Cameron Stuart
CS 362-tesreport

As a whole, the testing of the provided dominion implementation forced the implementation of a variety of different tools, some that were previously unused. This was mostly due to the complexity of a larger program that was written by a different and unknown author. So testing dominion was effectively stepping into an unknown and complex project that was ripe with bugs, some minor, and some which were huge oversights that crippled the game as a whole. Unit tests allowed for concentrated forays into specific areas of the code, while random testers bombarded large areas that may have otherwise been skipped over.

Unit tests were the first tool used to start testing dominion, and proved to be both simple and pretty effective at finding specific bugs. The custom nature of a unit test allowed for trying specific values within different functions as if the user was stepping through the code manually. This also made it easy to determine what bugs were when the test failed. The largest downside to the unit tests were part of what made it easier, their tester defined focus. Many times the intuition that guided the test's focus could yield some bugs, however more obscure bugs were condemned to obscurity, never to be uncovered with this methodology. As a whole the code coverage of unit testing was only around 38%, which on its own suggests a massive amount of uncovered bugs simply by not covering surrounding code. The low code coverage and specific nature of unit tests really opened up the possibility of random testing.

Random testing allowed for a much greater volume of testing to be done on the dominion code. Mostly this came in the form of a large amount of different inputs, many of which a manual tester may not have come up with. This approach was able to yield 100% coverage of targeted parts of the dominion code, specifically certain card effects. That coverage suggests a much greater level of confidence in the targeted code for the bugs that it found, for if those were fixed then that section likely was much more resilient. One of the key downsides to the random tests were that the results of them were not always as easy to analyze as those from unit testing. There was also times where the tests yielded a great deal of failing cases, but most of which a real game of dominion wouldn't likely encounter. So one of the first improvements to the random tests for dominion was to continue to narrow their range of inputs, while still maintaining the same level of code coverage.

The final method of testing code utilized some random testing to create an entire random tester that could play a game of dominion with random inputs. One of the hardest parts of this tester was to create a good oracle that could decipher failures in the code in a way that was relevant, rather than just un-caught bad inputs. What added to this challenge was the dependency on a game of dominion on the entire state of the game, so often times the arguments passed to a function didn't cover the entire story of the bug. This random tester was ultimately used for differential testing with a classmate's implementation of dominion. For me I used the implementation made by user "jiangzh" along with a script I made which ran my random tester on both implementation and then compared the output. In this section I again ran into similar difficulties that I had running the tester on my code alone, since it was difficult to determine the meaning behind differences. The differential testing would've probably been more helpful for a program that took in more inputs at a time, or if simply the scope was focused in a bit more to a specific region of the dominion code rather than the whole implementation. The positives to this

method of testing was a high amount of code coverage which was over 60%, much higher than the two previously used tools.

The second classmate implementation of dominion that I tested was the user “ellibran”. The testing of this implementation was different from the last since all of the test that I used on my own implementation were used on their implementation, not just the random tester. The results were similar to my own, not exactly, but enough that would suggest a high degree of reliance on the dominion API from my tests as well as tests that would also look for invalid inputs that might not come up in a user run game of dominion. The most difficult part of testing other classmate’s implementation for me ended up simply moving their entire environment to the flip servers where I could work on it. Much of those challenges were from attempting to merge the functionality of our two different make files since the rules were almost always different, and occasionally there was an implementation I would try to test but I had tests that were two specific to my dominion implementation to work with theirs that utilized different functions and header files.

After testing three different implementations of the same program with many of the same tests for each one there are a few clear conclusions as well as lessons for future testing. Overall, the reliability of my implementation of dominion is more or less quite similar to the two other ones that I tested. As of now dominion can actually be played, mostly if the user sticks to valid input only and doesn’t make any mistakes when passing arguments for functions. However, none of the testing efforts yielded much higher than 60% code coverage so I would have to at a very minimum say that the reliability of these dominion implementations absolutely cannot be much greater than 60%. Currently with the bugs that were found, that reliability would still be quite a big lower than the code coverage value. In the future it would be much more helpful to treat the testing with a bit more of a “black box” approach and rely more on the API for handling requests. That way the tests are more universal and can be easily ported to similar implementations and still look for the same bugs.