DREW WOLFE | ONID: WOLFEDR
CS 362-400 WINTER 2018
ASSIGNMENT: 3

BUGS

It's probably helpful to outline the effects of the Assignment 2 refactor considering there are intentional bugs written into the code and thus effect the outcome of the card and unit tests.

CARD	BUG / EFFECT
great_hall_card	CLEAN function. No bugs
council_room_card	BUG: Should provide another BUY turn, but does not
village_card	BUG: Provides four (4) ACTION turns rather than two (2)
adventurer_card	BUG: Increments the drawn treasure by four (4) rather than one (1)
smithy_card	BUG: Draws four (4) cards but should only draw three (3)

The following outlines the bugs discovered during a single instance of each test being run.

TEST / FUNCTION	FAILURE
UNIT TEST 1: SHUFFLE()	Deck shuffled by shuffle() function FAILED
UNIT TEST 3: UPDATECOINS()	COINT COUNT – COPPER, Expected output: 8, Actual output: 3, FAILED
UNIT TEST 3: UPDATECOINS()	COINT COUNT – SILVER, Expected output: 13, Actual output: 3, FAILED
UNIT TEST 3: UPDATECOINS()	COINT COUNT – GOLD, Expected output: 18, Actual output: 3, FAILED
CARD TEST 1: SMITHY	FAIL: Incorrect HAND COUNT after playing SMITHY. Expected output 7, actual output 8
CARD TEST 1: SMITHY	FAIL: Incorrect DECK COUNT after playing SMITHY. Expected output 2, actual output 1
CARD TEST 1: SMITHY	FAIL: Incorrect SCORE after playing SMITHY. Expected output 1, actual output 2
CARD TEST 2: ADVENTURER	FAIL: Incorrect DECK COUNT after playing ADVENTURER. Expected output 3, actual output 4
CARD TEST 3: COUNCIL ROOM	PLAYER-1 number of buys increments FAILED
CARD TEST 4: VILLAGE	FAIL: Incorrect ACTIONS count after playing VILLAGE. Expected output 3, actual output 5

For the most part, the test failures align with the known/manufactured bugs, however the updatesCoins() unit test could be an issue with the test logic, an issue with the card being used to test the function (in this case, it would make sense that the test would fail if ADVENTURER is played, and it is the first in the available testCards array), or an actual bug within the function. At the very least, the test outlines an actionable set of troubleshooting tasks that, if followed, should alleviate the issue(s).

UNIT TESTING

I'll start by saying I took a test suite approach to summarize the impact of all the tests, that is, I intentionally did not remove the .gcda file between each gcov execution in the MAKEFILE. I took this approach because it seemed like from an outsider's perspective, the most likely scenario would be that s/he would run all the available tests to get a holistic perspective on existing test coverage as a means to hunt-down a bug or continue to increase the code base' coverage. This cumulative approach is also significantly easier to read; a point worth considering when specificity can actually be a hindrance to targeting a problem.

This said, in the process of building the MAKEFILE, and ultimately in the process of building the individual tests, I did build it so that it could produce a non-cumulative output file. By removing the .gcda file between every gcov execution, it effectively exposes each test's strengths and weaknesses, and eliminates cross-coverage obfuscation. Of course, the downside to this is that the output file is excessively long, making manual review (and especially test coverage comparisons) especially time consuming. You can execute this by running the $ make exhaustiveTests which outputs to exhaustiveunittestresults.out

For clarification, the following results are from the unittestresults.out file, and are a cumulative summary of the effect of all existing tests. The gcov -b (branch probabilities) and -f (function summaries) flags were set in the runtests MAKEFILE target. Additionally the dominion.c.gov file is concatenated to the end of the unittestresults.out file.

