Build #2 - Refactoring Document

Potential Refactoring Targets:

- The targets for refactoring were identified based on several key issues and challenges within the codebase that needed improvement. Here's an explanation of how these targets were found:
  - Duplication of Operations
  - Inadequate Documentation and Commenting
  - Unclear Transition between Map Editing and Gameplay
  - High Coupling of Elements
  - In summary, the refactoring targets were identified by addressing issues related to code duplication, inadequate documentation, unclear state transitions, and high coupling. The goal of the refactoring process was to address these issues and improve the overall quality, maintainability, and understandability of the code.
  - A list of the potential refactoring targets:
    - **Handling commands in various locations:**
      - The commands were initially being extracted in multiple methods and validated in various places, which made it tough to understand the flow during the execution of these commands.Flow of the game.
        - Ex: The gameplayer, assigncountries and deploy commands were being handled in their respective method, and in each of these methods we were validating and extracting the command.
    - **Folder structure:**
      - All the classes were placed in a single folder, which posed challenges for code navigation, debugging, and various programming tasks due to its lack of organization and clarity.
        - Ex: All test related files were place in test folder, all the program related files were placed in the main folder. Further categorization was necessary as the complexity of the program would increase.
    - **Separate Map Elements from Gameplay Elements:**
      - The auxiliary methods were scattered all around the program, making it difficult to access them.
        - Ex: To check if the player has conquered a country, we used to check in game engine but not in the player class itself.
    - **Maintaining the state/phase of game:**
      - Transitioning from the map part of the game to the orders part of the game proved to be challenging without the correct method calls. We were able to overcome this challenge by implementing the state pattern.

    - Ex: We were not able to move from map editor part to gameplay part within the execution, we had to re run the program to access map editor if we were in gameplay and vice versa.

- **Command Pattern:**
  - Implementing the gameplay phase using the command pattern became necessary as the number of commands increased. Consolidating all the commands in one place would have led to a cluttered and confusing codebase, making it difficult to understand.
    - Ex: If we need to implement all the commands without command pattern, then we need to carry all the information along the execution. In command pattern, details regarding the execution are held by the respective command object which will make it easier to carry around the program and execute.
- **State Pattern:**
  - The implementation of the state pattern was necessary in the game to ensure that it can run in different phases. This helps to prevent players from taking actions related to one phase while they are in another, ensuring a clear separation of game phases and their associated actions.
    - Ex: We were able to implement the process of issuing orders and executing orders using a infinite loop and exit/break until the player suggests to. But there was no maintenance of what is happening after each loop. But in state pattern it will be clear regarding which state we are in.
- **Observable Pattern:**
  - Using print statements was not effective in understanding the flow of the game. Furthermore, print statements are only accessible while the program is running, and they do not provide a structured way to monitor and track what's happening at each step of the game. To address these issues, we decided to implement a method that allows us to comprehensively observe the game's progress.
  - As part of this restructuring, we separated the program into distinct components. The logic-related classes were moved to the models, handling user input and controlling the execution of models were shifted to the controllers, and the responsibility for logging and displaying valid or invalid messages was transferred to the views.
  - To achieve this organized approach, we employed the Observer Pattern. This pattern enables us to establish a robust system for tracking and managing the game's state and actions, enhancing the overall clarity and control in the game's architecture.
- **Printing/Outputting proper valid/invalid messages:**
  - We revamped the way we handle messages and output for players during the game. It was imperative to establish a structured and organized approach to managing messages and outputs to ensure clarity and effectiveness.

    - Ex: Printing a message which says “Invalid Input” would be very ambiguous, instead if there is a little information like “Cannot deploy to <country> as the no. of armies is more than what the player has.” using which the user can correct the input and run the game which would be a better implementation.

