Introduction

This document outlines potential refactoring operations to improve the design and maintainability of the application. The refactoring focuses on two key areas:

1. **Refactoring code management from a simple project to a Maven project**.
2. **Refactoring the code to use the State Pattern** to manage the phases of the application.
3. **Refactoring to use the Command Pattern** to implement orders.

For each refactoring operation, the document identifies the targets, explains the rationale, provides a before/after depiction, and lists the tests that apply to the classes involved.

---

1. Refactoring Code Management from Simple Project to Maven Project

Targets Identified

The application is currently managed as a simple project with manual dependency management and no standardized build process. Migrating to a Maven project will improve dependency management, build automation, and project structure.

Refactoring Operations

Target 1: Migrate to Maven

- **Before:** The project has a flat directory structure, and dependencies are manually downloaded and added to the classpath.
- **After:** Restructure the project to follow Maven conventions, including a `pom.xml` file for dependency management and build configuration.
- **Steps:**
    - Create a `pom.xml` file to define project metadata, dependencies, and build plugins.
    - Restructure the project to follow Maven's standard directory layout

    - Move existing dependencies to the `pom.xml` file.
    - Configure Maven plugins for compiling, testing, and packaging the application.
- **Tests:**
    - Ensure the project builds successfully using `mvn clean install`.
    - Verify that all unit tests pass using `mvn test`.
    - Confirm that the application runs correctly after packaging.
- **Why Necessary:** Improves dependency management, standardizes the build process, and makes the project easier to maintain and share.

Before/After Depiction

Before:

```
project/
  lib/              # Manually added JAR files
  src/              # Mixed source and test code
```

After:

```
project/
  pom.xml           # Maven build configuration
  src/
    main/
      java/         # Application code
      resources/    # Configuration files
    test/
      java/         # Test code
      resources/    # Test resources
```

2. Refactoring to Use the State Pattern for Application Phases

Targets Identified

The application has four distinct phases:

1. **Map Editor Phase**: Allows users to create and edit maps.

2. **Game Startup Phase**: Initializes the game, including map loading, player setup and assignment of countries.
3. **Order Creation Phase**: Players create orders (e.g., deploy, advance, cards).
4. **Order Execution Phase**: Executes all player orders in a turn-based manner.

These phases are currently managed using conditional logic (e.g., `if-else` statements), which makes the code rigid and difficult to extend. The State Pattern is ideal for encapsulating the behavior of each phase into separate classes, making the code more modular and easier to maintain.

Refactoring Operations

Target 2: Map Editor Phase

- **Before**: The map editor logic is embedded in a monolithic class with conditional checks for the current phase.
- **After**: Create a `MapEditPhase (Phase)` class that encapsulates all map editor behavior. The context class (e.g., `GameEngine`) delegates phase-specific behavior to this state.
- **Tests**:
    - Test that the `MapEditorPhase` correctly initializes the map editor.
    - Test that transition to other phases (e.g., `StartupPhase`) are handled correctly.
- **Why Necessary**: Reduces complexity by isolating map editor logic and makes it easier to add new features or modify existing ones.

Target 3: Game Startup Phase

- **Before**: Game startup logic is mixed with other phase logic in a single class.
- **After**: Create a `StartupState (Phase)` class to handle player setup, map loading, and validation.
- **Tests**:
    - Test that the `StartupPhase` correctly initializes the game.
    - Test that transition to the `OrderCreationPhase` occur after successful startup.
- **Why Necessary**: Separates concerns and ensures the game startup logic is reusable and testable.

Target 4: Order Creation Phase

- **Before**: Order creation logic is tightly coupled with the main game loop.
- **After**: Create an `OrderCreationPhase (Phase)` class to handle order creation and validation.
- **Tests**:
    - Test that the `OrderCreationPhase` correctly processes player commands.

    - Test that transitions to the `OrderExecutionPhase` occur after all orders are created.
- **Why Necessary**: Encapsulates order creation logic, making it easier to modify or extend.

Target 5: Order Execution Phase

- **Before**: Order execution logic is intertwined with other phase logic.
- **After**: Create an `OrderExecutionPhase (Phase)` class to handle the execution of all orders in a turn-based manner.
- **Tests**:
    - Test that the `OrderExecutionPhase` correctly executes orders.
    - Test that transition back to the `OrderCreationPhase` occur after order execution.
- **Why Necessary**: Isolates order execution logic, improving readability and maintainability.

Before/After Depiction

Before:

```
class GameEngine {
   private String currentPhase;

   void run() {
      if (currentPhase.equals("MapEditor")) {
         //MapEditor mapEditor =  New MapEditor()
         //Editor logic
      } else if (currentPhase.equals("Startup")) {
         //Startup startup = new Startup()
         //Startup logic
      } // ... and so on
   }
}
```

After:

```
interface Phase {
   void setPhase(GameEngine engine);
}

class MapEditor implements Phase {
   void handle(GameEngine engine) {
      // Map editor logic
      engine.setPhase(new StartupPhase());
   }
```

```
}

class GameEngine {
   private Phase currentPhase;

   void setState(Phase state) {
      this.currentPhase = state;
   }

   void run() {
      currentPhase.handle(this);
   }
}
```

3. Refactoring to Use the Command Pattern for Orders

Targets Identified

The application currently implements orders (e.g., deploy, advance, cards) using procedural code, which makes it difficult to add new order types or modify existing ones. The Command Pattern is ideal for encapsulating each order as an object, enabling features like undo/redo and queuing orders.

Refactoring Operations

Target 6: Order Implementation

- **Before**: Orders are implemented as methods in a single class, with conditional logic to determine the type of order.
- **After**: Create an `Order` interface with an `execute()` method. Implement concrete classes for each order type (e.g., `DeployOrder`, `AdvanceOrder`, `CardOrder`).
- **Tests**:
    - Test that each concrete order class executes correctly.
    - Test that orders can be queued and executed in sequence.
- **Why Necessary**: Encapsulates each order as an object, making the code more flexible and extensible.

Before/After Depiction

Before

```
class OrderExecution {
    void processOrder(String orderType) {
        if (orderType.equals("Deploy")) {
            // deploy logic
        }
    }
}
```

After

```
interface Order {
    void execute();
    void validate();
    void print();
    void preValidate();
}

class DeployOrder implements Order {
    void execute() {
        // Move logic
    } // ... and so on
}

class OrderExecution {
    private Queue<Order> orderQueue;

    void processOrders() {
        for (Order order : orderQueue) {
            order.execute();
        }
    }
}

interface Order {
    void execute();
}

class MoveOrder implements Order {
    void execute() {
        // Move logic
    }
}
```