Refactoring Document BUILD #3

Group: W17   Course: SOEN6441   Date: 2023-11-29

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. **Apply Adapter Pattern**

   In Build2, there are only MapEditor for editing map file. But it needs to read Conquest Map so that it applies the Adapter Pattern.

   How to Identify: This is required to realise some specific functions.

   ```
   public class ConquestMapEditor {
       public ConquestMap loadConquestMap(String p_fileName){
           return l_conquestMap;
       }
   }
   ```

   Follow the Adapter Pattern to add MapEditorAdapter extends from MapEditor, it will do a translation from ConquestMap to DominationMap.

2. **Use List or Array**

   Choosing between a list and an array in Java depends on a fixed-size collection or a flexible collection. Lists offer greater flexibility, ease of use, and compatibility with generic types, providing type safety.

   How to Identify:

```
String l_mapInfo = l_text.split("\n\n")[0];
String l_continents[] = l_text.split("\n\n")[1].split("\n");
String l_territories[] = l_text.split("\n\n")[2].split("\n");
```

Arrays here are enough for function logic, but it would be better to refactor them to list since the text length is flexible..

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 String d_mapInfo;
   private ArrayList<Continent> d_continents = new ArrayList<>();
   private ArrayList<Country> d_countries = new ArrayList<>();
   ```

   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. **Apply Strategy Pattern**

   To apply the Strategy Pattern, it defines a family of algorithms encapsulated in separate classes, each implementing a specific strategy. Create an interface or abstract class to declare the common method signature.

   How to Identify:

   ```
   protected PlayerStrategy(Player p_player, List<Country> p_countryList, LogEntryBuff
       d_player = p_player;
       d_countryList = p_countryList;
       d_logEntryBuffer = p_logEntryBuffer;
   }

   protected abstract Country toAttack(Country p_sourceCountry);
   protected abstract Country toAttackFrom();
   ```

```
protected abstract Country toMoveFrom(Country p_sourceCountry);
protected abstract Country toDefend();
public abstract Order createOrder();
```

The most important thing for Strategy Pattern is that it should show the same features as well as a human player. Same function, same format but completely automate to generate commands.

5. **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.

6. **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 $ArrayList < Player > ()$.

7. **Recovery same game**

   When trying to load a game from a game file, it needs to ensure the game state is completely the same as the circumstances of the game saved.

   How to Identify:

```
GameEditor.loadGameFromFile("src/main/resources/Games/" + p_commands[1]);
System.exit(0);
```

When the subgame is launched, the previous game is stopped and exits immediately after the subgame ends.

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. **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.

10. **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.

11. **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.

12. **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.

13. **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.

14. **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.

15. **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.

16. **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.

17. **Extract tool wildly used**

    Some functions in the project are very similar but with different parameter types or return types. If extracted to a single file, it will enhance the code's readability and easy to expand.

    How to Identify:

```
Country l_newCountry = new Country(l_country.getCountryId(), l_country.getContinent
l_newCountry.setArmies(l_country.getArmies());
l_newCountry.setOwner(l_country.getOwner());
l_newCountry.setNeighborCountries(l_country.getNeighborCountries());
l_newCountryList.add(l_newCountry);
```

In *Country.java*, this model has a complex structure so that it could not be easily hard copy instead of copying reference. This feature is very common because many classes need a hard copy.

2 Actual Refactoring Targets

2.1 Why Choose

1. **Apply Adapter Pattern**

   The decision to apply the Adapter Pattern in this context is driven by the need to integrate functionality from the ConquestMapEditor into the existing MapEditor structure. In Build2, there is currently only a MapEditor for editing map files, but to enable specific functions, it becomes necessary to read Conquest Map files.

   The identified refactoring target involves creating a MapEditorAdapter that extends from MapEditor, serving as an adapter to translate operations from ConquestMap to DominationMap. This allows for the seamless integration of the ConquestMapEditor functionality into the existing MapEditor framework, promoting code reusability and maintaining a consistent interface while accommodating the specific requirements of handling Conquest Map files.

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

   The recommendation to create more getter and setter methods for attributes, instead of directly accessing them is motivated by the principles of encapsulation and data integrity. By encapsulating access to these attributes, it enhances the control over their modification and enforces data integrity.

   The explicit use of getter and setter methods provides a level of abstraction, making it easier to manage and validate data changes. Additionally, if modifications to the attribute's behavior or validation logic are required in the future.

3. **Apply Strategy Pattern**

   To apply the Strategy Pattern is driven by the desire to encapsulate a family of algorithms that are currently implemented in separate classes, each representing a specific strategy for the Player.

   Each specific strategy, such as "AggressivePlayerStrategy" or "DefensivePlayerStrategy," would then implement this interface or extend the abstract class, providing concrete implementations for the strategy-specific behavior. This design allows for interchangeable strategies, enabling the Player to switch between different algorithms dynamically. The identified methods represent the common features of these strategies, ensuring a consistent format while automating the generation of commands. This adherence to the Strategy Pattern enhances code modularity, promotes maintainability, and allows for easy extension with new strategies in the future.

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

   Change a null return value to void is based on the principle of making the code more self-explanatory and reducing potential issues related to null values.

   In situations where a method doesn't need to provide any meaningful information upon completion, changing its return type to void is suggested.

   This avoids the ambiguity associated with a null return value and makes the intent of the method clearer. By using void, it explicitly communicates that the method doesn't produce a result that needs to be captured or utilized by the calling code. This refactoring is particularly beneficial when null returns might lead to potential bugs or misunderstandings in the calling code.

5. **Extract tool wildly used**

   Creating a utility or helper class to encapsulate and centralize functions that are commonly used across the project.

   In the DeepCopyList, there is an operation to create a new instance of a Country with similar properties to an existing one. The suggestion is to extract this logic into a utility class, making it easily accessible and reusable throughout the project. By consolidating such commonly used functionality into a single file or class, you enhance code readability, reduce redundancy, and make it more convenient to expand or modify these operations in the future.

2.2 Why Exclude

1. **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.

2. **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.

3. **Replace string splicing with string format**

   There are scenarios where string splicing to information might be easy for developing reasons, and string format could not handle some special conditions. In such cases, adhering to the string splicing for easy use might be a reasonable compromise.

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. **Apply Adapter Pattern**

   Before: There are two separate classes called MapEditor and ConquestMapEditor.

   After: There is a new class called MapEditorAdapter which will invoke functions from ConquestMapEditor if the map file is recognized as a conquest map.

   Why necessary: Make it extendable for adding new features.

   Test: MapEditorTest, MapEditorAdapterTest in Junit, to test whether the conquest map is loaded as expected.

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

   Before: Public attributes in some classes are directly accessed.

   After: Transfer them to protected or private attribute with getter and setter.

   Why necessary: Considering data security, it is necessary to prevent data from being modified immediately.

   Test: testDriverDemoStart, testDriverDemoShowmap and testDriverDemoEnd in Junit to test whether the attribute works the same.

3. **Apply Strategy Pattern**

   Before: Players interact with the terminal and issue orders from command input.

   After: Many strategies stimulate player's behaviour and automatically generate orders without the player's interaction.

   Why necessary: Strategy Pattern is required, and when the map becomes so complex, it is impossible to play with human interaction

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

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

   Before: return null from subclasses

   After: no return

   Why necessary: Null has the potential to cause failure in future development, void makes sure the parent class won't receive an unexpected return value.

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

5. **Extract tool widely used**

   Before: Classes call their own private function to do some work.

   After: There is an integrated class called DeepCopyList for coping with complex strucure.

   Why necessary: Make code expandable.

   Test: the same tests in testDriverDemoStart and testDriverDemoEnd verify that every function is run as well as before refactoring.