SOEN 6441 (Advanced Programing practices)

Department of Computer Science and Software Engineering

Build 03: Refactoring

Professor:
Joey Paquet

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

April 2024

Contents

1 Build 03/Architectural Design **2**
1.1 Architectural Design Diagrams 2

2 Build 03/Refactoring **5**
2.1 Refactoring Process 5
2.1.1 Potential Refactoring Targets: 5
2.1.2 Refactoring Operations 5
2.2 Changing Package Structure 5
2.3 Implementing Pattern Designs 6
2.3.1 Adapter Pattern 6
2.3.2 Strategy Pattern 7
2.4 Other Refactrings 8
2.4.1 Refactoring to save the state of the game and loading the saved game . . . 8
2.4.2 Refactoring to include tournament phase in the game 9
2.4.3 Refactoring to display order execution information as a separate abstract method 9
2.4.4 Refactoring to clear existing logs before running the game from starting . . 10

Build 03/Architectural Design

1.1 Architectural Design Diagrams

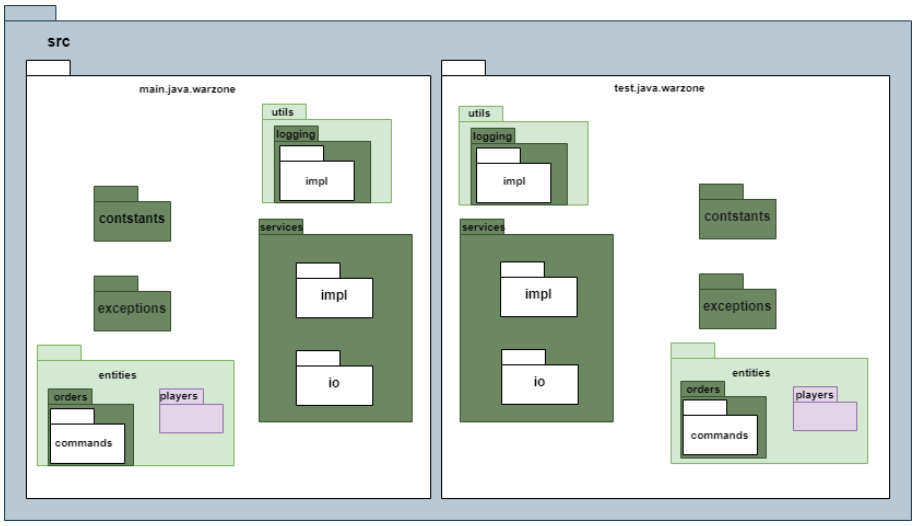

Figure 1.1: Package Diagram-Build03

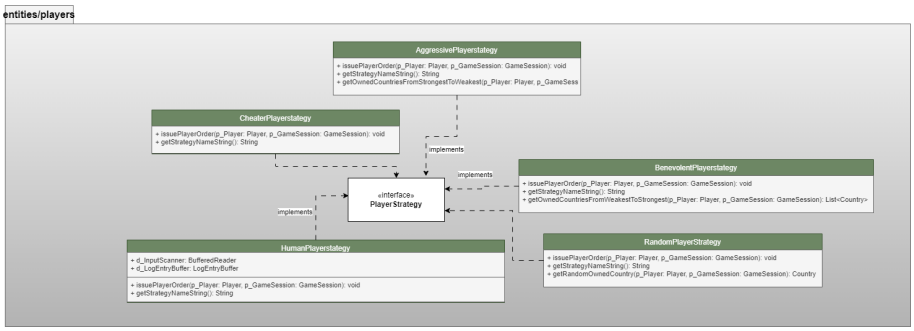

Figure 1.2: Entities Package Diagram/players

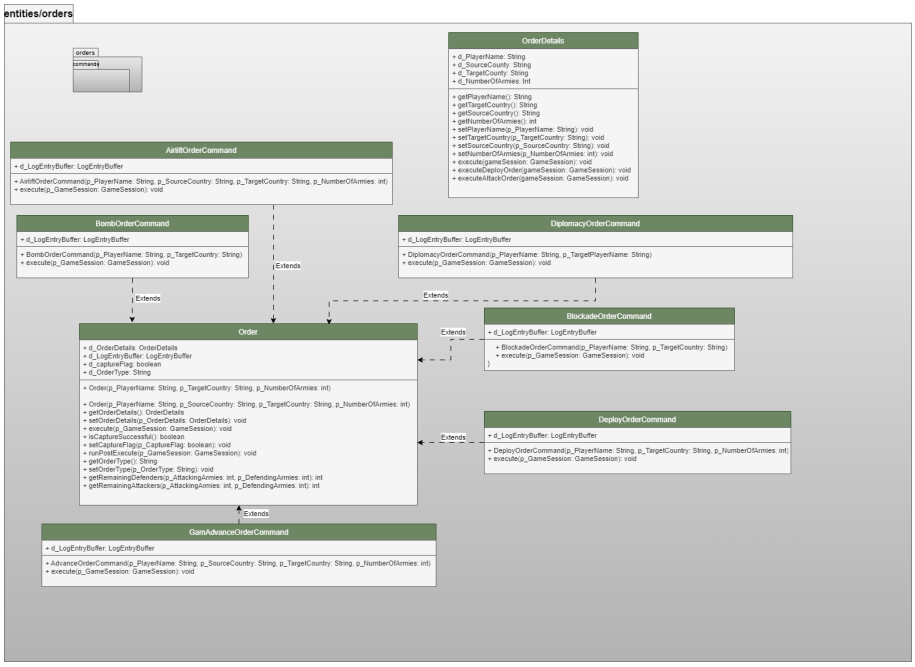

Figure 1.3: Entities Package Diagram/orders

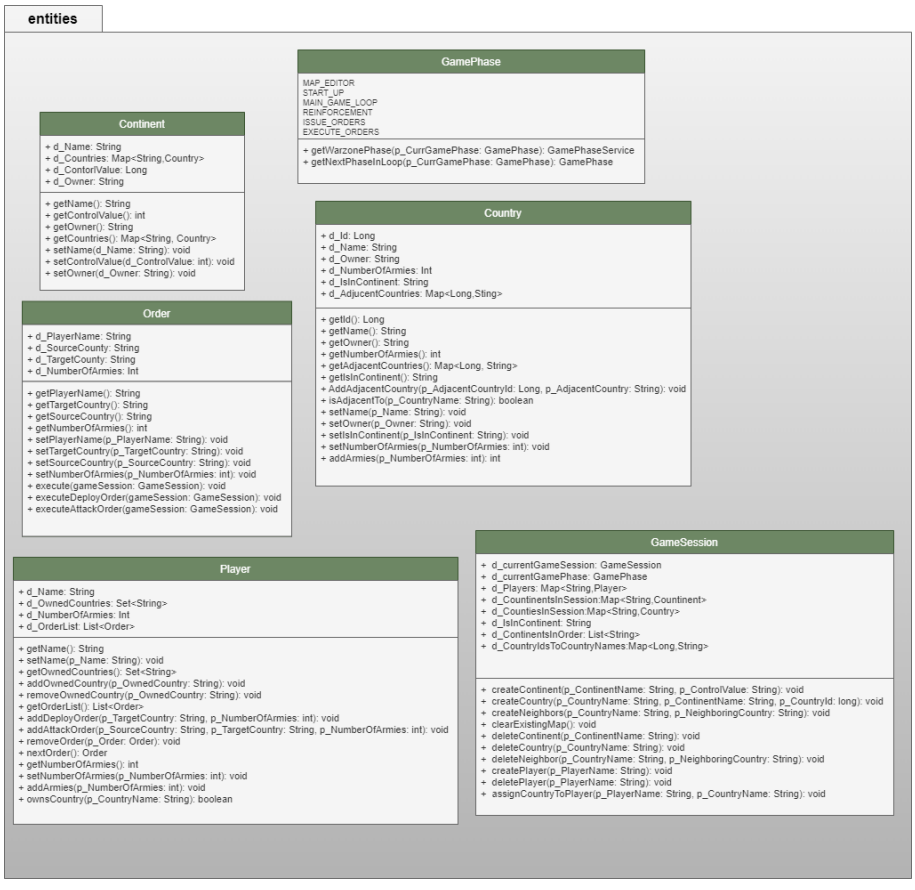

Figure 1.4: Entities Package Diagram/base entities

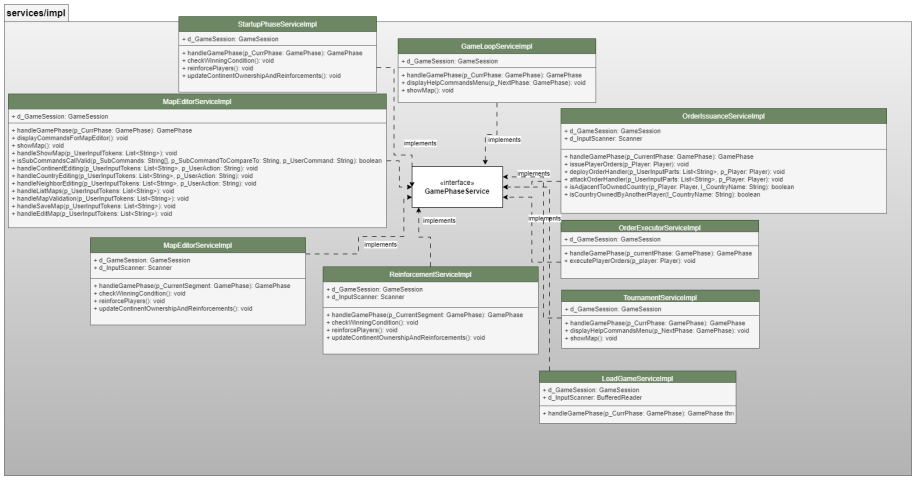

Figure 1.5: Services Package Diagram/impl

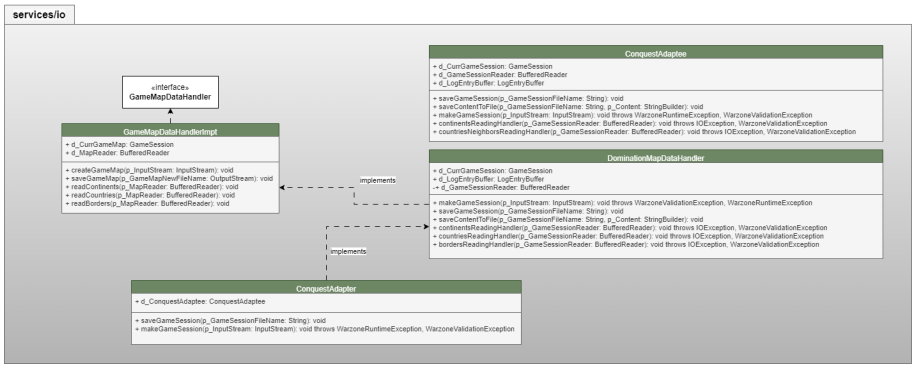

Figure 1.6: Services Package Diagram/io

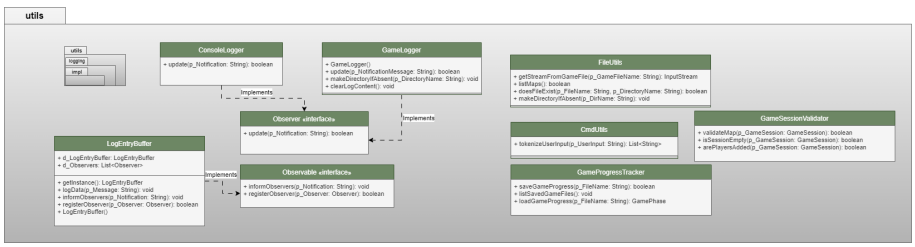

Figure 1.7: utils Package Diagram

Build 03/Refactoring

2.1 Refactoring Process

2.1.1 Potential Refactoring Targets:

Following mentioned are some of the refactoring targets observed while implementation of build1, build2 and enhancements that are required for build3:

1. Refactoring to include Conquest game map format by using Adapter Pattern.
2. Refactoring to include different player behaviours using Strategy pattern.
3. Refactoring to save the state of the game and loading the saved game.
4. Refactoring to include single and tournament mode in the game.
5. Refactoring to move methods to utils for code reusability.
6. Refactoring to display order execution information as a separate abstract method.
7. Refactoring to clear existing logs before running the game from starting.
8. Refactoring to make hierarchical directories if doesn't exist.
9. Refactoring to handle more scenarios for game map extensions handling.
10. Refactoring to use regular expressions for commands splitting.
11. Refactoring to modify access modifiers to increase encapsulation.
12. Refactoring to include error handling using observer pattern.
13. Improve console outputs for more user friendly error messages.
14. Implement additional test cases.
15. Improve JavaDocs for better understanding of methods.

