

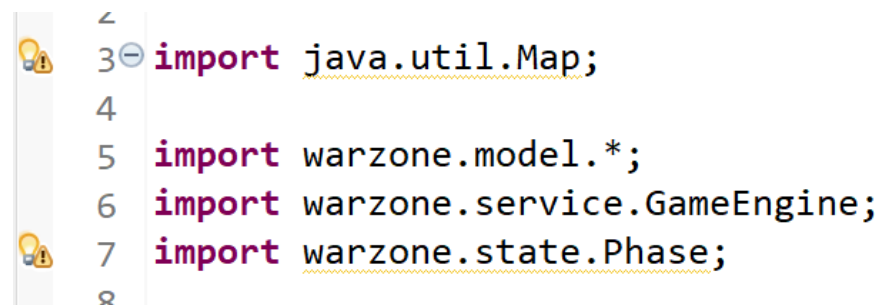
Find the refactoring targets

- Review the Project of Build one
- Brain storm after receiving the new spec of build 2#

1. Potential refactoring targets

1.1 Removing some unnecessary import in the head of the file:

After reviewing the codes, we find that we have many imported packages that are never used in some java files, so we can remove them.

A screenshot of a code editor showing the import section of a Java file. Line 3 has 'import java.util.Map;' with a lightbulb icon to its left. Line 5 has 'import warzone.model.*;'. Line 6 has 'import warzone.service.GameEngine;'. Line 7 has 'import warzone.state.Phase;' with a lightbulb icon to its left. Line 8 is empty. The code is syntax-highlighted with colors: 'import' in purple, package names in blue, and class names in black.

```
2
3 import java.util.Map;
4
5 import warzone.model.*;
6 import warzone.service.GameEngine;
7 import warzone.state.Phase;
8
```

1.2 simplifying the code in MapService.java:

After discussion, we know that there are some conditions check before the real operation on the object. We can put there condition check into another method and let it return the result of validation to make the current method only process the operation.

```
// condition1: check if more than one country
int l_countryCount = d_gameContext.getCountries().size();
if ( l_countryCount <= 1 ) {
    GenericView.printError("The map should contain more than one country.");
    return false;
}
// condition2: check if each country belongs to one continent
for (Country l_countryTemp : d_gameContext.getCountries().values()){
    if(l_countryTemp.getContinent() == null) {
        GenericView.printError("Each country should belong to one continent.");
        return false;
    }
}
// condition3: check if more than one continent
Map<Integer, Continent> l_continent = d_gameContext.getContinents();
if ( l_continent.size() <= 1 ) {
    GenericView.printError("The map should contain at least one continent.");
    return false;
}
// ...
```

1.3 hard coding in RouterService.java:

Through discussion, we find that the Action only has two types, '-add' and '-remove'. We can use enumeration to represent it.

```
283     for(Action l_action: l_actions) {
284         //todo add it in the property file
285         String l_actionArray = "-add,-remove";
286         if(l_actionArray.indexOf(l_action.getAction()) > -1) {
287             l_routers.add(new Router(l_controllerName, l_action.getAction(), l_action.getParameters()));
288             GenericView.printDebug("Add an action to a router");
289         }
290         else {
291             l_routers = new LinkedList<Router>();
292             l_routers.add(createErrorRouter(ErrorType.BAD_OPTION.toString()));
293             GenericView.printDebug("Meet an error when adding an action to a router");
294             return l_routers;
295         }
296     }
```

1.4 simplifying the method in MapService.java:

By reviewing the code, we find that we can move this operation to another method. We should avoid a method with too many lines and logic. If so, we can separate the responsibility to other methods.

```
//build the content using StringBuilder
StringBuilder l_map = new StringBuilder();
l_map.append("; map: " + l_fileName);
l_map.append("\n; map made with the 6441 Super Team");
l_map.append("\n; 6441.net 1.0.0.1 ");
l_map.append("\n");

l_map.append("\n[files]");
l_map.append("\npic " + l_fileName + "_pic.jpg");
l_map.append("\nmap " + l_fileName + "_map.gif");
l_map.append("\ncrd " + l_fileName + "europe.cards");
l_map.append("\n");
Map<Integer,Integer> l_continentIdMapping = new HashMap();
l_map.append("\n[continents]");
```

