Software Testing Final Report

CS 362: Software Engineering II

David Taylor

2012

Kiwi

6/6/2012

David Taylor  
CS 362: Software Engineering II  
Final Test Report

INTRODUCTION

There were 3 stages to my testing effort. My first involved Unit Tests on BuyCard, my second involved Random Tests on BuyCard, coverage testing, as well as incorporating the already written testDrawCard. Finally, the last involved doing Random Tests and using CBMC on the implementation of Adventurer, and incorporating the testShuffle functions written. I only had time to do CBMC on my own code, as it took too much time to do it on the other peoples’ in my group. I also did some Delta Debugging on my test suite to find the root of peoples’ faults. The last part, I included some testing of the Smithy card, though it is super exciting. Below is the report on these 3 stages:

PRELIMINARY DIFF RESULTS:

Below I establish what sort of changes people made to their dominion.c baseline file, so I know where there are areas for errors to occur. This is merely a summary of the results of a diff between their code and the baseline code.

My code: A diff between my code and the original showed changes between in the buyCard function implementation, between line 249 and 271. Then also I made changes to the Adventurer Card implementation, between lines 600 and 618. My total changes revolved around buyCard and ventureer, that is all.

Lesliew: This person made changes to the buyCard implementation, 254 to 279, and the Adventurer Card implementation, 605 to 654. I did notice some minor changes that were made to DrawCard, which I feel improved the implementation. There were also some minor changes to other functions, but these were to assist with the Adventurer implementation.

Ellingsn: This person made no changes to the dominion.c file. I wish I had known this before I started running tests on their code, so I would know not to test it. It wasn’t until after the results from my test came through that I realized that there was no difference.

Omaara: This person only made changes to the buyCard implementation, lines 252-282, and to the Adventurer implementation, lines 606-641. No other changes were made to the dominion.c file.

Shearini: The diff for this person’s file was quite confusing because it printed out the entire dominion.c file, meaning that some change was made to everyline, though unlikely. Looking closer, every line was off by a certain offset and certain variables were changed. However the only real implementation changes were those pertaining to buyCard. There was no changes made to the Adventurer implementation.

Nicolgl: This person made a lot of changes to their dominion.c file. Quite a lot of it was additions for DEBUG constants, which is useful when testing and debugging, but there were also some major changes made to other functions, especially those functions that were somehow involved with buyCard or Adventurer. These changes revolved around endTurn, isGameOver, and scoreFor. There were other minor changes, but looking closely at them, they seemed to have influenced the code in a positive manner.

Tangke: Looking at this person’s code after the diff, all I could see is changes made to the buyCard implementation and to Adventurer, there were no other changes made, unless it affected the buyCard or Adventurer implementation (variable definitions, constants etc.)

Mcconnjo: Most of the changes that this person made revolved around commenting out useless code and replacing it with better code, in some areas, otherwise left commented out. The main additions were made to the buyCard functionality and to the Adventurer card. No other changes were made to other functions.

Nguyenta: This person made no changes to the dominion.c file. I wish I had known this before I started running tests on their code, so I would know not to test it. It wasn’t until after the results from my test came through that I realized that there was no difference.

STAGE 1: UNIT TEST ON BUYCARD

**Approach**: My first approach to writing a test report for the first time, I took the approach of doing a type of unit testing. When testing BuyCard, I took 3 approaches of tests. Firstly I tested whether the BuyCard implementation checked whether the player had enough buys in order to make a purchase, whether the player had enough coins, and whether there is enough in the supply count in order to buy the card. So I checked corner cases, when there was 0 buys remaining, when you had infinite or 1000 buys, when you have 0 coins, or infinite, or when there is enough of the supply to buy or not enough. So basically, when the testSuite runs, it will output the attempts to buy cards and tell you if there is anything wrong with the outputs. I did not use any assert statements, nor any random testing. I merely used a bunch of memcmp to check the game state to what it should be. Also, I do have a general testing scenario, which checks to make sure that the player’s coins, numBuys and discardCount are changing more than they should between buys. I will now describe exactly what my 3 types of tests do.  
 testNumSupply: this specifically tests that the supply of the card being bought is decrementing properly, and that the right card supply is decrementing properly. Also makes sure that the right number of coins are being subtracted from the player, and that the discard count and order are equivalent to what it should be.  
 testNumCoins: This test looks at whether there is enough coins, and specifically looks at corner cases, when there are no coins and the user can still buy cards of cost 0. Testing whether the buying of high cost cards with more coins is done in the other tests.

testNumBuys: This looks at the corner cases of buying, when the player only has 1 buy and 0 buys. The other cases of buying is covered in the other tests. It checks to make sure that when the player has ran out of buys it will prevent them from buying again. It does the correct purchase for when the player has plenty and there is plenty of supply remaining.

