Erin Alltop
CS362 – Winter 2018
Assignment 4 – Random Testing

A note on compiling – The Makefile has rules for each random test case. To compile, run 'make clean' and type 'make randomtestadventurer', 'make randomtestcard1', and 'make randomtestcard2'. After this, the related *.out files will have the results and coverage of each test.

Random Testing

For this assignment, we were tasked with creating random testers for three cards, one of which is Adventurer and at least one other is a card we have previously tested. I started planning the random tester by thinking of how I could make the testing truly random. After some brain storming, I decided to randomize a game as much as possible for 1000 iterations. I randomized whose turn it was, the bonus coins, choices, and hand position. I originally attempted to randomize the number of players as well but was running into some undefined behavior that I was unable to debug properly within enough time, so I left the number of players not randomized.

For each iteration, the game is re-randomized and copied into a static game state that can be compared to the test game state. Then I called the cardEffect function and began testing for as many things as I could think of to obtain as much coverage as possible. For each card I tested the most obvious effects that I know the card can do from the card description, as well as other potential effects that should not occur, to hopefully catch those changes if any. Here are the tests that I made for each card –

Adventurer

- The current player draws two cards
- The two cards drawn comes from the current player's deck
- The first card drawn was a treasure card
- The second card drawn was a treasure card
- No bonus coins were obtained
- The other players' discard counts were unchanged
- The other players' deck counts were unchanged
- The other players' hand counts were unchanged
- The Victory card supply counts were unchanged
- The Kingdom card supply counts were unchanged

Great Hall

- The current player draws one card
- The one card drawn comes from the current player's deck
- The current player obtained one additional action
- No bonus coins were obtained
- The other players' discard counts were unchanged
- The other players' deck counts were unchanged
- The other players' hand counts were unchanged
- The Victory card supply counts were unchanged
- The Kingdom card supply counts were unchanged

Council Room

- The current player draws four cards
- The four cards drawn come from the current player's deck
- The current player obtained one additional buy
- No bonus coins were obtained
- The other players' discard counts were unchanged
- The other players' deck counts decreased by one
- The other players' hand counts increased by one
- The Victory card supply counts were unchanged
- The Kingdom card supply counts were unchanged

