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.

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.

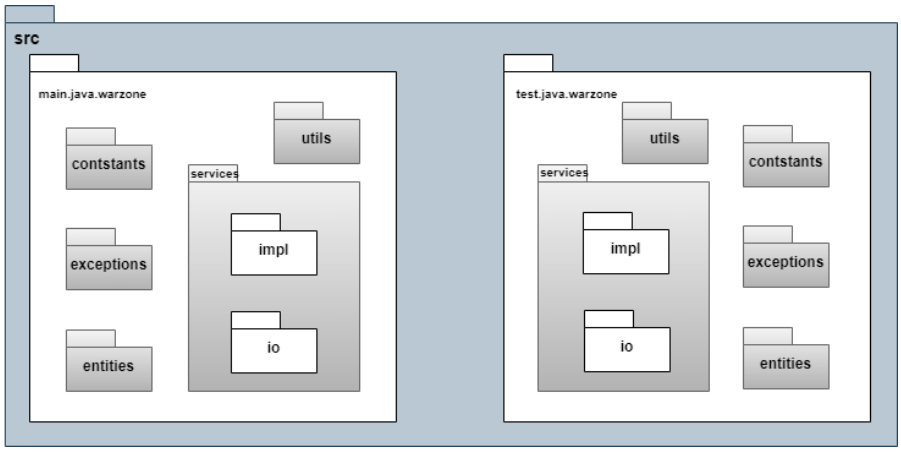

Figure 1.1: Package Diagram before refactoring

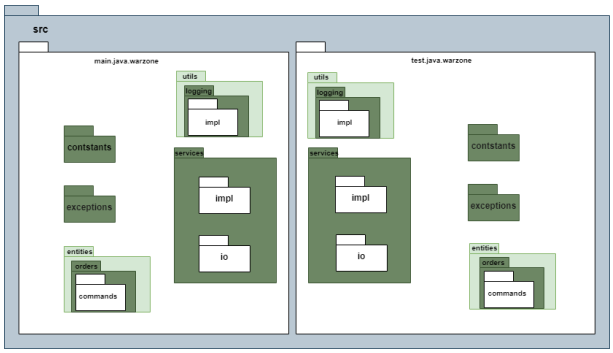

Figure 1.2: Package Diagram after refactoring

1.2.1 entitiy package

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

1.2.2 util package

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

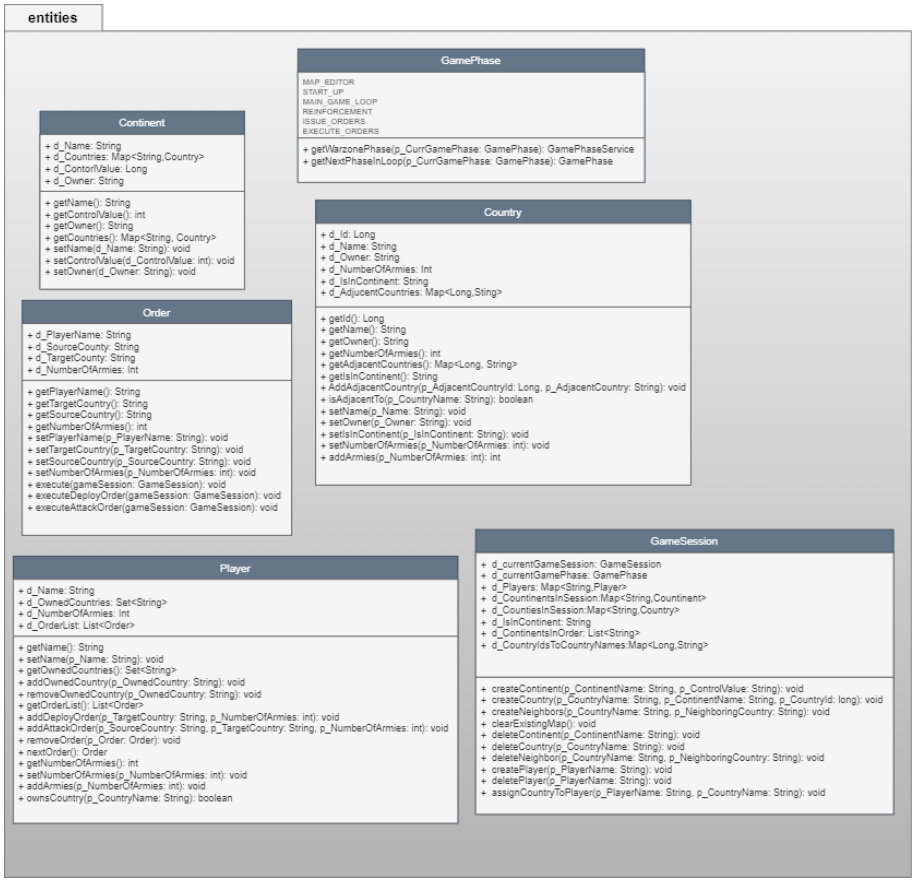

Figure 1.3: Entity Package Diagram before refactoring

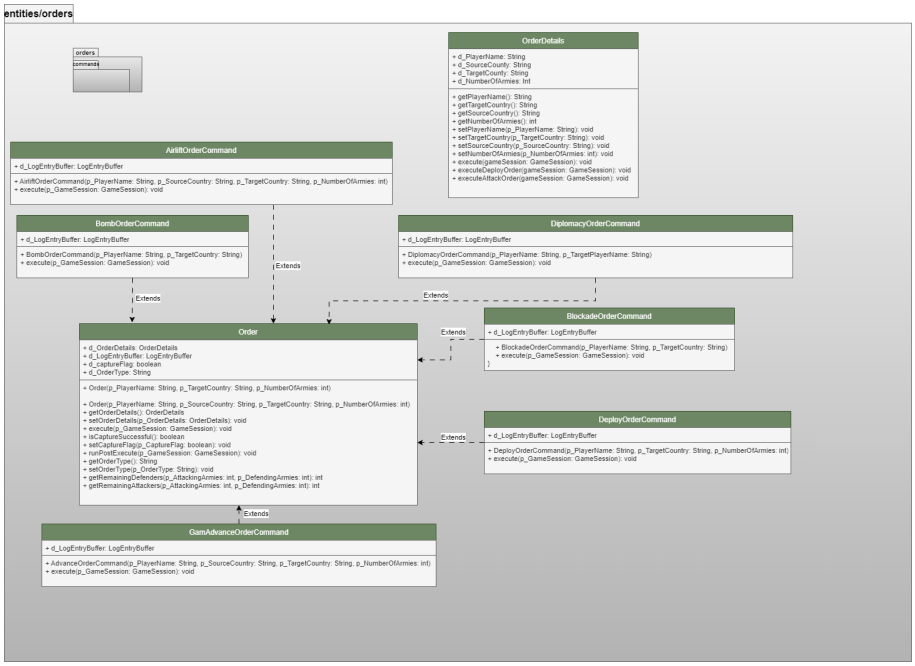

Figure 1.4: Entity Package Diagram after refactoring

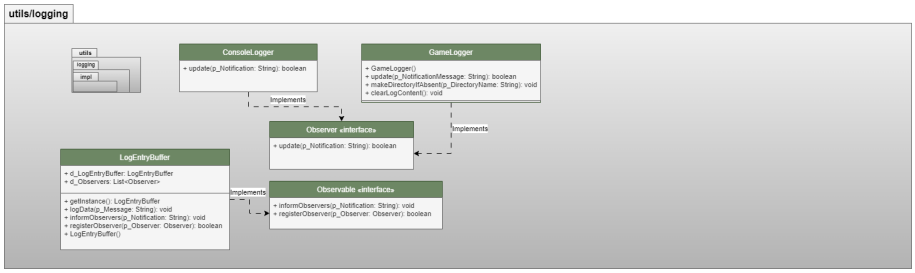

Figure 1.5: Entity Package Diagram after refactoring

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 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 observer interface and act as observers that perform specific actions when they are notified of log messages by the LogEntryBuffer.

```
public class LogEntryBuffer implements Observable {

    /** Singleton log entry buffer object */
    3 usages
    private static LogEntryBuffer d_LogEntryBuffer;

    /** List of observers to be informed about the change */
    2 usages
    private List<Observer> d_Observers = new ArrayList<>();

    /** Private constructor to implement singleton ...*/
    1 usage   nassrin87
    private LogEntryBuffer() {}

    /** Method to get singleton instance of LogEntryBuffer ...*/
    nassrin87
    public static LogEntryBuffer getInstance() {...}

    /** Method to log data and inform the observers ...*/
    nassrin87
    public void logData(String p_Message) { informObservers(p_Message); }

    1 usage   nassrin87
    @Override
    public void informObservers(String p_Notification) {...}
```

