Class: CS-362
Term: Summer 2017
Author: Jon-Eric Cook
Date: July 30, 2017
Assignment: #4

Random Testing

Development of the random testers for adventurerCard, smithyCard and council_roomCard was tedious and meticulous. Starting off, the example code from testDrawCard was used to get an idea of how to do random testing in the first place. Once this code was fully understood, it was modified to be used on the above functions. Modifying the code meant tweaking it so that it would not cause a crash in any of the functions.

For the smithyCard random tester, one modification was setting the playedCardCount state variable to something random but within the bounds of 0 and MAX_DECK-1. This had to be done because the discardCard function that was called in smithyCard increments the playedCardConut variable. If this were to start out at the upper bound and then be incremented, it was would cause a seg fault as it would cause an attempt to access out of bounds of an array. Another modification was setting the whoseTurn variable to the randomly selected player. This had to be done because the cardEffect function called whoseTurn function to get the current player. To be able to properly compare game states between the one that was modified manually and the one modified by cardEffect, the same player had to be used. In regards to the checkSmithyCard function, this was were the manual modifications to the game state took place. All the actions that would have taken place in the smithyCard function, were executed in this function. As a result, it was then possible to compare game states and check for bugs. The way of checking if the two game states matched are as follows: compared hand counts, compared deck counts, checked if drawCard failed, checked if cardEffect failed, checked if discardCard failed. In the randomtestcard1.out file, all the fail counts were reported. By increasing the number of iterations, the % coverage was increased because more random values were tested against the code and this caused more of the code to be ran. Also, in an attempt to check the correctness of the specifications, each attribute that was changed due to the smithyCard function executing was compared to its counterpart in the correct result of the smithyCard function being executed manually.

For the council_roomCard random tester, one modification was setting the playedCardCount state variable to something random but within the bounds of 0 and MAX_DECK-1. This had to be done because the discardCard function that was called in council_roomCard increments the playedCardConut variable. If this were to start out at the upper bound and then be incremented, it was would cause a seg fault as it would cause an attempt to access out of bounds of an array. Another modification was setting the whoseTurn variable to the randomly selected player. This had to be done because the cardEffect function called whoseTurn function to get the current player. To be able to properly compare game states between the one that was modified manually and the one modified by cardEffect, the same player had to be used. Another modification was setting numBuys to 1 because the council_roomCard function increments it. The last and largest modification was setting all the other players in the game to have the same starting deck, hand and discard count values as the selected player. This was done because the council_roomCard caused all the other players to call the drawCard function. In regards to the checkCouncil_RoomCard function, this was were the manual modifications to the game state took place. All the actions that would have taken place in the council_roomCard function, were executed in this function. As a result, it was then possible to compare game states and check for bugs. The way of checking if the two game states matched are as follows: compared hand counts, compared deck counts, checked if drawCard failed, checked if cardEffect failed, checked if discardCard failed, compared the other players hand and deck counts, compared the numBuys counts. In the randomtestcard2.out file, all the fail counts were reported. By increasing the number of iterations, the % coverage was increased because more random values were tested against the code and this caused more of the code to be ran. Also, in an attempt to check the correctness of the specifications, each attribute that was changed due to the council_roomCard function executing was compared to its counterpart in the correct result of the smithyCard function being executed manually.

