Spike Madden
maddens
CS 362

Intro:

This paper's going to discuss my experience testing and debugging Dominion. I'll be analyzing two of my classmates' code, amidong and hansemik, as well as mine. I will be running the various tests I've written for the assignments in this class and looking at the code coverages. I'll also look at the reliability of their dominion code based on test results. If I run into any discrepancies, I'll look for bugs to try to explain the differences in reliability.

I've never played dominion or heard about it until this class so this whole experience was new. I had a lot of fun learning the game and playing in class which helped me learn the basics. I've also debugged code using gdb and with print statements in my previous class, but I've never dealt with a project this big. It was a daunting task but I think it was really valuable to learn how to debug other people's code.

Testing:

I started off testing with some basic unit and card tests that were written for Assignment 2. There're four unit tests and four card tests that can be run on a dominion.c. The results of the tests on my classmates' code are listed below. All tests passed unless noted otherwise.

Unit Tests

Unit Test 1

This test compares the cost of the cards with the expected values using the getCost() function.

maddens: 5.00% of 580
amidong: 5.03% of 576
hansemik: 5.07% of 572

Unit Test 2

This test checks to see if the kingdomCards function returns the correct cards. The test accomplishes this by comparing a set of kingdomCards generated by a for loop and from the kingdomCards function.

maddens: 2.24% of 580
amidong: 2.26% of 576
hansemik: 2.27% of 572

Unit Test 3

This test checks the basic compare function that returns -1, 1 or 0 depending on if the right value is bigger, smaller, or equal to the left.

maddens: 1.03% of 580
amidong: 1.04% of 576
hansemik: 1.05% of 572

Unit Test 4

This checks the supply count gameState struct.

maddens: 0.86% of 580
amidong: 0.87% of 576
hansemik: 0.87% of 572

Unit tests went as expected. They covered similar percentages in both amidong's and hasnemik's dominion code. The unit tests only cover a small percentage as they are only testing a very particular method in the whole dominion file. The code and the tests seem reliable.

Card Tests

Card Test 1 (Gardens)

maddens: 17.76% of 580
amidong: 18.06% of 576
hansemik: 16.08% of 572

Card Test 2 (Smithy)

maddens: 20.17% of 580
amidong: 20.49% of 576
hansemik: 18.88% of 572

Card Test 3 (Great Hall)

maddens: 20.17% of 580
amidong: [FAILED] 20.83% of 576
hansemik: 19.23% of 572

Card Test 4 (Outpost)

maddens: 19.66% of 580
amidong: 19.97% of 576
hansemik: 18.71% of 572

Card tests were also straight forward. The percentages went up a little bit because the cards are more common than the functions tested in the unit tests. All tests passed except for amidong's Great Hall effect. I was looking through the card effects that were rewritten to introduce bugs in the dominion code. By using static analysis and comparing return values in my dominion to his, I was able to notice that in greatHallEffect, the return value was switched from 0 to 1. This is what caused the card test to fail.

In Assignment 3, we were asked to write a random test generator for two Dominion cards, including the adventurer card. I decided to test Feast as well. Random testing allows for a lot of testing with little processing power needed. The increase in coverage shows the effectiveness of random testing over very specific unit and card tests.

Random Tests

Random Test Adventurer

maddens: 24.48% of 580
amidong: 23.96% of 576
hansemik: 25.00% of 572

Random Test Card (Feast)

maddens: 23.10% of 580
amidong: 23.44% of 576
hansemik: 22.20% of 572

The random tests for adventure and feast were similar to the card tests. They covered about the same percentages as they were testing the effects of one card. All tests passed and code across the board seems reliable.

Finally, in Assignment 4, we were asked to write a full tester for dominion that would play out the whole game. I based my model off of playdom but altered the key idea of testing all cards instead of just smithy and cutpurse. By implementing the random testing element into playdom's structure, I was able to cover a lot more of dominion.c. The results are shown below.

Test dominion

maddens: 64.14% of 580
amidong: 65.63% of 576
hansemik: [infinite loop]

The test dominion is a more complete test so the increase in test coverage makes sense. The coverage on my dominon.c as well as amidong's were similar with around 64-65%. Unfortunately, the test did not complete for hansemik's dominion.c. It goes into an infinite loop in the buyCard method. I'm not really sure why it's getting stuck in buyCard because my test case is written in a way that it'll break if the player can't buy anything.

Conclusion:

In conclusion, I've definitely learned a lot about testing. I've learned the conceptual side of testing as well as the gritty actual work side of testing. I've learned about the different methods of testing: blackbox/whitebox, static, coverage (path, branch, statement), random, exhaustive and all were important in this project of debugging dominion. Debugging a project of this size was something I'd never done before, and was extremely difficult. It's hard to locate bugs when you can't really trust any of the code. Agan's principles were key in breaking down the problems and locating the causes of the errors.

The collaboration aspect of the class was also really nice. I've had multiple professors tell me in classes that we're never going to be working alone in the workplace and that we're most likely going to be reusing old code and refactoring it instead of writing new code. I think it's an extremely valuable tool to be able to read and comprehend other's code and I really think the tools I learned in this class will be applicable in the future.