# Final Test Report

Cameron McDonnell

Going into this class I don't think I had a good grasp of what testing really was. I had done some unit testing in the past but I now know that it was a bit half-baked and not as useful as it could have been. Going through the different approaches to testing and figuring out what kind of testing is useful for particular situations is a very useful skill to add to my coding abilities. Before this class I had very little experience with debugging code I didn't write, except for maybe code written by group members for projects. Looking at dominion.c for the first time was pretty intimidating (mainly because how long and unorganized it was). But after playing the card game in real life and getting to know the rules it was much easier to make sense of what each function was for and how the game should work.

I found unit testing to be useful for checking if an implementation works the way you intend for reasonable inputs. This seems particularly useful when writing code because you can create a unit test fairly quickly and it can show you what is wrong, or just to verify that something is working the way it is supposed to. I found it to be less useful when trying to find errors that occur with certain game states (ex: Hand is empty and you try and play a card). Unit tests are a great way to find errors in a particular part of a program, but not necessarily great for finding errors when parts are combined. The metaphor would be a mechanic working on a car with a dead headlight. The mechanic takes the battery out of the car, tests it, sees that its fine after testing, and moves on to the headlight. He removes the headlight, tests it, sees its working and can conclude that there is a problem somewhere in between the battery and the headlight. Now this is of course useful information but it takes time and the problem might only occur when it's raining or something weird which you might not even think to test for.

Random testing is definitely my favorite. It can help you find those weird niche cases that cause things to fail. Like if in the mechanic example a shop could test all the different weather conditions and different speeds the car is moving and see what combinations cause a failure. This is extremely useful and can bring deep rooted issues to the surface. One problem with random testing is reproducing a bug, I would sometimes find bugs in one of my implementations using random testing but couldn't reproduce the bug with a unit test because the state that caused the bug was difficult to reproduce. To solve this, you can save the state and analyze it, but it can still be difficult to spot the problem. In this sort of scenario Tarantula or Delta Debugging can really help quickly spot the problem.

One of the most important parts of debugging the game was actually knowing how to play. I was lucky to have a friend who owns the physical card game so I got a good grasp of how the game is supposed to work and what each card is supposed to do. If I had not played the game before I think it would have been much more difficult to debug the game. This experience gave me a lot of respect for testing because there are cases where you have to test code written by someone else and you might not fully understand what it is supposed to do. So before you even look at the code you have to do research to just figure out what it needs to do.

Overall I think this class made me a much better tester and a better programmer. In previous classes testing was an afterthought but this class showed me how powerful and important it can be. My initial tests were pretty weak but I think by the end of this class I can now write much more powerful and comprehensive tests.

**<u>Testing Smitjaco:</u>**

My code was fairly similar to Smitjaco's code. He chose some different cards to implement and test but he approached it in what seems to be a similar manner. He had overall better coverage of 71% where as mine was 62%, I Think this was because he chose to implement some of the cards with longer effects and I chose to do the shorter ones. As for reliability it is difficult to say who's is more reliable. Neither of our implementations caused crashes when randomly testing the overall game. I would maybe say that his is more reliable because he reworked the more complex cards and smoothed out some of those larger wrinkles.

**<u>Testing Danm:</u>**

Again we had similar code. Danm took my approach and also refactored the shorter cards like village and smithy. I could not get good overall coverage information because the random testing of the whole game would produce a segmentation fault. The seg fault occurs when the game is supposed to end and count points. I did not delve much deeper into finding this bug because I had already found some to report and could not find the definitive source of the error. His cards however had good coverage with his worst at 80%. I would say that my dominion is more reliable than his because it doesn't crash, but his coverage for cards is very good. I do think that most people in our class won't really have the most reliable code unless they did a big overhaul. The base code was pretty patch worked from the beginning.