Refactoring

Concordia University

Department of Computer Science and Software Engineering

Advanced Programming Practices

SOEN 6441 --- Winter 2024

Professor: Joey Paquet

Students: Team 14

Mahdieh Shekarian-40279366

Parsa MohammadRezaei-40277489

AmirMohammad RezaeiPour-40279239

Farid Farahmand-40241630

Mohammad Aminan-40277902

Omer Sayem-40226505

Potential refactoring targets.

Explain how you have identified the potential refactoring targets.

Individual Analysis:

Each team member was assigned the task of conducting an independent analysis of the codebase. We examined the code, identified potential refactoring targets, and documented our findings.

Meeting and Discussion:

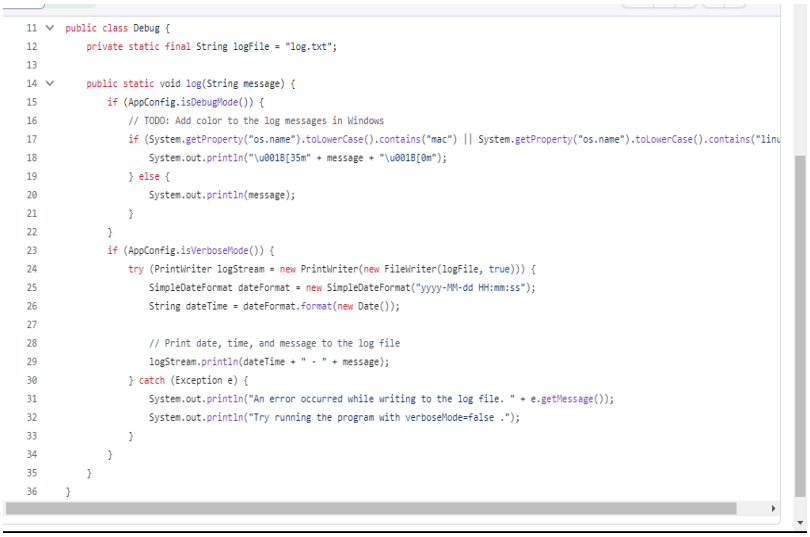
Following the individual analysis, a team meeting was held to discuss the findings and select the most promising refactoring targets. During the meeting, each team member presented their identified targets, highlighting the rationale behind their choices.

Refactoring Target Selection:

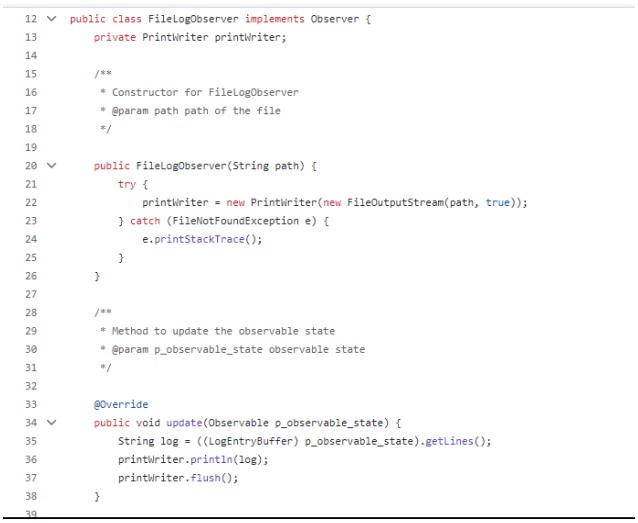
After a thorough discussion, the team reached a consensus on the top five refactoring targets to prioritize. The selected targets were chosen based on their potential to significantly improve the codebase and address common pain points. Careful consideration was given to the feasibility of the refactoring tasks and their alignment with the project's current goals and timelines.

List of 5 applied refactorings:

- **Refactor Logging System**
  - The old logging system was used to log debug outputs.
  - The new logging system logs at every single print in the whole code base.
  - The new logging system uses the Observer Pattern.
  - This refactor did not need any tests, since it contains very little logic.



