Experience Testing Dominion

When I first starting testing I was overwhelmed. Prior to this class I have had little to no experience testing such a complex and large program that wasn't coded by me. Perhaps the most frustrating thing about this dominion code wasn't the abundance of bugs, but the lack of documentation and comments. This made understanding the code very difficult since it is a relatively long program. However the format of the class was structured in such a way that allowed me to slowly ease myself into the functionality of the code. Unit tests are the most basic of tests, and required little knowledge of the how the code tied together, and therefore were easy to implement. Random testing required knowledge of which variables could differ from game to game and from turn to turn such as the deck size, discard size, hand size, player count, card placement, and kingdom cards. It is crucial to random testing that we simulate many different states of the game in order to get coverage across lots of the code. This is where bugs could be hiding.

The most difficult part of testing dominion for me was creating the full game random tester. Creating a full random game of dominion requires complete knowledge of the code base and how they tie together. I was surprised by how many different functions there were to complete different tasks in dominion. This made it easy in the way that I did not have to hard code any dominion functionality so most of the test code was calling the correct functions at the right times with the right arguments. Through creating the test dominion code I realized how many different ways one could go about implementing a random game of dominion. When I was looking at the other repositories for inspiration on how to structure my game test code, I realized that views on how to best implement this varies from person to person, so the only way testing dominion would make sense to me would be to code it myself from top to bottom. Since my full random tester averaged about 600 turns to finish a game due to its randomness, when checking the differentials when run with two different dominion implementations it was sometimes hard to decipher what is different or went wrong. Bugs that change the score for instance would only cascade into larger score differentials as the game progressed, and different card purchases/plays would change all the results from then on when compared to another dominion implementation. When I saw such a difference between two different dominion outputs, I was overwhelmed and didn't know where to start looking for bugs. I think full game testing is useful when code has most bugs ironed out via unit tests and small random testers. Otherwise, the dominion codes differ too much.

The most pleasant surprise during my testing was the implementation of tarantula fault localization. When I first learned about it in lecture, I thought that it would be difficult to implement but by utilizing gcov and python, it was a pretty simple and short program. The hardest part was the trial and error of getting the parser to

correctly extract the line executions from the file and correctly formatting the input of the tarantula program. The output of the tarantula program was extremely valuable. It allowed me to zero in on the fault as soon as the program completed. Searching through a file with numbers ranging from 0 to 1 based on suspiciousness is much easier than examining the output of multiple gcov output files and keeping track manually of what is executed during failed tests and passed tests.

Status and view of reliability of Mckenna Jones' Dominion Implementation

This dominion implementation is not reliable. Through my tarantula implementation on his code trying to find out why steward failed during some executions, I realized that he did not implement his refactoring correctly. I saw that when I was calling steward it was also executing code inside the tribute case directly below steward. This is a critical error. The bug is that the cases in card effect were incorrect. Cases must return values in c, or else when the case is called it will keep executing code farther down below. Mckenna was only calling the refactored card function, not returning its value. This means that every time one of those functions is called, it will try to execute code that it shouldn't be. Another bug is in the scoreFor function which was caught by my unit test and my full tester. The turns should start with 3 points for each player however scoreFor only iterates up to discard count when searching through the deck. So if the discard count is less than the deck count, it will never iterate fully through the deck, missing some cards that would affect the score. This bug is easy to miss because the moment the discard deck reaches equal or more than the size of the deck size, the scoreFor function will operate like expected. However anyone with knowledge on how dominion is played will know that 3 estate cards are initialized in each player's decks at the beginning of the game. This should be caught during the first few turns of the full game tester or in a scoreFor unit test. Due to these bugs the status of this dominion code currently is functional but very incorrect, with many bugs.

Status and view of reliability of Jake Jeffrey's Dominion Implementation

Jakes code was plagued by the same bug Mckenna's was in that the scoreFor function was not implemented correctly. Even though this is a 2 second fix, it is easy to overlook because it does not cause unusual behavior past the first few turns and does not cause errors or segfaults. This only minorly affects the reliability of the code. Jake implemented his refactoring correctly, so his code is reliable in that his card effects should run predictably. The only other bug that was caught through my test suite was the Great Hall Action card. This, like the scoreFor function is not very severe and is a 2 second fix as well therefore it does not affect reliability too much.

Code Coverage

For both dominion implementations, my test suite, which is the combination of my random testers, unit tests, and full game tester only found these bugs. Regarding the code coverage of my test suite, I would only consider Jake's code reliable only in the scope of my tests. Mckenna's code definitely needs to have these bugs corrected before further testing, which is very necessary. Even though my unit tests have a code coverage of 30.38%, which is relatively good considering it only tested 4 functions and 4 cards, I would not consider testing to be complete. This percentage may seem good at first sight but I have the knowledge that many functions have gone untested. This is indication that code coverage is not something to solely base the production of tests on.

My full game tester resulted in an average of 73% code coverage. This also seems like a good percentage but most bugs go undetected because they do not follow a certain branch of execution. So even though code may have been executed, it may not have been executed in the right game state with the correct precondition to create an error/fault. Even though I made an effort to make the AI as unpredictable as possible, I would not be completely confident that a dominion implementation is bug-free even if it matched the same output when run with another dominion code implementation. Given more time, I would manually implement edge cases and certain branches of plays to my full game tester to make it more thorough.