Jaydeep Rotithor

My Experience Testing Dominion

Throughout this course, I have used various methods to achieve good test coverage of the C implementation of the card game dominion. Initially, I refactored 5 of the card effects into their own functions and introduced bugs into 2 of them. These cards were: great hall, smithy, outpost, sea hag, and. After I did this, I created 4 unit tests for 4 cards. These unit tests were designed to see how well the card function achieved its desired effect. I also created 4 unit tests for various functions, and a constraint was that at least 2 of these functions had to be 5 or more lines of code long. I then implemented a random tester for 3 of my cards, and these cards were: adventurer, council room, and great hall. For the fourth assignment, I implemented a random tester of the entire game of dominion. I was able to get 60% coverage of dominion.c with this tester. I then compared my dominion implementation to that of a classmate and checked for differences and then documented the differences.

First of all, I refactored 5 cards. The cards that I refactored were: great hall, smithy, outpost, sea hag, and village. These functions were all fairly short, and so this was a very simple procedure. I also introduced bug in 2 of these cards, and I would later use various debugging methods to find the bugs that I introduced. All in all, this section of the class was very simple, and I did not encounter any difficulty in completing it.

For the second part of the class, I created unit tests for 4 of the cards and for 4 functions in the dominion code. I made sure that at least 2 of the unit tests for functions tested functions that were 5 or more lines of code. The cards that I tested were: great hall, smithy, gardens, and remodel. The functions that I tested were: isGameOver, numHandCards, fullDeckCount, and getCost. I was able to acquire coverage results for all of these functions, and while I did a good job of covering the functions, I did not do so well with regards to covering the entire dominion code, and this is likely because the functions that I tested were not very large. Also I only tested 8 functions total, which is a small subset of the actual number of functions in the entire dominion program. Throughout this assignment, I did not encounter much difficulty either in completing the required tasks, and this was because it was also relatively simple and had no hard coverage limits.

As part of the third assignment, I created a random tester for 3 cards. The cards that I tested were: adventurer, council room, and great hall. I had to achieve 100% line and branch coverage for at least one of the cards. I did this for the council room card. I was able to get 100% line coverage for the great hall card, but it had no branches so I did not count it toward the requirement. After completing this assignment, I determined through experimentation that the random testers had higher coverage than the unit tests. The reason for this was that the random testers tested a larger subset of inputs, including many inputs that would never occur in real life games. Unlike the last 2 assignments, I encountered some difficulties in completing this assignment. I had to alter my testing functions significantly to cover and adequately test all of the code in council room, since it was a large function of about 17 lines. Also, I did not get very good coverage figures for the adventurer card, coming in at about 34%.

The fourth assignment consisted of creating a random tester for the entire game of dominion. I was required to get a coverage level of 60% for all of the code in dominion.c, and to compare my

dominion to that of a classmate. I took the code from the leed repository and created a bash script that would tell me if the dominion results were equal or not. By testing this out, I found out that my implementation was not equal to that of my classmate. However, I could not tell which one was correct, or even if either was correct, since both implementations can be different and containing bugs at the same time. In order to compare the implementations, I created a bash script called diffdominion, and made it print TEST PASSED if the implementations were equal and TEST FAILED if the implementations were unequal, and the latter was what actually occurred. I found this assignment to be the most difficult of the first 4 assignments. This was because it is difficult to test out an entire game of dominion, as the code is about 1350 lines, and 60 percent of that requires me to get at least 810 lines of coverage with my tester. Also, I had to make sure that my bash script was compatible with both my and my classmate's implementation of dominion, and to do this, I had to move a few files around in my directory to make sure the paths were correct.

As part of the final project, I had to accomplish several tasks. The first task was to document 3 bugs in code from my classmates. I tested out their code and found that certain functions were not working properly, and so I documented the bugs and indicated what the correct code should have been. The second task was to document the process of finding a bug in my code. To do this, I ran my code using the UNIX server and inspected the outcomes. I found that the smithy function was not working properly, so I ran it through a debugger, and indeed its outputs were incorrect. I then inspected my code and found that one of the parameters in a function in the smithy implementation was wrong. The final task of this project was to choose one of 3 options, and I chose to do mutation testing. I found an online mutation tester, and with it I generated 10 mutants of my dominion.c program. I then ran my random dominion tester on each of these mutants, and the end result was that all of the mutants were killed. I figured that this was likely due to the mutation generator generating mutations that were semantically significantly different than the original.

Throughout this process, I used code from several of my classmates, and I got varying results on the reliability of their code. First, I used the dominion.c program from the leed repository for a run comparison. I was not able to completely assess the reliability of the implementation. This was because I only had my implementation to compare it to, and considering that I found a bug in my code, my implementation was not perfect. I think that their implementation was relatively reliable, because all of the outputs from their program made sense, and no variables were unnecessarily altered. However, I was only able to achieve 54.74% coverage overall on their code, as opposed to 60.64% for my own dominion.c. Second of all, I also tested some code from the steveric repository. I found that the code in this repository had a significant bug. I did not compare my code to that implementation, but through testing and visual inspection, I was able to find a significant bug within the feast card in the implementation. The rest of the code seemed relatively reliable, but that function seemed to alter the outputs from what they should have been. I achieved 53.04% coverage overall when testing this dominion.c implementation, but this was likely because the feast function here contained significantly fewer lines of code than mine did, lowering my overall coverage for this implementation.