Finally, my test Suite checks to make sure that the memory allocation of the correct game state is the same as that which is produced by the implemented dominion code. This is merely a memcmp of the correct game state with the expected and correct game state.

The test suite doesn’t explicitly test the coverage of the dominion code, but based of my implementation, and looking at the code implemented by each person’s dominion and what my own test suite code covers, I appear to reach approximately 30-35 lines of code per test run. Many of these lines are hit more than once. Below is an example of my unit testing suite:

int testNumSupply(int card, struct gameState \*dom, struct gameState \*cor, int difference){

int supply = dom->supplyCount[card];

int i = 0;

int errors = 0;

memcpy(cor, dom, sizeof(struct gameState));

supply = dom->supplyCount[card];

for(i = 0; i < supply + difference ;i++){

if(i < supply){

cor->discard[0][cor->discardCount[0]] = card;

cor->coins -= getCost(card);

cor->supplyCount[card]--;

cor->numBuys--;

cor->discardCount[0]++;

}

buyCard(card,dom);

}

if(generalBuyCardTest(card, dom, cor))

errors++;

return errors;

}

int testNumCoins(int card, struct gameState \*dom, struct gameState \*cor) {

dom->coins = 0;

memcpy(cor, dom, sizeof(struct gameState));

buyCard(card, dom);

if(getCost(card) == 0){

cor->numBuys--;

cor->discard[0][cor->discardCount[0]] = card;

cor->discardCount[0]++;

cor->supplyCount[card]--;

}

if(generalBuyCardTest(card, dom, cor))

return -1;

else return 0;

}

STAGE 2: RANDOM TEST ON BUYCARD

**Approach:** As my plan from the previous test report, I wanted to do more random testing. In this test block I did just that. I implemented a random test function, which tests the code on the 3 different unit tests I did before. I kept my unit tests when testing code just to make sure that the results were the same, as well as to extend my test suite. So first I made a routine that tests randomly to try and purchase a random card, out of all of them that are available. This will run 5000 times on the game to check to make sure that the player can buy all cards available, with 1 buy and 10 coins, enough to buy 1 card. This won’t actually buy the card, but merely test that its purchasable and that the user can in fact purchase it. It cannot confirm a correct purchase, that is what test case 2 and 3 do. The second random test is buying in particular the Adventurer card, though any could have been used, with which a player has a random number of buys and also only has 10 coins. This routine also runs 5000 times. Finally, the last routine tests to see, if you have 1 buy and a random amount of coins that you are able to only buy the cards that you have with the random amount of coins you have available. At the end of all these tests, the programme will tell you how many of the 5000 tests for each subroutine have failed. I also extended my previous plan of doing coverage testing, this time more precise. I ran gcov to see how much of the code I was covering in my tests. As I do my final report, I will try to get as high as possible. Also I did some random testing on buying a specific card, in this case the adventurer card, so I will look specifically in my next report how successful people were with implementing the effects of that card.

This below is an example of a random test case:  
printf("Purchase random card\n");  
 for (i = 0; i < 5000; i++) {  
 g.numBuys = 1;  
 g.coins = 10;  
 ret = buyCard(k[rand()%11], &g);  
 if (ret == -1) {  
 Failed++;  
 }  
 Ran++;  
 }  
 printf("%i ran, %i failed\n", Ran, Failed);

STAGE 3: RANDOM TESTING ON ADVENTURER AND MORE

**Approach:** When I went about designing the code for this card, I felt like using random testing, because it is a small function, there isn’t a lot too it, and doing thousands of tests on it would cover the functionality of what the card should do to make sure it is correctly implemented. After a person uses the adventurer card, I made sure of many things, that their deck count was correct, their discard count was correct, their hand count should have increased, and that their numactioins should have decreased. Later on I used both CBMC on my Adventurer to make sure my implementation was correct, but since it took a while to edit the code to make it work for my code, I decided to not do that for other peoples’ code. I finally did a set of delta debugging to pinpoint where errors were occurring. Here below is an example of my Adventurer test code:  
if (!(post->deckCount[p] >= 0) ||( post->deckCount[p] >= (pre.deckCount[p] - 1 + pre.discardCount[p]))){  
 printf ("DeckCount wrong by %d \n", post->deckCount[p]);  
 error+=1;  
 } else printf ("Correct Test \n");  
if (!(post->discardCount[p] >= 0) ||( post->discardCount[p] >= (pre.discardCount[p] + pre.deckCount[p] - 1))){  
 printf ("Discard is wrong \n");  
 error+=1;  
 } else printf ("Correct Test \n");  
 if ((pre.handCount[p] + 2) != (post->handCount[p])){  
 printf ("Handcount wrong\n");  
 error+=1;  
 } else {  
 printf ("Correct Test \n");  
 pre.handCount[p] += 2;  
 }

