Ramcharan Sudarsanam
CS 362

Test Report

Testing the dominion project has been an interesting experience. Up until now, in all the classes we had to write our own programs and then make sure we just got the right output for a certain input. Since this is rarely the case in real life, I think the experience of being given a codebase that we had no part in developing, and then being asked to understand it and find the possible errors in it was definitely a required experience. I've learnt that testing a codebase means a lot more than writing test cases like we did in 361 and also learnt about the different testing tools available for testing different aspects of a program. Moreover, following the path of the bug in the code is extremely hard in this codebase because of the large number of attributes and functions. Hence, it is ideal to use the testing tools that were taught in this class.

Code coverage was the initial metric I used for testing but I soon realized that test coverage only shows one side of the story. Certain functions such as the card effect are frequently used. The card effect function is responsible for the actions of a card but if a certain card isn't bought, then it is rare that the code ever gets executed leading to a low code coverage. For my unit tests, I ranged from 75% to 88% in test coverage. The possible reason for this wide range was the low number of lines in the tester file. Since the fail branch code is not executed when the test passes and vice versa and the number of lines are less, the coverage is low. I feel that this is where code coverage does not provide the right picture since some lines would obviously not run. In my card tests, I ranged from 83% to 85%. Unit tests and card tests were relatively easy tests to create but writing random tests for the whole game was particularly hard because of the need to have a complete idea of what's going on in the game. At first, my random testing was averaging around 58% and after increasing my loop count for random testing, it is now at 61%. This increase in code coverage was due to the different random values accessing different parts of the code. As far as reliability goes, I don't think that my code can be called reliable as of yet. Right now, only certain parts of the code such as certain cards and functions were tested closely. I think that even though my code coverage is 61%, adding more testing techniques like mutation testing will strengthen reliability confidence and push out any bugs that have been missed.

The classmate's code I tested was Dennis Lee (leed). To get an idea of his code's reliability, I again looked into the code coverage and the pass/fail outputs of those tests. His unit test coverage ranges from 88% to 92% which I think is reasonable. His card test code coverage ranges from 90% to 95%, but one of his card test fails. His adventurer card does not pass, however. The biggest issue I noticed in his code was that when I ran his random test for the whole game, it kept crashing randomly. After investigating using print statements and divide and conquer, when the remodel card was played with an empty hand, the cost of a null pointer was looked up which was causing the seg fault. Although running the code with an empty hand would be rare, it is still an error that needs to be fixed. Since it is actually causing a crash of the program, it could be labelled as a moderate level error. Other than this, most of his code was executed as expected. Dennis' code coverage for dominion in the random test for the whole game was 57%. Although this is not the best level for code coverage, I thought that it was a reasonable amount since I and a lot of people got a code coverage close to this value. As far as reliability goes, I think that as most of the people's code, it is not reliable. Just like mine and Alex's, the code coverage is too low for dominion to be called reliable. Since the seg fault occurred after running the program just for a couple of time, I think that his code cannot be labelled reliable.

I also tested Alex Nguyen's (ngyualex) dominion code and I looked to his unit and card tests to get an idea of his code's reliability. In his unittestresults.out file, it can be seen that his unit card tests have a range of 86% to 93% which is higher than what I had. However, some of his assertions failed in the tests. It is not clear if he intended to them to fail or they actually failed due errors in the dominion code implementation. The outpost card, the embargo card and the whoseTurn function failed. After going thorough his code, I noticed that the whoseTurn function fails to return the correct player's turn and the embargo card does not increase the coins. Moving on to the code coverage, I noticed that his random test for adventurer had a really good coverage of 93%. Moreover, his tests adventurer card including the random test for the whole game never crashed and the values were right for the adventurer card. The random test function for village also had 93% coverage and passed all tests. His random tester for smithy card had 100% test coverage. So, to test the reliability of his code, I ran the random test for the whole game and he achieved a code coverage of 45% of dominion. I think that this value is on the lower side and his code needs to be tested by other methods. However, it is to be noted that his code did not crash. But, I would still rate his code as unreliable due to the low test coverage and the bugs in his whoseTurn function and embargo code. I think that more testing will lead to more bugs that are lurking the code.

In conclusion, I thought that testing dominion was a pretty difficult experience but it was necessary for us to understand the problems faced when testing a program. Understanding what to test, learning how the program is actually supposed to work, and writing the code to implement the tests were difficult. I thought that if we could employ other methods to test this program rather than mainly limiting to coverage, we would be able to narrow down on bugs and solving them. Searching online, I saw a lot of tools to solve bugs, but finding resources to search for the bugs was difficult. Coming back to the point I made in the first paragraph, refactoring and understand the unfamiliar codebase was definitely a big challenge.