- **Handling exceptions:**
  - As the complexity of the program is increasing, an increase in occurrences of exceptions is expected. So handling of any sort of exception is needed.
    - Ex: If the user tries to load a map which isn’t there in the game, then the user would receive an FileIOException, which exits from the game. Instead if we handle it and let the user know the issue in a simple language, it would be better.
- **Implementing of more and better accessors/mutators:**
  - Due to the ambiguity of data types and data structures being used, it is crucial to implement proper accessors and mutators.
    - Ex: If we are searching for a player, based on the player’s name, it is better to have a method which will return the player object or name instead of having to search the list of players wherever necessary.
- **Implementing of constants class:**
  - To store all globally used constants, including strings, characters, and numbers that are compared to user input and should remain constant throughout the execution. As this practice enhances code readability and maintainability, doing this would be a great benefit.
    - Ex: Instead of using hard coded values, like no of default armies or command names, it is better to store them as a constant. When the value changes we need to only change the value of the constant, need not update everywhere there is a usage.
- **Naming Convention:**
  - To improve code readability and understandability, it's essential to use meaningful variable and method names. Descriptive names make it easier for developers (including yourself) to comprehend and work with the code effectively.
    - Ex: Having a descriptive names such as numArmiesToDeploy, numArmiesToAdvance will be more readable than just having numArmies.
- **Javadoc:**
  - Due to the above refactoring conditions, the Javadoc does not match the classes, methods, and variable names. So refactoring the java docs was necessary.
    - Ex: As the player class has more methods which are not needed in Build 1, we need to update the Javadoc to describe all the new functionalities added.
- **Commenting:**
  - To enhance code visibility and readability as the code structure changes, it's important to improve the implementation of comments.

- Ex: Previously we had game engine, which had all the logic of gameplay phase. Now to understand where we are when we are reading a statement, the comment would help in letting the user know what input we have and what we are trying to process.

Actual Refactoring Targets:

- List of the five Actual refactoring targets:
  - **Command Pattern:**
    - Tests Added:
      - performNegotiateCard()
      - performAdvanceCommand()
      - performDeployCommand()
      - performBombCard()
      - performBlockadeCard()
      - performAirliftCommand()
    - Tests Modified: None
    - Reason:
      - Maintaining one class per order is advantageous because it allows us to treat each order as an object. Each class can encapsulate all the details and information related to a specific order, making it easy to store and manage. These objects can later be used for execution or processing, as they contain comprehensive information about the order.
    - Before:
      - The driver method housed all the necessary methods since there was only a single deployment order. These methods were executed in response to user input. First they were added into player's order list via issue_order() and then executed one by one via next_order().
    - After:
      - Transitioning between game phases has been streamlined, ensuring ease of movement while also enforcing necessary restrictions. For example, once players set up and start the game, returning to the setup phase is restricted. After the setup phase, order issuance and execution are only allowed and performed until players exit or a player wins.
  - Observable Pattern:
    - Tests Added: None (The required log files are being created, and output is generated properly).
    - Tests Modified: None
    - Reason:

- Implementation of MVC Structure in the program would help in reduction of complexity of the program, here the Model will contain the business logic, View will help in showing the status of the game to the players and log the status of the game. And also maintaining of logs and printing proper and meaningful messages on console is a good practice. Due to this debugging is also simpler.

- Before:
    - All the files were in a single folder. Printing of messages to the console was not that meaningful and also not in a consistent manner.
- After:
    - Comprehensive logging of all events during the game's runtime has been implemented, significantly improving the clarity and comprehensibility of console output.

- **State Pattern:**
    - Tests Added:
        - JUnitTestSuite{}
    - Tests Modified:None
    - Reason:
        - The execution process has distinct phases: the startup phase, during which only map functions, player addition, and country assignment are valid; the order issuance phase, where processing and storing of player orders is valid; and the order execution phase, where each player's orders are executed one by one. Each functionality is limited to its respective phase.
    - Before:
        - The whole gameplay was being handled by the game engine class. Moving from map editor section to gameplay section was difficult as we had to rerun the program and then choose the phases.
    - After:
        - Each order is associated with a distinct class that implements the Order interface. This interface encompasses all the essential methods required for storing order details, performing validation, and executing the order. By mandating that the classes implementing the Order interface use these methods, consistency in handling orders is maintained.
- **Handling Commands:**
    - Tests Added:
        - testPerformDeploy()
        - testPerformAdvance()
        - testBombCard()
        - testBlockadeCard()
        - testAirliftCard()
        - testNegotiateCard()

- Tests Modified:None
- Reason:
  - Implementing the handling of all commands in one place would be a better approach since all orders/commands need to be managed effectively, and consolidating them in a single location improves code readability.
- Before:
  - Handling of command in multiple places and validating in multiple places.
- After:
  - Handling, validating and extracting of commands in a single place.

- **Map Validation:**
  - Tests Added:
    - testMap()
    - testLoadMap()
    - testMapValidity()
  - Tests Modified: None
  - Reason:
    - Ensuring the map's validity before starting a game or saving it is crucial, as it contributes to a smooth gameplay experience.
  - Before:
    - A game can be started or a map can be saved with limited map validation.
  - After:
    - We have introduced various functionalities to validate maps.