SOEN 6441 (Advanced Programing practices)

Department of Computer Science and Software Engineering

Build 02: Refactoring

Team 01:

Niloufar Pilgush
Nasrin Maarefi
Jerome Kithinji
Ali Sayed Salehi
Fateme Chaji

Professor:
Joey Paquet

March 2024

Contents

Build 02/Refactoring

1.1 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.

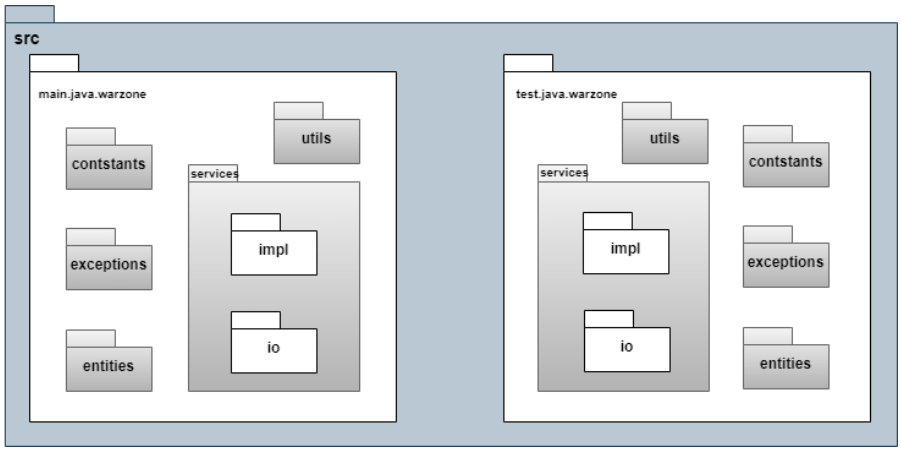


Figure 1.1: Package Diagram before refactoring

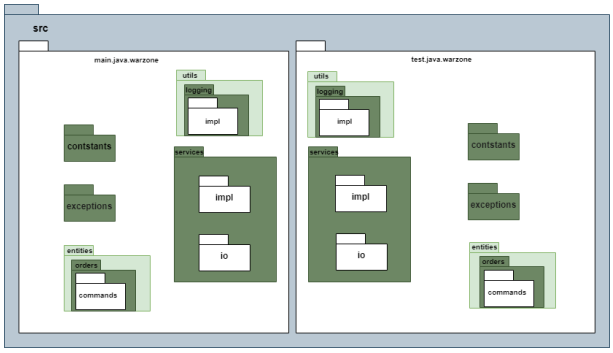


Figure 1.2: Package Diagram after refactoring

1.1.1 entitiy package

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

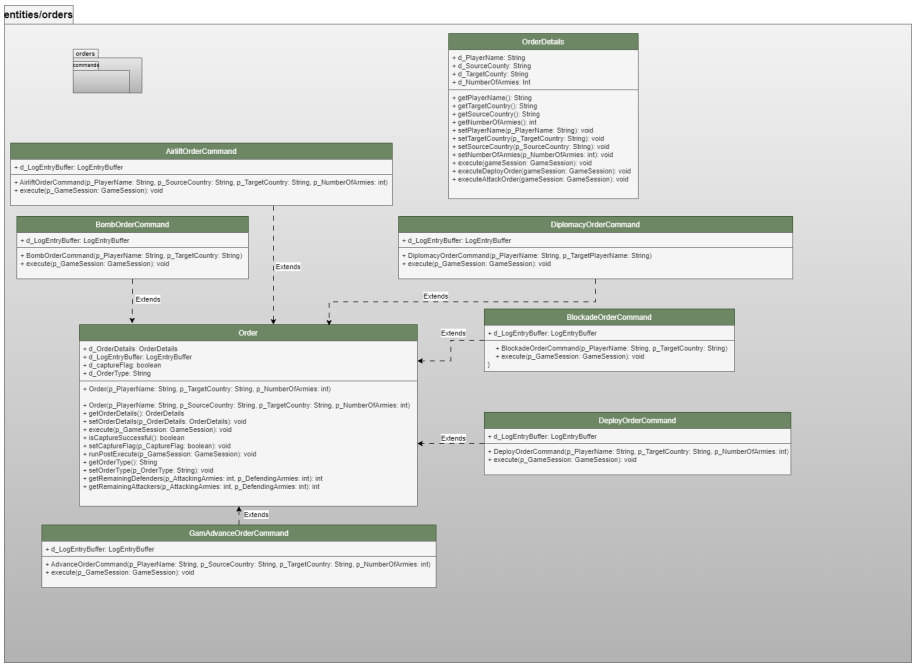


Figure 1.3: Entity Package Diagram after refactoring

1.1.2 util package

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

