SOEN 6441 (Advanced Programing practices)

Department of Computer Science and Software Engineering

Build 02: Refactoring

Professor:
Joey Paquet

Team 01:
Niloufar Pilgush
Nasrin Maarefi
Jerome Kithinji
Ali Sayed Salehi
Fateme Chaji

March 2024

Contents

Build 02/Refactoring

1.1 Refactoring process

1.1.1 Potential refactoring targets:

Following mentioned are some of the refactoring targets observed while implementation of build1 and enhancements that are required for build2:

1. Usage of double-ended queues for storing the list of orders.
2. Implementation and verification of correct state transition from current state.
3. Implementation of command pattern to implement the different types of orders.
4. Implementation and registration of Observers for log file and console.
5. Refactoring for storing `OrderDetails` as a separate POJO instead of the `Order` class.
6. Implementation of probability of attackers and defenders.
7. Standardisation of Java project packages.
8. Including xDocLinting for all JavaDocs in Maven.
9. Refactoring of logging for effective gameplay.
10. Implementation of test cases for uncovered scenarios.
11. Improved naming of test cases (scenario + expectation).
12. Include more comments for better understandability and modifiability.
13. Usage of warzone constants instead of hard-coded strings.
14. Making JavaDoc compilation a mandatory step in build.
15. Refactoring for unused imports and indentation.

1.1.2 Actual Refactoring targets

Following are the actual refactoring operations which were important from the perspective of time complexity (Usage of Queues), code management (Inclusion of Order Details), more usage of OOPS principles (Polymorphism for Orders), testing of code (better naming (expectation + scenario)), understandability and maintainability of code (inclusion of comments). These refactoring targets aim to enhance the code quality, efficiency, and maintainability which are important software development principles.

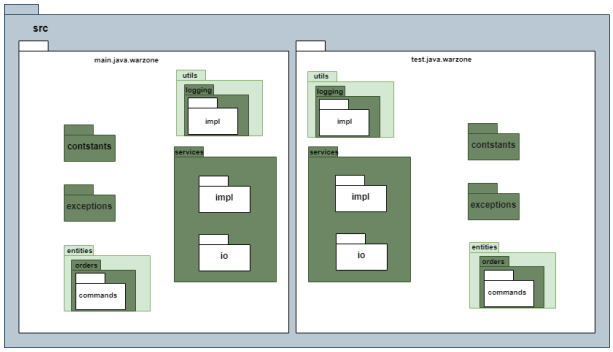


Figure 1.1: Package Diagram before refactoring

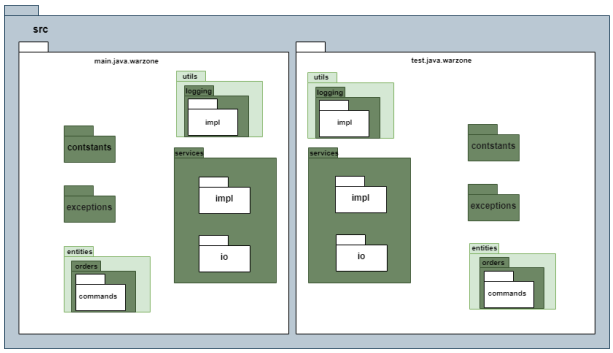


Figure 1.2: Package Diagram after refactoring

1.2 Changing Package Structure

Reorganizing the package structure, particularly within the "entities" package, to introduce a new subpackage named "orders" and further, a subpackage within it named "commands". This reorganization aimed to cleanly implement the command pattern, allowing for a more structured approach to handling various game orders such as advance, deploy, bomb, blockade, airlift, and diplomacy commands.

Modifying the "utils" package to include a new subpackage named "logging" and another subpackage within it named "impl" to implement the observer pattern. This change was directed towards creating a more robust and flexible logging mechanism where log entries could be observed and handled by different logging entities like console loggers and game loggers.