2.1.2 Refactoring Operations

Following are the actual refactoring operations we have done through build 03:

2.2 Changing Package Structure

Reorganizing the package structure, particularly within the "entities" package, to introduce a new subpackage named "players". This reorganization aimed to cleanly implement the strategy pattern, allowing for a more structured approach to handling various game strategies such as aggressive, cheating, and random strategies by players.

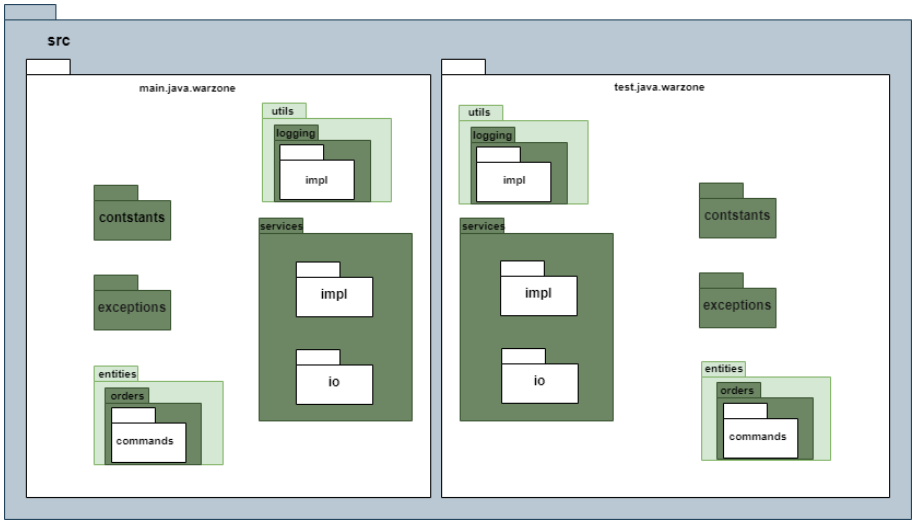

Figure 2.1: Package Diagram-Build02

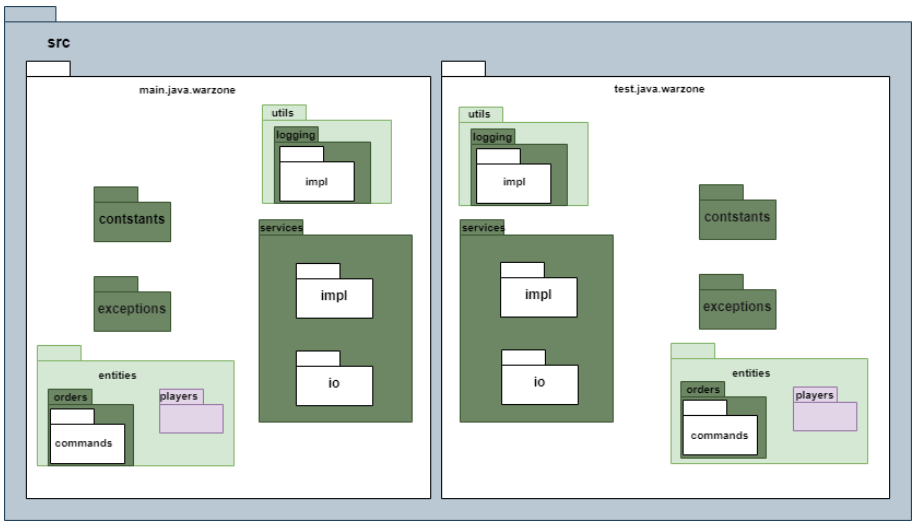

Figure 2.2: Package Diagram-Build03

2.3 Implementing Pattern Designs

2.3.1 Adapter Pattern

By utilizing the Adapter pattern, the system gains flexibility and extensibility, as it can now accommodate different map formats without the need for extensive modifications to existing code. It facilitates the integration of the Conquest game map format with existing systems designed for other map formats. This pattern involves an Adapter class, ConquestAdapter, which acts as an intermediary, allowing the DominationMapDataHandlerImpl class to interact with the ConquestAdaptee. In this scenario, the ConquestAdaptee class serves as the Adaptee, providing implementations for reading and writing data in the Conquest format. It handles tasks such as saving game sessions and loading map data. The ConquestAdapter class, on the other hand, implements the same interface as the existing DominationMapDataHandlerImpl class, enabling seamless interaction between the two components.

