Trevor Hammock

6/5/16

CS 362

Testing Dominion

Dominion, a strategic deck building card game that contains a large collection of kingdom cards with their very own unique set of effects. Effects that also tend to interact with other cards which further adds to the madness and complexity that is Dominion's rule set. The goal of CS 362 was to teach us, the students, various testing techniques using a poor implementation of a Dominion card game as a case study. Throughout the term we have created unit tests, random tests, and even a whole random game generator as a means to test and trigger all of the various bugs within that poor implementation of Dominion. For this test report I will discuss my viewpoints on the various testing processes and the reliability of the Dominion code that we received after running said tests on it.

First off we have unit testing which is writing a piece of code (preferably small) that triggers a specific case or bug for the module (typically a function) that you are testing. For Dominion we wrote multiple unit tests that covered various cards and the Dominion API. The unit tests were very tedious to write but writing a specification for a module or triggering a path in the code was relatively easy. They typically did not cover vast amounts of the Dominion code and they were also not guaranteed to cover every possible case, potentially allowing undefined/unexpected behavior to slip through. Personally if I wanted to test a specific behavior of the code and see if it was working correctly, I typically wrote unit tests due to their simplicity and the convenience of running a previously passing test to search for regressions in the code.

Next we have random testing which is writing code that blasts the module you are testing with random input as an easy way to trigger every possible case or bug within the code. A random tester, given enough time will eventually generate an adequate amount of random input that covers every line of code or even every possible path that the module can take. Unlike unit testing, acquiring good coverage requires a small amount of code and its also possible to generate test cases that you would normally not test for; allowing them to find unexpected bugs that can commonly slip through a unit test. However writing a full specification for a random tester is relatively difficult because one must write assertions such that they can handle every possible case a random tester can encounter. The main reason I generally had a hard time writing random testers was that I had to predict every possible way my module could output itself. And if I encountered output that I would never expect to happen or forgot to write an assertion in the first place then that test case that triggered a bug would slip though the random tester. Random testers in my opinion are one of the most powerful testing techniques that

we have learned but as a consequence are much more difficult to get right which can let regressions slip through.

Finally we have a random game generator which is writing a special random tester that plays an entire game of Dominion. The main difference between a random tester and a random game generator is that the random game generator simulates a random game of Dominion which manages to cover very large sections of the code. One particular strength of a random game generator is that it can test multiple modules in a single run and it can also run unique combinations of modules that interact with each other like running a card effect whose output depends on another card effect. Out of the three techniques discussed in this paper, no other technique can achieve entire project coverage that matches what a random game generator could do. For example on average I achieved 40% coverage for a random tester but got 70% coverage for a singular run of the random game generator (which can get better results with more runs). Unfortunately just like random testers, a random game generator suffers from the same exact faults but to a much more severe degree. A random game generator must have a full specification for an entire Dominion API that has assertions which cover every possible output one could encounter in a game of Dominion. Making it very likely that there are some cases that the generator fails to cover, letting numerous potential bugs slip through. Also writing a good AI that plays all types of games one might encounter in a game of Dominion is extremely difficult. Because of this my very own random game generator when ran with a million different seeds at most got 98% coverage of Dominion (whereas all of my random testers easily got 100% coverage in a fraction of the time).

After throwing the various testing techniques at the poor implementation of Dominion, I have come to the conclusion that the implementation is very unreliable but fixable. The implementation of Dominion that we received is over 1,300 lines of code filled with bug(s) in almost every module that it contains. Besides shuffling and drawing a card, I managed to find at least one bug in two of my classmates Dominions by primarily using my unit tests that get at least 40% coverage per test. On the bright side, a large majority of the code is written with right ideas/algorithms in mind but typically possess a logic error or two that cause it to be faulty. Which is why I was able to fix my copy of Dominion and remove virtually all of the bugs (that I can find) without having to restructure the code. If given a week or two worths of time, it is actually possible for one to find and fix all of the bugs in the Dominion code (by checking and writing most of 1,300 lines). However the code possess some structural problems which make the code a little bit of pain to fix/work with. There were many times where I wanted to rewrite Dominion's super structure so it I did not have to write over 90 characters just to add a singular card to a specific players hand. Also if in the future one would want to expand the

features or card sets available in Dominion, they would have to add and work around a implementation that was not designed to scale past its current feature set. Given that Dominion is only around 1,300 lines of code, rewriting/designing the code from scratch will make it much easier to write and maintain which will be crucial to keeping it bug free throughout its development.

For our software testing class, we worked with a faulty implementation of Dominion to learn and tryout the various testing techniques. First there is unit testing which is very simple to write and target specific cases for but has a hard time covering every possible case. Then we have random testing which covers code and cases much more easily but is tricky to write assertions that cover every possible output a module can possess. Finally we have random game generators which get very good coverage and cases that you would see during a typical execution but are extremely hard to write assertions for and need to also have a very good AI to achieve the coverage that a tester typically wants. In conclusion, the poor Dominion implementation that we received is fixable but so unreliable that it might be better to rewrite the code from scratch to make it much easier to work. Testing as I have learned through my experiences with Dominion, is a very complex subject that has no standard solution to find/resolve bugs. But by writing good testing code using the various techniques that have been introduced to us, we can become better software engineers who are better equipped to deal with the monstrosity that is software development.