Tang, Kevin

CS362

Test Report 1

4/29/2012

**Buy Card Testing**

**Usage:***make clean*

*make testdom*

*./testBuyCard*

**Approach:** Random testing.

**Relevant Fields:** numBuys, coins, discard, discardCount, supplyCount.

**Checks:**

* Ample coins required for purchase
* Card existence in supply
* Discard stack properly adjusted
* Discard stack count properly adjusted
* Number of buys remaining properly adjusted
* buyCard does not modified anything it shouldn’t

**Results:** Number of errors detected.

**Analysis of Test**

The test is designed to run two thousand tests. In main of testBuyCard.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 (int i = 0; i < sizeof(struct gameState); i++) {

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

}

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

RandomSeedVal = floor(Random() \* 121);

Card = k[(int)floor(Random() \* CARD\_VAR)];

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

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

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

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

game.whoseTurn = Players;

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

Once values are assigned they are sent to the actual testing code; testBuyCard; which checks the provided game state structure against a copy of the structure that is returned by buyCard. I feel that while it is basic test, testBuyCard is effective in finding bugs within buyCard. Prior to sending the game state structure to buyCard, testBuyCard first checks to see if the random state should theoretically be permitted to buy a card and flags if it can or cannot. Regardless of this flag, the game state is sent to buyCard. Upon return, if buyCard returns a SUCCESS status then testBuyCard checks to see if this aligns with the flag. This identifies whether buyCard should’ve failed or succeed and throws a fault appropriately. By now, if all is well, testDrawCard compares the returned state to a known correct version of the state. Below is a snippet of the vital code in testBuyCard.

//*[post = modified by buyCard; pre = original]//*

if(pre->coins < getCost(card) || pre->coins <= 0){

++faults;

}

if(supplyCount(card, pre) <= 0){

++faults;

}

ret = buyCard(card, pre);

if(ret==0 && faults <= 0){ //buyCard success

post.phase = 1;

post.coins = post.coins - getCost(card);

post.supplyCount[card]--;

post.discard[currentPlayer][post.discardCount[currentPlayer]] = card;

post.discardCount[currentPlayer]++;

post.numBuys--;

}

else if(ret==0 && faults > 0){

return 1; //Fault.

}

**Analysis of Testing**

**Mine:**

testBuyCard was used to successfully test my own implementation of buyCard as well as a couple other college’s implementation. Upon running the test against my own buyCard it was immediately discovered that my implementation caused a segmentation fault. I was able to pinpoint the segmentation fault to a call of updateCoins. I have not been able to find out why this was troublesome so I commented it out since the number of coins was already placed into the coins field in the structure. Other than the segmentation fault my buyCard implementation seems to be in proper functional order.

**Ellingsn:**

When I ran testBuyCard on their code every iteration failed. To investigate what might be causing this problem I discovered that their implementation is incomplete. 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 testBuyCard on their code I discovered that they had the same implementation as Ellingsn, thus the test results were identical.

**Shearini:**

When I ran testBuyCard on their code every iteration failed. After examining their code I discovered that their implementation was missing some vital functionality such as decreasing the supplyCount and buy actions remaining.

**Mcconnjo:**

When I ran testBuyCard on their code every iteration failed. Upon viewing their code I saw that their implementation is very similar to Shearini’s but with the slight variation that the unary decrement operator was used incorrectly. Like Shearini’s version, their code also failed to check if a buyCard was permitted to begin with given the amount of cards or money available to a player.

**Plans:**

For the future I plan to figure out the problem with updateCoins in my implementation and to explore whether it is necessary to update the structure with the players current amount of coins become checking to see if they have enough coins to complete a buy.

**Conclusion**

Writing the test for buyCard was entertaining and it really allowed me to see how one might implement random testing in a variety of scenarios. While testing, the bug with my implementation and updateCoins was a bit frustrating because I was having trouble figuring out why it was causing a segmentation fault. Regarding the tests on my college’s buyCard implementations, I was a little dismayed since it seemed that much of the code for their buyCard’s was not complete.