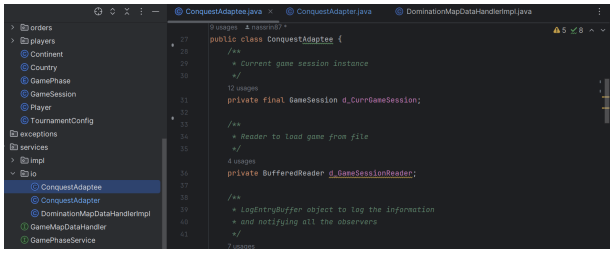

Figure 2.3: Conquest Adaptee Class

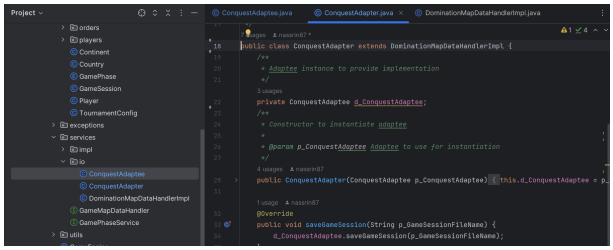

Figure 2.4: Conquest Adapter Class

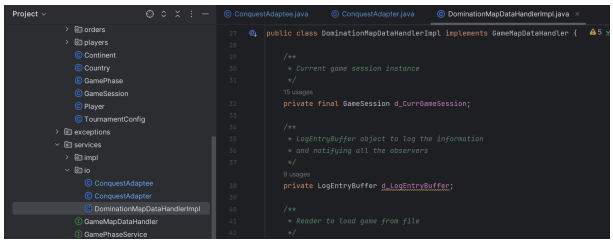

Figure 2.5: DominationMapDataHandlerImpl Class

2.3.2 Strategy Pattern

The Strategy Pattern is a behavioral design pattern that enables selecting an algorithm at runtime. It allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. The following classes encapsulate different strategies for issuing player orders in a game session, allowing for flexibility and interchangeability of behaviors at runtime.

AggressivePlayerStrategy: This class represents the implementation of an aggressive player strategy. The `issuePlayerOrder` method prioritizes centralizing forces before attacking. It deploys reinforcements to the strongest country, launches attacks from this point of strength, and rearranges its armies to consolidate power in a single location.

BenevolentPlayerStrategy: This class represents the implementation of a benevolent player strategy. The `issuePlayerOrder` method focuses on protecting weak countries by deploying armies to the weakest country and never attacking. It then moves its armies to reinforce these weaker countries.

CheaterPlayerStrategy: This class represents the implementation of a cheater player strategy. The `issuePlayerOrder` method conquers all immediate neighboring enemy countries and then doubles the number of armies on its countries that have enemy neighbors. It directly affects the map during the order creation phase to achieve this behavior.

HumanPlayerStrategy: This class represents the implementation of a human player strategy. The `issuePlayerOrder` method allows the human player to issue various types of orders, such as deploying armies, advancing, bombing, blockading, airlifting, and negotiating, based on user input through the console.

RandomPlayerStrategy: This class represents the implementation of a random player strategy. The `issuePlayerOrder` method selects a random owned country, deploys armies to it, and launches an attack from it to a random adjacent enemy country.

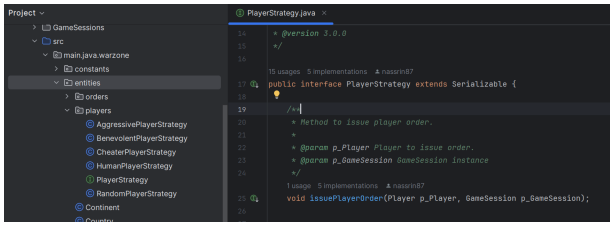

Figure 2.6: Player Strategy Interface

2.4 Other Refactrings

2.4.1 Refactoring to save the state of the game and loading the saved game

These operations are handled by two methods in the new class added named "GameProcessTracker" in "utils" package.

