

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.

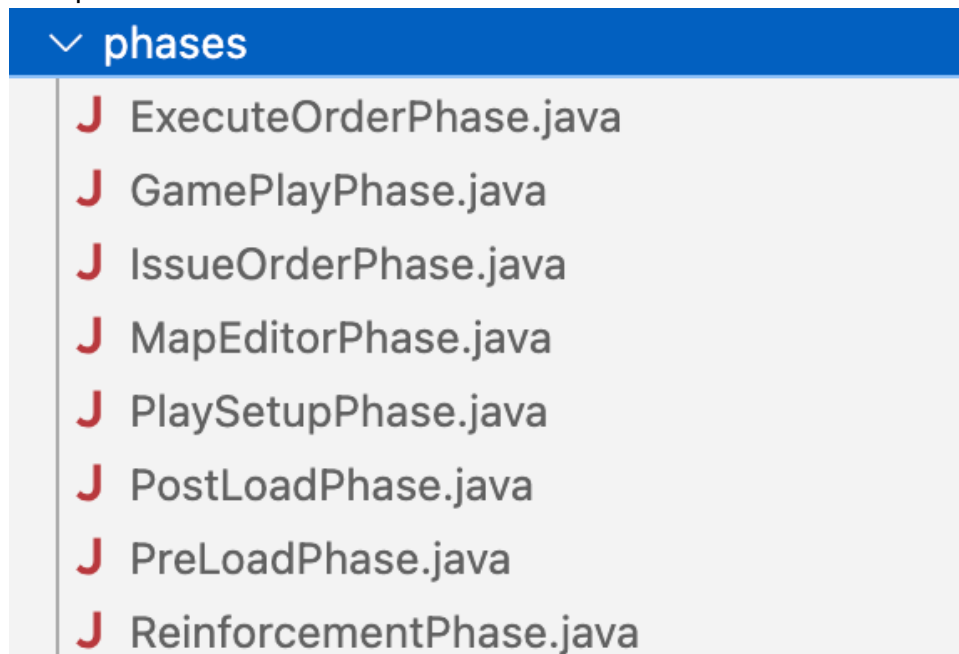
1. Implement the State pattern for the application. Specifically in Map Editor and Game Play (Startup, Issue order, and Order execution)
2. Implement the Command pattern for Orders.
3. Implement the Observer pattern for state change logging besides using the default console logger.
4. Display proper 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 (_mainCommand) {
    // Map editor d_phase commands
    case Constants.USER_INPUT_COMMAND_LOADMAP :
        String[] _mapName = _argList[1].split(regex:"/");
        d_logger.log(Constants.CLI_LOAD_MAP + _mapName[_mapName.length - 1]);
        this.d_phase.loadMap(_argList[1]);
        break;
    case Constants.USER_INPUT_COMMAND_SAVEMAP :
        this.d_phase.saveMap(_argList[1]);
        break;
    case Constants.USER_INPUT_COMMAND_SHOWMAP :
        d_logger.log(Constants.CLI_SHOW_MAP);
        this.d_phase.showMap();
        break;
    // Constants.USER_INPUT_COMMAND_EXITMAP :
}
```

```

J App.java J GameEngine.java J Phase.java X J PlaySetupPhase.java
src > main > java > com > w10 > risk_game > models > J 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

J App.java J GameEngine.java J Phase.java J PostLoadPhase.java X J PlaySetupPhase.java
src > main > java > com > w10 > risk_game > models > phases > J 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 log to the console for build 1.

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

1. LoggerTest class: `testLogging`

▼ loggers	
J ConsoleLogger.java	1
J FileLogger.java	2
J LogEntryBuffer.java	7

ConsoleLogger, FileLogger & LogEntryBuffer classes

```

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

```

J FileLogger.java 2 x
src > main > java > com > w10 > risk_game > utils > loggers > J FileLogger.java > FileLogger > update(String)
28     _l_writer.close();
29 } catch (IOException e) {
30     System.out.println(Constants.LOGGER_FILE_ISSUE);
31 }
32 }
33
34 /**
35  * The update function appends a string to a log file.
36  *
37  * @param p_data
38  *     The parameter "p_data" is a string that represents the data to be
39  *     written to the file.
40  */
41 @Override
42 public void update(String p_data) {
43     try (BufferedWriter _l_writer = new BufferedWriter(new FileWriter(Constants.LOGGER_FILE_NAME, append:true))) {
44         _l_writer.append(p_data);
45         _l_writer.newLine();
46     } catch (IOException e) {
47         System.out.println(Constants.LOGGER_FILE_ISSUE);
48     }
49 }
50

```

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.