Within this testing phase I also felt the need to do more extensive coverage testing on my everyone’s code to see where I could write more code to test the other functions in the dominion.c file. This gave me some ideas of where I could go with testing other functions since the code is buggy in other areas.

COVERAGE RESULTS:

This is where I am going to list the information I got from gcov on how much of the code I covered. In the future, I will want to have test cases that test more lines of code. Originally, I was getting only about 18-20% with just with individual unit test cases, but this really helped with the random testing, and increased the percentage. I was hoping by the end I could get it to about 70-80%. I am happy though with the coverage I was able to get, though it would have been nice to have gone a bit higher. I decided to go beyond coverage testing, and more into random testing, since coverage testing doesn’t completely test if it is correct, but the results are interesting.

|  |  |
| --- | --- |
| User tested on: | Gcov coverage test %: |
| Taylodav (my code) | 60.33% |
| Lesliew | 70.0% |
| Ellignsn | 54.325% \*\* |
| Shearini | 53.25% |
| Omaara | 56.175% |
| Nicolgl | 0% - didn’t compile |
| Tangke | 0% - didn’t compile |
| Mcconnjo | 54.2% |
| nguyenta | 54.325% \*\* |

\*\* these people either dropped the class or had nothing working when I came to test their code.  
These results are for the functions that I actually tested, Adventurer and the buyCard and DrawCard.

ADVANCED COVERAGE RESULTS

One I ran the basic gcov, I decided to go deeper and find out how much each function was being covered in the testing phase. Below lists the branch coverage and function coverage of each person I tested.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Function: | My code | Lesliew | Ellignsn | Shearini | Omaara | Nicolgi \* | Tangke | Mcconnjo | ngyuyenta |
| **Compare()** | 83.33 | 83.33 | 83.33 | 83.33 | 83.33 | 0 | 83.33 | 83.33 | 83.33 |
| supplyCount() | 100 | 100 | 0 | 0 | 100 | 0 | 100 | 100 | 0 |
| fullDeckCount() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **whoseTurn()** | 100 | 100 | 100 | 0 | 100 | 0 | 100 | 100 | 100 |
| handCount() | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| numHandCards() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| isGameOver() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Scorefor() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Getwinners() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **getCost()** | 63.33 | 0 | 60 | 63.33 | 63.33 | 0 | 63.33 | 63.33 | 60 |
| discardCard() | 0 | 63.33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **gainCard()** | 61.54 | 92.31 | 0 | 0 | 61.54 | 0 | 61.54 | 61.54 | 0 |
| **updateCoins()** | 81.82 | 84.62 | 81.82 | 81.82 | 81.82 | 0 | 81.82 | 81.82 | 81.82 |
| **buyCard()** | 100 | 81.82 | 100 | 100 | 100 | 0 | 90.91 | 90.91 | 100 |
| **shuffle()** | 100 | 100 | 100 | 100 | 100 | 0 | 100 | 100 | 100 |
| **drawCard()** | 39.13 | 36 | 39.13 | 39.13 | 39.13 | 0 | 36 | 36 | 39.13 |
| endTurn() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **cardEffect()** | 10 | 8.98 | 3.46 | 0 | 0 | 0 | 9.35 | 9.35 | 3.46 |
| playCard() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| initializeGame() | 83.61 | 83.87 | 83.61 | 83.61 | 83.61 | 0 | 83.87 | 83.87 | 83.61 |
| Branches exectuted: | 69.22 | 65.82 | 58.94 | 43.32 | 51.58 | 0 | 68.88 | 68.88 | 58.94 |
| Taken at least once: | 46.78 | 47.96 | 37.28 | 36.18 | 39.42 | 0 | 47.04 | 47.04 | 37.28 |

