SOEN 6441 - Advanced Programming Practices

Refactoring Document

Project - Risk Game (Build #2)

Fall 2023

Submitted by:-

Rishi Ravikumar
Anuja Ajay Somthankar
Yusuke Ishii
Nimisha Mavjibhai Jadav
Abhigyan Singh

Submitted to:-

Dr. Amin Ranj Bar

Potential Refactoring Targets

Below is the list of potential refactoring targets from the Build 1 of the Risk game project.

1. Command validation for invalid option types and param length - This task is motivated by the fact that we need to ensure commands are used correctly to avoid runtime errors.
2. Command Validation for invalid game phase commands - It is critical to ensure the integrity of game logic and to identify invalid game phase commands in order to avoid unexpected behavior.
3. Adding constants to command types - Strings can make code less readable and more prone to errors. Examining the code for hard-coded values can help you identify these constants.
4. Consolidate loadMap() into sub-functions - We discovered that the loadMap() function is overly complex, making it difficult to understand and maintain.
5. Removing dead code - We identified this by seeing code that appears to have no effect on the program's behavior.
6. Updating the access modifiers of the methods - This was discovered to ensure that methods have appropriate levels of visibility in order to maintain encapsulation and control access to specific functionalities.
7. Updating unclear variable names - This was found by identified variable names that were not descriptive or didn't accurately represent the purpose of the variable.
8. Add com.org to packages - We identified this in order to follow best practices in package naming convention.
9. Consolidate inputParser() into private sub-functions - The function can be simplified and be made more modular by dividing it into private sub-functions.
10. Adding constants for the valid commands - This was identified to improve the code readability and reduce the likelihood of errors caused by directly using command strings in the code.
11. Consolidating repeated if conditions - Repeated conditional statements that could be consolidated into a single location, reducing redundancy and making the code more concise.
12. Replacing if..else..if with switch..case - This target was chosen to improve code readability and maintainability in situations where a switch statement would be preferable to a series of if-else statements.
13. Adding common set up functions for uniform tests - There was a need to standardize the test setup process in order to make it more consistent and efficient.
14. Simplifying methods in the GameManager class - Methods in the GameManager class could be made more concise and focused on specific responsibilities, thus improving readability and maintainability.

15. Composing methods in the util package - The util package's methods could be organized or structured in a more logical and cohesive manner, which can improve code organization and readability.

Reasons for choosing the actual refactoring targets over others

Because they address critical aspects of code quality, maintainability, and functionality, the five refactoring tasks chosen were most likely prioritized over the others:

1. *Command validation for invalid option types and param length* - It is critical to ensure that commands are validated correctly in order to avoid unexpected errors or code misuse.
2. *Command Validation for invalid game phase commands* - This is critical for maintaining the game logic's correctness and reliability and preventing unintended behavior.
3. *Adding constants to command types* - Introducing constants to command types improves code readability and reduces the risk of errors due to magic numbers or strings.
4. *Consolidate loadMap() into sub-functions* - Breaking down a complex function into smaller sub-functions makes the code more modular, understandable, and maintainable.
5. *Removing dead code* - Eliminating dead code is critical for reducing code bloat, improving readability, and avoiding confusion about the actual functionality of the code.

These tasks are likely to be prioritized because they have a direct impact on the correctness, maintainability, and efficiency of the codebase, all of which are critical aspects of code quality.

Actual Refactoring Targets

Below are the 5 refactoring targets chosen from the above mentioned list, mainly because of the new requirements established in Build 2.

1. Command validation for invalid option types and param length

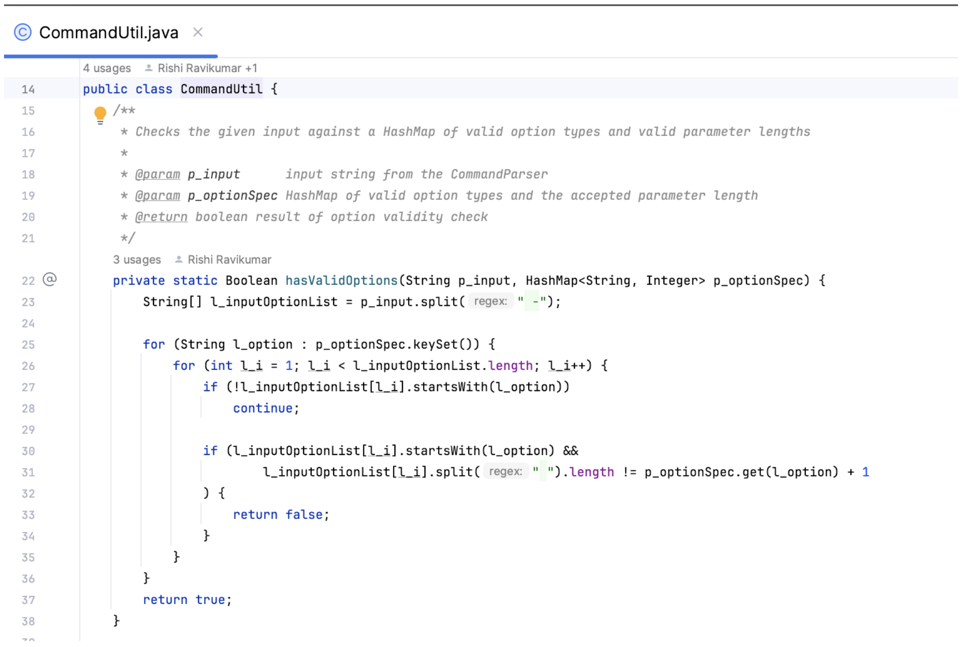
Before:

The code lacks comprehensive validation for command options and parameters in the pre-refactoring scenario. This means that the program may accept invalid options or parameters if proper checks are not performed. As a result, it may cause unexpected behavior or errors, potentially jeopardizing the application's integrity. The codebase may contain numerous ad-hoc validation checks scattered throughout it, making it difficult to maintain and prone to inconsistencies.

