

Refactoring Introduction

Refactoring in the map package

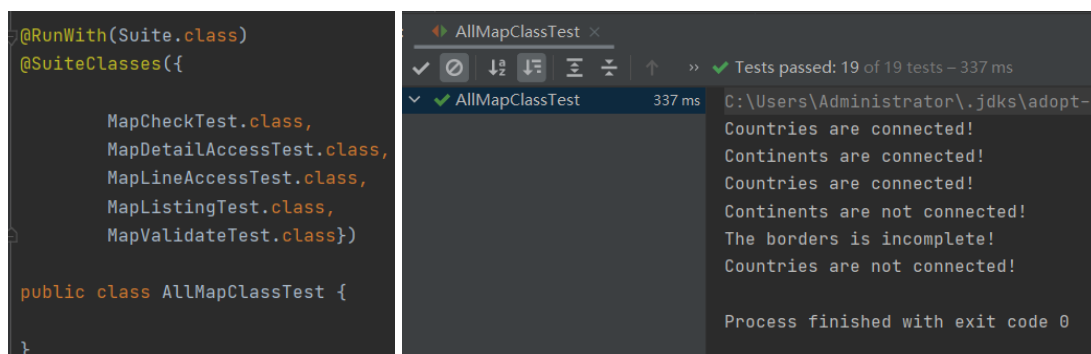
(1 Selected Refactoring option has been marked)

In the map package, there are 5 classes related to the detecting map function. So, those classes have to merge or modify. We listed 5 potential targets to refactor:

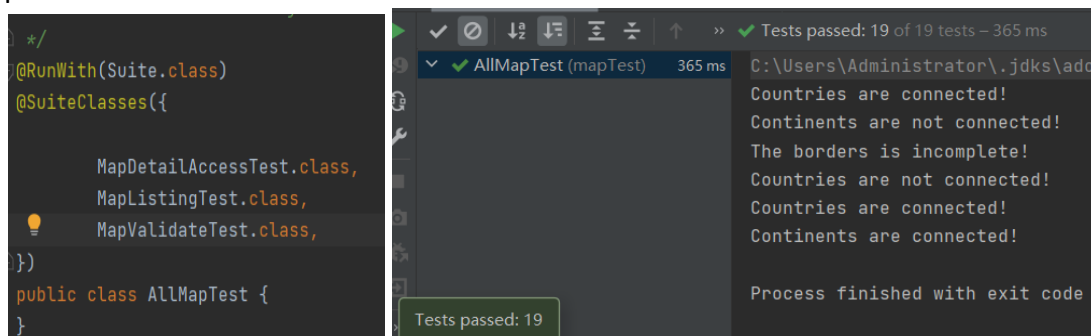
1. Combining all classes in the map package into one class.
2. Creating one class that inherits all map classes in the map package.
3. Combining all classes in the map package except the class that can detect maps.
4. Combining classes that are related to get the file information(the country's storage location), and get the map information(functions that return a list of countries or continents).

Option #1 and #3 may cause a bigger mess than before because the new class may be too long to maintain. Option #2 is easy to call, but it doesn't make the project structure clear. Therefore, #4 is more useful than other options. The methods classified by the function are more convenient to call.

The following two graphs show the results of the test runs before the refactoring. Nineteen tests are all passed.



After we refactor, there are only three classes which relate to the map validate function. The MapCheck.java combined into the MapValidate.java, The MapLineAccess.java combined into the MapDetailAccess.java. The following two graphs show the results of the test runs after the refactoring. Nineteen tests are all passed.

