Chongxian Chen
Professor Alex Groce
CS372 software engineering II
Test Report

Test on Dominion

Dominion is a very fun game, but it doesn't seem to be very easy to write from the program perspective. I download a Dominion game from app store on my iPad and start playing it. I found it a lot of fun in the beginning. I played a couple tutorials and in order to be familiar with the cards, I also googled dominion cards so I know better about the rules. And I practice them in the game. After a couple games I found that the game is heavily depends on what cards are selected for the kingdom cards. Some kingdom cards selection could be really boring while some good combination will make the game really fun.

After being familiar with the game, I start looking into the code to debug it. At first, the size of the files and the line numbers make me feel the game is really hard to debug. I start with reading the dominion.h. The header file includes a lot of comments and the basic structure of the game. The cards, are smartly used as enums so we don't have to remember what number represents what card. Then I look into the helpers functions. I then check the helpers functions in dominion.c. I decide to know their structure first.

In dominion.c, I paid particular attention to the function calls in each function. For example, in cardEffect, there are multiple calls to discardCard and gainCard. Thus I gets to know when and where to use these functions.

Getting to know the names and instantly recall what they are made for and how to use them takes a long time especially when you have more than 5 parameters in cardEffects and some other functions. But luckily the gameState structure include a lot of useful information and we can quickly check it in the dominion.h. Overall I think it is not impossible to have a good understanding of the structure.

After getting familiar with functions in the game, I start refactor some card functions so there is not a huge function. I found refactoring not too hard because I only have to pass some parameters. The only thing that needs to be properly considered when refactoring is that the function sometimes have a return value to indicate if the card is successfully played. After refactoring, I have to compare the refactor functions' return value and reflect them back to the original cardEffect function.

The next is to do unit test. The unit test is not too hard to test on especially there are just some long functions, others are short or only has one line. The unit test is very helpful I should have test all unit functions before I move on. Because these functions are called multiple times in the later when we need to test each individual cards. Without the confidence that these unit functions are correct, we will not sure where the mistake is when we test card.

This applies to Agan's principle that we should divide the code and conquer. We need to make sure the basic, small unit functions are right before we can moving on. It is much easier to debug a small chunk of code then fell into a complex logic calling all kinds of functions while you are not sure which of these might cause an error.

After the unit testing, I moved on to test the cards. It is not really easy to test the cards. But luckily we get some functions that can add a card to your hand directly. But some cards still need a complete deck to play with. For example, the adventurer will need a couple treasure cards and action cards in the deck to really make sure if the card is played properly. One other thing that makes testing card unit test hard is that you need to use a couple functions or even have an opponent to test with. For example, witch will need a opponent, thus we need to have end turn and almost a finished game to check if the curse is actually working at the end of the game.

A complete random game generator is actually easier than I expected at first. I devised a solution to make the players play smart. That is, in the first few rounds, the players in the game only buys treasure cards. And they will try a couple times to buy higher value treasure cards like gold instead of a silver when they have enough money. This process guarantees that the players will have enough money to buy those fancy, high-price action cards. In the 2nd phase of the game, the players start to buy action cards. With the same process, the players will try a couple times to buy a higher value cards before they choose to buy a low value card when they have enough money. This method makes sure they players will likely to buy all kinds of cards in the long run.

Then the players will try to find an action card and play them in my random game generator. The last phase of the game is to buy all treasure, actions, victories cards to end the game. With the proper handling in the first two phase, the game usually end around 100 rounds. I think this is a fairly good result. The cards are played evenly and I quickly get more than 60% of code coverage within just a few runs. With just more than 20 runs I even get close to 80% of the code coverage. Overall this proves that my code is relatively reliable. And I checked the line coverage of the code, most special cases are used.

I also ran all my unit tests and random game generator on my classmate Shuai Peng's game. There is only one card test that failed during the process. And then I ran the random game generator. With the scheme I designed, I quickly gain code coverage on Shuai's dominion implementation, too. Thus I conclude that Shuai has a reliable dominion implementation, too.

I then ran all my unit test and random game generator on another class Zheng Gan Zheng's game. The unit test all passed. But the card test failed 3 out of 4. The feast even goes into infinite loop. All the three random test cards failed, too. But with the random test generator, I can get to 68% of code coverage with 15 runs. Though this is not as good as my previous, but it is still not too bad. But since it fails most of the card test, I think the implementation is not very reliable.