Devin Foulger
CS 362
June 6th 2016

CS362 Report

It is my opinion that this implementation of dominion is acceptable. Using different forms of testing I have come to the conclusion that this form of dominion is playable, as there are no jarring bugs and crashes. However, testing dominion was a larger task on its own to accomplish. I used many methods of testing such as unit testing, random testing, and testing whole games of dominion. I also came across smaller issues that aren't bugs when testing the other student's dominion implementation. Some of these problems consisted of the changing of dominion in a way that would make it shorter, or that the naming conventions are not the same.

Testing dominion was a challenging task to accomplish. One thing that I learned was that even though a test may cover all lines in a function, that doesn't mean that the test is actually a good test. However, out of all the testing methods used, I feel that random testing provided the most when testing dominion. Random testing allowed me to play the game in bizarre ways and it also allowed me to gather unusual results. It also provided a way to gain high code coverage, especially when randomizing the kingdom cards that are used. On the other hand, unit testing provides the programmer with the ability to code very specifically. For example, it would be possible for me to program a very specific AI that only plays dominion in a certain way. This is how I approached testing dominion as a whole. The player one AI would only purchase and play certain cards, whereas player two would do the same except it would play different cards compared to player one. This allowed me to gather many different results and cover quite a large portion of the code. Unit testing also helps if you want to think like a player. If you know how to play dominion, you know which moves a player may take and which cards they may trash. It also allows you to know what results you should be receiving at the end of each function. However, this is why I think unit testing sometimes falls short. You have to make a lot of assumptions about the code. This goes against one of Agans' rules and I have experienced firsthand what can happen if you make assumptions about the code. Mutation testing also played a large role in determining if my test suites were even capable of catching errors. My test suite managed to kill 133 of the 157 mutants that were generated. Overall, I was able to hit 61 percent code coverage with my project. I'd say that this means dominion is reliable.

In order to test student's code with my own tests, I had downloaded their github repos. From there, I removed their tests and included my own as well as my makefile. This allowed for me to use my tests. The first student's code passed all of my tests. However, there were a couple of complications. For example, the user did not have a specific function that I had created for the card tests, so I wasn't able to test that card specifically. Luckily, with my implementation of the AI from assignment 4, I was able to test quite a few different cards. This student's dominion code also seemed to be very similar to mine. This particular student's dominion crashed when running my random card test which tested village. I think this is due to the fact that they changed their trash flag to something else, but I wouldn't necessarily call this a bug. I was able to reach 52 percent code coverage with this particular student's code. I am unsure as to why it isn't closer to 60 percent, as it doesn't seem to be much different compared to mine.

The second student's code also passed all of my tests. Again, there were some minor complications though, like functions that I had created the student doesn't have. Their implementation of dominion may have been different to mine though. I was able to achieve 64 percent code coverage with the implementation of dominion. This is higher than what I even accomplished on my own implementation. Overall, I would say that the results I received were similar to what I experienced with the first student. However, for some odd reason, my random tests received make errors. I suspect this is due to their implementation of dominion, as I stated earlier. If they had removed something and moved it to an external h file, then this would make sense as to why I would have higher test coverage and why my random tests don't work. The AI that I created ran flawlessly though. I think with the full game player, I was able to achieve such high code coverage. The code coverage here also leads me to believe that this version of dominion is reliable.

In the end, both students implementations of dominion managed to pass all of my tests. Of course there were a few problems regarding the card tests because I did not share the same naming conventions as other students. I also may have not even picked the same cards that they decided to refactor. This ultimately would make it hard to test their functions unless I completely remade my card tests depending on the cards they picked and how they implemented them. Overall, testing other students code also confirmed that this version of dominion is indeed reliable. Code coverage would range anywhere from 52 to 64 percent. These numbers help provide a general idea that the code is reliable. I also think that because both users implementation managed to pass and use a large portion of my test, it also shows how reliable dominion is. In the end, even if the code is buggy in some places, the game is still playable. It is especially true, because unlike random testing, users will be making logical choices that won't result in odd situations.