Random Testing and Generated Tests

Gabriel Jonas

CS362, Winter 2017

I. Introduction

The goal of this project was to create a better understanding of random testers. Using the previously built code of the Dominion board game in Java, the test suite implemented an additional testing generator program and a handmade random testing process. Included is an explanation of the coverage tool, the random tester, leftover bugs, and a summary of information.

II. Coverage Tool

The tool that was chosen for this assignment was Evosuite. The tool worked well for a random generator, but eventually did not help find any bugs in the Dominion code. The random generation tool covers a small fraction of the branches in tests with random choices (such as card), and while it covers most top-level API it does not go thoroughly through each input. The report of code coverage is given below:

CLASS	PlayDominion	GameState	Player	Card
INSTRUCTION_MISSED	56	131	13	22
INSTRUCTION_COVERED	32	459	669	1085
BRANCH_MISSED	2	12	4	13
BRANCH_COVERED	0	42	38	99
LINE_MISSED	10	23	6	6
LINE_COVERED	8	93	127	216
COMPLEXITY_MISSED	1	8	3	11
COMPLEXITY_COVERED	2	30	32	65
METHOD_MISSED	0	0	0	0
METHOD_COVERED	2	11	14	14

The time it takes to generate these tests is fairly small (with a given seed it takes 10 minutes to generate tests for 5 classes), and generates around 50% coverage. While this is a good start for tests, it does not make an adequate testing suite.

One of the difficulties of working with this tool is that it mainly relies on the API to generate tests. If something is consistently wrong with the code, then the tests generated will think that is part of the actual program and not catch the bug. The other portion that makes this difficult to work with is attempting to integrate this with maven (there were several errors at the beginning which required research to fix).

As an overall impression the tool is a good starting spot, but for more accurate coverage it is better to create separate unit tests that specifically target portions of the code.

III. Random Tester

The random tester operates on a simple process, given in pseudocode below:

- Initialize game with 2-4 players
- While game is not over
  - For each player in the game
    - For each card the player is able to play
      - Update state tracking
      - Compare the correct output state with the actual state
    - Test player treasure phase
    - Update state tracking
    - Test player buy phase
    - Update state tracking
    - Test player end phase
    - Test game state

The code coverage increase with the random test is roughly 4%, which can be attributed to being another random process with different branch coverage of individual cards on top of adding another layer of testing for correctness in a variable state instead of static (as is with unit tests). The code coverage difference is shown below.

Code coverage without random tester:

CLASS	PlayDominion	GameState	Player	Card
INSTRUCTION_MISSED	88	7	1	33
INSTRUCTION_COVERED	0	551	678	1049
BRANCH_MISSED	2	4	1	10
BRANCH_COVERED	0	46	41	98
LINE_MISSED	18	3	1	8
LINE_COVERED	0	100	130	206
COMPLEXITY_MISSED	3	4	1	9
COMPLEXITY_COVERED	0	32	34	65
METHOD_MISSED	2	0	0	0
METHOD_COVERED	0	11	14	14

Code coverage with random tester:

CLASS	PlayDominion	GameState	Player	Card
INSTRUCTION_MISSED	56	131	13	22
INSTRUCTION_COVERED	32	459	669	1085
BRANCH_MISSED	2	12	4	13
BRANCH_COVERED	0	42	38	99
LINE_MISSED	10	23	6	6
LINE_COVERED	8	93	127	216
COMPLEXITY_MISSED	1	8	3	11
COMPLEXITY_COVERED	2	30	32	65
METHOD_MISSED	0	0	0	0
METHOD_COVERED	2	11	14	14

The main efforts in creating this coverage tool was not attempting to find and fix bugs in the code, but attempting to find and fix bugs in the random test generator itself.

Overall the tester is a good way to implement tests, as it allows for bugs to be caught in runtime.

IV. Bugs

The process of bug fixing in my code is a simple process. Create the code, create a code to test what the output should specify, and if the test errors then look at where the code specifically erred. Using the output state of the code up until that point, I will implement ”here” statements to see what specifically happens during the code execution, and eventually find the root cause of the bug. This method has worked on both debugging the code, and debugging the unit tests and random test generator.

V. Conclusion

In conclusion using random test generators, while expensive in time, are a good starting method of testing edge cases and having a baseline of tests. In order to increase test coverage, it is a better idea to still build individual unit tests to check more important sections of code that the random tester may not have touched.