Aaron Berns
Assignment 4

Random Testing

randomtestadventurer.c tests the implementation of the Adventurer card. The write the test I started by studying the testDrawCard.c file. I followed the same format in that I set up a partially randomized game state a large number of times inside of a for loop in main() and then, inside the same for loop, passed each state in turn to a testing function that contained assert statements to check that the each state was changed in the expected ways and was not changed in unexpected ways. The trickiest part of this was figuring out which parts of the state could be completely random and which could be randomized within a limited range. I looked at the portions of the state that the Adventurer card accesses and changes and tried to use as large a range as possible for each. Certain things like the deckCount and handCount had to have a minimum in order for the Adventurer card to be able to draw cards from the deck until two treasure cards were drawn. I set the deck to a minimum of 10 and a maximum of MAX_DECK. I set the hand to a minimum of 5 and a maximum of MAX_HAND. It also turned out to be necessary that there be at least 2 treasure cards in the deck for the Adventurer card to work so I ran a loop that picked two random indexes within the deck and set them to copper. I used three assertions to check that the Adventurer implementation worked as expected. The logic behind them was the state after the card is 'played' will be changed in a few ways. First, the last two cards in the player's hand should be treasure cards, so I checked for that. The second change in state should be that the deckCount should be changed in that it should reflect the two treasure cards added to the hand and that any cards drawn that were not treasure cards should be added to the discardCount. So the post state deckCount should be less than the pre state deckCount by a total of the two treasure cards plus the growth in the discardCount. After all of this I was having the assertion that the second to last card in the post state hand be a treasure card fail randomly. It was really hard to find out why this was happening so I hard coded each state variable and then randomized each one at a time until I found that the discardCount has to be initialized to 0 or there will always be a possibility of an error. That was the only variable that I was unable to randomize.

randomtestcard1.c tests the implementation of the Smithy card. Two things main things need to happen when this card is played: three cards should be drawn from the player's deck and the smithy card should be discarded by a call to the discardCard() function. The test asserts that these conditions are met while introducing as much randomness as possible. The player's handCount of the post call state is checked to see if it has increased by two cards: the three drawn minus the Smithy card. The player's deckCount is checked to see if it has decreased by three. Finally the index where the Smithy card resided in the player's hand before the call is checked to see that it contains something other than a Smithy card. Randomness was introduced by filling the gameState with random values. The number of players was randomized in the one to four range, the deckCount between MAX_DECK and 10 to ensure that the ability to draw new cards was unaffected. The discardCount and playCardCount were both set to a number between zero and MAX_DECK because their values are used to index into arrays, so they need to be positive and within bounds. The handCount had to be one or greater so that the Smithy card is in the hand to be discarded. I randomized the index of the Smithy card within the hand and used that index to pass to the handPos argument of the callSmithy() function. 20,000 iterations provided ample coverage as the function is relatively simple. The only difficulty came from making sure that the state variables were such that the call to discardCard() within the Smithy implementation wouldn't cause a segmentation fault by using a negative index when accessing the playedCard array.

