Tang, Kevin

CS362

Final Report

June 6, 2012

**Final Test Suite**

Usage:*make clean*

*make all*

Output:*test.out*

Approach: Random testing.

Focus:cardEffect(adventurer), cardEffect(sea\_hag), playCard, drawCard, buyCard, initializeGame, shuffle.

Results: Number of errors detected per test.

INTRODUCTION

There are seven areas within dominion that my test suite focuses on. Several reasons justify choosing these specific areas out of the many possibilities; some of the areas are extremely trivial and the most sensible way to test these would be to simply re-implement the functionality. Furthermore, some potential choices are dependencies of areas that I have chosen and are accounted for accordingly; therefore re-testing these particular areas would not yield any additional insight.

In this final test suite I have bundled the buyCard and adventurer cardEffect tests from the first two stages with two new tests on the sea\_hag cardEffect and playCard. Also included are refactored versions of tests on shuffle and drawCard with an alternative implementation of testInit for testing initializeGame.

IMPROVEMENTS

In test report 2 which tested the adventurer cardEffect, I encountered a technical difficulty where a memcmp of the entire gameState returned from cardEffect and of the tests version of the gameState failed which lead to just comparing each piece of the structs instead. This alternative method strangely did not fault. While implementing the new tests I was able to devise a better randomized gameState initialization which solved the problems I was having with the memcmp of the gameStates in test report 2. All my tests now include a form of this updated gameState randomization which is shown in figure 1.

Other improvements were made regarding the core dominion code. In order to properly implement a test for the cardEffect sea\_hag, I looked up the official specifications for sea hag and designed according to this. Upon testing it was discovered that the implementation of sea\_hag in the core dominion code was inaccurate. Much of the card indexing and incrementation was out of order and in excess. One of the increment instructions was largely displaced to the right of the rest of the code and on the same line as some other code, which makes me think that this was an unintentional artifact. I issued a bug report depicting this in my directory in PROJECT/bug-reports/.

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)+1;

game.numPlayers = Players;

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

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

// make valid supply counts for all type of cards in the game.

for(i = 0; i<CARD\_VAR;++i){

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

}

game.playedCardCount = 0;

memset(&game.playedCards,0,sizeof(int)\*MAX\_DECK);

game.whoseTurn = Players-1;

for(i = 0; i < Players; ++i){

game.deckCount[i] = (int)(Random() \* (MAX\_DECK-50));

game.discardCount[i] = (int)(Random() \* (MAX\_DECK-50));

//put some valid cards in the deck.

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

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

}

//put some valid cards in the discard.

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

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

}

game.handCount[i] = (int)(Random() \* (MAX\_HAND-10)) + 2;

//set the first card to be adventurer

game.hand[i][idx] = adventurer;

//fill up the hand with cards.

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

game.hand[i][c] = k[(int)(Random()\*CARD\_VAR)];

}

}

}

Figure 1

I was also able to improve my implementation of adventurer due to a bug discovered by a peer. While the bug was not documented in the bug reports directory I came across reference of it by reading their test reports. It turns out that I forgot to include code to discard the adventurer card after it was played. After fixing my implantation in the core dominion code I also improved my tests of the adventurer cardEffect to test for discard as well.

TESTING: buyCard

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

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 modify anything it shouldn’t.

This test consists of two thousand tests focused on buyCard. Each iteration generates new random states that are duplicated and one of the copies are passed through the implementation of buyCard while the other copy is modified by the testBuyCard function to what the specification states should happen when buying a card. testBuyCard first checks to see if the random state should theoretically be permitted to buy a card and flags if it can or cannot. If buyCard returns a SUCCESS status then testBuyCard checks to see if this aligns with the flag. This identifies whether buyCard should have failed or succeeded and throws a fault appropriately. By now, if all is well, testDrawCard compares the returned state to a known correct version of the state. Figure 2 shows some of the checks that are performed during iterations.

Figure 2

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

++faults;

}

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

++faults;

}

ret = buyCard(card, pre);//= if they have money and card exists.

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;}

if(memcmp(&post, pre, sizeof(struct gameState)) != 0){return 1;}

Prior to testing I ran a file comparison on my team member’s implementations of dominion.c with the original dominion.c. This allowed me to save time during testing by determining which implementations are incomplete or not attempted and excluding them from unit tests.

ellingsn:  
 This implementation has incomplete dominion code; determined with diff.

omaraa:

This implementation of buyCard resulted in 841 faults out of 2000 tests. A failure percentage of 42% seems pretty high. Upon investigation I discovered that their buyCard implementation functions properly except that it fails to change the state’s phase for the purchase. After adding this functionality their implantation passed my test. I submitted a bug report outlining the problem and possible fix.

taylodav:

Their implementation of buyCard failed the test every time. After inspection it turned out that the state’s phase was not changed and that they left the original version of buy card in there also. Furthermore, their checks for adequate coins and supply counts only checked if they were equal to zero. It is necessary and safer to check whether it is less than or equal to zero. After adding in some code to change the state and removing the original buyCard code their implementation passed. I submitted a bug report outlining the problem and possible fix.

lesliew:

Their dominion suffered a segmentation fault immediately. Under visual inspection their implementation of buyCard seemed to check for all the necessary things except for setting the state phase. I did not investigate the source of the segmentation fault.

nicolgl:

Their code does not compile. No further testing.

nguyenta:  
 This implementation has incomplete dominion code; determined with diff.

shearini:

This implementation failed every iteration. After inspection it seems that their code lacked some logical checks against supply amounts and never gives the player the purchased card. Additionally they retained the original code remnants.

mcconnjo:

This implementation does not perform any safety checks before allowing the player to buy a card. The state phase is not changed nor does the player actually obtain the purchased card. Their code is the same as the original with just two additional lines.

