Refactoring operations - Build 3

Potential targets

1. Implemented various player behaviors
2. Moved phase execution calls
3. Spread out the logic to create game
4. Separated the creation of views and registering observer to entities.

1. Implemented various player behaviors

Refactoring technique - Replace type code by strategy

We had implementation of phases in the *Player* class, which had to vary depending upon the player's behavior. *Player* class and *GamePlay* shared the code of execution of phases. We took a step back and tried to analyse the steps which had nothing to do with the player's information. This was done to identify the actions which were only taken place on the countries.

We move the responsibility for different phases as follows:

1. Startup strategy - for a given *country* and budget of *army*, a player's behavior should fill the army count to that country and return the left over.
2. Reinforce strategy - for a given list of owned countries and *map*, a player's behavior should estimate the reinforce count and fill the reinforce army count to the owned countries.
3. Attack strategy - for a given *map* and list of owned countries, a player's behavior should chose whom to attack, how many times to attack and with whom to attack.
4. Fortify strategy - for a given *map* and list of owned countries, a player's behavior should chose the countries to move the armies between them.

With these strategies, we moved the code:

1. To update *phase*
2. To update *player*
3. Update ownership of a *country*
4. Dropping players in case lost all the countries

These responsibilities were not affected by the different *player* behaviors. Thus, we decided to divide the responsibilities over a set of objects and let them collaborate. The overall execution of phase becomes a combined effort between the *Game* class implementation and the player's behavior implementing the type of *phase*.

2. Moved phase execution calls

Refactoring technique - Move a method

Since, all the execution calls had to invoke as a method of *player* class. The following factors encouraged us to move the methods:

1. In last build's refactoring technique we moved the *currentPlayer* and *currentPphase* inside game.

2. *Map* is also accessible to *Game* class and we needed to pass *map* as a parameter to all the strategies calls.

This made us realise that their will be major *'getMap*()' calls and *GamePlay* will know too much about the *map* class.

3. Spread out the logic to create new game

Refactoring technique - Collapse hierarchy

Earlier, the *GamePlay* class used to receive the file name and count of players, thus, all the logic to create a *Map* was encapsulated. This hinder our test cases creation where we had to create the *Game* instance just to access the *Map*.

As for this build, we realised there will be test cases which will involve map to be accessed. Thus, keeping *Map* encapsulated from the *Usecase* layer was unnecessarily adding complexity.

Game now receive a Map object and players objects which can be easily created in the test cases without any file dependency.

4. Separated the creation of views and registering observer to entities

Refactoring technique - Extract method

Separated and moved code for registering views to new methods (*registerGame(), registerPlayers(), registerCountries()*). This enabled us to follow DRY principle as we had to re-initialise the views and thus, it was required for us to register these new views to the observables.