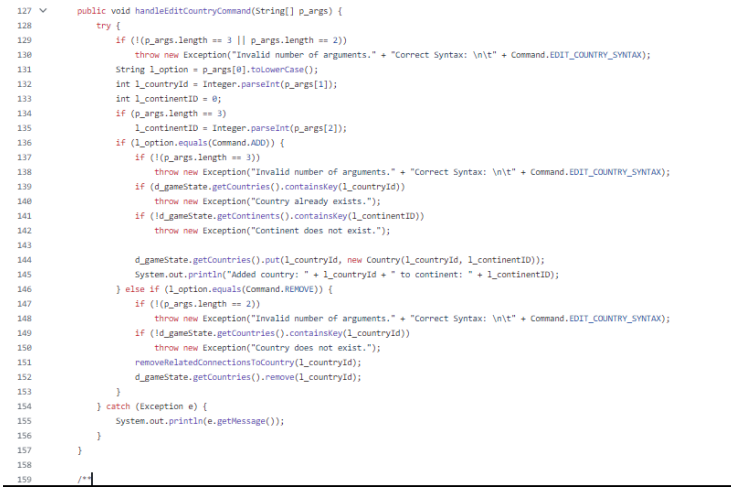
```
public class Debug {
    private static final String logFile = "log.txt";

    public static void log(String message) {
        if (AppConfig.isDebugMode()) {
            // TODO: Add color to the log messages in Windows
            if (System.getProperty("os.name").toLowerCase().contains("mac") || System.getProperty("os.name").toLowerCase().contains("linu
                System.out.println("\u001B[35m" + message + "\u001B[0m");
            } else {
                System.out.println(message);
            }
        }
        if (AppConfig.isVerboseMode()) {
            try (PrintWriter logStream = new PrintWriter(new FileWriter(logFile, true))) {
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
                String dateTime = dateFormat.format(new Date());

                // Print date, time, and message to the log file
                logStream.println(dateTime + " - " + message);
            } catch (Exception e) {
                System.out.println("An error occurred while writing to the log file. " + e.getMessage());
                System.out.println("Try running the program with verboseMode=false .");
            }
        }
    }
}
```



```
public class FileLogObserver implements Observer {
    private PrintWriter printWriter;

    /**
     * Constructor for FileLogObserver
     * @param path path of the file
     */

    public FileLogObserver(String path) {
        try {
            printWriter = new PrintWriter(new FileOutputStream(path, true));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * Method to update the observable state
     * @param p_observable_state observable state
     */

    @Override
    public void update(Observable p_observable_state) {
        String log = ((LogEntryBuffer) p_observable_state).getLines();
        printWriter.println(log);
        printWriter.flush();
    }

}
```

- **Take Adding/Removing Country Logic Out of CountryController**
  - Too much unrelated logic had made the code less readable

- Adding a country to GameState makes more sense to be a method of GameState, rather than CountryController.
- Already had tests



```
public void handleEditCountryCommand(String[] p_args) {
    try {
        if (!(p_args.length == 3 || p_args.length == 2))
            throw new Exception("Invalid number of arguments." + "Correct Syntax: \n\t" + Command.EDIT_COUNTRY_SYNTAX);
        String l_option = p_args[0].toLowerCase();
        int l_countryId = Integer.parseInt(p_args[1]);
        int l_continentID = 0;
        if (p_args.length == 3)
            l_continentID = Integer.parseInt(p_args[2]);
        if (l_option.equals(Command.ADD)) {
            if (!(p_args.length == 3))
                throw new Exception("Invalid number of arguments." + "Correct Syntax: \n\t" + Command.EDIT_COUNTRY_SYNTAX);
            if (d_gameState.getCountries().containsKey(l_countryId))
                throw new Exception("Country already exists.");
            if (!d_gameState.getContinents().containsKey(l_continentID))
                throw new Exception("Continent does not exist.");

            d_gameState.getCountries().put(l_countryId, new Country(l_countryId, l_continentID));
            System.out.println("Added country: " + l_countryId + " to continent: " + l_continentID);
        } else if (l_option.equals(Command.REMOVE)) {
            if (!(p_args.length == 2))
                throw new Exception("Invalid number of arguments." + "Correct Syntax: \n\t" + Command.EDIT_COUNTRY_SYNTAX);
            if (!d_gameState.getCountries().containsKey(l_countryId))
                throw new Exception("Country does not exist.");
            removeRelatedConnectionsToCountry(l_countryId);
            d_gameState.getCountries().remove(l_countryId);
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

/**
```

```
public class CountryController {
    public void handleEditCountryCommand(String[] p_args) {
        Debug.log("p_args[" + i + "]: " + p_args[i]);
        if (p_args[i].equalsIgnoreCase("-add")) {
            assert i + 3 <= p_args.length;

            i += 2;
        } else if (p_args[i].equalsIgnoreCase("-remove")) {
            assert i + 2 <= p_args.length;
            i += 1;
        } else {
            throw new Exception();
        }
    }
    Debug.log("handleEditCountryCommand: passed number of arguments check.");
} catch (exception | AssertionError e) {
    CustomPrint.println("No command was executed. Invalid number of arguments. " + "Correct Syntax: \n\t" + Command.EDIT_COUNTRY_SYNTAX);
    return;
}
int argumentCount = 0;
for (int i = 0; i < p_args.length; i++, argumentCount++) {
    try {
        String l_option = p_args[i].toLowerCase();
        int l_countryId = Integer.parseInt(p_args[i + 1]);

        if (l_option.equals(Command.ADD)) {
            int l_continentId = Integer.parseInt(p_args[i + 2]);
            i += 2;
            Country newCountry = new Country(l_countryId, l_continentId);
            d_gameState.addCountry(newCountry);
            CustomPrint.println("Added country: " + l_countryId + " to continent: " + l_continentId);
        }

        if (l_option.equalsIgnoreCase(Command.REMOVE)) {
            i += 1;
            if (!d_gameState.getCountries().containsKey(l_countryId))
                throw new Exception("Country does not exist.");
            removeRelatedConnectionsToCountry(l_countryId);
            d_gameState.getCountries().remove(l_countryId);

        }

    } catch (NumberFormatException e) {
        CustomPrint.println("Error with EditCountry at index " + argumentCount + " : " + " Invalid country ID.");
```