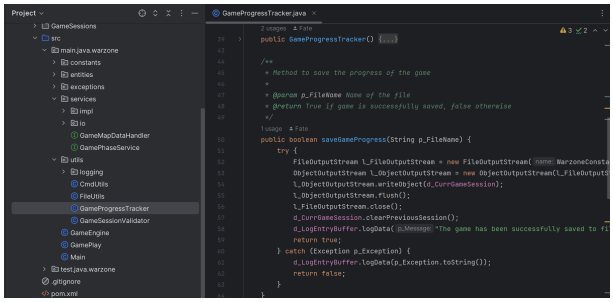

Figure 2.7: Save Game Progress

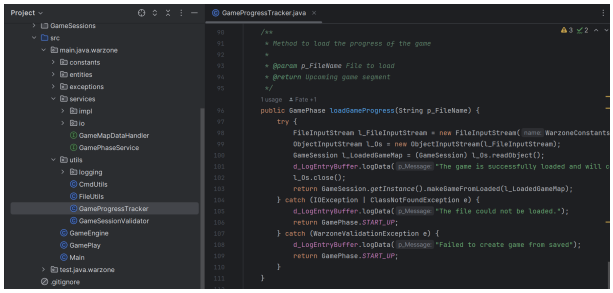

Figure 2.8: Load Game Progress

2.4.2 Refactoring to include tournament phase in the game

To address tournament phase of the game, new item as "TOURNAMENT" added to GamePhase class.

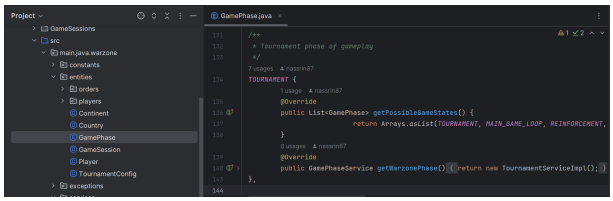

Figure 2.9: Tournament: new phase

2.4.3 Refactoring to display order execution information as a separate abstract method

A new method named "displayCommand" added to Order class. The purpose of the displayCommand method in the AdvanceOrderCommand class is to log or display information about the execution of the advance order. This method is overriden from within the execute method of the Command classes to provide a record of the command execution.

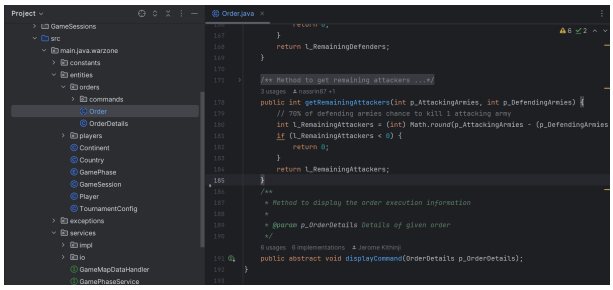

Figure 2.10: displayCommand method

Figure 2.11: Command Information

2.4.4 Refactoring to clear existing logs before running the game from starting

A new method named "makeDirectoryIfAbsent" added to GameLogger class to guarantee that the directory structure required for logging exists before attempting to write log files. It ensures that log files have a designated location to be stored and prevents potential errors that could occur if the directory doesn't exist during logging operations.

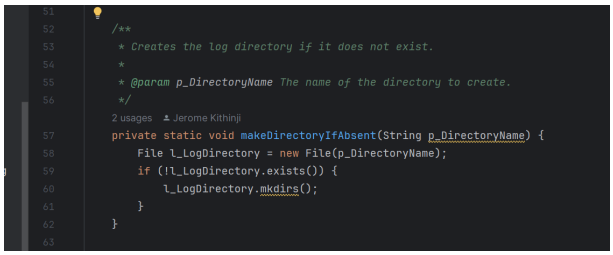

Figure 2.12: makeDirectoryIfAbsent method