```
156 /**
157  * The function checks if a given country name is a neighbor of the current
158  * country.
159  *
160  * @param p_neighborCountryName
161  *     The parameter p_neighborCountryName is a String that represents
162  *     the name of a neighbor country.
163  * @return The method is returning a boolean value. It returns true if the given
164  *     neighbor country name is found in the list of neighbor countries, and
165  *     false otherwise.
166  */
167 public boolean hasNeighbor(String p_neighborCountryName) {
168     for (var neighborCountry : d_neighbors.values()) {
169         if (neighborCountry.getCountryName().equals(p_neighborCountryName)) {
170             return true;
171         }
172     }
173     return false;
174 }
175
```

```
156 /**
157  * The function checks if a given country name is a neighbor of the current
158  * country.
159  *
160  * @param p_neighborCountryName
161  *     The parameter p_neighborCountryName is a String that represents
162  *     the name of a neighbor country.
163  * @return The method is returning a boolean value. It returns true if the given
164  *     neighbor country name is found in the list of neighbor countries, and
165  *     false otherwise.
166  */
167 public boolean hasNeighbor(String p_neighborCountryName) {
168     return d_neighbors.values().stream()
169         .anyMatch(l_country -> l_country.getCountryName().equals(p_neighborCountryName));
170 }
171
```

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  /**
159   * The function checks if a given country id exists in a list of owned
160   * countries.
161   *
162   * @param p_CountryId
163   * @return An integer representing the ID of a country.
164   * @return The method is returning a boolean value.
165   */
166  public boolean hasCountry(int p_CountryId) {
167      if (d_countriesOwned == null) {
168          return false;
169      }
170      for (int l_country = 0; l_country < d_countriesOwned.size(); l_country++) {
171          if (d_countriesOwned.get(l_country).getCountryId() == p_CountryId) {
172              return true;
173          }
174      }
175      return false;
176  }

```

Before

```

/**
 * The function checks if a given country ID exists in a list of owned
 * countries.
 *
 * @param p_CountryId
 * @return 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.

```

42  /**
43   * The function checks if a map has been created by verifying if there are
44   * continents and countries present.
45   *
46   * @return The method is returning a boolean value.
47   */
48  public boolean isMapCreated() {
49      return this.d_continents.size() != 0 && this.d_countries.size() != 0;
50  }
51
42  /**
43   * The function checks if a map has been created by verifying if there are
44   * continents and countries present.
45   *
46   * @return The method is returning a boolean value.
47   */
48  public boolean isMapCreated() {
49      return !this.d_continents.isEmpty() && !this.d_countries.isEmpty();
50  }
51

```

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

```

OUTLINE
{} com.w10.risk_game.models
  Player
    d_name
    d_countriesOwned
    d_orders
    d_leftoverArmies
    Player(String, List<Country>, List<Order>, int)
    getName() : String
    setName(String) : void
    getCountriesOwned() : List<Country>
    setCountriesOwned(List<Country>) : void
    getOrders() : List<Order>
    hasCountry(int) : boolean
    setOrders(List<Order>) : void
    getLeftoverArmies() : int
    setLeftoverArmies(int) : void
    deployArmies(int) : void
    addArmies(int) : void
    issueOrder() : void
    nextOrder() : Order
    checkValidForm(String[]) : boolean
    checkValidOrder(String) : boolean
    checkValidCountry(List<Country>, String) : boolean
    checkValidArmy(int) : boolean

