SOEN 6441 Advanced Programming Practices

REFACTORING DOCUMENTATION

Group W10 – Build 2

Group members:

- Omnia Alam
- Yajing Liu
- Sherwyn Dsouza
- Darlene Nazareth
- Duy Thanh Phan
- Md Tazin Morshed Shad

Potential Refactoring Targets

The list below compiles the potential refactoring targets. They are primarily taken from the new requirements of Build 2 and problematic issues arose when developing for Build 1.

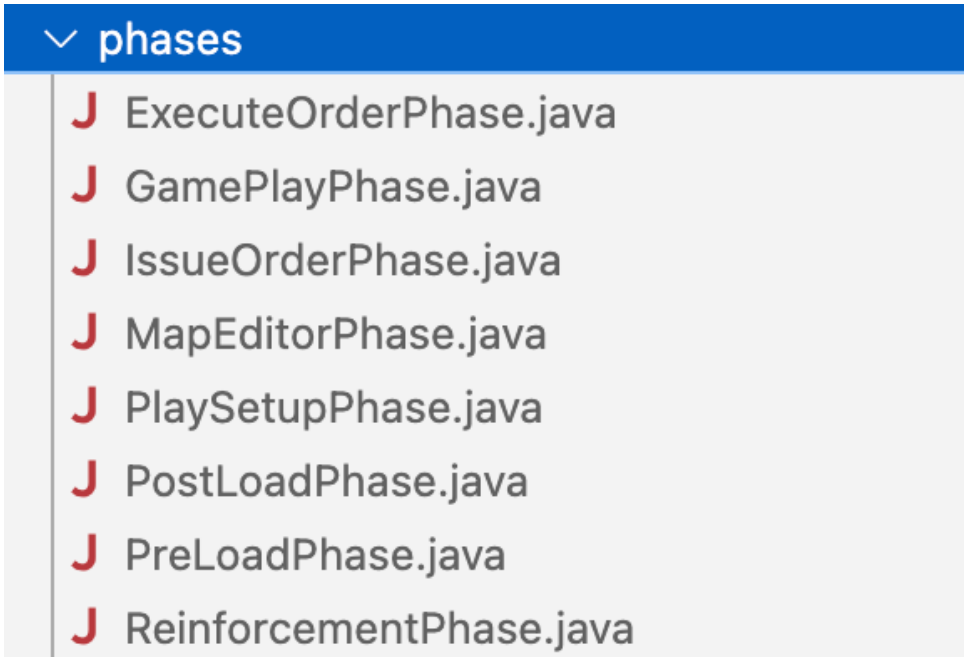
1. Implement the State pattern for the application. Specifically in Map Editor and Game Play (Startup, Issue order, and Order execution)
2. Implement the Command pattern for Orders.
3. Implement the Observer pattern for state change logging besides using the default console logger.
4. Display proper messages for incorrect commands.
5. Improve exception handling for incorrect commands and illegal states in the application.
6. Improve naming conventions for classes, functions and variables.
7. Improve the project folder structure to align better with the MVC structure and support maintainability.
8. Add additional test cases for the existing code base.
9. Remove unused imports, functions and variables.
10. Improve Javadoc content.
11. Reorganize Constants to be separated by responsibilities.
12. Use modern, recommended Java syntax to replace some existing code snippets.
13. Convert all commands to enums.
14. Named functions descriptively and to ensure Single Responsibility Principle.
15. On saving map, a player is able to move forward in the game phase.
16. Move validation for orders from Player class to different concrete implementations of Orders

Actual Refactoring Targets

1. Implement State pattern for changing between phases

State pattern Refactoring scenarios:

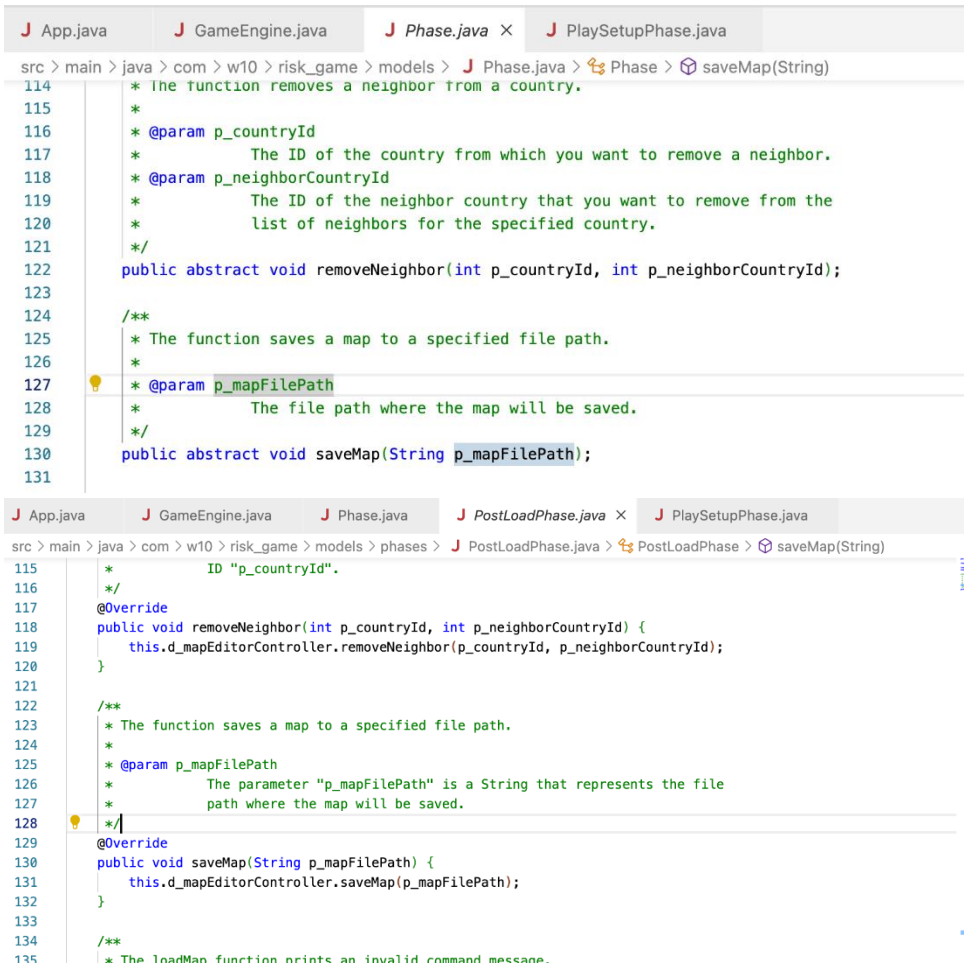
1. Renamed `GameEngineUI` to `GameEngine`.
2. Split `GameEngine` to `MapEditorController` and `GameEngineController`.
3. `MapEditorController` contains logic related to Map.
4. In order to apply State Pattern, we had to add a “phases” folder that contains all the different phases:



All phases of Risk Game

5. To use the state pattern, we had to refactor the App and GameEngine classes to invoke the phases.

```
switch (l_mainCommand) {
    // Map editor d_phase commands
    case Constants.USER_INPUT_COMMAND_LOADMAP :
        String[] l_mapName = l_argList[1].split(regex:"/");
        d_logger.log(Constants.CLI_LOAD_MAP + l_mapName[l_mapName.length - 1]);
        this.d_phase.loadMap(l_argList[1]);
        break;
    case Constants.USER_INPUT_COMMAND_SAVEMAP :
        this.d_phase.saveMap(l_argList[1]);
        break;
    case Constants.USER_INPUT_COMMAND_SHOWMAP :
        d_logger.log(Constants.CLI_SHOW_MAP);
        this.d_phase.showMap();
        break;
```



```
App.java | GameEngine.java | Phase.java | PlaySetupPhase.java
src > main > java > com > w10 > risk_game > models > Phase.java > Phase > saveMap(String)
114     * The function removes a neighbor from a country.
115     *
116     * @param p_countryId
117     *            The ID of the country from which you want to remove a neighbor.
118     * @param p_neighborCountryId
119     *            The ID of the neighbor country that you want to remove from the
120     *            list of neighbors for the specified country.
121     */
122    public abstract void removeNeighbor(int p_countryId, int p_neighborCountryId);
123
124    /**
125     * The function saves a map to a specified file path.
126     *
127     * @param p_mapFilePath
128     *            The file path where the map will be saved.
129     */
130    public abstract void saveMap(String p_mapFilePath);
131
```

```
App.java | GameEngine.java | Phase.java | PostLoadPhase.java | PlaySetupPhase.java
src > main > java > com > w10 > risk_game > models > phases > PostLoadPhase.java > PostLoadPhase > saveMap(String)
115     *            ID "p_countryId".
116     */
117    @Override
118    public void removeNeighbor(int p_countryId, int p_neighborCountryId) {
119        this.d_mapEditorController.removeNeighbor(p_countryId, p_neighborCountryId);
120    }
121
122    /**
123     * The function saves a map to a specified file path.
124     *
125     * @param p_mapFilePath
126     *            The parameter "p_mapFilePath" is a String that represents the file
127     *            path where the map will be saved.
128     */
129    @Override
130    public void saveMap(String p_mapFilePath) {
131        this.d_mapEditorController.saveMap(p_mapFilePath);
132    }
133
134    /**
135     * The loadMap function prints an invalid command message.
```

6. We have also refactored our `GameEngineTest` to `MapEditorControllerTest` and `GamePlayControllerTest`.

2. Use Command pattern to implement the Orders

We only have the deploy order in the previous build. Thus, we have the following code structures before refactoring operation:

1. Player class has function `issueOrder()` to create orders
2. Order class is a concrete class having function `execute()` to implement the logic of executing deploy order.
3. `GameEngine` gets the player's orders from players' using the `next_order()` method, then executes the orders by calling the `execute()` method of the order

This code structure worked well in build 1. However, it is not suitable for build 2 since we need to import more concrete orders in this build. Thus, using the Command pattern is necessary to handle these new types of orders in this build. We have the following code structures after the refactoring operation:

1. Player class has function `issueOrder()` to create orders. This `issueOrder()` function allows users to issue different types of valid concrete orders based on users' inputs.

2. Order class is changed to an abstract class. This class has an abstract function `execute()`.
3. Concrete order classes (deploy, advance, bomb, blockade, airlift and diplomacy) extend this abstract Order class and implement concrete `execute()` functions.
4. `GameEngine` gets the player's orders from players' using the `next_order()` method, reorganizes these orders in the sequence deploy -> airlift -> diplomacy -> others, and then executes these orders by calling the `execute()` method of the orders

All the tests that apply to the class involved in the refactoring operation:

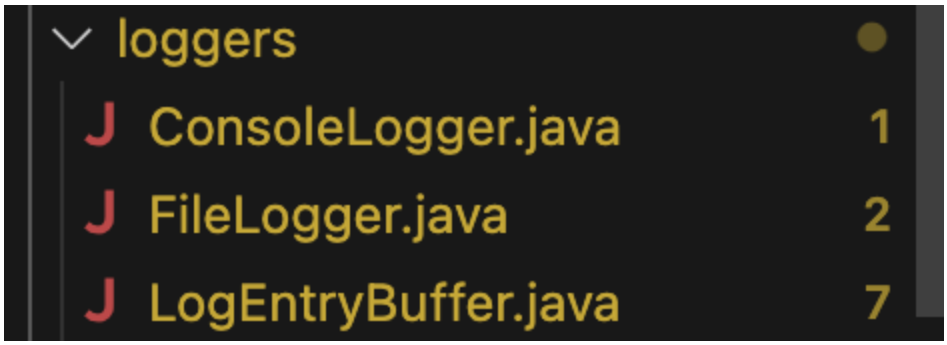
1. `PlayerTest class: issueDeployOrderTest, issueAdvanceOrderTest, issueBombOrderTest, issueBlockadeOrderTest, issueAirliftOrderTest, issueDiplomacyOrderTest`
2. `DeployTest class: executeTest, checkValidDeployInputTest, checkValidCountryTest, checkValidArmyTest`
3. `AdvanceTest class: testSameOwnerTransfer, testBattleWon, testBattleLost, testDeployAfterBattleWon`
4. `BombTest class: testBombNeutralCountry, testBombEnemyCountry, testCountryOwnership`
5. `BlockadeTest class: testReinforceOwnCountry, testReinforceNeutralCountry, testReinforceEnemyCountry`
6. `AirliftTest class: testAirliftExecution`
7. `NegotiateTest class: executeTest, validateOrderTest`