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. The results from the tests performed on my college’s buyCard implementation were a little surprising. I had thought that there would have been more of an improvement since my initial buyCard tests in my first report.

**Summary of buyCard tests**

|  |  |  |  |
| --- | --- | --- | --- |
| *Member* | *Faults* | *Coverage* | *Issues* |
| ellingsn | N/A | N/A | Incomplete |
| omaraa | 841 | 100% | Missing state change |
| taylodav | 2000 | 100% | Unsound Logic, Code artifacts, Missing state change |
| lesliew | N/A | N/A | Segmentation fault |
| nicolgl | N/A | N/A | Codes does not compile |
| nguyenta | N/A | N/A | Incomplete |
| shearini | 2000 | 100% | Missing logic checks, Code artifacts, No card given |
| mcconnjo | 2000 | 100% | Missing logic checks, Code artifacts, No card given |
| *mine* | *0* | *100%* |  |

TESTING: cardEffect(adventurer)

Relevant Fields:playedCards, playedCardCount, 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.
* playedCards stack properly adjusted.
* playedCardCount properly incremented.
* The adventurer card is discarded.
* If any other part of the struct was changed.

This test consists of two thousand tests focused on the cardEffect adventurer. Each iteration generates new random states that are duplicated and one of the copies are passed through the implementation of cardEffect while the other copy is modified by the testAdventurer function to what the specification states should happen when an adventurer is played. Before proceeding with comparing the results of cardEffect, testAdventurer first determines what the results should be by counting the number of treasure cards up to a maximum of two. This is limited by only permitting a single shuffle throughout this entire process. Once all this data is collected testAdventurer then compares if the game state returned by cardEffect has 0, 1, or 2 additional treasure cards. If there are no additional treasure cards then testAdventure checks to see if the handCount is one less to show that the adventurer was discarded. A memcmp is used to compare the rest of the states to ensure that nothing absurd was modified. Figure 3 exposes a snippet of testAdventurer’s vitals.

Figure 3

Figure 3

if((cardEffect(adventurer, 0,0,0,pre,0,0) != 0) || whoseTurn(pre) != currentPlayer){

return 1;

}

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

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

(pre->handCount[currentPlayer] == post->handCount[currentPlayer]) ||

(pre->handCount[currentPlayer] == post->handCount[currentPlayer] - 1)){

//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;}

ellingsn:  
 This implementation has incomplete dominion code; determined with diff.

omaraa:

This implementation of adventurer nearly failed every time. Upon investigation I discovered that their adventurer implementation drew cards from the player’s deck until two treasure cards were obtained. According to the specification of adventurer, this is not what should happen.

taylodav:

Their implementation of adventurer resulted in an infinite loop. After inspection it seems that if two treasure cards are not found in the deck it will shuffle and look some more. The infinite loop exists because you can shuffle as many times as you want to keep looking for more treasure. According to specification, adventurer only permits one shuffle if you cannot find enough treasure in the deck. I submitted a bug report that outlines the problem.

lesliew:

Their implementation of adventurer ended up with a segmentation fault. Under visual inspection it seems that it iterates through each card in the player’s deck for treasure cards but do not stop when the deck has been exhausted. I submitted a bug report that outlines the problem.

nicolgl:

Their code does not compile. No further testing.

nguyenta:  
 This implementation has incomplete dominion code; determined with diff.

shearini:

This implementation failed every iteration. Upon visual inspection it seems that their code does not discard the adventurer after finding treasure cards and does not allow shuffling of the deck when it is exhausted.

mcconnjo:

This implementation produces a segmentation fault. Under visual inspection it seems that the segmentation fault occurred because they allow an unlimited number of shuffles. Through each iteration of the player’s deck, each non-treasure card is placed into a temporary hand and if less than two treasures is found the implementation will shuffle. Without the proper checks and safety measures, this array indexing will eventually result in a segmentation fault.

CONCLUSION

Reading the results of my peer’s report 2 allowed me to revise my test for cardEffect’s adventurer. The results of this test were quite poor and once again it was somewhat surprising that many implementations resulted in segmentation faults or were not coded to specifications.

