Test Report

The process of testing dominion was definitely and eye opening experience. Before taking this class, I never considered an effective process for debugging code and testing it to make sure that it was working as expected. I have been introduced to it in the past but never in the amount of depth that this class has made me experience. However I always knew how much debugging and fixing code consumes the time of the developer. From my personal experience, I have discovered that development take only a small portion of the time while almost eighty percent of the time is spent debugging and of that most of the time is just looking for what/where the bug is.

I think the most important aspect of testing that I came to discover was the ability to completely understand the API that this under test. In the past I always thought of QA or the testing team as the engineers that were not strong enough to develop so they just run tests. That could not be further from the truth. When starting to debug dominion in the first assignment, I felt that I have a fairly good understanding of how dominion worked. However, during this final project, I have looked back on my past unit tests and card tests and realize that they could have been better implemented. This was due to my better understanding of dominion throughout the term. This just proves the fact that to be able to write effective tests, you have to have complete understanding of the API under test. It takes a deeper understanding of the code to be able to take the tool and manipulate it in a way to reveal faults.

I took the dominion implementations of the following users to test reliability and coverage: user “hollidac”, and user “houw”. To test the two dominion sources, I used my “testdominon.c” tester created in assignment 4 with a modification to return the error code if any of the dominion functions failed. I ran the testdominion tester three times with the following seeds to ensure a sufficient amount of coverage: 42, 43, and 44. I recorded the following code coverage for the two dominion implementations:

74.83% for dominion.c located in hollidac’s repository
78.03% for dominion.c located in houw’s repository

The coverage reported shows that a significant amount of the dominion code was covered in just three tests. Though coverage does not directly mean that both the test and the source code is good, it does usually correlate to fully functioning code. However by examining the discrepancy between the two coverages shows that the two implementations work differently. This increases the chances that the two dominion codes are functioning differently, meaning that at least one of the implementations is incorrect.

To verify this suspicion, I used the diffdom tool that was also created in assignment 4 to do differential testing. I examined the two outputs and discovered that the two dominion

implementations have the same output for the first part of the output. Then during one of the card plays, the houw dominion run decided to buy while hollidac decided to not buy a card. This small discrepancy made the two different test runs go in completely different directions. Since they both used the same seed for the test dominion run, the differences exist in how the dominion code is written.

However the issue with the differential testing is the fact that I did not have a working implementation at my disposal to compare the output of the testdominon run. This is essential for random testing to be effective. Though my testdominion.c file returns an error code if any of the dominion functions fail, without verifying that a working implementation works with my code without a return error, it is almost impossible to verify if the error codes are caused by an error in my code or is actually being caused by the dominion code.

I used this method to test holllidac's and houw's code because it had the most complete testing and the most coverage. Since we have not developed a full testing suite and some of my card tests and random tests overlapped, there was no other option in the arsenal of testing that we created during this class that would have as good of results as using the testdominion.c tool. The testdominion.c tool has the ability to execute all of the functions that the unit tests, the card tests, and the random tests which is appropriate for finding out the overall reliability of an API.

This is a perfect example of the difficulties of testing as a whole. It is almost impossible to completely verify if a program is working one hundred percent. The tools that we have developed over this term are just solutions to this problem so we can get closer and closer to that goal of determining if dominion is completely working. Each method has it's benefits and it deficits and we must weigh those to determine which test is best for a particular situation.

I overall really enjoyed the experience of testing dominion. It was a great example of what most software engineers experience in the real world: taking an existing piece of code and either using it or developing off of it. I have noticed this to be true in my own personal experience during different classes and internships. In a way and forced us to go through Agans' rules of debugging before we even learned what those rules are. Dominion is a game with a well documented API that we were able to get a grasp on and the source code is messy and yet simple enough for us to be challenged in understanding how it works and where the possible bugs are. For the most part, we had to use the testing techniques set out in every assignment to be able to fully get dominion working, even though that was not a requirement. This class even taught me how to actually use a debugging tool (GDB) that I had learned about before but never found the need or use for it until now. Testing dominion was a great experience that helped me discover the fun and value in testing software.

Word Count: 1050