3. Use the Observer pattern to implement the file logger and console logger

The decision to undertake this refactoring was driven by multiple factors. Not only was it a prerequisite for build 2, but it also significantly contributed to the overall ease of maintaining, enhancing, and testing the application. The primary focus of this refactoring effort centered around the handling of log information. Specifically, it involved the implementation of the `ConsoleLogger` and `FileLogger` as Observers and `LogEntryBuffer` as an Observable. This enables the `LogEntryBuffer` to write to both the console and a designated text file, named "`log.txt`" simultaneously. Prior to this refactoring, we have made use of Java's *System.out.print()* to always to log to the console for build 1.

All the tests that apply to the class involved in the refactoring operation:

1. `LoggerTest class: testLogging`

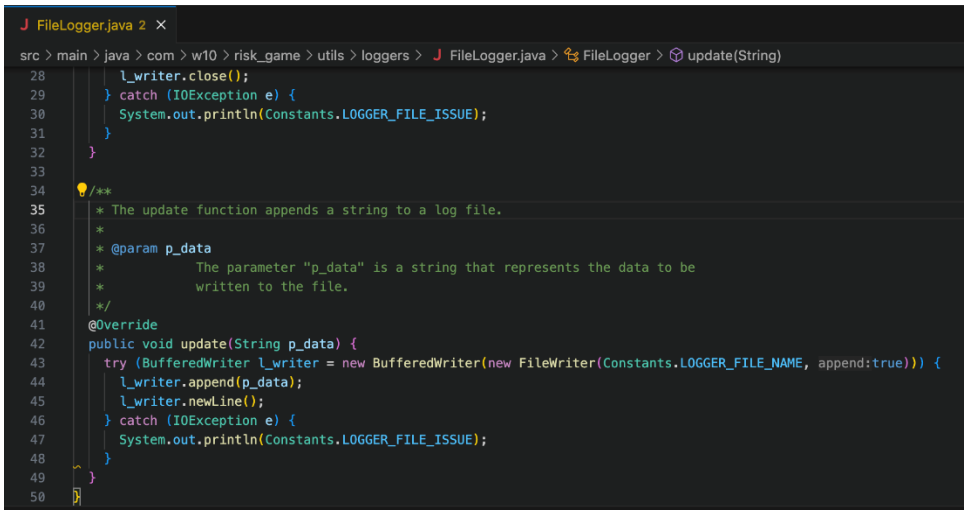


ConsoleLogger, FileLogger & LogEntryBuffer classes

```java
public class ConsoleLogger implements Observer {

  /**
   * The update function prints the given data.
   *
   * @param p_data
   *            The parameter "p_data" is a String that represents the data that
   *            needs to be updated.
   */
  @Override
  public void update(String p_data) {
    System.out.println(p_data);
  }
}
```

ConsoleLogger class implementing the Observer Class



FileLogger class

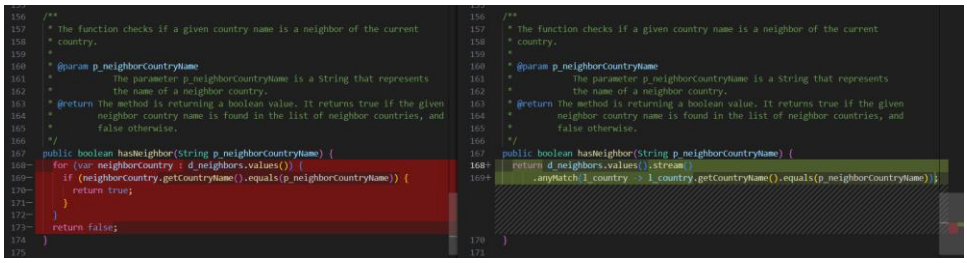
```
// Display Player Cards
if (!l_player.getPlayerCards().isEmpty()) {
  Logger.log(MessageFormat.format(Constants.SHOW_PLAYER_CARDS,
      Joiner.on(separator:", ").join(l_player.getPlayerCards())));
} else {
  Logger.log(Constants.SHOW_PLAYER_CARDS_EMPTY);
}
```

