Isaac Chan
6/4/16
Dr. Alex Groce

Dominion Test Report

Testing Dominion:

From my previous projects, I scoffed at the idea of testing one game for the whole 10 week term. It was a humbling experience once I started and realized the scope of the game. It is a whole other story to test a program that I had no input in creating. Reading through and gaining an understanding of a game was a difficult initial requirement. The code that is given is such a mess that just following through it was a difficult task. Once I started reading the code in cardEffects, the game began to make sense. I spent a lot of my time trying to trace through code.

From past projects, I was familiar with unit tests. These unit tests were written as the code was written, so it was relatively easy to slowly build up a test suite that would test parts of the code. I quickly realized that the unit tests I knew were extremely insufficient for such a large and complex program. Just returning a value and affirming that the output was correct was not at all adequate for the dominion game. After assignment 2, I saw my 30% code coverage and realized that there was a lot more to testing than I realized.

Enter assignment 3. Random testing was a brand new concept I have never thought of before. I had to again affirm my understanding with the card parameters to achieve 100% coverage. I really enjoyed doing random testing and I think it's an elegant method to brute-force test code.

I enjoyed doing differential testing, although I don't think I got a lot of success with it. It would have been better if a working implementation of dominion was supplied. But that might have made finding bugs too easy. Everyone's code that I tested was exactly the same as mine, so I didn't have really any results from it.

Code Coverage Information, Status and Reliability of Classmates' Dominion Code:

melloc:

My unit tests, card tests, and random tests all pass on melloc's code. I was surprised to see them achieve 3-4% higher coverage on melloc's dominion implementation. The only difference that I was able to note from static analysis was that we had different refactored cards. However, since mine barely crossed the threshold of 30%, this result may not have any meaning.

My random tests ran much slower on melloc's code than mine. I suspect that my feast card refactor may have had an unfavorable effect on the output. I think that the code would run through mine with incorrect parameters and instantly return 0, when it was actually working in his un-refactored cardEffect.

Ultimately, the status and reliability of melloc's dominion code was basically the same as mine: minimally changed from the base dominion. The results we had were extremely similar, give or take a couple extra lines of coverage. The only differences we had were the refactored cards, in which I might've broken the feast card. I cannot be reliably quoted on either dominion implementation being correct, given the minimal differences. Since the majority of my tests passed on both games though, I can assume that neither game is very broken.

pengs:
Again, my unit tests, card tests, and random tests passed on pengs's code. Again, they achieved minimally higher coverage than mine. This time, I decided to check on the cards he chose to refactor. He chose simpler cards than I did, including smithy, remodel, and mine. I think this may have been a contributing factor in the higher test coverage. The likelihood of an incorrect refactor with my cards is much higher.

However, this time with my whole-game test, the test would freeze. I could not compare my implementation versus his with my difftest tool. I did not spend too much time on it, but this leads me to not being able to draw any conclusions about his implementation. It does not give me much to compare to my 62% coverage on my game.

With the failure of testing pengs's dominion.c, this once again brings me back to the encompassing difficulty of creating a test suite for the entire dominion game. Even though I can say with pride that I achieved 62% coverage with a random tester, this still leaves 38% of lines that were untouched. That's more than 500 lines that were skipped. Making a complete test suite for the complex game of dominion is incredibly difficult.

Final thoughts:
Dominion is a tough game to test. Even given a perfect implementation, there is a significant barrier to entry of initial complexity for even understanding the game. With our buggy, incorrect implementation, it was an even greater struggle to trace through the spaghetti code. Additionally, the complex and specific rules had to be understood in order to catch the edge cases.

Built into the code were an abundance of gamestate-reliant cases. This made getting high code coverage extremely difficult. I can't imagine random testing being the method to reach 100% and would guess hundreds of specific unit tests being the only way.

From the view of a tester, there are several ways to work with such a program. As I found throughout the term, my understanding of the game was often flawed and not as robust as I had originally thought. It's incredibly important to know how the game is supposed to be played, and understand the nuances in game and card rules. This would make tracing through the program much more enlightening.

I also found that I became fixated on achieving higher code coverage. However, code coverage is not the only relevant statistic. It is easy to get a high percentage if I wrote tons of unit tests to hit those cases that a random tester is unlikely to find. However, unit tests only can test the output of a function and with a complex game like dominion, the tester needs to see how functions interact, not just to make sure a function can work by itself with ideal inputs.

Random tests are another way to test code and coupled with unit tests, can be a strong tool to test for a robust final program that simple output-verifying unit tests cannot do by themselves.

Overall, I learned a lot about becoming a more effective software tester by tackling the dominion game. Strategies and lessons learned gave me a better understanding and experience in testing a complex program.