Jacob Broderick
Dominion Test Report

The Software Engineering 2 class offered at Oregon State University teaches students first hand how to test real world applications. In the real world, many programs are already made poorly, and need to be tested. The Dominion code we were handed was able to compile, but was not always achieving the intended results of function calls. The goal of this class was to test this code. There were many difficulties in ensuring that the Dominion code was functional through the duration of this class. The project was poorly maintained and poorly developed before being handed to a test engineer (or me playing the role of test engineer). The testing process of this code is to first test the individual units with traditional unit testing and random testing, then to test the entirety of the function with random inputs. This was to ensure good code coverage and make sure there are no bugs with the running of the code. This testing was able to make sure that many branches of this program were properly executing.

The ideal way to start testing this program would be to create small atomic tests of all of the individual parts and make sure they work as intended. These are called unit tests. If all of the parts are working, then it will be easier to find any problems when testing how they work together. We were assigned the task of making unit tests for a number of card effects and functions. The card effects were relatively easy to find a few test cases for, but it was harder to find every possible test case. Most of the cards had undesirable effects that were not at all related to what the card was supposed to do. An example of this was the Smithy card did not always discard itself after being played, or the Village card would not always give the extra actions that it was supposed to give. These edge cases required extra thinking outside the box in order to fix. A solution to this problem was assigned to us. This solution is random testing.

Random testing a set of cards ensured a more broad code coverage. This is due to the fact that we were able to test a wide range of cases and ensure that all of the results were correct. This kind of testing is less relevant given clear, concise, and single functions, due to single units being easier to find test cases for. Unfortunately, some of these functions are not single units but rather giant behemoths that do many different things at once. This is a very useful testing style for these kinds of functions because of how hard it is to predict all of the possible combinations of inputs and outputs. Although this form of testing assists unit testing, it does not lead to testing the program as a whole. This usually requires integration testing.

For the Dominion program, we were tasked to write an integration test that tests random playthroughs of the entire game. This ensures that all of the functions were properly able to work. If ever the program stopped working or did not work, then the program would let us know. If it were properly written, we would know where as well. This test style ensured the biggest coverage of the entire Dominion suite. After running it ten times, it already was at 40% coverage. The more that it was run, the higher that code coverage was. If I were to write a more in depth test suite, I would have ran integration tests on specific groupings of functions rather than all of them at once. This would make it easier to keep track of the data as the

individual functions run. The testdominion test was a success at what it was trying to accomplish. There are many less useful testing styles available.

One testing style that I was introduced to in this class was mutation testing. Mutation testing is changing the program itself and running the test on the adjusted program. If the tests pass, even if their function changes, it is likely that the test is not testing for the right things. In order to do a mutation test, you kill the mutants that fail and take a look at the passing mutants. Often times making deliberate changes to your tests might be more effective.

Another good solution to this problem is to use test driven development (TDD). Writing the tests first and writing the code to adhere to them makes sure the tests pass because the code works. This also tends to lead to better code assuming the tests are working well. This was not a topic covered in the scope of the class, but is a useful testing tool that I have used many times in personal projects and work projects.

There were other topics covered in the class that were not necessarily required for testing the Dominion code. There was Tarantula testing which would find ratios of failed tests to passed tests to determine what needs to be changed in the code or tests. There are also many testing tools that aren't applicable to this project, but would be useful for other projects.

Debugging is another topic that was covered in this class. Proper debugging is helped by having proper tests so that one can discover the location of the bugs, as well as their existence. This aids in fixing them, which will reduce the amount of time necessary to maintain the code. This is the reason that a good team should always include a tester, especially in agile teams.

There are many different aspects of testing that often get overlooked with an initial education in software development. The main topics of testing that one must learn are unit testing, integration testing, and system testing. During Software Engineering 2, students were taught about each of these topics in a way that would give them a decent preparation into the real world. Dominion was a realistic project that was big enough and bad enough to represent some project that could be handed to someone at a regular job in the programming world.