Mihai Dan
Software Engineering II
Final Test Report

Dominion Testing Report

Introduction – General Testing Experience

Testing is imperative while developing software. Before taking Software Engineering II, the extent that my testing went to was checking if the function or program returned what was expected. I was aware that there was a branch of computer science that dealt with program testing, but I did not know it was as complex as it is. Testing Dominion was a rewarding experience, but definitely frustrating at times.

The biggest challenge I encountered when given this code was the lack of knowledge I had about the author's intentions. The source code for Dominion is long and uncommented, making it hard to know follow. I found myself spending a lot of time trying to understand and decipher the logic behind the code. The other challenge I was faced with was learning the rules to Dominion. Having no knowledge of the game prior to this class, I read over the game rules manual several times to get a grasp of the game. I referred to the game manual multiple times throughout testing in order to confirm that certain functions are behaving properly.

Performing unit tests on the Dominion source code was the first formal encounter I have had with testing. I quickly found out that unit testing is not the most efficient for such large pieces of code because of the sheer amount of tests one would need to write. The certain blocks of code I did test ran correctly after debugging, but there was still a lot more code that was not covered.

One solution to the inefficiency of unit tests was performing random tests and whole game generators. Random testing can be automated and allows for extraneous inputs or values to be tested. Some values may not be thought of when writing unit tests. When performing random testing on the Dominion source code, I was able to cover more cases with writing the same amount of code as a unit test. In my opinion, whole game generators are the best testing method for the purposes of this class. Automating a round of Dominion allows the tester to observe the behavior of the functions and check to see if the game is being played right. This method of testing yielded for the most coverage of my Dominion source code. This makes sense because it covered more code by actually playing a whole round of dominion.

Testing the entirety of the source code for Dominion is unrealistic due to its length and complexity. This class evaluated several methods of testing and getting as close as possible to achieving full coverage. I have a completely different view on testing now that I have been exposed to it and have practiced it. The importance of testing has not been as clear as after having taken this class. It has shown me that there is an appropriate method of testing for any type of code, whether it is unit or differential.

Code Coverage, Status, and Reliability of Classmates' Code

George Harder – *hargerg*

Before running any of the code, I expected the code coverages to be similar across the different versions of Dominion. This is partially due to the fact that we all started with the same Dominion source code, only performing some refactoring of it. When running my unit tests on George's code, I got a coverage of 32.80% of the Dominion source code. This result was close to the coverage of my source code, which was 34.95%.

The interesting difference was found in the random tests. When ran on my source code, the random card tests yielded a coverage of 44.98% each. When ran on George's source code, random card test 1 yielded 36.20%, random card test 2 yielded 41.22%, and random adventurer test yielded 43.19% coverage. I was surprised to see that my Dominion code achieved more coverage from the same tests. My assumption is that this is due to the refactoring that we had done on our source code.

The whole game generator led to the same coverage in both cases, at 31.00%. This came as no surprise because the game generator has a specific set of actions it covers, and it will cover them in both implementations.

The only adjustment that needed to be made to test George's code was due to the refactoring. I had to change some of the function calls in my unit tests to accommodate for the cards that he had refactored.

Pranav Ramesh – *rameshp*

The unit tests lead to a higher coverage in Pranav's source code than my own. The code coverage for Pranav's source code was 38.46% and mine was 34.95%, as previously mentioned. Again, my only thoughts behind this difference are that the refactoring altered coverage. This is shown in the difference the function calls for the cards refactored by Pranav. Calling a refactored card still goes through the cardEffect() function, while mine each have their own function call. The switch statement is traversed even if there was refactoring, leading to more coverage.

Opposite to the unit tests, the random card tests performed very poorly on Pranav's source code. I was shocked to see that the coverage for random card test 1, random card test 2, and random adventurer test were 32.72%, 31.84%, and 29.47% respectively. The source of this low coverage is a mystery to me. I ran the tests several times and got the same results.

No different than George, Pranav’s source code yielded a coverage of 30.95% from the whole game generator. The .05% difference can be accounted for by the slight difference in refactoring.

The slight alterations I had to make to the test files to work were changing function calls to adjust to the refactored cards done by Pranav. This was no different than what I had previously done for George’s source code.

Conclusion and Final Thoughts

Overall, testing the Dominion source code was as much rewarding as it was challenging. The most challenging aspect of this was diving in to someone else’s source code and understanding their logic. I had to understand the code fully in order to attempt to perform tests. Even with that in mind, I feel that tackling a big project such as this one was the best way to learn how to test.

We purposefully broke the game with extraneous test cases in order to check for correctness. On top of that, we performed random tests on certain functions and card effects. I think that the most effective method of testing we performed was implementing a whole game generator. This allowed us to see the game played automatically and check for correct output.

Testing Dominion introduced me to several strategies to implement when approaching a large document such as this one. It showed me that testing is more complex than I had previously thought. This knowledge is useful when creating any type of program, whether it is a game or something work related. Being exposed to the troubles that can arise when testing will be beneficial in future projects.