Refactoring Document BUILD #2

Group: W17 Course: SOEN6441 Date: 2023-11-07

Potential refactoring targets. • Explain how you have identified the potential refactoring targets. • List of at least 15 refactoring targets.

Actual refactoring targets. • Give the rationale explaining how you have chosen or excluded some of the potential refactoring targets. • List 5 actual refactoring targets.

Refactoring operations For each of the 5 actual refactoring targets. • List all the tests that apply to the class involved in the refactoring operation. • If not enough tests exist for these classes, add more and list them. • Explain why this refactoring operation was deemed necessary. • Give a before/after depiction of the refactoring operation.

1 Potential Refactoring Targets

1. **Pull Up Field**

   Creating a parent class, like CommandHandler, to extract common attributes and methods from different handlers.

   How to Identify: This helps avoid duplicated fields such as

   ```
   public GameEngine d_gameEngine;
   ```

   Every CommandHandler has similar fields, so extracting them enforces a more organized and consistent structure.

2. **Consolidate Duplicate Conditional Fragments**

   Extract common conditional logic into a single location.

   How to Identify:

   ```
   System.out.println(l_response);
   return "stayCurrentPlayer";
   ```

   In PlayerCommandHandler, many conditions form a response like l_response. Most of them need to stay current player for issuing orders, so reducing code redundancy and making it easier to manage and update.

3. **Create more getter and setter for attributes**

   Instead of directly accessing d_ attributes, it's advisable to create getter and setter methods for these attributes.

   How to Identify:

   ```
   public GameEngine d_gameEngine;
   ```

   Some classes have access to GameEngine, and it could be invoked immediately since it is a public attribute. If create a getter and setter for this attribute, it provides encapsulation and makes it easier to enforce data integrity and modify behavior if needed in the future.

4. **Reorganize demo driver**

   This suggests restructuring the demo driver code to improve its clarity and organization, making it easier for developers to understand and maintain.

   How to Identify: The demo driver could be organized in various forms because it has the privilege to call all functions from the Controller. But it would be better to make it higher readability for further extension.

5. **Declare type first**

   In Java, specifying the type when creating collections is considered good practice.

   How to Identify: For example

   ```
   this.d_players = new ArrayList<>();
   this.d_playerConquerInTurn = new ArrayList<>();
   ```

   It is identified by the Java IDE, a different version has different warnings, and some of them recommend specifying the type when defining a new variable. New $ArrayList <>$ () could be preferred over $ArrayList < Player >$ ().

6. **Pull Up Constructor**

   This involves creating a constructor in a parent class and using the super keyword to initialize common attributes.

   How to Identify:

```
public CommandHandler(GameEngine p_gameEngine) {
    d_gameEngine = p_gameEngine;
    d_logEntryBuffer = p_gameEngine.getD_logEntryBuffer();
}
```

It also reduces code duplication in constructor logic.

7. **Integrate Controller**

   Combining parts of different classes, leads to a more cohesive and maintainable design.

   How to Identify:

```
public IssueOrders(GameEngine p_gameEngine, MapCommandHandler p_commandHandler) {
    super(p_gameEngine, p_commandHandler);
}
```

   Like IssueOrders and ExecuteOrders, they have parts of GameEngine to define the game process. It would be better to combine them into the GameEngine.

8. **Replace class with a method**

   Creating methods within a class to replace the functionality of entire classes when the class's primary purpose is encapsulated in a single method.

   How to Identify: Also, IssueOrders and ExecuteOrders, have only one method and are called by GameEngine. So they could be replaced by a new function in GameEngine.

9. **Separate complex class**

   Splitting a complex class into smaller classes with distinct responsibilities can improve code organization and maintainability.

   How to Identify:

```
public void main(String[] p_args) {
    Scanner l_sc = new Scanner(System.in);
    GameMap l_map = d_map;
    while (true) {
```

```
String l_userInput;
l_userInput = l_sc.nextLine();
String l_commands[] = l_userInput.split(" ");

// solve the command
switch (l_commands[0]) {
case "showmap":
    l_map.getD_mapView().showMap();
    break;
```

Such as MapService, has both functions of reading the user's input and analysing it to form a response. These functions are too complex so they could be separated.

10. **Apply State Pattern**

    This involves creating a class structure for the State Pattern, like MainPhase and SubPhases, to represent different states in the application. This can make the code more modular and extensible.