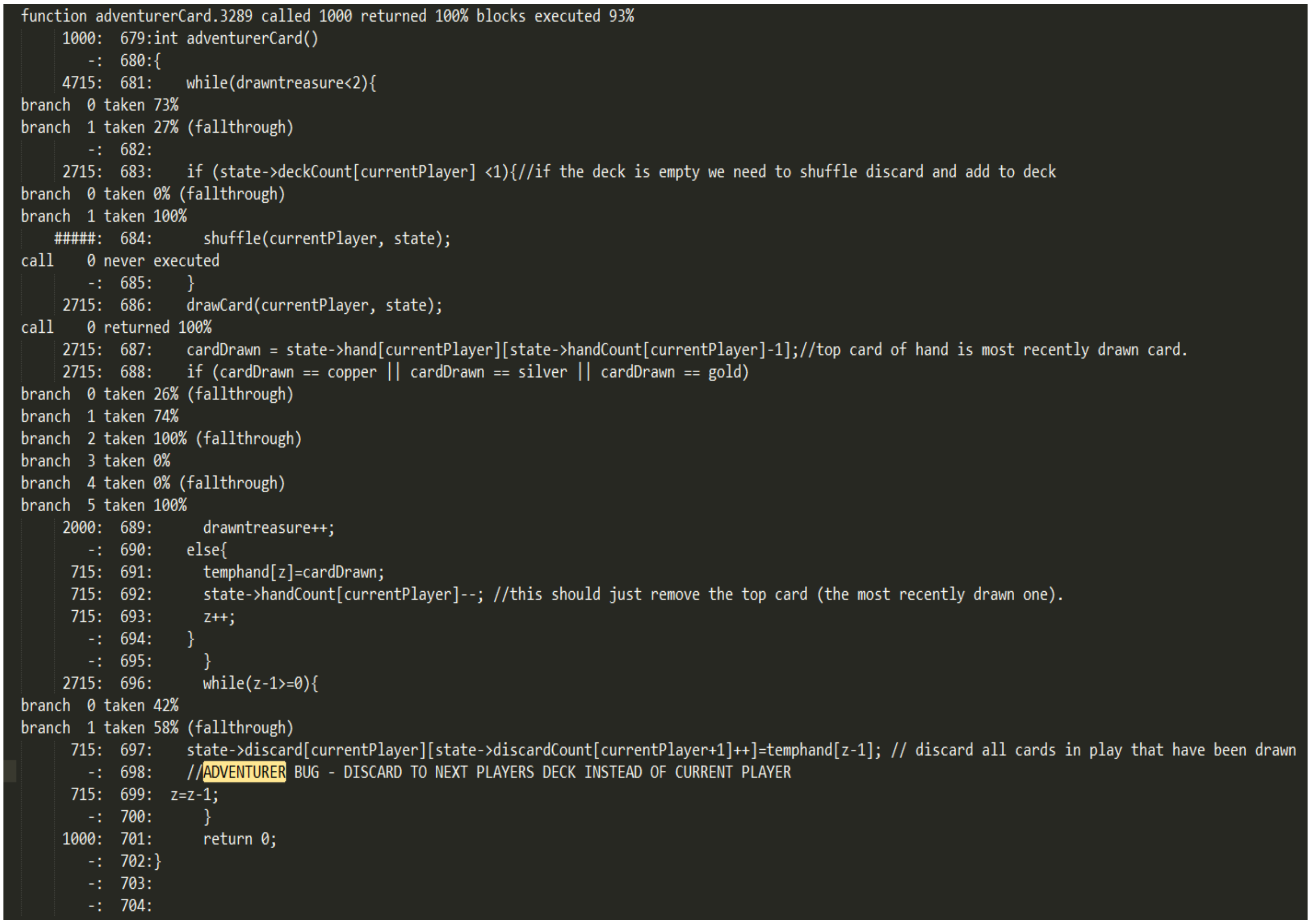
By checking the states of the other players and supply counts that should not have changed, we are able to catch unexpected changes to the game, and by randomizing the game we are able to test a variety of different game states to determine if any undefined behavior is happening. This also allowed me to get a greater amount of branch coverage to test as much as possible for the card than previously done in the unit testing.

Originally I had print statements for each failed test, but as the iterations were so great, I changed my approach to have a char array to hold the name of test that failed. If the test failed, I added the name of the test to the array to be printed at the end of the testing to indicate which tests failed. I also printed out how many tested failed vs how many tests were run.

Code Coverage

For the Adventurer card, I the function was called 1000 times, returned 100% and 93% of blocks were executed. It appears the 7% coverage lost was due to the shuffle function not executing. This would be because the deck never had few enough cards to force this function to execute. An improvement could be made here to set up my random games with a random amount of player deck, hand, and discard counts as well which would give an opportunity to invoke the shuffle function.