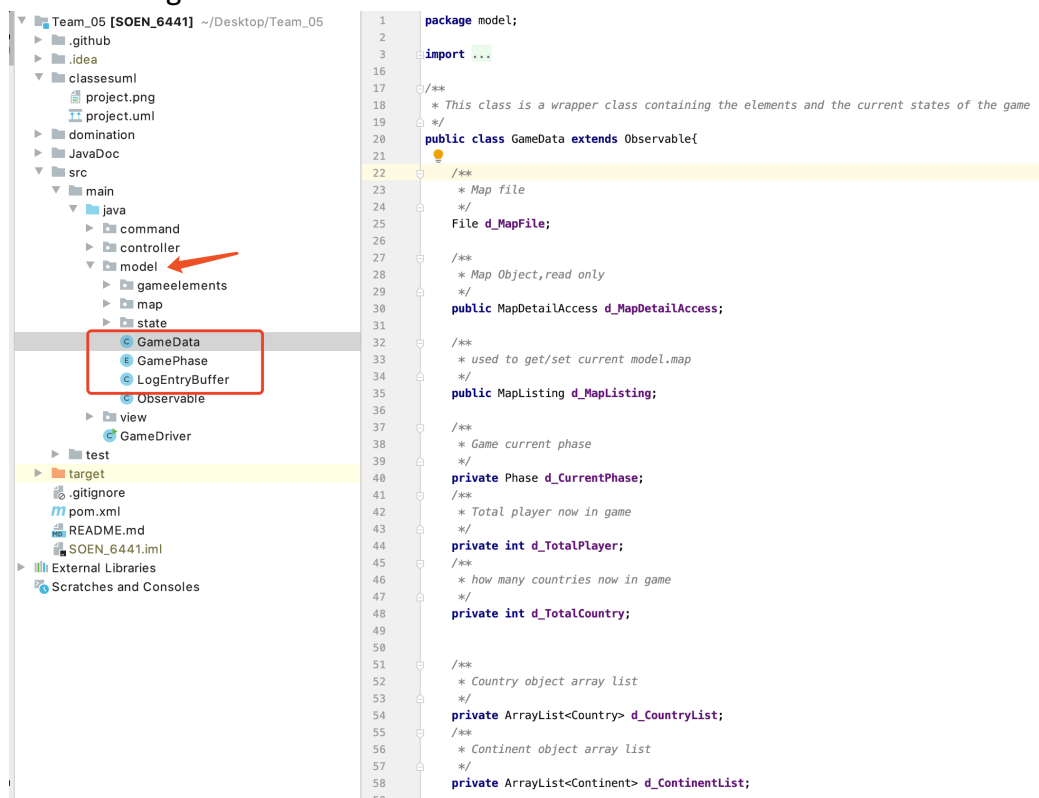


Refactoring GameEngine/GameData to MVC structure

(we are not fully implement MVC, View part is not complete)

(2 Selected Refactoring options have been marked)

5. In order to introduce the observer mode in the current version, we move all relevant game elements and game data from the previous version to a package named model as the program data in the model that responds to requests for data about its state. Also, Some classes like (GameData.java, GamePhase.java, LogEntryBuffer.java) in the package model extend Observable.java class that used to allow observers to watch the data change of them.

