CS 362 Dominion Testing Report

Rikki Gibson

1 Design and language

The choice to write this program in C forces programmers to manage a lot of difficult little things that are easy to get wrong and very easily go unnoticed. The program is defined with a somewhat small and limited notion of the human interface that will be used to play it. There are lots of parallel arrays or lookup table style solutions for things, such as an enum for all the cards in the game, and then accessor functions that take an int and produce some attribute associated with that card, such as its price. The program is not designed by thinking first of how the problem domain can be best expressed but rather by thinking of how to easily deliver a C representation. One unfortunate manifestation of this is the choice to pass around values named "choice1", "choice2", "choice3" rather than come up with some data types or additional functions that represent a player playing a particular card, that are not necessarily the same between each card. In addition, the compiler's ability to emit useful warnings and the debugger's ability to provide useful information about program state is obfuscated by the use of the int type instead of the enum CARD type.

2 Testing methods

I wrote a suite of 9 unit tests which cover about 21% of the core dominion.c file. These were fairly simple to write and tested only fairly trivial assumptions about the program. At least two meaningful bugs were revealed by them, one involving a for loop using the wrong count to access elements of an array, and another involving a programmer likely misunderstanding what a particular function is supposed to do.

My random testers expand the coverage to 35%. Each random tester is simply tasked with randomly constructing a viable game state, passing it to a function that plays a particular card, and checking that certain invariants hold after the other function returns. In practice, the random tester only identified cases where overflow causes strange behavior. This probably comes down to the fact that it's hard to devise good invariants, and not all cards or functions have easily identifiable invariants that can be exercised by random testing.

My full game differential tester gets 68.5% coverage after running it with many random seeds. Instead of creating a random but sane game state, it uses dominion's built in functions for initializing a game and then performs random (but mostly valid) moves while logging the outcome of moves and changes in game state. The implementation of dominion.c was swapped out between my own and a classmate's version, and the outputs of the two random players were compared. This does succeed in finding quite a few differences between versions, but it is difficult to go from the error back to the fault that caused it. This is aggravated by the fact that neither my version nor my classmate's version is a "definitive" implementation which can be relied upon to be correct.

I also wrote a tarantula tool which will run a given set of test cases, and attempt to localize faults, giving each line a "suspiciousness" score based on which lines execute in failing test cases and which lines execute in passing test cases. This tool provided modest gains over merely using gcov to find lines covered in a failing test.

3 Reliability evaluation

My copy of dominion should be considered basically unreliable, as I did not make a concerted effort to fix faults besides a handful that were identified by doing assignments 1-4. My unit tests and random testers give the same outputs on my code as they do on two of my classmates', so I am assuming that the three of us have approximately equally reliable code.

4 Improving reliability through testing

More unit tests could in fact succeed at fleshing out the specification of the game, especially when it comes to card behaviors. In addition, the actual interface through which gameplay will occur perhaps should be better defined. A realistic approach to testing the game would simply be to bring on some friends or employees or the public as beta testers and have them report bugs. When a bug report comes in, try to write an automated test case that reproduces it, and then write a fix.

5 Improving reliability through design

This dominion implementation suffers from using C primitives which require you to manage the length of lists and make choices like arbitrarily defining the highest size a deck can be in memory. The programmer's task could be simplified by pulling in at minimum an arraylist library so that adding and removing items can be simpler and so that enumerating the list can be made less error-prone.

The codebase would benefit from a simplification of the data that functions consume, in order to reduce the number of potentially bad assumptions that maintainers are prone to make. Many functions would benefit in maintainability by simply consuming data about one player instead of consuming the entire game state, and certain shared, arbitrarily repurposed fields (coins being one of the worst offenders) are a huge impediment to maintenance.