Class: CS-362
Term: Summer 2017
Author: Jon-Eric Cook
Date: July 16, 2017
Assignment: #3

Bugs

The test suit was run against the following functions: adventurerCard, smithyCard, villageCard, council_roomCard, updateCoins, isGameOver, shuffle and gainCard. Upon executing the test suit, bugs were found in the following functions: adventurerCard, smithyCard, villageCard and council_roomCard. There were no bugs found in the following functions: updateCoins, isGameOver, shuffle or gainCard.

The adventurerCard bugs were as follows: the wrong number of cards were added to the current player's hand, the wrong number of cards were drawn from the current player's deck, the wrong number of treasure cards were added to the current player's hand, the wrong number of treasure cards were drawn from the current players deck, the adventurer card itself was not discarded after it was played.

The smithyCard bugs were as follows: the wrong number of cards were added to the current player's hand, the wrong number of cards were drawn from the current player's deck.

The council_roomCard bugs were as follows: the wrong number of cards were added to the current player's hand, the wrong number of cards were drawn from the current player's deck.

The villageCard bug was as follows: the wrong number of actions were added to the state.

Unit Testing

gcov was used to measure code coverage for all the tests in the test suit and the findings were quite interesting and informative. It is important to note that the contents of the dominion.c.gcov file was sent to the unittestresults.out file after each unit test was run. This was done to see how each unit test changed the number of times a function was run and or how many times a certain line of code was executed. It is also important to note that the goal for each unit test was to explore each path through every tested function. The mindset behind this was as follows: by executing every line of code in the target function, due to exploring each path, it could be said that the unit test would be successful, have solid coverage and be best suited for finding bugs.

Starting off, the code coverage percentage for updateCoins and dominion.c was 89.80% and 17.08% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called 8 times. This number was expected as the function was indeed called 8 times. The code coverage percentage of 17.08% was based on the entire dominion.c file. Upon further review of the updateCoins function itself, the unit test was able to reach every line code at least 5 times. The unit test was designed to enter into the for loop as well as each if statement there after.

Second up, the code coverage percentage for isGameOver and dominion.c was 84.85% and 18.92% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function

was called 3 times. This number was expected as the function was indeed called 3 times. The code coverage percentage of 18.92% was based on the entire dominion.c file. Upon further review of the isGameOver function itself, the unit test was able to reach every line code at least once. The unit test was designed to enter into the first if statement, the for loop, the if statement in the for loop and the last if statement.

Third up, the code coverage percentage for shuffle and dominion.c was 82.35% and 19.08% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called 11 times. This number was expected as the function was indeed called 11 times. The code coverage percentage of 19.08% was based on the entire dominion.c file. Upon further review of the shuffle function itself, the unit test was able to reach every line code at least once. The unit test was designed to enter into the first if statement, the while loop and the for loop at the end of the function.

Fourth up, the code coverage percentage for gainCard and dominion.c was 86.49% and 21.54% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called 5 times. This number was expected as the function was indeed called 5 times. The code coverage percentage of 21.54% was based on the entire dominion.c file. Upon further review of the gainCard function itself, the unit test was able to reach every line code at least once. The unit test was designed to enter into all four if statements.

Fifth up, the code coverage percentage for adventurerCard and dominion.c was 98.36% and 24.46% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called once. This number was expected as the function was indeed called only one time. The code coverage percentage of 24.46% was based on the entire dominion.c file. Upon further review of the adventurerCard function itself, the unit test was not able to reach every line code at least once. The unit test did not explore the first if statement in the first while loop. As a result, the deck was never checked if it was empty and needed shuffling. Also, the unit test did not explore the else statement of the second if statement in the first while loop. As a result of this, drawing a non treasure card as never faced and dealt with. Lastly, the last while loop was not explored. As a result of this, the non treasure cards that may have been drawn were never dealt with. Sadly, the unit test was designed to enter into all while loops and if statements but it did not.

Sixth up, the code coverage percentage for smithyCard and dominion.c was 97.87% and 27.23% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called once. This number was expected as the function was indeed called only one time. The code coverage percentage of 27.23% was based on the entire dominion.c file. Upon further review of the smithyCard function itself, the unit test was able to reach every line code at least once. The unit test was designed to enter into the for loop and execute the discardCard function.

Seventh up, the code coverage percentage for council_roomCard and dominion.c was 97.92% and 29.23% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called once. This number was expected as the function was indeed called only one time. The code coverage percentage of 29.23% was based on the entire dominion.c file. Upon further review of the council_roomCard function itself, the unit test was able to reach every line code at least once. The unit test was designed to enter into both for loops, increase the number of buys and execute the discardCard function.

Eighth up, the code coverage percentage for council_roomCard and dominion.c was 97.96% and 30.15% respectively. Upon review of the dominion.c.gcov file, it was noticed that the function was called once. This number was expected as the function was indeed called only one time. The code coverage percentage of 30.15% was based on the entire dominion.c file.

Upon further review of the council_roomCard function itself, the unit test was able to reach every line code at least once. The unit test was designed to execute the drawCard function, increase the number of actions and execute the discardCard function.

As a whole, unit testing was quite successful. The only problematic unit test was the one belonging to the adventurerCard function. Based on the results from said unit test, improvements need to be made so as to achieve a higher coverage percentage in hopes of finding potential bugs.

On the topic of edge and boundary cases, each were assessed on a case by case basis. For each function, be it a generic dominion function or a card function, effort was put forth to determine what were in fact the edge and boundary cases. A lot of times the function itself took these edges cases into account and had if statements to catch them. For example, in the gainCard function, the first block of code is an if statement that checks if the supply pile is empty. If the supply pile is indeed empty, different actions needed to transpire for the game to continue. As a result, the gainCard returned -1. Seeing how some functions had these blocks of code to catch the edges cases, test code was then written to be sure to enter into those said blocks of code. This technique was employed for the other functions in an attempt to address the edge and boundary cases.

Unit Testing Efforts

To build the test suit, a large portion of the effort was actually put forth to truly understand how each function affected the state the of dominion game. Once a solid baseline was determined, the next step was the construction of an assertTrue function that would be used to test if a certain test passed or failed. With the assertTrue function completed and a sturdy idea of how the game worked and changed, the tests for each function were written.

When writing the tests for each function, there was a little bit of a conflict on what to actually test. Are tests suppose to be written to test if the functions handle wrong inputs? For example, all the functions require a state variable. Are test suppose to be written to see how each function acts when it is given a char instead of a state variable? Are test suppose to be written to try and break the game? A lot could have been done, almost too much.

Once it was decided to really only test the code that was in each function, things got a little easier. The code for each function was reviewed and studied and from that, it was determined what state, conditions and or variables had to be in place to execute each line of code in a given function. Adequate setup was done for each function so as to touch each line of code and test its actions. When a certain function altered the state of the game, a comparison between an unaltered state and a normal one was done to confirm that the correct actions were taken by the function. This was the main technique for testing the eight functions.