Hai Nguyen

Alex Groce

CS 362 – Software Engineering

9 June 2015

Final Report

From my experience with dominion and its code. I have gained the knowledge of a wide range of testing material available to use now that can apply to all of the programs that I create. The projects and assignments overall have given me a good aspect in looking at code that was not created by me, not knowing its structure, having to study rules of the game and its implementation, and modifying my own part to implement fixes and tests that stress the program to make its functionality and quality improved. The different ranges of tests that I now know of can be applied in many different situations depending on its size such as a small exhaustive test, or a large random tester. With a combination of the different tests, they can be applied to any software in the real world for debugging and even discovering some important bugs. Dr. Groce's experience with NASA and stress testing for Curiosity specifically interested my hobby for space technology and involvement. It brings in a perspective of the massive amount of work and time that it takes to just create a tester that never really accounts for every scenario given the timespan of the project. When applied to the dominion code, even a simple thousand line program was able to fulfill a coursework worth of workload for testing.

The first assignment was the introduction assignment, which was very simple and just got the students accustomed to the code and was able to implement their own functions. In general, it was a good way to get us reporting our implementations and getting used to github and svn for those who did not have the experience with it. Working with the second assignment was our first testing assignment in which I implemented tests shown as an example from class. These tests were initially hard to setup, but once coded, the design seemed to work well with the dominion code. The unit tests initially were more difficult to figure out than the card tests. The unit tests involved calling more functions within those units than cards did, in which case required me to create more tests for those other functions. It was also difficult not working within the onid server and trying to gcov from my executable. There was a problem with CodeBlocks and programming dominion that created a massive amount of unnecessary files and excessive space, which made it difficult sometimes when pulling and committing from git because adding files was constantly needed. Creating a random tester in assignment three turned out to be more difficult than expected as well. The thing that helped the most was the slides for random testing that contained the example code for creating random generators. The adventurer card was difficult at first discovering its bug because it just segmentation faulted every time after running the tester. It later was not found until after discussing the issue with other classmates and using my debugging tool on my IDE. As the amount of files began to increase, so did the amount of files listed in the Makefile. In general, its format was the same, except that for my PC, I had to make all every time to test one file and execute it from my command line. It was definitely not how the project was designed from my perspective but that is how I adapted to it. Finally for assignment four, creating testdominion.c was not particularly difficult, but it took an immense amount of time creating the file itself and creating its own game. The hard part was creating gameResults and discovering differences with other student's codes. Initially I was not finding any differences until I had to add more tests in my testdominion.

Coverage for my code was decent at around 24% coverage for testUseAdventurer and 26% for testUseFeast. Even with those tests for the cards I was able to find several bugs including adventurer not discarding after its use or different seeds going into random infinite loops. Since most of the time, we were not required to fix those bugs, I left them alone for the most part. The only time that I fixed bugs were the when they did not allow my tester to finish all the way through. Coverage for my testdominion.c accumulated to 69.42%. These were averaged based on the different times I ran test dominion with different seeds and slight changes to the functions.

Based on the two classmates I ran my tests for, I would not be confident with either of their reliabilities for the implementation of the dominion code. From initial examination, both of the codes were very similar in lines and the amount of extra code added from the original and it seemed like most of the code for their dominion.c remained untouched.

The first classmate's code I reviewed, I was able to find three bugs very fast with my unit tests. Initially I did not account for the unit tests using the cardEffect function, so initially I began with a build error, but after the fix, the unit test immediately revealed an error in the expected value or useFeast. This particular bug returned one card higher in the hand than expected, which seemed to be an existing error in the code. Other bugs for the classmate included an infinite loop and a segmentation fault with my testdominion.c. Coverage for my classmate's code was 82.59% after 50 runs of full game tests.

The second classmate's code I reviewed was improved, but still unreliable. Just like the previous code, most of it was untouched, only edited parts that were part of the assignments and part of my build itself did not build until editing parts of his dominion code. Once the code was run, I gained an 85.21% coverage rating after 50 runs of full game tests. Bugs in this code included Steward discarding the incorrect card, and Feast going into an infinite loop and lowered the player's coin count.