1.5 change vague method names in MapService.java:

After reviewing the code, we find that the name of this method is not straightaway. The DFS is the name of an algorithm. We can use its full name(depth first search).

```
private void innerDFS(int p_cur, int p_low[], int p_disc[],
                    boolean p_stackMember[], Stack<Integer> p_st, LinkedList<Object>[] p_list,
                    List<List<Integer>> p_resList) {
```

1.6 Unclear naming of methods in NeighborService.java

By reviewing NeighborService.java, we find that the function – add() and remove() – is named unclearly in this file is unclear. We suggest changing it into addNeighbor() and removeNeighbor().

```
31 public boolean add(int p_countryID, int p_neighborCountryID) {
32     Map<Integer, Country> l_countries=d_gameContext.getCountries();
33     if(!l_countries.containsKey(p_countryID) || !l_countries.containsKey(p_neighborCountryID))
34         return false;
35 }
36
43 /**
44  * This method will remove a country from its neighbor.
45  * @param p_countryID the ID of the country that should be removed
46  * @param p_neighborCountryID the ID of its neighbor
47  * @return the result of operation
48  */
49 public boolean remove(int p_countryID, int p_neighborCountryID) {
50     Map<Integer, Country> l_countries=d_gameContext.getCountries();
51     if(!l_countries.containsKey(p_countryID) || !l_countries.containsKey(p_neighborCountryID))
52         return false;
53 }
```

1.7 Never used local members in Country.java

Some private members are not necessary, such as xPosition and yPosition in map. Though it is recorded in some of the map file, it is not used at all in our implementation. Therefore, they could be removed as a refactoring.

```
11 public class Country {
12
13     private int d_countryID;
14     private String d_countryName;
15     private Player d_owner;
16     private int d_xPosition;
17     private int d_yPosition;
18     private int d_armyNumber = 0;
19     private Map<Integer, Country> d_neighbors;
20     private Continent d_continent;
21     private CountryState d_countryState;
22 }
```

1.8 Never used local members in StartupService.java

We review the StartupService.java and find that: in function assigncountries(), there is a local value l_ctr never actually used. It should be removed.

```

304 //Looping variables
305 Country l_country;
306 Player l_player;
307 int l_ctr = 0;
308 int l_playerIndex = 0;
309
310 //Loop through each country to assign to a random player
311 for(Integer l_countryID : l_countryIDs) {
312     //Reset the index once each player has been assigned a country
313     if(l_playerIndex >= l_playerNames.size()) {
314         l_playerIndex = 0;
315     }
316
317     l_country = d_gameContext.getCountries().get(l_countryID);
318     l_player = d_gameContext.getPlayers().get(l_playerNames.get(l_playerIndex));
319
320     l_country.setCountryState(CountryState.Occupied, l_player);
321
322     //Update the looping variables
323     l_playerIndex++;
324     l_ctr++;
325 }
326
327 return true;
328 }

```

1.9 Unsuitable construction of LinkedList in MapService.java

We read from the warning and find that the construct of the `l_continntAdjList` in `validateSubGraph()` is not a good practice of java language.

```
l_continentAdjList = new LinkedList[l_continent.size()];
```

It should be:

```
LinkedList<Object> b = new LinkedList<>();
```

```
LinkedList<LinkedList<Object>> a = new LinkedList<>();
```

```
a.add(b);
```

1.10 Unconventional naming of method in MapService.java

One of the experienced developer in our group indicate that the `ifConnected()` method is not following the naming convention as usual. It should be `isConnected()`.

```
public boolean ifConnected(int p_size, LinkedList<Object>[] p_list) {
```

For other 5 refactoring target, we will discuss in section 2.