After:

In preparation for the addition of new commands and features to our software, it was critical to implement a set of predefined methods to ensure that our validation mechanisms could adapt and accommodate these changes seamlessly. These methods were created to improve validation functionality by allowing us to check the following items when processing user commands:

- Invalid options passed - The preset methods allow us to determine whether or not the options passed in a command are valid. This includes ensuring that the command contains recognized options and that they follow the expected format.
- Invalid number of options args for a particular option - The preset methods allow you to validate whether the user-supplied arguments match the requirements of a specific option. This includes ensuring that the correct number of arguments are provided, as well as that they are of the expected type and format.

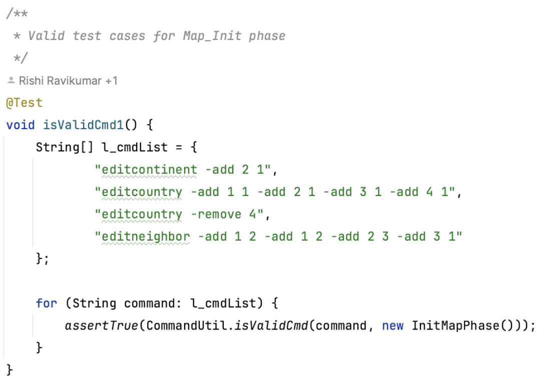


CommandUtil.java

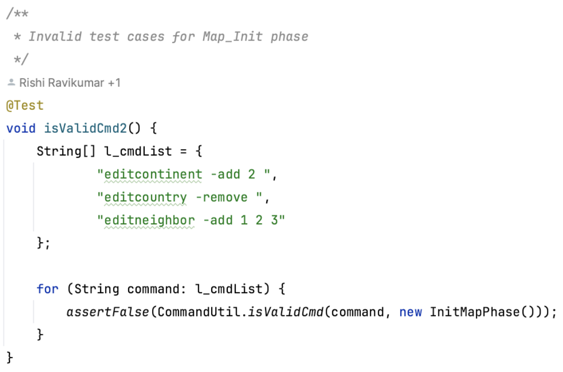
```java
4 usages  Rishi Ravikumar +1
public class CommandUtil {
    /**
     * Checks the given input against a HashMap of valid option types and valid parameter lengths
     *
     * @param p_input      input string from the CommandParser
     * @param p_optionSpec HashMap of valid option types and the accepted parameter length
     * @return boolean result of option validity check
     */
    3 usages  Rishi Ravikumar
    private static Boolean hasValidOptions(String p_input, HashMap<String, Integer> p_optionSpec) {
        String[] l_inputOptionList = p_input.split( regex: " -");

        for (String l_option : p_optionSpec.keySet()) {
            for (int l_i = 1; l_i < l_inputOptionList.length; l_i++) {
                if (!l_inputOptionList[l_i].startsWith(l_option))
                    continue;

                if (l_inputOptionList[l_i].startsWith(l_option) &&
                        l_inputOptionList[l_i].split( regex: " ").length != p_optionSpec.get(l_option) + 1
                ) {
                    return false;
                }
            }
        }
        return true;
    }
```

After Refactoring

Test cases:



```java
/**
 * Valid test cases for Map_Init phase
 */
Rishi Ravikumar +1
@Test
void isValidCmd1() {
    String[] l_cmdList = {
            "editcontinent -add 2 1",
            "editcountry -add 1 1 -add 2 1 -add 3 1 -add 4 1",
            "editcountry -remove 4",
            "editneighbor -add 1 2 -add 1 2 -add 2 3 -add 3 1"
    };

    for (String command: l_cmdList) {
        assertTrue(CommandUtil.isValidCmd(command, new InitMapPhase()));
    }
}
```



```java
/**
 * Invalid test cases for Map_Init phase
 */
Rishi Ravikumar +1
@Test
void isValidCmd2() {
    String[] l_cmdList = {
            "editcontinent -add 2 ",
            "editcountry -remove ",
            "editneighbor -add 1 2 3"
    };

    for (String command: l_cmdList) {
        assertFalse(CommandUtil.isValidCmd(command, new InitMapPhase()));
    }
}
```

The following is included in unit tests for the refactored code:

@Test isValidCmd1(): Because "editcontinent -add 2 1" is a valid command, it should still return true."editcountry -add 1 1 -add 2 1 -add 3 1 -add 4 1" should still return true because it is a valid command. "editcountry -remove 4" should also return true because it is a valid command.As a valid command, "editneighbor -add 1 2 -add 1 2 -add 2 3 -add 3 1" should still return true.

@Test isValidCmd2(): "editcontinent -add 2 " should still return false because it has an incomplete option and lacks the necessary parameters. "editcountry -remove " should still return false because it lacks the necessary parameters. "editneighbor -add 1 2 3" should still return false because it has an excessive number of parameters.