**Summary of cardEffect(adventurer) tests**

|  |  |  |  |
| --- | --- | --- | --- |
| *Member* | *Faults* | *Coverage* | *Issues* |
| ellingsn | N/A | N/A | Incomplete |
| omaraa | 1972 | 100% | Incorrect implementation |
| taylodav | 2000 | N/A | Incorrect implementation, Infinite loop |
| lesliew | N/A | N/A | Segmentation fault |
| nicolgl | N/A | N/A | Codes does not compile |
| nguyenta | N/A | N/A | Incomplete |
| shearini | 2000 | 100% | Incorrect implementation, Missing state change |
| mcconnjo | N/A | N/A | Segmentation fault |
| *mine* | *0* | *100%* |  |

TESTING: cardEffect(sea\_hag)

Relevant Fields:playedCards, playedCardCount, deckCount, hand, handCount, discard, discardCount

Checks:

* Only the top card is changed.
* Top card changed to a curse.
* Top card is discarded
* discard stack count properly adjusted.
* playedCards stack properly adjusted.
* playedCardCount properly incremented.
* The sea\_hag card is discarded.
* If any other part of the struct was changed.

This test consists of two thousand tests focused on the cardEffect sea\_hag. Each iteration generates new random states that are duplicated and one of the copies are passed through the implementation of cardEffect while the other copy is modified by the testseahag function to what the specification states should happen when a sea hag is played. Before proceeding with comparing the results of cardEffect, testseahag manually modifies the “local” version of the game state so that the top cards of every player’s deck except the current player is replaced with a curse card. Finally a memcmp is used to compare the states to ensure that nothing absurd was modified.

I visually inspected other group member’s dominion code to see if anyone else had spotted this and fixed their code so that I could test their implementations. Unfortunately this was not the case and thus I was only able to perform the test on my own code. The improved sea hag implementation is displayed in figure 4.

case sea\_hag:

for(i = 0; i < state->numPlayers; i++){

if(i != currentPlayer){

state->discard[i][state->discardCount[i]] =

state->deck[i][state->deckCount[i]-1];

state->discardCount[i]++;

state->deck[i][state->deckCount[i]-1] = 0;

}

}

discardCard(handPos, currentPlayer, state, 0);

return 0;

Figure 4

CONCLUSION

Discovering that the original sea hag implementation was incorrect and then re-implementing it was fun. I issued a bug report that outlines the problems with the current sea hag so that others may incorporate the necessary changes. Since it is unlikely that others will update their dominion code at this time, I wish that it was discovered earlier in the term.

TESTING: playCard

Relevant Fields:phase, deck, hand

Checks:

* In the correct phase.
* The card is greater than or equal to an adventurer and smaller than or equal to a treasure\_map.
* If the player has enough actions to play a card.

This test consists of two thousand tests focused on playCard. Again, each iteration generates new random states that are duplicated and one of the copies is passed to playCard and the other is checked manually by testplayCard. This test was particularly simple since the bulk of playCards functionality consists of three basic safety checks. If the player is in the correct phase, if the player’s chosen card is “playable” and if the player has enough actions to play the card. If these checks succeed the program flow is then sent to cardEffect. My code was tested and successfully passed the test. I did not bother testing the other members’ code because each of the implementations is the same since the playCard provided by the original dominion code is correct.

CLOSING

CONCLUSION

The results from running the entire test suite are disappointing. One third of the code implementations either had trouble compiling or resulted in segmentation faults somehow. A majority or implementations were incorrect or did not pass the random tests. Overall this was an enjoyable project and through it I learned a significant amount of testing knowledge. I attempted to use CMBC but realized that at this point and based on the maturity of the code to test, the effort and time to use CBMC on every team members code would not be worth the amount of knowledge gained, if any, in addition to that already gained through the random testing. Random testing is an effective and very manageable testing technique. It is particularly useful during the early stages of development because decent and quick test results can be obtained this way.

DOMINION COVERAGE STATISTICS

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Percentage Of Faults For:** | | | | | | |
| ***Member*** | ***Cov. (% of lines)*** | ***adventurer*** | ***sea\_hag*** | ***buyCard*** | ***playCard*** | ***Init*** | ***drawCard*** | ***shuffle*** |
| ellingsn | N/A | N/A | N/A | N/A | 0% | 0% | 0% | 0% |
| omaraa | 35.01% of 537 | 98.6% | 100% | 42.05% | 0% | 0% | 0% | 0% |
| taylodav | 32.23% of 546 | N/A | 100% | 100% | 0% | 0% | 0% | 0% |
| lesliew | 25.86% of 553 | N/A | 100% | N/A | 0% | 0% | 0% | 0% |
| nicolgl | N/A | N/A | N/A | N/A | 0% | 0% | 0% | 0% |
| nguyenta | N/A | N/A | N/A | N/A | 0% | 0% | 0% | 0% |
| shearini | 32.85% of 554 | 100% | 100% | 100% | 0% | 0% | 0% | 0% |
| mcconnjo | 30.84% of 535 | N/A | 100% | 100% | 0% | 0% | 0% | 0% |
| *mine* | *35.65% of 547* | *0%* | *0%* | *0%* | *0%* | *0%* | *0%* | *0%* |