Nicholas Desilets

Professor Groce

CS 362

Final Project Report

When I first started this class and first started playing around with this implementation of dominion it appeared to work well enough as I hadn't noticed any odd behavior. While the code itself looked horrific in terms of it's consistency and formatting, the behavior and functionality all seemed to be reasonably acceptable. I ran a beautifier on the code early on to clean up the formatting and spacing into something that was consistent and looked decent. After I had done that it seemed okay overall and I was wondering how we were going to break this code or found out any gotchas throughout the term.

I remember during the first assignment when we were first diving into the code and modifying it, that's when I started to realize some weird things going on such as the 600+ line switch statement. It was kind of nice being able to clean it up somewhat but the switch statement was still huge since refactoring only five cards pales in comparison to the rest of the cards within the switch statement. We didn't do any testing in the first assignment but after learning about some testing techniques I began to think about how you might test the code in future assignments.

The second assignment introduced some basic testing techniques where we were actually testing the functionality of some of the cards. I was able to find some fairly serious bugs. For example, if you initialize the game with only one player it will immediately crash. I would have expected at least a warning of some sort or that the game could continue with one player. One of my unit tests for the smithy card found that sometimes it would not add exactly three cards to the deck like it was supposed to. Learning how to use gcov in addition to unit testing was interesting since it actually shows you how much code your tests are covering. This gives you one indicator on how effective your tests actually are. For example if they're not covering much code then you know your tests are not thorough enough and you are probably missing a lot of cases. However, code coverage by itself is not necessarily an indicator of a “good” test suite.

Our testing started to get more thorough in the third assignment where we had to have some tests reach 100% code coverage for a specific function. I remember in the previous assignment code coverages were only around 30% or so for the whole dominion file. In this assignment the coverages had improved by about 10-20% since I ran my tests hundreds of thousands of times each with randomized seeds in order to increase the chances of more code being reached. This was definitely a better testing solution than the previous assignment where only one test was ran per card. However, it still wasn’t that great overall given my greatest code coverage was somewhere around 40-50%.

The fourth assignment built upon this idea of randomized testing and extended it to play a whole entire game instead of just a few cards. Similar to the previous assignment, I setup my test to iterate about 100,000 times in order to increase the chances of reaching more logic branches and such. Using this method, I was able to achieve between ~60-70% code coverage using a series of randomized tests that plays through a whole game. I thought this was alright considering that the random tester did not play every single card in the game but was restricted to only a relatively small portion of cards. If we had to have the tester play and test every single card that was playable, I probably would have seen code coverage that was closer to 80-90% or so. While that may sound good, I think it is important to note here that just testing the game and card functionality alone is not sufficient for a well rounded testing suite.

The introduction of mutants into my testing suite was one way of improving my testing suite to become more well rounded and complete. In the previous assignments, we were mainly only testing what was possible/defined in the game code. This is fine for testing functionality but I don’t think it does a whole lot for the stability of the code or it’s ability to handle odd inputs or odd scenarios that are difficult to come up with on your own. Using the recommended code mutation tool (cmutate) I was able to generate about 2500 variations of dominion.c before the tool itself crashed. Mutations allow you to generate many different scenarios each with slightly different conditions and give you a different perspective on what may crash the program or maybe cause it to behave incorrectly. Unfortunately, I wasn’t able to run all my tests on every single mutation because of time issues since each test takes some time to complete and there were thousands of them. Additionally, after only 40 mutations it hit an infinite loop condition where the whole testing suite stopped because it was waiting for the current test to finish. There is more information about this in the mutant.txt file. I believe mutation testing is very useful for finding weird or obscure bugs that you may not have thought of.

In order to improve the testing suite I believe we would also need to implement fuzz testing on inputs for the program where the player(s) actually interact with and play the game. Our tests so far are just generating instances of the dominion game and testing out the game in an automated fashion to find errors. One other good way to test the game would be to test the program that the player uses to play dominion. By using fuzz tests it would be possible to quickly generate many different kinds of player inputs and analyze the behavior of both the “driver” of the game (playdom) and also the dominion code itself.

Ultimately, after iteratively improving my test suite throughout the term, I found that while the game may appear to be okay at face value, there are in fact some rather serious bugs that need to be addressed. Some of these bugs include: when running randomized game tests, playing the feast card always caused the game to encounter an infinite loop; initializing the game with only one player caused it to immediately crash which does not lend to a good user experience. Fully testing every card in the game would also likely yield more issues. Using unit testing I was able to find these bugs relatively easily versus having to find these bugs manually by analyzing the code and running the game to try and get it to crash. My final verdict on the dominion code is that it is good enough in most cases but there are some edge cases that need to be addressed in order to say that it is reasonably stable.

stewadan dominion.c:

I grabbed the dominion.c code for stewadan from the class repository and ran my unit tests on it. I did not notice any significant differences from the outputs of my tests using his dominion.c code. I think there may have been slightly lower test coverages in some of the card tests but the rest seemed to be fine. The code coverage for the random game tester came out to be same as mine at 68%. I think the main difference in stewadan's code and my code was that his trace statements were a lot more verbose than mine. Running a single unit test on his code would result in about 200,000 lines of output. Given the similarities in unit testing, I would say the stability and quality of code is about the same as mine.

washburd dominion.c:

Similar to above, I grabbed washburd's dominion.c code from the class repository and ran my testing suite on it. The results are fairly similar with the large majority of tests passing but some failing (happens to my code too). However, the code coverage that I get with my testing suite seems to be slightly better than stewadan's code. The difference isn't a whole lot though (few percent) and could be just from the random nature of my tests. One noteworthy thing is in my smithy card test I think every test actually failed with washburd's code (~300k lines indicating failure), so there is definitely something up with that. Aside from that, every other test seemed to be about the same as mine. Given the total failure of the smithy card function with my testing suite, I would say the majority of his code is probably good but the smithy card issue needs to be looked at. I looked into it and the smithy card function is drawing two cards at most when it should be drawing three cards.