Nathan Burnett

June/4/2016

CS 162

My Experience with Dominion

At first I was surprised by the idea of spending an entire class working with a large amount of very poorly written code. Once we started into the assignments in the course, I quickly got intimidated. The code had an overwhelming effect for a few reasons. For one, it has been the largest piece of code that I have been given to work on (that I had no part in developing). There were multiple files containing a large sum of code, and understanding the code at first was very difficult. Each function seemed to be chained with four other functions, and I had to scroll all through the code just to understand one simple statement. Second, the code had more problems than I thought I'd be able to come close to understanding or fixing. The idea of finding bugs in an immense collection of poorly written code that -at the time- I hardly understood seemed impossible, and was frightening at first. Lastly, there appeared to be many layers of poorly written code. Meaning that if an operation was not behaving correctly, I could not be sure if the first called function was causing the fault, or the second called function, and so on. These were the concerns I faced at the beginning of the course, but each of these concerns turned out to be later addressed, and allowed me to learn from the experience.

Of the intimidating factors mentioned, they seemed to be centralized around the ideas of poorly written code and lots of it. Luckily, the structure of the class helped to address both of these ideas, and soon showed that the process would be achievable. The factor of poorly written code went along with the many ideas of the course revolving around fault localization. At first we wrote what I thought were very simple unit tests. However, like we discussed in our lectures, I found that the unit tests were too broad, and I would have to split them up. Upon beginning to split up the unit tests and finding exactly which parts of the unit tests were failing, fault localization became much easier and I was able to locate exactly where faults were occurring. The next intimidating factor that was addressed was the idea of seeing a lot of code for the first time. This factor went along nicely with the idea of unit tests, but before writing unit tests I had to understand how the game was played. While I did try to read about the structure of the game, I

found this boring and inefficient. Instead I downloaded the Dominion application, and played about 100 games of Dominion online. After Playing the game enough I had a thorough understanding of gameplay, and understood basic strategy. Once I had an understanding, I was able to write unit tests. The unit tests helped to split a large complex code into many smaller pieces of code, and showed what features of the code worked. The structure of our class also helped by developing from the most basic to the most complex testing.

The first tests implemented in the course were unit tests. Unit tests were both something that everyone could easily conceptualize, and something that most people had seen before. By then moving into random testing, we saw how our unit tests could be expanded to catch edge cases and other rare errors. Upon moving into full game random testing, we saw that we could look for more in testing than just features, and we could check for the quality of code in general. After learning a few of the basic testing principles through our own dominion implementations, we were ready to judge the quality of the dominion codes of other students.

The first student who's code I looked into has the username almutnaw. When first looking through this student's code, I did not notice any major differences from the original dominion implementation. The first step when looking into the student's code was to run the unit tests. From the unit tests I found that the student had a few basic cards working, but they had not corrected a few simple yet critical bugs in the code. The student had not corrected the bug to allow a user to play with multiple players, and they had not corrected the bug to score the game correctly. Without the ability to score the game correctly, a game cannot accurately determine a winner, giving the bug critical importance. The reason I noticed this bug in particular, was because this bug was simply a copy and paste syntax error, and likely should have been fixed by most implementations. The unit tests ran covered 32.6% of the student's code, which was close to the expected number. After running the few unit and card tests, I used my full game random tester on the student's dominion code. I was unable to run a full game test without the code falling into an infinite loop. I believe that somewhere in the students' playCard logic they left potential for an infinite loop that multiple cards could cause. Without begin able to complete a full random test, I was unable to determine an accurate coverage.

The next student who's code I looked at has the username merdlera. When first looking through this student's code, I noticed that the code looked very different. After further

examination, I noticed that the student had moved functions around differently, but the functionality was very similar to the other implementations I had seen. I again started by running my unit tests on the student's code. Once again, I found that the student had not fixed both the multiple users and the scoring bugs. The unit tests again created 32.6% coverage on merdlera's implementation, which was not a big surprise given that we likely did not significantly change the few functions the unit tests covered. When moving on and running my full game random tester, this students code was not reliable. The code worked fine about 80% of the time (and showed 46% coverage), but another 20% of the time the code segmentation faulted.

From testing the implementations of two other students, I learned how important the few changes I made to my own implementation may have been. When running my random tester, my code gave proper results every time. However, I got inconsistent and incorrect results when using the implementations of other students. I only fixed a few bugs throughout my time using dominion, but in ways I cannot be sure of, they ended up making my code work in situations where other implementations did not. Overall I found that most students did not change their code significantly, but the small changes made may have been more significant than I originally thought.