* Did not compile when I came to do tests || **BOLD**: functions that were intentionally tested or run in my test suite

TESTING RESULTS

*Unit Tests:*

**Tests**: I first ran it on my own code to make sure my tests were correct, then I ran it on others code.  
 My code: I ran this on my own code, and noticed that my own buyCard function doesn’t actually decrement the supply properly, nor did increment the players deck count properly. Plus I noticed that it would only attempt to buy a card when the player had buys not equal to zero. So if it was hard coded that the player had less than zero buys, it would still buy it. So I changed it to buy only when it was greater than zero. I also made modifications to the buyCard so it decremented the supply count properly and the coin count properly. I also noticed that my implementation doesn’t like it when there are zero coins. I will look into that, but the reason I believe is I have a conditional that checks on when there is more than zero coins buy something, not when there is zero or more. Now that I have it working on my code, I plan to test it on other students’ code. It gives accurate results about where something is wrong.  
 lesliew: When I ran my testing code on lesliew dominion code, I noticed quite a few problems. In my first test of BuyCard Test Coins, I noticed that the count of coins was never equal to that of the test or of the expected value. When buying a card, the buying function doesn’t decrease the coin count of the player by the correct amount. Each time it was off by -1, meaning that they have more coins left over than they should from a buycard. When running my BuyCard Test Buys, I see some more problems. Not only do I again see the problem of the coin count being off, I also see the numBuys being different from the expected value. Not only this, but also the discard count for the player is not what is to be expected Finally, running my third type of test BuyCard Test Supply, the results are similar. The coin count is not decremented properly, the numBuy count and discard count are not changed correctly either. Another problem is that the supply count of the cards of each type are not decreasing properly either. The code must not be decrementing the supply of the each type of card when they are being bought. Looking closer at the implementation, I notice that they only implement buyCard on coin cards, i.e. silver, copper and gold. Therefore, they don’t take into account cards of no value, like curses, and they don’t take into account the other types of cards on the field. In order to fix this, I would recommend looking generally at buying cards, and not buying specific cards. Look specifically at each card’s value, decrement the player’s coin cost by the cost of the card, even if it is zero, decrease the buy count, increase your discard count, and decrease the supply count, and this should solve a lot of the problems that are present.

Shearini: When I tried my test suite on this person’s code, there were only very minor faults. I noticed that when I ran my BuyCard Test Coins, the only problem that occurred was the fact that the numBuys left for the player did not match what was expected, it was off by 1. The correct of coins left was correct. Whey I ran the BuyCard Test Buys, the same problem occurred, the number of buys was off by 1. Finally when I ran the BuyCard Supply Test, the same fault occurred, the numBuys of the player was off by the amount of times the attempt of a BuyCard was ran. The supply count was correct though. This implies that the problem lies in the fact that the player is not decrementing the number of buys correctly for a player. Taking a closer look at the implementation of BuyCard, I see the problem. The numBuys of the player is actually never decremented. The easiest fix would be to add one line which decremented the count of the buys for a player. Another problem I noticed as well is that the buyCard never takes into account the possibility of buying a card that has a value of 0, like a curse card. This can be changed by altering the coin check conditional from [if (state->coins < getCost(supplyPos))] to [if (state->coins <= getCost(supplyPos))]. Once these changes are made though, newer faults could occur, so I am not saying that these are all the faults present.

Ellingsn: When I ran my testSuite on this person’s code, I noticed a lot of errors. When I ran my BuyCard Test Coins test, I noticed that the player’s coin count is decremented properly. However, the player’s discard count is off by 1. When I ran my BuyCard Test Buys, I realized some more problems. The player’s coin count is not always decremented properly, and the player’s numBuys are not decremented properly. Finally, my BuyCard Test Supply came to the following conclusion. The numbuys is definitely not decremented properly, the player’s coin count is also not decremented properly. Taking a closer look at the implementation, I notice that there was no change to the original code, and so the implementation is not correct at all. An attempt by this student needs to be completed first.

*Random Tests[BuyCard]*

**Test:** I ran this on my own code, and noticed that there were still some errors in my code that my original tests weren’t catching.  
 My own code: First off I noticed that my random tests showed me a loophole in my test, mainly that it was still running when I ran out of cards to buy. Once all the cards were bought, it was failing. Not a big deal, I just made a test case that covered this. Once this was implemented, it worked great. But I didn’t want to perfectly fix my dominion.c code, so someone else could catch my mistakes. My code fails on the 1st test case 0%, second test case 12% and the third test 8%. There are very minor bugs which I hopefully fix eventually.