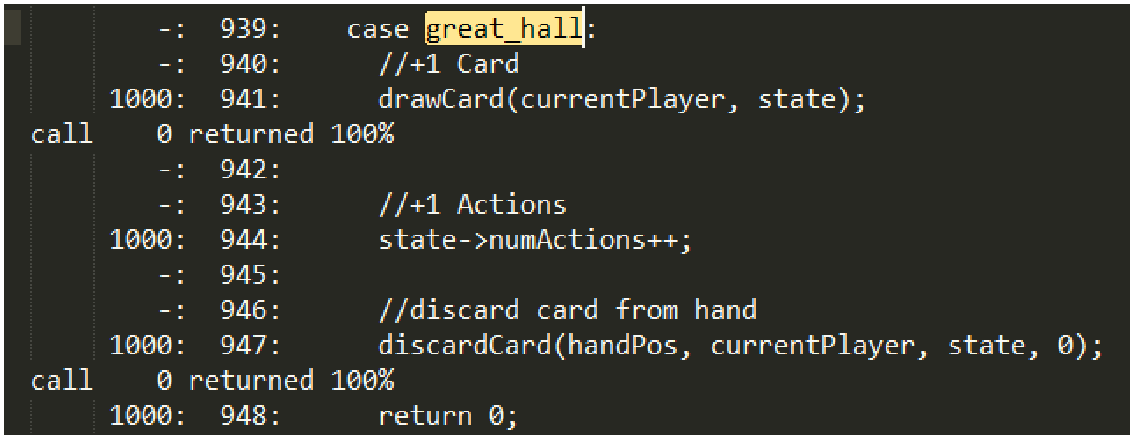
Adventurer Output



```
function adventurerCard.3289 called 1000 returned 100% blocks executed 93%
     1000:  679:int adventurerCard()
        -:  680:{
     4715:  681:    while(drawntreasure<2){
branch  0 taken 73%
branch  1 taken 27% (fallthrough)
        -:  682:
     2715:  683:    if (state->deckCount[currentPlayer] <1){//if the deck is empty we need to shuffle discard and add to deck
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:  684:      shuffle(currentPlayer, state);
call    0 never executed
        -:  685:    }
     2715:  686:    drawCard(currentPlayer, state);
call    0 returned 100%
     2715:  687:    cardDrawn = state->hand[currentPlayer][state->handCount[currentPlayer]-1];//top card of hand is most recently drawn card.
     2715:  688:    if (cardDrawn == copper || cardDrawn == silver || cardDrawn == gold)
branch  0 taken 26% (fallthrough)
branch  1 taken 74%
branch  2 taken 100% (fallthrough)
branch  3 taken 0%
branch  4 taken 0% (fallthrough)
branch  5 taken 100%
     2000:  689:      drawntreasure++;
        -:  690:    else{
      715:  691:      temphand[z]=cardDrawn;
      715:  692:      state->handCount[currentPlayer]--; //this should just remove the top card (the most recently drawn one).
      715:  693:      z++;
        -:  694:    }
        -:  695:      }
     2715:  696:      while(z-1>=0){
branch  0 taken 42%
branch  1 taken 58% (fallthrough)
      715:  697:    state->discard[currentPlayer][state->discardCount[currentPlayer+1]++]=temphand[z-1]; // discard all cards in play that have been drawn
        -:  698:    //ADVENTURER BUG - DISCARD TO NEXT PLAYERS DECK INSTEAD OF CURRENT PLAYER
      715:  699:  z=z-1;
        -:  700:      }
     1000:  701:      return 0;
        -:  702:}
        -:  703:
        -:  704:
```

For the Great Hall card, I achieved 100% branch coverage in testing. This only took a few seconds to complete. All functions within this card were called appropriately.

