Aileen Thai
CS 362
Testing Dominion

Personal Testing Experience

Overall, testing Dominion was a struggle, but a good learning process. Most notably, I learned that testing isn't something I would pursue as a future career... Nonetheless, it was useful, and I'd like to get a bit more practice in the future.

Of the assignments we partook in during this class, I enjoyed unit testing the most. While not as powerful or efficient in comparison to the other methods of testing, they were the easiest to understand (and the most intuitive to write, in my opinion). Reliability-wise, unit tests are pretty faulty, since they don't allow for variety (in the data you give it). They're rather well suited for quick jobs, but I can see why one wouldn't want to write an entire test suite consisting of only unit tests. The somewhat slap-dash mindset that comes with writing them would easily lead to a maintenance nightmare.

As for the random tests... I ended having to spend a lot of time rewriting mine since I wasn't thorough enough the first time. My main fault came with the number of times I ran through the tests. While I randomized the parameters of the cards that I tested, I neglected to run through the tests multiple times as a single player. Instead, I opted to randomize the number of players, and ran through the card's testing that way. Considering that the max amount of players for our implementations of Dominion was four, that greatly limited how many times the card was played. When I ended up revising my random tests to run several hundred times (on average), it was much easier to detect bugs. Mainly, because the sheer amount of times that tests were run, increased their chances of showing up. The only downside to this (that I found), was that the .out files generated by the tests could no longer be parsed manually.

One of the things that I struggled with the most in this class, was writing the game generator. While I was able to reach the desired amount of coverage, I found it slightly disappointing that I had to run the game generator several times to do so. Definite improvement could be made in how the action phase was implemented and automated. There were also several parts in dominion that I didn't cover in my generator, such as kingdomCards(). For that particular function, I wasn't sure how to randomize the unique integer inputs off the top of my head, so I ended up disregarding the function. In the end, I ran out of time before I could go back and include it in the game generator. Other functions/cards that I never figured out include ``estate`` and ``feast``.

Many of the other problems I encountered had to do with neglecting to fix buggy code. While the tests helped me determine/localize issues, I never got around to doing anything about them -- which was a mistake. Ignoring bugs is only a temporary solution, after all... I regret not going back and reformatting all of dominion.c as well. Midway through the term, I

got about a quarter of the way through reworking the code but apathy won out. In hindsight, it would have been less of a pain to debug a file that didn't hurt to look at.

Status and View of Others' Code Reliability

kannas:

It seems that her unit tests are somewhat questionable. Looking at the unittestsresults.out file, all six of the listed tests resulted in (at least) one failed assertion. Of these, a card test for ``smithy`` was included. After viewing the test case, it looks like one of the assert statements was incorrectly defined. Other than that though, the test was solid. After a brief overview of her other unit tests, it seemed that most of them were failing not because of a fault with the Dominion implementation itself, but rather because the tests were either checking for the wrong things or missing bits of code.

Coverage results for the random tests were all very high, with the lowest being 93.75% (out of 16 lines). The card being tested was ``gardens``. It seems that a line within the main function was missed, but it is difficult to determine which without more information.

Her random game generator looks to be very thorough. It has a function to select the random cards used to populate the deck, as well as one to check the affordability of a card before its purchase during the buy phase. There are also functions that randomize the selection and performance of an action as well as the players' choices. However, she does state in her differential.txt that 60% coverage was only reached when the generator was seeded with a value at or near 60. I'm as to what would cause this to occur.

nguyalex:

His unittestresults.out file was very informative, and it was shown that three out of the eight unit tests fail. The functions/cards that resulted in a failed assertion were: whoseTurn(), ``embargo``, and ``outpost``. It looks like ``embargo`` and ``outpost`` fail their tests because of the cards' reliance on the working functionality of updateCoins(). However, it is a little hard to say why the test for whoseTurn() is failing. My guess is that the assert statement is checking for the wrong thing.

Coverage results for random testing were all fairly high. Unfortunately, the .out files for the random tests were not as informative as that of the unit tests. From what I can tell though, there is only a drop in coverage because of an 'else' statement that acts as a failsafe in case the user doesn't provide a seed argument.

Again, the random game generator seems like it covers all the bases. However, it doesn't look like the starting deck is randomized. Also, much of the functionality seems to be contained in the main function, which may lead to some difficulty in debugging down the line.

In Summary

The first student had a more reliable random game generator in my opinion. This is mainly due to the way she split up her functions within the file. However, the second student had more comprehensive unit and random tests (especially since the first student incorrectly defined her assert functions).