Brandon P Kyriss

CS 362 _ Software Engineering 2

Assignment 4: Random Testing

Section I, Random Testing:

General design: We have already established that my C is not super elegant. Again, these tests are not super elegant and streamlined. In order to ensure that I was testing as many edge cases and possibilities as possible, for each test, I set up a TestState game state with several randomized elements. I added a short function randomInt that accepts a max and min int value as parameters and returns a pseudorandom integer using the system clock to seed rand. For each run through a card function, my tests choose a random player between 1 and 4 (zero-based 0 – 3), then gives that player a hand consisting of a random number (between 1 and MAX_HAND) of random cards (selected from 10 kingdom cards and the 3 treasure card types), with at least one of the card being the card that is being tested. A random number of random cards (between 0 and MAX_DECK – the count of cards in the player's hand) is then placed in the current player's discard pile and the player's deck is populated with a random number of random cards (between 0 and MAX_DECK – (hand count + discard count). This setup ensures that the player's total card count is always within the bounds of the number of game cards, even though this is really an edge case that will never occur in the actual game (no single player will ever have all of the cards).

In retrospect, I could have made the test even more random by randomizing the kingdom cards used in the game, but this would likely not reveal any useful information since the only card that gets used is the card being tested. A factor that could be made slightly more random is the hand position of the card being tested. I always add it as position 0 in currentPlayer's hand, but I could have loaded the player's hand and then randomly changed one position in the hand to the target card. Another factor that is not accurate to a real game state is that I do not keep the counts of each kingdom and treasure card accurate to what exists in the real game. I use all cards in the game, but a player could have more than 10 of any kingdom card, etc. This should not really be relevant to the test and would add unnecessary complexity to the test suite.

For each test, I save pre and post states for each attribute I wish to compare to expected values, and compare those values.

One thing I do lack after thinking about the URL validator are some test cases to test setups that should fail to validate all possible fail states.

Test 1: randomtestadventurer.c

For the adventurer test, I compare the count of treasure cards in the deck, hand, and discard pre- and post- run and also check to make sure the card is discarded (NOTE: I previously added discard to the 'clean' dominion.c to make sure that my tests behave as they are supposed to on a function that works properly). In order to cover all of the branches and statements in the adventurer cardEffect, the post deck and hand counts are checked for three possible cases:

1) The currentPlayer's preDeckTreasureCount was > 1. This is a case where the player's deck + discard contains at least 2 treasure cards to be drawn and we expect the draw to remove 2 treasure cards from the deck + discard , add 2 treasure cards to the hand, subtract one card from the hand (adventurer), and add one card to cardsPlayer (adventurer).
2) The currentPlayer's preDeckTreasureCount was 1. This is a case where the player only has one treasure card in his or her deck + discard. In this case, we expect 1 treasure card to be removed from the deck + discard, 1 treasure the be added to the hand, one card to be removed from the hand (adventurer discard) and one card to be added to cardsPlayed (adventurer).
3) The currentPlayer's preDeckTreasureCount was 0. This is the case where no treasure will be drawn. The treasure count in the deck + discard should remain the same (0), the hand count should be reduced by one for the discard of the card played, and the cardsPlayed count should increment by one.

Test 2: randomtestcard1.c

This is a random test for the Smithy card. The setup is similar to that of adventurer. For this test, I need to unsure that player is drawing up to 3 cards depending on the state of the deck and discard. This has a few possible scenarios that are checked in comparison:

1) The player's deck contains 3 or more cards. In this case, the deck count should be reduced by 3, the hand count should be preHandCount + 3 – 1 (discard smithy), and the cardsPlayed pile should increment by one.
2) The player's starting deck contains less than 3 cards. In this case, the cards will be shuffled. For this scenario, the deck + discard should decrease by the number of cards that can be drawn after shuffle. This has a few subcases depending on whether the shuffled deck has 3 or more cards. I cover all cases in my comparison (shuffled deck had 3 or more cards, 2 cards, one card, or no cards to draw). In each case, the deck+shuffle should decrease by either 3 if there are at least 3 cards available following shuffle, or by the number of cards remaining in the shuffled deck if < 3. The hand should increase by the number of cards removed from the deck or deck + discard – 1 for discard of the smithy, and the playedCards count should increase by 1.

Test 3: randomtestcard2.c

This is a random test that tests the Village card. The setup for this card, state-wise, is the same as above. For this test, I need to make sure that the player is drawing 1 card and gaining 2 actions. Checking for actions is easy. I simply make sure that 2 actions have been added numActions in the post-state. The check for drawing 1 card is modification of the smithy draw check with simpler cases because the only scenarios are that there is 1 card to draw in the deck, there are no cars in the deck, but at least one card in discard, or there the deck and discard are both empty.

This test has 100% statement and branch coverage of its respective function. When tested on my 'clean' dominion code, it catches no errors, when tested on my buggy code, it fails ~99% of the time because my bug breaks the + 1 card, so it only passes iteration where there are no cards to draw in the deck or discard.

Section II, Code Coverage:

Test 1: randomtestadventurer.c

This test has 100% statement and branch coverage. When tested on my 'clean' dominion code, it catches no errors, when tested on my buggy code, it fails ~98+ percent of the 100000 iterations because the bug I introduced prevents the player from drawing treasure. The only instances it passes are states where there are no treasures to draw, which have the same outcome as having no treasure in the deck/discard.

Test 2: randomtestcard1.c

This test has 100% statement and branch coverage of its respective function. When tested on my 'clean' dominion code, it catches no errors, when tested on my buggy code, it fails the majority of the iterations (99+%), but passes in rare cases where there are no cards in the deck or discard to be drawn since my bug prevents the player from drawing any cards and this state leads to no cards being drawn.

Test 3: randomtestcard2.c

This test has 100% statement and branch coverage of its respective function. When tested on my 'clean' dominion code, it catches no errors, when tested on my buggy code, it fails the majority of the iterations (99+%), but passes in rare cases where there are no cards in the deck or discard to be drawn since my bug prevents the player from drawing any cards and this state leads to no cards being drawn.

Each of these tests takes less than a minute to run on FLIP. I managed to cover 100% of the code for each function without making modifications because I planned out all of the branches on paper before writing out the code, so I specifically designed the tests to cover all statements.

Section III, Unit vs Random:

The coverage for my unit and random testing was the same as far as statement and branch coverage go because I specifically designed tests to cover all statements/branches in the functions by analyzing the possible scenarios that might occur (no cards to draw, etc.). My random tests are likely better at detecting faults because I test more possible places where the program could break. The unit tests only tested a few very specific scenarios designed to look at a single case of input. The random tests test all edge cases and will catch scenarios that I did not explicitly need to test for in my unit tests to get 100% coverage. An example of this is that my unit test for smithy simply tests whether 3 specific cards are drawn from a deck that has 5 total card and whether or not the smithy gets correctly discarded. This test covers all statements in the smithy effect code, but it doesn't look at all possible card counts for the deck and discard. Only my random test for smithy tests cases where the player can only 0, 1, or 2 cards to see if there are faults in this process. The same is true of the other tests.

Overall, the random tests should be much better at catching faults because the unit tests have very narrow scope and do not test all edge cases.