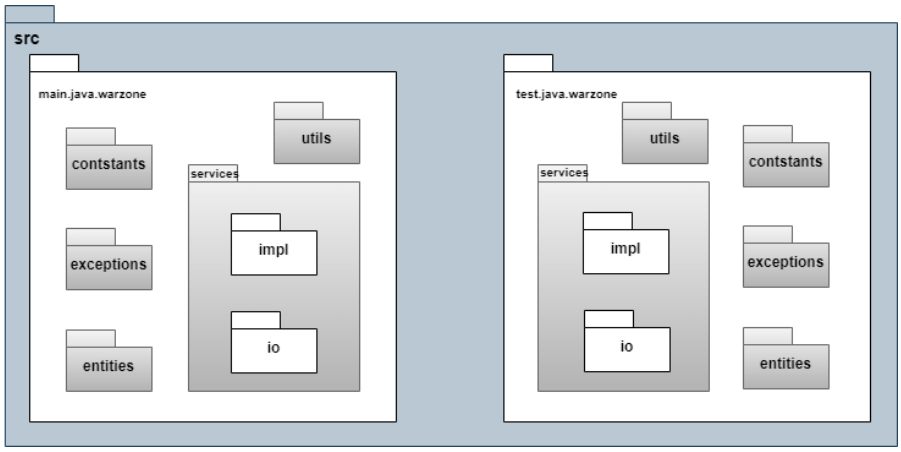


Figure 1.3: Package Diagram before refactoring

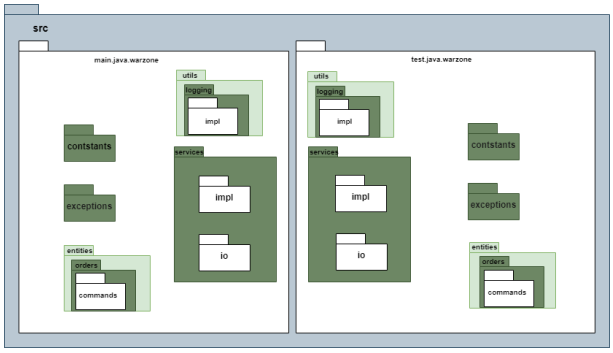


Figure 1.4: Package Diagram after refactoring

1.2.1 entity package

Adding new subpackage named “orders” and subpackage named “commands” to orders to implement command pattern design.