Example of the Logger logging to both file and console simultaneously

4. Use modern, recommended Java syntax to replace some existing code snippets.

Replaced Loops with Stream API:

Utilized the Java Stream API to simplify and make code that processes collections more readable.



Used Enumerations:

Replaced game constants with enums to improve type safety and code readability.

```
package com.w10.risk_game.commands;

import com.w10.risk_game.utils.Constants;

// The code snippet is defining an enum called `Command`.
public enum Command {

  loadmap(Constants.USER_INPUT_COMMAND_LOADMAP), showmap(Constants.USER_INPUT_COMMAND_SHOWMAP),
  editmap(Constants.USER_INPUT_COMMAND_EDITMAP), savemap(Constants.USER_INPUT_COMMAND_SAVEMAP),
  validatemap(Constants.USER_INPUT_COMMAND_VALIDATEMAP),editcontinent(Constants.USER_INPUT_COMMAND_EDIT_CONTINENT),
  next(Constants.USER_INPUT_COMMAND_OPTION_NEXTPHASE), editcountry(Constants.USER_INPUT_COMMAND_EDIT_COUNTRY),
  editneighbor(Constants.CLI_GAME_PLAYER_REMOVE),gameplayer(Constants.USER_INPUT_COMMAND_GAMEPLAYER),
  assigncountries(Constants.CLI_ASSIGN_COUNTRIES),deploy(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_DEPLOY),
  advance(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_ADVANCE),bomb(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_BOMB),
  blockade(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_BLOCKADE),airlift(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_AIRLIFT),
  negotiate(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_NEGOTIATE),commit(Constants.USER_INPUT_ISSUE_ORDER_COMMAND_COMMIT),
  none(commandName:""),quit(Constants.USER_INPUT_COMMAND_QUIT);

  private final String name;

  Command(String commandName) {
    this.name = commandName;
  }

  /**
   * The getName() function returns the name of an object.
   *
   * @return The method is returning the value of the variable "name".
   */
  public String getName() {
    return name;
  }
}
```

Eliminate Dead Code:

Removed unused classes, methods, and variables to reduce code clutter and improve code maintainability.

Replaced some for loops with enhanced loops:

We used modern syntax to increase readability.

```
158  * countries.
159  *
160  * @param p_CountryId
161  *            An integer representing the ID of a country.
162  * @return The method is returning a boolean value.
163  */
164 public boolean hasCountry(int p_CountryId) {
165     if (d_countriesOwned == null) {
166         return false;
167     }
168     for (int l_country = 0; l_country < d_countriesOwned.size(); l_country++) {
169         if (d_countriesOwned.get(l_country).getCountryId() == p_CountryId) {
170             return true;
171         }
172     }
173     return false;
174 }
```

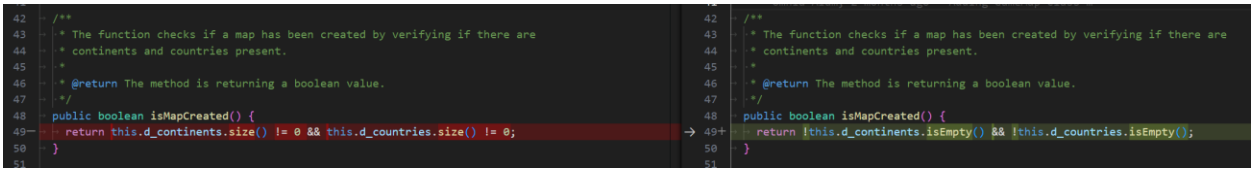
Before

```
/**
 * The function checks if a given country ID exists in a list of owned
 * countries.
 *
 * @param p_CountryId
 *            An integer representing the ID of a country.
 * @return The method is returning a boolean value.
 */
2 usages  Omnia *
public boolean hasCountry(int p_CountryId) {
    if (d_countriesOwned == null) {
        return false;
    }
    for (Country country : d_countriesOwned) {
        if (country.getCountryId() == p_CountryId) {
            return true;
        }
    }
    return false;
}
```