Figure 1.6: LogEntryBuffer Class

1.3.2 Command Pattern

The abstarct Order class: acts as the Command interface in the Command Pattern, declaring the execute method that all concrete commands (AdvanceOrderCommand, DeployOrderCommand,

```java
public class ConsoleLogger implements Observer {

    1 usage  Ali Sayed Salehi
    @Override
    public boolean update(String p_Notification) {
        System.out.println(p_Notification);
        return true;
    }
}
```

Figure 1.7: ConsoleLogger Class

```java
public class GameLogger implements Observer {

    1 usage  Jerome Kithinji
    public GameLogger() { clearLogContent(); }

    /** Handles the update notifications by logging them to a file. .
    1 usage  Jerome Kithinji
    @Override
    public boolean update(String p_NotificationMessage) {...}

    /** Creates the log directory if it does not exist. ...*/
    2 usages  Jerome Kithinji
    private static void makeDirectoryIfAbsent(String p_DirectoryName)

    /** Method to clear the existing logs in the log file. */
    1 usage  Jerome Kithinji
    private void clearLogContent() {...}


}
```

Figure 1.8: GameLogger Class

etc.) will implement. The AdvanceOrderCommand, AirliftOrderCommand, BlockadeOrderCommand, BombOrderCommand, DeployOrderCommand, DiplomacyOrderCommand classes are concrete implementations of the Order abstarct class. 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. 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.

1.3.3 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. 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 state pattern, every service is validated and then moves to next state.

```java
public abstract class Order {

    /** Member to hold details of the order */
    12 usages
    private OrderDetails d_OrderDetails;

    /** LogEntryBuffer object to log the information ...*/
    2 usages
    private LogEntryBuffer d_LogEntryBuffer;

    /** Boolean to check if capture was successful */
    2 usages
    private boolean d_captureFlag = false;

    /** Order type */
    2 usages
    private String d_OrderType;
```

Figure 1.9: Order Class(Command Interface)

```java
public class AdvanceOrderCommand extends Order {


    /** LogEntryBuffer instance for recording log data. */
    5 usages
    private LogEntryBuffer d_LogEntryBuffer;
```

Figure 1.10: AdvanceOrderCommand Class

1.4 Changing Data Structure

Using doubly ended queues for inserting and retrieving the Orders instead of a list which helps to improve the time complexity of the order execution.

1.5 Test Refactoring

The refactoring process, while often focused on production code, also extends to test code, which is equally important for maintaining the health and robustness of a software project. Test refactoring aimed at enhancing the clarity, efficiency, and maintainability of our test suite, ensuring that it continues to effectively validate our application's functionality through its evolution. This section outlines the approaches we took to refactor our test code and documentation.

1.5.1 Test Code Refactoring

During the refactoring process, significant enhancements were made to the test suite to improve its quality, maintainability, and effectiveness. Test cases were reorganized to better mirror the production code structure, facilitating easier navigation and comprehension. A clearer and more descriptive naming convention for test cases was adopted, focusing on the scenario being tested and the expected outcome, which aids in understanding the purpose of each test at a glance.

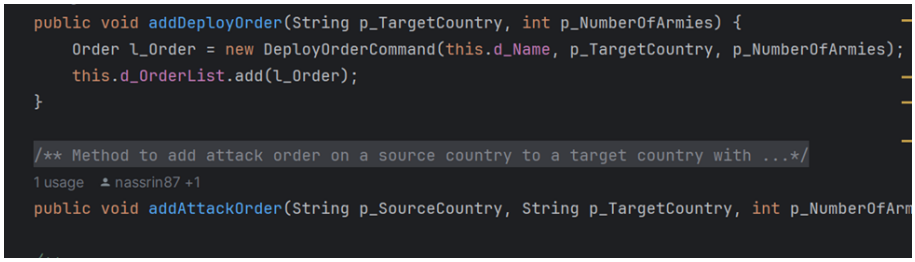

Figure 1.11: addDeployOrder Method in Player Class (invoker)

Figure 1.12: GamePhase Class before refactoring and state pattern

Coverage was expanded to include previously untested scenarios, significantly reducing the risk of regressions and ensuring a more robust application. Assertions within test cases were refined for greater precision and relevance, employing more specific assertion types and conditions to enhance the suite's ability to detect issues and regressions accurately. Additionally, comprehensive JavaDoc comments were added to each test class and method, providing essential documentation to support understanding and maintenance of the test suite.

