Connor Sedwick
CS 362 SP 16
June 4, 2016

Test Report

Testing Dominion has been an experience. Having some experience in writing unit tests for other programs I did not find it extremely difficult to do so for this game. Over the weeks of assignments and having practice writing unit tests, random tests, and a game player, I have come to understand both testing and the inner workings of dominion.c. Testing dominion.c has also shown me that even when the game runs well, there is still a chance that there are logic errors within the source code. The errors are sometimes also in the testing code, however.

In writing unit tests for the cards and certain game functions I had at times written a bug into my own tests that gave them an inability to fail. At times I also wrote a test that was testing certain branches in the game and only getting coverage, but not checking for errors. It became apparent to me after this that code coverage does not give the tester much information on what is broken in the code. It did show what wasn't being tested which did help in the event a new bug came up after I tested and passed the covered lines.

Much of Dominion's gameplay relies on the use of kingdom cards in order to move the game along, decide a winner, and end. Recognizing that kingdom cards are very important to the game I decided to focus much of my effort on understanding what each card does and to check if it was implemented properly. This also required that I make sure the functions that are used for players to buy cards was also working. In a sense I used a hierarchy to find the bug. I would first look at the card to ensure it was properly written, then step back to see if it was correctly labeled in the cardCost function, and then I would check to see if the buyCard function was being properly invoked and responding properly.

Although, the game is written in a manner that works, there is much that could be refactored to save space. For example, in the cardEffect function there are many cases that are at least ten lines long because each case has to run differently depending on the card played. Part of these cases is factored well, especially in the cases that the effect requires that a user draw and discard cards in which a function is called to do so. There are cases though that are extremely lengthy and could do with some refactoring to make the case statements easier to read as well as make it easier to sort out bugs in lines of code. Because each card will have its own function it will be in a sense isolated and easier to work on.

To test the cardEffect function effectively I found that the assignment for creating random tests made it much easier to hit on different "if statements" for the same card. This really made it easier to generate a larger code coverage than basic unit tests. I averaged about 60% coverage with both my unit tests and random tests where unit tests made up around 24%. Doing so also increased the chances of finding faulty code in the card's implementation. This didn't only apply to cards though. When running tests on the parts of the game that calculated

player currency, I found that randomly assigning values made it more possible to see if the coins were being calculated properly and that each different type was being implemented.

I would like to say that the portion of the testing that required the implementation of a whole-game generator was not as difficult as one might think. To write my whole-game generator I made use of the playdom.c file and modified it to run more cases and test multiple card plays and strategies. On average it took only about 6 different tests with different seeds in order to reach 61% coverage. I do feel that my whole-game generator could use some refactoring to make it more efficient and run smoother. It does reproduce results depending on the seed it is passed.

In reviewing the works of my peers in this assignment and their Dominion code I can see that the two I paid close attention to, foulderd and healean, did little to change the overall code of the game, but did refactor the kingdom cards to make them easier to work with. From my testing one of them implemented a bug into their refactoring as well. Because they only refactored some code and added minor bugs the reliability of the code is probably still low due to the bugs found in the original code. I would say that the code for the game is still reliably buggy, but with some nice refactoring to make it easier to read.

In taking this course and messing around under the hood of Dominion I have to say that I have learned much in how to test for bugs and identify them. I have also learned that caution should be taken when looking at test results as they may be the product of a faulty test. It is apparent that testing takes some time and that there is some method to the madness it entails. When I had originally written my tests I did not make use of oracles and instead just threw everything into the main and effectively had poor testing, but when workshopped and properly implemented my testing became more effective providing me with both passing and failing cases instead of just one or the other. From my experience, it is important to understand how the game is expected to be run. Earlier on in the term when working with Dominion I did not understand exactly what it was I was testing. This made things difficult when it came down to identifying bugs because I did not know what the game was supposed to be doing. From my understanding it is best that a tester understands how a product works before testing it, but that it is not impossible to fix bugs in code without knowing how the game is played. Kind of like how even a blind squirrel can find a nut every once in a while.