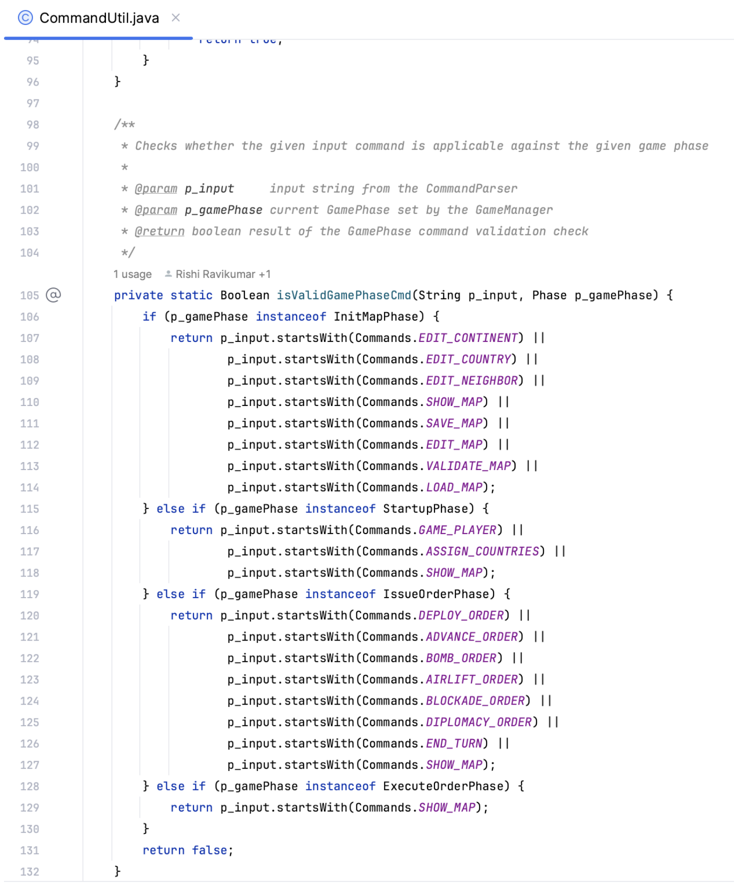
2. Command Validation for invalid game phase commands

Before:

The code in the pre-refactoring scenario lacks proper validation for game phase commands. It makes no distinction between valid and invalid commands in relation to the current game phase. As a result, the application may accept or process commands that are inappropriate for the current game phase. This can result in inconsistencies, errors, or unexpected behavior, jeopardizing the game's logic and user experience.

After:

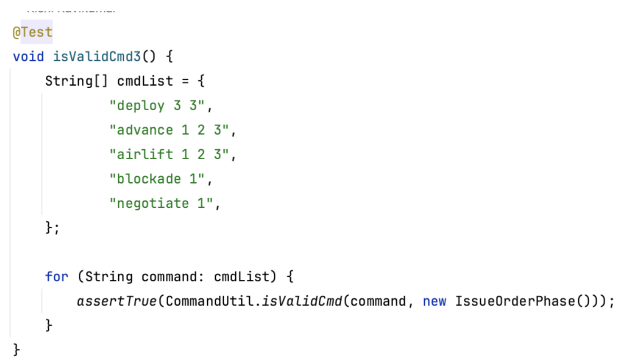
We recognized the importance of implementing a robust set of checks to validate each command within the context of the current game phase in anticipation of the upcoming addition of new commands to our software. To maintain order and consistency, it's crucial to validate whether a given command is appropriate and valid for the current game phase. This means ensuring that the command is relevant and permissible in the specific context of the ongoing gameplay.


CommandUtil.java

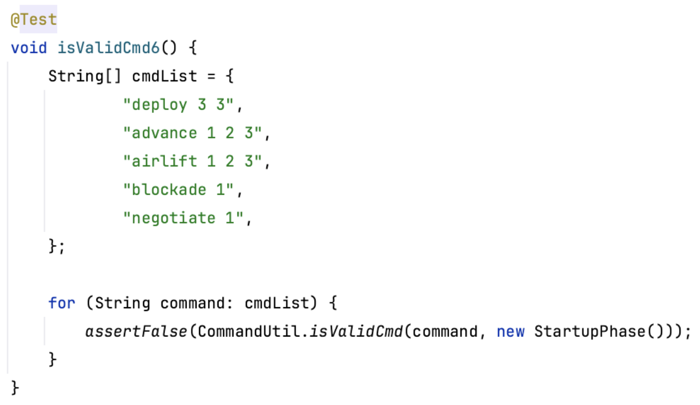
```java
        }
    }

    /**
     * Checks whether the given input command is applicable against the given game phase
     *
     * @param p_input     input string from the CommandParser
     * @param p_gamePhase current GamePhase set by the GameManager
     * @return boolean result of the GamePhase command validation check
     */
    1 usage   Rishi Ravikumar +1
    private static Boolean isValidGamePhaseCmd(String p_input, Phase p_gamePhase) {
        if (p_gamePhase instanceof InitMapPhase) {
            return p_input.startsWith(Commands.EDIT_CONTINENT) ||
                    p_input.startsWith(Commands.EDIT_COUNTRY) ||
                    p_input.startsWith(Commands.EDIT_NEIGHBOR) ||
                    p_input.startsWith(Commands.SHOW_MAP) ||
                    p_input.startsWith(Commands.SAVE_MAP) ||
                    p_input.startsWith(Commands.EDIT_MAP) ||
                    p_input.startsWith(Commands.VALIDATE_MAP) ||
                    p_input.startsWith(Commands.LOAD_MAP);
        } else if (p_gamePhase instanceof StartupPhase) {
            return p_input.startsWith(Commands.GAME_PLAYER) ||
                    p_input.startsWith(Commands.ASSIGN_COUNTRIES) ||
                    p_input.startsWith(Commands.SHOW_MAP);
        } else if (p_gamePhase instanceof IssueOrderPhase) {
            return p_input.startsWith(Commands.DEPLOY_ORDER) ||
                    p_input.startsWith(Commands.ADVANCE_ORDER) ||
                    p_input.startsWith(Commands.BOMB_ORDER) ||
                    p_input.startsWith(Commands.AIRLIFT_ORDER) ||
                    p_input.startsWith(Commands.BLOCKADE_ORDER) ||
                    p_input.startsWith(Commands.DIPLOMACY_ORDER) ||
                    p_input.startsWith(Commands.END_TURN) ||
                    p_input.startsWith(Commands.SHOW_MAP);
        } else if (p_gamePhase instanceof ExecuteOrderPhase) {
            return p_input.startsWith(Commands.SHOW_MAP);
        }
        return false;
    }
```

