Steven Silvers
CS 362
6/5/2016
Final write-up

The entirety of the Software Engineering II course was focused around testing the card game dominion, implemented in C, in a variety of ways. The first step in being able to test any software system is to be able to understand the software system at hand. Since this was supposed to be an implementation of an already existing game, before I even started looking at the code I went out and read the rules for dominion. After learning as much as I could from the rulebook, I went and found sites online where I could play dominion for free and got some hands on experience with actually playing the game. All of this work before even touching the code led to an easier time writing tests, especially the system test for assignment four, because I didn't have to keep going back and looking how the game is supposed to be played and in what order things are supposed to happen.

The first assignment consisted of refactoring five cards within the dominion code which overall was not very difficult. If anything this assignment was more to get familiar with the dominion code and didn't involve testing the dominion code in any way. The way that the cards were initially implemented was in one massive switch statement within a function called card effect. I found that the easiest way to refactor would be to leave the switch statement intact, but instead of having each card case contain the code for the card's operation, I had the case call a function that would handle the card's operation. This helped clean up the switch statement and ultimately made the dominion code easier to read. It did not however change how the program functioned as all that was essentially done was to move code from one location to another, but if a feature were to be added to the dominion code in the future where it was necessary to access a card's operation outside of the cardEffect function, that would be possible because of this refactor.

Assignment two was the first part of this class that involved writing tests, more specifically unit tests. We had to write four unit tests that tested functions within the dominion code, and four tests that tested cards within the dominion code. Unit testing is fairly straightforward, take a small chunk or unit of code that you want to look at, typically a single function and come up with a test to make sure that the unit gives output that is appropriate in regards to its input. All of the functions that I did unit tests on worked properly, but a couple of the unit tests for the cards revealed bugs in the code. I found that in the salvager card coins are not added equal to the trashed card's cost, and I also found that the Sea hag card does not get discarded when played. I got about 73% coverage on each tested unit, and 34% coverage of dominion.c overall. The reason the coverage was not 100% on the tests was due to various if statement conditions that were not met by the unit tests.

Assignment three dealt with random testing, making sure that certain cards behaved properly. I found that random testing was probably the most helpful form of testing dominion, since there are so many different possible combinations of inputs to the various functions. The only real use for unit testing for dominion in my opinion is to test specific edge cases, other than that random testing should be the approach used to test the dominion code. In my random tests a looked at the adventurer card, as well as the smithy and village cards. I got 100% coverage on all three cards with random testing, this was easy to achieve because I could run hundreds of test quickly and easily, including ones with invalid input so that error cases would be reached as well.

In the final assignment I looked at system testing dominion as a whole, basically creating a tester that would play randomly generated games of dominion over and over trying to find bugs. My system test was able to achieve 85% coverage of the dominion.c code after running about five games, I believe I wrote a pretty strong system test for dominion.c that was also very helpful in doing the class project. I used this system test to compare my dominion code against that of two of my classmates, Paul Lantow and Alec Merdler. I found Paul's code to actually be less reliable than the original dominion code, because his remodel card was refactored in a way that can cause a core dump. His remodel function forces variables choice1 and choice2 to equal certain values every time, which can lead to illegal plays that ultimately cause dominion to fail. Coverage wise I would get about 78% coverage of dominion.c after running my system test five times successfully. Alec's code had a hiccup where it would occasionally core dump when trying to play the adventurer card, I tried looking for a cause but couldn't find one but I would still say his code is less reliable than the original dominion.c code that we received. My system test got around 75% coverage of Alec's dominion.c after five successful runs.The only changes I made to the dominion code besides the original refactor in assignment one was fixing the Sea_hag card so that it discards after being played, so I would say that my code is more reliable than the original dominion code.

For the final project in this class I chose to do mutation testing to see how strong my system test is. Using a publically tool I generated compilable mutants of dominion.c and then ran my system test against them to see how many I could kill. I managed to kill all of the mutants, but I believe part of this to be due to drastic changes made by the mutation tool to the dominion.c code. For example, in the kingdomCards functions, some of the mutants messed with how much memory was allocated to the kingdom cards array, so instead of "10 * sizeof(int)," it would be "10 % sizeof(int)" or some other operator. There were also mutants that broke the endTurn function, so as soon as the first player attempted to end their turn the game would essentially freeze.

Ultimately I found software testing to be a lot more interesting than I originally thought it would be, and a lot of these tools will be things that I keep with me as I continue my career in the software field. I believe that being able to write good tests is an essential skill for good programmers, because what good is code if you can't make sure that it works.