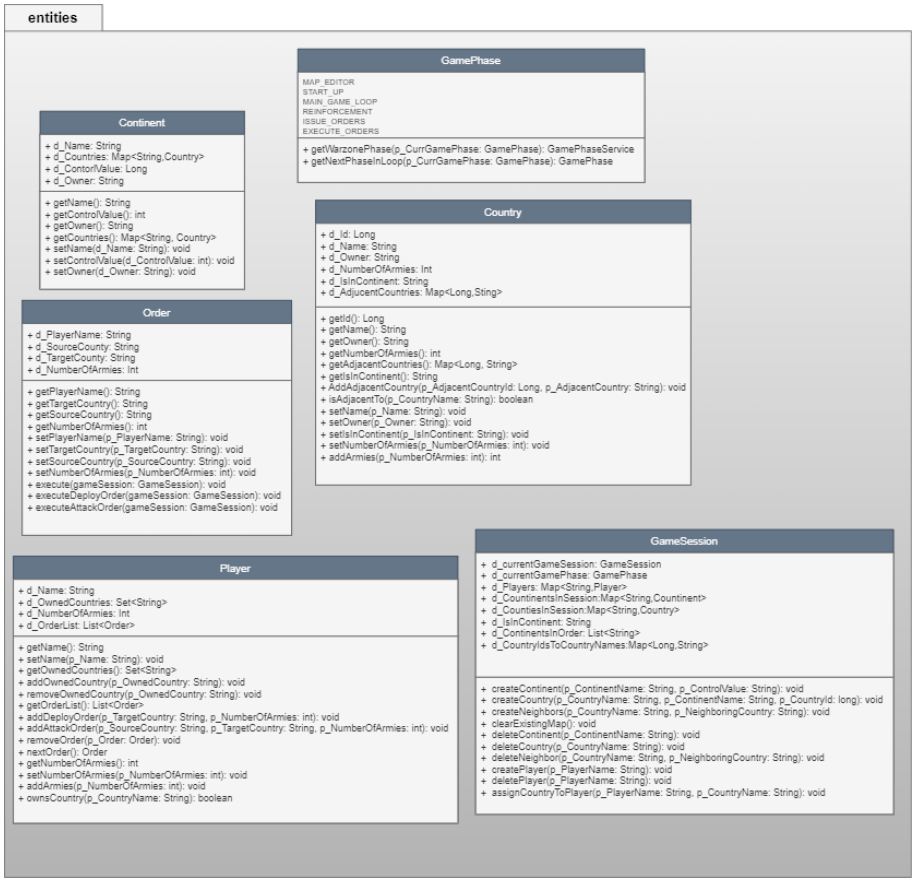


Figure 1.5: Entity Package Diagram before refactoring

1.2.2 util package

Changing in “utils” package and adding new subpackage named “logging” and subpackage named “impl” to logging to implement observer pattern design.

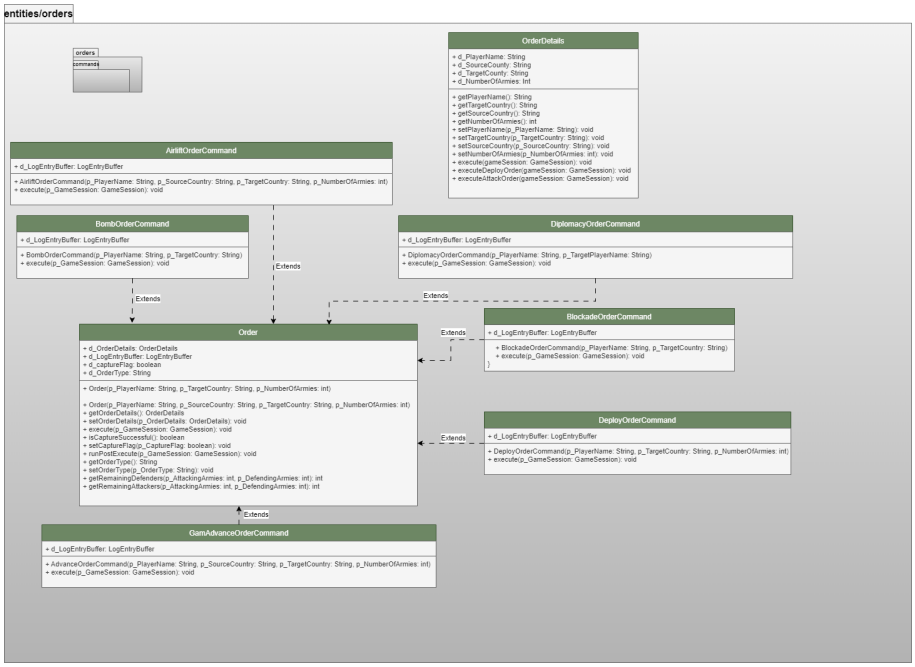


Figure 1.6: Entity Package Diagram after refactoring

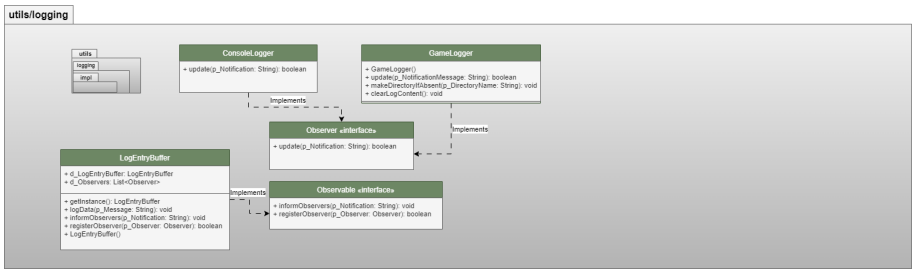


Figure 1.7: Util Package Diagram after refactoring

1.3 Implementing Pattern Designs

1.3.1 Observer Pattern

The Observable interface is implemented by objects that wish to notify observers about changes.

