Christopher Martin
CS 362
Dr. Groce
June 9 2015
WC:

Dominion Testing Experience Report

The first assignment did not cover testing. All we did was refactor code, which means to change the code is some way. We moved five cards from cardEffect into their own functions. Although, no testing or debugging was preformed, bugs were two bugs introduced. These bugs were later used as the bugs to be found in the project.

Assignment two is where actual testing happened. In this assignment unit tests were written for four cards and four functions. Unit tests are small bits of code that setup states and conditions for a function, then verify the output or result of the function is correct. Most of the Dominion functions I tested were trivial. I wrote tests for compare, isGameOver, updateCoins, and getCost. I did not find any major bugs in these functions other than not testing for null pointers. Functions like isGameOver and updateCoins would operate on invalid game state and return the correct result. That is not really a bug because the behavior is correct, but I would have programed the functions more defensive such that they would check valid game state first. For example in updateCoins, player 7 can have coins calculated even though the game only works for four players. With all the unit tests for Dominion functions I get about 5.03% of 597 line in Dominon.c. The four cards that had unit tests were Smithy, Village, Steward, and Outpost. Again, I did not find any bugs other than not handling null point for game state. The code coverage for the cards over all of Dominion was 19.43%. After completely assignment two was able to cover about 20% of the dominion code just by writing eight unit tests. That seems really good just buy a few unit tests. I would feel better though if I found more bugs. It just doesn't feel right to write unit tests and get no bugs.

Assignment three was special because it was not normal unit testing, it was random testing! I have done unit tests before for other classes but I have never done random testing. I was a bit confused on how to do it at first but I got the idea that you want to make states for each card random to catch more bugs that you could think up buy doing unit test asserts on specific conditions. Random testing is better because it can generate all sorts of different values. The random test for adventure covers 21.94% of dominion.c. This should not be too supprising because of all the helper functions that are called by setting up the game and playing the card. The random test runs 300, covers all the code for adventure, but found now bugs. May be I need more state variables to be random. The other random test was for village and I was able to find a bugs with it. The only way to get Village to seg fault, is by giving a player or hand position out of bounds, or by giving a null game state. I added code for out of bounds player and hand position values. Code coverage of this card was 19.26%. Even tough random testing did not find many bugs for me it did cover a good about of the code base.

Assignment four was like assignment three in that it required random testing. But, this time we also included differential testing. Differential testing means running the same test on different code bases to see if the results are the same. Code coverage varies because each time the program is run a different cards are played so different functions will be called. Running the random tester on my dominion gave code coverage from 52.76% to 60.64%. This is the largest code coverage that I was able to get from just running one test. So for code coverage a random tester seems to do a better job than just simple unit tests. I ran my test dominion code with my version of dominion.c and knoppjo version. There are about five cases where the two version have no differences when run with the same seed. I know I modified my dominion.c file in several places. I did not check to see if knoppjo changed his dominion.c file. I figure that because I made the changes there will be differences, but I am not sure as to why there are so many non similar games results. The coverage from running the random tester on knoppjo code came to around 62%. It is interesting that there is a slightly more coverage with his version of dominion. Maybe the code is smaller.

The project in part one and two used unit tests for find bugs and writing reports on them. I did this type of process at community college in a software management class, but there we used a bug ticketing system, called Traq. That was really cool because you could list the bugs you found, set the severity and priority. Then other people would claim bugs and fix them. After the fix, they would run your unit tests to verify that the code was correct. Why we do not do the same at this school is beyond me. The students would learn a whole lot more if we did that. I mean, its is how the practical real world works right? Anyways, for part one and two I just talked with knoppjo to find out what bugs he made from assignment one, then I wrote unit tests to verify that the bugs existed; see targets runTestDominion, bug2, bug3, and part3_baron in my Makefile. Part three of the project I bug3 from part two, where Estate count would go below zero. I wrote a unit test for it called part3_baron. It is basically the same makefile rule as bug3. For part four I ran my random tester against knoppjo and kundec dominion.c file. I was able to get both to run with out any problems. Both are reliable based on my random tester. Coverage for each was knoppjo 57.20% and 61.28% for kunde. The results are near what I got for running the random tester on my code.

Over all fun easy class. I learned a little but not a whole lot. Have a great summer and play more games!