Lesliew: So when I tested his code, I noticed a few minor things. First off, when I ran my first instance of tests, the random test on buying every card succeeded, so there is no problem buying a card. The area where the problem arose was in the second and third test cases.In the second test case, 1722 tests of the 5000 failed, so not terrible, but not very good either. Lesliew dominion code failed 34% of the time. The third test case was slightly worse, at 40%. When I took a closer look at the code, I noticed that the player was only buying specific cards, in this case, cards with value, copper, silver or gold. So there was no way to determine whether or not the user was buying the correct card. The reason my first test case succeeded was because it was testing to see if the cards were actually possible to buy, and that they could in fact be bought, not that they were bought. My second 2 test cases actually try buying cards.

Ellingsn: Since I noticed that the code is in fact the original code, it has not been changed what so ever, I made these observations. The original code will work in some ways. Using the first set of random tests I noticed that the code will be able to buy every card in the deck. It passed all the random tests of the first test case to test buying any random card. However on the other 2 test cases, it really failed. Test case 2 failed 1717/5000 times, which is a fail rate of 34%. It really doesn’t work. Then the 3rd random test case also failed miserable, with 41% fail rate. So the original code is really broken. I understand why it was offered as one of the code segments that needed to be edited, it is broken.

Omaara: When running this person’s code, it was more successful than the other tests. The first test case, of trying to buy every card succeeded with no hiccups. It was 100% successful, meaning that their code will buy every card when run. However test case 2 and 3 were a different story. So test case 2, using a random amount of coins to buy a card, failed 32% of the time. This means that they don’t cover certain cases of buying a card. Looking closer at the code, I notice it was just an extension of the original code. I noticed that it doesn’t look at the number of buys a player has, but instead just the number of coins and the card supply. Finally, the 3rd test case also failed, but not as much, only19% of the time. I believe it failed for the same reason as the second test case. To improve this code, I would recommend changing the code so that it checks to make sure that there is the right number coins, necessary number of buys and that there is the right supply count. This should fix all the problems occurring in the code.

Shearini: This code, just like the others, works correctly with the first random test case. It passes the 1st random test case of trying to purchase any random card. However the other 2 test cases failed. For the second random test case it failed 32% of the time. There is probably something wrong with buying a card with various number of available buys. The final test case, checking to see if you can buy a card with random number of coins also failed. It failed 19% of the time. This means that it will allow you to buy a card when you have the number of coins, sometimes or it could let you buy a card even when you don’t have the right number of coins. Looking closer at the code, I made some of the following observations. The code never takes into account the number of buys the player has, nor the number of supply that are left in the pile for the specific card. This is the root problem of the code. It is failing because it doesn’t check to see if your buy count is legitimate, you could have 0 for your num buys. Nor is it checking to see if there is any of that card left. Checking these conditions should solve the problems with the random tests.

Nicolgl: when testing this person’s code, I was undable to compile their code. It was givning me errors that I didn’t want to try and solve. I attempted a couple quick fixes, but it seemed more involved and therefore this person can change their code to compile themselves.

Tangke: this person’s code also didn’t compile. I noticed that it was certain errors with their makefile accepting C99 standards. Maybe their makefile is formatted differently, but again, I didn’t want to try and change it. Also they had redefinitions of certain constants in their code which was preventing compilation. It shouldn’t take much to fix. The main problems are on line 661, 682 and 684 of their code, dominion.c, inside their implementation of Adventurer. Once this is fixed, I should be able to test their code for buycard.

Mcconnjo: This person’s code actually compiled and gave convincing results. The first random test case for buying any random card, always succeeded. This is good, you can buy any card. However the other 2 test cases failed badly. The second random test case, to see if they can buy the card with a random number of buys fails 32% of the time. They are probably not looking to see if the number of buys if enough to buy the card. Even more surprising is that the 3rd test case, testing with random number of coins failed 40% of the time. This is not very good at all. When I took a closer look at their code, I noticed a lot of minor things. It is very similar to the original code, except for a few things. It makes sure to decrement the coin count, and the buy count, but doesn’t actually check to make sure the player has enough buys or cards prior to buying. This should be something that is fixed in order to get the code working fully. Just add a couple if conditions looking at the numbuys and the numcoins and probably the supply count, and you should have working code.