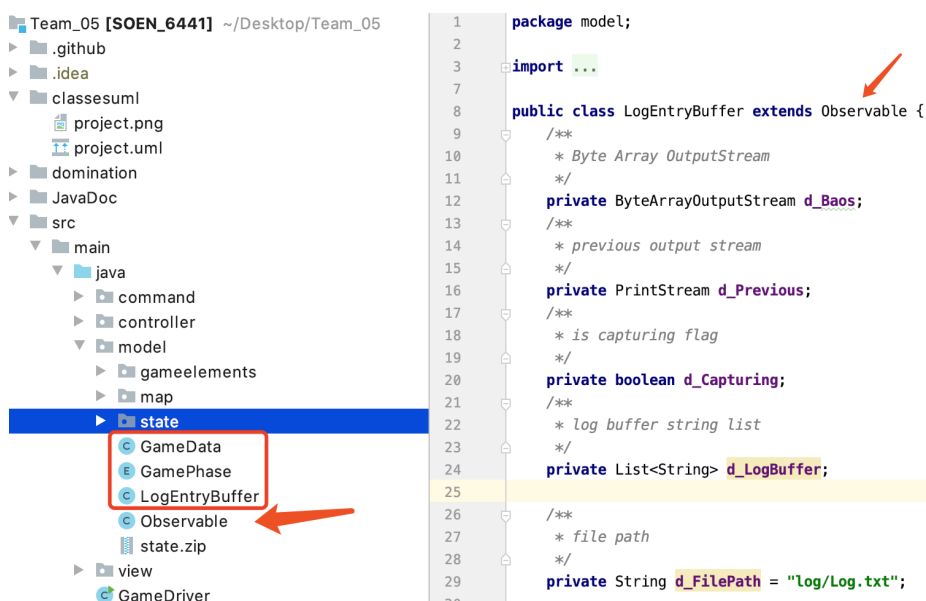


The screenshot shows an IDE with a project tree on the left and a code editor on the right. The project tree shows a package structure with 'model' highlighted. The code editor shows the implementation of the 'GameData' class, which extends 'Observable' and contains various game state variables.

```

1 package model;
2
3 import ...
4
5 /**
6  * This class is a wrapper class containing the elements and the current states of the game
7  */
8 public class GameData extends Observable{
9
10    /**
11     * Map file
12     */
13    File d_MapFile;
14
15    /**
16     * Map Object, read only
17     */
18    public MapDetailAccess d_MapDetailAccess;
19
20    /**
21     * used to get/set current model.map
22     */
23    public MapListing d_MapListing;
24
25    /**
26     * Game current phase
27     */
28    private Phase d_CurrentPhase;
29
30    /**
31     * Total player now in game
32     */
33    private int d_TotalPlayer;
34
35    /**
36     * how many countries now in game
37     */