GCOV BRANCH PROBABILITIES & FUNCTION SUMMARIES

```
Function 'updateCoins'
Lines executed:100.00% of 11
Branches executed:100.00% of 8
Taken at least once:100.00% of 8
No calls

Function 'gainCard'
Lines executed:100.00% of 13
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 1

Function 'discardCard'
Lines executed:84.62% of 13
Branches executed:100.00% of 6
Taken at least once:50.00% of 6
No calls

Function 'cardEffect'
Lines executed:5.37% of 205
Branches executed:12.85% of 179
Taken at least once:3.35% of 179
Calls executed:8.62% of 58

Function 'smithy_card'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
Calls executed:100.00% of 3

Function 'adventurer_card'
Lines executed:100.00% of 18
Branches executed:100.00% of 12
Taken at least once:100.00% of
12
Calls executed:100.00% of 3
```

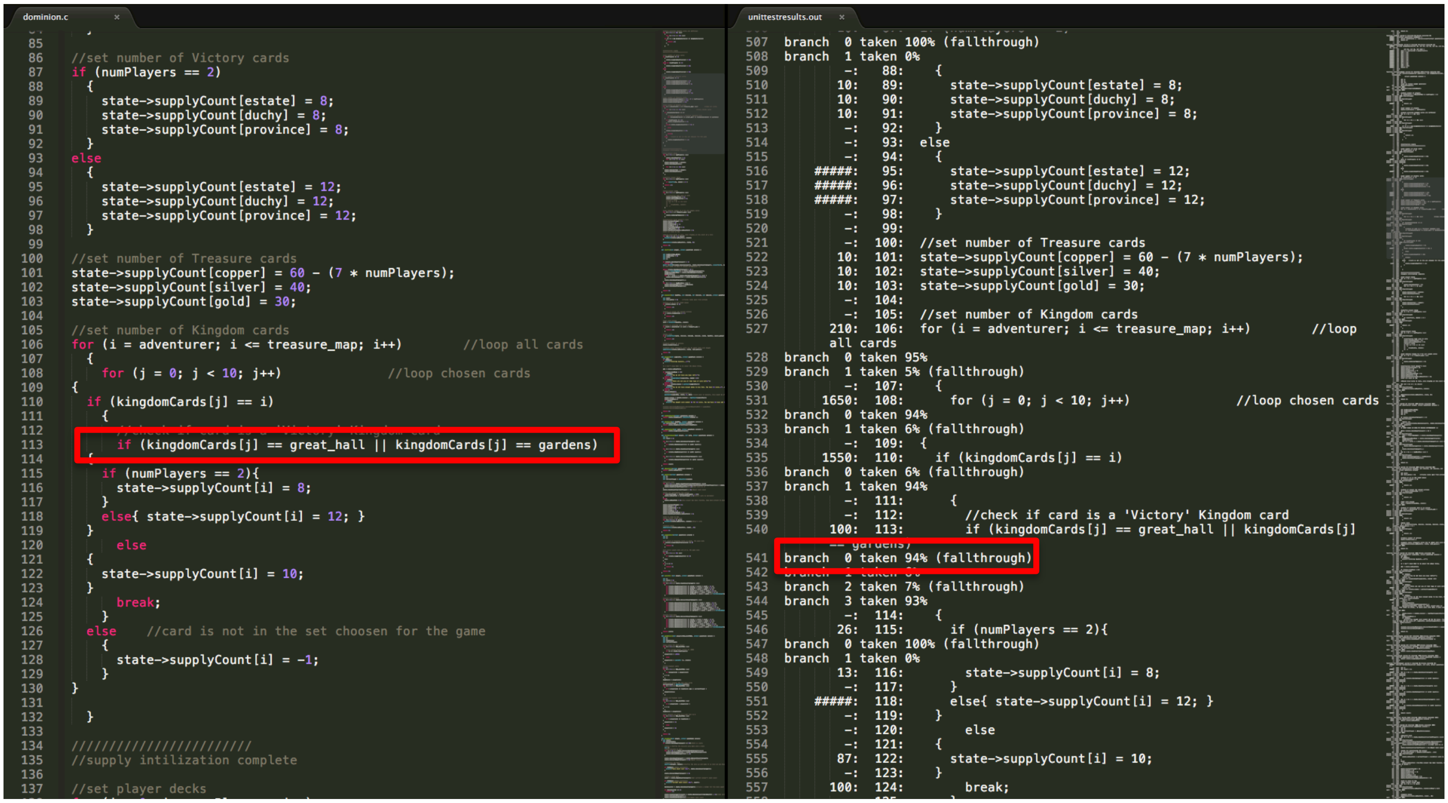
```
Function 'council_room_card'
Lines executed:100.00% of 9
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 4

Function 'village_card'
Lines executed:100.00% of 6
No branches
Calls executed:100.00% of 3

Function 'great_hall_card'
Lines executed:0.00% of 6
No branches
Calls executed:0.00% of 3

Function 'getCost'
Lines executed:23.33% of 30
Branches executed:100.00% of 28
Taken at least once:17.86% of 28
No calls

Function 'drawCard'
Lines executed:95.45% of 22
Branches executed:100.00% of 6
Taken at least once:83.33% of 6
Calls executed:100.00% of 1

Function 'getWinners'
Lines executed:0.00% of 24
Branches executed:0.00% of 22
Taken at least once:0.00% of 22
Calls executed:0.00% of 2

Function 'scoreFor'
Lines executed:100.00% of 24
Branches executed:100.00% of 42
Taken at least once:69.05% of 42
Calls executed:0.00% of 3
