Alex Schultz

Professor Groce

CS362, Spring 2016

6 June 2016

Final Project

Test Report

This is a test report documenting my experience testing Dominion throughout the 2016 Spring term. When I first signed up for Software Engineering II, I thought we were going to write a program from scratch, testing and building as the term progressed – similar to what we did in Software Engineering I. I am glad this class did not turn out how I expected because joining the work force and starting from scratch is not very likely. Understanding how to read unfamiliar code and implement tests to find bugs is extremely important, so I am grateful to have learned the process of debugging, the challenges it brings, and sufficient methods to do so.

What I found most challenging, initially, is that the code was relatively unstructured and difficult to follow. Truly, without consulting outside resources, this code was hardly readable. I did find it very challenging writing tests because dominion is complicated. There are many cards with multiple components, so it took some time to get familiar with Dominion. I had never played Dominion, but assignment 1 became clear after I understood what functionality each card is supposed to have. Thank goodness for the Wiki on Dominion! I refactored the Council Room, Remodel, Adventurer, Feasts, and Smithy cards. Assignment 2 required writing unit tests for four functions, so I implemented tests for whoseTurn, getCost, numHandCards, and buyCard. Action card testing include Steward, Sea Hag, Outpost, and Great Hall. These unit tests were relatively straight forward after understanding how they are supposed to work. Testing action cards required a more robust approach, including game simulation and adding assertions to verify the card has been implemented and played correctly. Multiple assertions had to be made for cards with various effects.

In Software Engineering I we used Test Driven Development, a process of only writing code until it fails, writing enough code so it produces the correct result, and then refactoring. Those three steps were ingrained, so that is how I started testing and debugging for Dominion. As the term progressed and I learned of Agans' Debugging Rules, I found it very helpful and supplemental to TDD. The first rule, understanding the system, was the first part of my debugging process. I cannot debug something I do not understand, so I first had to ensure that I understood each card's functionality. His second rule, make it fail, is essentially the first step in TDD, so I definitely used that along with Agans' fifth rule: change one thing at a time. Software Engineering I provided a decent foundation for unit testing, so I used that in conjunction with new ideas found in this class to write tests. Newer concepts such as gcov and random testing

were challenging, but I can see how useful they are to ensure sufficient code coverage and error handling. Random testing generates random, independent inputs, so that definitely helps verify a program's validity by finding bugs. Gcov is an excellent and easy way to find code coverage, which was very useful in this class.

Random testing the adventurer card was a challenging, but worthwhile experience because I had never done anything like it before. The random test was initialized with random cards in addition to a random number of players. Each player would receive 5 cards to start out. As I stated in debugging.txt, I noticed that the adventure card was not correctly discarding the non-treasure cards. Before random testing, total code coverage was about 30%, but afterwards I was able to achieve 100% statement and branch coverage for the adventure card. There is still a lot of code untested, but it is clear to see the immediate benefits of random testing. Generating a random tester to play a complete game of Dominion was actually really interesting. It was extremely challenging with regard to implementation, but I did learn a lot. One reason it was challenging is because we would first need to know what the code is doing in each function before writing the actual code. Without a correct implementation of Dominion, it is highly unlikely to run a complete game. It took a lot of time staring at dominion.c to verify that each card was written into the full game tester. When all was said and done, I achieved over 75% code coverage of the dominion code.

Using this tester against the code of several of my peers, I found that their dominion code was in fact playable, but not one hundred percent correct. I was not surprised due to the challenging nature of this course, but it was nice to see that they all have over 60% code coverage, which was the goal. Testing the full game allows us to play it and find errors when unexpected behavior occurs. It would have been a lot nicer to have a one hundred percent correct implementation of dominion to compare testing suites, but I suppose that would not have been as much of a challenge.

As I stated before, I learned a lot in this class. Concepts like random testing and automated testing along with Agans' rules will definitely serve me well in the future. It was nice to learn gcov in depth along with Tarantula because it truly reveals a lot of about the program. Unit testing, integration testing, and system testing are very important concepts, and I feel like Software Engineering II has better prepared me for the world outside of college. Dominion was a project that had little documentation, hardly any structure, and it was incredibly difficult to know how or where to begin. A lot was accomplished over three months, and I can clearly see how relevant this class is to the programming world. If programmers are not accurately documenting their functions, writing sufficient test cases, and have little code coverage, then it's going to be a very stressful and difficult time for the next programmer who has to maintain or restructure that program.