    How to Identify: State Pattern is a general design pattern which should be applied into the project.

11. **Rename Setter and Getter method**

    Ensuring a consistent naming convention for getter and setter methods.

    How to Identify:

```
public String getName() {
    return this.d_name;
}
public int getD_reinforcementPool() {
    return d_reinforcementPool;
}
```

    In *Player.java*, it is obvious that Getter functions of different attributes follow different formats. Changing them improves code readability and maintainability.

12. **Replace counter with size()**

Instead of using explicit counters, using the size() method on collections can make the code more concise and less error-prone.

How to Identify:

```
private ArrayList<Continent> d_continents = new ArrayList<Continent>();
private ArrayList<Country> d_countries = new ArrayList<Country>();
private int d_continentCount = 0;
private int d_countryCount = 0;
```

Actually, $d_continentCount$ and $d_countryCount$ are equal to $d_continents.size()$ and $d_countries.size$. However, using a counter has the potential to cause bugs because they might not be updated in time.

13. **Replace string splicing with string format**

    Using string formatting functions instead of manually concatenating strings.

    How to Identify:

```
System.out.println("Game is ended by Player \"" + p_currentPlayer.getName() + "\"."
String l_response = String.format("Player \"%s\" has signified.", p_currentPlayer.g
```

    Some parts use string concatenating and others use string format. In general, string format can improve code readability and reduce the risk of errors.

14. **Narrow scope of global constant**

    Limiting the scope of global constants to only where they are needed can reduce potential conflicts and improve code organization.

    How to Identify: All global constants are defined in $GameConstant.java$, but they actually are just called in specific places.

15. **Isolate default setting**

    Keeping default settings isolated and not spread throughout the codebase.

    How to Identify:

```
public static void defaultGameMap(GameMap p_gameMap) {
    p_gameMap.addContinent(1, 3);
    p_gameMap.addCountry(1, 1);
    p_gameMap.addCountry(2, 1);
    p_gameMap.addNeighbor(1, 2);
    p_gameMap.addNeighbor(2, 1);
}
```

It is convenient to set some default Game Environment before the function test, but it would be better to integrate them into a file rather than hard code in the demo driver, which makes it easier to manage and change default values.

16. **Apply Observer Pattern**

    Instead of using return values, using the Observer Pattern can improve the flow of information in the codebase and enhance code maintainability.

    How to Identify:

    ```
    return "stayCurrentPlayer";
    ```

    In *PlayerCommandHandler.java*, the return value is like a signal for GameEngine to decide whether to move to the next player. This mechanism is not very comprehensive to some extent.

17. **Change null return value to void**

    When a method doesn't need to return a value, changing its return type to void can make the code more self-explanatory and reduce potential issues related to null values.

    How to Identify: When there is no important information returned after a function, it always returns null. It might cause potential bugs so void type could be better.

2 Actual Refactoring Targets

2.1 Why Choose

1. **Pull Up Field**

   In Build1, the project has MapCommandHandler and PlayerCommandHandler, and they are developed by different individuals. However, in Build2, GameCommandHnadler is designed to add new features from GamePlayService. In order to make CommandHandler uniform, it creates a parent class and pulls up fields for them.

   CommandHandler as a parent class also makes it extensional, so it is convenient to add a new class like GameCommandHnadler.

2. **Reorganize demo driver**

   The demo driver used to be designed as a demo service in Build1, this service just performs like an interface for user input commands. In that case, we focus on its functions instead of readability. As a result, there is a need to reorganize it to present game flow clearly.

   Moreover, a driver is a concept from some design patterns like command pattern and state pattern. When applying these patterns, the demo driver must follow a specific style.

3. **Integrate Controller**

   The controller initially is designed as three parts to obey the game process. However, after applying the state pattern, game flow is reflected in state transfer rather than GameEngine. There is no need to slit the GameEngine which may cause confusion and low efficiency for debugging.

   Another reason is that LogWritter and LogBufferEntry are added as new features for recording the game process into log files. They are parts of the Controller. When issueOrders and executeOrders are just parts of GameEngine, it is recommended to integrate them into GameEngine rather than single classes.

4. **Separate complex class**

   In Build1, mapService is too complex to some extent. It is like an integrated package to modify a game map, and it has functions for receiving and handling commands for users.

