Build 3 – Refactoring Document

To pinpoint potential refactoring candidates in the build (#2), the focus was placed on the following criteria:

- Methods containing multiple sets of logic.
- Repeated use of similar logic in various locations.
- Classes with an abundance of methods.
- Extensive nesting and complex conditional logic structures.

Refactoring Targets:

1. Refactor to use Optional Instead of Null in certain methods such as findCountryByName in Airlift.java

2. Refactor command creation using the Builder pattern for clear and customizable construction of game actions in Warzone, enhancing readability and flexibility.

3. For improved error handling, consider modifying the "loadFile" method within the "loadmap" function to throw an IOException when an error occurs during file reading.

4. The "updatePlayers" method in the "playerservice" has multiple responsibilities, such as parsing a player's name, handling player-related actions (additions or removals), and updating the game state. To improve code structure, it's advisable to decompose this method into smaller, more focused functions to handle each of these tasks separately.

5. As part of the refactoring process in the "playerservice" while removing a player, consider introducing a custom exception that should be thrown when a player is not found.

6. The class variables in the "showmap" class, including `d_players`, `d_gameState`, `d_map`, `d_countries`, and `d_continents`, should be changed to private access. Additionally, it's recommended to create public getter methods to provide controlled access to these variables, ensuring encapsulation.

7. The extensive "showMap" method would benefit from decomposition into smaller, more specialized methods, each responsible for handling specific tasks.

8. Modify the "executeLoadMap" method to make use of dependency injection for the `d_mapService`. The method should no longer depend on an internally stored `d_mapService`, allowing it to accept any implementation of the `MapService` that can be provided during invocation. This change enhances flexibility and testability.

9. To adhere to the Single Responsibility Principle and improve code organization, consider breaking down the "startup" class and other similar phases into smaller, more specialized classes. Each of these new classes should be responsible for a single aspect of the game's startup phase, such as input handling, map editing, game player actions, and so on. This approach enhances code modularity and maintainability.

10. Wherever applicable, use the Java Stream API to streamline loops and operations on collections for improved readability and maintainability of your code.

Potential Refactoring Targets:

11. Strategy pattern
12. Adapter pattern
13. Extract game state specific methods including loadGame and saveGame to separate class GameService
14. Update issueOrder phase methods to randomly update whether next order is required or not for handling automatic players, as user won't play a role in this gameplay phase.
15. Based on strategy of player issueOrder method is adapted to accept new orders rather than from user input

Strategy pattern:

Before: Initially all the orders were given by the user for each player

After: There are five distinct player behaviors: Aggressive, Benevolent, Cheater, Human, and Random. These behaviors align with their described characteristics and orders are executed based on their behaviour. The logic for the previous command input has been transitioned to be in line with the actions of a human player.

Reason: To adjust for these behavior patterns, the system has been modified to cater to the specific actions and tendencies associated with each type of player.

Added Test Cases:

1. testInitialOrder – Checks whether Order creation is deploy initially. (AggressiveTest.java)
2. testDeployOnStrongestCountry – Check whether aggressive player deploys armies on strongest country. (AggressiveTest.java)
3. testDeploy - Check whether benevolent player deploy armies on weakest country. (BenevolentTest.java)
4. testAttack - Check whether benevolent player attacks to weakest neighbor. (BenevolentTest.java)

Modified Test Cases (if any): None

Adapter Pattern:

Before: The game supported only one map format (i.e. Domination map).

After: The system has been updated to support read and write operations for two map file formats: the original domination format and conquest format. To achieve this, the map loading format has been refactored to implement the adapter pattern, allowing seamless interaction and adaptation between the different map file formats.

Reason: Gives user option for multiple map formats, in addition it becomes easy to add any new map format in the existing code.

Added Test Cases:

1. testReadMapFile– This test case is used to read domination map.. (DominationMapTest.java)
2. testReadMap - This test case is used to read conquest map. (ConquestMapTest.java)
3. testAddContinent- This test case is used to add continent in conquest map. (ConquestMapTest.java)
4. testRemoveContinent- This test case is used to remove continent in a conquest map. (ConquestMapTest.java)
5. testAddCountry- This test case is used to add country in a conquest map. (ConquestMapTest.java)
6. testRemoveCountry- This test case is used to remove a country in a conquest map. (ConquestMapTest.java)

Modified Test Cases (if any): None

Extract game state specific methods including loadGame and saveGame to separate class GameService:

Before: Individual phase classes handled all the game related commands respectively.

After: Implemented a GameService class to handle certain game related methods such as loading and saving a game state.

Reason: To implement the features of load game and save game in warzone

Added Test Cases:

1. testValidSaveGame(): Tests valid savegame command.
2. testInvalidSaveGame(): Tests invalid savegame command.
3. testValidLoadGame(): Tests valid loadgame command.
4. testinvalidLoadGame(): Tests invalid loadgame command.

All tests were added in GameEngineCtxTest.java.

Modified test cases: None

Update issueOrder phase methods to randomly update whether next order is required or not for handling automatic players, as user won't play a role in this gameplay phase:

Before: User is prompted to enter yes or no whether to enter additional orders.

After: The issueOrder phase methods were updated to randomly determine if the next order was necessary for managing automatic players, removing the user's involvement in this gameplay phase.

Reason: To set a limit on the number of commands an automatic player can execute during a turn.

Added test cases: None

Modified test cases: None

Based on strategy of player issueOrder method is adapted to accept new orders rather than from user input:-

Before: In issue order phase, user was prompted to enter orders and those orders are executed respectively.

After: Issue method adapted to accept orders generated randomly from different player strategies.

Reason: To adapt to different behavioural patterns

Added test cases:

1. testReadMapFile– This test case is used to read domination map.. (DominationMapTest.java)
2. testReadMap - This test case is used to read conquest map. (ConquestMapTest.java)
3. testAddContinent- This test case is used to add continent in conquest map. (ConquestMapTest.java)
4. testRemoveContinent- This test case is used to remove continent in a conquest map. (ConquestMapTest.java)
5. testAddCountry- This test case is used to add country in a conquest map. (ConquestMapTest.java)
6. testRemoveCountry- This test case is used to remove a country in a conquest map. (ConquestMapTest.java)

Modified test cases: None