The `LogEntryBuffer` class implements the observable interface and acts as the observable entity. It maintains a list of observers and notifies them about new log messages through the `informObservers` method. Both `ConsoleLogger` and `GameLogger` implement the observer interface and act as observers that perform specific actions when they are notified of log messages by the `LogEntryBuffer`.

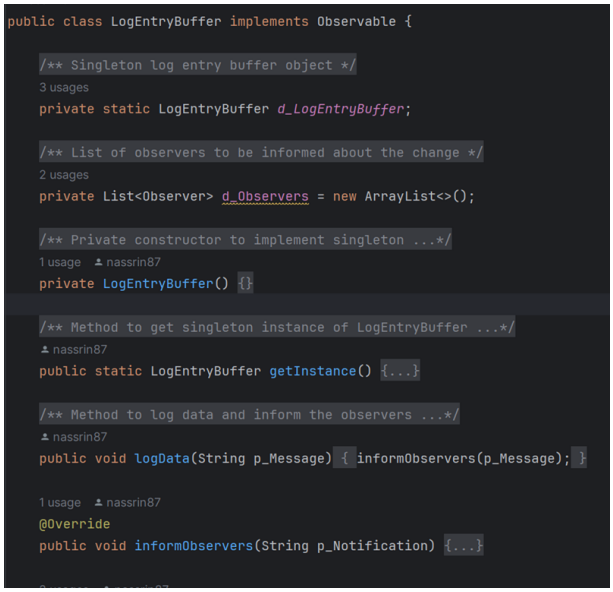


Figure 1.8: LogEntryBuffer Class

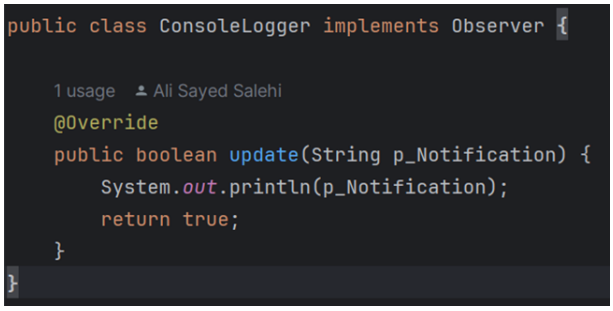


Figure 1.9: ConsoleLogger Class

1.3.2 Command Pattern

The abstract `Order` class acts as the Command interface in the Command Pattern, declaring the execute method that all concrete commands (AdvanceOrderCommand, DeployOrderCommand, etc.) will implement.

Each of these classes encapsulates all the information needed for the execution of a specific order, such as advancing troops, deploying armies, or negotiating diplomacy. The `Player` class acts as the invoker.

The `AdvanceOrderCommand`, `AirliftOrderCommand`, `BlockadeOrderCommand`, `BombOrderCommand`, `DeployOrderCommand`, `DiplomacyOrderCommand` classes are concrete implementations of the `Order` abstract class.

It holds and manages orders (commands) through methods like `addDeployOrder`, `addAttackOrder`, etc. These methods create command objects and add them to the player's order list.

