SOEN 6441 - Advanced Programming Practices

Submitted To: Prof. Joey Paquet

Submitted By: Group 22

Team Members	
Student ID	**Name**
40289562	Inderjeet Chauhan
40292088	Saphal Ghimire
40256283	Apoorva Sharma
40238072	Vibhor Gulati
40279035	Mohammad Zaid Shaikh

Potential refactoring targets

Before moving from Build I to II, we noted 20 potential refactoring targets that would improve the performance and quality of the code.We also have considered reusability, implementation, single responsibility principles, simplification of complex functions, code duplication, proper use of object oriented concepts and many more. We found a few optimizations that we can do in edit country, edit country and loadmap functions. Also, we realized that we have not used exceptions for error handling, so we kept it as a potential refactoring target. We also saw the need of using interfaces more extensively to define consistent contracts for classes, improving code flexibility and promoting object oriented programming. We needed to implement single responsibility principles and remove overused codes like if else and switch statements. We were talking inputs from multiple places and we needed it to be refactored to be handled from the same place.

The potential refactoring targets are listed below:

1. Refactor Edit Continent
2. Refactor Edit Country
3. Refactor Loadmap
4. Create exceptions for MapErrors and other errors
5. Refactor executeCommand function
6. Proper Command Validation Service
7. Use of Interfaces
8. Improve Documentation
9. Apply Single Responsibility Principles
10. Overuse of if else and switch statements and change certain data types.
11. Remove complexity and simplify function
12. Implement logger functionalities
13. Improve command validation
14. Remove Reinforcement as service
15. Increase use of getter and setters
16. Remove inputs from Player class

17. Generalize Order methods
18. Cleanup game controller and have proper turn management
19. Increase null value check
20. Generalize the process of taking inputs in different phases

Actual refactoring targets

From the 20 major refactoring targets, we selected 5 refactoring targets to work on. As we had to implement state pattern in phase, command pattern in orders and observer pattern for loggers, we understand the importance of refactoring command pattern. The command validator that is designed should be able to handle all the cases that are required and should accept multiple arguments in the command.

Major Refactoring Targets:

1. **validateCommand()**
    a. Hints for the correct command
    b. Accepts multiple arguments in the command
    c. Returns Command[]

2. **executeDeploy() -> execute()**
    a. Changed to set orders
    b. Managed to implement command patterns to handle orders from a single place.

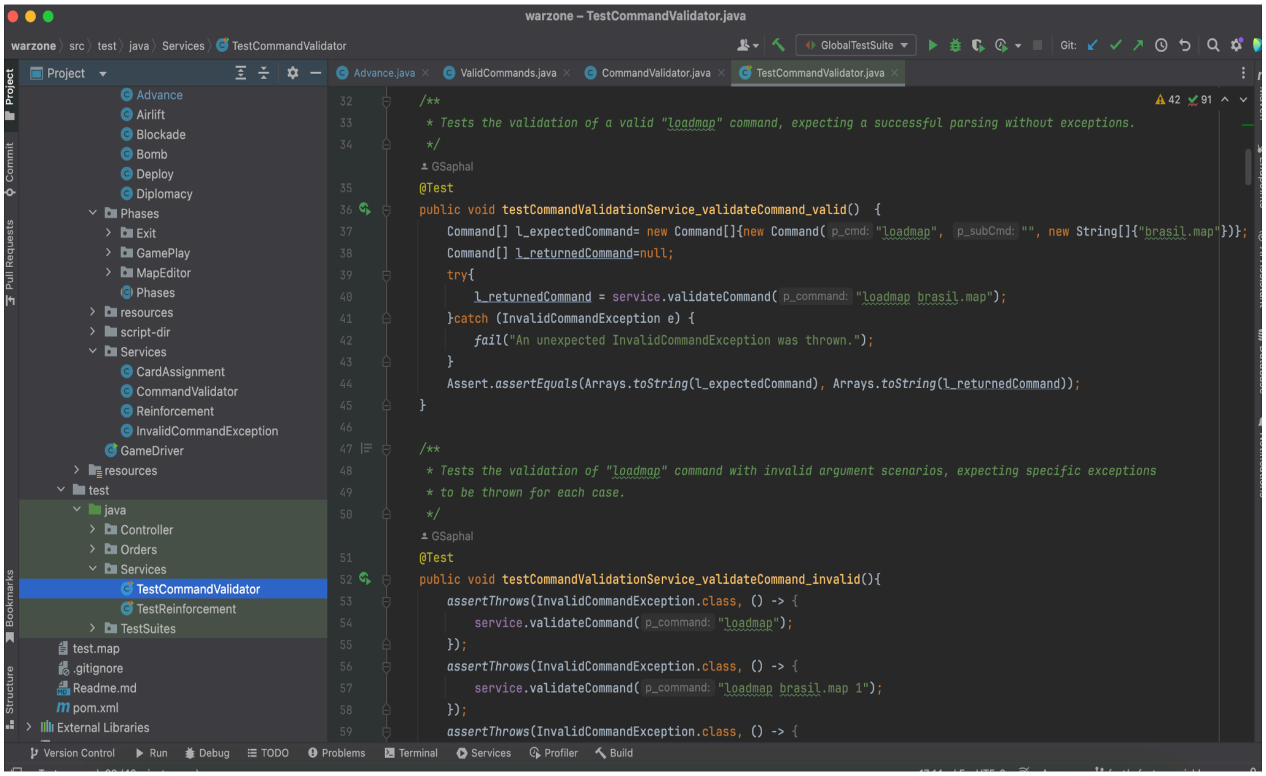
3. **issueOrder()**
    a. Removed command inputs from the function and moved it to CommandValidator
    b. Now parses the correct orders from a valid command object.

