Alex Uldricks
CS 362 Final Project

Introduction

Testing a huge chunk of lightly commented, poorly formatted, and incredibly ugly code for a game I've never heard of was something of a daunting prospect. I'd be lying if I said I wasn't a little confused as to where this class was going when I was handed this task. The professor insisted that the previous software engineering course wasn't necessary, so it seemed doubtful that we would build on the things I learned there, and yet, my experiences in Software Engineering I had left me feeling as if I had covered all the basic steps of development. After leafing through the mess of code that is our initial dominion, however, I knew I had so much to learn if I was going to even begin to make heads or tails of the whole affair, let alone attempt to fix it.

While I learned a bit about unit testing prior to this course, if I'm being frank, my debugging methods were pretty crude, usually just searching for logic errors by hand or using a simple debugger to insert breakpoints. My experiences with unit tests felt essentially obligatory, not terribly applicable, and certainly not something I would make a habit out of. My experience testing dominion exceeded my expectations for this class by not necessarily focusing on too many new techniques, but rather, making the concepts I had already learned seem practical and desirable. By the end of my time in CS 362, I feel not only as if I had learned about these concepts in an academic context, but more importantly, I had been taught the practical ins and outs of these forms of testing. If you had asked me what level of code coverage is ideal for random tests, or which cases are important to cover in any unit tests, I would have been pretty clueless, and now that I have both been given and implemented a lot of that information, I feel pretty confident that I'll be using these testing procedures moving forward to brute force some results out of my code, instead of squinting at it tiredly for hours.

Testing Assignments

Unit testing and random testing were the parts of dominion testing that most resonated with me throughout the course. Modularizing pieces of the overall picture and determining how to flush out problems by throwing tons of cases at them was a task I ended up relishing, and the lectures gave me a much better idea of both how to effectively implement these tests as well as their pros and cons. Just looking at dominion.c as a whole, the notion of acquiring high levels of code coverage was insanely intimidating, but once I began to break down each card, the ways I might extend that to other cards and functions came into a much sharper focus, and by the time I was required to write my complete game implementation, it wasn't so difficult to acquire reasonably strong coverage. In that sense, the progression of the assignments worked as an excellent curve, while technically each involved disparate tasks, the increased familiarity with testing concepts and the dominion code was indispensable to making the next project feel manageable.

In terms of specific coverage numbers, my unit tests started out weaker than my random tests would become, followed by a pretty sharp dropoff when attempting to test the whole mess at once. My unit tests focused on pretty simple cards, and as a result attained about 75 to 80 percent coverage despite not being incredibly thorough. Random testing combined with my experience with unit tests let me achieve better numbers, between 84 and 92 percent, while the complete implementation only ended up reaching about between 62 and 63 percent coverage, just over the requirement set by assignment 4. While I wish my total implementation had been better in this regard, it was definitely the assignment that challenged me the most, and as a result I also learned the most about the code as well as testing,

and I feel fairly confident that future attempts at such tests will be much stronger.

Classmate Code

In all honesty, I hadn't examined much classmate code for coverage numbers or specific bugs before the end of the term, so my knowledge of their practices or the numbers they reached is far from thorough. For the sake of my bug reports, as well as this writeup, I've examined a lot of their outputs from their unit, random, and complete gameplay tests for coverage and reliability. Firstly, I took a look at the repository of andrewsm while looking for code to test, and found that while his reported coverage with his unit tests weren't terribly high, the code coverage I achieved with my tests were similar or higher than what I achieved with my code. Since my modifications to dominion were fairly minimal, just inserting bugs in assignment 1 and doing what I had to in order to make each assignment compile and run, I feel pretty confident in saying that their code had more work put into it, and is more reliable overall. The one bug I ended up unearthing with my unit tests was pretty simple, so I'm sure it was inserted during assignment 1, his village card subtracted two actions instead of adding two, which is easy enough to fix. User jiangzh had similar bugs for my unit tests to find, which I also believe to have been inserted in the first assignment, and while his repository seemed similarly unmodified, his code managed to achieve slightly higher coverage from my unit and random tests than andrewsm. Assuming this is representative of any practice in particular, I would say that his card functions from assignment 1 seem to have received some decent polish, as well as a few other cards within dominion.c having a handful of fixes. Looking through github, I found a lot of the code out there to be similarly reliable to my own, and this was reflected in my experience testing the code of andrewsm and jiangzh.

Conclusion

I feel extremely confident in saying that CS 362 has changed my approach to debugging in many many ways. This writeup focuses primarily on my work with dominion code, but the information I was provided by the professor did a great job of being specific and teaching positive debugging practices, both in terms of what to aim for as well as what to avoid. Lessons learned and Agans' principles both instilled good habits in me, and combining that with the practical testing experience I gained through testing dominion, it's impossible to see myself approaching bugs in my code the same way as I have in the past. Unit testing and random testing both seemed like foreign concepts, not worth the time or effort for any of my work, but it's easy to see now just how relevant and powerful the techniques I have learned are, despite how situational they might seem.