My experience in testing dominion was rather tedious, though I presume that testing is tedious in and of itself. When writing unit tests I looked for every single small detail that could have been changed and wrote an assert for each line. I did this to absolutely make sure that the function did what it was programmed to do, and I did this for a few of the basic functions. I made sure that the game setup function worked correctly, as well as some of the other more simple functions. I initially did not want to stretch myself too far on writing those tests because I was rather lazy, but I see now that I should have gone for some of the more important functions so that I could use them to test other people's code more effectively. Ideally, I would have written a unit test for each function, but I had not the time nor the patience. There especially was no need to go to that great depth, because this is only a class. If I were developing for a game company, or some important agency like NASA I would be writing all those tests to make sure that my code works 100%.

Unit tests for small functions and basic game functions are fairly easy to test, because you can just make a struct that would hold all the possibilities relevant to that small function and test it from there. The hardest part about testing dominion is testing the individual cards, because there are a huge number of possibilities for the game state to have, and it would be impossible to go through each state and test them accordingly. This is where the card tester comes in. The card tester takes the same approach as the unit tests, but it has to be smart about it. The card tester makes some assumptions about what the game state is going to be so that it can narrow down the possibilities for the tester. These assumptions are a valid thing to make, as it would be a very slim chance that the number of players variable was set to 5 in the struct before it went into the function. This is then upgraded in the random card testers, where the tester generates random numbers for almost everything. It does sane things, such as randomly generating the number of cards based on the randomly generated number for each array of cards. It doesn't blast the card with nonsense, because that would generate a whole bunch of errors that wouldn't be seen in a normal, or even abnormal game state.

Writing these testers was tricky, because I had to keep in mind what these cards were supposed to do, and what they could even possibly do to the game. I did not have the foresight to make sure that the cards stayed within the bounds of the card effect, but I have learned about the dangers of letting loose a random function on the game state. What really helped me find the worst bugs was the random game tester. This found all sorts of infinite loops that I don't think I would have found otherwise. They were glaring errors, but if your attention wasn't called to them, you would be none the wiser.

On the coverage side of things, I found that my individual tests did not have good coverage rates. The unit tests were around 5%, but mostly because I picked small functions to test. The card test programs were a little bit better, with around 10-15%. The random card tester was around 20%, and the random game tester on the first run achieved around 60%. After being run 20 times with different seeds, the coverage reached 95%. I think that 5% might be attributed to some of the smaller functions that I didn't use to set up the game, specifically the kingdomcards function. This is fantastic coverage, and fortunately it has allowed me to find the bugs within the code that much better. During mutant testing, I was able to kill all mutants but one, which did not compile. My random tester printed out the entire game state during every turn, so it was easily able to spot differences in the output due to mutants. Very small things like embargo tokes not being placed, or but things like cards not existing. Overall, I found that the random tester was very fulfilling.

The code that my classmates had in their repositories was far from good. One classmate barely did anything to fix the code provided to us at the beginning of the term, so all existing bugs within the code remained. This makes the code far too unreliable to use, and it would need a major rewrite to fix all the small things that were wrong with it. Not to mention the bugs that he introduced at the beginning of the term. Another classmate found some bugs himself and corrected them, which made the tester run much smoother, but he still was not able to find some of the game breaking bugs that would send the game into an infinite loop.

The one dangerous thing about finding out the differences between your classmate's code and

your code, is that if your classmate's program runs into an infinite loop and you're writing the output to a file, it may very well fill up the rest of the space in your hard drive. I experienced this myself and ended up with a file that was around 30 GB. In the end, it was a very interesting experience trying to deduce what issues there were with other people's code and how we can find out what is wrong in the details.