4. **nextUserInput()**
    a. Removed repetition of user input functions from Game Controller and Players and moved it to a central command validator
    b. Throws a proper InvalidCommandException
    c. Accepts a ValidCommand data type, so it can be extended to add any commands

5. **Type of Order list changed from Queue to Array List**
    a. Consists of OrderObject i.e Deploy, Advance, Airlift, Bomb and Diplomacy

validateCommand()

The refactored validateCommand() method now provides hints for the correct command and accepts multiple arguments. It returns an array of Command objects, allowing for more flexibility in handling complex commands with multiple parameters. This approach streamlines command validation by providing clear guidance to users on the correct command format while accommodating various command structures. Additionally, the use of Command objects enhances modularity and enables easier integration with other components of the system, promoting code reusability and maintainability.


warzone – TestCommandValidator.java
warzone src test java Services TestCommandValidator
GlobalTestSuite
Advance.java
ValidCommands.java
CommandValidator.java
TestCommandValidator.java

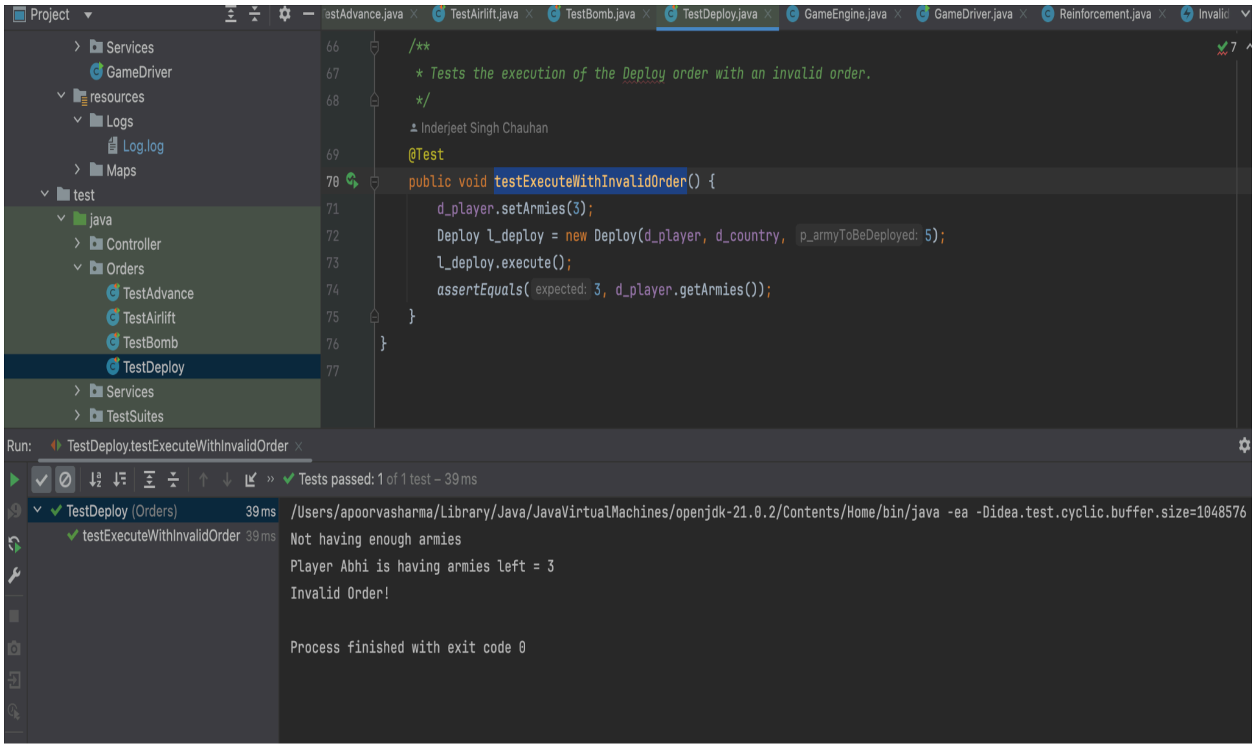
```java
/**
 * Tests the validation of a valid "loadmap" command, expecting a successful parsing without exceptions.
 */
@Test
public void testCommandValidationService_validateCommand_valid()  {
    Command[] l_expectedCommand= new Command[]{new Command("loadmap", "", new String[]{"brasil.map"})};
    Command[] l_returnedCommand=null;
    try{
        l_returnedCommand = service.validateCommand("loadmap brasil.map");
    }catch (InvalidCommandException e) {
        fail("An unexpected InvalidCommandException was thrown.");
    }
    Assert.assertEquals(Arrays.toString(l_expectedCommand), Arrays.toString(l_returnedCommand));
}

/**
 * Tests the validation of "loadmap" command with invalid argument scenarios, expecting specific exceptions
 * to be thrown for each case.
 */
@Test
public void testCommandValidationService_validateCommand_invalid(){
    assertThrows(InvalidCommandException.class, () -> {
        service.validateCommand("loadmap");
    });
    assertThrows(InvalidCommandException.class, () -> {
        service.validateCommand("loadmap brasil.map 1");
    });
    assertThrows(InvalidCommandException.class, () -> {
```

