Elsie Phillips
CS 362
6/4/15
Final Project Part 4

My Experience Testing the Dominion Implementation: *Not a Love Story*

This course has really expanded my toolset for testing software. Throughout the course I have had my testing paradigm changed. I feel much better equipped to tackle tracking down bugs in my own code. Before this course I had no experience testing other people's code and trying to fix bugs in it. Going into this course that was a very intimidating prospect. Doing this for the dominion code was far from easy, but I feel like the experiences I had doing it has made me a much better tester. My main takeaways from this course were: understand what the code is supposed to do and to keep a record, and only change one thing at a time.

Understand the Code

One of the biggest challenges that I ran into initially was not having a good understanding of what the code was supposed to be doing. As I mentioned above, I had never really tried working with someone else's code before. It was really difficult trying to write those initial tests because the code I was testing was not based on my logic, and I hadn't done any of the legwork and research it and to write it.

I quickly discovered that you cannot try to test someone else's code blindly. Rather, to write good tests that are even remotely useful you need to really sit down and try to understand what the code is *supposed* to be doing. This isn't alway immediately obvious just looking at the program itself. Some comments are helpful, others are not. There was a very significant research component that went into writing many of my tests, especially the unit tests for the cards. I had to google what the cards were supposed to be doing because many of the comments in the cards did not make their function immediately obvious.

Writing tests also got much easier as I developed a better understanding of the flow of the program. At first I just made assumptions about how things were being done and in which order rather than tracing the route through the program. Once I stopped making those assumptions and started actually reading the code, it got much easier to understand what was happening, and debugging got marginally less intimidating.

Keep a Record

I wish I had started keeping detailed notes of my debugging efforts earlier. I only started doing this while debugging my full game random tester, but it made debugging so much easier. It really helped being able to see relationships on paper between what I was trying and what was happening. I also noticed that it reduced my debugging time, since I wasn't trying the same things over and over. Dr. Groce's rule of thumb with a benchmark of 20 minutes worked well for me. I ran into a nasty bug with an infinite loop. I set a timer for 20 minutes and did what I could do. After the 20 minutes, I began keeping the log. Normally bugs like that take me between 1-2 hours to fix. With the log it took me about 30 additional minutes to identify and fix the bug. I now really value this workflow and will continue to use it well beyond this class.

Only Change One Thing at a Time

This has plagued me many times. I will make a couple of theories about why something is happening and will change all of them. In the moment I'm generally thinking something along the lines of "Well if it's one of these things, it will now be fixed." However that is really faulty logic, because the conditions will not be the same after you have changed multiple things.It is really easy to introduce additional bugs this way. I've become much more disciplined about making incremental changes to my program.

Testing My Code

From my experience with this implementation, a person would only be marginally be better off working from this existing code in a rewrite of the program as opposed to scrapping the entire thing and starting from scratch. My testing revealed that many of the really important functions like discardCard and updateCoins are severely broken and seriously impair proper gameplay. However based on my unit tests and random tests, many of the smaller simpler function work adequately if not elegantly, like kindomCards. I found that looking at coverage for my unit and card tests was not terribly insightful based on the structure of the program. The unit tests were testing either a single function or a single case in a function in a large file. The most coverage i was able to get from my random unit tests was about 5% of the entire dominion.c code. I feel like coverage would be much more insightful if it was being measured over a much smaller amount of code. I felt it was useful in testing my full game tester for instance, where I was able to get 60% coverage of dominion.c.

Testing Other's Code

Of the two classmate's code that I tested, neither seemed more reliable than the other. After running diff on their dominion.c files I noticed that neither had made significant changes beyond refactoring their 5 cards and adjusting the formatting. Aileen's randomtester was able to achieve 56% code coverage, while Lee's was only able to achieve 36%. To me this means that Aileen's code is better tested, but is not necessarily more reliable.