Eddie Christopher Fox III

February 19, 2017

Assignment 4 Randomized Test generator development

Usage information:

Type in make randomtestadventurer to make the executable

Use the executable using randomtestadventurer <seed> <number of tests to run>

I managed to get 100% branch and statement coverage with 20,000 tests, which took less than half a minute to run.

Finally, type make randomtestadventurer.out

I separated the statements for making the random test executable and the out because the results were getting messed with.

I did not finish random testing of 2 other cards, but at least I gave good documentation here of the first one.

Contains the development of your random testers, including improvements in coverage and efforts to check the correctness of your specification by breaking the code.

When developing my random testers, I decided I would do adventurer (as was mandatory) and two other cards that I had written unit tests for. This way, I would have a base from the unit tested code and have a base with which to improve / iterated on with the random methodology. I originally decided to set it up so that there would be three arguments when calling the automated tests. <executable name> <seed to randomize testing with> <number of tests to perform>. Each randomized card test has a main function to handle arguments and randomized seeding, and then a void test function that handles the actual testing. Eventually, I realized to perform multiple tests, I could just put the testing function in a for loop and call it multiple times instead of passing a number of tests parameter to the testing function itself.

The previous method of constructing the game using a game state struct and a variable of integer to hold the 10 kingdoms that I used in the unit tests was sufficient, so I decided to use it again. I decided to give the game a random seed (as there wouldn't be any interesting results if the game always played out with the same randomization), and set a number of players between 2 and 4. I made the deck size a fluctuating value from 0 to 20, but left an option for people to configure it if they wished. Originally I considered making the deck contain between 1 and 20 cards, but I realized I should test cases where there is no cards in the deck to see what happens. In general, I made sure to print important variables on each pass for debugging purposes.

For adventurer specifically, I created a discardCount that had the same range as the deck size, but +1, in order to test counterintuitive cases where the discard number exceeds the number of cards in the deck. I set every card in the discard pile and deck to copper besides the last card in the deck, which had a random kingdom card. Based on the deck size, I would discard all coppers if

the deck was sufficiently small in order to introduce extra cases. My testing of adventurer would test discard randomly, and test random deck sizes, discard amounts before playing adventurer, and the effects of playing adventurer with a randomized last card. I printed out relevant variables such as deck count and card count after playing adventurer as well for comparison purposes, and to confirm that the variables had changed appropriately. Originally I had no use for the assertrue function, and just put it at the beginning jus tin case I would need it, but later I used it to asser the proper execution of the adventurer function, and to make sure the number of cards in the hand were appropriate.

Make sure to discuss how much of adventurer and the other cards' code you managed to cover.

According to randomtestadventurer.out, my coverage was as follows for the playAdventurer() function, which is the code that is called when the adventurer card is played.

Lines executed:100.00% of 17

Branches executed:100.00% of 12

Taken at least once:83.33% of 12

Calls executed:100.00% of 2

I managed to achieve 100% state and branch coverage, with 83.33% taken at least once.

The updateCoins() function, which I called in the test case, had high but not 100% coverage, which is understandable, as covering that function was not the focus of this unit test.

Was there code you failed to cover? Why?

While I had 100% line and branch coverage, the fact that 83.33% of branches taken at least once suggests there was about 17% of branch options that were not taken, despite every branch with options being covered. I probably need to look more carefully at my conditionals.

For at least one card, make your tester achieve 100% statement and branch coverage, and document this (and how long the test has to run to achieve this level of coverage). It should not take more than five minutes to achieve the coverage goal (on a reasonable machine, e.g. flip or flop).

See coverage information and usage information above.

Compare your coverage to that for your unit tests, and discuss how the tests differ in ability to detect faults. Which tests had higher coverage – unit or random? Which tests had better fault detection capability?

It’s hard to exactly compare the coverage of the randomized test functions written to the unit tests, because of certain conditions. The unit tests printed their coverage to an out file, but I think the statement and branch coverage reported, along with calls executed, was cumulative among all the card tests. Perhaps it was just coincidence, but every card test increased in coverage in every metric from the previous one. Perhaps I had gotten better at writing card tests in a short time, but I doubt it, as I designed them in mostly the same way in every test for the sake of simplicity.

With these disclaimers in mind, we can directly compare the effectiveness of the testing. My first card unit test achieved 29.64% line execution, 31.25% branch execution, 22.36% branches taken at least once, and 16.13% calls executed. By the last card, this had increased to 33.75% line execution, 35.58% branch execution, 26.20% branches taken at least once, and 21.53% of calls executed.

In comparison, my randomized tests achieved 100% statement and branch coverage. While the randomized tests had much better coverage, I would have to say that my unit tests had better fault detection. Quickly glancing at the test I crafted for adventurer, despite covering most of the function, every test said passed. Now it’s possible tests only fail occasionally, but because my code was bugged, it should have failed more often. I think this is because I didn’t use enough trueassert statement to test the various variables. I only used trueassert once, so even if things didn’t work according to plan and showed up in the variables, I would have had to manually compare the values to the expected values on each test (infeasible with the large number of tests I did), instead of seeing a blatant pass or failure from trueassert.

This is not to say that random testing has better coverage but worse fault detection. If I had more time to spend crafting the conditionals and assert statements, I’m sure I could have picked up much more faults with the function.