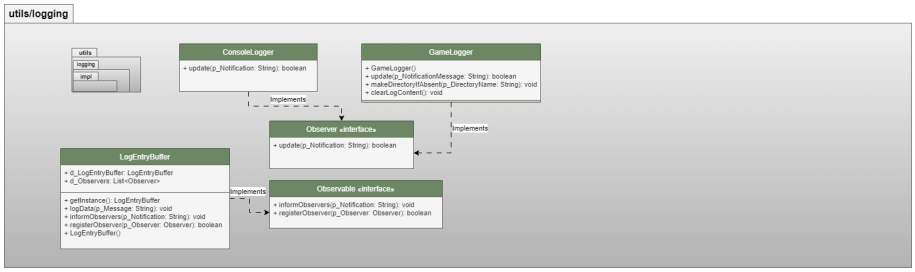


Figure 1.4: Entity Package Diagram after refactoring

1.2 Implementing Pattern Designs

1.2.1 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 1.5: LogEntryBuffer Class

```
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 1.6: ConsoleLogger Class

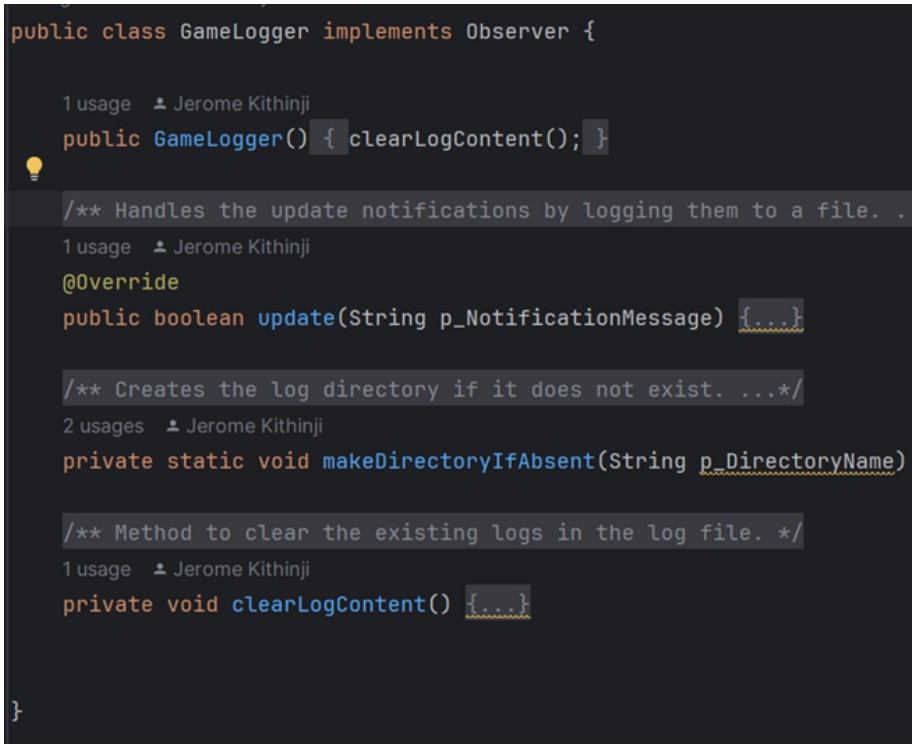


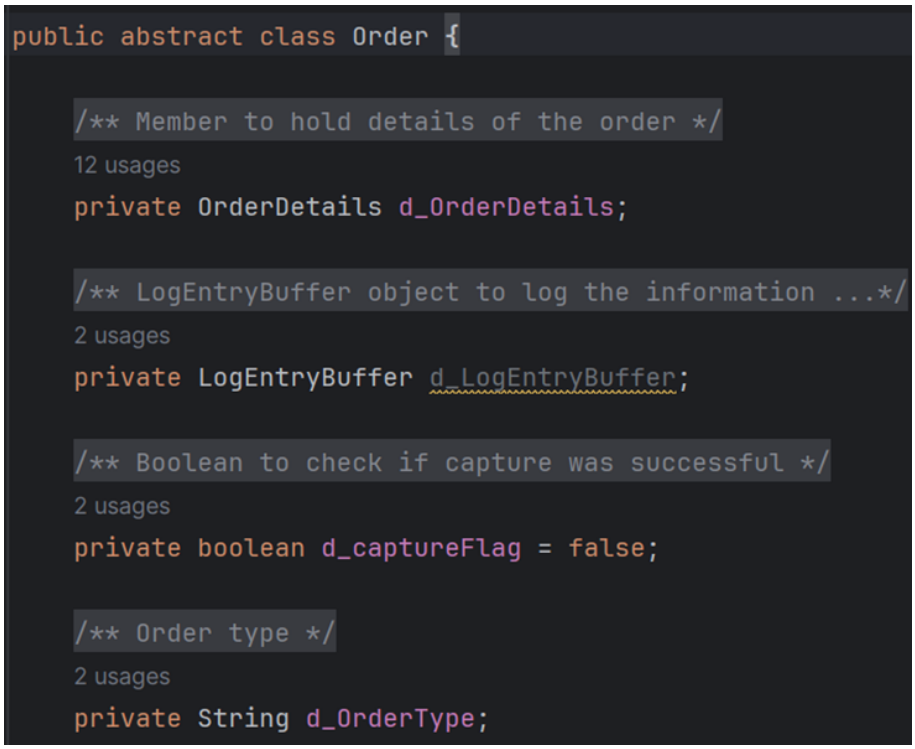
Figure 1.7: GameLogger Class

1.2.2 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, 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.

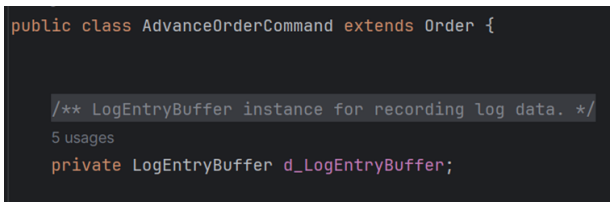
1.2.3 State Pattern

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`.



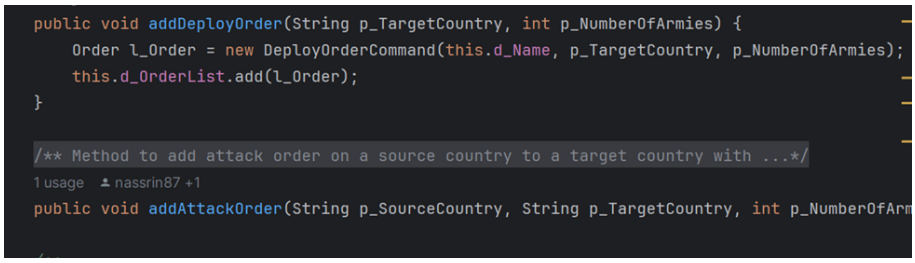
```
public abstract class Order {