After Refactoring

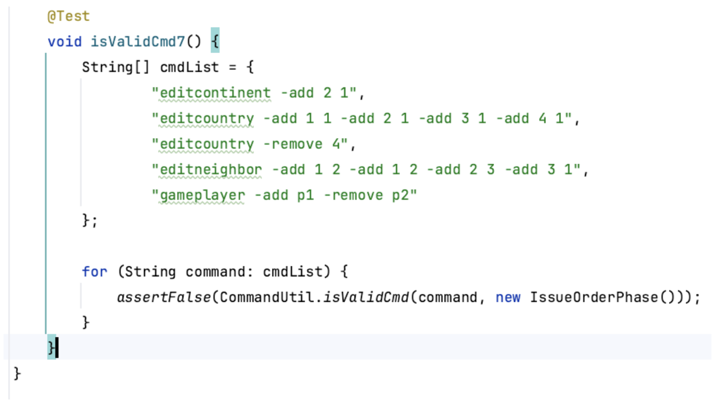
Test Cases:



```java
@Test
void isValidCmd3() {
    String[] cmdList = {
            "deploy 3 3",
            "advance 1 2 3",
            "airlift 1 2 3",
            "blockade 1",
            "negotiate 1",
    };

    for (String command: cmdList) {
        assertTrue(CommandUtil.isValidCmd(command, new IssueOrderPhase()));
    }
}
```



```java
@Test
void isValidCmd6() {
    String[] cmdList = {
            "deploy 3 3",
            "advance 1 2 3",
            "airlift 1 2 3",
            "blockade 1",
            "negotiate 1",
    };

    for (String command: cmdList) {
        assertFalse(CommandUtil.isValidCmd(command, new StartupPhase()));
    }
}
```



```java
    @Test
    void isValidCmd7() {
        String[] cmdList = {
                "editcontinent -add 2 1",
                "editcountry -add 1 1 -add 2 1 -add 3 1 -add 4 1",
                "editcountry -remove 4",
                "editneighbor -add 1 2 -add 1 2 -add 2 3 -add 3 1",
                "gameplayer -add p1 -remove p2"
        };

        for (String command: cmdList) {
            assertFalse(CommandUtil.isValidCmd(command, new IssueOrderPhase()));
        }
    }
}
```

The following is included in unit tests for the refactored code:

@Test isValidCmd3(): validates valid commands for the "IssueOrderPhase". It anticipates that the following commands will return true: "deploy 3 3", "advance 1 2 3", "airlift 1 2 3", "blockade 1", and "negotiate 1".

@Test isValidCmd6(): contains invalid test cases for various commands in the "InitMap Phase" and expects them to return false in the "StartupPhase."

@Test isValidCmd7(): returns false due to invalid commands for 'IssueOrder' phase.

