Introduction

Testing the implementation of Dominion provided in this class was fairly challenging, primarily due to the lack of good documentation in the code. The header files (dominion.h, dominion_helpers.h) were helpful, but somewhat misleading (such as buyCard line 444, “I don't know what to do about the phase thing.”).

Unit Testing

The majority of my testing experience is in unit testing, so I was excited about getting the opportunity to unit test dominion. However, without the convenience of testing libraries and frameworks such as JUnit or Python’s unittest, as well as mocking libraries that allow you to stub functions and control the external forces when unit testing, this was a very difficult task. I looked into some unit testing frameworks for the C programming language, such as Check, CUnit, and GNU Autounit, but after weighing the learning curve and setup time against just raw C testing, I decided that it would not be worth it for this class alone. If in the future I needed to debug C code, I would probably invest time in learning Check (it had the most recommendations on Stack Overflow). After writing raw unit tests in C, I now have a greater appreciation for the testing frameworks that I use every day for Python and Javascript; they make writing quick unit tests for regression much faster.

Writing unit tests for dominion primarily involves setting up the game state, invoking a function with some parameters, and asserting that the output is what it should be and that no exceptions occurred. It was somewhat difficult to write black box unit tests for the code without inserting print statements or exception handling using try/catch blocks because the behavior of the code is not always clear. The most significant example of this was when I was unit testing the implementation written by *liujiaw*, and I needed to set several print statements in order to see what was happening inside of the my_Smithy() function. I was able to achieve 90% test coverage on this function because I could find exactly the parameters that were needed to invoke almost every line of the function.

Bugs:

- When trying to test getCost, I encountered a segfault. This could have been because of bad testing parameters, however.

Coverage:

- I was able to reach about 26% test coverage in my tests.
- This is a very alarming figure considering only 8 functions were directly tested. It tells us that the code is very tightly coupled, because over a quarter of the code is being run by calling these 8 functions.

Impressions:

- Unit testing this program is very difficult. This is because the code is not well documented, which makes it difficult to know exactly what parameters a certain function should take, what assumptions are made about the arguments, and what

the function should return.
- Testing this code is interesting. It is like spell checking a thesis written in another language. Sometimes you have to assert basic principles of a function just to learn what you should actually be testing.

Random Testing

Before this class, I had no experience with random testing. All of the unit and integration tests I wrote only tested parameters specified by me in the test code, and there was no randomization of inputs or state variables. Writing random tests allowed me to gain a higher level of test code coverage without writing a huge amount of single unit tests to cover all edge cases. Of course, the significant edge cases should still be hardcoded into your test suite, but random testing can hit a greater number of use cases. My implementation of random testing started as just a random generator for dominion unit tests with randomized state settings and inputs, but eventually evolved into a more cohesive testing experience with the whole game tester, which will be discussed later. Below are some of the cards I tested using random testing:

council_room:

I was able to achieve 100% coverage for this function. It only requires a quick 10000 iterations in order to get it to reach this coverage level. My unit tests were not able to reach this level of coverage, simply because they were not hitting the right parameters. However, I feel that unit tests are able to catch the edge cases that are obvious to the developer as they write the code, and are suitable for testing those cases. Random testing is better suited for brute force testing the functions, which is still important.

adventurer:

I had a lot of trouble testing this function due to segfaults. I was able to use gdb to run a backtrace on the function,

and found that the segfault was occurring when it "discards all cards in play that have been drawn". The statement "state->discard[currentPlayer][state->discardCount[currentPlayer]++]" was trying to access an array value way out of bounds. I suspect this was due to the random testing extreme bounds, and I was unable to fix the test to get it to pass.

smithy:

This function had not been refactored, and was still in the large switch statement in cardEffect. However, I was able to isolate the state changes that it makes to the gameState and effectively test them.

Whole Game Testing

When comparing two implementations of Dominion using my diffdominion Bash script, I was able to consistently achieve a different output between the two programs. I believe this is because of the nature of random testing used in the testdominion.c program; the game state is randomized, therefore the both programs will run differently. I was also getting different test coverage for each implementation test, therefore the gcov output was different. It is hard to say

which implementation is more correct, because there are too many variables in testing. This is not an ideal case for differential testing because we do not have an implementation that we know is 100% correct to test against and find differences. The highest code coverage that I was able to achieve on my own implementation was 78%, while in the same run the other Dominion program reached 67%. This is not enough to assure correctness of either program, and the random test generator needs to be improved to hit specific functions, such as adventurer card, which had very low percent coverage.