src > main > java > com > w10 > risk_game > models > Player.java > Player > checkValidForm(String[])
274  /**
275   * This function is used to check the country id. The country id should be one
276   * of the countries owned by the player
277   *
278   * @param p_orderType
279   * @return boolean value to show whether the country id is valid
280   */
281  public boolean checkValidOrder(String p_orderType) {
282      String l_orderType = p_orderType;
283      if (!l_orderType.equals(anObject:"deploy")) {
284          System.out.println(Constants.PLAYER_ISSUE_ORDER_INVALID_ORDER_TYPE);
285          return false;
286      }
287      return true;
288  }
289
290  /**
291   * This function is used to check the country id. The country id should be one
292   * of the countries owned by the player
293   *
294   * @param p_countries
295   * @param p_countryId
296   * @return boolean value to show whether the country id is valid
297   */
298  public boolean checkValidCountry(List<Country> p_countries, String p_countryId) {
299      for (Country country : p_countries) {
300          if (country.getId() == Integer.parseInt(p_countryId)) {
301              return true;
302          }
303      }
304      System.out.println(Constants.PLAYER_ISSUE_ORDER_INVALID_COUNTRY);
305      return false;
306  }
307
308  /**
309   * This function is used to check the number of armies. The number of armies
310   * should be less than the number of leftover armies
311   *
312   * @param p_num
313   * @return boolean value to show whether the number of armies is valid
314   */
315  public boolean checkValidArmy(int p_num) {
316      if (p_num <= 0) {
317          System.out.println(Constants.PLAYER_ISSUE_ORDER_INVALID_ARMIES_ZERO);
318          return false;
319      }
320      if (p_num > d_leftoverArmies) {
321          System.out.println(Constants.PLAYER_ISSUE_ORDER_INVALID_ARMIES);
322          return false;
323      }
324      return true;
325  }

```


New implementation – Build 2

```

OUTLINE
{ } com.w10.risk_game.commands
  Deploy
    d_player
    d_countryId
    d_num
    Logger
    Deploy(Player, int, int)
    getCountryId() : int
    getNum() : int
    execute() : void
    ValidateOrder(Player, String, String) : boolean
    CheckValidCountry(List<Country>, String) : boolean
    CheckValidArmy(Player, int) : boolean
    CheckValidDeployInput(String[]) : boolean

src > main > java > com > w10 > risk_game > commands > Deploy.java > Deploy > CheckValidDeployInput(String[])
147  Logger.log(Constants.DEPLOY_INVALID_ARMIES_ERROR);
130  return false;
131  }
132  if (p_num > p_player.getleftoverArmies()) {
133  Logger.log(Constants.DEPLOY_INVALID_ARMIES);
134  return false;
135  }
136  return true;
137  }
138
139  /**
140   * This function is used to check the input format for deploy command. The input
141   * should have three parts (one string and two positive integers)
142   *
143   * @param p_inputArray
144   * @return boolean value to show whether the input format is valid
145   */
146
147  public static boolean CheckValidDeployInput(String[] p_inputArray) {
148  // Step 1: Check the length of the input
149  if (p_inputArray.length != 3) {
150  Logger.log(MessageFormat.format(Constants.PLAYER_ISSUE_ORDER_NOT_CONTAIN_ALL_NECESSARY_PARTS, ...arguments:"deploy",
151  "three"));
152  return false;
153  }
154  // Step 2: Check whether the country id is positive integer
155  String l_countryId = p_inputArray[1];
156  String l_num = p_inputArray[2];
157  for (int i = 0; i < l_countryId.length(); i++) {
158  if (!Character.isDigit(l_countryId.charAt(i))) {
159  Logger.log(Constants.PLAYER_ISSUE_ORDER_COUNTRY_ID_NOT_INTEGER);
160  return false;
161  }
162  }
163  // Step 3: Check whether the number of armies is positive integer
164  for (int i = 0; i < l_num.length(); i++) {
165  if (!Character.isDigit(l_num.charAt(i))) {
166  Logger.log(Constants.PLAYER_ISSUE_ORDER_ARMIES_NOT_INTEGER);
167  return false;
168  }
169  }
170  // Step 4: Return true if the input format is valid
171  return true;
172  }
173  }
174

```