For the adventurerCard random tester, one modification was setting the whoseTurn variable to the randomly selected player. This had to be done because the cardEffect function called whoseTurn function to get the current player. To be able to properly compare game states between the one that was modified manually and the one modified by cardEffect, the same player had to be used. Another modification was setting the discardCount to 0. This had to be done because while the adventurerCard function looks for treasure cards it discards cards that aren't treasure cards. By this happening, if the discardCount is not zero it can grow larger than 500, the limit of the hand and deck array. If this were to be allowed to happen, it would cause a seg fault. Another modification is setting the deck count and hand count to a random number within the bounds of 3 and MAX_DECK or MAX_HAND. This had to be done because if the deck or hand were to be empty the adventurerCard would never find treasure card to be able to exit the while loop. The last main modification was placing at a minimum of three treasure cards in the deck. This was to avoid the infinite loop mentioned above. In regards to the checkAdventurerCard function, this was were the manual modifications to the game state took place. All the actions that would have taken place in the adventurerCard function, were executed in this function. As a result, it was then possible to compare game states and check for bugs. The way of checking if the two game states matched are as follows: compared hand counts, compared deck counts, checked if drawCard failed, checked if cardEffect failed, checked if shuffle failed, compared treasure counts. In the randomtestadventurer.out file, all the fail counts were reported. By increasing the number of iterations, the % coverage was increased because more random values were tested against the code and this caused more of the code to be ran. Also, in an attempt to check the correctness of the specifications, each attribute that was changed due to the adventurerCard function executing was compared to its counterpart in the correct result of the adventurerCard function being executed manually.

Code Coverage

The code coverage for smithyCard, council_roomCard and adventurerCard were all near 100%. Because both smithyCard and council_roomCard were not terribly complicated functions, they both had 100% coverage. Also, because it was possible to randomly create a game state that would touch each line of code. adventurerCard, on the other hand, was just short of 100% coverage. This was due to not being to test an empty deck. If the deck were to be empty, it would cause a seg fault. The following if statement was not executed:

```
if (pre.deckCount[p] < 1)
{
        shuffle(p, &pre);
}
```

The empty deck scenario would cause second while loop to attempt to access out of boundary memory for the temphand array and the discard array. Many hours were spent analyzing the adventurerCard code and no random scenario fit the bill to be able to execute all the code.

Running the random tester for smithyCard with 10,000 iterations took around 8 seconds. As an example of 100% code coverage, bellow is the coverage report for the smithyCard function in dominion.c file.

```
      -:  676:void smithyCard(int currentPlayer, int handPos, struct gameState *state){
      -:  677:  int i;
      -:  678:  //+3 Cards
      -:  679:  // bug -> allows the player to add 4 cards to their hand, instead of only 3
 100000:  680:  for (i = 0; i < 4; i++)
      -:  681:  {
  40000:  682:          drawCard(currentPlayer, state);
  40000:  683:  }
      -:  684:
      -:  685:  //discard card from hand
  10000:  686:  discardCard(handPos, currentPlayer, state, 0);
  10000:  687:}
```

<u>**Unit vs Random**</u>

Oddly enough, the coverage for the unit tests created in assignment-3 was very similar to that of the random tests. All the coverage for the unit tests was 100%. Although the percentage of coverage were similar, the number of times a line of code was executed was drastically different. Comparing the random tester to the unit tester, each has their own strengths and weaknesses. The unit tests have a lower ability to detect faults as it may only test a few cases and only once. On the other hand, random testing can be more rigorous in its tests as it can test non normal values and combinations of them, hence the randomness. The random test approach has a lot better ability to detect faults because for one, it is not just one test, it could be millions of different tests. Where the unit tests shine is in their ability to be written up relatively quickly and to at least confirm the function acts as designed. The random tester shines in that it can really stress the function by using weird and odd values to try and break it.

An example of this is the smithyCard function. In the unit tests, each line of code was ran at least once and a generic game state was used. Now, in the random testers, each line of code was ran at least 10,000 times and each time there was a different game state. Granted, there was a lot more work in setting up the random tester than the unit test, the proof is in the pudding as they say. Each test has its place and use. There really isn't a once size fits all for all things to be tested.

On the topic of when to use a random tester or a unit tester, it comes down to when in the project one is. If the project is in its early stages and the code is prone to be changed, a unit tester may be used just to confirm that the functions react and work as designed. Seeing how it takes more time and effort to construct a random tester, it would be a bummer to do all that work and then have the source code change and break the random tester. Now, when the project is closer to being set in stone, a random tester could be used to really do a stress test on the source code. Seeing how a lot more runs can be done in random testing, this could be used to test the scenarios beyond just the edges cases that may be tested in unit testing.