Report on Quality of Implementation

By Tristan Bourvon, Nick Braat and Lilia Motley

Introduction

At first glance, the game seems to work perfectly. After playing around ten games, no unintentional behavior nor any errors have occurred. Every card implemented works properly and the game state gets updated nicely. Even the “show history” part functions correctly, making it even easier to check whether cards function as expected.

Implementing the Booty Bay Bodyguard with taunt was a pleasure. After looking at the existing stealth functions and cards, putting in an additional card was very straightforward. A little bit of exploring went into the valid target functions when a minion with taunt was on the board, but once that was figured out this worked really well.

Design choices made

Some functions, like *‘attack-two-or-less?’* on the *Cabal Shadow Priest* are defined in the cards definitions. This is not in line with the rest of the definitions file, where all tests and functions are only loaded. It would be better to place these functions elsewhere and just load them into the definitions.

Only one new file is added besides the firestone.core and firestone.construct, which is firestone.api. This results in pretty big construct and core files, which can be problematic in some cases. When looking for a function, sometimes it is hard to predict where to find it. However, when using an IDE like IntelliJ this does not result in unmanageable code and is therefore not a real problem in this case. The new api file caused some confusion at first, but it is basically a file which with all the functions called by the edn-api. It seems a little bit like double work, but it is a design choice that can be understood.

Function names are (almost) all self-explanatory, which makes the code in general really nice and easy to read. This also comes in handy when searching for a function: if you search for a keyword, the appropriate function will always show.

Separating out the tests for each card by putting them into a test file was a very smart idea, as it kept the code less cluttered. Said tests were also very thorough, testing for almost ten different scenarios in some cases.

Level of abstraction

The level of abstraction was appropriate for the project and made adding new functions a breeze. When implementing taunt, the valid-attack? function worked wonders. Just adding a new argument to that function took care of the rest. Getting valid-targets etc all relies on the single valid-attack? function which means changes only need to be made at one point.

When adding The Black Knight, the codebase again showed how well abstracted and simple to use it was. All that had to be done was copying the definition of a similar minion (like Big Game Hunter) and replace the target-condition (which is an easy to understand name) with *':target-condition taunted?'*. Some tests were of course added and they all passed, just as when we tested them in the actual game.

When implementing Lightwarden, it was incredibly easy to add a new kind of trigger. The function *handle-triggers* broadcasts a trigger to all minions with certain arguments. Adding a trigger receiver to a card is also very easy because the name just has to match what is used in *handle-triggers*: nothing is hardcoded there.

Tests of the engine

All functions in the engine are tested in a proper manner. Test never assume (with an exception of the create-game) that the state is being structured in a certain way, making them flexible. As far as we could tell, tests included all relevant scenarios for that function. Tests are also used to explain the function in a way, which is really nice.

Tests of the cards

Tests of the cards are separated from the definitions, which provides a nice overview. The file is only used for tests, and it keeps the definition file clean. The only downside is that you have to switch between separate files when figuring out what exactly a minions effect is, but the descriptions of the card in the definition take care of it most of the time.

The Battlecry of the card Abusive Sergeant does not seem to work.

Misplaced logic

We honestly couldn't find any major examples of misplaced logic. Everything was straightforward and readable.

Modelling of the state

The state is modeled neatly. The full state is only hardcoded once, in the create-game function. This helps to understand what you are dealing with but doesn't cause any problems on other places when the state changes. When mapping the core state to the client state, mostly easy to understand functions and definitions are used. Optional parameters are sometimes included, be it in a somewhat unclear mapping function. However, a comment above this function explain what is being done here and from

that point, it is pretty easy to change and read the function. All the mapping functions are tested against the spec, making sure no errors occur on that part.

Responsibilities of the functions

All function are pure, which means they only have one responsibility. Functions don't produce side effects when used and only do exactly what they should do. This is really nice when implementing new features using some old functions because you know what the return value will be. One exception might be *'handle-triggers'*, but since the function name is really clear about what it does it does not result in some unexpected effects.

Conclusion (would we buy for half a million?)

To conclude, we were really impressed with this code. Everything seemed to work and it looks like it scales really well. Tests are implemented everywhere and function names and comments are clear when they should be. Implementing new functionality didn't cause any problem whatsoever, and it feels like this code base has a lot more potential in it. Therefore, we would gladly advise buying this code when we would have the opportunity.