2. Actual refactoring targets

2.1 Refactoring architecture using state pattern:

- Reason: The game can be divided into different phases, we can use different states to represent them so that the same command can perform different actions in different phases. For example, if we want to get some help, we can input 'help' . We can overwrite the 'help()' function in each subclass of state class to give hints according to the current phase to the player, because each phase has its own order that cannot be used in other phases.
- Compare:

We use MVC pattern in the first version to process the actions of players. For instance, if a player wants to get some help at the beginning of the game, he enters 'help' into the console. There is a field to indicate current phase in the game context. The controller will invoke the 'printHelp()' method in the service. The trouble is this method can be very complicated and hard to maintain in the future, because we need to write all different content into the method. The code can be following:

```
public void printHelp() {  
    if (d_currentState == "MapEditor") {  
        // print all commands that can be used in the MapEditor phase  
    }  
    else if (d_currentState == "GamePlay") {  
        // print all commands that can be used in the GamePlay phase  
    }  
    // .....  
    else {  
    }  
}
```

The situation will be better when we refactor the architecture by state pattern. We can create different classes for every phase in the game, and overwrite the 'printHelp()' methods. The responsibility is divided into the corresponding class, so the code is easier to understand and maintain after refactoring. The code can be following:

```
public abstract class Phase {  
  
    abstract public void printHelp();  
  
}
```

```

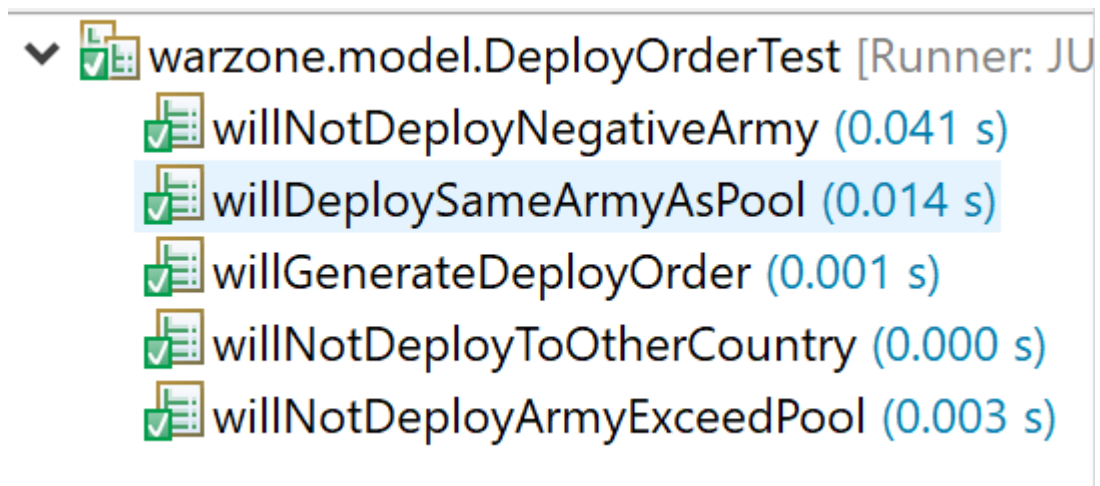
public class MapEditor extends Phase {

    public void printHelp() {
        // print all methods that can be used in the MapEditor phase
    }

    public class Reinforcement extends Gameplay {
        public void printHelp() {
            // print all methods that can be used in the Reinforcement phase
        }
    }
}

```

- Test Cases: there is just 'deploy' order in the first scrum.



2.2 Refactoring architecture using command pattern:

- Reason: the game engine executes the orders from players in round-robin fashion, which are stored in the list of player. The execution of the commands is at a different time compared to where they are created. Besides, we do not want to know what the actual command we are executing now. If we use command pattern, we can use a clear and unified interface to execute the command.
- Compare:

If we do not use command pattern, we have to execute a certain command immediately after it is created. The extension of the code will be very awful. We will find it is very difficult to add new commands, delete and modify the existing orders. The code will be following:

```

public void execute(String command) {
    switch(command){
        case "deploy":
            CommandService.executeDeploy(command);
        case "advance":
            CommandService.executeAdvance(command);
            // ...
    }
}

