Mark Bereza
CS362

CS362 Test Report

The testing experience as a whole, despite how many bugs are left remaining in my copy of dominion and those of my classmates, has been extremely valuable simply because I was able to learn about every part of the testing stack, from static analysis to unit testing to designing random test suites to debugging, from both a theoretical perspective (via the lectures and the reading) and from a practical standpoint by being dropped into the deep end, given a task, and given little instruction as to how to accomplish. It's in environments like these that I learn the best.

Simply in my desire to complete the assignments week to week, I forced myself to become comfortable with, even proficient in, a wide array of tools and environments that will likely be useful down the road both in and out of testing. These included learning how to use a git repository for version control and constructing makefiles that compile many test suites simultaneously just within the first two weeks. In addition to those, I also strengthened my skills when it comes to writing quick and dirty batch scripts to automate tasks like generating coverage information, diff'ing output files from two different set of tests, and even automating running my test suite on over 3000 mutants and interpreting the results. Although I had used debuggers in past that were included with development tools, this testing ordeal was my first real experience with using gdb to not only debug the dominion code, but also errors in my own test suites like seg faults, infinite loops, and wrong return values.

The availability of such tools helped changed my mindset regarding testing to a healthier one. Before, when I encountered an elusive or game-breaking bug I would often devolve to stubbornly putting printfs everywhere, commenting out huge chunks of code, and agonizing over every line in the source code. Options are important because when your current method isn't working, you can continue to productively attack the problem by simply switching up tactics. Instead of squinting at the source code hoping that lightning will strike, I can instead start parsing through the lines in the debugger. Instead of putting printfs everywhere suspicious, I can conduct modular unit tests to eliminate working functions from the pool of potential garbage.

I also somewhat addressed an oft repeated fallacy in the CS community - that one should work smarter not harder and that a little brainstorming and design can save you lots of brute force work in the long run. While this is likely legitimate when it comes to designing applications, effective testing practices spit in the face of this idea. Often times, I found my quick and dirty mindless random tests told me much more than my expertly crafted and purposeful unit tests simply because I could run the former literally millions of times over countless variations of parameters.

One frustrating thing about my experience with testing and debugging over the course of this term was simply how hard it is to get a feeling of completeness or even adequacy, and I'm beginning to speculate that this issue might just be one inherent to the problem of testing. Not only is success a fuzzy concept, but even the metrics used to define success are under constant scrutiny. Code coverage is in this weird position where if you're under 70% you're probably doing something wrong but 100% doesn't mean you've caught every bug. Not even close, in fact. It also contradicts intuition since it's very easy to believe that as you run more code, the more likely you are to catch a bug, but the reality is that if anything coverage's correlation with bug detection in an emergent property of huge differences in coverage.

A combination of interest and frustration with this metric motivated me to pursue mutation testing for my final project. In the process, as described in mutation.txt, I became familiar with Cygwin and compiling/building software, just another tool on the massive tool belt this testing course has provided me. If I have the time, I will likely continue to study and practice using the cmutate tool. I'm very curious as to what needs to be done to get a drastic increase from the 27% kill rate I obtained. Especially since my attempt to increase it did absolutely nothing.

One big takeaway my efforts have left me with is that priority and severity are very important concepts when it comes to debugging. This was especially apparent when I spent days writing a comprehensive test suite to play hundreds of games of dominion, only to have that same test suite hang or loop indefinitely or producing nonsensical output simply because there were extremely large bugs and even errors with the design of the dominion.c code that managed to obscure all the nuanced and careful checks I designed my test suite to conduct and instead resulted in output that could've been just as easily produced by a particularly lucky unit test.

