**Random Testing**

In developing these random test suites, I used the unit tests developed in the previous assignment as foundation. The tests for correctness served as the basis of the oracle for the random testers. For example, for the Adventurer card random tester, I split the tests into the pre-condition and the post-condition. The pre-condition tested hand count, discard count, and deck count. The post-condition tested that the hand count increased by 2. If it did, then make sure that the discard pile increased by 2 and the deck count decreased by 2.

On top of this, I randomized every variable within the gamestate. I also set up a flag that would allow the tester to test that these other variables did not change between the pre-condition and post-condition. Since testing these other values increase the test count by a few hundred tests per test, I made this option disabled by default.

Finally, once I made sure this randomized test worked for one instance, I set a for loop and set a TEST_COUNT constant variable to set the number of test counts per run.

**Code Coverage**

Each of the random testers achieved 100% statement and branch coverage for the functions they tested. However, they did not achieve 100% statement and branch coverage for the whole dominion.c code. They each achieved about 30% coverage.

Below is the times required to run each program. Time was recorded using the C clock() function. Thus, this is the time elapsed on the CPU, it is not a reflection of time elapsed from the prespective of the user.

| | |
|---|---|
| randomtestadventurer | .703 seconds |
| randomtestcard1 | 1.903 seconds |
| randomtestcard2 | 2.2 |

**Unit vs Random**

Both the unit tests and the random tests achieved 100% statement and branch coverage. However, the random tests were much better at detecting faults. In order to capture some of the more subtle bugs I implemented, I had to revise my unit tests several times. At times, it required a certain degree of creativity and meticulousness to capture the bugs. For the random test, however, it was trivial to capture bugs.

On the other hand, the random test is a bit like “drinking from a firehouse”. While it was successful in capturing faults within the software, it then became a challenge to pinpoint exactly where the issue was and what conditions led to it. In this instance, it is clear to see how domain partitioning would be a useful heuristic. Moreover, this methodology could be improved by applying Zeller’s delta-debugging tools to narrow in on aspects of the domain and specify how the faults erupt.