```

```
Function 'isGameOver'
Lines executed:100.00% of 10
Branches executed:100.00% of 8
Taken at least once:100.00% of 8
No calls

Function 'endTurn'
Lines executed:100.00% of 20
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 3

Function 'whoseTurn'
Lines executed:100.00% of 2
No branches
No calls

Function 'fullDeckCount'
Lines executed:0.00% of 9
Branches executed:0.00% of 12
Taken at least once:0.00% of 12
No calls

Function 'supplyCount'
Lines executed:100.00% of 2
No branches
No calls

Function 'handCard'
Lines executed:100.00% of 3
No branches
Calls executed:100.00% of 1

Function 'numHandCards'
Lines executed:100.00% of 2
No branches
Calls executed:100.00% of 1
```

```
Function 'buyCard'
Lines executed:76.92% of 13
Branches executed:100.00% of 6
Taken at least once:50.00% of 6
Calls executed:100.00% of 4

Function 'playCard'
Lines executed:78.57% of 14
Branches executed:100.00% of 10
Taken at least once:60.00% of 10
Calls executed:100.00% of 3

Function 'shuffle'
Lines executed:100.00% of 16
Branches executed:100.00% of 8
Taken at least once:100.00% of 8
Calls executed:100.00% of 2

Function 'initializeGame'
Lines executed:83.87% of 62
Branches executed:95.65% of 46
Taken at least once:80.43% of 46
Calls executed:100.00% of 5

Function 'kingdomCards'
Lines executed:0.00% of 13
No branches
No calls

Function 'newGame'
Lines executed:0.00% of 3
No branches
No calls

Function 'compare'
Lines executed:100.00% of 6
Branches executed:100.00% of 4
Taken at least once:100.00% of 4
No calls

File 'dominion.c'
Lines executed:48.22% of 562
Branches executed:53.96% of 417
Taken at least once:36.93% of 417
Calls executed:39.00% of 100
```

