Drew Hamm
6/8/2016
CS 362

Test Report

After first learning that we would be testing an implementation of dominion I was pretty excited because it related to a game I was familiar with but quite unsure as to what value I could gain from the experience. I felt quite confident with my abilities to tests programs so I was not worried about the classes assignments. Although the idea of testing didn't phase me, I was a bit startled upon discovery of the poor state in which my copy of dominion was implemented.

The first assignment had me undergo the process of becoming a "subject expert". I thought the term was interesting and that it provided insight into what makes a good tester. Without knowledge of the overall goals of the system, a tester is limited to fixing bugs that are either already known, or simply require common sense to find.

After familiarizing myself with the rules and cards of the game I started the second task which was to refactor five different card effects. The goal in refactoring was to enable the card effects to be tested individually. I ended up refactoring six different cards without much difficulty. During assignment two I had to start creating unit tests for the card effects. This process was made easier by the first assignment refactoring. I would have liked to have created unit tests for all the cards for the sake of completion but time constraints limited me to simply meet the assignment's requirements. My unit tests were straightforward in that I checked that the state transition followed the rules of dominion for the given card and its effect. While testing I started to realize that the provided implementation tried to combine several different steps into a single function. Some of those steps didn't even follow the order in which the rules would allow.

In order to complete the second assignment I had to learn more complex makefile instructions than I have used previously. I found that there was a bit of a learning curve involved before I could successfully output my results into unittestresults.out. Part of the challenge was learning how to output the test coverage percentages. I've used test coverage tools in visual studio but working directly through the command line is something I was less familiar with. I ended up getting around 35% test coverage from the card effect unit tests which I found a bit surprising.

Assignment three was an introduction to random testing. I've used random testing to some extent previously so this assignment was more practice. One problem I had with this assignment was trying to only generate random states that would actually be possible in a game of dominion. An example would be to ensure no players were randomly given curse cards in a game where no cards in play had the ability to give players curse cards. I realized that I was working my way into creating a full specification for the game. I learned later that I could simply fill the game struct with random data, check the initial state, run my test, and check for the expected state transition. Although I was using a different form of testing for this assignment, my test coverage did not improve between cards tested using my previously implemented unit tests.

Assignment four was the most interesting assignment given during class. Although the assignment mentioned that the exercise was not to write a "good" AI dominion player, since I'm enrolled in CS 331 Intro to AI this term I ended up entertaining the thought. At first I wanted to implement some sort of minimax algorithm for each player to determine what cards to play during the action phase along with the choices as well as what cards to buy during the buy phase. My second thought was to create utility agents that tried to buy cards that fit a particular deck schema. If I would have had time to implement the utility agents, I would have liked to keep track of the best deck schema though a genetic algorithm implementation. Although I spent a bit of time think about the AI that I would never implement, before I could get on with the rest of the assignment I had to go back and look through the rules of dominion again. I ended up just randomly choosing what cards to play and buy during a player's turn as allowed by the rules and the current state of the game. I also included the possibility for the player to not play and action and separately to not buy a card. While thinking about how I would test the full game and looking up rules I started to realize how much the current implementation differed from the actual game. Some of these differences were simply doing steps out of order or in taking them in the wrong game phase.

During assignment four I realized that I could have saved myself time If I would have wrote my previous tests in such a way they I could reuse them for this assignment without refactoring. It seems that when working with random tests in which the initial state is generated randomly, the test involves checking the initial state and through a simple rule system determines what the next state should be after the method under test is ran. I would have preferred to write my unit tests to simply receive a given an initial state and the next state then check to see if the transition was correct. When I created my unit test initially, I had a file for each method under test that received no arguments. The file tested different edge cases separately within main. A better solution would have been to implement the files such as unittest1.c as a suit of unit tests all using the same state transition test designed for each individual method under test.

I ended up with test coverage around 67%. After comparing my implementation to another student's implementation using my tests, I was able to notice some cases where our implementations were different by the difference in passing and failing tests. The value with theses comparisons is that I should be able to pick and choose the implementations of card effects that are working correctly from either to be combined into a single dominion game. Taking this one step further, I could collect all students implementations and combine only the working sections after running theses tests. Although combining the best work from each student sounds great at first, I was able to notice implementation changes between just mine and a fellow student’s dominion game that would require more work to combine than just a copy and paste job as anticipated. This also pointed out the weakness of reliability of my code and the code of a couple other student’s dominion implementations that I tested.

Overall this class taught me new ideas relating to testing and gave me some experience putting those ideas to use. I found that unit testing was quite familiar, whereas some of the random testing was new. An initial challenge I faced was learning how to use makefiles along with gcc and gcov. The most challenging aspect of this class was to learn how to use delta debugging. I found that the python script was not current for python 3 which caused me to put in quite a bit of work before I could run it on my system. This process would have been less difficult if I was more familiar with python. Delta debugging would have been less challenging if I could have found more examples but by searches turned up only minimal information. I would like to implement more automated testing in the future and make use of the tools learned in this class.