```

It is a different situation when we use command pattern. We can encapsulate the concrete command into different instances and initiate them with parameters. We can just return the concrete instance of the command after they are created, so we can execute the command at a different time and site. We can also do more operations on it, such as storing or reverting it in the future. The code can be following:

```

public abstract class Order {

    /**
     * current Order Type
     */
    protected OrderType d_orderType;
    /**
     * command which create this order
     */
    protected String d_command;

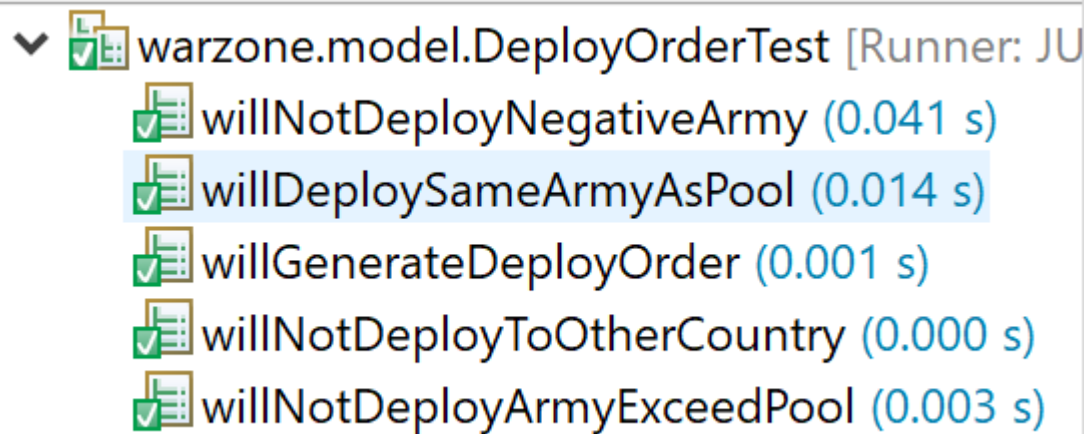
    /**
     * Current Game Context
     */
    protected GameContext d_gameContext;

    /**
     * get Order Type
     * @return Order Type
     */

    switch(l_orderName.toLowerCase()){
        case "deploy":
            return createDeployOrder(l_commandInfos);
        case "advance":
            return createAdvanceOrder(l_commandInfos);
        case "bomb":
            return createBombOrder(l_commandInfos);
        case "blockade":
            return createBlockadeOrder(l_commandInfos);
        case "airlift":
            return createAirliftOrder(l_commandInfos);
        case "negotiate":
            return createNegotiateOrder(l_commandInfos);
    }
}

```

- Test Cases: there is just 'deploy' order in the first scrum.



2.3 SimplifyingMapEditor

- Reason: when we edit map, we have different operations for adding, removing and modifying continent, country and borders respectively. The code in the first version is difficult to understand, which uses four Boolean field to indicate the current stage. We can use four enumerations to do that thing.
- Compare:

If we do not refactor out codes, we need a lot of if-else to process continent, country and borders respectively, and we also need extra fields to indicate the current operated object to use different methods for it. The code will be difficult to maintain and extent in the future. The code is following:


```

//use boolean to record the different parts in file
boolean l_processingFiles = false;
boolean l_processingContinents = false;
boolean l_processingCountries = false;
boolean l_processingBorders = false;
LoadMapPhase l_loadMapPhase = null;

while (l_scanner.hasNextLine()) {
    String l_line = l_scanner.nextLine();

    // determine which part it is
    // file part
    if(l_line.equals("[files]")) {

        l_processingFiles = true;
        l_processingContinents = false;
        l_processingCountries = false;
        l_processingBorders = false;

        l_loadMapPhase = LoadMapPhase.FILES;
        l_line = l_scanner.nextLine();
    }

    // continents part
    else if(l_line.equals("[continents]")) {

        l_processingFiles = false;
        l_processingContinents = true;
        l_processingCountries = false;
        l_processingBorders = false;

        l_loadMapPhase = LoadMapPhase.CONTINENTS;
        l_line = l_scanner.nextLine();
    }
    // ....
}