randomtestcard2.c tests the implementation of the Steward card. For solid testing, this implementation needs to have all three routes of its main if/ if else/ else repeatedly followed. The branching is triggered by the value in the choice1 variable, which can be any integer value but anything other than 1 or 2 will lead to the third branch. If the value of the variable is not controlled, the third branch will get much more coverage than the other two, so to keep it relatively even, I passed a random value between 1 and 3 when testing. The first branch leads to drawing two cards, so the deck has to have at least that number. I randomized the range of the deckCount to between 2 and MAX_DECK. An assert statement tests that the deckCount has been decreased by 2 if choice1 is equal to 1. If choice1 is equal to 2, the coin count is increased by 2. The initial value of state->coin can be any integer because any integer can be increased by 2, so it remained completely random. Asserting that the coin count has increased by 2 tests this portion. The third branch calls for two cards to be discarded and trashed from the deck, the index of the cards being passed to choice2 and choice3. This meant that the indexes needed to be real and not the same. They also can't be the index of the steward card being played because it is discarded at the end. The index of the steward card was randomly assigned from the hand indexes based on handCount, which was randomly assigned from 3 to MAX_DECK so that a minimum of 3 cards would be available to discard, one of them being the steward card. Once the steward card was assigned, the choice2 card was assigned from the hand indexes and checked to make sure it was not the steward card. choice3 was then assigned from the hand indexes and checked to make sure it was not the steward or choice 2 card. Having choice1 equal to 3 led to an assertion that three cards were discarded and of those, only the steward card increased the playedCardCount as the other two should have been trashed. This assertion allowed for the bug I introduced, which kept the other two cards from being trashed to be caught. Finally the pre and post hand arrays are scanned for steward cards and the number found kept track of. An assertion checks to see that the number of steward cards has decreased by one, allowing for the possibility that a player has more than one. This final assertion catches the fact that sometimes, when choice1 is 3 and two other cards at discarded, the code swaps index of the last card in the hand with that of the discarded card. If the last card is the steward card, then the handPos index will no longer contain the steward card and as a result the steward card will remain in the deck and different card will be discarded as a result. This is an actual bug in the code that was also caught in the unit testing.

Code Coverage

I was able to get 100% statement and 100% branch coverage for the Adventurer card implementation tested in *randomtestadventurer.c*. I was only able to do this by running a very large number of iterations. Anything below the hundred thousand range led to the possibility that one branch in particular would not be covered: the condition that the deckCount reaches 0 and needs to be shuffled. This condition didn't happen often because the randomizing of the ;deckCount led to some large decks that didn't run out. I remembered from a lecture that in cases like this, it's good to just increase the number of test iterations to increase the likelihood that a condition will arise. This turned out to be the somewhat be the case here although out of 1,000,000 iterations I was unable to create this condition. I ended up adding a second set of iterations in which the deckCount was hardcoded to 0 and the discardCount was larger than 10 with at least two treasure cards.

I was able to get 100% statement and 100% branch coverage for the Smithy card implementation with relatively few iterations. The implementation has less branching than the adventurer card so there are less possible routes to cover. The real complexity of the card is the call to discardCard(). The Smithy card is never trashed so incidental coverage of the discardCard() function is incomplete. Some additional tests could lead to more complete

coverage of this function but I'm assuming that completely covering functions called within the target function is not a requirement.

I was also able to get 100% statement and 100% branch coverage for the Steward card while maintaining a high degree of randomness in the values passed to the function. Each branch was visited a large number of times so I would say that the testing was pretty solid.

Unit vs. Random

For the Adventurer card implementation, the unit tests and the random testing both did a good job of getting coverage and of finding the bug I introduced. The unit tests were able to get 100% statement and branch coverage by using carefully a few carefully constructed game states. Thus the code was covered but was only really run a few times. The random testing achieved the same percentage of coverage but additionally ran the code several thousand times. When comparing the thoroughness of the two approaches, in this case the random testing presented many more opportunities and situations for the code to break than the unit tests so I would say that it was much more thorough and I would feel much more confident saying that the code works properly after doing the random testing. In addition to finding the bug I introduced, just like the unit testing did, the random testing made clear that certain conditions will break the code. An example of this is that for the Adventurer card to work, there have to be at least two treasure cards in either the deck or the discard pile. This made the connection between the two piles more apparent than during the unit testing.

For the Smithy card implementation, the unit tests and the random testing again both did a good job of getting coverage and of finding the bug I introduced. The coverage percentage was the same at 100% but again the random testing introduced many more combinations of values than the unit testing so I feel that passing the random testing provides much more confidence in the correctness of the implementation. The bug was found by failing the assertion that the deckCount should decrease by three. The bug causes four cards to be drawn instead of three.

When testing the Steward card, which I also wrote unit tests for, both types of testing generated 100% coverage, however the random testing was able to find the bug I introduced whereas the unit testing didn't. I attribute this to having a larger number and of test cases and variety of test values, which generated situations where not trashing the two chosen cards was caught when comparing the post state to the pre state. Another win for random testing.