Russell Pochis
CS362 (section 001)
9 June 2015
Test report

A bug is, at its core, the violation of an explicit or implicit contract made by a unit of software. In the case of my Dominion implementation, the contracts mostly involve the effects of functions. For example, the `shuffle` method isn't supposed to create or delete cards, only move them around in an array. The main problem with contracts in general is that very often they exist only in the minds of the developers, and even if they're written down somewhere, they're easy to overlook. Holding an entire program in your mind at once is essentially impossible for all but the simplest programs. The problem is that in focusing on one part of a program, you have to put the rest of the program out of your mind, and that makes it easy to forget about what each component is supposed to do. Part of what made this Dominion implementation so difficult to work with was the complexity of the contracts between functions. The Dominion rulebook serves as an authoritative source of information about how the game is supposed to be played, but that has little relation to the intricacies of the code itself.

Debugging my Dominion implementation has greatly clarified for me the difficulty of reasoning about code with implicit contracts. For example, any Dominion player knows that you can't just trash a card whenever you want, but the nature of the C language makes it very easy to accidentally do that, and because that rule applies to the whole program rather than to any specific function, it's not documented anywhere in the code. It's simply something you always need to be thinking about when you're handling cards. Not only did I find many bugs in which cards were lost, duplicated, or otherwise mishandled, I made many of those same mistakes myself when attempting to fix other bugs. The only way I can think of to prevent future mistakes like those is to require all card movement to take place in functions dedicated to moving cards, so that programmers would have a way to abstract over the complexities of array manipulation.

A related problem is the fact that the C language makes it especially difficult to deal with some of the common objects in Dominion. The data type used to store cards is simply a signed integer, which means the language makes no attempt to ensure the validity of the cards. That's the programmer's job, and there are many places in this program in which the programmers failed to ensure that every card the program sees has a valid value. C also makes it difficult to work with collections of objects, providing no array bounds checking and no protection against uninitialized array elements. Everything the programmer is responsible for is a way for them to introduce a bug. This is the cause of a lot of the bugs I've seen in this Dominion implementation, like buffer overruns and invalid cards.

In spite of this, I was able to make progress in fixing some of the major bugs in this program, through a combination of unit and random testing. The unit tests I wrote turned out not to be terribly helpful, but the random testers were extremely useful in finding and diagnosing bugs.

The first two random testers I wrote test the Adventurer and Baron cards. Each one initializes a new game, chooses a random player, fills their hand, deck, and discard with random cards, and then plays an Adventurer, asserting various things along the way. Randomness provided an unexpected but welcome advantage over the unit tests I wrote: tests could be both longer and more varied than if I had to manually write each one. Writing unit tests is tiring, and it's difficult to think of a sufficient variety of test cases to achieve decent coverage of the system under test. While a random tester is more difficult to verify, once it's verified it can keep churning out test cases long after a human would have given up writing unit tests. Both of those testers found at least one serious bug, and had I continued developing them I have no doubt I would have found more.

The other random tester I wrote tests entire Dominion games, using a rudimentary set of strategies that keeps the games long and varied. The assertions it makes are less frequent and much less

detailed, but it's provided a lot more information than any other tester or test I wrote. Part of the reason is that when you have a lot of different tests, you're more likely to find bugs than if you just have a few similar tests. I found several bugs by looking at the logs of failing test cases, and when one test case didn't make the problem obvious, usually another test case did. The log came in especially useful because I could look through the previous few turns and check for impossible situations that could have led to the failure in question. Another reason my random tester was useful is that C programs are prone to crashing on most architectures. What C lacks in runtime safety it partially makes up for by producing fragile programs, allowing certain classes of errors to be detected merely by looking at whether the program terminates normally in a reasonable amount of time. I was able to find several bugs just by checking for crashes and infinite loops and then using a debugger to set watchpoints and check for unexpected values.

While statement coverage doesn't tell the whole story, it does give a general idea of the completeness of a test suite. By using my full-game random tester alone, I managed to cover 93% of the lines in `card.c` and `dominion.c` (about 552 of 596). Most of the lines my random tester doesn't cover have to do with invalid input, so I'm not terribly concerned about the number being less than 100%. More complex coverage metrics might give more detailed information about why those lines weren't covered, but most lines in this program are fairly simple, so coverage metric is not a major concern.

In spite of the decent test coverage, I have little confidence I found even a majority of the bugs in this implementation. Bugs slipped through my testers on many occasions before I finally found them while diagnosing other bugs, and I'm aware of several severe bugs I wasn't able to fix in time. The amount of state the program caries makes it difficult to test every aspect. Not only that, thinking of assertions to write is still something the programmer has to do, and no amount of randomness can make the computer intelligent enough to find new types of errors it hasn't been programmed to find. The first step I would take if I were to attempt further testing is to improve the number and quality of assertions. For instance, it would be good to check that cards are never created and are only destroyed when being trashed. It would also be worthwhile to check that the game is always ended at the correct time. These and other assertions would help improve my confidence in the correctness of the implementation, which is admittedly low at the moment.