Nguyenta: Since I noticed that the code is in fact the original code, it has not been changed what so ever, I made these observations. The original code will work in some ways. Using the first set of random tests I noticed that the code will be able to buy every card in the deck. It passed all the random tests of the first test case to test buying any random card. However on the other 2 test cases, it really failed. Test case 2 failed 1717/5000 times, which is a fail rate of 34%. It really doesn’t work. Then the 3rd random test case also failed miserable, with 41% fail rate. So the original code is really broken. I understand why it was offered as one of the code segments that needed to be edited, it is broken.

*Random Tests [Adventurer]:*

For this, I did a series of checks using random test cases. From here, I categorized, based on how many tests I did, how many tests passed/failed. Each category tested the basics of the game which changes when an Adventurer card was played. Below are the testing results for the people in my group.  
 My code: Firstly I would like to point out that my code wasn’t correct and that there was a minor bug intentionally placed, where I had a != statement, I put =! And this altered things. Anyway, from this I got the following results from my random test of Adventurer: 3906 failed test cases of my 2000 tests. Looking through the errors I received in the results, the error was my DeckCount is off by -1, meaning that something in my deck was changing unintentionally. When I made the above alteration, from =! To !=, and changed one of the parametres I had returned, it game me 0 failed tests. I left the bugs in though for other people to check on. I also didn’t decrease my numActions after I used the Adventurer card.

Lesliew:So when I tested this person’s code, I got 4011 errors returned from my random test cases. Looking at the errors, on occasion, the discard count and the hand count were incorrect for many tests. Also the DeckCount seemed to be off by random amounts (14, 23, 24). Looking closer at the code I notice some things. Firstly, the change in hand count was never correctly implemented. Also this only covered the case when a coin card is found in your deck, and not when you run out. Occasionally, a long loop could occur if no coins were found, in which case the deck count could continually be decreasing incorrectly. If I were to change this, I would change the while loop to only go through the deck once, if not 2 coins found, then check discard included. This would be the easy fix, though more would need to be done. Also, something key they are missing is to decrement the numActions counter at the end of using the Adventurer card. This seems to be a common mistake.

Ellingsn: Running my adventurer test suite on this persons code, it returned 4127 errors. The problem where errors were occurring most often is in the incrementation of the handcount and the Deckcount. Looking closer at the code, I can try to understand why. Looking at it, I noticed there was no implementation for Adventurer card at all.

Omaara: Running my test suite for adventurer on it, the problem is, it segmentation faulted on the first test case. Even when I tried it other times, it continued to seg fault. The reason why it could be seg faulting is because they have handcount[currentplayer]-1 which could be leading to an out of bounds array error, which is causing the seg fault. I personally feel, looking closer at the code that an out of bounds on the array for hand[] is occurring.

Shearini: Again, running my test suite on this person’s code, I notice again, that it seg faulted, when it ran. I feel there is something minor occurring in the Adventurer code that causes this. If I look closely at the code, I notice some areas where it could be improved. I think there is some problems with the handCard and numHandCards functions which is causing errors to return. Since I never used these functions, and my random test suite is designed to deal with them, I would say that is the root cause of the problem. I think before making a more in depth solution, I would analyse these 2 functions more and see if there are any problems. Also, something key they are missing is to decrement the numActions counter at the end of using the Adventurer card. This seems to be a common mistake.

Nicolgl: Again, this person’s code did not compile, ergo I couldn’t test the adventurer functionality and therefore I cannot determine whether or not it would work. Though this is not testing, looking at the code from a programmer’s perspective, I feel that it should work correctly, however I can’t prove with absolute certainty it will without actually running it. But the logic behind it seems to be correct.

Tangke: When I ran this person’s code, I noticed that there were 4762 errors on 2000 random test cases. Looking closely at the results file, I notice that the DeckCount is many times off by -1, and the Discard and HandCount are many time wrong. I feel it means that it is only satisfying a specific test case and that there is other cases it is not catching or solving. Looking closer at the code, I think I may have discovered where the problem is. The code only covers the case that the player actually has 2 treasure cards in their deck, and that there isn’t any need to shuffle in the discard pile or anything like that. If the person were to implement this functionality, I think it would work more correctly.

Mcconnjo: Unfortunately, this person’s code also seemed to seg fault, just like the others. A closer inspection of this person’s code reveals the following observations. I don’t fully understand the idea behind their do while clauses, and what the point of the temp hand is. If it is being used as a temporary hand of the cards you are dealing out of the deck to find treasure cards, then it is a good idea, but they are not adding them correctly to the discard pile at the end. Also, something key they are missing is to decrement the numActions counter at the end of using the Adventurer card. This seems to be a common mistake.

