Elias Rademacher
CS 362
6/9/15

My Software Testing Experience

I have learned many things about software testing and about programming in general over the last ten weeks. Through all the tests I wrote for Dominion, I got much better at tracking down bugs in a program. I got three main takeaways from this: know the code, read the code, and don't trust your tests.

Know the Code

A very valuable lesson I learned through writing tests for dominion is that in order to write tests for a program, you need to know how the program works. This is the first of Agans' principles: understand the system. When I first started writing tests for dominion.c, I felt overwhelmed. I didn't understand the program and I couldn't tell if a certain component was doing what it was supposed to be doing because it was so complex and cryptic. There were also several times that I wrote incorrect tests because I thought dominion.c was trying to get the job done in one way, when in fact it was doing it another way. It has been remarkable how much easier testing dominion.c has gotten as I have come to understand the program. Like I said, it was hard at first trying to write tests when dominion.c didn't make sense to me, but I never found an easy way to get over this stage. I just had to keep writing tests and reading over the dominion code to see where they were failing.

Thus far, I have discussed the importance of understanding the inner-workings of the program you are testing – that is, the implementation. But it is even more important to understand what the program is supposed to be doing. This requires domain knowledge. This kind of knowledge is even more important because with you can write black-box tests even when you do not have access to the source code. Dominion is a fairly complicated game. There are only a few basic rules, but there are a nearly infinite number of scenarios that can happen from game to game. There are many different cards used and they all have different effects. Fortunately, I had played Dominion once or twice before, so I did not have to dive in as a complete beginner, but I still made sure I played the game several times during this term. Knowing the rules of the game by experience made writing tests for Dominion much easier, because I did not have to constantly consult a rulebook to make sure dominion.c was behaving properly. I did, however frequently consult the internet to make sure certain cards were doing the right things.

Read the Code

I have learned about many fascinating ways of finding bugs lately. I wrote all kinds of tests, and they helped me locate the general area of bugs greatly. But in the end, I found that there is no substitute for going to a suspicious part of the target program, and reading it carefully, line by line. There were several times when I would tweak my tests for a long time and pour over their output in frustration, only to end

up finding the bug by doing something that I could have done all along: read the code. For some reason, it takes more energy and focus for me to try to read the questionable code and understand it than to just try to write more and better tests, but there are actually a lot of mistakes that are readily apparent to the careful eye.

Don't Trust Your Tests

One thing that I was surprised to find out over this term is that, more often than not, the problem is with my test, not the code I'm testing. This is probably largely due to the fact that I am still inexperienced when it comes to writing tests. But even when I was trying very hard to be careful, errors came up in my tests. There were "bugs" that existed in dominion.c for weeks before I found out they were bugs in my own tests! I'm still not sure what the best way is to avoid this, but I think the main thing is that if you don't quickly find a bug in the target code, then you should thoroughly review your tests. In other words, don't trust your tests.

Review of Other Dominion Code

Huangma's implementation of dominion.c is the most reliable of the ones that I tested. The results of the tests run on Huangma's code can be seen in ./testResults/ huangma. There were two main tests that I ran on it: a suite of 8 unit tests, and testdominion.c which produces a log of the game being played that I used for differential analysis with my dominion.c. I used gcov to gather coverage information for the unit tests. The unit tests covered 8% of the lines in dominion.c and executed 11.3% of its branches. Based on coverage information that I got from gcov at a later time, testdominion.c probably covered about 65% of the lines in dominion.c.

Huangma's implementation of dominion.c had a bug such that it appeared that the players did not all start the game with 3 victory points. This was actually because of a mistake in the function that returns a player's score, scoreFor(), on line 444 of dominion.c. The loop that looks through a player's deck was iterating up to the size of the player's discard pile instead of the size of their deck. I found this bug by analyzing the output of testdominion.c. It is a bug that I had actually fixed earlier in my own implementation of dominion.c. This bug was easy to find and easy to fix. Other than this fault, huangma's dominion.c worked well. However, given the small size of my tests, it almost certainly contains many more bugs.

Perhaps the least reliable implementation of dominion.c that I encountered was that of luans. I ran the same tests on luans' code that I ran on huangma's code (described above). luans' dominion.c had the same bug as huangma's, except that luans' also caused a segmentation fault. I was not able to hunt down the cause of this seg fault. This seg fault is a very serious bug, because it renders the program completely inoperable. Also, the fact that it is relatively well hidden adds to its severity. Fortunately, I could reproduce it every time a ran the tests, so I could have tracked it down eventually. If it had only cropped up now and then, but still frequently enough to be a nuisance, then it would have been even worse.

Elias Rademacher
CS 362
6/9/15
I found that overall, I like testing. It can be very frustrating at times, but not having tests to show you where your program is buggy is even more frustrating. Testing really is a scientific process, because you have to form a hypothesis and test it until you find it to be true. It requires you to jump out of your comfort zone and learn rapidly, because like I said, you must gain an intimate knowledge of what the code you are testing does before you can even start writing effective tests.