip/. [Accessed: 22- Jan- 2017].

[7]"What is Regression testing in software?", *Istqbexamcertification.com*. [Online]. Available: http://istqbexamcertification.com/what-is-regression-testing-in-software/. [Accessed: 22- Jan- 2017].