Nguyenta: Running my adventurer test suite on this persons code, it returned 4127 errors. The problem where errors were occurring most often is in the incrementation of the handcount and the Deckcount. Looking closer at the code, I can try to understand why. Looking at it, I noticed there was no implementation for Adventurer card at all.

*General Results: Test\_Smithy*

So when I created the tests for this function, it took me some time to work out and so I went above and had to look into the rules of what the Smithy card actually did. Once I wrote up a test suite to test this card, to make sure the num actions had decreased, the cards in your hand has increased by 3. That is the sole purpose of this card, easy. Well when I tested this card, I noticed that it seemed to seg fault (core dumped) on **everyone’s code**. Looking closer, I think I know why it seg faulted. Maybe it was just my implementation of the test, but I couldn’t see what was wrong with my test. This makes me believe since there is only one function being called in the Smithy case, that there is something wrong with the draw card functionality. Since I am only getting about 36% coverage on drawCard, there is probably some bug in a corner case of code in which the Smithy card is activating everytime. If I had more time, I would try and test the drawCard functionality more and determine where the problem was that is occurring the Smithy problem of seg faulting. I will say that if I manually change the code such that Smithy adds 3 hard coded cards to the hand, my tester does reveal correct results everytime, which gives me a stronger feeling that there is a bug in the drawCard function.

COMMUNICATING FAULTS:

When it came to communicating faults, I had written up a bug report text file which contained bug faults I had found on a specific person’s code. I then placed this file in a folder in the BUG REPORTS folder in the PROJECT. I would create a folder, unless it already existed, and placed the txt file inside. Since I did not let the other people in my group know about their faults until later in the term, I noticed that the same faults were reoccurring. I would say for next time, it would be a good idea to email, or use the beaver source ticket system to let people know of their faults.

When it came to people communicating with me about the faults in my code, I had a folder in the Bug-Reports folder, and I expected people to place text files in there with my problems. I know my code was not perfect, so I know there were bugs, and in my Adventurer implementation I know there was an inifinite loop, which I never changed because it only occurred in very special circumstances. I would have like to have fixed it but what is the fun with testing someone’s code if there is no fault. I also had a couple minor bugs in my buycard implementation which I left to see if people would find them. Overall, as of Sunday of week 9, I have not heard from anyone about any of the faults in my code and I did not want to dig around other people’s Bug Report folders to see if there was anything about my code.

*CBMC:*

As mentioned earlier, I did use CBMC on my own code, since it was very time consuming to try running CBMC on everyone’s code as it requires a lot of code cutting. When I ran CBMC on my own code, I ran it on my Adventurer implementation. I yielded results that showed that in cases where I had at least 2 treasures in my deck, my code was proven to be correct. However, if you needed to go into your discard to shuffle and cycle through those cards, it didn’t get proven correct. I understood that it had to do with my code not implementing the discard pile for adventurer correctly. Once I noticed where my error was, I made some modifications to it, and I was able to successfully prove that my Adventurer implementation, according to CBMC, was correct. I went further than the implementation in class by adding more assumes and asserts to prove complete functionality of the Adventurer implementation. CBMC was really useful with the assume, do and assert combinations. When it came to assuming something, it made it easy to make sure that useless test cases that random testing would have tested for, aren’t checked. This also made it random in a predictable manner. It helped with specifying what to do during the proving steps. When the doing steps ran, it narrowed down certain cases that would actually happen, not those that wouldn’t. Then the asserts basically proved whether or not the code you ran/tested is correct to the results that you should have predicted. I found it a really useful tool, and I would use it again, especially if it is on small logic C programmes.

*Delta Debugging:*

Because my previous tests on other peoples’ code was yielding many results which showed their code was wrong, I felt the need to isolate where the problems were occurring. Using Delta Debugging and my own logic in reading the code, I was able to pinpoint where the errors were in each person’s code, and that is how I determined what to write for people in their bug reports, unless it was blatantly obvious. I was able to understand where the errors or faults were, and it became a very useful tool for me. The exact results that I was able to determine from running my delta debugging script on other peoples’ code I placed in the bug report on in their respective folders in the bug reports folder. It has been sythensized to be more clear, and I add some of my understanding and logic to it so it seemed clear what sort of errors people were getting. I only did this for people whose code compiled and who actually made modifications to their code in the first place, so in total, only 5 people in my group had this done for them by me.