1.5.2 Test Java doc Refactoring

The JavaDoc refactoring for test code aimed at enhancing clarity, consistency, and completeness across the documentation. JavaDoc formatting was standardized across all test classes and methods to improve readability and navigability. Each test method's JavaDoc now features a clear description of the test's purpose, the scenario it addresses, and the expected outcome, clarifying the intent behind each test. Parameters and return values are thoroughly documented, offering a full view of the test methods' inputs and outputs. For tests with specific preconditions or post-conditions, these aspects are explicitly noted, aiding in the understanding of the test setup and outcomes. Additionally, tests are categorized with annotations (e.g., unit tests, integration tests, performance tests), enhancing the ability to execute and analyze tests selectively. These refinements have significantly bolstered the test suite's readability, maintainability, and effectiveness, contributing to the software's overall quality and reliability.

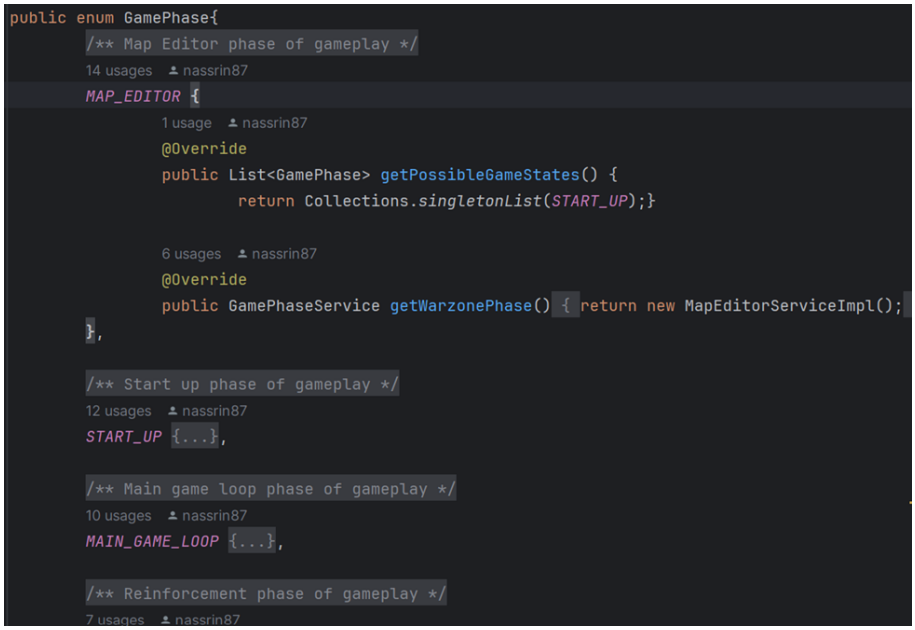

Figure 1.13: GamePhase Class after refactoring

Figure 1.14: using order list before refactoring

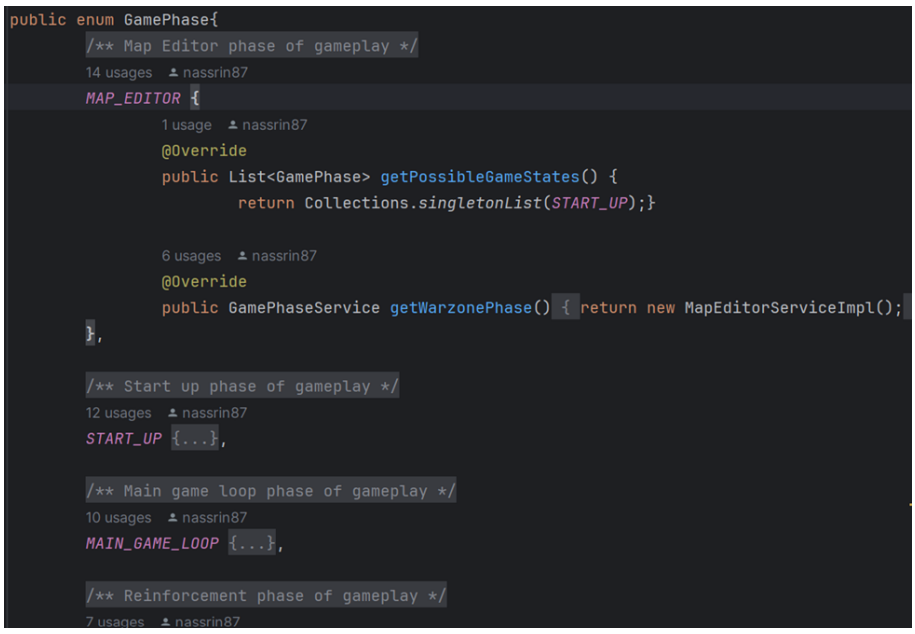

Figure 1.15: using order list before refactoring