Drake Seifert

CS362 Assignment3

Choice of Inputs

In general all inputs involve testing both characters, manipulating arguments to cover as much code as possible, and accounting for corner cases within a card or function. There of course is the risk of finding a bug but having it be the fault of another function within the tested card or function, however this is very hard to pinpoint with one hundred percent accuracy. Regardless, it's good to keep in mind that for a large amount of tests, a failed test doesn't necessarily imply that the specific function being tested is at fault.

The following are the highlights of what was tested for each card and function.

Unit Test 1; **updateCoins()**:

- Player gets number of coins based on card value (1, 2, or 3)
- Player gets (card value) * (number of that card in hand) coins
- Copper ⇒ 1, Silver ⇒ 2, Gold ⇒ 3
- Bonus points from action phase added to total

Unit Test 2; **gainCard()**:

- Should not play if supply is empty or card is not used in game
- Based on toFlag value:
    - 0: Player card added to discard array, discard count++
    - 1: Player card added to deck, deck count++
    - 2: Player card added to hand, hand count++

Unit Test 3; **drawCard()**:

- For an empty deck, discard pile is shuffled back in, discard pile count = 0
- Player Deck count decrements by one
- Player Hand count increments by one
- Other player's hand remains unchanged

Unit Test 4; **isGameOver()**:

- Game ends when no province cards or three supply piles have no cards
- Should not affect gameState

Card Test 1; **Smithy**:

- Player should receive exactly 3 cards
- 3 cards should come from player's own pile
- No state change should occur for other players
- No state change should occur to the victory card piles and supply piles

Card Test 2; **Adventurer**:

- Assuming the player has at least 2 treasure cards:
    - Player receives at least 2 cards in their hand
    - Player has at least 2 more gold than they did before
    - Other player states remain unchanged
- If the player has less than 2 treasure cards:
    - the function does not get caught in an infinite loop

Card Test 3; **Village**:

- Player has +1 card
- Player has +2 actions
- Other player state's remain unchanged

Card Test 4: **Mine**:

- Verifies player is holding a treasure card in hand
- If a player chooses:
    - Copper ⇒ gain a silver
    - Silver ⇒ gain a gold
- Player cannot receive a gold from a copper
- Player coins represent updated amount

Bugs

updateCoins()
My first unit test with updateCoins() was smooth sailing. There were no bugs that I managed to locate with my tests and overall the functionality is solid. In my next test with gainCard() however, there were multiple issues that could potentially arise if used incorrectly.

gainCard()
Essentially, gainCard() doesn't have much of a capacity to recognize invalid cards, whether the card's value is -1 or is outside the scope of the enum CARD list. Something else I found interesting was my test for every card in the game, which consistently failed the function. I am not sure why this would be the case, but it is definitely the most legitimate bug I could find within the function. If I were to guess, I believe that not every card listed within the enum CARD list is actually used, and may be messing with other aspects of the program that expect certain inputs. If this is the case then the enum CARD list should probably comment out unused cards to prevent potential confusion and future bugs. Other areas of the function concerning player card numbers and the game state seemed to work properly.

drawCard()
The drawCard() function passed all tests when the player has cards in his deck. This is good because for the most part this will be the case. However, problems began to arise when the player deck was empty. That is, the numbers didn't quite seem to add up correctly. From what I understand, the discardCard() function takes a player's card from their hand and appends it to their discard pile, which is essential to this test. If something buggy were happening within discardCard or shuffle() then that would most definitely interfere with the functionality of drawCard().

isGameOver()
A rather large bug revealed itself in isGameOver(); the game fails to end when 3 or more supply piles are empty. I found this pretty surprising seeing as there are only two ways the game can end, and this is one of them. The function also failed for 4 supply piles being empty.

Smithy
As expected, I easily revealed the bugs I had written into the smithy card function. Instead of drawing 3 cards, the player draws 5. Both the player's hand count and deck count easily reveal this. I also found a bug I didn't expect to find; when the other player draws the smithy card, the previous player's cards were affected. This shouldn't have happened seeing as I reset and reinitialized the gameState.

Adventurer
In addition to the adventurer card bugs, I found a bug within my own testing procedure that explained why I was getting false negatives from my tests for the smithy card. I was not manually changing the whoseTurn variable to be set to the current player being tested, which was throwing up all sorts of weird information. I also found the player wasn't getting enough coins from the adventurer card, which could be due to the bug I created last week. Lastly, The adventurer card will crash the game if the player does not have any treasure cards.

Village
Village worked well for the most part, which is good because the card isn't very complicated, and the actions it performs (no pun intended) are pretty black and white. The only bug I pulled from the village card is the bug I created, where the village card is never discarded. This makes it so it can be played endlessly, giving the player potentially unlimited amount of action points.

Mine
As expected, mine showed the bugs I had entered last week. Oddly enough, I misunderstood the bug I had put in last week, assuming that it allowed the player to purchase a gold with a copper card. I had gotten it backwards however, and essentially stopped the card from working completely. This revealed itself in my testing quite quickly as the card is basically worthless with my subtle addition.

Coverage

In general it seems that both my unit tests and card tests tend to cover about twenty percent or so of the lines within the dominion.c source code. That number expresses the main statement coverage for the unit tests and interestingly is relatively even for each test within my test suite. I'm assuming the statement coverage is only one fifth of the dominion code because dominion is such a large program, and there is an entire set of cards and other rules that would never be triggered from the tests I used. For the most part, twenty percent of the dominion code is used to thoroughly execute a single function or use a card multiple times.

The percentage of branch coverage varied from test to test, but in general the first four unit tests within the functions I chose had about twenty percent branch coverage, while the card tests had about twenty-five percent branch coverage. I also found this interesting, but this makes sense seeing as the cards had more specific manipulations for testing purposes. In my card tests, I often had to reset the player's hand and change various stats that could have easily caused in increase in branch coverage.

My boundary coverage tended to be about fifteen percent for both my function tests and my card tests. Similarly to statement coverage, this isn't overly surprising seeing as the dominion code is almost 1500 lines. The calls executed tended to be very similar percentage-wise to the boundary cases.

To increase these numbers, a variety of steps could be added. For example, the test suite could go further in depth by simulating a couple rounds of the game, keeping track of both players' stats and general information along the way. In this way, you will most likely cover a majority of the dominion code, and then will have additional information to look and verify its correctness. The downside to this method however is the amount of time it would take to create, as well as the excess information that would have to be accounted for.