Dominion Test Report

Wenbo Hou
CS362 SP16
Alex Groce
6/5/16

I. Introduction

In this term, we debugged the source code of a deck-building card game, Dominion. The source code (dominion.c) contains necessary game actions including, initializing the game, performing cards' effects, drawing or discarding cards, buying cards, and some other helper functions. For more information about Dominion, please go to Dominion Strategy Wiki.

To debug the source code efficiently and accurately, we debugged functions in the source code individually. Then, we test different action cards' effects with both the unit test method and the random test method. Finally, we created a random test generator to simulate the whole game. Furthermore, I also evaluated our testers' efficiency through mutation testing.

II. Unit Tests

The target functions I chose were DrawCard (), isGameOver (), getCost (), and updateCoins (). These functions were basic parts of the game's architecture, which make it important to write it correctly. To debug these functions, I initialized the game object based on each function's content in each unit test. For example, I created three different game objects with different ending situations for the function isGameOver (). Those ending situations corresponded to winning cases in the function isGameOver ().

After executing all four unit tests, I found that the code coverage of dominion.c was 16.40%. Then, I checked the dominion.c. gcov for the code coverage of tested functions. All tested functions, except DrawCard (), got over 90% code coverage. When I reviewed my unit tests at the end of this term, I noticed the low code coverage of the function DrawCard (). The cause of this strange result was that I only considered one case in DrawCard (). There were two cases in DrawCard (): drawing hand to an empty hand and drawing hand to a non-empty hand. Unfortunately, I ignored the second instance when I wrote the unit test for DrawCard ().

The target action cards I chose were Smithy, Village, Steward, and Salvager. Test strategies of action cards were similar to that of functions. I customized player's hand card, deck, and the discard card holder to meet the basic requirement of performing the tested action card's effect. For action card that required the player's choice, I manually chose which action to perform. For instance, the action card Steward required the player make a choice among three actions: drawing two more cards, getting a two-coin bonus, and trashing two cards. I tested Steward three times with three options.

I get several wrong results after executing card tests. So, I went to the cardEffect () function in Dominion.c to check the source code. I found that the index of the played card sometimes changed due to the card effect. Consequently, the discardCard () function could not discard the played card or discarded a wrong card. Then, I checked code coverage of dominion.c with the gcov tool to make sure all possible cases covered. The result was satisfying, and all branches of test cards' effect were tested.

The unit test is qualified enough to debug single function for a large program. When the programmer finishes the general architecture of the program and starts to test the program's functionalities, unit tests is not effective to find bugs. When writing unit tests, programmers test the game subjectively. Thus, they can ignore some details and miss some bugs. So, developers introduce a new testing method, Random Test.

III. Random Card Tests

Random tests require the programmer to write a program to generate all non-tested attributes randomly to simulate test situations as many as possible. When writing random test generators for actions cards, I randomly customized the player's hand cards, deck, discarded cards, and action choices. Cards I chose to test were Adventurer, Steward, and Smithy.

After running random tests for 50 times, I checked the test results and the function coverage of dominion.c separately. I only found bugs in cardEffect ()'s adventurer section. My random tester cannot get the correct hand count and discarded card count. Then, I checked the function coverage for each random card testers. The Adventurer random tester got 10.53% executed line coverage and 20.23% branches coverage for function cardEffect (), which was not conclusive. I then checked the dominion.c.gcov to examine the executed line in the function cardEffect (). I found that all codes about adventurer card were executed. This fact showed that my random tester for adventurer card was effective, so my random tester for Steward did. The function coverages for function cardEffect () in dominion.c after I have executed random tests for card Steward were: 8.13% executed lines and 15.61% executed branches. However, dominion.c.gov showed that I tested all cases for Steward's card effect. The function coverage of Smithy was straightforward because the card effect of Smithy is an independent function. The function coverage I got was 100% for both executed lines and executed branches.

IV. Random Tests for the Whole Game

After random tests for partial functionalities, I started to write a random tester to simulate the full game play. I wrote a program called testdominion.c to play the dominion game for 50 times automatically. In testdominion.c, all external variables including the number of players in this game, which card to play, and action choices were created randomly. This tester would show the finial winner onto the screen.

After executing the random test generator, I got 80.32% code coverage for dominion.c. This code coverage showed that most functionalities of dominion.c were tested. In my case, I could not guarantee the dominion.c works well because I only debugged a small part of the whole game. If a programmer finished debugging all functions and partial functionalities, over 80% code coverage of the whole program was a strong proof that showed the program does not have any bug.

V. Mutation Test to Evaluate Random Card Tests

To ensure the efficiencies of my tests, I applied mutation testing to evaluate my testers. First mutation testing requires programmers to rewrite the source code in small ways. Second, the programmers try their tester on the mutants. If the tester detects bugs in one mutant, the mutant is killed. A good tester should be able to kill all reasonable mutants. If mutants are alive for certain testers, the programmer should improve their tester to kill it. I did mutation testing for my three random card testers. For each tester, I created five mutants including value mutations, Decision Mutations, and Statement Mutations.

Random test generators for Smithy killed all mutants, but other two only killed two or three mutants. The test generator for Adventurer was failed to check shuffling card, removing and discarding non-treasure cards from the player’s hand. This fact showed that I should test more carefully and focus on more details. The random test generator for Steward was failed to detect wrong input in the discard function because of the unchanged player index. To fix this, I randomly choose which player to play the tested card.

Mutation Tests helped me to improve my random tests’ performances to find more bugs. For industrial level software development, I think that the mutation testing is an indispensable part of debugging because it can detect ignored details in software testers. Through the mutation test, software companies can make better products and save efforts on patching bugs in the future.

VI. Evaluations of Group Members (Jiawei Liu & Zhi Jiang)' dominion.c

The evaluations of group members' dominion.c have two parts: Face to Face discussion and actual tests on their code. We held a group meeting to talk about the test report and ideas about testing dominion.c on Monday.

According to Zhi's previous test reports and his statements, I knew that he did a great job on debugging dominion.c and detected several bugs. I also applied all my tests on his dominion.c and found two more bugs and reported bugs to him. If he fixed bugs in his dominion.c, his source code could be trustworthy.

Another dominion.c I evaluated was Jiawei's. He stated that He found two tiny bugs when doing unit tests. I tried all my tests on his source code and found one more bug. Based on my experience, his code was good if he fixed that bug.

We also discussed some architecture-level drawbacks of the dominion.c. We all found that the discardCard () functions sometimes went wrong if the player's hand card changed. We also noticed that there is a missed discardCard () in case adventurer of cardEffect (). If we fixed these bugs in our source codes, the game would perform flawlessly.

VII. Conclusion

The most valuable thing I learned from this term's work is that systematic debugging and timely documentations are important for software developments, even small programs. When I debugged the dominion.c, I noticed that the author provided lots of comments on the source code and APIs for necessary data. This fact saved my time on digging the inner structure of the game, which make the debugging process easier and more efficient. With the course's instructions, we debugged dominion.c from separate functions to necessary functionalities, to the whole game performance. This systematic debugging strategy makes debugging jobs easy to finish and highly orderly. Debugging works in this term were also highly related, which helped me to track my job continually. I also learned that the code coverage and Agans' rules were good references for debugging works. The mutation test is another exciting feature in this term. It inspired me on how to improve tests to make the program work better.