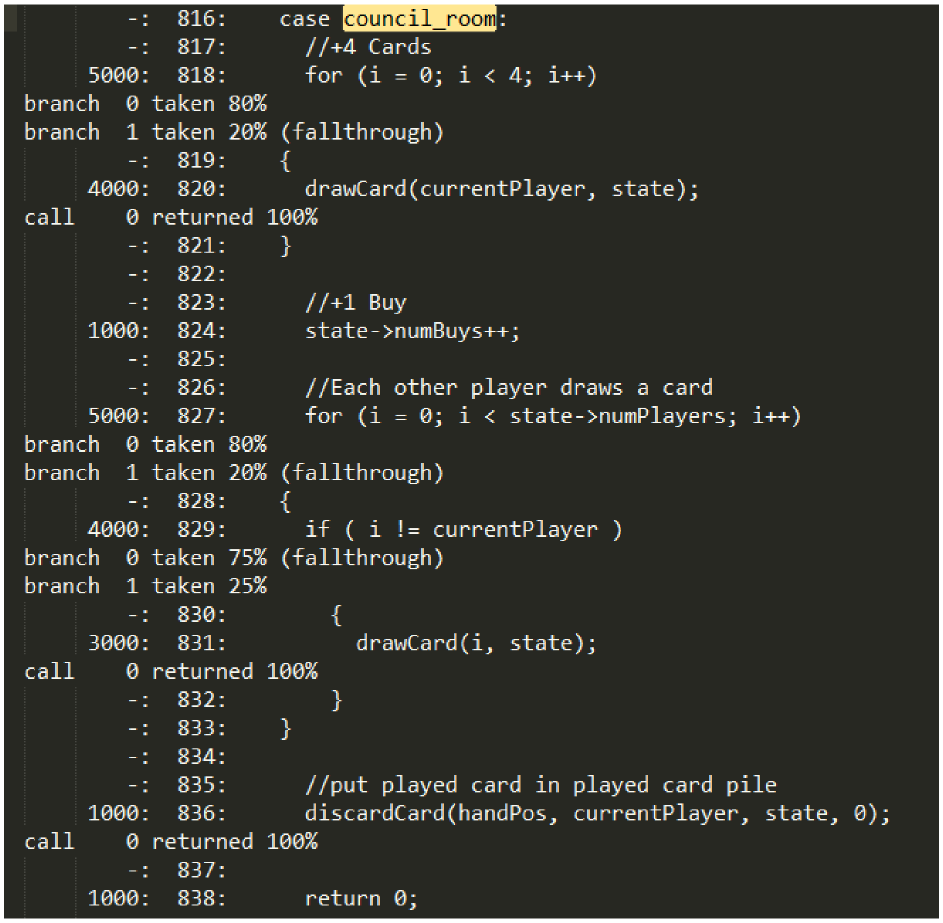
Great Hall Output



```
        -:  939:    case great_hall:
        -:  940:      //+1 Card
     1000:  941:      drawCard(currentPlayer, state);
call    0 returned 100%
        -:  942:
        -:  943:      //+1 Actions
     1000:  944:      state->numActions++;
        -:  945:
        -:  946:      //discard card from hand
     1000:  947:      discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
     1000:  948:      return 0;
```

Lastly, for the Council Room card I also achieved 100% branch coverage. Each function was appropriately taken and all branched were taken at least 20% of the time.

