Matthew Zakrevsky
Cs362: Software Engineering II
Spring 2015

My experience testing dominion was interesting at best. This is due to the nature of the dominion source code. This source code is inconsistent in style and implementation. The code looks like it was given to fifteen people, cut up and then combined into a Frankensteinian monster of such proportions that even Mary Shelly would be unable to recognize it. There is so little inconsistency that trying to write code that made use of the source code was half of the challenge. This lead to overall low coverage and low level testing as a result of being unable to understand what certain functions were supposed to perform and what you needed to pass in in order to make use of those functions.

In my unit testing I found it difficult to make use of certain cards as a result of how the playCard and the cardEffect functions were written. These functions were best used in a full game implementation of the dominion due to the nature of how the functions interacted with the game state. Many of the time the cards that I wanted to test were prevented from being played as a result of having to have a least a single action available in order to play the card. In my feast test I found that even manually setting the number of actions available would still result in the cards not being played. This was compounded with the requirement of having to have other various fields such as still having a valid number of cards within the state to work with in order to have the correct results from playing a certain card.

In full game implementation it was easier to get the functions to perform as you wanted them to. This is due to the constant reset every turn that will allow certain functions to perform the as they are supposed. Since the hand is replaced every turn cards such as mine or remodel have the chance to complete their functionality unlike with unit testing where you had to manually recognize that you needed to refill the or shuffle a deck to get the correct results. There was less stretching and jumping through hoops to replicate a game when isolating a specific function or card effect. In the case of unittest1.c where I was testing the scoring function the number of times I needed to use gainCard, which I feel is a sub-par function to buyCard, was much more than necessary.

The results of differential testing also had some eye opening results that were produced when their code was used for testing a full game implementation. In may cases there was very little change between the results of the implementation(see diffresults.out), often no more than a different card being played or a being bought. The more serious of problem such as a game ending earlier than another was often a result of a more serious bug from a random card such as a feast being played. In many cases the testing bias created by the simple choice tree would result in invalid card being played as the number of treasure or victory cards outweighed the number of action cards that were in the deck. In comparison to my class mate's implementation that I used for the differential testing there were very little changes and as a result the differential testing was very similar in comparison to my own implementation.

I believe that the overall reliability of my classmates' dominion code is very low. There are many bugs in this code that were not fixed from the original file that continue to persist through the code. Much of this code is not functional and unreliable at best and often results in incorrect output and interactions in testing. This is the case in not only whole game testing implementations, but also in unit testing of specific functions and card implementations of their code. Many cards still have major bugs such as infinite loops, bounds errors, and often result in adverse actions that can ruin the state of the game in many small ways. This is often seen in lack of handling of invalid actions that will force the state to do unreasonable action such as always changing the score or playing cards that should not be played as action cards, such as victory point and treasure cards.

In my opinion the overall reliability of the dominion code is very low and while my own testing was not of the best quality. This is an opinion that has been formed through my experience and the overall coverage of of my classmates' code. While I can produce only about 50.26% using my own

implementation I believe there is room for improvement that can be done within the dominion.c file that each of use worked with over the course of the term. When testing Andrew Gass's code and Harrison Kaiser's code I found that my overall test coverage with their implementation of dominion that my testing only completed 49.66% and 46.19% respectively. Now this is a low number but it is important to note that this test was very randomized in nature but had static kingdom cards.

This is to be expected as a result of how badly written the dominion code was written and how I chose to implement the random tester. The major reason that this was the case was due to the infinite loops that occurred in three of the card effects that I did not have time to go and fix in order to make sure that I had a larger random set of kingdom cards to work with. If I had implemented the random kingdom selection the overall coverage would have rose as there would have been a larger set of cards to work with thus increasing the overall coverage.

The major changes I would make to this dominion file would be to rebuild the entire dominion.c file as an object oriented based design that would use each card as an object. This would allow more concise functionality and allow cleaner source files. This would help to handle the major discrepancies that are found as a result of the many different authors working on the file from over the last few years.

Overall this was a very good course to have taken. I learned a lot regarding the major ideas of debugging and testing, how to write effective tests, understanding code written by others, learning how to use code written by others, and making use of new and important programs on the Oregon State University servers that will be able to make use of in the future. The skills that I have leanred in the last term have been very useful in understanding the concepts of Software Engineering I.(fun fact you can take both at the same time). I will remember what I learned and will incorporate it into my tool box for accomplishing future goals that I want to complete.