Alec Shields

Dominion Test Report

My experience testing dominion involved creating eight unit tests, two random tests, and one random game player used to compare dominion implementation. Using all these different methods taught me a lot about the work needed to test a project and what advantages and disadvantages each kind of test has.

Unit tests are a sort of precision instrument when it comes to testing. They test for very specific things and work well for things like a function or a small section of code. As code begins to become more complicated it becomes more difficult to use and expensive to use unit tests for multiple reasons. It takes a good amount of programmer thinking to come up with good unit tests for complex code. In addition, these unit tests will only cover the common cases and edge cases that the programmer could think of when writing the tests. In a complicated program there are likely many edge cases that aren't being considered. In addition to this unit testing won't easily find bugs that require running things many times or in specific orders as the amount of unit tests it would take to test all the different possibilities requires far to many resources. As for coverage, unit tests would tend to cover a fairly small portion of code as they are normally pinpointed tests to check that sections of code are performing correctly.

Random testing covers a lot of the weaknesses of unit tests. Random tests are great for testing with many different starting states and for testing different paths through a program. While many of the tests that will be generated are trivial or similar to previous tests, some of them will discover new paths that a programmer writing unit tests might have taken a long time to find. Herein lies the greatest reason to use random testing: computer are much cheaper than programmers. Millions of stupid tests will often find more bugs than a suite of a few hundred unit tests and will be far cheaper. That said, there are downsides to random testing. Because it's random it can be easy for it to not catch common problem values such as 0 or max int. This can be easily remedied by mixing in some edge case unit tests into your random tester. It can also be a good deal more difficult to write than unit tests so it requires more programming skill. Mistakes can be costly as you may need to run the tester a long time in order to catch bugs, so a small mistake could result in a lot of wasted time. Random testing is likely to have better coverage than unit tests by the merit of simply having more tests that are likely to be different and follow different paths through the program.

My random game player had a lot of interesting advantages over the preview two methods. It made it easy to see which action caused a crash in the program which is useful to isolate the problem code. It also had an advantage in coverage, by running enough games of dominion it would be easy to get a very high amount of coverage. If the game works properly it wouldn't cover the failing cases but the percent of coverage would be high nonetheless. Just running most the code is a huge benefit as it would take countless unit tests or targeted random test to accomplish the same feat. Running almost all the code has the benefit of finding really loud bugs like seg faults, which are very important to catch. The downsides of this tester than simply steps through the program and logs what it does; it doesn't check that what it is doing is right so it can't find very many bugs on it's own. This means that after we have the logs from this random tester we need some to use this information to find out where the programing is behaving incorrectly. One option would be to write a script or program that knows the rules of what the program can do and is supposed to do in certain situations. Then have it parse the log file and make sure it is following the rules. For instance with dominion the script could keep track of the hand size and make sure that it is correct at each step of the game. This can be a potent method but can be very difficult and time consuming as it is similar to reimplementing portions of the original code. This can also lead to the problems of bugs in your random tester as it gets more and more complicated. Another issue with this method is that as you add more information to the logs for the script to parse, the logs become less and less human readable and it becomes more difficult to tell what has caused the problem when it finds a bug. In assignment 4 we used this log producing program a different way, to compare two different implementations of the dominion code. This was useful and interesting in this specific scenario but it isn't common to have a working, bug-free implementation of the code you are working just lying around. However, this can become a powerful tool to show whether or not your test suite is sufficiently good at finding bugs. This can be done using mutant testing, generating slightly different versions of your source code, running them through your random tester and seeing how many your test suite can "kill", or find bugs with. You can compare the log of these generated by the random path tester in order to see if the bug is "non-trivial", as in it actually affects the functioning of the code. All in all, having a tester that walks through the entire program has it's uses but it likely best to be used in conjunction with other forms of testing as a way to make sure that the test suite can catch bugs.

I compared my dominion code to that of minksc and found it to have a few differences from my own. I used my random dominion player and found that in several seeds our cards had different effects and in some, his code crashed. Running unit tests also showed issues with the feast card. This is expected as we were instructed to add bugs. My tests against olsenw showed an incorrect implementation of the mine card as well as various differences when using my comparison tester. My final conclusion is that we all started from buggy code and made are code buggy in many various ways. I think the moral to take away from this class is to write intelligent code from the start and test what to write in order to save future headaches and frustrations.