What's particularly interesting is that there isn't as much of a correlation between the severity of a bug and how easy it is to find. After spending hours in the debugger and scouring the source code for the cause of these bug-obfuscating failures, I just gave up. I instead resorted to simply running my test suites on other people's dominion files as a way of evaluating my tests' effectiveness. In particular, I ran every test I produced these past two months on two different dominion.c files: one produced by student with username eastwooc and another with the username hollidac. I recorded the output of these tests onto two separate text files and here were my findings:

```
eastwooc's Dominion.c

unittest1:
       TEST SUCCESSFULLY COMPLETED.

unittest2:
       FAILED ASSERTION: Checking to see if numActions has increased by 1.

unittest3:
       TEST SUCCESSFULLY COMPLETED.

unittest4:
       TEST SUCCESSFULLY COMPLETED.

cardtest1:
       TEst SUCCESSFULLY COMPLETED.

cardtest2:
       TEST SUCCESSFULLY COMPLETED.

cardtest3:
       TEST SUCCESSFULLY COMPLETED.

cardtest4:
       FAILED ASSERTION: Checking if Village is played successfully.
       FAILED ASSERTION: Checking if Village isn't in hand.
       FAILED ASSERTION: Checking if Village is in discard.
       FAILED ASSERTION: Checking if actions have increased by 1.

randomtestcard1:
       TEST SUCCESSFULLY COMPLETED.

randomtestcard2:
       TEST SUCCESSFULLY COMPLETED.

randomtestadventurer:
       FAILED ASSERTION: Checking if number of treasure cards in deck/discard/played decreased
by correct amount.
       FAILED ASSERTION: Checking if hand size increased by correct amount. (Occasionally)
       FAILED ASSERTION: Checking if deck/discard/played size decreased by correct amount.
       FAILED ASSERTION: Checking if number of treasures in hand increased by correct amount.
(Occasionally)
       Number of Adventurer remaining in hand fails: 16/100.

testdominion:
       TEST SUCCESSFULLY COMPLETED.

Code Coverage: 93.99%
```

```
hollidac's Dominion.c

unittest1:
        TEST SUCCESSFULLY COMPLETED.

unittest2:
        TEST SUCCESSFULLY COMPLETED.

unittest3:
        TEST SUCCESSFULLY COMPLETED.

unittest4:
        TEST SUCCESSFULLY COMPLETED.

cardtest1:
        FAILED ASSERTION: Checking if Feast isn't in hand.

cardtest2:
        TEST SUCCESSFULLY COMPLETED.

cardtest3:
        FAILED ASSERTION: Checking if Smithy is in discard.

cardtest4:
        FAILED ASSERTION: Checking if Village is in discard.

randomtestcard1:
        FAILED ASSERTION: Checking if Feast can't be played.

randomtestcard2:
        TEST SUCCESSFULLY COMPLETED.

randomtestadventurer:
        FAILED ASSERTION: Checking if number of treasure cards in deck/discard/played decreased
by correct amount.
        FAILED ASSERTION: Checking if an Adevnturer was removed from the hand.
        FAILED ASSERTION: Checking if hand size increased by correct amount.
        FAILED ASSERTION: Checking if deck/discard/played size decreased by correct amount.
        FAILED ASSERTION: Checking if number of treasures in hand increased by correct amount.
(Occasionally)
        Number of Adventurer remaining in hand fails: 92/100.

testdominion:
        Loops forever.

Code Coverage: 37.59%
```

At first glance, these two results don’t seem drastically different, but I would posit that eastwooc’s version is significantly more functional, even if it’s still bug-ridden. This is because it can successfully and consistently run entire games of dominion without blowing up or hanging forever. When I run my test suite on hollidac’s version, or mine, or most, the suite continues running without end. This specific bug wouldn’t be such a big deal if it wasn’t for the fact that the majority of my coverage results from my testdominon file, which cannot run on the buggier versions, leaving my total coverage significantly lower (37.6% vs 94.0%).

If I were an actual tester working for a company trying to push out this dominion game, I would focus all my resources (and encourage others to do the same) on first fixing the bugs that make good testing difficult or impossible. When fighting with your hands tied, it might be better to slip out of the rope before punching.

All in all, this class has made me think of testing less as a thing that causes me to pull my hair out and more as a thing that might be my focus in grad school or even my career someday. That’s plenty great if you ask me.