Council Room Output



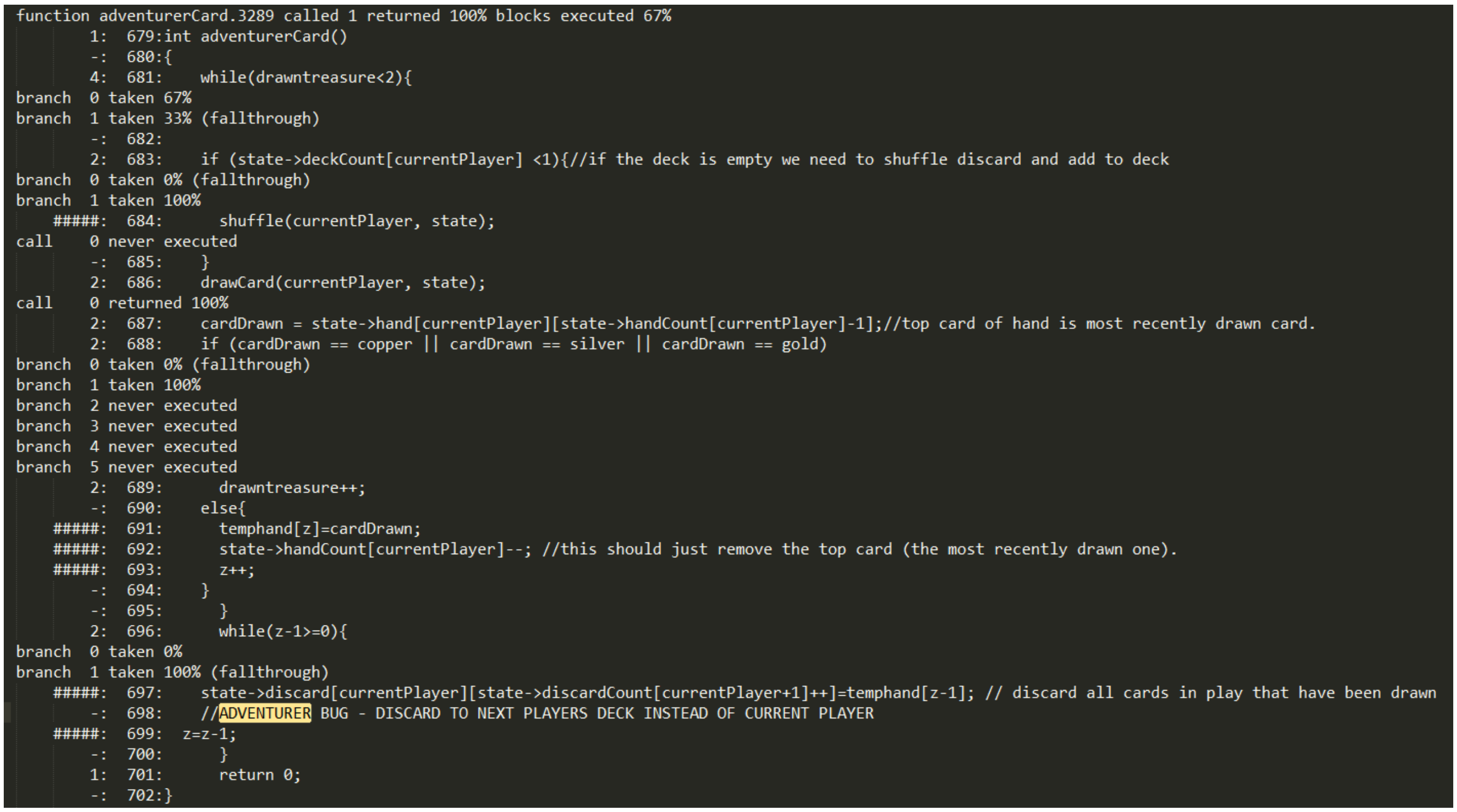
```
        -:  816:    case council_room:
        -:  817:      //+4 Cards
     5000:  818:      for (i = 0; i < 4; i++)
branch  0 taken 80%
branch  1 taken 20% (fallthrough)
        -:  819:    {
     4000:  820:      drawCard(currentPlayer, state);
call    0 returned 100%
        -:  821:    }
        -:  822:
        -:  823:      //+1 Buy
     1000:  824:      state->numBuys++;
        -:  825:
        -:  826:      //Each other player draws a card
     5000:  827:      for (i = 0; i < state->numPlayers; i++)
branch  0 taken 80%
branch  1 taken 20% (fallthrough)
        -:  828:    {
     4000:  829:      if ( i != currentPlayer )
branch  0 taken 75% (fallthrough)
branch  1 taken 25%
        -:  830:        {
     3000:  831:          drawCard(i, state);
call    0 returned 100%
        -:  832:        }
        -:  833:    }
        -:  834:
        -:  835:      //put played card in played card pile
     1000:  836:      discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
        -:  837:
     1000:  838:      return 0;
```

Unit vs Random

The first thing I needed to do was fix my coverage commands in my Makefile for my unit tests, which I did improperly for Assignment 3. After I correctly tested for the coverage within Dominion, I was able to evaluate the coverage and compare it to the random testing results.

For the Adventurer card, I definitely had better coverage in my random testing. For the unit test, I achieved 67% branch coverage, versus the 93% in random testing.

