Charles Henninger
CS 362
Final

Debugging

Testing dominion was less a trial of my (fairly minor) skill in debugging and proper testing, and more trial of patience. To be frank, the dominion code was bulky, ugly, and only "worked" in a very relaxed definition of the word. However, this was the point; to be given a software testing playground fraught with bugs and errors and to have us try out different ways of finding, categorizing, visualizing, and fixing bugs. Well, maybe not fixing them, but certainly finding them. Despite having to work with some of the grossest code I've seen in my, admittedly short, coding career, I think that I learned valuable skills for testing and debugging code. I also learned many was Not to debug and test code, which honestly is almost as valuable as knowing the right way.

The first phase of this term's process of testing and debugging the dominion code was writing unit testing for refactored card functions and game functions such as compare and whoseTurn functions. The cards that I refactored were council room, sea hag, and adventurer. One of agan's principles quickly became apparent in this initial stage; I first started trying to refactor and test the cards without knowing how to play dominion at all, which was a frustrating error on my part. Without knowing how the game is played it was near impossible to write substantial unit tests about how the cards were working. Writing a unit test to simply see whether the card function ended without an error didn't really show if the card had done what it was meant to do, and without knowing what the card was meant to do in the first place, I had no knowledge about what the assert calls should be measuring. This, of course, shows the importance of agan's principle concerning "reading the manual", or in simpler terms, "get some background knowledge on what you're working with before you run around trying to do anything, you hasty idiot".

After unit testing came a more in depth testing strategy: random testing. This was where the true crappiness of the dominion code became apparent. The random testing seemed to be different every time I ran the program, and like the naive fool that I am, I went looking for the reason why. Suffice it to say that I did not find out why, and during the process of me meddling in the code to get some answers, I ended up changing a lot of things all at once. After my quest for answers was finished, I couldn't hope to recall all the changes that I had made. This caused a myriad of issues for me down the road, and really drove home another of agan's principles: change one thing at a time. Despite this, I learned a great deal about the concept of code coverage. Code coverage is not a foolproof way to gauge the correctness of code, however it does play a role similar to code smells; if your code coverage is low, it's pretty likely that there are bugs that you are missing, or that the faults that you are experiencing are occurring in pieces of code that you aren't testing. All in all, random testing was an educational, and humbling, lesson on how to know whether your test suite is actually accomplishing what it is

suppose to. I did learn however that my test suit was not nearly up to par with what this monster of a code needs; despite having reasonably high code coverage (~60-70%), there were still a huge number of bugs occurring.

While using differential testing, I became aware that while some are doing as I have done concerning the dominion code (fixing blatant bugs when I come across them, but not really going out of my way to fix the dominion game as a whole), there were some who when above and beyond on their dominion code; fixing almost all of the bugs that one could find in a decently long session of debugging and testing. During the most recent phase of debugging we used differential testing to compare two versions of dominion in order to see where they varied and where they were the same. I had two friend's code, one similar to mine and the other version one that had been scrubbed nearly free of bugs. With this latter version, the one that had gotten some much needed TLC, it was extremely easy to find where bugs were occurring that were causing the faulty outputs in other versions. While this method of debugging may not be extremely useful unless you have a version of your code that is pretty much correct, it is very effective when it is possible to use it.

In conclusion, almost all of the methods used in this class's curriculum were interesting and educational, although some may not have been useful in every situation. Unit testing and random testing are great ways to check the functionality of your code and to ensure that your testing suite is doing it's job. When differential testing works like a charm when it's feasible to use it, it's one of the strongest methods to see how and why a code version is working improperly. However, I can't really think of a time when you would have a correct version of a code and not be able to use it beyond running it against another version of the code. That being said, out of all the methods that we learned about over the course of this class, I think the coolest and most useful is the tarantula approach. The way it singles out high risk sections of your code and helps you visualize them is extremely useful when debugging. However, to say any one method is the best or most useful is inaccurate. All of the methods we used have positives and negatives, and each has a situation that they should be used in. Despite having to work with such a nasty piece of code, I'm impressed with how much I learned over this class while debugging it.