Pranav Ramesh
CS 362
Final Project

Dominion Test Report

Introduction – Testing Experience

Computer Science is not only about programming, in fact there are many aspects that collectively computer science. One of the main aspects is testing which is a very crucial and integral part in software engineering. Until I took this class, the extent of my testing was just getting the given function or program to output what it needed to output. And sometimes I wouldn't even know exactly what I did to get that given output, as it sometimes was guess work. Then when I entered this class and we were given the Dominion code, I was quite overwhelmed and did not know what to expect. Now being at the end of the class, I can definitely say that working with Dominion and learning proper testing greatly helped me understand the importance of adequate testing.

Since the Dominion code was nowhere near perfect or close to working for that matter it was a challenge to understand all the functions and what role they served in the game play. Another challenge that I faced was that I did not know the rules to Dominion and they did not seem very intuitive so there was a bigger learning curve there that put me a little behind. To get a better idea as to how the game worked I watched different videos that played the game as well as reading documentation on the game.

After gaining some understanding of Dominion, I formally began to test the source code that we were provided in class. Since the code was lengthy, I had to come up with a way of testing efficiently as there is no way to test the entire code in one go. Since the game was based on cards, I decided to a select number of cards by writing different unit tests to make the test pass. And most of the time, the tests did not pass on the first attempt. By figuring out what made the test fail, it was relatively easy to pinpoint where the bug was and how alter to the code to make sure it passed every time. Through this iterative process, I came to realize there is no quick and easy shortcut for testing. The next type of testing that I learned was random testing and very quickly realized that this provided a more accurate response as to whether a test passed or failed. The reason behind this is that when a random test is set up, it will test the extreme cases as well as the base cases. Holistically, this is a better approach to testing and could help pinpoint bugs faster.

This term I started developing small applications for demo purposes and used the skills of testing from this class to achieve quality code. My applications have been no longer than five hundred lines of code so it was significantly easier to test each aspect of it. But after being exposed to Dominion's lengthy source code, working on my small projects seemed like a breeze. I was able to create unit tests as well as random tests and was successful in getting all of them to pass. The other thing that made my projects easier was that I wrote the source code so I knew what every function's output should be and the logic behind it. Going forward I will make sure to use these testing practices as they are the most appropriate way to test and debug code, instead of using the infamous "print statement model".

Code Coverage of Classmates' Code

When looking at other classmates' code, I was not sure what to expect in terms of code coverage since we were pretty free to do whatever testing we'd like on the code apart from the assigned parts in the homework. Since we all started with the same source code, I also expected some sort of consistency so this was a good experience seeing what creative ideas others had when writing tests and trying to get a good code coverage.

Mihai Dan (danm)

When I conducted my unit tests on the source code that I altered, I received a 38.46% code coverage. In comparison, Mihai's source code that he altered yielded a 34.95% code coverage. It was surprising to see our code coverage be so similar even though our style of testing was completely different. After running the tests multiple times, I came to the conclusion that the reason why there was a difference in our code coverage was because of the way we refactored our code. When I did my testing, I tended to refactor after every major change I made and then I used the gcov tool to measure code coverage.

In Mihai's random tests he achieved higher test coverage all around. Looking back at my source code, I realized that I did not refactor as much as I should have when conducting my random tests. That could possibly be why my code coverage values were in the low twenty percent range, while Mihai's code coverage was in the forty percent range.

George Harder (harderg)

As previously stated, the code coverage I received on my source code was 38.46%. In comparison, when I ran George's source code it yielded a 34.45% in code coverage. At this point, I had to assume that there was a discrepancy in percentages due to our refactoring styles. What I realized with George's code is that he took another approach and had the card go through the cardEffect() function rather than separating each card into their own function and then calling each function within the switch statement.

George's random tests yielded relatively high code coverage. My random tests, as previously mentioned, did not return very high code coverage and I can pretty confidently say that it was primarily due to refactoring.

Conclusion

Overall, I learned more in this class than I ever imagined. Testing code like Dominion is no walk in the park as it requires a lot of attention to detail. But because of that it made me a more careful tester by making sure every possible value was tested and accounted for. I will definitely implement the tools I learned in this class into my future classes and projects as I hope to build from them and become a successful software engineer.