```

Things will change after refactoring it. We use a single field to indicate the current state. When we want to add something into the map, what we need to do is to add a new enumeration into the class. The code can be following:

```

while (l_scanner.hasNextLine()) {
    l_line = l_scanner.nextLine();

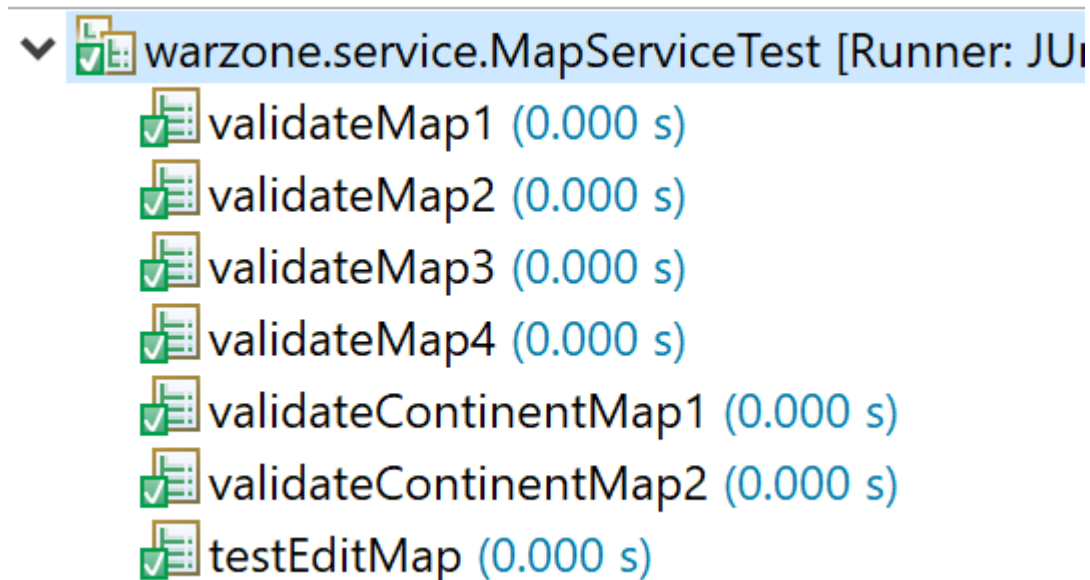
    // determine which part it is
    // file part
    if(l_line.equals("[files]")) {

        l_loadMapPhase = LoadMapPhase.FILES;
        l_line = l_scanner.nextLine();
    }
    // continents part
    else if(l_line.equals("[continents]")) {

        l_loadMapPhase = LoadMapPhase.CONTINENTS;
        l_line = l_scanner.nextLine();
    }
    //....
}

```

- Test cases:



2.4 Erasing the hard coding:

- Reason: hard coding should be avoided in every program, because it will be confusion after changing our codes frequently. A good solution is to define the part that changes frequently or will change in the future. We can put this part into the configuration files. If we need to change this part, we can modify it in the configuration file directly, so we can avoid changing our code.
- Compare:

As we mention in 2.4, we categorize commands into two types, simple commands and complex commands. The first version of our code is hard coding. We construct two