The highlighted functions are those that were covered in Assignment 3. It's generally satisfying to know that they all have 100% line and branch coverage. As mentioned above in the BUGS section, I cannot confirm that the coverage is 100% accurate, but it's a start. The findings are particularly nice in the sense that they make coverage shortages explicit and specific. I know exactly what functions could use additional coverage and what functions are, at least as it relates to blanket coverage, covered. I didn't take the time to determine whether each function had open boundaries or, if not, what those boundary values are, so I cannot speak to the boundary test coverage. I used pre-defined MAX and MIN values in a couple tests, but I hardly think this can be considered adequate boundary testing.

The concatenated output of dominion.c.gcov with branch probabilities appears to be where a fair portion of the gcov value rests. A simple cmd + f to find "fallthrough" accompanied by a relatively high percentage seems to be a fruitful means to find bug opportunities.



```
  //set number of Victory cards
  if (numPlayers == 2)
    {
      state->supplyCount[estate] = 8;
      state->supplyCount[duchy] = 8;
      state->supplyCount[province] = 8;
    }
  else
    {
      state->supplyCount[estate] = 12;
      state->supplyCount[duchy] = 12;
      state->supplyCount[province] = 12;
    }

  //set number of Treasure cards
  state->supplyCount[copper] = 60 - (7 * numPlayers);
  state->supplyCount[silver] = 40;
  state->supplyCount[gold] = 30;

  //set number of Kingdom cards
  for (i = adventurer; i <= treasure_map; i++)        //loop all cards
    {
      for (j = 0; j < 10; j++)                //loop chosen cards
  {
    if (kingdomCards[j] == i)
      {
        //check if card is a 'Victory' Kingdom card
        if (kingdomCards[j] == great_hall || kingdomCards[j] == gardens)
    {
      if (numPlayers == 2){
        state->supplyCount[i] = 8;
      }
      else{ state->supplyCount[i] = 12; }
    }
        else
    {
      state->supplyCount[i] = 10;
    }
        break;
      }
    else    //card is not in the set choosen for the game
      {
        state->supplyCount[i] = -1;
      }
  }

    }

  ////////////////////////
  //supply intilization complete

  //set player decks
```

```
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
        -:   88:    {
       10:   89:      state->supplyCount[estate] = 8;
       10:   90:      state->supplyCount[duchy] = 8;
       10:   91:      state->supplyCount[province] = 8;
        -:   92:    }
        -:   93:  else
        -:   94:    {
    #####:   95:      state->supplyCount[estate] = 12;
    #####:   96:      state->supplyCount[duchy] = 12;
    #####:   97:      state->supplyCount[province] = 12;
        -:   98:    }
        -:   99:
        -:  100:  //set number of Treasure cards
       10:  101:  state->supplyCount[copper] = 60 - (7 * numPlayers);
       10:  102:  state->supplyCount[silver] = 40;
       10:  103:  state->supplyCount[gold] = 30;
        -:  104:
        -:  105:  //set number of Kingdom cards
      210:  106:  for (i = adventurer; i <= treasure_map; i++)        //loop
      all cards
branch  0 taken 95%
branch  1 taken 5% (fallthrough)
        -:  107:    {
     1650:  108:      for (j = 0; j < 10; j++)                //loop chosen cards
branch  0 taken 94%
branch  1 taken 6% (fallthrough)
        -:  109:  {
     1550:  110:    if (kingdomCards[j] == i)
branch  0 taken 6% (fallthrough)
branch  1 taken 94%
        -:  111:      {
        -:  112:        //check if card is a 'Victory' Kingdom card
      100:  113:        if (kingdomCards[j] == great_hall || kingdomCards[j]
      == gardens)
branch  0 taken 94% (fallthrough)
branch  1 taken 6%
branch  2 taken 7% (fallthrough)
branch  3 taken 93%
        -:  114:    {
       26:  115:      if (numPlayers == 2){
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
       13:  116:        state->supplyCount[i] = 8;
        -:  117:      }
    #####:  118:      else{ state->supplyCount[i] = 12; }
        -:  119:    }
        -:  120:        else
        -:  121:    {
       87:  122:      state->supplyCount[i] = 10;
        -:  123:    }
      100:  124:        break;
```

UNIT TESTING EFFORTS

I've written very simple inline unit tests for a small handful of larger projects in the past, but those were written largely as a means to eliminate the sources of a larger bug/problem. While I initially began writing the tests with the intention of providing the best possible coverage, it was only after I began running gcov, and after re-watching the lectures on random testing, (and after speaking with some classmates that had far more complex testing conventions that I employed) that I realized that I could be doing a significantly better job. That said, for an initial effort, I think it was a reasonable effort and an exceptionally eye-opening experience.

The basic approach I took for the function tests was to break down each function's objective and determine whether completing the objective could be influenced by parameters the function accepted. I generally wrote individual tests that made comparisons between a previous game state and the a game state influenced by the function. I would add or subtract a given struct attribute to the function-effected game state as a means to create an equality between the two states. Knowing that the addition or subtraction of the struct attributes were static, or hard-coded, I knew that if the game states varied, then something went wrong with the function as a result of a specific input/parameter.

I could have been more thorough, especially in hindsight, but the underlying approach, of which I obviously cannot take credit, proved to be sound. It generally took the same approach toward the card tests as well.