Test Report 2

Kristen Bartosz

CS 362

Thursday, May 17, 2012

## Approach

I have implemented some random testing code for the buyCard function. This test makes sure that all necessary pre-conditions are met before a card is bought. This includes having enough coins, actions, and a phase change. TestBuyCard creates random game states by providing variables with random numbers. These variables include the player, the number of supply cards available, the number of coins, the number of buys, the number of cards in discount. This test also just puts in a few hard-coded values including handcount=5 and embargoTokens=0. These don't necessarily directly affect the process of buying a card. Currently, this test runs through 150 different random states to test buyCard on.

```
int checkBuyCard(int p, int who, struct gameState *post) { //p means supplypos
  struct gameState pre;
  int r;
  memcpy (&pre, post, sizeof(struct gameState));

  r = buyCard (p, post);


  if((pre.numBuys >= 1) && (supplyCount(p, &pre) >=1) && (pre.coins >= getCost(p))){
         pre.phase=1;

         pre.discard[who][ pre.discardCount[who] ] = p;
         pre.discardCount[who]++;
         pre.supplyCount[p]--;

         pre.coins = (pre.coins) - (getCost(p));
         pre.numBuys--;

         assert (r == 0);
  }

  assert(memcmp(&pre, post, sizeof(struct gameState)) == 0);

}
```

Above is the code for the function checkBuyCard. This function tests buyCard by first creating a copy of the gameState that is passed as a parameter. Then buyCard is run on the original gameState. The copied version (pre) is then run through the correct steps needed to purchase a card. At the end, memcmp is used to compare the states of the original and the copied gamestate. If the buyCard function that is being tested returns a state the same as the copied one, then the buyCard function must be correct. Any other result means that there was a difference and that there is a bug in the code.

In the past Test Report, this code I had was a little buggy, and so was my buyCard function. I ended up making a few changes to both the test and the buyCard function in dominion.c. First, I changed it so that the card goes into the discard pile instead of the hand. That was just a misunderstanding of dominion. Second, I changed it so that when a card is bought, the phase is then set equal to 1.

## Test Statistics and Coverage

When I run this test on my dominion code, 10.18% of 550 lines of dominion.c are executed. This is only a small percentage because I am only testing buyCard instead of the whole program.

```
      150:  251:int buyCard(int supplyPos, struct gameState *state) {
        -:  252:  int who;
        -:  253:  if(DEBUG){
      150:  254:    printf("Entering buyCard...\n");
        -:  255:  }
        -:  256:
        -:  257:
        -:  258:
      150:  259:  who = state->whoseTurn;
        -:  260:
      150:  261:  if(state->numBuys < 1){
       22:  262:          printf("You do not have any buys left\n");
       22:  263:          return -1;
      128:  264:  }else if(supplyCount(supplyPos, state) <1){
       13:  265:          printf("There are not any of that type of card left\n");
       13:  266:           return -1;
      115:  267:  }else if(state->coins < getCost(supplyPos)){
       29:  268:          printf("You do not have enough money to buy that. You have
%d coins.\n", state->coins);
       29:  269:           return -1;
        -:  270:  }else{
       86:  271:          state->phase=1;
        -:  272:
       86:  273:          gainCard(supplyPos, state, 0, who); //card goes in discard,
this might be wrong.. (2 means goes into hand, 0 goes into discard)
        -:  274:
       86:  275:          state->coins = (state->coins) - (getCost(supplyPos));
       86:  276:          state->numBuys--;
       86:  277:          printf("You bought card number %d for %d coins. You now have
%d buys and %d coins.\n", supplyPos, getCost(supplyPos), state->numBuys, state-
>coins);
        -:  278:  }
        -:  279:
        -:  280:
        -:  281:
        -:  282:
       86:  283:  return 0;
        -:  284:}
```

Everyline of buyCard is covered multiple times. Out of 150 random gamestates, 86 result in actual purchases. That is 57.3% of the time a card was actually bought. 15% the player had no buys left, 9% there were not any cards left to buy, and 19% the player did not have enough money to buy the card. I intend to implement a more general random test for the entire dominion code in the next test report. I'm not sure how I'm supposed to present this information in a table.

## Bugs

While I was trying to update my testing code, I was getting a seg fault and also, the states of the original and the copied after actually purchasing a card never seemed to match up. I thought this

was a bug in my testing code. After staring at it a while, I realized that my test had actually done its job and found a bug in my buyCard implementation in Dominion. In my buyCard implementation I wrote

```
state->supplyCount[supplyPos]--;
gainCard(supplyPos, state, 0, who);
```

This was what was causing the states to be unequal. I discovered that the function gainCard already decrements supplyCount[supplyPos] when it was called, so my buyCard was actually doing it twice. That is why the states of the test and the dominion code never matched up. Now my testBuyCard should be correct as well as my buyCard implementation in dominion.c.

## Group members

I ran my testBuyCard on turcottm's dominion code and it resulted in

```
assertion "memcmp(&pre, post, sizeof(struct gameState)) == 0" failed: file
"testBuyCard.c", line 58, function: checkBuyCard
```

This means that there must be a bug in this student's code. When I look at turcottm's buyCard implementation, I see

```
if (state->numBuys > 0) {
        updateCoins (who, state, -(getCost (supplyPos)));
        gainCard (supplyPos, state, 0, who);
        state->numBuys--;
    }
```

This means that a card is getting bought as long as there are enough buys, regardless of the amount of coins the player has or if there are any cards left in the pile to buy. This is a bug.

I ran my test on gibsonro's dominion code as well and it also resulted in

```
assertion "memcmp(&pre, post, sizeof(struct gameState)) == 0" failed: file
"testBuyCard.c", line 58, function: checkBuyCard
```

When I look at gibsonro's buyCard implementation I see that his buyCard goes beyond what my test tests for. He has implemented actionBonus as well, which I have not. This may be what is causing the difference between the test state and his buyCard implementation. This doesn't necessarily mean his code is wrong, it may just be that my tester code is not complete, yet.

I ran my test on luisramg's dominion code and resulted in

```
assertion "memcmp(&pre, post, sizeof(struct gameState)) == 0" failed: file
"testBuyCard.c", line 58, function: checkBuyCard
```

I am realizing that this isn't very helpful to actually figuring out what has exactly gone wrong with the code. When luisramg's buyCard function buys a card it includes

```
        gainCard(choice1, state, 0, currentPlayer);//Gain the card
        state->numBuys--;
```

This is missing the fact that coins should be decremented by however much the purchase is. This may be one reason why the test and the dominion's code don't match up.

# Bug Reports

A member of my group reported to me

```
Infinite Loop in cardEffect for Feast
Best Solution: set x = 0 after printf("None of that card left, sorry!\n")
Other Notes: Might want to check choices to see if valid, my test program sends -1 for
choice1 which is invalid
```

I haven't had a chance to look at Feast yet, but I plan on looking at this for the next testReport.

# For the Future

For the future, I plan on generating more random tests to cover all (or a higher percentage) of dominion.c. I hope to get higher coverage on more of the lines in dominion.c. I also plan to create bug reports for the people in my group to let them know what bugs I have discovered. I also plan to address the bug report that was sent to me about an infinite loop in Feast. I will try and update my dominion code accordingly.