Dominion Testing

The dominion implementation we were given was broken beyond reason. It was purposely broken to make us dive into the life of software testing. The first thing to be done was browser through the code to try and understand how the game was structured and how certain things were put together. It was obvious from looking at it that the code was sort of a conglomerate of multiple people's code. Generally when those people have not coordinated beforehand the conjoined product tends to be a bit of dumpster fire.

My initial assumptions were accurate. The application itself did compile, but is near non-functional as a functioning game. I would argue that while many functions are problematic the foundation of how the application works is not well thought out. For example the card implementations are just a simple structure with an increment and decrement expectation of the "pile" number. Meaning the card piles are reliant on the calling functions to deal with the pile. It would have been much smarter to create a stack data structure and functions into the game so that none of that functionality is relied on per-card function. You can see an example of this in my adventurerCard function that uses a generic POP and PUSH functions. Abstraction and modularization is always a good thing.

We started testing by doing some refactoring of the project. In this case the refactoring was a basic modularizing of card functionality, rather than having each switch case in the switch statement each switch calls a function of the card implementation. This helps clean up the code in two ways. The first is that it shortens the massive ridiculous switch statement in the Dominion.c. Secondly it allows us to use the card functions elsewhere should the need arise.

Apart from this it also makes finding bugs with a debugger easier as we can see which function the program went into before an error occurred, along with seeing the variable inputs and return values. Another helpful part of modularizing the card functions is it allows us to attempt swarm and random testing by targeting specific functions in our test code (more so swarm in this case).

The next step was to write unit tests for specific functions and then for specific cards. The two types of unit test meet different needs, consider that cards will only be called under specific circumstances while the unit tests will be called in most cases and will behave differently under different circumstances. As an abstraction I think that the card tests would be much better suited when used with swarm testing because of the above reason. You can more closely hammer a card with different variables when it is one of the specific targets rather than in a full random test. This was evident to us in class when Professor Groce demoed putting a very obscure bug in his AVL tree. It just wasn't going to happen unless we are specifically running over an area. On the flip side the unit tests probably would work a lot better with random testing for the inverse reason. Overall unit testing yielded some major flaws in how the game was put together. It found a flaw in the coin management, 3 of my 4 cards I chose to test, and a bug with game instantiation and calling a nil card type.

Then we got into random testing and line coverage. Which is where a lot of bugs were exposed in Dominion.c. Random testing is cool, really cool, because it will try thing you could never have imagined doing in a normal test or unit test. And that is the beauty of it, you can't trust users to not doing something insane, or for something to go wrong somewhere outside programmer control. It's the job of the programmer to handle those edge cases gracefully. Random testing will try all of those extreme bizarre cases and will expose holes when they are

present in the code. My random testing was getting nearly a 20% fail rate from tests that weren't doing anything special, just trying different card and numerical options. This allowed me to go in and look specifically at the problem, which turned out to be an issue in how the card piles were being incremented and decremented, and thus determining when a game should be over. The game either wasn't ending when it should, or it was recording completely wrong scores at the end of the game. The specific card causing issues was adventurer and the victory point recorded. While I never got time to fix the victory function, repairing the adventurer card yielded immediate improvement to the success rate of random testing. That sort of information would be very powerful for developers and QA alike.

Knowing a change worked is much better than knowing a change avoided the problem.

My friend Trevor Hammock's dominion code is amazing. He did an amazing job rebuilding a lot of the program and fixing countless bugs. There are still problems, but from what we discussed they are problems with the underlying structure that should have been addressed early on in the development process. I would say his code and game is presentable and playable by all means, but not something we could expect a user to pay for. My other good friend Sam Lichlyter is even worse than mine. He didn't have the time to devote to cleaning up a lot of the segmentation faults in his code even. It shows when I ran my random testing on his code and it segfauled halfway through.

My summary of the Dominion project and its overall stability. It is stable in terms of it being able to run. It is/was not stable in terms of avoiding incorrect results. To put it another way; the code is like a clothes dryer that turns off long before the clothes are actually dry. It

functions and does what it is supposed to, but the end result is not what a user wants and not what a developer expects. And eventually a user will go back to a clothes line if it is not fixed quickly. But, that is getting too much into the business side of software. The bottom line is, the Dominion code was really bad and basically non-playable, but it would run.

That is my assessment of the Dominion code testing, it was some rough code to dig through, but I learned a lot from it.

-Tanner Fry