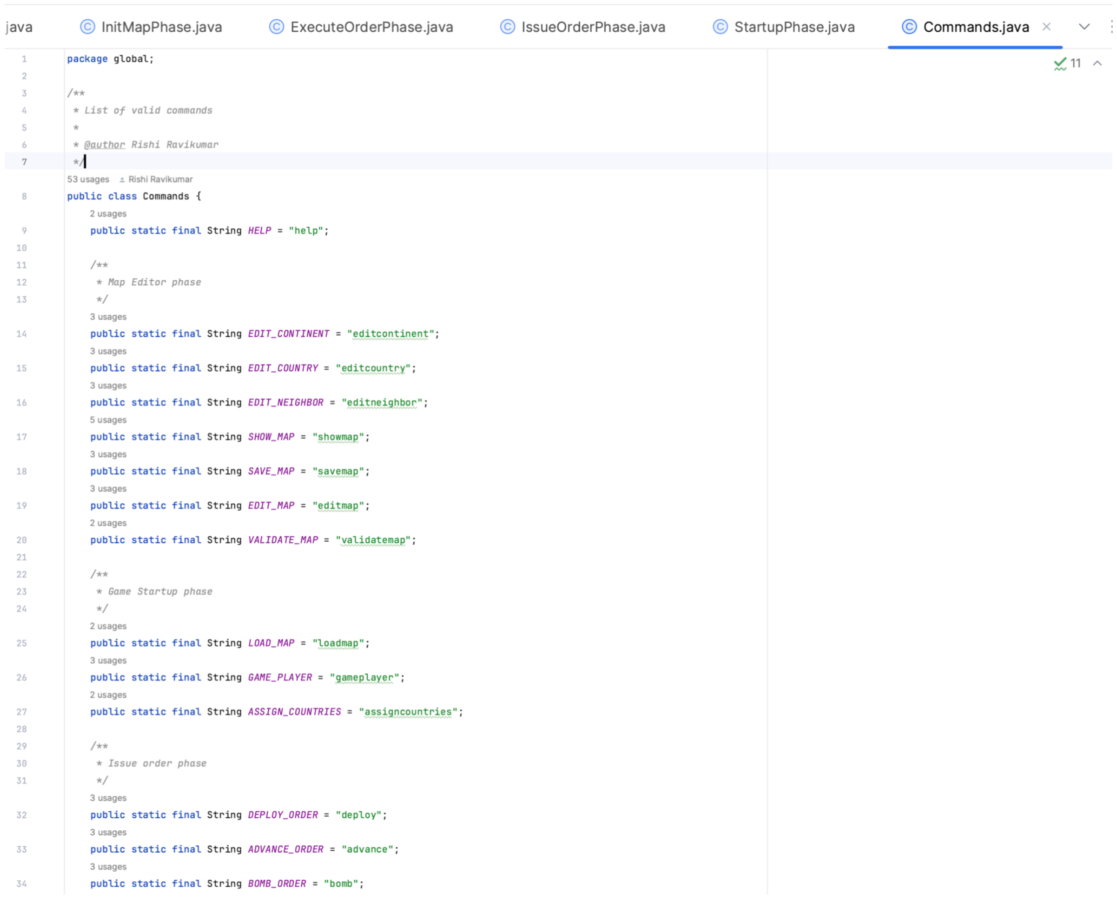
3. Adding constants for command types

Before:

The code in the pre-refactoring scenario is based on hard-coded command types scattered throughout the codebase. Without any centralization or clear naming conventions, command types are represented as string literals. Because of the lack of constants, the code is less readable and maintainable. It also raises the possibility of errors due to typos or inconsistent command types.

After:

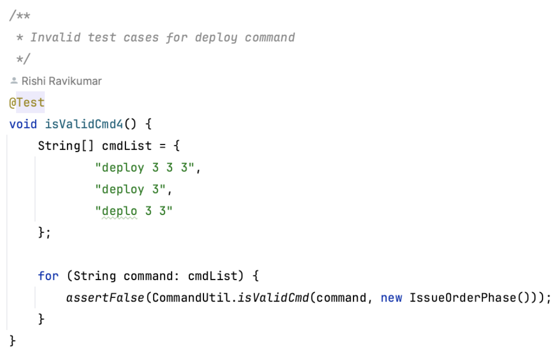
The codebase has been improved as a result of the refactoring by introducing constants for command types. Instead of using raw string literals or numerical values, the code now employs well-named and well-defined constants. This modification improves code readability and maintainability, lowering the likelihood of errors and making it easier to understand the purpose of various command types. Constants improve code consistency and clarity, making it more robust and developer-friendly. Overall, the refactoring produces a more organized and trustworthy codebase.



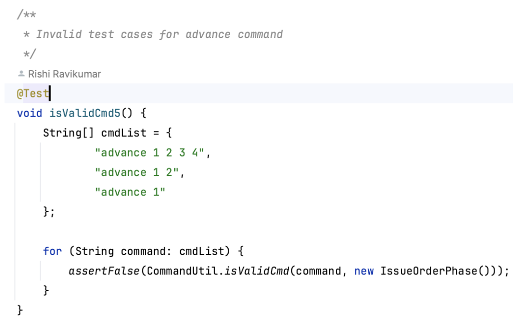
```
package global;

/**
 * List of valid commands
 *
 * @author Rishi Ravikumar
 */
public class Commands {
    public static final String HELP = "help";

    /**
     * Map Editor phase
     */
    public static final String EDIT_CONTINENT = "editcontinent";
    public static final String EDIT_COUNTRY = "editcountry";
    public static final String EDIT_NEIGHBOR = "editneighbor";
    public static final String SHOW_MAP = "showmap";
    public static final String SAVE_MAP = "savemap";
    public static final String EDIT_MAP = "editmap";
    public static final String VALIDATE_MAP = "validatemap";

    /**
     * Game Startup phase
     */
    public static final String LOAD_MAP = "loadmap";
    public static final String GAME_PLAYER = "gameplayer";
    public static final String ASSIGN_COUNTRIES = "assigncountries";

    /**
     * Issue order phase
     */
    public static final String DEPLOY_ORDER = "deploy";
    public static final String ADVANCE_ORDER = "advance";
    public static final String BOMB_ORDER = "bomb";
```

After Refactoring

Test cases:



```
/**
 * Invalid test cases for deploy command
 */
@Test
void isValidCmd4() {
    String[] cmdList = {
            "deploy 3 3 3",
            "deploy 3",
            "deplo 3 3"
    };

    for (String command: cmdList) {
        assertFalse(CommandUtil.isValidCmd(command, new IssueOrderPhase()));
    }
}
```



```
/**
 * Invalid test cases for advance command
 */
@Test
void isValidCmd5() {
    String[] cmdList = {
            "advance 1 2 3 4",
            "advance 1 2",
            "advance 1"
    };

    for (String command: cmdList) {
        assertFalse(CommandUtil.isValidCmd(command, new IssueOrderPhase()));
    }
}
```

The following is included in unit tests for the refactored code:

@Test isValidCmd4(): Returns false as invalid commands.

@Test isValidCmd5(): Because it has too many parameters, "advance 1 2 3 4" returns false.Because it has an incorrect number of parameters, "advance 1 2" returns false. Because it lacks the required number of parameters, "advance 1" returns false.