After

Use recommended syntax for empty list checking:

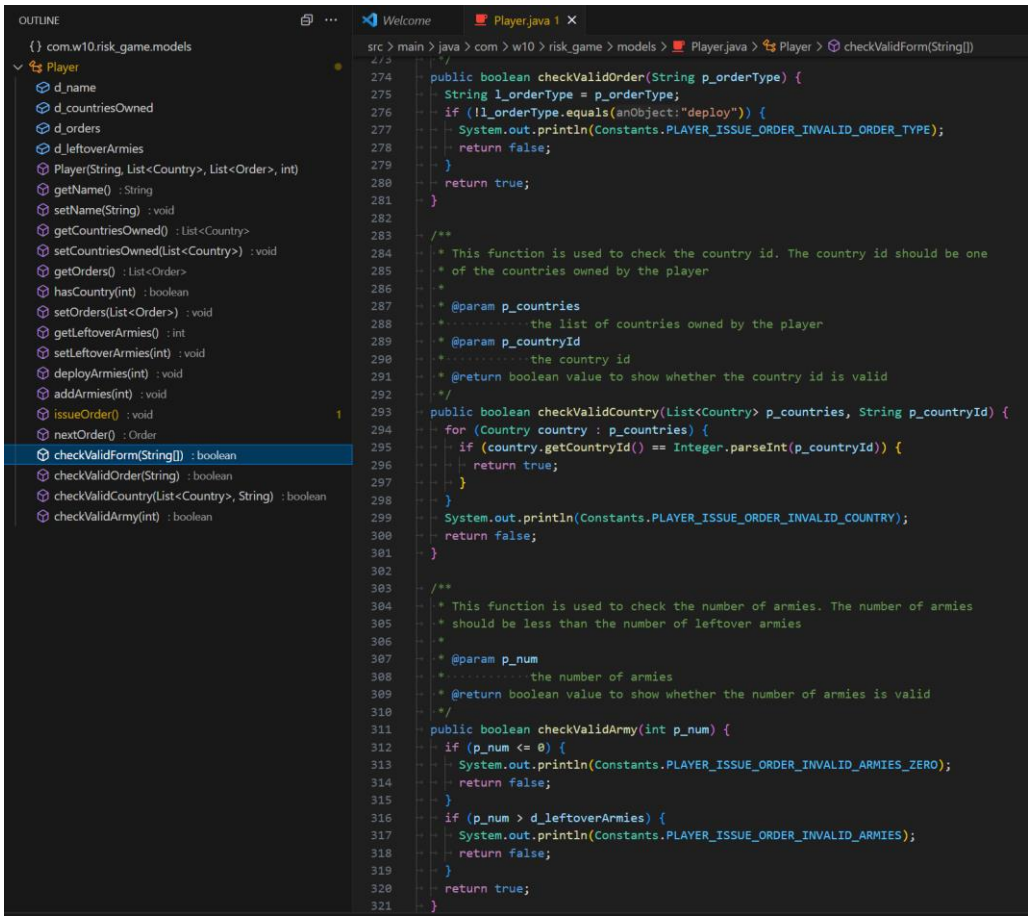
We used the correct syntax for empty list checking as it's shorter and recommended by various linters.



5. Move validation for orders from Player class to different concrete implementation of Orders

In the old implementation, we included the method to check the validation of user's input for deploying in `Player` class since we only had one order at that time. However, we support more types of orders in build 2. If we still contain all input check methods for all kinds of orders in `Player` class, it will be clogging up the `Player` class. Thus, in build 2, we performed a refactor to move those methods to corresponding concrete order classes. For example, we moved the `checkValidForm` method inside the Player class in build 1 to the `CheckValidDeployInput` method of the Deploy class in build 2. The test involved is `checkValidDeployInputTest.`

Old implementation – Build 1



New implementation – Build 2