Dennis Lee

CS 362

Experience Testing Dominion

Software Engineering II has given me the opportunity to develop unit tests, random tests, and a whole game generator. Originally, I had thought this class was going to be about writing the entire dominion game while writing unit tests for each function along the way, but it turned out to be different as we were given the system and asked to test it. This different perspective allowed me learn a lot about the process of debugging and how to approach a bug. Overall, I had a good experience testing dominion.

One of the first assignments was to refactor the code and write unit tests for functions. These assignments were a bit easier to complete as I learned about these concepts in Software Engineering I. In Software Engineering I, we previously wrote unit tests for the code that we made. This made it easier for me to be able to write unit tests for the dominion code. For my unit tests, I was able to achieve at least around a 90% coverage for the unit tests on the cards and other functions. The rest of the assignments were a bit more difficult as new material was introduced, such as random testers and gcov. I struggled a bit with the whole game generator as I needed to know all the functionality of the entire game. I feel that this struggle gave me useful experience in approaching a problem, as looking at the interface and header files can help give a better understanding of how the whole system works.

For the unit tests, I had a range of 88.24% to 95.24% test coverage. For the random testers, I was able to get a 98.28% test coverage for the adventurer card. For this smithy and village card, I was able to achieve a 98% and 100% test coverage. For the random tester for the whole simulation of the game, I was only able to get around 50% to 70% test coverage. Although my test coverage was average, I do not think that my dominion code would be reliable because only a few of the cards and functions were tested compared to all the other functions and cards that are in the dominion game. I think that by adding more tests, such as mutations and differential testing would be very beneficial to improving the reliability to my dominion code.

Classmate: nguyalex

To determine the reliability of the dominion code, I first started by looking and running their unit tests, random tests, and whole game generator. In their unittestresults.out file, their coverage for the card effects and unit tests ranged from 86.67% to 94.74% coverage, which is decent for these types of tests. I did notice that some of tests had failed assertions and I did not know if this was supposed to be a failed assertion that is supposed to fail or an actual failed assertion. The failed assertions are in the whoseTurn function, the outpost card, and the embargo card. The failed assertion for the embargo card comes from the fact that there is no coin increase. The failed assertion for the whoseTurn is pretty self-explainable, the function does not return the correct player’s turn.

The random tests done by nguyalex had good coverage. His random test adventurer card had a 93.75% coverage and all the tests run through the adventurer function did not crash. His next random tester for the smithy card had a 100% test coverage. The smithy card has a relatively easy functionality as it adds 3 cards to the user's hand. The last random tester was for the village card. This had a test coverage of 93.75% and passed all the tests. These results show that the adventurer, smithy, and village card are all reliable in his dominion code.

The final test to determine the reliability of his dominion code was to run his whole random game generator that simulates a game of dominion. His code was able to get a 45.39% coverage of the entire dominion.c code. His simulation of the game did not crash at all and was able to successfully able to determine the winner. Although this was a success, there is still too much that needs additional testing. In order to view this code as reliable, the bugs in the whoseTurn function, the other cards, and more coverage is needed. Ultimately, I believe that this dominion code may contain bugs that have not been found and needs more testing, but I doubt that anybody's code, including mine, could be seen as fully reliable as only a few cards and functions were tested during the duration of this course.

Classmate: sudarsar

To determine the reliability of this classmate's code, I used the same process as mentioned before. He was able to average around 82% test coverage for his unit and card tests. All of his tests had an assert statement that correctly passed, but some of the tests had a low coverage. For example, his unit test function for successfully creating a new game only covered 75% of the actual function.

The random testers were also relatively successful. It is shown in his code that the random testers that he implemented were run about 5000 times to ensure that the functions were running correctly. Compared to my implementation of the random testers, he was able to run more iterations to those particular cards to make them more reliable.

The whole game generator was able to execute about 67% test coverage. Similar to nguyalex's and my code, I do not think that I can say that his dominion code is more or less reliable. Although he was able to iterate through the random testers more, it was only for one function. There are plenty of functions and card functions that were not tested in dominion.

Conclusion

Overall, I learned a lot about how to approach and find a bug when introduced to a new system. Even though the dominion code was provided for us, it still took some time to understand what the buggy dominion code was trying to do. Although it was expected that the buggy code would make finding bugs easier, the game had to be understood first. The random testers along with unit tests that we implemented was a good idea to find any bugs. Ultimately, the lessons learned in this class gave me a useful experience and better understanding in testing when introduced to a new system.