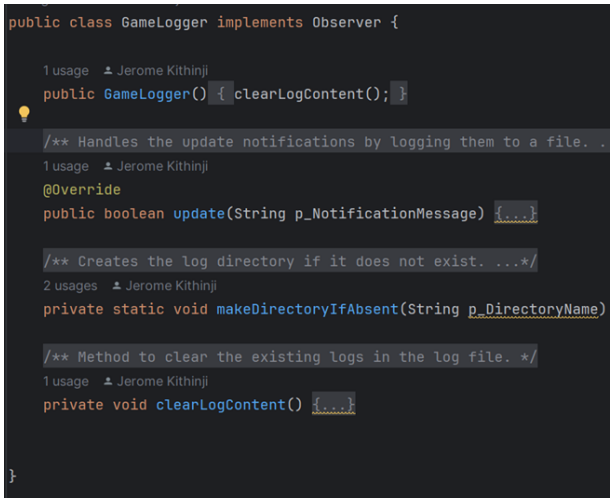


Figure 1.10: GameLogger Class

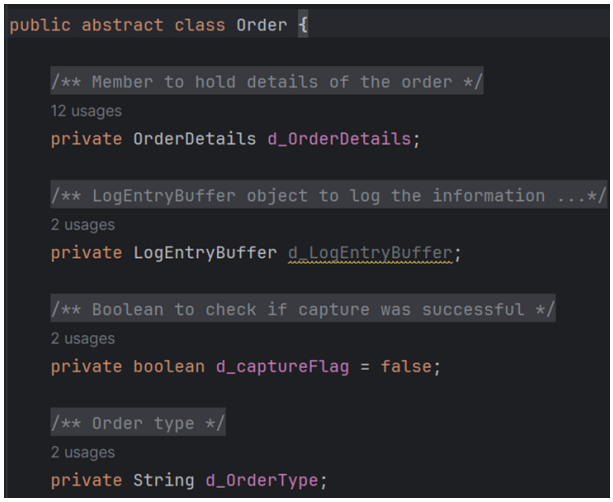


Figure 1.11: Command Pattern: Order Class Hierarchy

1.4 Changing Data Structure

Using doubly-ended queues for inserting and retrieving the Orders instead of a list helps to improve the time complexity of the order execution process significantly. The adoption of doubly-ended queues (Deques) facilitates more efficient operations at both ends of the queue, which is particularly advantageous when dealing with a dynamic set of orders that may need quick updates or retrieval. This structural change is aimed at enhancing the efficiency and performance of the order management system within the software.

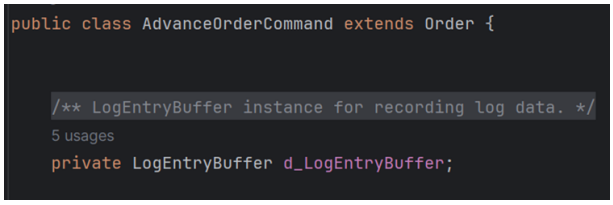


Figure 1.12: AdvanceOrderCommand Class

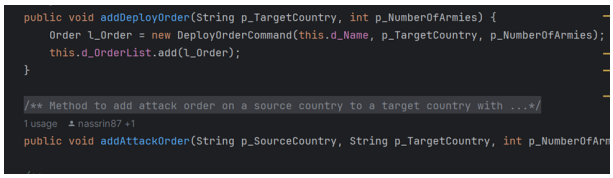


Figure 1.13: addDeployOrder Method in Player Class (Invoker)

1.4.1 State Pattern

The `GamePhase` enum includes each phase of the game (like REINFORCEMENT, ISSUE_ORDERS, EXECUTE_ORDERS, etc.) represented as an enum constant, implementing specific behavior associated with that phase.

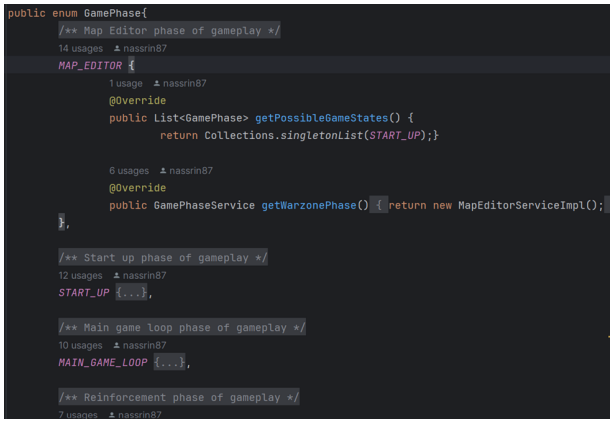


Figure 1.14: State Pattern: GamePhase Enumeration

Each state has a `getWarzonePhase()` method that returns an instance of `GamePhaseService` specific to that phase, demonstrating how behavior changes with the state. For example, REINFORCEMENT returns a new instance of `ReinforcementServiceImpl`, and ISSUE_ORDERS returns a new instance of `OrderIssuanceServiceImpl`. By using the state pattern, every service is validated and then moves to the next state.

