Jose Rodriguez
06/09/2015

Dominion Test Report

My Experience:

As a Junior in Computer Science, nothing could have prepared me for the beast known as Dominion.c. Never did I imagine I would be debugging code so poorly written, It would be easier to rewrite yourself. I was way underprepared for this feat, but with determination, I was able to conquer Dominion.c. To accomplish this, I first had to familiarize myself with a game I had never even heard of, Dominion. This game quickly became one of my favorite games of all time. After hours after losing, frustration and eventually, my first win, I was ready to dive head first into the world of dominion.c. At first glance, the dominion.c seemed like the hieroglyphs of a dead ancient language, but through sheer determination I was able to decode and familiarize myself with dominion.c. Next, I had to familiarize myself with a git repository hosting service which I had never used. Luckily, I was able to catch on quickly and never made the dreaded mistake of pushing the outdated repository and deleting my classmates hard work.

Now comes the testing. First, were the unit tests. These tests were fairly straight forward and didn't really give much information. The purpose of these tests were really to get us into the mindset of testing and finding bugs. These test were all mostly successful and were able to catch a few bugs but didn't really give much coverage and didn't really tell you much. These unit tests usually had a coverage of about 15%. The card tests were rather reliable though. These tests were able to catch a majority of bugs in the card implementations. Unit tests are more appropriate for smaller functions, to ensure your bug fix works or to further isolate a bug you found using a different testing method.

Then, we began doing the random testing. Here is where we really started to crack down on bugs and make more reliable and accurate tests. Random testing allowed us to run tons of tests in small amounts of time. We were able to get really high coverages using random testing. With the use of random testing we could find those pesky bugs snuck right past our unit tests undetected. We found bugs in adventurer that were not detected by the tests and a found bugs using random test card that my card tests were also unable to uncover. We were able to increase our coverage and reliability with the use of random testing. Overall, random testing is a great way to start testing. It gives you a great range of reliable test in small amounts of time and easily find bugs. Once you find these bugs you can use unit tests to accurately reproduce and isolate bugs.

Finally, we got to the big one, testing dominion as a whole. This was no easy feat. This took a long time of planning and using all of our prior knowledge from the course to implement our most reliable and well covered test yet. This test was able to randomly play and test a full game of dominion. After countless hours of implementation, the grand test was built. Finally, there was nothing stopping us from finally curing the disease known as Dominion.c. WIth this test we were able to crack down on bugs like a police raid at a meth lab. With the best coverage we've had all term, we knew this test was the most reliable we have created yet. Quickly, we were able to determine a huge chunk of the errors that reside in dominion.c. Finally, we have completed the quest we started at the beginning of the term. Full random tests do a great job of really finding the major bugs. Since it is playing a full game like a person would with multiple outcomes and great speed, you'll easily be able to encounter vital bugs that can detrimental to your software.

In conclusion, I have learned a lot about testing this term and have narrowed good testing down to three key points. First, you need to be somewhat familiar with the code you are testing. Even in black-box testing you need to be familiar with the inputs, the outputs and what the function does. Next, you need to ensure your test is reliable. Looking for bugs with a buggy test helps no one. You need to ensure your test is accurately and correctly testing what you need it to. Finally, you need to ensure you are getting decent coverage on what you are testing, otherwise you are not really testing what you are intending to test. Overall, I will take this knowledge with me and use it in my future endeavors. Hopefully, I will never have to take another look at Dominion.c again.

Wentzj Dominion Code:

This implementation of testing dominion.c is well put together. Test dominion has great coverage and seems to be quite reliable. There could be some improvements with the unit and random tests. These tests had coverages ranging from 2.18 to 20.84 % of all lines executed. Overall, these tests seem appropriate. The tests do exactly what is needed of them and are capable of catchhing various bugs. Some of these tests could be more reliable by making more assertions, for example in one for the unit tests (isGameOver()), there was at least one other case which was not tested (when the game is not over). Overall, good coverage and decent reliability.

Camusd Dominion Code:

This implementation of testing dominion.c is also well put together. It was a little hard to follow at first but these tests were well built. The unit tests has great coverage and almost all of them had 100% of lines executed (in the functions they were testing). The card tests however, seemed to have had some errors and were not able to produce a coverage. This is due to a bug that occurs when calling these card functions, not in the test itself. Test dominion, just like the unit tests, also has great coverage and was well put together. The code was very "clean" and easy to follow along. It did exactly what it needed to do and seemed quite reliable. Overall, these implementations had great coverage and are very reliable, with the exception of the card tests.