Adventurer Unit Test Output

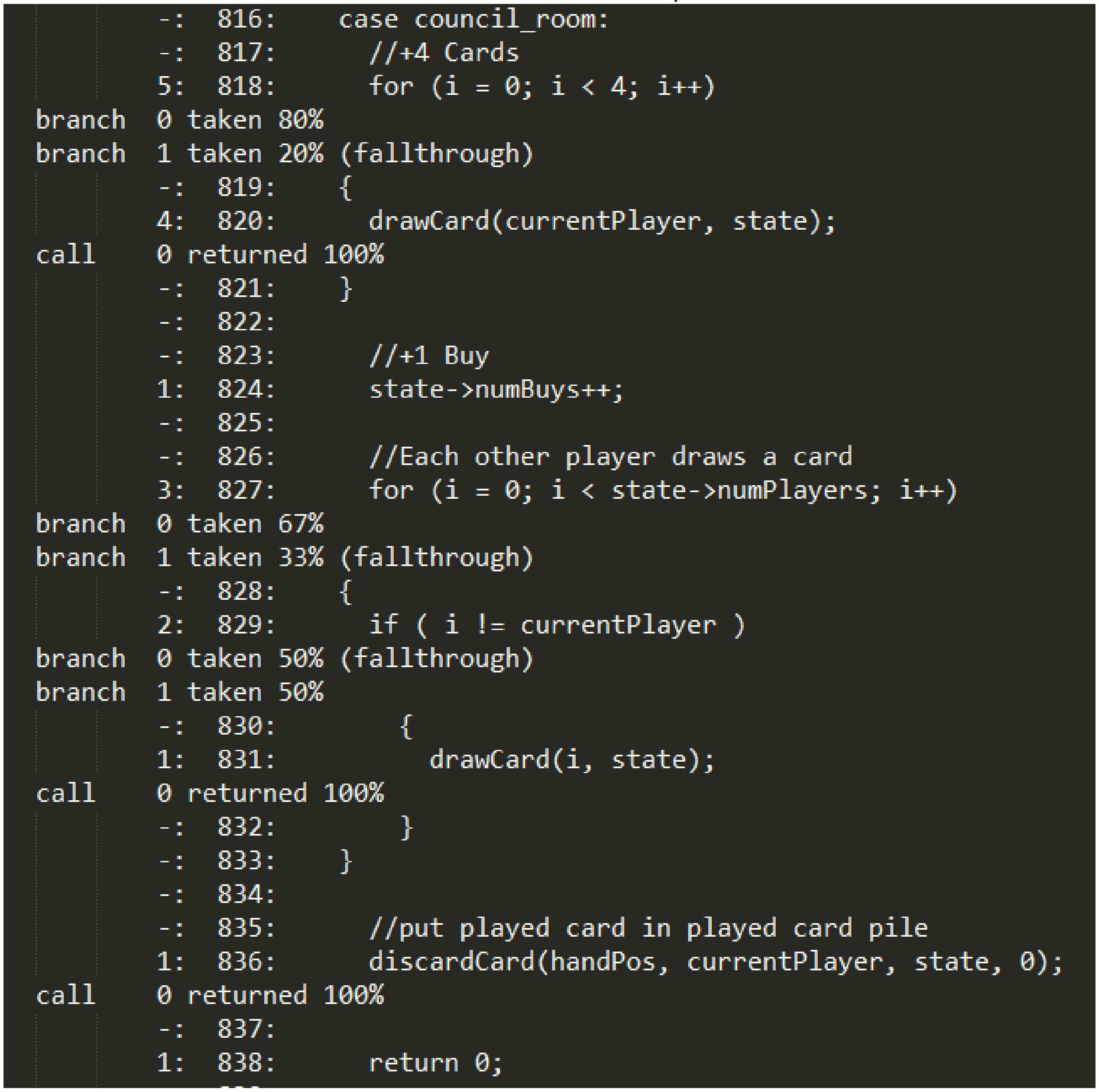


```
function adventurerCard.3289 called 1 returned 100% blocks executed 67%
        1:  679:int adventurerCard()
        -:  680:{
        4:  681:    while(drawntreasure<2){
branch  0 taken 67%
branch  1 taken 33% (fallthrough)
        -:  682:
        2:  683:    if (state->deckCount[currentPlayer] <1){//if the deck is empty we need to shuffle discard and add to deck
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:  684:      shuffle(currentPlayer, state);
call    0 never executed
        -:  685:    }
        2:  686:    drawCard(currentPlayer, state);
call    0 returned 100%
        2:  687:    cardDrawn = state->hand[currentPlayer][state->handCount[currentPlayer]-1];//top card of hand is most recently drawn card.
        2:  688:    if (cardDrawn == copper || cardDrawn == silver || cardDrawn == gold)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
branch  2 never executed
branch  3 never executed
branch  4 never executed
branch  5 never executed
        2:  689:      drawntreasure++;
        -:  690:    else{
    #####:  691:      temphand[z]=cardDrawn;
    #####:  692:      state->handCount[currentPlayer]--; //this should just remove the top card (the most recently drawn one).
    #####:  693:      z++;
        -:  694:    }
        -:  695:      }
        2:  696:      while(z-1>=0){
branch  0 taken 0%
branch  1 taken 100% (fallthrough)
    #####:  697:    state->discard[currentPlayer][state->discardCount[currentPlayer+1]++]=temphand[z-1]; // discard all cards in play that have been drawn
        -:  698:    //ADVENTURER BUG - DISCARD TO NEXT PLAYERS DECK INSTEAD OF CURRENT PLAYER
    #####:  699:  z=z-1;
        -:  700:      }
        1:  701:      return 0;
        -:  702:}
```