When GameCommandHandler is created, some parts of mapService are similar to it. So splitting the complex class into MapCommandHandler and MapEditor is helpful for adding more features but avoid making it more complex.

5. **Apply State Pattern**

   State Pattern is required in Build2. As we know, sometimes we want to be able to alter the behaviour of an object when its internal state changes and make it easy to add new varying behaviour that comes with new states.

2.2 Why Exclude

1. **Consolidate Duplicate Conditional Fragments**

   Exclusion can be justified when the conditional logic appears similar on the surface but serves different purposes under the hood. Attempting to consolidate such logic might lead to convoluted and error-prone code. In these cases, maintaining separate conditionals for clarity may be the better choice.

2. **Create more getter and setter for attributes**

   There are scenarios where direct access to attributes might be necessary for performance reasons, and introducing getter and setter methods could introduce unnecessary overhead. In such cases, adhering to the naming convention while allowing direct access might be a reasonable compromise.

3. **Rename Setter and Getter method**

   The existing naming convention for getter and setter methods is already consistent and easily understood by developers working on the code, renaming them to follow a different format may not provide significant benefits and could increase cognitive load.

4. **Replace counter with size()**

   In cases where explicit counters are used for more complex logic that involves iteration, conditions, or multiple data structures, replacing them with size() might oversimplify the code and obscure its actual behavior.

5. **Replace string splicing with string format**

   When string splicing is used in a way that enhances code readability and maintainability, such as for constructing complex messages or queries, switching to string formatting may not improve the code's clarity and can make the code less concise.

6. **Narrow scope of global constant**

   If a global constant genuinely needs to be accessible and used in multiple parts of the codebase, limiting its scope could lead to unnecessary code duplication or create difficulties when changes are needed.

7. **Isolate default setting**

   Default settings, when well-organized and documented, may not require further isolation, especially if they don't interfere with other parts of the code and their purpose is clearly understood.

8. **Apply Observer Pattern for issueOrder Phase**

   Exclusion can be justified when the existing approach to notifying players is simple and effective, and implementing the Observer Pattern could introduce unnecessary complexity and coupling between components.

9. **Change null return value to void**

   If a method returning null serves a meaningful purpose, such as indicating an intentional absence of data, changing it to void may remove valuable information and make the code less expressive.

3 Refactoring Operations

1. **Pull Up Field**

   Before: There are two separate classes called PlayerCommandHandler and MapCommandHandler.

   After: Create parent class CommandHandler, let both of them inherit from the parent class and extend a new class.

   Why necessary: Make it extendable for adding a new GameCommandHandler.

   Test: testPullUpMapField, testPullUpPlayerField in Junit, to test whether the field remains the same performance.

2. **Reorganize demo driver**

   Before: Manually set phase in DemoService to present features of gameplay and output information of phase transferring.

   After: Integrate game flow and phase transferring into GameEngine, simplify demo driver's function which is called start() of GameEngine.

   Why necessary: Follow State Pattern, make demo service as a driver for feature presents.

   Test: testDriverDemoStart, testDriverDemoShowmap and testDriverDemoEnd of Build1DemoTest and Build2DemoTest in Junit to test whether the game still runs correctly.

3. **Integrate Controller**

   Before: There are two separate classes called IssueOrders.java and ExecuteOrders.java

   After: No extra classes, but two methods in GameEngine called issueOrdersInTurn() and executeAllCommittedOrders(). Their function is similar to classes before.

   Why necessary: Follow State Pattern, and enhance GameEngine integration.

   Test: testGameEngineIssueOrders and testGameEngineExecuteOrders in Junit to test whether issueOrders and executeOrders perform the same before and after refactoring GameEngine.

4. **Separate complex class**

   Before: A complex class called MapService

   After: Two classes MapEditor and MapCommandHandler involve functions of MapService.

Why necessary: MapService is hard to modify and add more features.

Test: testMapService in GameEngineTest.java shows that mapservice performs the same after splitting.

5. **Apply State Pattern**

   Before: No clear concept of Phase as a state in Game flow. Just parts of GameEngine.

   After: Create a lot of Phase and SubPhase to define each Command's response under state circumstances.

   Why necessary: Need to apply State Pattern.

   Test: the same tests in testDriverDemoStart and testDriverDemoEnd verify that phase could be transferred as expected.