Zachary DeVita

CS 362

March 12, 2016

**Test Report**

I found working with Dominion to be rather difficult, more so than I would have expected. Many people in the class came with prior exposure the Dominion; they knew of the game, had played it, etc. I had never even heard of the game, let alone, played the game or understood the rules or gameplay. Even with this disadvantage I still assumed that it couldn’t be very difficult to write a few functions and test the game. Reluctantly, I had to learn the game. The tough part about learning the game was that I couldn’t just learn from the source code. The source code had an inordinate amount of bugs and incorrect commenting, so it was impossible to differentiate the correct code from the bugs. This meant I had to physically refer to a copy of the game instructions when writing tests.

**Unit Testing**

The initial form of testing I conducted was a series of unit tests. This form of testing provides the same effect as a simple print-line test, but it is contained a separate .c program. With these tests, I was able to isolate small testable statements in the source code and verify that they function properly.

I conducted unit testing for four cards and four non-card functions in the dominion.c code. On this specific testing assignment I decided to test the entire functionality of the cards and functions by having a unit test for every condition. The results of my unit tests are as follows:

File 'dominion.c'

Lines executed:32.59% of 580

Branches executed:37.17% of 417

Taken at least once:21.58% of 417

Calls executed:30.00% of 100

I found that each of my unit tests covered between 16% and 30% of the dominion.c code. There is not much I can infer based upon the results of the individual coverage tests. A total of 32.59% of dominion.c lines are covered by all the unit tests. This tells me that my unit testing covers about a third of the program which is pretty good for only testing 6 functions and 2 switch statements. Obviously, all lines would theoretically need to be executed for the testing to be most effective. One other problem is that merely looking at coverage does not tell you if the tests were effectively written or if the conditions were even met which would allow the tests to fail.

The unit tests revealed bugs in two of the tests done. Based on the nature of the failures and some static analysis of the source code I was able to find the bugs which I suspect are responsible. Upon testing the function scoreFor(), I was able to pinpoint an inconsistency in the results being returned. I was using fixed cards containing known values but commonly receiving incorrect results from the function. I found a bug in the function scoreFor() at line 441, in the third for loop. When the score is being evaluated from the deck, the ‘for’ loop needs to iterate from i to deckCount. Currently it iterates from i to discardCount.

//score from deck

for (i = 0; i < state->discardCount[player]; i++)

The second bug I found was in the ‘mine’ card. The 'mine' card is used to trash one treasure card from the player's hand in exchange for a treasure card of higher value. For instance, a copper could be trashed and the player could pick up a ‘silver’, or a ‘silver’ could be trashed and the player could pick up a gold. The problem with 'mine' was that it was that the unit test was failing because the same card that was supposed to be trashed was being returned. I found the error to be in line 710 of my implementation, in the conditional of the third 'if' statement. The ‘>’ operator in the following statement is supposed to be the ‘<’ operator.

if ( (getCost(state->hand[currentPlayer][choice1]) + 3) > getCost(choice2) )

return -1;

**Random Testing**

Random testing was initially a much more complicated form of testing for me, and it required a stronger knowledge of the Dominion gameplay. My comprehension of random testing is still questionable. I think this form of testing is far more useful when searching for a specific bug based on the nature of a problem. I also think it would be much more useful in source code where there are few detectable bugs and running thousands of tests is necessary to find failing cases; by using code which is littered in bugs to the extent of our implementations, it is extraordinarily difficult to be have any confidence in which statement is creating the fault. This led me to second guess myself on a couple of potential bugs I “may” have found. An experiment needs variables and a control, but our experiment had variables and an intermittent control.

I ran random testing on the card ‘salvager’ and the card ‘adventurer’. The coverages are as follows:

|  |
| --- |
| Running Random Card Test for Salvager  Lines executed:32.24% of 580 |
|  |
| Running Random Card Test for Adventurer |
| Lines executed:21.72% of 580 |

For testing Salvager I began by creating a loop running my randomized tests 10,000 times. Then I randomized variables like the number of players, deckCount, discardCount, and handCount. I then looked at the functionality of the card in both the dominion.c code and Google.com, and I figured out that the variables which the card directly affected were the player’s ‘coins’ and ‘numBuys’. I then compared the initial values for these variables with their values after the card had been played. I found zero bugs with ‘numBuys’ of the value for ‘coins’ after the card had been played. I did however have issues with player numbers.

The game requires 2 to 4 players. If, for instance, 4 players are hard-coded into the program, and I hard-code the player who is used for conducting the tests to player 1, the test runs with no bug. If the player is hard-coded to 1, 2, or 3, then no bugs occur. There are supposed to be 4 players possible though, and if player 0 or play 4 is hard-coded, then there are bugs 100% of the random test iterations.

For testing Adventurer I did much the same as I did with Salvager. I began by creating a loop running my randomized tests 10,000 times. Then I randomized variables like the number of players, deckCount, discardCount, and handCount. I then looked at the functionality of the card and figured out the variables which the card directly affected. I compared initial values for these variables with their values after the card had been played, and again found no bugs with the handCount or the value for coins. I again ran into the identical problem as before with the player numbers. This bug is not rooted in either functions being tested, and would require far more testing to determine its origin.

**Classmates’ Dominion Implementations**

I tested and analyzed the Dominion implementations of Jason Klindtworth and Minsung Choi. I found the differences in coverage to be negligible, within fractions of a percent differences from each other.

**Unit Test Coverages**

Zachary DeVita

Lines executed:32.59% of 580

Jason Klindtworth

Lines executed:32.71% of 576

Minsung Choi

Lines executed:32.22% of 580

**Random Test Coverages**

Zachary DeVita

|  |
| --- |
| Running Random Card Test for Salvager  Lines executed:32.24% of 580 |
|  |
| Running Random Card Test for Adventurer |
| Lines executed:21.72% of 580 |

Jason Klindtworth

|  |
| --- |
| Running Random Card Test for Salvager  Lines executed:32.38% of 576 |
|  |
| Running Random Card Test for Adventurer |
| Lines executed:21.81% of 576 |

Minsung Choi

|  |
| --- |
| Running Random Card Test for Salvager  Lines executed:32.21% of 580 |
|  |
| Running Random Card Test for Adventurer |
| Lines executed:21.66% of 580 |

**Testing Results**

All the same bugs found in my implementation were present in both classmates’ implementations as well as a couple additional bugs found in Jason’s implementation.

On line 682 and located in the ‘mine’ card, in the second ‘if’ statement of Jason’s source code, I found a bug in the conditional. The conditional requires the card selected as ‘choice2’ to be two different values at the same time in order to return true which is impossible to achieve.

(choice2 > treasure\_map && choice2 < curse)

In order to fix this bug it is recommended to replace the ‘&&’ operator with the ‘||’ operator.

On line 744 and located in the ‘village’ card, I found a bug in the incrementing operation of ‘numActions’. ‘numActions’ is incorrectly incremented by +3 when it should be incremented by +2 when the village card is played.

state->numActions = state->numActions + 3

In order to fix this bug it is recommended to replace the ‘+ 3’ with ‘+ 2’.

I am confident in saying that all three of our implementations of Dominion are absolutely unreliable. It appears that no improvements on any of the source code has been done, or at least detected by my testing. If bugs that were found during the assignment testing processes were actually fixed, then I believe the results from my analysis would be notable.