38    private int d_TotalCountry;
39
40    /**
41     * Country object array list
42     */
43    private ArrayList<Country> d_CountryList;
44
45    /**
46     * Continent object array list
47     */
48    private ArrayList<Continent> d_ContinentList;
49
50 }

```



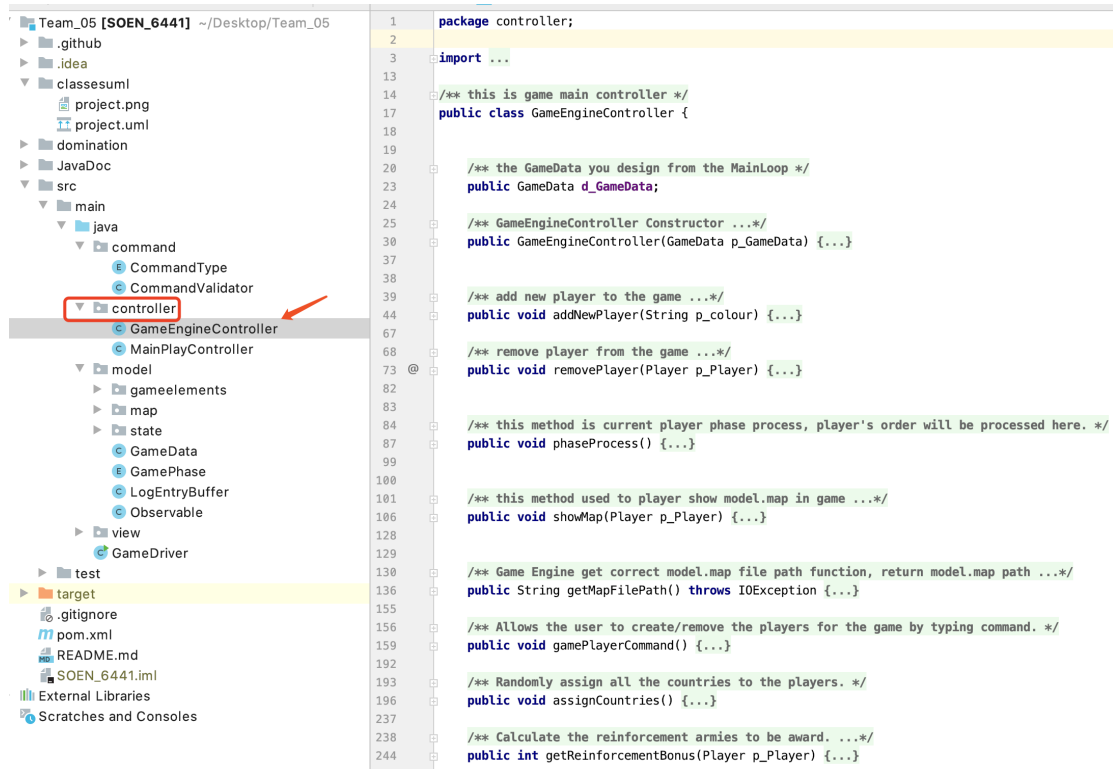
The screenshot shows an IDE with a project tree on the left and a code editor on the right. The project tree shows a package structure with 'model' highlighted. The code editor shows the implementation of the 'LogEntryBuffer' class, which extends 'Observable' and contains logging-related variables.

```

1 package model;
2
3 import ...
4
5 /**
6  * Byte Array OutputStream
7  */
8 private ByteArrayOutputStream d_Baos;
9
10 /**
11  * previous output stream
12  */
13 private PrintStream d_Previous;
14
15 /**
16  * is capturing flag
17  */
18 private boolean d_Capturing;
19
20 /**
21  * log buffer string list
22  */
23 private List<String> d_LogBuffer;
24
25 /**
26  * file path
27  */
28 private String d_FilePath = "log/Log.txt";
29
30 }