executeDeploy() -> execute()

The method executeDeploy() has been refactored to execute() within the Orders interface, aligning with the command pattern. This change centralizes order execution logic within the interface, allowing different order types to implement their specific execution behavior. By consolidating order execution under a single method, the codebase becomes more maintainable and extensible, enabling easier addition of new order types while ensuring consistent execution semantics across all orders.

This upgrade enhances modularity, clarity, and extensibility in order handling, allowing for better validation, execution, and reporting of game actions.

Screenshot after refactoring the executeDeploy() method to execute() method.



```
/**
 * Tests the execution of the Deploy order with an invalid order.
 */
@Test
public void testExecuteWithInvalidOrder() {
    d_player.setArmies(3);
    Deploy l_deploy = new Deploy(d_player, d_country,  p_armyToBeDeployed: 5);
    l_deploy.execute();
    assertEquals( expected: 3, d_player.getArmies());
}
}
```

Tests passed: 1 of 1 test – 39 ms

```
/Users/apoorvasharma/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java -ea -Didea.test.cyclic.buffer.size=1048576
Not having enough armies
Player Abhi is having armies left = 3
Invalid Order!

Process finished with exit code 0
```

issueOrder()

The issueOrder() method has been updated to improve its functionality and maintainability. Previously, this method directly accepted command inputs, leading to tightly coupled code and potential duplication of logic across different parts of the system.

With the latest changes, the responsibility for validating commands and parsing them into orders has been delegated to a new component called CommandValidator. This separation of concerns enhances code organization and promotes reusability, as the command validation and parsing logic can now be centralized and easily maintained.

Now, the issueOrder() method solely focuses on executing orders based on validated and parsed commands obtained from the CommandValidator. This decoupling ensures cleaner, more modular code, making it easier to understand, test, and extend the order issuance process.

nextUserInput()

The refactored nextUserInput() method removes the repetition of user input functions from both the GameController and Player classes, consolidating them into a central command validator. This centralization promotes better code organization and maintenance by eliminating redundant code and ensuring consistency across the application.

The method now throws a proper InvalidCommandException to handle cases where the user inputs an invalid command, enhancing error handling and providing meaningful feedback to users.