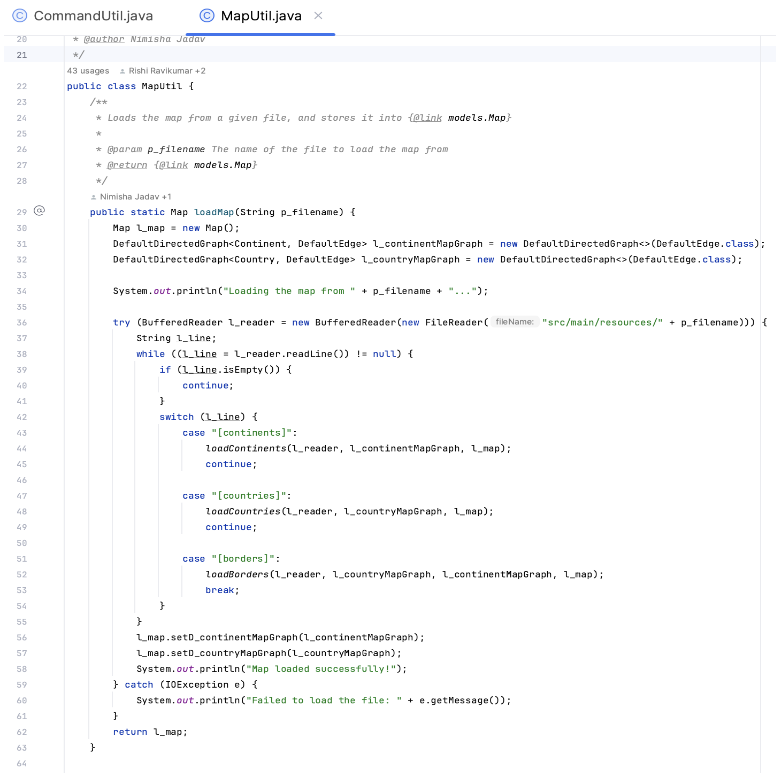
4. Consolidate loadMap() into sub-functions

Before:

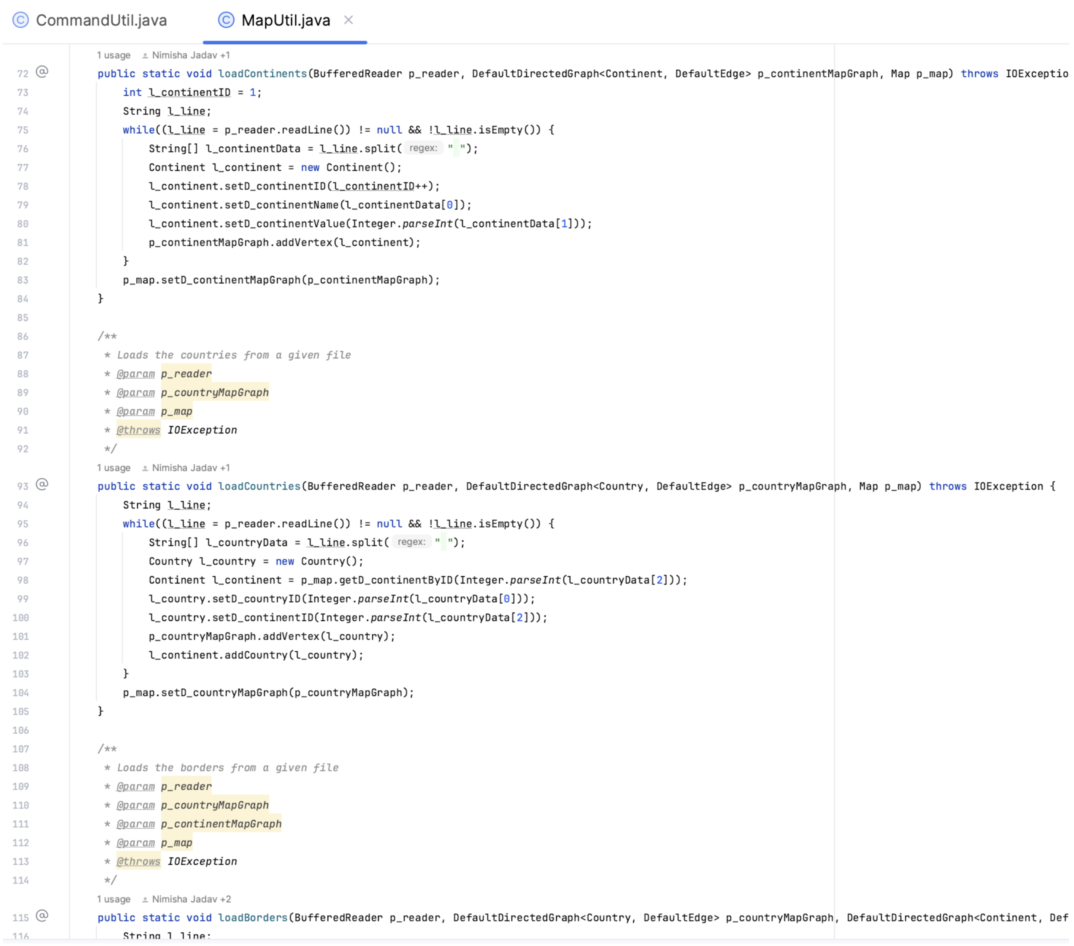
The loadMap() function in the pre-refactoring scenario is a monolithic and complex piece of code responsible for loading a map within the application. It is disorganized and difficult to understand. This single function contains all of the map-loading logic, making it difficult to maintain, extend, or debug. The absence of sub-functions leads to code duplication and decreases code reusability.

