Emily Longman
6/6/16
CS362

Test Report

An exploration of different testing methods in Software Engineering

Using an implementation of dominion to learn different forms of testing in this class has been a great experience. Through it I've been able to learn the strengths and weaknesses of techniques like unit testing, random testing, different forms of coverage, and differential testing. Having no knowledge of the rules of dominion beforehand, learning how our patched together version of it worked was a challenge of its own. Thankfully the initial refactoring assignment gave me a chance to understand the structure of the code a little better and to introduce my own bugs for later use.

The second assignment where we created unit and card tests gave us the first testing challenge. This was tough for me as I had thought I understood unit tests after Software Engineering I, but I realized I tended to write ineffective one. My attempts at these and the card tests yielded low coverage numbers. It's obvious now that I was only really testing a few lines in each. Live and learn.

The next assignment may have been my favorite, as I found random testing interesting and much more effective than my pitiful unit tests. I was able to get roughly 30% coverage on the whole game which is passable but certainly not optimal. I was able to get a couple cards specifically up to 100% coverage with random testing and a lot runs. This meant that there were a lot of bugs elsewhere in dominion that I hadn't even touched.

Fourth came our full game tester which was interesting because it was surprisingly straightforward to test such a seemingly vast amount of code. In the real world of course dominion is a tiny project. Focusing on the game as a whole rather than hoping to get coverage through the small window of a few functions was much better for my numbers. I doubled my coverage, reaching 62% on mine and 61% on a classmate's. Differential testing was interesting and an obviously useful tool, but was somewhat cumbersome with dominion. It was difficult to tell which implementation was "correct" as there are so many oddities in the code. In a case where the outcomes are more cut and dried, as well as with a program that outputs data in a more readable way, this would be a great tool in one's testing belt.

Finally the mutant testing that I did for this project provided what seems like a way to test one's tester. I spawned nearly 3000 mutants which covered a large portion of possibilities. However the outcomes of testing and killing these mutants seemed to provide more information on the effectiveness of my test suite than on the correctness of the code itself. While it's coverage was broad, it was tough to use the mutants themselves to uncover and locate errors since they were often introducing ones of their own.

These were my own experiences with testing in this class, but how well did I do compared to others? I decided to compare my numbers to two other students in the class, Dennis Lee and Trevor Hammock. I looked at their coverage percentages and how they did things differently than me so see what these unique solutions would yield.

First let's look at Dennis. When running his tests it took me about 5 seconds to realize that he had been far, far more thorough than I had. His unit tests covered roughly 90% each, which made me never want to look at mine again. I gained a very small portion of my pride back when looking at his random tester, on which his best coverage was 98% on one of the cards, while I was able to get 100%. This was a short lived victory though because when I increased the number of runs he also reached 100%. When it came to testing the entire game he and I were similar, both right around the 60% mark. This I think is more a sign of how much junk code is in dominion rather than a shortcoming of our testers. On the whole his tests had slightly more coverage than mine, but I don't think that either of us made dominion any more reliable, we simply agreed that nearly 40% of it is rarely used.

This was just one other person though, how does another classmate compare? Trevor has done some serious work on making the entirety of dominion a little less of a monster and it shows. His unit tests were able to get around 50% coverage on the whole game, which is awesome. He also of course had 100% coverage on the unit tests themselves. His random tests kept this lofty performance going, also having 100% coverage on their own. He was also able to achieve 100% on the *whole thing* when they were run enough times. That's just crazy. His optimized and patched version of dominion is almost better than his tests. His tester for the entire game can achieve about 60% in just one run. In 20 it got up to about 90%. My tests couldn't say much for the reliability of dominion, but his clearly show that his dominion is far better and more reliable.

So what does all this information mean? What it boils down to is that I've learned a lot about testing methods and their effectiveness. I've also learned when they're ineffective, and how to choose the best testing methods for the situation. Coverage is important but it isn't everything. Being able to locate the problems and fix them without breaking more things is equally important. I came into this class expecting it to be a little dry, and certainly not fun, but I ended up learning a lot more than I expected. I see now why this is a requirement for the capstone project; I never would have been able to adequately assert the validity of my code without some of the techniques and concepts of this class.