Andrew Gass
CS 362
Test Report

Testing Dominion

Testing dominion.c has been an interesting experience. It feels like it was written either by different people, or at a variety of different times and states of mind. There are partial implementations of concepts, or ways of handling different aspects of the game, that are just dropped at different places. It has extensive errors, both in the logic that is implemented and in the fact that there's logic that isn't. For instance, the code simply…doesn't leave players with hands. The struct for the game has space for hands for each player, but the way the code is implemented, players aren't given hands at the end of their turn after discarding. Also, when they play cards, they go into a "playedCards" array in the struct where they just stay. They should get put into discard by the end of the turn, perhaps right before the playedCardCount is set to 0, but it's not. The array entries will get overwritten, since the count is used to index, but it's basically a black hole for cards. The entire code is kind of like this; buyCard even has a comment "I don't know what to do about the phase thing." The commenting, debug options, and variable declarations vary, and are usually not satisfactory. I suppose this makes it good code to test, since the tests won't just pass every time, like they might with actual functional code. For assignment 4, however, this made it a challenge to figure out if the tester as a whole was working. I wouldn't personally have written such a high level test without having first tested and ironed out the individual functions; it's like building a house without measuring the lumber as you cut it. Maybe working in small groups, dividing tests of all the functions between members, and having each responsible for fixing the individual code they're testing, would be a way to have a set of working functions upon which to test the overall game performance. Of course, you'd then need well defined input and output conditions for the functions, which right now is also lacking.

I think this program over all would be better written in a more object oriented program, especially something with global variable. Though this code can theoretically work in C, it would probably be much more straightforward to maintain and read in almost any other modern language. The complexity isn't very high, so the speed C could theoretically offer over a more OO language isn't a huge consideration.

kaiserh: Harrison Kaiser

File 'dominion.c'
Lines executed:81.85% of 562
Creating 'dominion.c.gcov'

Harrison has what I find to be a surprisingly good coverage of dominion, given that some of the functions simply can't be called without getting locked in a loop. The code in his unit tests and random tests is generally easy to read. His dominion tester is split into reasonably sized functions, with well labeled declarations; there are even prototypes of the functions at the top of the file. Commenting is at reasonable levels. He made extensive formatting changes to dominion.c, and quite a few code improvements. The increased regularity of notation is a welcome improvement.

His code and makefile executed as would be expected from the assignments, and output is both extensive and well labeled. If you only had the output file, you could still piece together what had been tested.
His testdominion makes the “choice” selection randomly from the hand, which doesn’t always make sense (some are meant to binary choices), but shouldn’t be an issue for a random tester. With the dominion implementation’s lack of range handling, though, it is quite likely to cause errors.

zakrevsm: Matt Zakrevsky

File 'dominion.c'
Lines executed:56.38% of 564
Creating 'dominion.c.gcov'

Matt’s coverage is closer to what I’d expect given the random nature of the testing and the fact that not all functions are testable. His unit tests are a bit harder to read, but do have headers with the function they test and his name, which is helpful. At least his supply count test, however, is inadequate, as it calls the buyCard function, which relies on the purchase proposed being a valid purchase, but no extra coins are stocked. The 37th line also buys gold instead of silver. There’s no way to tell from the output of this test if supplyCount is returning a valid value. Formatting in the unit tests is also somewhat irregular, including column alignment and capitalization in print statements. In his buycard test, he has two unused variables, one of which is set twice. One of his unit tests returns EXIT_SUCCESS, while the other return 0. His adventurer random test only seems to check for crashing, but doesn’t record circumstances besides the test number. His random test card tests the minion card, but doesn’t have title header his other tests have; similar to the adventurer random test, it doesn’t record the circumstances of any of the tests, just the test number, though parts seem to have been written with a print command in mind. Some variables are stored each loop, but not ever read or operated on.
His testdominion doesn’t establish a random set of cards for the games, though he does have code commented out that might do that. Also commented out are an assortment of what I believe are debugging print statements, which would probably be better if behind some method of debugging check, instead of commented out. Additionally, there are bits of old code that are simply commented out. Some of the print statements could use some fixing up.
As far as logic goes, it seems like the use of random commands to modulo 3 means the lower cards of the hand are more often selected for, instead of anything higher than index 2 in the hand. Also, there doesn’t seem to be any logic against the number of buys allowed per turn. The code also uses the same print function as it does for every turn to print the final end state, but most of what it prints isn’t helpful, and there’s no indication of who won or what scores were.
The outputs of his tests are hard to follow, requiring in most cases reading the code to know what any of it means.