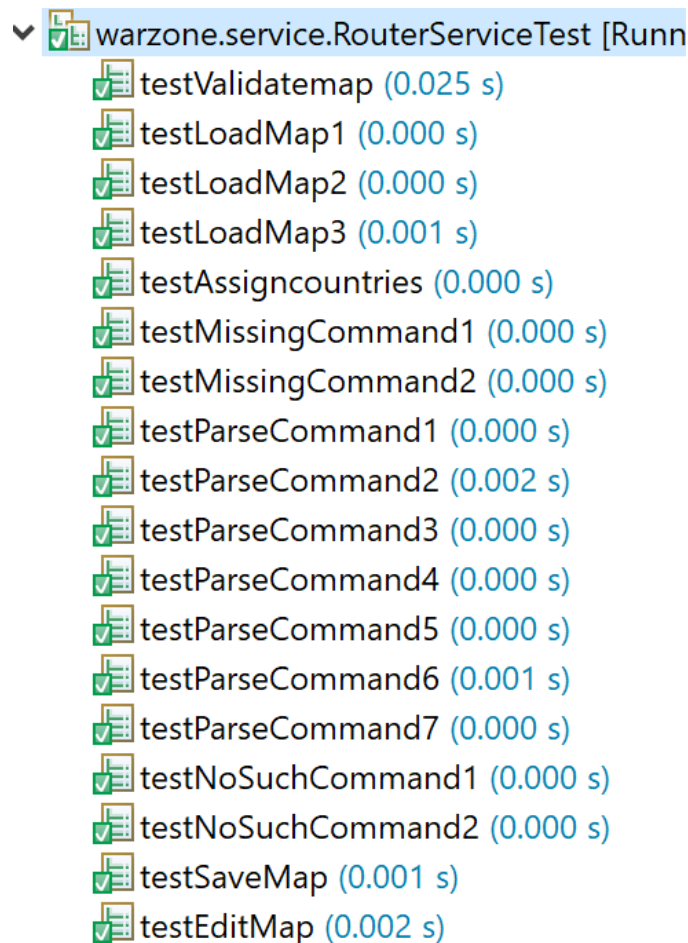
strings containing all commands, and we check the entered command whether is in the string to get the type of the command. The code is following:

```
String l_complexCommand = ",editcontinent,editcountry,editneighbor,gameplayer,";  
String l_simpleCommand = ",loadmap,editmap,savemap,assigncountries,validatemap,showmap,help,"  
    + "qamode,play,next,reboot,startup,mapeditor,";
```

To avoid hard coding, we can put them into the configuration file. It is easier to modify the code in the future, and we do not need to change our codes when we add new commands.

```
#command type  
complexCommand=",editcontinent,editcountry,editneighbor,gameplayer,";  
simpleCommand=",loadmap,editmap,savemap,assigncountries,validatemap,showmap,help,qamode,play,next,"
```

- Test Cases:



The screenshot shows a test runner interface with a tree view. The root node is 'warzone.service.RouterServiceTest [Runn' (partially visible). It contains 20 sub-items, each representing a test case. Each test case is marked with a green checkmark icon and a duration in parentheses. The test cases are: testValidatemap (0.025 s), testLoadMap1 (0.000 s), testLoadMap2 (0.000 s), testLoadMap3 (0.001 s), testAssigncountries (0.000 s), testMissingCommand1 (0.000 s), testMissingCommand2 (0.000 s), testParseCommand1 (0.000 s), testParseCommand2 (0.002 s), testParseCommand3 (0.000 s), testParseCommand4 (0.000 s), testParseCommand5 (0.000 s), testParseCommand6 (0.001 s), testParseCommand7 (0.000 s), testNoSuchCommand1 (0.000 s), testNoSuchCommand2 (0.000 s), testSaveMap (0.001 s), and testEditMap (0.002 s).

2.5 Aggregate separate actions into one method

- Reason: Every time we change the player of a country, we will have to change the owner in Country and the conquered list in Player. We decide to refactor this

because it may cause a disaccord between data if someone forget to do one of the actions.

- Compare:

Before refactoring, each time we set a player, we have to both call the `setOwner()` method and put the country into the `conqueredCountries` list.

```
public void setOwner(Player p_owner) {  
    this.d_owner = p_owner;  
}
```

```
l_country.setOwner(l_player);  
l_player.getConqueredCountries().put(l_country.getCountryID(), l_country);
```

After refactoring, we will do these two tasks into `setOwner`. Also, we have more validation checked before actions.

```
/**  
 * set the owner of the country  
 * @param p_owner the Player who owns the country  
 */  
public void setOwner(Player p_owner) {  
    if (d_owner != null)  
        d_owner.getConqueredCountries().remove(this.getCountryID(), this);  
    this.d_owner = p_owner;  
    if (p_owner != null)  
        p_owner.getConqueredCountries().put(this.getCountryID(), this);  
}
```

- Test Cases:

✓ CountryTest (warzone.model)	15 ms
✓ testSetOwnerSuccess	15 ms