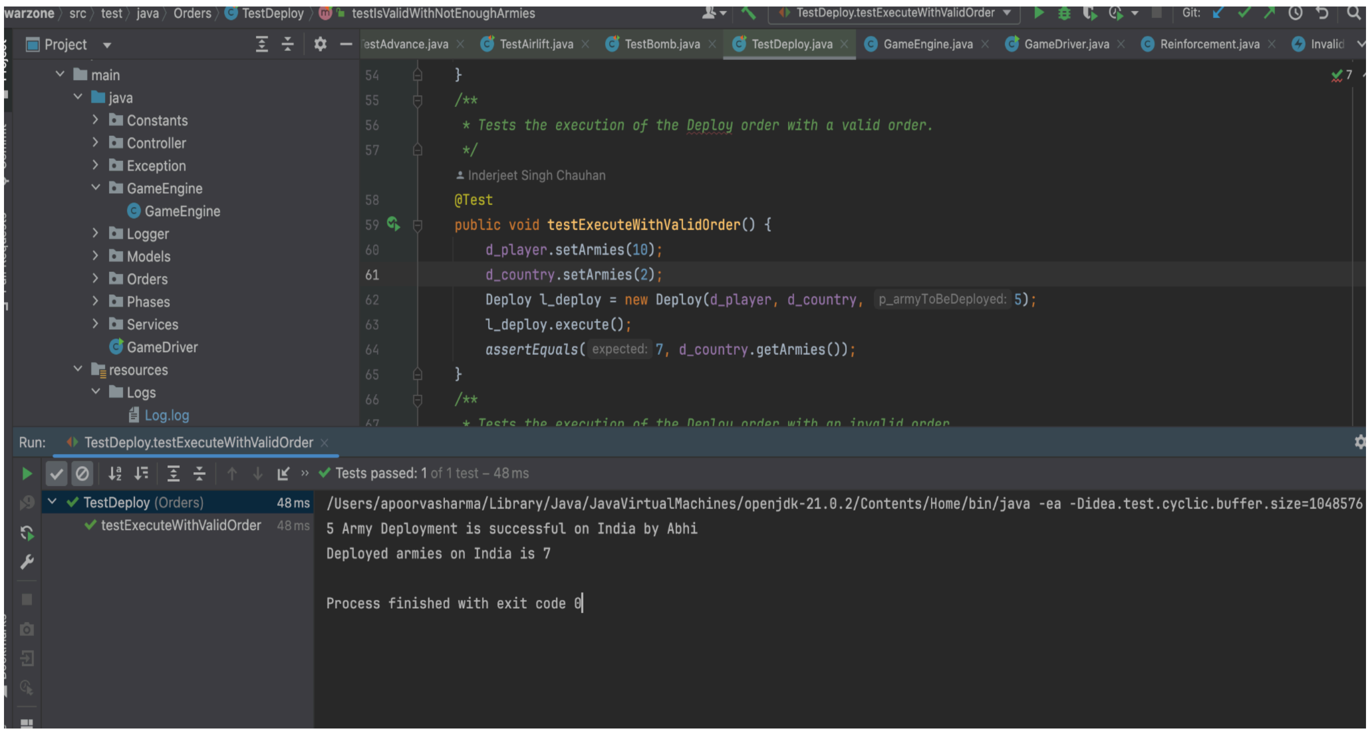
Furthermore, nextUserInput() now accepts a ValidCommand data type, allowing for greater flexibility and extensibility. This means that the method can be easily extended to support additional commands in the future without requiring significant modifications to its implementation. Overall, these changes improve code modularity, readability, and scalability.

Type of Order list changed from Queue to Array List

Previously, the player's object utilized a Queue to store orders. These orders were retrieved sequentially using the next_order() method and subsequently executed. After execution, the outcomes were verified through assertions.

This refactoring was motivated by the observation that the time complexity of finding elements in an ArrayList is more efficient compared to that of a Queue. This improvement is attributed to the direct access provided by ArrayList's indexing strategy.

Screenshot after refactoring Orderlist from Queue to Array List.



```java
    }
    /**
     * Tests the execution of the Deploy order with a valid order.
     */
    @Test
    public void testExecuteWithValidOrder() {
        d_player.setArmies(10);
        d_country.setArmies(2);
        Deploy l_deploy = new Deploy(d_player, d_country, p_armyToBeDeployed: 5);
        l_deploy.execute();
        assertEquals( expected: 7, d_country.getArmies());
    }
    /**
     * Tests the execution of the Deploy order with an invalid order.
```

Tests passed: 1 of 1 test – 48 ms

```
/Users/apoorvasharma/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java -ea -Didea.test.cyclic.buffer.size=1048576
5 Army Deployment is successful on India by Abhi
Deployed armies on India is 7

Process finished with exit code 0
```