Tang, Kevin

CS362

Test Report 2

5/17/2012

**Adventurer Testing**

**Usage:***make clean*

*make all*

**View:** *test.out*

**Approach:** Random testing.

**Relevant Fields:** cardEffect.adventurer, deckCount, hand, handCount, discard, discardCout

**Checks:**

* Only a maximum of two treasure cards placed into hand
* deckCount accurately reflects the changes
* Discard stack properly adjusted
* Discard stack count properly adjusted
* If any other part of the struct was changed

**Results:** Number of errors detected-if any.

**Analysis of Test**

The test is designed to run one thousand tests. I reworked my testBuyCard.c so that it would model an adventurer test. In main of testAdventurer.c the game state is first filled with random information then particular fields in the game state structure are assigned random values that lie within valid ranges. I feel that this is the most effective way to test for arbitrary values that may cause problems. The snippet of source code below shows the randomization and assignments.

for (n = 0; n < NUM\_TESTS; n++) {

for (i = 0; i < sizeof(struct gameState); i++) {

((char\*)&game)[i] = (int)(Random() \* MAX\_DECK);

}

Players = (int)(Random() \* 2);

game.numBuys = (int)(Random() \* 4);

game.coins = (int)(Random() \* 10);

game.supplyCount[Card] = (int)(Random() \* 30);

game.whoseTurn = Players;

game.deckCount[Players] = (int)(Random() \* MAX\_DECK);

game.discardCount[Players] = (int)(Random() \* MAX\_DECK);

//put some valid cards in the deck.

for(c = 0; c < game.deckCount[Players]; ++c) {

game.deck[Players][c] = (int)(Random() \* CARD\_VAR);

}

//put some valid cards in the discard.

for(c = 0; c < game.discardCount[Players]; ++c) {

game.discard[Players][c] = (int)(Random() \* CARD\_VAR);

}

game.handCount[Players] = (int)(Random() \* MAX\_HAND);

//set the first card to be adventurer

game.hand[Players][0] = adventurer;

//fill up the hand with cards.

for(c=1;c<game.handCount[Players];++c){

game.hand[Players][c] = k[(int)(Random()\*10)];

}

Once values are assigned they are copied then sent to the actual testing code; testAdventurer; where one of the copies is sent to cardEffect and the other is manually changed to simulate the result of an adventurer card. Then, the test calculates if it is possible to find two treasure cards and if these are indeed the cards that were drawn to the player’s hand. These values are checked with the returned state from cardEffect. If everything matches up to here a comparison is done on each member of the game state structs. While debugging my test I stumbled upon something strange: if I compared both structs as a whole the comparison would fail sometimes but if I individually compared each member then it would pass (maybe some padding problems?). If everything is still in order then the test returns and the iteration passes. If something fails then the iteration records the fault in which it supplies to the user later. Below is a snippet of the vital code in testAdventurer.

if((pre->handCount[currentPlayer] == post->handCount[currentPlayer]+2) || (pre->handCount[currentPlayer] == post->handCount[currentPlayer]+1) || (pre->handCount[currentPlayer] == post->handCount[currentPlayer])){

//printf("Past 2\n");

//Count the treasure cards in the modified hand.

for(c = 0; c < pre->handCount[currentPlayer]; ++c){

currentCard = pre->hand[currentPlayer][c];

if(currentCard == copper || currentCard == silver || currentCard == gold){

TreasureCounta++;

}

}

//Count the treasure cards in the original hand.

for(c = 0; c < post->handCount[currentPlayer]; ++c){

currentCard = post->hand[currentPlayer][c];

if(currentCard == copper || currentCard == silver || currentCard == gold){

TreasureCountb++;

}

}

//Compare the results of treasure cards to see if they are within the proper range.

if((TreasureCountb+2 == TreasureCounta) || (TreasureCountb+1 == TreasureCounta) || (TreasureCountb == TreasureCounta)){

post->handCount[currentPlayer] = pre->handCount[currentPlayer];

memcpy(post->hand[currentPlayer],pre-> hand[currentPlayer],sizeof(int)\*MAX\_HAND);

}else{ return 1;}

}else{ return 1;}

**Analysis of Testing**

**Mine:**

testAdventurer was used to successfully test my own implementation of adventurer as well as a couple other college’s implementation. Upon running the test against my own adventurer I came across a strange behavior with memcmp (noted above) but was able to find an alternate way to test the struct. My implementation seems to be in working order.

**Ellingsn:**

When I ran testAdventurer on their code every iteration succeeded. However, my test reported that none of the proper changes were made either. Upon inspection it seems that they have not implemented adventurer. This also goes for their buyCard implementation. They had included only the action of inserting the bought card into the player’s discard pile. This is only part of the functionality that buyCard is required to have.

**Nguyenta:**

After running testAdventurer on their code I discovered that they had the same implementation as Ellingsn, thus the test results were identical. This is also the case for buyCard.

**Shearini:**

When I ran testAdventurer on their code the test resulted in a segmentation fault immediately. After examining their code I discovered that their implementation did not model the proper behavior of adventurer. Their implementation placed a non-treasure card that is drawn directly back into the discard instead of a different location. Also, they decrement numActions within adventurer-something that is already done upon returning from cardEffect.

**Mcconnjo:**

When I ran my test on their code I immediately resulted in a segmentation fault. Upon viewing their code I saw that their implementation decremented the players deck twice per iteration. However, I think the problem resides elsewhere since gcov shows that the test was unable to touch adventurer. Additionally, their buyCard implementation is incomplete.

**Gcov:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Coverage:** | % of dominion.c | % of adventurer | % of buyCard |
| Me | 13.15% | 100% | 100% |
| Ellingsn | 5.77% | 100% (2 lines) | 100% |
| Nguyenta | 5.77% | 100% (2 lines) | 100% |
| Shearini | 4.69% | 0% | 100% |
| Mcconnjo | 4.84% | 0% | 64% |

**Plans:**

For the future I would like to gain more insight on the strange memcmp problems that I was running into.

**Conclusion**

Writing testAdventurer was entertaining because I was able to really tweak and control the game state. While I did run into problems with memcmp, the work around seem allowed me to really find out which fields in particular had problems versus trying to visually compare the structs upon a compare failure. I would have liked to have seen more completed implementations but of the ones I encountered I enjoyed seeing their ideas.