Yipeng Song

Alex Groce

CS 362

6 June 2016

Test Report

My first impression of the dominion code is terrible. The code is unstructured and hard to read. What's more, I have never been writing test code for such a complicated card game in my previous coding experience. In order to write test code for the dominion.c, I search the game rule online for this game and try to fully understand how I should play this game.

Through research, I've found that the goal of the game of Dominion is to build a deck to earn more victory points than one's opponent(s). Action, coin, and victory cards can be purchased from the supplier using coins from a player's hand during his or her run. Each play is allocated a starting deck of 10 cards, 7 coopers and 3 estates. Each turn consists of an action and a buy phase, with the turn initially have one allowed action and one allowed buy. Actions cards that a player is able to play from his or her hand may increase the number of cards in the hand, the number of buys he or she can make, or the number of actions can take. The game ends when all of the providence cards have been purchased, or three sets of the cards are all purchased from the supplier. The winner is one whose deck has the highest victory point score at the end of the game.

After understanding the rule of this game, I ran the code and succeeded in playing the game for the first few times. However, I was running into errors as I kept playing it. While the game is playable, it is not reliable for the end user. I take the time to find the evidence that the dominion code does have bugs.

To achieve this goal, I implement 8 unit test cases, 2 random tests and 1 random game playing test. The 8 unit test cases are focused on unit testing of 4 functions in the game of Dominion code that we all start with and testing the implementation of 4 action cards in the cardEffect function.

For the four functions, I choose buyCard(), updateCoins(), numHandCards(), and getCost(). For the four action cards, I choose treasure_map, salvager, great_hall, and outpost. These unit tests are pretty straight forward. We are testing to see if the functions are working properly. For the action cards, I try a more rigorous testing method, which is simulating the game and add some assertions to see if the card has been played and if the value is correct when the card is played. After implementing, I run all tests and get coverage of 33.92% of the dominion code. This is respectable because I am able to only implement and run a couple of tests across 8 functions. There is still a lot of code that is left untested.

For the random test, we are tasked to test the adventurer card. The random adventure test is initialized with random cards and a random number of players and those players would have 5 random cards to start with. In the adventure function, I notice that once the adventure card has been played, it will not discard the cards in play that have been drawn if the counter of the temp hand is equal to 1. This will disrupt the hand count and might cause the game to crash or not work properly. To fix it, I need to change the condition in the second while loop from z –1 >= 1 to z – 1 >= 0.

After random testing for cards, we are asked to create a random tester that will play a complete game to fully test our dominion code and find any more bugs. This is hard to implement. Because we first need to know what the code is doing at every step before writing the actual code. What's more, I need to make sure that every card has been written into my tester. It is a teasing process that have me going through the dominion code over and over again. My test suite does not find any bugs, however, looking at the outputs of the system revealed a few unique cases where cards did not behave as expected again. With my entire test suite, I am able to achieve over 80 percent coverage of the dominion code.

Besides these, I also use my tester to test my classmates' dominion code. All three of the peers that I have tested so far on their dominion code, the game is playable but none of them are fully correct. But the good news is, all the test coverage of those three classmates are above 60 percent. Through full game random tests it shows that mine and a lot of peers' results are fairly similar including some bugs because we are testing a very similar dominion code. However, a decent amount of bugs and results vary quite a lot. This just shows that full game testing is mainly good for comparison of other's code rather than finding the problems within the code as it lacks detailed searching.

It has been a very valuable experience for me as I have never tested a code as big as dominion before. It has also been an effective learning experience for me to witness all the possible test methods that could be useful for a variety of situations. Through the whole testing dominion code, I feel it has made a large improvement in the way I design, implement and test code. I feel this way because if there is a large strength in the basics of programming it can drastically reduce the amount of bugs that you might come across along the way. While as a programmer I was already comfortable with unit tests this process has opened my eyes to random testing and shown how useful this could be to reduce human error. I feel that a lot of what I have learned from this course is directly applicable to other situations outside of this project.