```

6. We refactored the *GameEngine.java* class in the previous version and move it from *gameplay* package to the package named *controller*, now the behavior of this class become the role of controller in Observer pattern and named *GameEngineController.java*



Related Tests

LogEntryBufferTest : writeTest

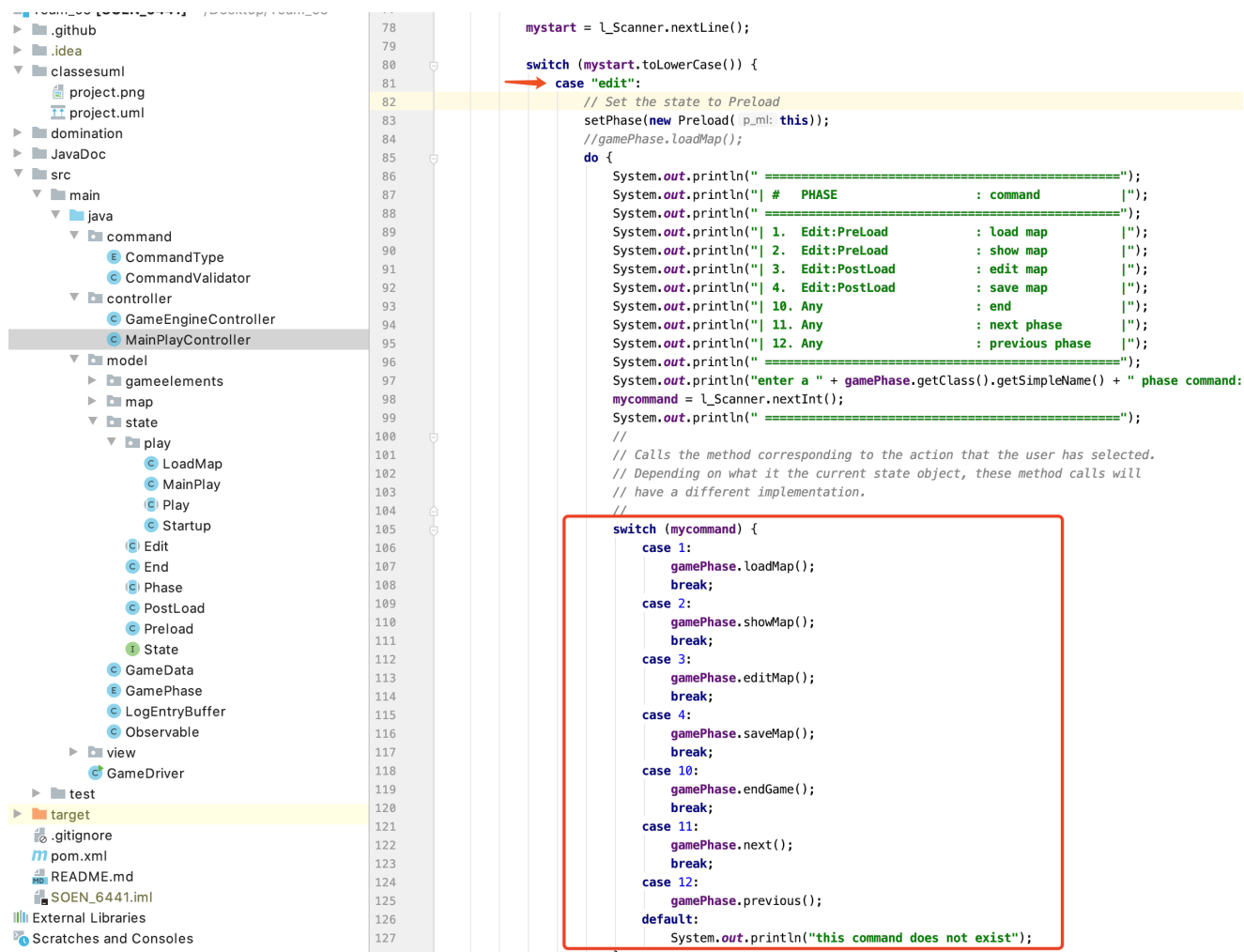
ViewTest: obsListAttachTest

GameDataTest: loadMapTest

Refactoring for State Pattern:

(1 Selected Refactoring option has been marked)

7. We removed the enum class *GamePhase.java*. Now, each game phase will base on the user input turning to the specified game phase in the state pattern, and remind the user what the current game phase they are in. Also, We changed the if-else statement in the previous version of Demo class *mainloop.java* to a switch case statement that is more consistent with the state pattern



8. User needs to enter the corresponding number for the represent game phase to get in before entering the command for the phase
9. We do not need to ask the user for which model (Map edit/Game play / Exit). they are going to get in. Users can based on the entered number to get into different states.
10. Divide the gameplay phase into 3 different states, reinforcement phase, issue order phase, execution phase. The user needs to enter the command for the current phase before getting into the next phase.

Related Tests

PlayStateTest: testEditState, testPlayState

StateTest: testEditState, testPlayState

GameEngine Test: testAddRemovePlayer, testShowMap

General refactoring:

(0 Selected Refactoring option has been marked)

11. Combine the showmap function of GameEngin and Editmap of previous version, so that the program can view the map by calling showmap function.
12. Improve the Mainloop constructor, so that we do not need to input a map file when we define the mainloop as the parameter in the main function.
13. combining the new function card in order creation phase (like Bomb, Blockade, etc.) into the issue order phase, so that the user can use the function card durning issue order state.

Refactoring Command Pattern:

(1 Selected Refactoring option has been marked)

14. Command Pattern allows us to hide the logic of order execution from the GameEngine. It also makes it easier for us to add more Order types to the game later if needed.
15. Combine function printInvalidCommandIncurrentPhase() and printInvalidArgument() into one method, so that the program will print the same error message when the user enters Invalid command arguments or Invalid command in the current game phase.

Related Tests

DeployOrderTest: testOrderInvalidGivenNullDestination,
testOrderInvalidGivenNegativeArmyNumber,
testOrderInvalidGivenDestinationNotInControl, testOrderValid

OrderFactoryTest: testDeployOrderCreation