After:

Refactoring code into smaller, more organized sub-functions has several advantages. For starters, it improves code readability and maintainability, making it easier for developers to comprehend, modify, and debug code. Second, it encourages code reusability because these smaller functions can be used in different parts of the program. It also improves code modularity, allowing for more efficient collaboration between development teams. Smaller functions also help with testing because they can be tested for correctness individually, resulting in higher code quality. Overall, breaking down code into smaller sub-functions results in a more organized, efficient, and robust codebase, saving time and lowering the risk of errors during development and maintenance.



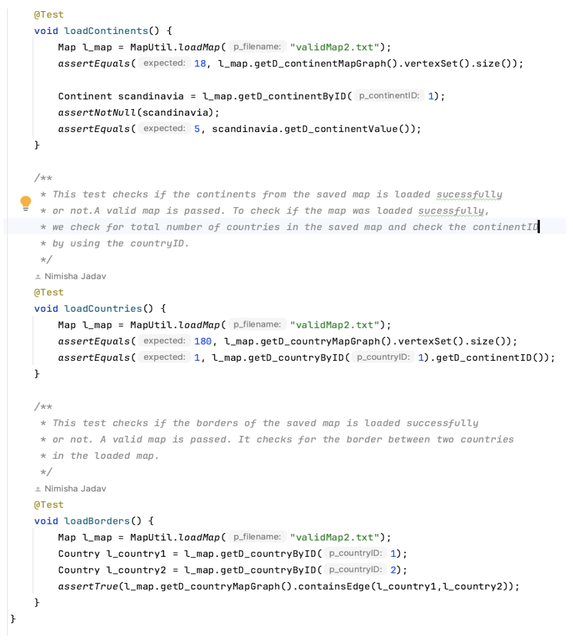
```java
 * @author Nimisha Jadav
 */
43 usages  Rishi Ravikumar +2
public class MapUtil {
    /**
     * Loads the map from a given file, and stores it into {@link models.Map}
     *
     * @param p_filename The name of the file to load the map from
     * @return {@link models.Map}
     */
    Nimisha Jadav +1
    public static Map loadMap(String p_filename) {
        Map l_map = new Map();
        DefaultDirectedGraph<Continent, DefaultEdge> l_continentMapGraph = new DefaultDirectedGraph<>(DefaultEdge.class);
        DefaultDirectedGraph<Country, DefaultEdge> l_countryMapGraph = new DefaultDirectedGraph<>(DefaultEdge.class);

        System.out.println("Loading the map from " + p_filename + "...");

        try (BufferedReader l_reader = new BufferedReader(new FileReader( fileName: "src/main/resources/" + p_filename))) {
            String l_line;
            while ((l_line = l_reader.readLine()) != null) {
                if (l_line.isEmpty()) {
                    continue;
                }
                switch (l_line) {
                    case "[continents]":
                        loadContinents(l_reader, l_continentMapGraph, l_map);
                        continue;

                    case "[countries]":
                        loadCountries(l_reader, l_countryMapGraph, l_map);
                        continue;

                    case "[borders]":
                        loadBorders(l_reader, l_countryMapGraph, l_continentMapGraph, l_map);
                        break;
                }
            }
            l_map.setD_continentMapGraph(l_continentMapGraph);
            l_map.setD_countryMapGraph(l_countryMapGraph);
            System.out.println("Map loaded successfully!");
        } catch (IOException e) {
            System.out.println("Failed to load the file: " + e.getMessage());
        }
        return l_map;
    }
```



```java
1 usage  Nimisha Jadav +1
public static void loadContinents(BufferedReader p_reader, DefaultDirectedGraph<Continent, DefaultEdge> p_continentMapGraph, Map p_map) throws IOExceptio
    int l_continentID = 1;
    String l_line;
    while((l_line = p_reader.readLine()) != null && !l_line.isEmpty()) {
        String[] l_continentData = l_line.split( regex: " ");
        Continent l_continent = new Continent();
        l_continent.setD_continentID(l_continentID++);
        l_continent.setD_continentName(l_continentData[0]);
        l_continent.setD_continentValue(Integer.parseInt(l_continentData[1]));
        p_continentMapGraph.addVertex(l_continent);
    }
    p_map.setD_continentMapGraph(p_continentMapGraph);
}

/**
 * Loads the countries from a given file
 * @param p_reader
 * @param p_countryMapGraph
 * @param p_map
 * @throws IOException
 */
1 usage  Nimisha Jadav +1
public static void loadCountries(BufferedReader p_reader, DefaultDirectedGraph<Country, DefaultEdge> p_countryMapGraph, Map p_map) throws IOException {
    String l_line;
    while((l_line = p_reader.readLine()) != null && !l_line.isEmpty()) {
        String[] l_countryData = l_line.split( regex: " ");
        Country l_country = new Country();
        Continent l_continent = p_map.getD_continentByID(Integer.parseInt(l_countryData[2]));
        l_country.setD_countryID(Integer.parseInt(l_countryData[0]));
        l_country.setD_continentID(Integer.parseInt(l_countryData[2]));
        p_countryMapGraph.addVertex(l_country);
        l_continent.addCountry(l_country);
    }
    p_map.setD_countryMapGraph(p_countryMapGraph);
}

/**
 * Loads the borders from a given file
 * @param p_reader
 * @param p_countryMapGraph
 * @param p_continentMapGraph
 * @param p_map
 * @throws IOException
 */
1 usage  Nimisha Jadav +2
public static void loadBorders(BufferedReader p_reader, DefaultDirectedGraph<Country, DefaultEdge> p_countryMapGraph, DefaultDirectedGraph<Continent, Def
    String l_line;
```

