Michael Chan

Cs 362

Test Report

I had already learned to work with test coverage in Software Engineering 1. We used Java and an IDE named IntelliJ to create a card game named Aces Up. It was supposed to be implemented with Model View Controller style layout, and all model code had to have full 100% line coverage because our professor insisted that we program the game with Test Driven Development, which meant that every single line of code that is written should have a test that checks it before writing any new code. I learned then the importance of line coverage, but I also learned that line coverage was a shallow way to look at test validity and how well the program ran. It's easy to fool an IDE into adding some line coverage, but if the tests are shallow, it could be as worthless as having no coverage. Other ways that showed me that line coverage was obsolete were the fact that having shallowly written functions would lower line coverage while also proving pointless to cover. One function that was in dominion.c that comes to mind is kingdomCards(). It's just a function that has like 14 lines of code but is just something that allocates an array based on the ints that are passed as parameters, which is probably something that can be done easily without a function. Writing tests for such a function would be stupid in my opinion, but I did it anyways to improve my line coverage for assignment 3.

In my unittests, I only managed to find one bug in village card not incrementing the number in hand by one properly. With all the tests I had a total of 32% coverage, with each test giving me 2-3%. The randomtesters I made could get 100% function coverage in the smaller cards, but for complicated cards like adventurer it would prove to be much more difficult to get coverage with random testing, due to the amount of branches that they can have from if statements and other logic. Assignments 2 and 3 taught me that getting good line coverage can prove the reliability of tests and for a program, but having unit tests examine specific things like branches hard to cover with random testing can really increase the reliability of a program. There were other methods for testing covered in class lectures and in powerpoints but they were beyond the scope of this class.

This class also showed the differences between unit and random testing pretty well. In test driven development, unit tests are expected, but we didn't learn much about random testing, which is a step up from just small incremental tests. Adding randomness and unpredictability to testing elements of a program can reveal the weaknesses much faster and quicker than writing a bunch of code for each specific branch that I want to look at. I remember the shotgun analogy when thinking of random testing, where the spreading of the shots can be compared to the randomness of the testing. The more randomness and more elements that are being called with the randomness, the better, because it stresses the program more and more bugs will become exposed in a shorter amount of time.

On the reliability of dominion code, I felt that dominion.c was built on a pretty shaky foundation. For most of the functions, it any variables were changed, it would most likely destroy the whole game and cause it to crash. I remember Alex telling us to not touch any of the shuffle functions week 1, so that has left an impression on me since then. Also, the readability of the overall program is kind of bad because there are a ton of gameState member variables being passed and being changed. It makes it hard to tell what's even going on. I reviewed yeja's dominion code and testers and irawank's.

In yeja's cardEffect, I think there was a mistake made with the bracketing for the switch cases. I remember for assignment 1 we were required to refactor 4 card effects into their own functions. He did the refactoring, but when he commented all the old code out, he didn't keep the switch cases so all 4 functions are technically called at the same time(I think, I haven't run his code but it works apparently). I have no idea how cardEffect would works but I'd honestly be surprised it still worked. I don't want to try to be too judgmental of code that I've only skimmed, but I can't say the code is very reliable. To examine the reliability, I looked at the test outputs produced and looked at the test c files. It says he had 100% coverage, but he was looking at the gcov of the test file and not dominion.c. In his unit test output the lines executed didn't increase after each test execution, which makes it hard to tell if each unit tester is actually covering anything. Regarding the actual dominion.c, it hasn't been modified much aside from the refactored functions so I can't really say much about that.

Irawank's cardEffect seems to have been refactored correctly, where the switch cases are still there and each function has their own case rather than all being called at the same time. It also doesn't seem like much else in dominion.c has been modified so I can't say much else on the reliability for that. Looking at his output files, it seems that he has very thorough outputs and decent coverage, due to him calling gcov with branch coverage and calls executed, which is much better than just showing total percent coverage. Also he's printing actual assert statements rather than just saying "test successful" or "test failed". However, it's obvious that a lot of his tests are showing errors, even more than mine or yeja's. This is probably due to the extra bugs that we were forced to put into our dominion.c for assignment 1. Looking at dominion.c I can see a few obvious things that are wrong in his refactored functions, but since they were placed there on purpose I won't fault it against his reliability since he's catching them with his random/unit testers.