Because my unit test only tested one scenario instead of many different scenarios, only certain branches were ever executed. In the random testing output, I can see that only one call was never executed (the scenario where 0 cards are in the deck), but in unit testing there are 5 branches that are never executed because the unit test did not cover every scenario extensively.

For the Council Room card which I also previously tested, the coverage for the unit test was 100% branch coverage like the random testing.

Council Room Unit Test Output



```
        -:  816:    case council_room:
        -:  817:      //+4 Cards
        5:  818:      for (i = 0; i < 4; i++)
branch  0 taken 80%
branch  1 taken 20% (fallthrough)
        -:  819:    {
        4:  820:      drawCard(currentPlayer, state);
call    0 returned 100%
        -:  821:    }
        -:  822:
        -:  823:      //+1 Buy
        1:  824:      state->numBuys++;
        -:  825:
        -:  826:      //Each other player draws a card
        3:  827:      for (i = 0; i < state->numPlayers; i++)
branch  0 taken 67%
branch  1 taken 33% (fallthrough)
        -:  828:    {
        2:  829:      if ( i != currentPlayer )
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
        -:  830:        {
        1:  831:          drawCard(i, state);
call    0 returned 100%
        -:  832:        }
        -:  833:    }
        -:  834:
        -:  835:      //put played card in played card pile
        1:  836:      discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
        -:  837:
        1:  838:      return 0;
```

While this is a good coverage, this is also a simpler card effect, and there are not as many scenarios that need to be tested to achieve the desired branch coverage.

Overall, the random testing has a much greater potential to achieve 100% branch coverage. To achieve the same amount of coverage with unit testing, you need to be able to account for every possible scenario which is easier to do in simple functions (like the Council Room card), but more difficult to do in more complicated functions (like the Adventurer card). The Dominion card program is a relatively simple program compared to other programs that have millions of lines of code. Unit testing for every possible scenario would be next to impossible in a very large and complicated program.

However, I believe if you were looking to test for a specific fault in a program, unit testing might be a better choice in many instances. For example, if I wanted to test the Adventurer card for its functionality in drawing treasure cards only, I could use a unit test to manipulate the deck treasure cards to test for copper, silver, and gold. In this way I can test this specific function within the function and quickly see if there is an issue or if it works as intended.

If I ran similar scenarios with random testing, even with accounting for other changes, I might not be able to see exactly where the fault occurs if there is one and in what scenario. A stated before, the unit tests are useful in small scale instances, but in even a large program this can be used to better control the test and the specific instances I am testing.