George Harder
Dr. Alex Groce
Software Engineering II- Final Report

Dominion Testing Report

General Comments on Testing Dominion

Testing Dominion was a fruitful if at times frustrating learning experience. To begin, one of the most challenging things about this undertaking was the inherent requirement that I, as the tester, had a working understanding of the Dominion code base. This was certainly easier said than done. Because of the cobbled-together nature of the code I spent much of my time trying to understand how all of the pieces fit together.

The next thing I quickly realized is that writing unit tests alone is an insufficient method for detecting and resolving bugs in such a large and complex program. While verifying that return values from function are valid and not indicating that an error occurred is important, it barely scratches the surface in terms of what would be considered a strong test suite for a program such as this.

Useful tools to fill the gap left by using unit tests alone include random testing and whole game random testing. I found that writing these testers was more interesting that writing unit tests and card tests alone. The random testing and whole game testing required a bit more critical thinking, which is one of the aspects of programming that I most enjoy.

When it comes down to it, accounting for all or most of the possible states of an intricate and lengthy program that plays a game like Dominion is a near impossible task. However, the tools that we learned in this class allow us testers to tackle this problem using a series of indirect but nonetheless effective methods. As I mentioned above, random testing and the whole game random player coupled with differential testing can give us some level of confidence that we have a functioning and playable version of Dominion.

Code Coverage, Status, and Reliability of Classmates' Dominion

danm

When running my unit tests and card tests on danm's code, I was able to cover 36% of the code. This was surprising to me, because running the unit tests and card tests on my own code only covered 32% of the lines. While this may not necessarily be significant, it is an interesting result. Adding in the random tests did not increase the code coverage significantly.

One disappointing albeit potentially insightful thing about running the whole game generator on danm's code was that it did not complete within five minutes. I have been struggling to figure out why this may be the case. The only significant differences between our code, based on brief static analysis, are the cards we refactored for Assignment 1. I am not sure why this would be causing the game generator to run for an unreasonable amount of time.

Because my generator fails on this implementation I am unable to confidently say that it is reliable code. However, because the unit tests and card tests pass I doubt it is severely flawed. If the game generator ran correctly I would be able to acquire more meaningful insights.

rameshp

Running my unit tests and card tests against rameshp's code yields similar coverage results to that of danm's. Once again I was surprised to find that this code was covered at a slightly higher rate than my own.

Once again I encountered the frustrating problem of my whole game generator not completing in a reasonable amount of time. This is especially vexing because on my version of dominion the generator completes almost instantly and raises the code coverage to over 65%. If I had been able to complete the whole game generation, I would have gained a lot more useful information about the state of rameshp's code.

The status of rameshp's code is, unfortunately not totally clear to me. I miss a lot of the code due to the complexity of the Dominion code base. Even if the whole game generator had completed, and we assume that it cover's roughly the same percentage of lines, we would still be missing about 30% of the code. This is likely a function of the many different conditionals that are not triggered during normal game play. This issue presents a significant roadblock to testing Dominion.

Summary and Conclusions

There are several major obstacles to testing Dominion. First, it is a difficult program to fully grasp. There is a multitude of functions and helper functions that are pieced together in a manner that is not always intuitive. Second, the code has to deal with many of the complex edge cases presented by dominion rules. In addition, there are a lot of possible error states that are accounted for in the code. This results in low coverage results for unit tests, card tests, random tests, and even the whole game tester that I created for Assignment 4. Lastly, understanding results from tests is a non-trivial task.

As a tester we have options to confront these issues. A strong understanding of the advantages and disadvantages of code coverage is an important one. While a high rate of coverage is frequently a good barometer of the quality of a test suite, it is not the only or the best one. If we wrote many more unit and card tests to cover the numerous functions in dominion.c we could vastly raise our total test coverage. However, this would not necessarily mean that our test suite was stronger. Because the functions in dominion.c fit together in such a complex manner, just verifying that they have correct return values when given nice inputs is not a useful way to test.

To counter the failings of simple unit tests, we can implement repeated random tests. In addition, we created a random whole game generator. These additional testing methods add variety and strength to our test suite in a way that simply raising test coverage cannot.

On the whole, testing Dominion taught me a lot about some of the difficulties and strategies to overcome said difficulties of testing complex systems. It gave me a chance to practice concepts we learned in class and gain a better understanding of software testing.