COMPARING MY TEST TO TESTALL.C COVERAG RESULTS:

Looking at the results for my code with that of the TestAll shows comparative results. On average the lines executed for TESTALL was 39.49% of all lines and and mine was 33% on all lines, so it is very comparable. I aim to try and improve this, hopefully get some coverage on all functions. I will not however have enough time to test this on all the people in my group. The bold is what I was able to improve the coverage to. If I had more time, I would have tried to have som coverage on all functions, and have tested it on all the people in my group, not just myself

|  |  |  |
| --- | --- | --- |
| Function: | My code | TestAll |
| Compare() | **100** | 100 |
| supplyCount() | 100 | 100 |
| fullDeckCount() | 0 | 0 |
| whoseTurn() | 100 | 100 |
| handCount() | **35** | 100 |
| numHandCards() | 0 | 0 |
| isGameOver() | 0 | 0 |
| Scorefor() | 0 | 0 |
| Getwinners() | 0 | 0 |
| getCost() | 63.33 | 83.33 |
| discardCard() | 0 | 0 |
| gainCard() | 61.54 | 69.23 |
| updateCoins() | 81.82 | 81.82 |
| buyCard() | 100 | 100 |
| shuffle() | 100 | 100 |
| drawCard() | 39.13 | 100 |
| endTurn() | 0 | 100 |
| cardEffect() | **14.8** | 10.8 |
| playCard() | **20** | 50 |
| initializeGame() | 83.61 | 83.61 |

EVALUATION:

Over this term I learnt a lot about various testing techniques, including random testing, delta debugging and CBMC to name a few. When it came to implementation I used unit tests at the beginning, because that is all I really knew. Then I found random testing to really useful and show consistent faults in code. I really liked random testing so that’s why I employed it for most of my project. I enjoyed using CBMC as well, though it was rather challenging and difficult to use, since you have to take out a lot of code from dominion.c for the proving to work correctly. I was only able to run CBMC on my own code, because it took far too long to change the dominion.c file in everyone’s folder to match what the CBMC file was meant to do. If I had more time, I would have tried to do this. I feel like I understand testing much more. If I were to do this project again, I think I would like to test more functions, more of the cardEffect functionality, and in general, more of the functions that are pivotal to a game of dominion.

So there were many reasons I did the sort of testing I did. Firstly random testing is so crazy cool and can get the job done. It helps us, especially with code like dominion that is so large and so many different permutations and combinations of cards and moves, that random testing really helps cover those cases that can’t be caught with normal unit testing. When using random testing, it makes testing dominion easier. But see the problem is getting all those small corner cases. That is why I introduced some unit tests at the beginning to get at those corner cases, those specific cases where random testing may not always get to, and if they do, you may not catch it when your process the results. Testing dominion is both hard and easy. Its difficult in the sense that there is so much variety and different cards, moves and functions that are present. However using random testing, it becomes quite easy to cover those functions and card and move combinations. The difficult part also is figuring what is correct and what isn’t, making sure you cover all the different changes to a player’s deck, discard, hand, coins count etc. This gets troublesome at times, and that is what makes the testing part fun, finding out what is correct and what your code ought to do, and what is wrong with it. I could easily apply what I learnt in testing dominion to real world examples. In the future when I come to writing code that had many test cases, writing a short test suite, though time consuming, could be quite beneficial. Also, even though I didn’t do any of this myself, I have come to understand how test based code writing can work, meaning writing your test cases before you actually write your code. I never understood it, but now I do, since you write your tests, then you code to match those tests, a bit counter-intuitive but it does make sense. Overall, I have learnt so much this term on how to write good code, and how to make tests more elegant and more tailored towards solving the problem. I have also been able to understand the power of RANDOM TESTING, and how expressive it can be. I like how random testing can cover so many test cases in such a small amount of time, and it is great in testing how likely it is your code is going to succeed or fail. Another greatly interesting tool which I would like to use more of is Delta Debugging. I really enjoy the use of python and using Delta Debugging was actually a lot of fun. Finally, I would like to use CBMC or similar tools. I like to prove things, and just showing results does not seem good enough, if I can say with certainty that my code is proven correct, that sounds so much better.

Overall this was a great term, and I felt like I learnt a lot of useful testing techniques which will help me in my future when I come to develop different software based systems. I greatly enjoyed testing and breaking code prior to this class, and this has made it that much more enjoyable, plus I have learnt great skills along the way.