1.5 Test Refactoring

The Test Refactoring process focuses on both functional and structural aspects. We've conducted an update of our testing codebase, informed by visual before-and-after comparisons to ensure thoroughness and accuracy.

1.5.1 Test Code Refactoring

The refactoring initiative introduced a new suite of tests and modifications to existing ones to enhance our game's robustness. This initiative captures our strategic improvements:

1. Map validation to ensure both map and continents are connected graphs, reinforcing the integrity of the game's geography.
2. Robust error handling of invalid map files to prevent crashes and ensure graceful failure.
3. Validation of the correct startup phase to confirm that game initialization follows prescribed rules.
4. Accurate calculations for the number of reinforcement armies, which is crucial for gameplay balance.
5. Execution validation for various orders, including source and target validation, country conquest dynamics, and end-game scenarios.

Each implementation class has a corresponding test class, maintaining a 1-1 relationship that fosters clarity and traceability in our codebase.

```
26 -     * Initializes the {@link GameMapDataHandlerImpl} for testing its functionalities related to game map handling.
27         */
28 -    public void testGameMapDataHandlerImpl() {

29             d_gameMapDataHandler = new GameMapDataHandlerImpl();
30         }

31         /**
32 -      * Validates that a game map can be created from a map file input without throwing exceptions.
33 -      * This test simulates the reading of a basic map file structure, asserting that the map creation process is successful.
34         */
35         @Test
36         public void testCreateGameMap() {
```

```
27 +      * Sets up the testing environment before each test. Initializes the {@link GameMapDataHandlerImpl} instance.
28         */
29 +     @BeforeEach
30 +     public void setUp() {
31             d_gameMapDataHandler = new GameMapDataHandlerImpl();
32         }
33 +
34         /**
35 +      * Tests the map creation functionality from a map file input. Ensures that no exceptions are thrown
36 +      * during the creation process when given a basic map file structure.
37         */
38         @Test
39         public void testCreateGameMap() {
```

Figure 1.15: Before and After: Setup Method Refactoring

1.5.2 Test Suites and Structure

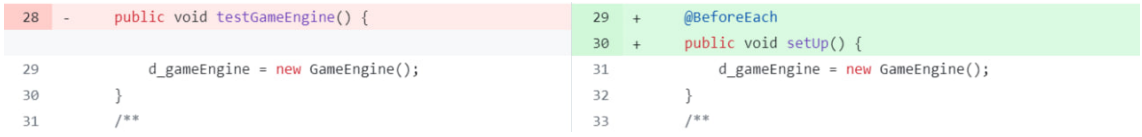
Central to our refactoring is a master test suite that encompasses all individual test cases. This suite, along with package-specific suites, allows for both granular and comprehensive testing approaches.

1.5.3 Test Java doc Refactoring

Our JavaDocs have undergone extensive updates to better reflect the intricacies of the test cases. The before-and-after comparison in Figure 1.17 highlights the enhanced clarity and detail now provided.

```
85                                                  85
86      @Test                                       86      @Test
87      public void testGetCountries() {            87      public void testGetCountries() {
88  -       Country testCountry = new Country(100, "Asia",  88  +       d_continent.getCountries().put("Iran", new
    "Yes");                                             Country("Iran", "Asia", 1L));
89  -       d_Continent.getCountries().put("Iran",      89  +
    testCountry);                                       Assertions.assertTrue(d_continent.getCountries().contain
                                                        sKey("Iran"));
90  -
    Assertions.assertTrue(d_Continent.getCountries().contain
    sKey("Iran"));
91  -       Assertions.assertSame(testCountry,
    d_Continent.getCountries().get("Iran"));
92      }                                           90      }
```

Figure 1.16: Refactoring: Get Countries Test and Associated Documentation



```
28  -   public void testGameEngine() {              29  +   @BeforeEach
                                                    30  +   public void setUp() {
29          d_gameEngine = new GameEngine();        31          d_gameEngine = new GameEngine();
30      }                                           32      }
31      /**                                         33      /**
```

Figure 1.17: Before and after game map testing and JavaDoc enhancements