After Refactoring

Test cases:



```java
    @Test
    void loadContinents() {
        Map l_map = MapUtil.loadMap( p_filename: "validMap2.txt");
        assertEquals( expected: 18, l_map.getD_continentMapGraph().vertexSet().size());

        Continent scandinavia = l_map.getD_continentByID( p_continentID: 1);
        assertNotNull(scandinavia);
        assertEquals( expected: 5, scandinavia.getD_continentValue());
    }

    /**
     * This test checks if the continents from the saved map is loaded sucessfully
     * or not.A valid map is passed. To check if the map was loaded sucessfully,
     * we check for total number of countries in the saved map and check the continentID
     * by using the countryID.
     */
    Nimisha Jadav
    @Test
    void loadCountries() {
        Map l_map = MapUtil.loadMap( p_filename: "validMap2.txt");
        assertEquals( expected: 180, l_map.getD_countryMapGraph().vertexSet().size());
        assertEquals( expected: 1, l_map.getD_countryByID( p_countryID: 1).getD_continentID());
    }

    /**
     * This test checks if the borders of the saved map is loaded successfully
     * or not. A valid map is passed. It checks for the border between two countries
     * in the loaded map.
     */
    Nimisha Jadav
    @Test
    void loadBorders() {
        Map l_map = MapUtil.loadMap( p_filename: "validMap2.txt");
        Country l_country1 = l_map.getD_countryByID( p_countryID: 1);
        Country l_country2 = l_map.getD_countryByID( p_countryID: 2);
        assertTrue(l_map.getD_countryMapGraph().containsEdge(l_country1,l_country2));
    }
}
```

The following is included in unit tests for the refactored code:

@Test loadContinents(): This test case determines whether the continents from the saved map were successfully loaded. It checks the number of continents in the loaded map as well as continent-specific attributes.

@Test loadCountries(): This test case checks the total number of countries and their associations with continents to ensure that countries from the saved map are correctly loaded.

@Test loadBorders(): This test ensures that the borders between countries in the loaded map are correctly established.

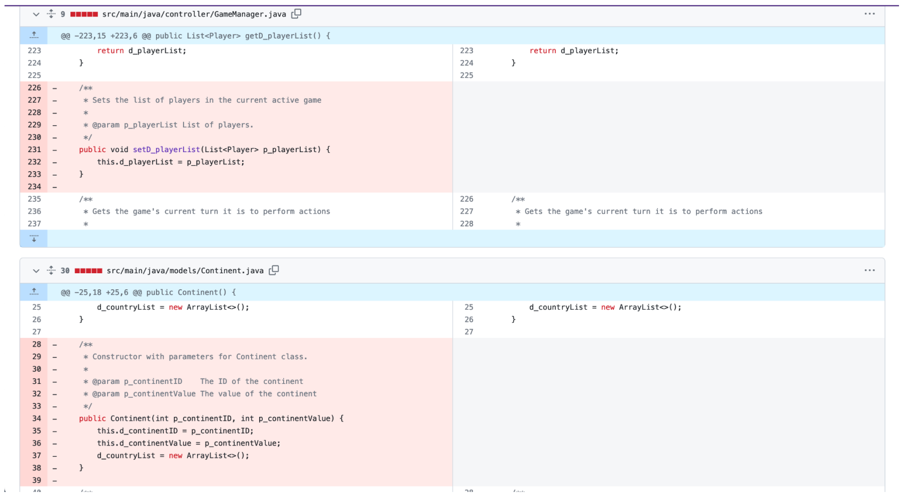
5. Removing dead code

Before:

The codebase contains sections of dead code in the pre-refactoring scenario, which are code segments that are no longer used or executed but still exist in the source files. This dead code clogs the codebase, making it more difficult to read and understand. It can also cause confusion about the program's actual functionality and add to the maintenance burden.

After:

Dead code must be removed during refactoring for several reasons. For starters, it improves code clarity and reduces confusion for developers who might otherwise be perplexed by the purpose and functionality of unused code segments. It also improves code maintainability because unused code can be a source of errors and unintended consequences if overlooked during updates or modifications. Furthermore, removing dead code can result in smaller, more efficient executables, reducing the memory footprint of the software and improving performance. In summary, removing dead code improves code quality, maintainability, and resource efficiency, resulting in a more manageable and reliable codebase.



```diff
9 src/main/java/controller/GameManager.java
@@ -223,15 +223,6 @@ public List<Player> getD_playerList() {
         return d_playerList;
     }

-    /**
-     * Sets the list of players in the current active game
-     *
-     * @param p_playerList List of players.
-     */
-    public void setD_playerList(List<Player> p_playerList) {
-        this.d_playerList = p_playerList;
-    }
-
     /**
      * Gets the game's current turn it is to perform actions
      *
```

```diff
30 src/main/java/models/Continent.java
@@ -25,18 +25,6 @@ public Continent() {
         d_countryList = new ArrayList<>();
     }

-    /**
-     * Constructor with parameters for Continent class.
-     *
-     * @param p_continentID    The ID of the continent
-     * @param p_continentValue The value of the continent
-     */
-    public Continent(int p_continentID, int p_continentValue) {
-        this.d_continentID = p_continentID;
-        this.d_continentValue = p_continentValue;
-        d_countryList = new ArrayList<>();
-    }
-
```

Removed dead code from controller and models package and the existing test cases still executes successfully after the removal of the dead code.