Nathan Healea
June 6, 2016
CS 362
Part 4: Test Report

Test Report

Thought out CS 362 Software Engineering II testing was the main focus of this course. We where giving a GitHub repository of mashed up Dominion code that rarely worked. Segfalt, wrong card effects and improper method calls lined the Dominon.c source file. As the class progresses we wrote unit test and debug the conglomeration that was code. Part of the solution was to do refactoring of card effect out of nested loops and switches, while using testing method such as delta debugging, tarantula, and mutant test to make sure our refactoring worked.

My dominion game came log pretty nicely. Once we got past the initialize part of the class we were able to make changes to the Dominion.c source file. The first part that had a huge impact and made testing easier was the refactoring of card effects into separate logic methods. The cards I chose to refactor was Smithy, Outpost, Seahag, Treasure Map and Remodel. I choose these cards since them seemed to come up the most during a game, with the exception of Seahag, and could be tested thoroughly if needed be.

In part of the class we ended up testing card effects and Smithy was one I chose along with many of the other students. Smithy's card effect was to draw three cards the discard one. In order to make sure that this card effect work properly I wrote a unit test that captured the players hand before the card effect was called the compared it to the player's hand once the effect had taken place. I found that it wasn't giving the write amount of card in my hand. I was receiving three cards instead of the two. After I applied my fix I was able to get an 87.50% code coverage. I keep trying to increase the coverage of my code but found not luck in doing so. I theorized that since it is called through the caredEffect method, which is a giant switch statement, the code coverage would not increase past a certain point. Since the test main focused was to only test Smithy the code cover did not increase.

Testing a method other than a card effect was another part of the class. I decided to test initializeGame to determine if there was any bug in the method. When starting to test this method I used Agans' Debugging rules to find and locate any errors. Agans' first rule "Understating the System" played a big rule in finding the error that I did. When constructing the unit test for the method I hand to understand how the game was supposed to be initialized and what values needed to be set for a game to be played. Agan's second rule "Make it fail" was used to find determine if what I had found was an error. I found that the players hand and deck count when the game was initialization was not correct. I noticed that all but player one hand would be set to 5. Agan's Third rule "Quit Thinking and Look" was used to find the errors location in Doninion.c. When looking I found that there was code commented out that added cards to the players hand. That code was uncommented and all the player hands got set to 5 except now the deck count of player 1 was 0. After further examination of the source code I notice that the initializeGame method also set game state for player 1 and drew them five more cards. Once those lines were removed the make state was fixed.

Testing other students GitHub repository came to be troublesome. Since some students did not follow all the naming conventions laid out in the class running our test suites on their code was difficult and resulted in changes in either there code or mine in order to get the test suites to work properly. The way I dealt with this issue was to copy all my cardtest, unittest, and Makefile over to the GitHub repository that I wanted to test. For this I looked the Smith card effect again know that both students has make changes to that cards' effect. When I can my test on their code Student A received a code coverage of 91.43% and Student B received a code coverage of 82.14%. I found it interesting that even though all three of our card effect were cover by my unit test that Student A got the highest code coverage. I found that after running Students A test and comparing it against mine and Student B, Student A had more reliable code. In one of the test I ran in Student B code I was able to find a segfalt error where the SupplyCount was not being set and when my test tried to access the array result in arbitrary value being caught and display as error.

In this class I noticed a reoccurring thing keep popping up. There are many different types of test that can be done out here, but it seems that there is not one finite way to test your code and validate it a 100%. One testing method that I tried was Mutation testing. I found that in order to get Mutation testing working there was gobs of set up that needed to be done and you had to compile mutants into you code then run it. As I ran my mutant files I notice my output were all the same and that I can't catching any mutants. Come to find out that I was compiling the default code and not the mutant code which was resulting in a false positive. Once I fixed my mistake in my script was catching 57 out of 227 mutants. This told me that my unit test was not very good, but also since the set up for mutation was so involved it was easy to introduced mistakes into your testing process.

In the end I have learned a lot when it came to testing and debugging code. I now can determine if someone code is reliable to compare to and to write unit test that will thoroughly test my code. Being able to use many different debugging method and techniques will came in handing when trying to find bug buried in different layers of code. When it comes to test it is important to know that no code will be 100% foolproof and that there is not one magical testing method to catch every bug.