    /** 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 1.8: Order Class(Command Interface)



```
public class AdvanceOrderCommand extends Order {


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

Figure 1.9: AdvanceOrderCommand Class



```
public void addDeployOrder(String p_TargetCountry, int p_NumberOfArmies) {
    Order l_Order = new DeployOrderCommand(this.d_Name, p_TargetCountry, p_NumberOfArmies);
    this.d_OrderList.add(l_Order);
}

/** Method to add attack order on a source country to a target country with ...*/
1 usage  nassrin87 +1
public void addAttackOrder(String p_SourceCountry, String p_TargetCountry, int p_NumberOfArm
```

Figure 1.10: addDeployOrder Method in Player Class (invoker)

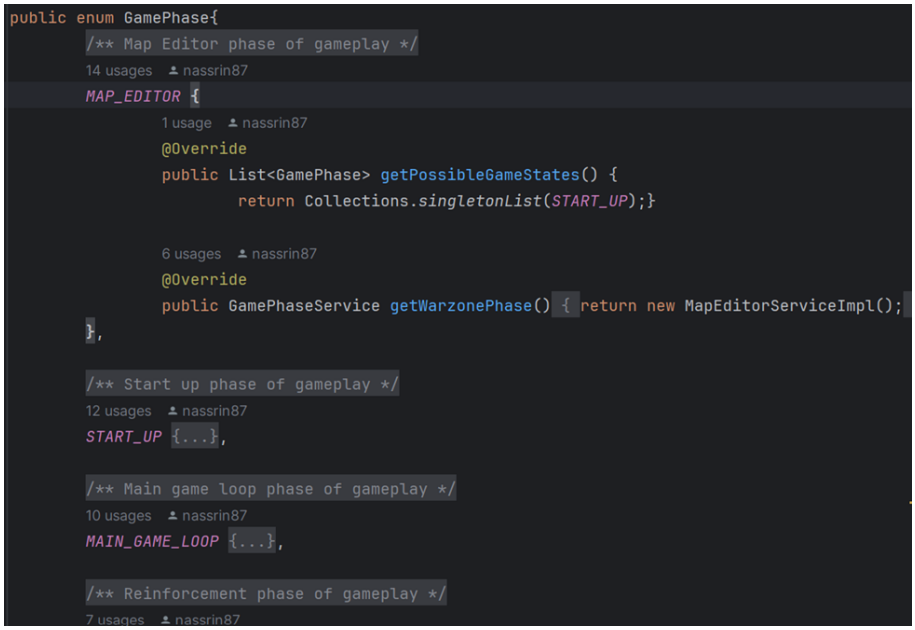


Figure 1.11: GamePhase Class