- **Take Adding/Removing Continent Logic Out of CountryController**
  - Too much unrelated logic had made the code less readable
  - Adding a continent to GameState makes more sense to be a method of GameState, rather than CountryController.

- Already had tests

```
165
166     public void handleEditContinentCommand(String[] p_args) {
167         try {
168             if (!(p_args.length == 3 || p_args.length == 2))
169                 throw new Exception("Invalid number of arguments." + "Correct Syntax: \n\t" + Command.EDIT_CONTINENT_SYNTAX);
170             String l_option = p_args[0].toLowerCase();
171             int l_continentId = Integer.parseInt(p_args[1]);
172
173             if (l_option.equals(Command.ADD)) {
174                 if (p_args.length != 3)
175                     throw new Exception("Invalid number of arguments." + "Correct Syntax: \n\t" + Command.EDIT_CONTINENT_SYNTAX);
176                 int l_bonus = Integer.parseInt(p_args[2]);
177                 if (d_gameState.getContinents().containsKey(l_continentId))
178                     throw new Exception("Continent already exists.");
179                 d_gameState.getContinents().put(l_continentId, new Continent(l_continentId, l_bonus));
180                 System.out.println("Added continent: " + l_continentId + " with bonus: " + l_bonus);
181                 //TODO: handle the savemap command
182             } else if (l_option.equals(Command.REMOVE)) {
183                 if (!d_gameState.getContinents().containsKey(l_continentId))
184                     throw new Exception("Continent does not exist.");
185                 d_gameState.getContinents().remove(l_continentId);
186                 removeRelatedCountriesToContinent(l_continentId);
187             }
188         } catch (Exception e) {
189             System.out.println(e.getMessage());
190         }
191     }
192
193
```

```
16  public class CountryController {
207     public void handleEditContinentCommand(String[] p_args) {
208
209         try {
210             for (int i = 0; i < p_args.length; i++) {
211                 Debug.log("p_args[" + i + "]: " + p_args[i]);
212                 if (p_args[i].equalsIgnoreCase("-add")) {
213                     assert i + 3 <= p_args.length;
214
215                     i += 2;
216                 } else if (p_args[i].equalsIgnoreCase("-remove")) {
217                     assert i + 2 <= p_args.length;
218                     i += 1;
219                 } else {
220                     Debug.log("errrrr");
221                     throw new Exception();
222                 }
223             }
224             Debug.log("handleEditContinentCommand: passed number of arguments check.");
225         } catch (Exception | AssertionError e) {
226             CustomPrint.println("No command was executed. Invalid number of arguments. " + "Correct Syntax: \n\t" + Command.EDIT_CONTINENT_SYNTAX);
227             return;
228         }
229
230         int argumentCount = 0;
231         for (int i = 0; i < p_args.length; i++, argumentCount++) {
232             try {
233
234                 String l_option = p_args[i].toLowerCase();
235
236                 if (l_option.equals(Command.ADD)) {
237                     int l_continentId = Integer.parseInt(p_args[i + 1]);
238                     int l_bonus = Integer.parseInt(p_args[i + 2]);
239                     i += 2;
240                     Continent newContinent = new Continent(l_continentId, l_bonus);
241                     d_gameState.addContinent(newContinent);
242                     CustomPrint.println("Added continent: " + newContinent);
243                 }
244
245                 if (p_args[i].equalsIgnoreCase(Command.REMOVE)) {
246                     int l_continentId = Integer.parseInt(p_args[i + 1]);
247                     i += 1;
248                     d_gameState.removeContinent(l_continentId);
249                     removeRelatedCountriesToContinent(l_continentId);
```

- **Refactor Order Class**
  - Validating order inside Order class
  - Now uses Command Pattern to both execute and validate.
  - Already had tests.
- **Refactor GameEngine to Use State Pattern**
  - Code from Game Engine is moved to multiple files.
  - Phase class and its descendants are created.

