Braxton Cuneo

Software Engineering II

Final Test Report

Software engineering, as a profession, is fun, weird, and hard, much in the same way that philosophy is fun, weird, and hard, because both are grappling with the same sorts of ideas. Both have a system by which one can leverage information to gain insight and entertainment about the world around us, software engineering has the hard surgical logic of computers and philosophy the intangible medium of axioms and thoughts. Both are filled with issues of what happens when two parts of a system are inconsistent or incompatible with one-another, like two functions written by two programmers that didn't know one-other having their code merged together by a third. Both deal with problems of causality, such as what defines the cause of an error during code execution. Both deal with the fact that there is never a single way of solving a problem, as demonstrated by the wide variety of debugging methods developed over the years. It is with this in mind and with great enjoyment that I must state that this has been a fun, weird, and hard class.

Without further ado, I would like to take use through the educational journey I experienced throughout this term, using the dominion code of two fellow class mates (usernames "harderg" and "henninch") for examples.

Unit Tests:

Prior to this class, unit testing was essentially all that I did along the lines of formal testing for my code. It was quick, dirty, and straightforward. This being said, I would not say I wrote "quality" unit tests by any means. Even in assignment 1, I'm not sure I quite had a grasp of what exactly I was testing for. I wrote for a lot of cases, but not all of the cases were perhaps completely necessary had I a better idea of what I was looking to resolve.

For example, my first unit test checked every single card in the supply to ensure that it had the correct amount of cards. This was not necessarily a hard bug to find and could probably be discerned just from looking at the code, but I viewed more tests as inherently better tests. One may almost agree with me until you notice that almost every single student's code except for mine fails my second unit test with a segmentation fault. This is because I decided that it was a good idea to ensure that, whenever someone accepts an index as an argument, one should always check to make sure that it is within bounds. Furthermore, I frequently reported back errors if a function did not throw an error code if I handed it bad input, regardless of whether or not it caught said input. This being said, I also covered a good deal of conditions that one may not consider to check, for instance ensuring that buyCard actually gives players curses if they buy embargoed goods. To these ends, one must judge selectively the results of my unit tests on other users.

Unsurprisingly, my unit test coverage is relatively high across both harderg's and henninch's code. Henninch ended up with 40.28% coverage whereas harderg received 37.99% coverage. Both have approximately the same rates of failure, as well, with many of the failures being for the same condition.

For instance, both have similar problems with the steward card's coin bonus effect. Both failed to have buyCard give curses to players that buy embargoed supplies. In addition, both did not remove any of the pre-existing errors in the card effect for sea hag, including not giving curses to people with empty decks and not rotating in the proper order when distributing curses.

I am not one to fault them for these failures, however, as this class mainly emphasized the finding of errors rather than the resolution of errors. There are many tests which both harderg and henninch failed that I would have failed as well.

Random Tests:

When random testing was first introduced to me in this class, it appealed to me in a lot of ways. First and foremost, as a man who grew up on games such as nethack and minecraft, which relied upon elements of randomness being used to powerful effect, I enjoyed the idea of applying that same sort of idea to testing. Secondly, random testers essentially write and test code for me and much faster than I could ever hope to achieve with unit tests. This being said, random tests can have their limits as well.

For instance, when executing my random card testers on harderg's and hennich's implementations, both resulted in segmentation faults. Since my random card testers do not constantly output diagnostic data about the game state, I can only get so much of an idea about what is causing the issue. Using gdb, one can discern that it is the result of accessing a bad index in the array "temphand" in the function "cardeffect," however I do not have any idea about the sort of condition leading up to that circumstance as it is highly randomized.

For the remaining two card testers, both do very well. Neither fail any tests for their smithy card effect, and both only fail a single test for their steward effects, the test checking the coin incrementation option. Plus, as far as bugs in dominion go, that is one of the most common and hard to root out, so I would not judge them too harshly on that.

Interestingly, harderg's coverage came out to 31.90% whereas henninch's coverage was 32.33% after executing 10 tests on each random card tester. It speaks to the power of random testing that two random tests could get the same sort of coverage as my 8 unit tests combined. Simply put, more varied cases when testing a system means better coverage, and random tests offer that in abundance.

However, it would be no good to discuss random testing without bringing up the full random tester that is implemented as part of assignment 3. Having the capability to try every single card effect and so much more provides a great upper hand when it comes to getting coverage. After performing just 7 runs of my implementation of "testdominion," harderg's implementation of dominion.c received a coverage of 70.7% whereas henninche received a coverage of 74.73%. This being said, the full random tester still suffered from the same problems as before. For example, harderg's dominion simply hangs after a certain point if one uses the seed 72, and henninch's has a similar issue with the seed 23. Opening up gdb, the problem becomes apparent. Neither of them have resolved the infinite loop in their feast effect code, meaning that every random card use by the random tester has some small probability of using feast wrong and trapping it in an infinite loop.

Taking a Step Back:

So, in the end, what does all of this indicate regarding the reliability of harderg's and henninch's code? That, while an important question, is but one side of the problem. One should also be asking what reliability does the testing software I wrote have in accurately finding and reporting errors. Furthermore, one should consider the possibility of false positives. For some of my unit tests, not returning an error value when given bad input was counted as a failure, but that was by no means in the specifications for dominion. It was at best reporting bad code smell.

If one needs any further indication of the fallibility of my testing code, they need only test my implementation of dominion, which is also far from perfect. I would have used my differential testing program in this report if it weren't for the fact that my code could not in any reasonable sense act as a good benchmark for differential testing. Mutation testing, even if I had it at my disposal, mainly checks for self-consistency in one's methods of checking for errors, and would only be useful for verifying the usefulness of my tests rather than the proper behavior of harderg's, henninch's, or my code.

With relatively high certainty, I can state that harderg's and henninch's code are not reliable enough to be considered usable in a professional implementation of a Dominion application, if only by virtue of the fact that the odds of so many false positives occurring is astronomically low. This being said, the faultiness of this code is tangential to the larger point of this class. We were given extremely buggy code because this is a class about finding bugs, and having buggy code makes it easier for beginners to do so. Furthermore, much like programming, debugging is a process best learned through hands-on learning, experimentation, and trial and error, a process which is rife with misunderstandings, mistakes and failures. This is perhaps the most valuable point I have learned from this course:

In philosophy, one must know their logical fallacies so they can think before they speak. So too must software developers know their coding fallacies so they can test before they launch.