- new tests were added for GameEngin in the file GameEnginTest File

```
parsamre
@Test
@DisplayName("Test to check conditions for entering issue order phase (when map us loaded)")
public void shouldNotGoToIssueOrderWhenNoPlayersAreAdded() {
    StartupPhase startup = new StartupPhase(gameEngine);
    gameEngine.setGamePhase(startup);
    mapController.handleloadMapCommand(new String[]{"canada.map"});
    mapController.handleSaveMapCommand(new String[]{"canada.map"});
    startup.goToIssueOrdersPhase();
    assertEquals(startup, gameEngine.getGamePhase());
}

parsamre
@Test
@DisplayName("Test to check conditions for entering issue order phase (when players are added.)")
public void shouldNotGoToIssueOrderWhenMapIsNotValid() {
    StartupPhase startup = new StartupPhase(gameEngine);
    playerController.handleGamePlayerCommand(new String[]{"-add", "player1"});
    gameEngine.setGamePhase(startup);
    startup.goToIssueOrdersPhase();
    assertEquals(startup, gameEngine.getGamePhase());
}
```

List of other 14 refactoring targets:

1. **Consistent Variable Naming:**
   Some parts of code has d_continentId and some parts has d_id, both pointing to the id of the object.

2. **String Concatenation:**
   In the Continent class, in the second constructor, the line this.d_name = "" + p_continentId; uses string concatenation to convert the p_continentId to a string. It would be cleaner to use String.valueOf(p_continentId) instead.

3. **Use Constructor Chaining:**
   The constructors can be simplified and made more concise by using constructor chaining.
   For example, in the Country class, the first and second constructors can call the third constructor, passing the necessary arguments. This avoids code duplication and makes the code more maintainable.
   For example, in the Game state, instead of explicitly initializing the member variables, the constructor can call another constructor, passing the necessary arguments.

For example, the default constructor Player() is currently empty. It can be removed since it doesn't serve any purpose. Alternatively, if we want to keep it for some reason, we can use constructor chaining to call the parameterized constructor with default values.

4. **Remove Unused Import**:
   The import statement import models.orders.Order is not used in the code in Player class and can be removed.
   It seems that the handleValidateMapCommand method is not used in the provided code. If it's not needed, consider removing it to reduce code clutter.
   The config.Debug import statement is not used in the provided code and can be safely removed.
   The code imports a few classes that are not used (controllers.*, models.orders.Order). We can remove these unused imports to keep the code clean.

5. **Improve getCountryIds():**
   In the Player class, the getCountryIds() method can be simplified using Java Streams. Instead of manually iterating over the countries and adding their IDs to a list, we can use the map() function to extract the IDs and collect them into an ArrayList.

6. **Grouping Options:**
   The options -add and -remove could be grouped together in a separate class or nested class to provide more organization and structure. For example, we could define an Option class with constants ADD and REMOVE inside the Command class.

7. **Use Exceptions for Error Handling:**
   Instead of printing error messages directly to the console, we can use exceptions to handle and propagate errors. This allows for better error handling and can make the code more robust. Create custom exception classes for different types of errors and throw them when necessary.

8. **Separate Input/Output from Business Logic:**
   The CountryController class currently handles both the business logic and input/output operations. It's generally better to separate these concerns by creating separate classes for input/output and delegating the appropriate tasks to them. This improves the code's modularity and testability.

9. **Encapsulate Data:**
   Instead of directly accessing and modifying the game state and other data structures from outside the class, consider encapsulating the data and providing appropriate getter and setter methods. This helps in maintaining data integrity and allows for better control over data access.

10. **Apply the Single Responsibility Principle (SRP):**
    The CountryController class currently handles multiple responsibilities, such as assigning countries, editing neighbors, editing countries, and editing continents. Consider refactoring the class into smaller, more specialized classes, each responsible for a single task. This improves code readability, maintainability, and testability.

11. **Use Try-With-Resources:**
    In the saveMap() method, use try-with-resources to automatically close the BufferedWriter and handle any exceptions that may occur during writing. This ensures that the writer is always closed, even in the case of an exception.
12. **Separate Concerns:**
    For example, the MapController class currently handles both file I/O and game map validation. It's generally better to separate these concerns by creating a separate class for file I/O operations and delegating the appropriate tasks to it. This improves the code's modularity and testability.
    For example, the takeOrderCommands method is responsible for both parsing user input and handling different order types. Consider splitting these responsibilities into separate methods or classes to improve code readability and maintainability.
    We can split the mainGameLoop method into smaller methods or classes to separate concerns and improve readability.
    The handleGamePlayerCommand method is responsible for both parsing user input and modifying the game state. We can split these responsibilities into separate methods or classes to improve code readability and maintainability.

13. **Consider Dependency Injection:**
    Instead of creating an instance of GameMapReader within the MapController class, consider passing it as a dependency to the constructor. This allows for better decoupling and easier testing, as the dependencies can be mocked or replaced with alternative implementations.

14. **Simplify conditional logic:**
    The code currently has nested if-else statements to handle different commands. Consider using a switch statement for better readability and maintainability.